

Chapter -15

The 80286,80386 and 80486 Microprocessors

Prepared By: Prof. Sunil K. Vithlani
Assistant Professor,
Department of Information Technology,
DDU, Nadiad

Basic Terminology of OS

- Time Slice Scheduling: In a full-fledged multitasking or multiuser operating system, part of the operating system determines when it is time to switch from one task to another is called, scheduler, dispatcher or supervisor.
- Preemptive Priority based scheduling: In the system which uses preemptive priority based scheduling, an executing lower priority task can be interrupted by a higher priority task.
- It is suitable to some control applications because it allows the most important task to be done first.
- Preserving the environment: The registers data, pointers, etc., used by an executing task are referred as its environment, state or context.
- Accessing Resources: Another problem encountered in a multitasking system is assuring that task have orderly access to resources such as printer disk drives, etc.

The Need of Protection:

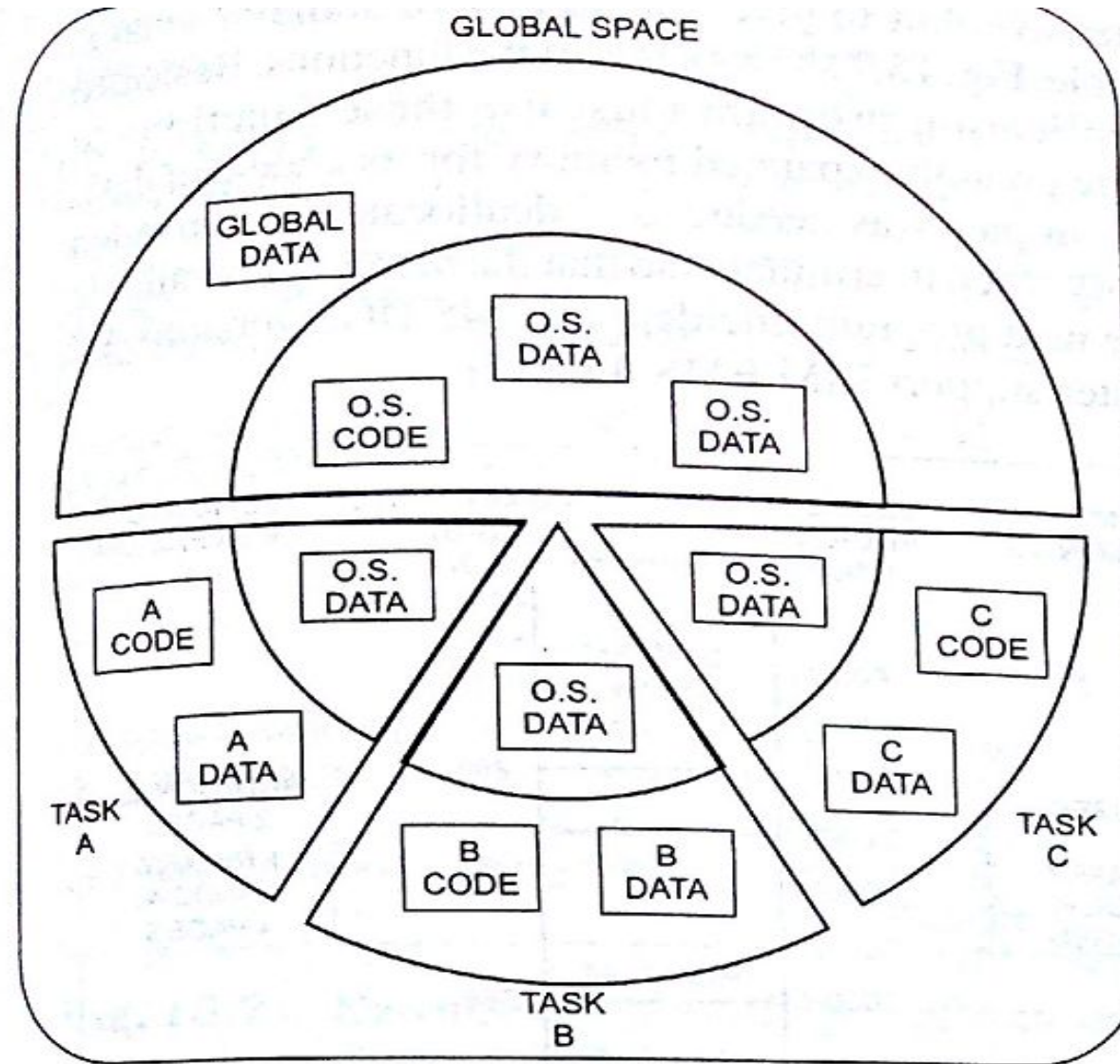
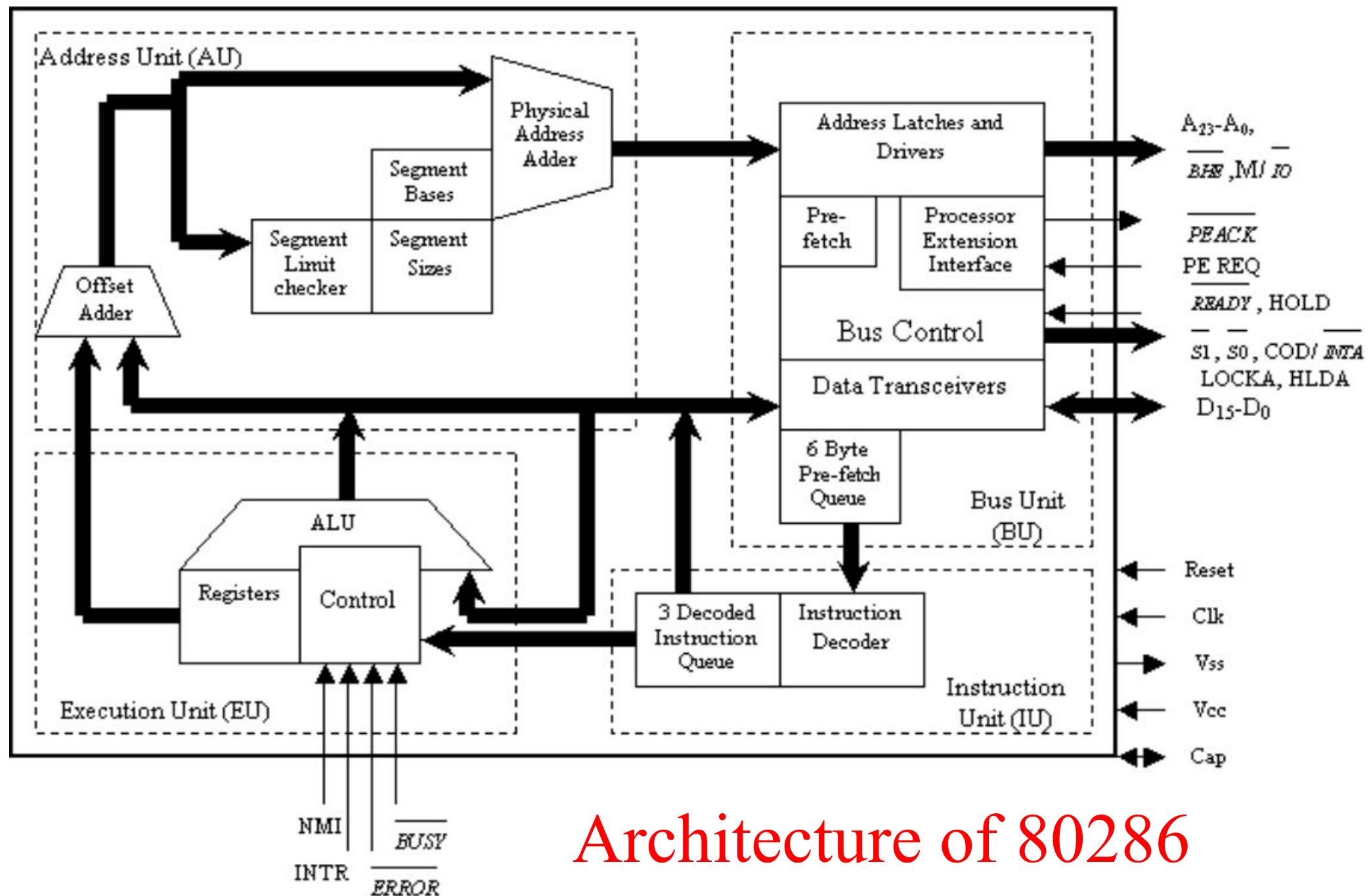


Fig. 15.4 "Onionskin" diagram showing two-level-protection scheme for multitasking operating system. (Intel Corporation)

Intel family of microprocessor, bus and memory sizes

| Microprocessor | Data bus width | Address bus width | Memory size |
|--------------------|----------------|-------------------|-------------|
| 8086 | 16 | 20 | 1MB |
| 80186 | 16 | 20 | 1MB |
| 80286 | 16 | 24 | 16MB |
| 80386 | 32 | 32 | 4GB |
| 80486 | 32 | 32 | 4GB |
| Pentium 4 & core 2 | 64 | 40 | 1TB |



Architecture of 80286

80286 Architecture

- 80286 Contains 4 processing units

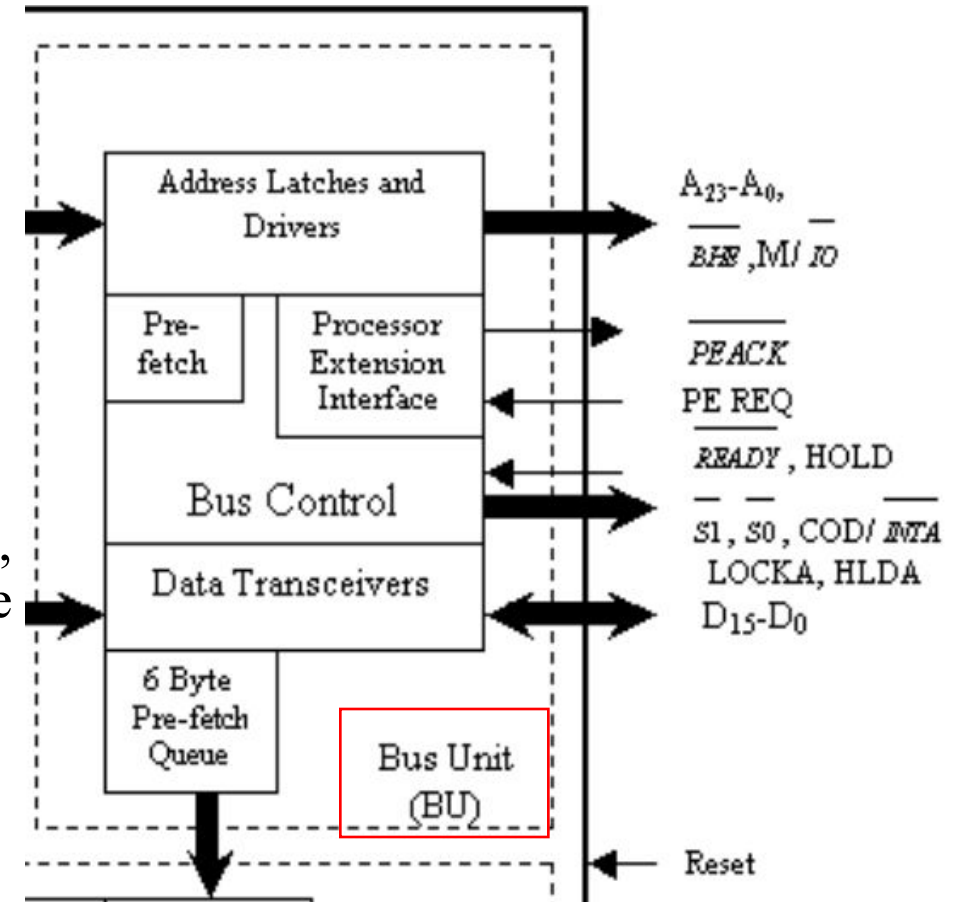
1. Bus Unit (BU)
2. Instruction Unit (IU)
3. Execution Unit (EU)
4. Memory Management Unit (Address Unit)

1. Bus Unit (BU)

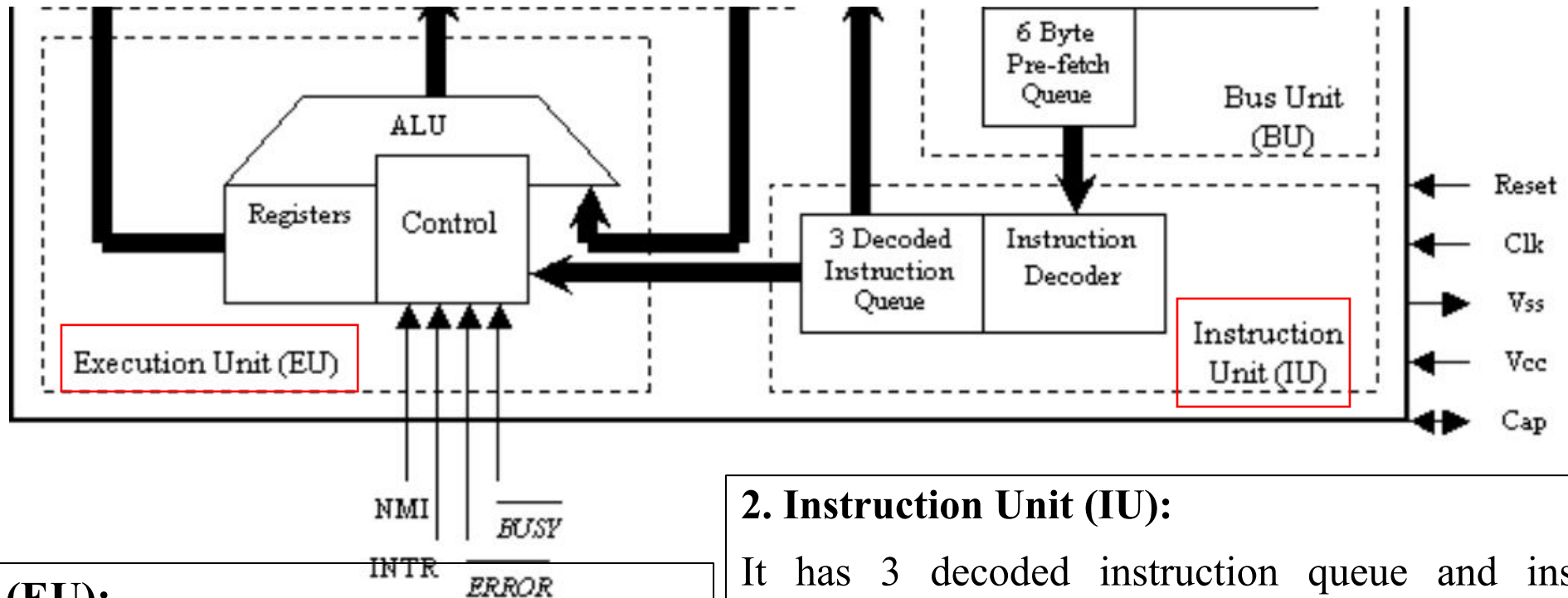
- It has address latches, data transceivers, bus interface and circuitry, instruction pre-fetcher, processor extension interface and 6 byte instruction queue.

Functions :

- To perform all memory and I/O read and write.
- To pre-fetch the instruction bytes.
- To control the transfer of data to and from processor extension devices like 80287 math coprocessor.
- Whenever BU is not using the buses for the operation, it pre-fetches the instruction bytes and put them in a 6 byte pre-fetch queue.



80286 Architecture



3. Execution Unit (EU):

It includes ALU, registers and the Control unit. Registers are general purpose, index, pointer, flag register and **16-bit Machine Status Word (MSW)**.

Functions :

- To sequentially execute the instructions received from the instruction unit.
- ALU result is either stored in register bank or sent over the data bus.

2. Instruction Unit (IU):

It has 3 decoded instruction queue and instruction decoder.

• Functions :

- It fully decodes up to three prefetched instructions and holds them in a queue.
- So that EU can access them.
- It helps the processor to speed up, as pipelining of instruction is done.

80286 Architecture

4. Memory management Unit (MMU) or Address Unit (AU):

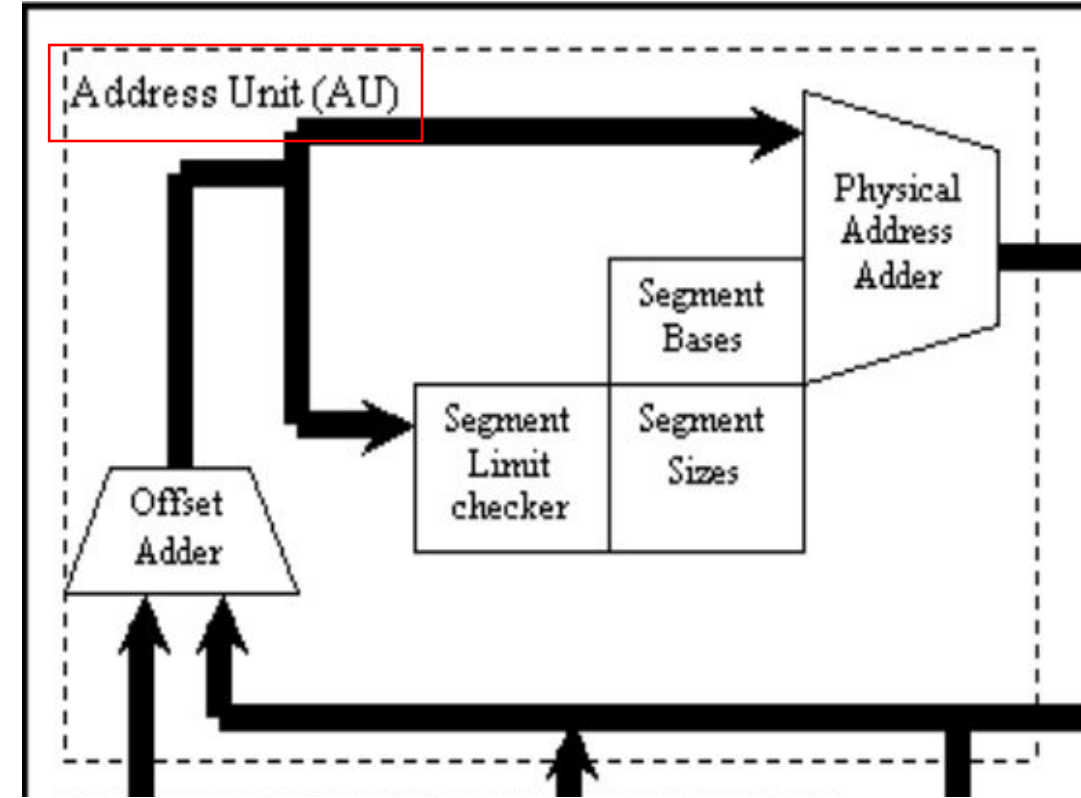
It consists of segment registers, offset address and a physical address adder.

Functions :

- Compute the physical address that will be sent out to the memory or I/O by BU.
- 80286 operate in two different modes

1. Real address mode:

- MMU computes the address with segment base and offset like 8086.
- Segment register are CS, DS, ES and SS hold base address. IP, BP, SI, DI , SP hold offset.
- Maximum physical space allowed in this mode is 1MB.

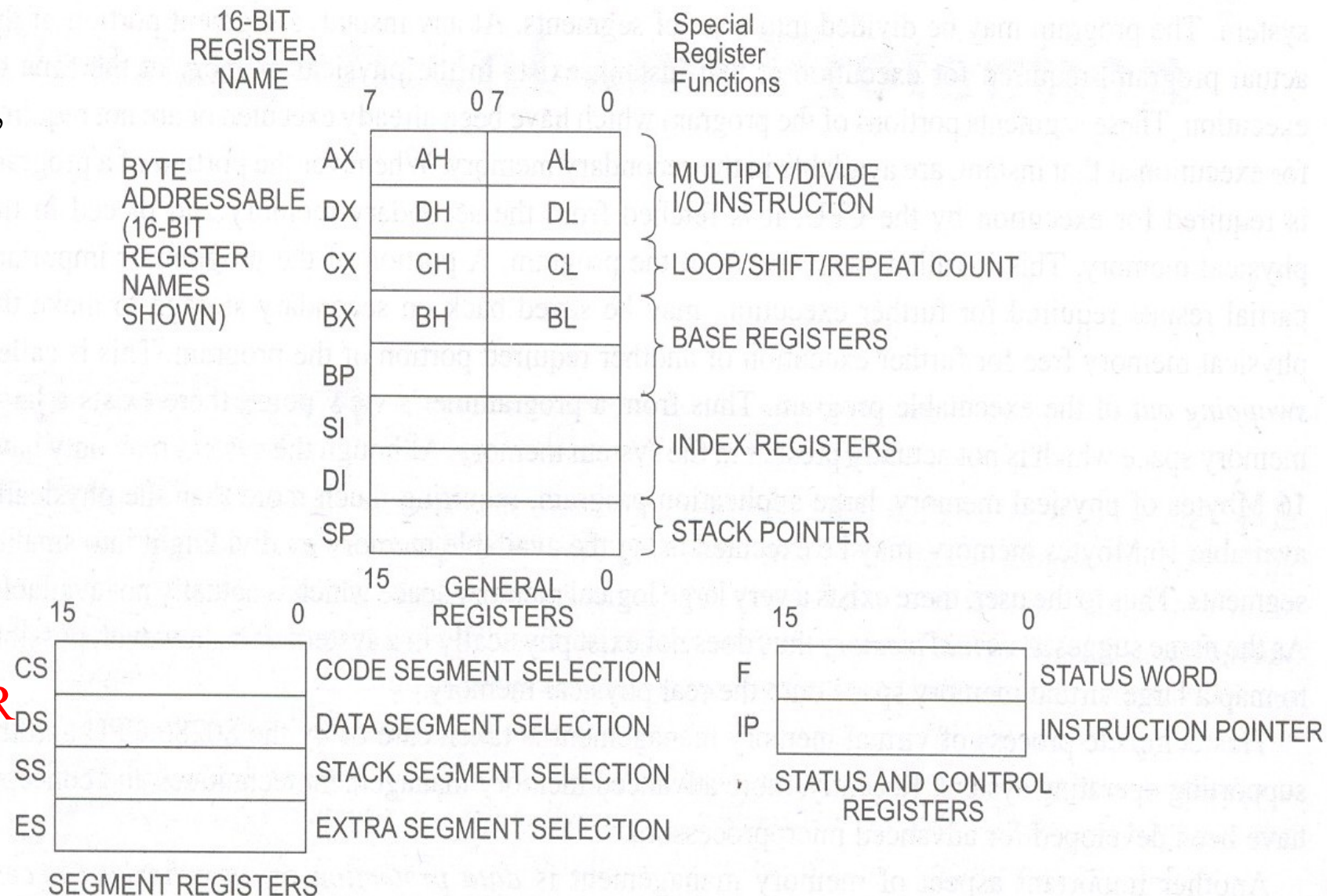


2. Protected Mode:

- All 24 address lines used and can access up to 16MB of physical memory.
- If descriptor table scheme is used it can address up to 1 GB of virtual memory.

Register organization of 80286

1. Eight 16-bit general purpose registers (AX, BX, CX, DX)
2. Four 16-bit segment registers (CS, SS, DS, ES)
3. Status and control registers (SP, BP, SI, DI)
4. Instruction Pointer (IP)
5. Two 16-bit register - **FLAGS, MSW**
6. Two 16-bit register - **LDTR and TR**
7. Two 48-bit register - **GDTR and IDTR**



80286 - Flag Register

| | | | | | | | | | | | | | | |
|----|----|------|----|----|----|----|----|----|---|----|---|----|---|----|
| 15 | 14 | 13 | 12 | | | | | | | | | | | 0 |
| - | NT | IOPL | OF | DF | IF | TF | SF | ZF | - | AF | - | PF | - | CF |

- **IOPL – I/O Privilege Level flag:**

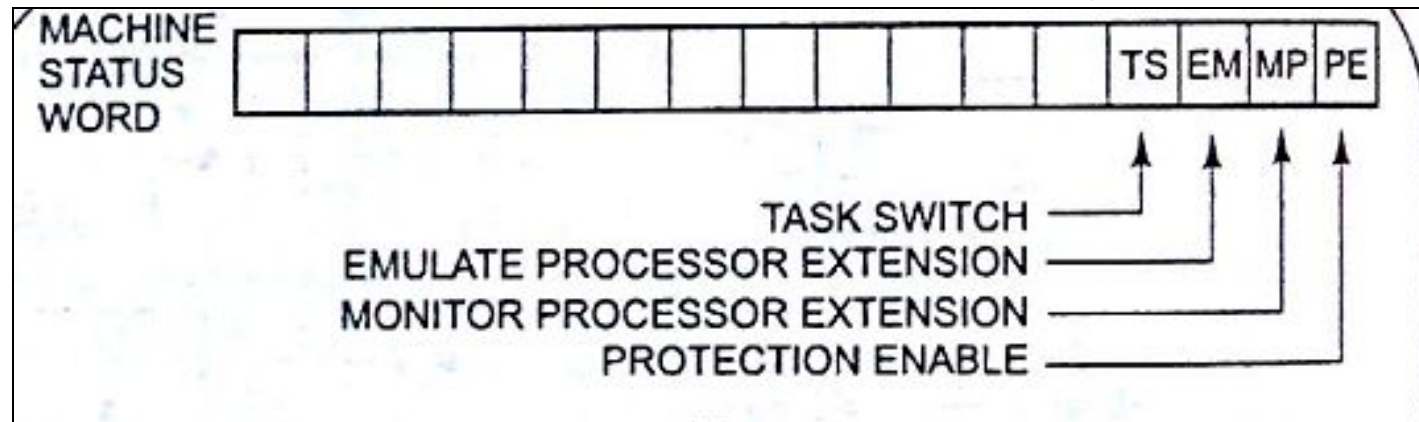
- 2 –bits are used in protected mode. (Bit 12 & 13)
- It holds the privilege level from 0 to 3. ‘0’ assigns to highest privilege whereas ‘3’ assigns to lower privilege level.

- **NT: Nested Task flag:**

- It is used in protected mode. (Bit 14)
- Bit is set when one task invokes another task.

80286 - Machine Status Word (MSW) Register

- **Bit 0** of the MSW is used to switch the 80286 into protected mode.
 - Once an 80286 is switched into protected mode by executing the LMSW instruction, the only way to get an 80286 back to its real address mode is by resetting the system. (PE=0 (Real Mode), PE=1 (Protected Mode))
- **Bit 1:** When MP=1, the 80287 floating point coprocessor is assumed to be attached. If MP=0, No coprocessor is attached.
- **Bit 2:** Emulation, indicates whether coprocessor functions are to be emulated or not. (IF EM=1, it generate exception 11 once it fetches floating point instruction)
- **Bit 3:** Task Switched, set and interrogated by coprocessor on task switches and when interpreting coprocessor instructions
- LMSW & SMSW instruction are available in the instruction set of 80286 to write and read the MSW in real address mode.



TS

EM

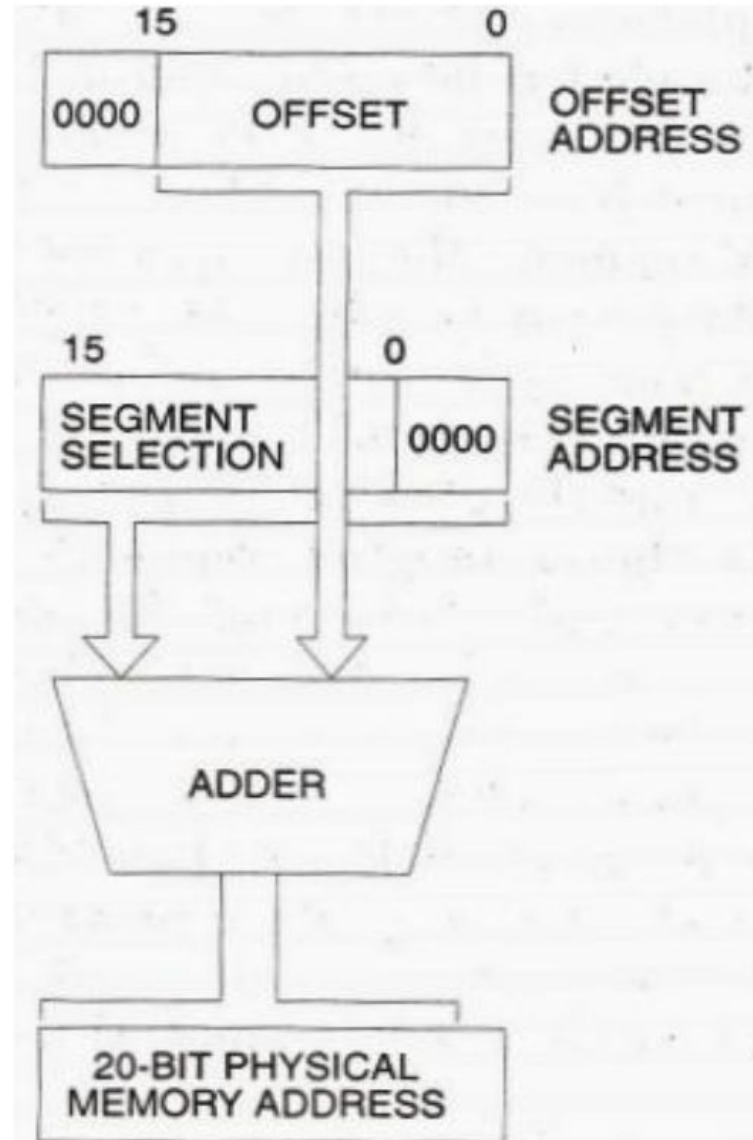
MP

PE

$p \xi = 0$ 1. Real Address Mode of 80286 (Just act as a fast 8086)

- Because of extra pipelining and other circuit level improvements, in real address mode also, the 80286 operates at a much faster rate than 8086, although functionally they work in an identical fashion.
- As in 8086, the physical memory is organized in terms of segments of 64Kbyte maximum size.
- In the real mode the first 1Kbyte of memory starting from address 00000H to 003FFH is reserved for **interrupt vector table**.
- The addresses from FFFF0H to FFFFFH are reserved for **system initialization**.
- When the 80286 is **reset**, it always starts the execution in **real address mode**.
- In real address mode, it initializes the IP and other registers of 80286.

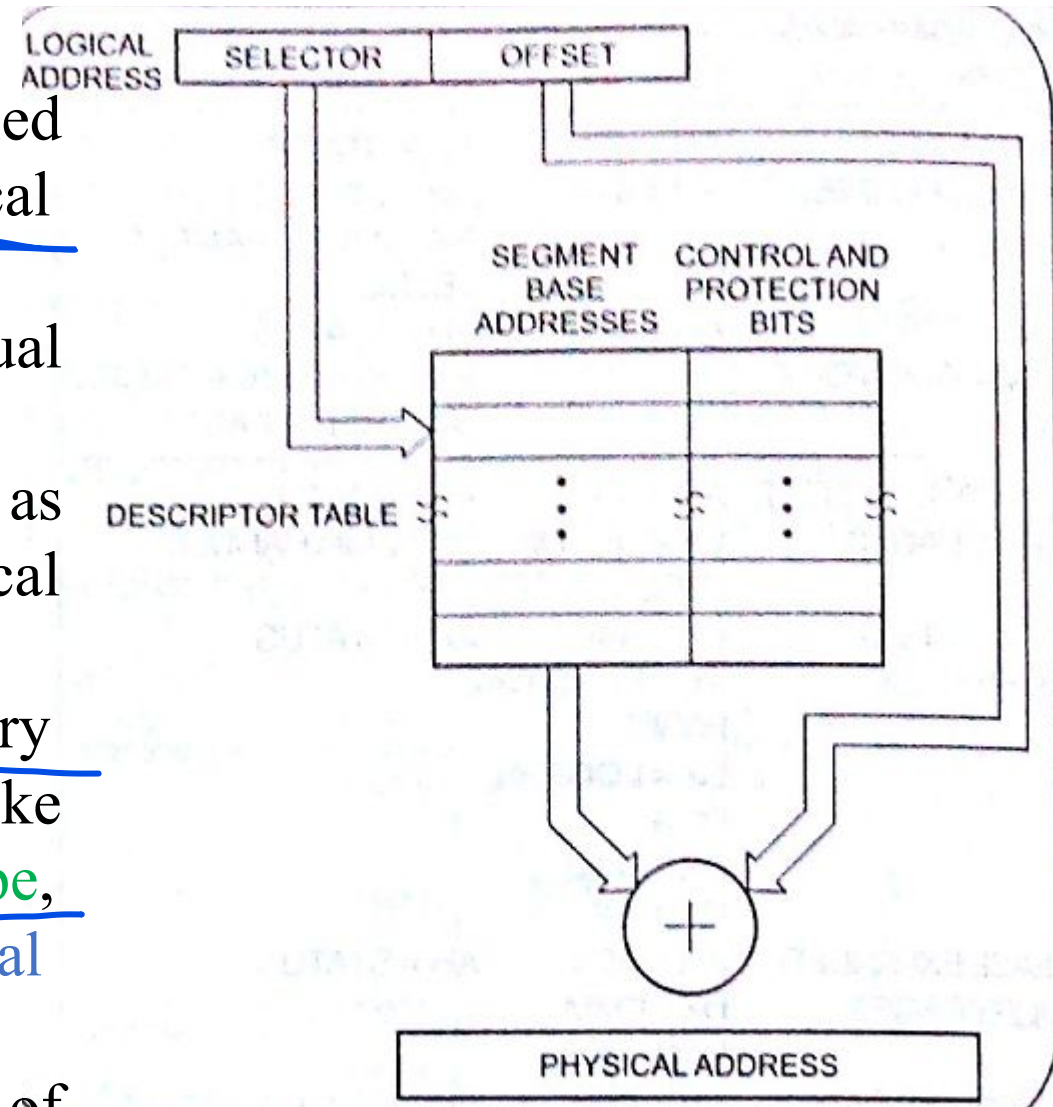
20-bit Physical Address generation in Real Mode



2. Protected Virtual Address Mode (PVAM)

- 80286 is the first processor to support the concepts of virtual memory and memory management.
- The concept of Virtual Memory is implemented using physical memory that the CPU can directly access and secondary memory that is used as storage for data and program, which are stored in secondary memory initially.
- The complete virtual memory is mapped on to the 16 Mbyte physical memory.
- If a program larger than 16Mbyte is stored on the hard disk and is to be executed, it is fetched in terms of data or program segments of less than 16Mbyte in size into the program memory by swapping sequentially as per sequence of execution.

- When you write an assembly language program, you usually refer to addresses by name.
- The address you work with in a program are called logical addresses because they indicate the logical position of code and data.
- The 80286 is able to address 1 GB (2^{30} bytes) of virtual memory.
- 80286 uses the 16-bit **content of a segment register** as a selector to address a descriptor stored in the physical memory.
- The **descriptor** is a block of contiguous memory locations containing information of a segment, like segment base address, segment limit, segment type, privilege level, segment availability in physical memory descriptor type and segment.
- Hardware reset is the only way to come out of protected mode





FEATURES OF 80286:

- 16-bit high performance CPU.
- 16-bit data bus and 24-bit address bus. Address directly $2^{24} = 16 \text{ MB}$
- Supports **1GB of Virtual memory**
- Supports real and protected modes
- In real mode, it works as the faster 8086 and access only 1 MB memory.
- In **protected mode** it can access 16 MB memory. This mode is used to implement the operating systems with multiuser and multitasking.
- Additional Instruction and Addressing modes
- Improved pipelining with four internal functional units
- External math coprocessor
- The register set is same as 8086 with an additional register **MSW (Machine Status Word)**
- ➔ Four Segment Privilege level. The segment with a lower privilege level can not access segment with a higher privilege level directly.

Intel family of microprocessor, bus and memory sizes

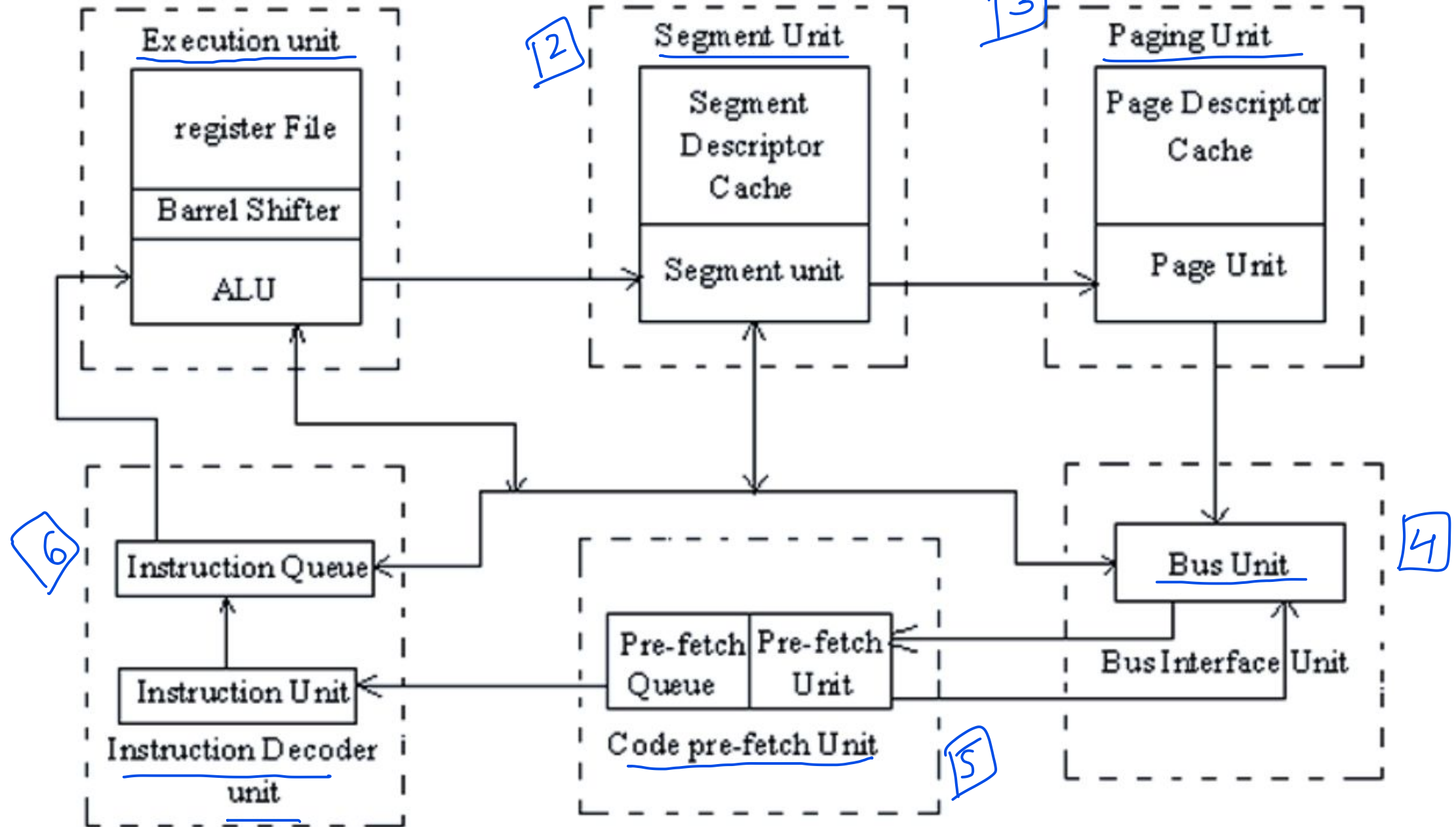
| Microprocessor | Data bus width | Address bus width | Memory size | U.M |
|--------------------|----------------|-------------------|-------------|------|
| 8086 | 16 | 20 | 1MB | No |
| 80186 | 16 | 20 | 1MB | No |
| 80286 | 16 | 24 | 16MB | 1GB |
| 80386 | 32 | 32 | 4GB | 64TB |
| 80486 | 32 | 32 | 4GB | |
| Pentium 4 & core 2 | 64 | 40 | 1TB | |

FEATURES OF 80386

- 1) 32bit data bus and 32 bit non multiplexed address bus so $2^{32} = 4\text{GB}$
- 2) Fully 32-bit CPU
- 3) Support 64TB Virtual memory
- 4) Operates in three modes **real, protected and virtual 8086**
- 5)  In real mode, it works as the faster 8086 and access only 1 MB memory.
- 6)  Protected mode provides MMU with memory segmentation, protection and multitasking.
 - Segment size from 1byte to 4GB .
 - Support paging mechanism to implement demand paging
- 7) Provides 32-bit extended registers, six 16-bit segment registers, protected mode registers, control registers, debug registers and test registers
- 8) Improved Instruction and new Addressing modes.
- 9) Improved pipelining with Six functional units

80386 Architecture

6 - Part

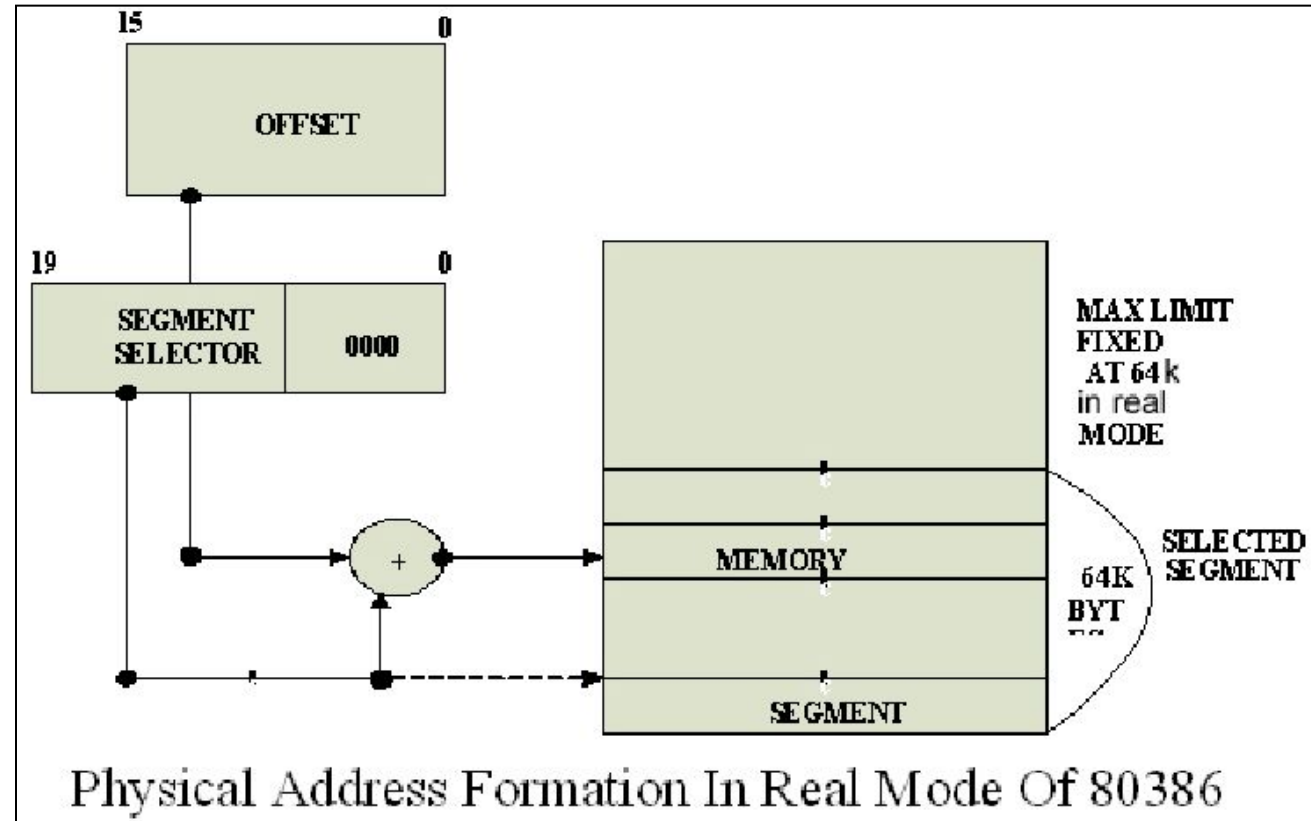


Real mode of 80386

- After reset, the 80386 starts from memory location **FFFFFFF0H** under the real address mode.
- In the real mode, 80386 works as a fast 8086 with 32-bit registers and data types.
- In real mode, the default operand size is 16 bit but 32-bit operands and addressing modes may be used with the help of override prefixes.
- The segment size in real mode is 64k, hence the 32-bit effective addressing must be less than **0000FFFFH**.
- The real mode initializes the 80386 and prepares it for protected mode.
- The segments in 80386 real mode may be overlapped or non-overlapped.
- The interrupt vector table of 80386 has been allocated 1Kbyte space starting from **00000000H** to **000003FFH**.

Real mode of 80386

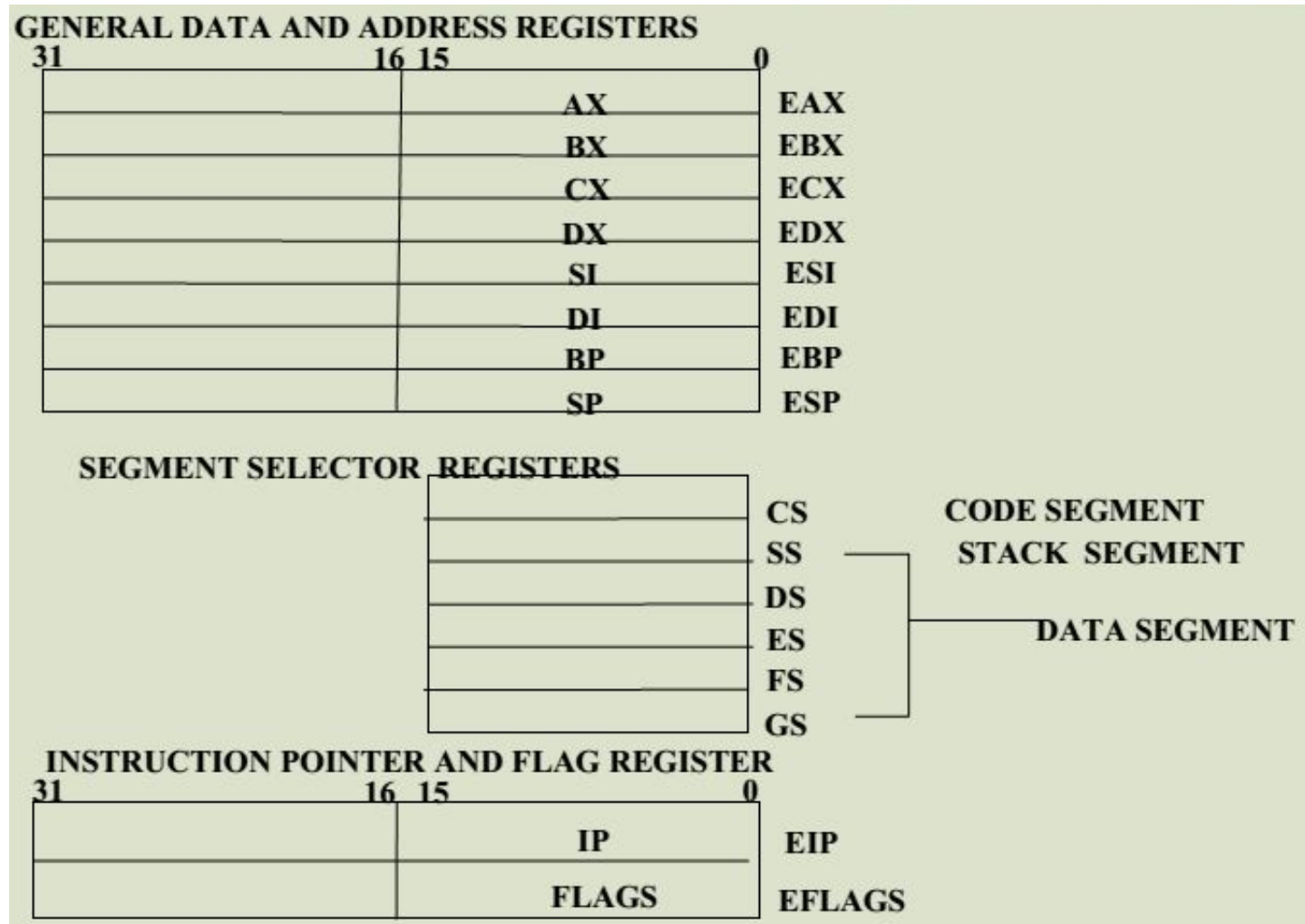
- In the real mode, the 80386 can address at the most 1Mbytes of physical memory using address lines A0-A19. (*20-bit*)
- Paging unit is disabled in real addressing mode, and hence the real addresses are the same as the physical addresses.
- Physical address calculation is same as 8086.



- The segment in 80386 real mode can be read, write or executed, i.e. no protection is available.
- Any fetch or access past the end of the segment limit generate exception 13 in real address mode.

Register Organization

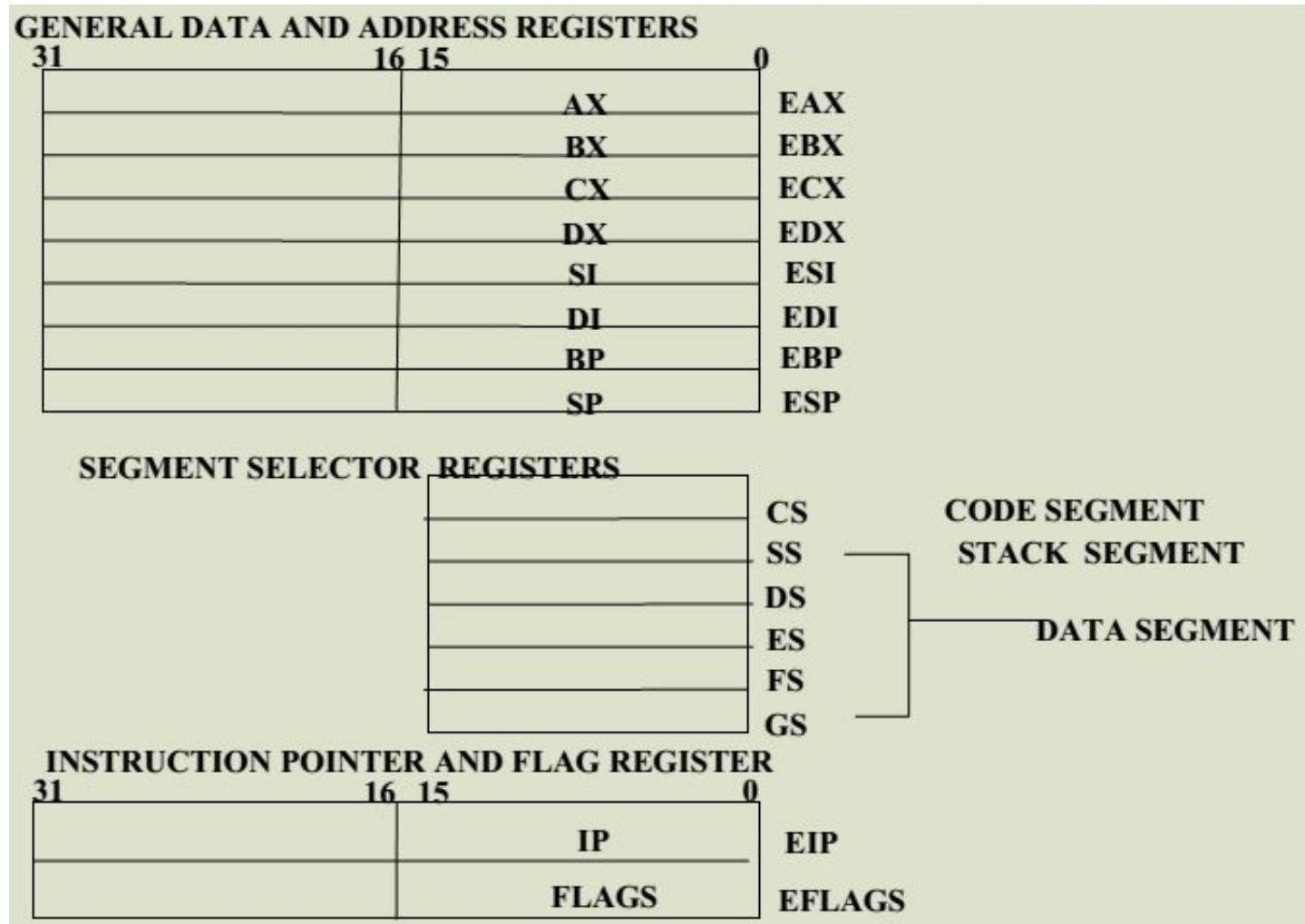
- The 80386 has eight 32 - bit general purpose registers which may be used as either 8 bit or 16 bit registers.
- A 32-bit register known as an extended register, is represented by the register name with prefix E.
- Example : A 32 bit register corresponding to AX is EAX, similarly BX is EBX etc.



- The 16 bit registers BP, SP, SI and DI in 8086 are now available with their extended size of 32 bit and are names as EBP,ESP,ESI and EDI.

Register Organization

- BP, SP, SI, DI represents the lower 16 bit of their 32 bit counterparts, and can be used as independent 16 bit registers.
- The six segment registers available in 80386 are CS, SS, DS, ES, FS and GS.
- The CS and SS are the code and the stack segment registers respectively, while DS, ES, FS, GS are 4 data segment registers.
- A 16 bit instruction pointer IP is available along with 32 bit counterpart EIP.



Flag Register of 80386

| | 31 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|-------|-----------------------|----|----|----|----|----|----|----|------|----|----|----|----|----|----|---|----|---|----|---|----|
| FLAGS | RESERVED FOR INTEL | | | | VM | RF | 0 | NT | IOPL | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | 1 | CF |

- The Flag register of 80386 is a 32 bit register.
- Out of the 32 bits, Intel has reserved bits D18 to D31, D5 and D3, while D1 is always set at 1.
- Two extra new flags are added to the 80286 flag to derive the flag register of 80386.
- They are VM and RF flags.

Flag Register of 80386

| | 31 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|-----------------------|----|----|----|----|----|------|----|----|----|----|----|----|---|----|---|----|----|----|---|
| FLGS | RESERVED FOR INTEL | | VM | RF | 0 | NT | IOPL | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | IF | CF | |

- *VM - Virtual Mode Flag:*
- If this flag is set, the 80386 enters the virtual 8086 mode.
- This is to be set only when the 80386 is in protected mode.
- In this mode, if any privileged instruction is executed an exception will generated.
- This bit can be set using IRET instruction or any task switch operation only in the protected mode.
- if VM=0, 80386 is in real or protected mode.

Flag Register of 80386

| | 31 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|-----------------------|----|----|----|----|----|------|----|----|----|----|----|----|---|----|---|----|----|----|---|
| FLAGS | RESERVED FOR INTEL | | VM | RF | 0 | NT | IOPL | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | IF | CF | |

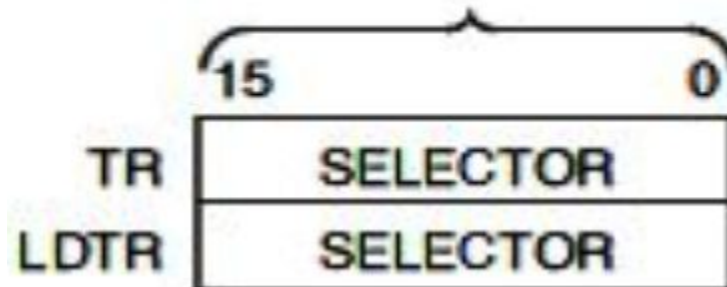
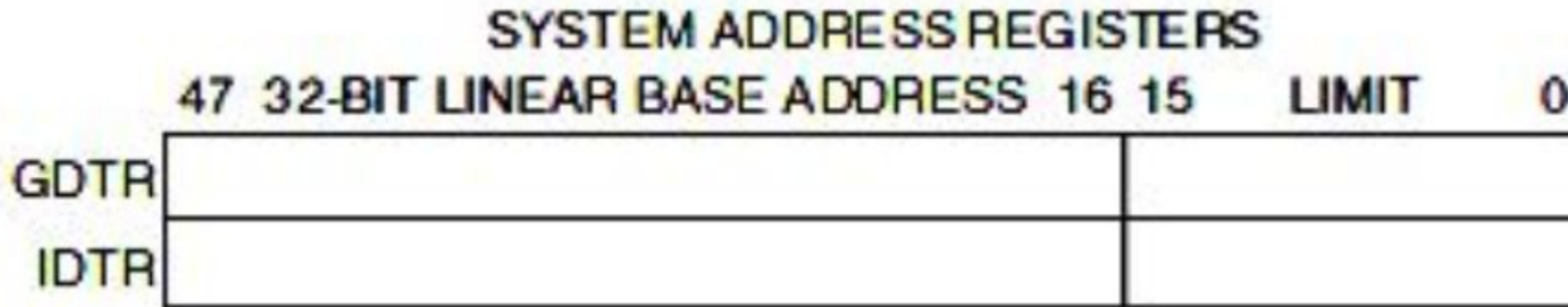
- *RF- Resume Flag:*
- This flag is used with the debug register breakpoints.
- It is checked at the starting of every instruction cycle and if it is set, any debug fault is ignored during the instruction cycle.
- The RF is automatically reset after successful execution of every instruction, except for IRET and POPF instructions.

Special Purpose Registers of 80386

- The special purpose register include four protected mode registers, four control registers, eight debug register and two test registers.

(1) Four Protected mode registers

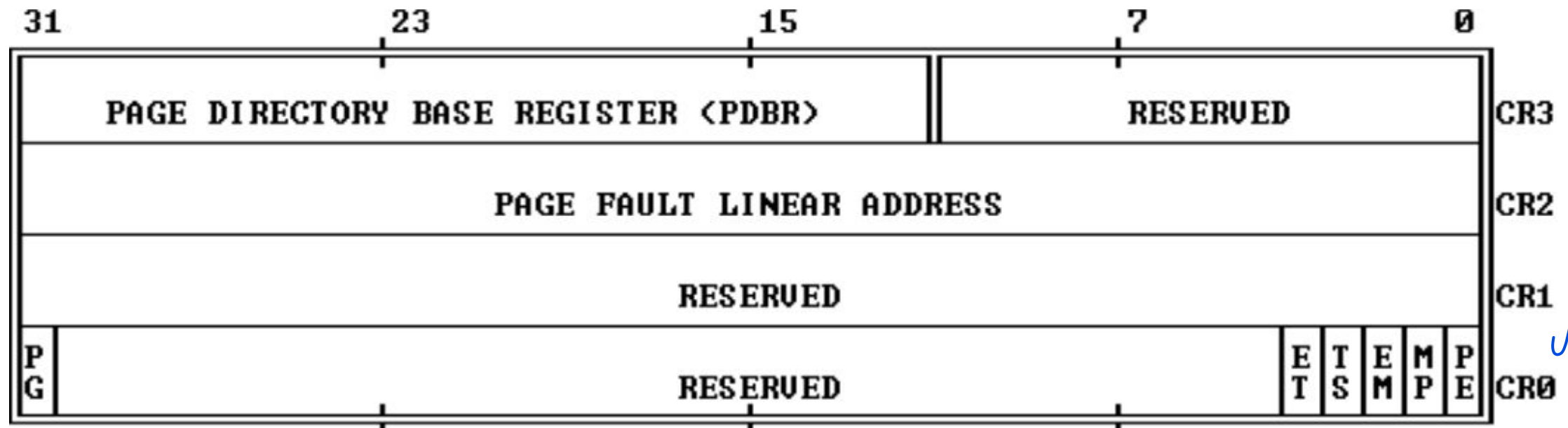
- GDTR (48-bits), IDTR (48-bits), LDTR (16-bits), TR (16-bits).
- They are used in protected mode for memory segmentation, protection and multitasking.



Special Purpose Registers of 80386

(2) Control Register

- The 80386 has four 32-bit control registers. CR0, CR1, CR2 and CR3.
 - CR0** contains total six flags.
- PE (Protection enable). PE=0 real mode & PE=1 protected mode
 - MP (Math/floating point coprocessor Present). MP=1 coprocessor is present. MP=0 No coprocessor is attached.
 - EM (Emulate floating point coprocessor): IF EM=1, generate Exception 11 when receives floating point instruction.
 - TS (Task Switched) The processor sets TS with every task switch.
 - ET (Extension type) coprocessor attached is 80287 or 80387
 - PG(Paging) PG=1 MMU is enable.
- CR1** is reserved while **CR2** and **CR3** are used for paging mechanism.



Special Purpose Registers of 80386

(3) Debug Registers:

- Debugging of 80386 allows data access breakpoints as well as code execution breakpoints.
- Intel has provide a set of 8 debug registers for hardware debugging.
- Two registers DR4 and DR5 are Intel reserved.
- Linear Breakpoint Address Registers:
 - ❑ The breakpoint addresses specified are 32-bit linear addresses
 - ❑ While debugging, Intel 386 h/w continuously compares the linear breakpoint addresses in DR0-DR3 with the linear addresses generated by executing software

| | |
|--------------------------------|---|
| 31 | 0 |
| LINEAR BREAKPOINT ADDRESS 0 | |
| LINEAR BREAKPOINT ADDRESS 1 | |
| LINEAR BREAKPOINT ADDRESS 2 | |
| LINEAR BREAKPOINT ADDRESS 3 | |
| Intel reserved. Do not define. | |
| Intel reserved. Do not define. | |
| BREAKPOINT STATUS | |
| BREAKPOINT CONTROL | |

DR0
DR1
DR2
DR3
DR4
DR5
DR6
DR7

8-Debug
Reg

Special Purpose Registers of 80386

(4) Test Registers:

- Two more test register are provided by 80386 for page caching
- They are used to control the testing of Translation Look-aside Buffer (TLB) of Intel 80386.
- TR6 is the command test register
- TR7 is the data register which contains the data of Translation Look-aside buffer test

| | |
|--------------|-----|
| TEST CONTROL | TR6 |
| TEST STATUS | TR7 |

TLB — Translation Look-aside Buffer (TLB)

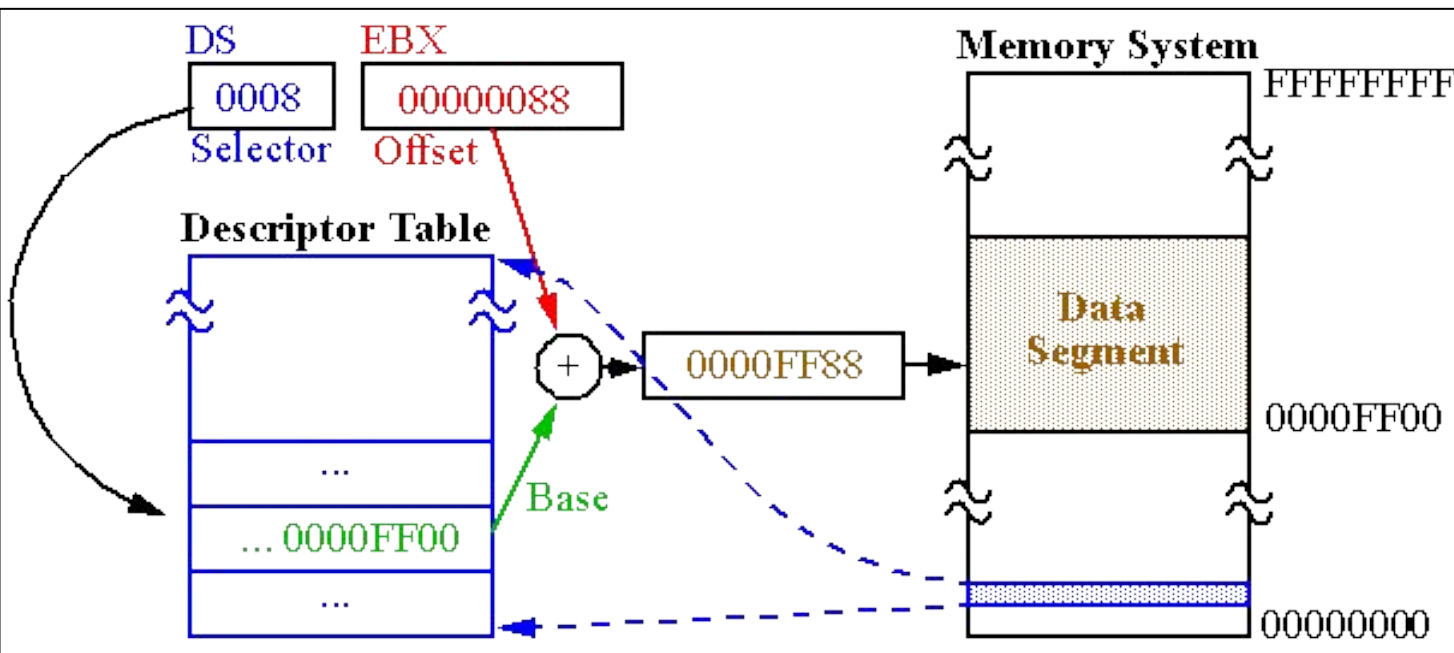
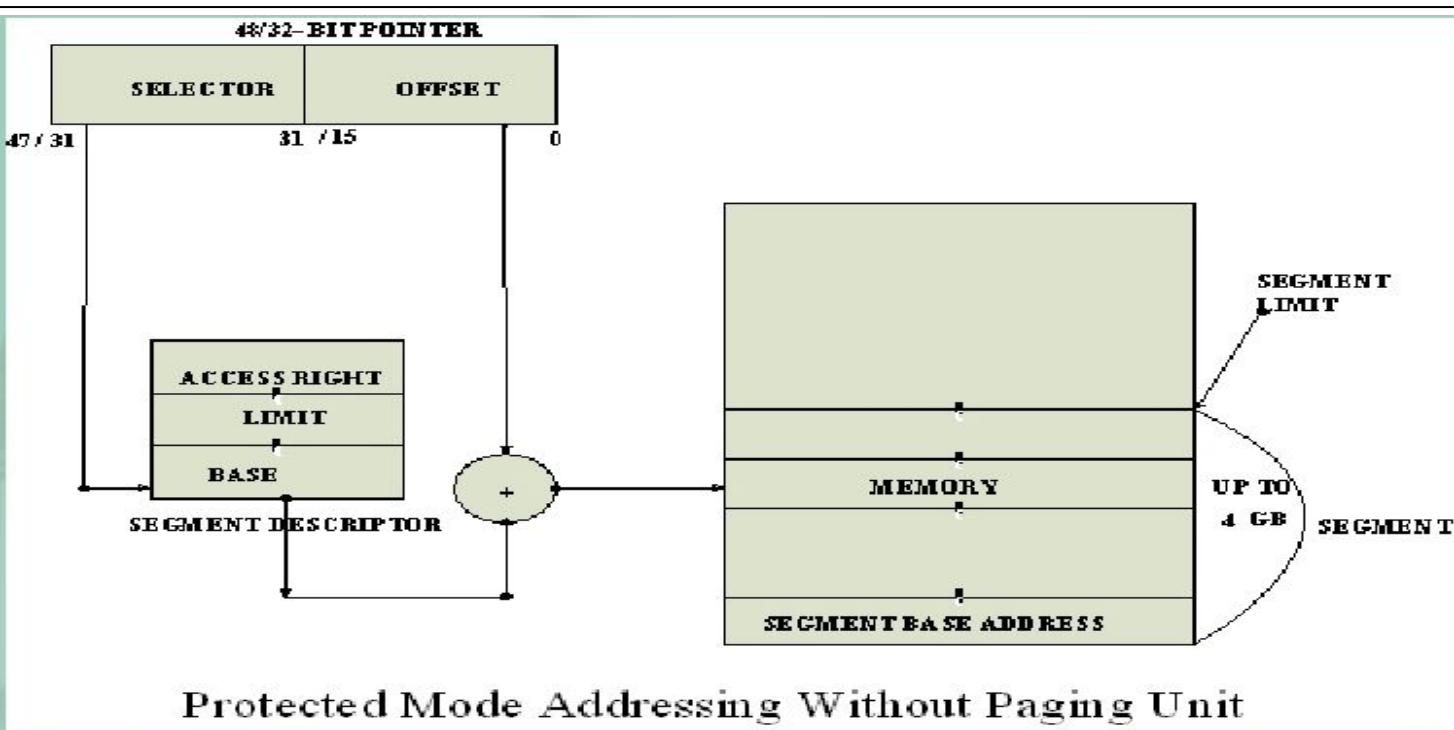
Protected Mode of 80386

- All the capabilities of 80386 are available for utilization in its protected mode of operation.
- The 80386 in protected mode support all the software written for 80286 and 8086 to be executed under the control of memory management and protection abilities of 80386.
- The protected mode allows the use of additional instruction, addressing modes and capabilities of 80386.

Protected Mode of 80386

ADDRESSING IN PROTECTED MODE:

- In this mode, the **contents of segment registers** are used as **selectors** to address descriptors which contain the segment limit, base address and access rights byte of the segment.
- The effective address (offset) is added with segment base address to calculate linear address.
- This linear address is further used as physical address, if the paging unit is disabled, otherwise the paging unit converts the linear address into physical address.
- The paging unit is a memory management unit enabled only in protected mode.
- The paging mechanism allows handling of large segments of memory in terms of pages of 4Kbyte size.



Protected Mode of 80386

- *Segments are interpreted differently in Protected Mode as compared to Real Mode:*
- *Segment register contains a selector that selects a descriptor from the descriptor table.*
- *The descriptor contains information about the segment, e.g., its base address, length and access rights.*
- *The offset can be 16 bits/32-bits.*

80386 Descriptor Format

| Base (31-24) | | G | D | X | U | Limit (19-16) | | P | DPL | | S | TYPE | | A | Base (23-0) | | Limit (15-0) | |
|--------------|----|----|----|----|----|---------------|----|----|-----|----|----|------|----|----|-------------|----|--------------|---|
| 63 | 56 | 55 | 54 | 53 | 52 | 51 | 48 | 47 | 46 | 45 | 44 | 43 | 41 | 40 | 39 | 16 | 15 | 0 |

Access Rights Byte →

| 15 | | 8 | | | 7 | | 0 | | | | |
|-------------|-----|---|------|---|-------------|---|---|---|------------------|---|--|
| Base(31-24) | | | | | G | D | X | U | Limit (19-16) | 6 | |
| P | DPL | S | TYPE | A | Base(23-16) | | | | | 4 | |
| Base(15-0) | | | | | | | | | | 2 | |
| Limit(15-0) | | | | | | | | | | 0 | |

80386 Descriptor Format

| 15 | | | | | 8 | | | | | 7 | | | | | 0 | | | | |
|-------------|-----|--|---|------|---|---|-------------|--|--|---|---|---|---|----------------------|---|--|--|--|--|
| Base(31-24) | | | | | | | | | | G | D | X | U | Limit (19-16) | 6 | | | | |
| P | DPL | | S | TYPE | | A | Base(23-16) | | | | | | | | 4 | | | | |
| Base(15-0) | | | | | | | | | | | | | | | 2 | | | | |
| Limit(15-0) | | | | | | | | | | | | | | | 0 | | | | |

- The Limit field defines the length of a segment in bytes minus 1.
- It is a 20 bits field with bits 0-15 defining the lower 16-bits and bits 16-19 defining the upper 4-bits.
- The value stored in limit field is length in bytes minus 1.

- E.g. if limit field is **003FFh** it means that valid offset range is 0000 to 003FFh and the **length of the segment is 00400h**.
- To define a segment with only 1 byte the limit field would be 00000h.
- To define a segment of 1MB the limit field would be FFFFFh.
- Above definition of Limit field is true when **G=0**, the length is measured in bytes.
- When **G=1** length is measured in 4KB(chunks)
- E.g. Limit field is 0000 and G=1 the length of segment is 4KB
- Limit field is 00001 and G=1 the length of segment is 8KB
- Limit field is FFFFFh and G=1 the length of segment is 100000h * 4 KB = 4GB

= last Addr + 1

80386 Descriptor Format

| | | | | | | | | | | |
|-------------|-----|---|------|---|-------------|---|---|---|------------------|---|
| Base(31-24) | | | | | G | D | X | U | Limit (19-16) | 6 |
| P | DPL | S | TYPE | A | Base(23-16) | | | | | 4 |
| Base(15-0) | | | | | | | | | | 2 |
| Limit(15-0) | | | | | | | | | | 0 |

| Type Value | Type of Segment |
|------------|--|
| 0 0 0 | Data segment, read only |
| 0 0 1 | Data segment, read/write |
| 0 1 0 | Stack segment, read only |
| 0 1 1 | Stack segment, read/write |
| 1 0 0 | Code segment, execute only |
| 1 0 1 | Code segment, execute/read |
| 1 1 0 | Code segment, execute only confirming |
| 1 1 1 | Code segment, execute /read confirming |

| Field | Position | Meaning | Function |
|----------------------------------|-----------|--|--|
| A (Access bit) | Bit 40 | It is set when a segment define by the descriptor is accessed. | A=0 segment has not been accessed A=1 segment has been accessed |
| TPYE (segment type) | Bit 41-43 | It define the type of segment (code, data, stack) | |
| S (System bit) | Bit 44 | If S=0 it is system segment descriptor. If S=1 non-system segment descriptor. | S=0 system segment descriptor S=1 code or data segment descriptor |
| DPL (Descriptor privilege level) | Bit 45-46 | It defines the privilege level of the segment from 0 to 3. | Segment privilege attribute used in privilege test. |
| P (Present bit) | Bit 47 | If P=1 segment is physically present otherwise it is not present in main memory. | |

80386 Descriptor Format

| | | | | | | | | | | |
|-------------|-----|---|------|---|-------------|---|---|---|------------------|---|
| Base(31-24) | | | | | G | D | X | U | Limit (19-16) | 6 |
| P | DPL | S | TYPE | A | Base(23-16) | | | | 4 | |
| Base(15-0) | | | | | | | | | 2 | |
| Limit(15-0) | | | | | | | | | 0 | |

| Field | Position | Meaning |
|---------------------|----------|--|
| U (User bit) | Bit 52 | It is not used by 80386 and is left for the user to define it. |
| X (Reserved bit) | Bit 53 | It is reserved by intel can not be used by user. |
| D (Default bit) | Bit 54 | This bit define the size of the operands. If D=0,operands are 16-bit. If D=1,operands are 32-bits. |
| G (Granularity bit) | Bit 55 | When G=0,the length is measured in bytes. When G=1 length is measured in 4KB (chunks) |

Example -1

| | | |
|-----------|-----------|---|
| 0010 0110 | 0100 0000 | 6 |
| 1011 0011 | 0111 1111 | 4 |
| 0000 1100 | 0000 0010 | 2 |
| 0000 0011 | 1111 1111 | 0 |

| | | | | | | | | | | |
|-------------|-----|---|------|---|-------------|---|---|---|------------------|---|
| Base(31-24) | | | | | G | D | X | U | Limit (19-16) | 6 |
| P | DPL | S | TYPE | A | Base(23-16) | | | | | 4 |
| Base(15-0) | | | | | | | | | | 2 |
| Limit(15-0) | | | | | | | | | | 0 |

- The segment start from the linear address: 267F0C02h (Base).
- The length of the segment is 1KB (Limit=003FFh).
- The segment is a user defined data segment with read/write permission (S=1,Type= 001).
- The privilege level of segment is 1 (DPL=01).
- The segment is physically present (P=1) and already access (A=1).
- The operand in the segment are 32-bits (D=1)
- The Limit in terms of bytes (G=0)
- What will be the length of segment if G=1? $1\text{KB} * 4\text{KB} = 4\text{ MB}$

80386 Protected mode

- The 80386 starts in the real mode whenever it receives the reset.
- We can switch from real mode to protected mode by setting PE bit in the CR0 register.
- The protected mode permits memory management, virtual addresses, paging protection and multitasking capabilities through MMU.
- In protected mode for each segment one segment descriptor is created and managed.
- For a multitasking system at a time plenty of segments are in memory, which are parts of different task. Which needs large number of segment descriptors so that memory areas described by them are accessed by the processor for executing different task.

Descriptor tables and Selectors

- **How does the 80386 organize a large number of segment descriptors so that they can be easily accessed to make references to segment spaces?**
- The 80386 organizes the segment descriptors by grouping them in the tables depending on the purpose of the segment descriptors.
- Three types of the 80386 descriptor tables are listed as follows:
 - 1. GLOBAL DESCRIPTOR TABLE (GDT)**
 - (GDTR) - LGDT instruction executed system is booted
 - 2. LOCAL DESCRIPTOR TABLE (LDT)**
 - (LDTR) – LLDT – instruction - highest privilege Level
 - 3. INTERRUPT DESCRIPTOR TABLE (IDT)**
 - (IDTR)

GLOBAL DESCRIPTOR TABLE (GDT)

- The GDT represents global memory are shared by all the software tasks.
- The 80386 protected mode needs only one GDT.
- The global descriptor tables stores the segment descriptors, each of 8 bytes.
- The GDT stores maximum $2^{13} = 8192$ (0 to 8181) descriptors, each of 8 bytes making **maximum size of GDT is $8192 * 8 = 2^{13} * 2^3 = 2^{16} = 64 \text{ KB}$**
- Descriptor - 0 in GTD (GDT[0]) is all 0s reserved by intel and known as **null descriptor**.
- **Selectors:**
 - The 80386 contains six segment registers.
 - The segment registers in the real mode have the same role as 8086.
 - The role of segment registers entirely changes in the protected mode, segment register act as selector.

Segment Register

| INDEX to descriptor | | TI | RPL |
|---------------------|---|----|-----|
| 15 | 3 | 2 | 1 0 |

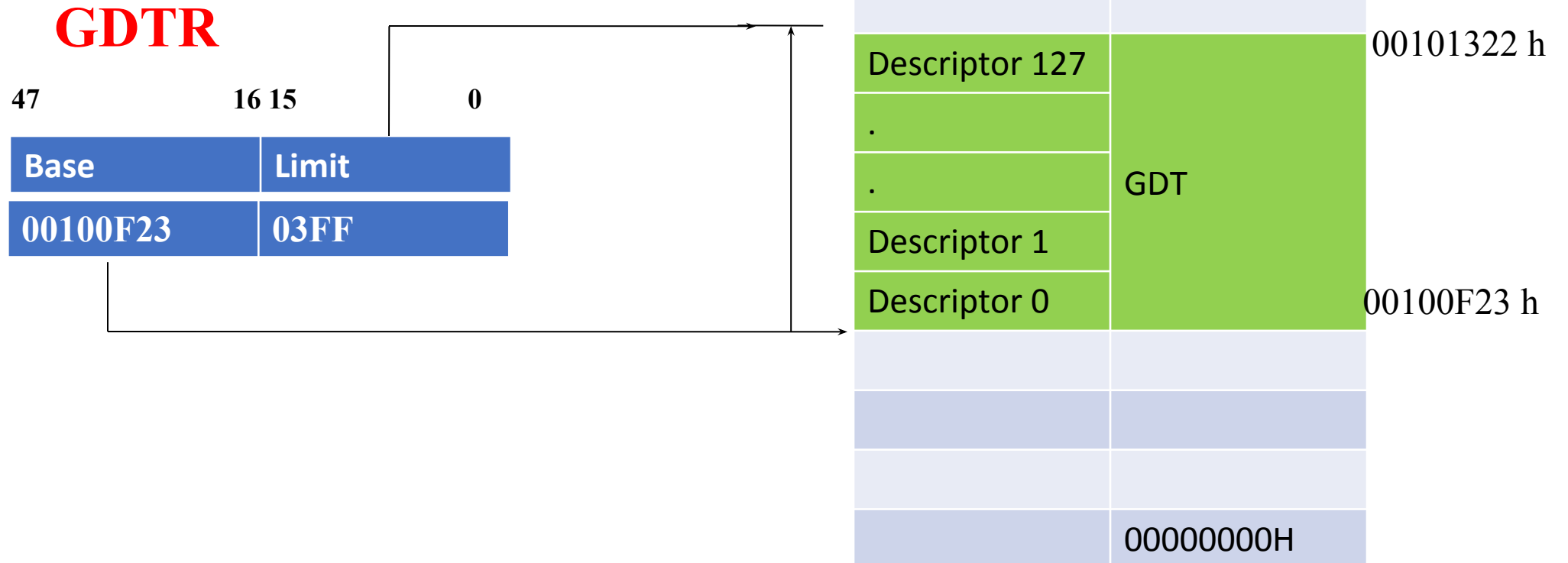
- RPL = requested privilege level (0 to 3)
- TI = Type indicator
 - TI=0 selector refers to GDT
 - TI=1 selector refers to LDT

Example - 2

Question: Given Base = 00100F23h and Limit=03FFh in the GDTR, What does it mean?

Answer:

- GDT Start from linear address 0100F23h
- The size of GDT is 03FFh+1=1024 bytes.
- The number of descriptor in GDT is 1024/8=128



Example - 3

Question: Given the selector value 2F32h, find the table referred to and the index of the pointed descriptor.

Answer:

- RPL = 10
- TI = 0 means refer to GDT.
- Index = 0010 1111 0011 0 B = $(1510)_{10}$
- It points to 1510 descriptor in GDT.

| 0010 | 1111 | 0011 | 0 | 0 | 1 0 |
|---------------------|------|------|---|----|-----|
| INDEX to descriptor | | | | TI | RPL |
| 15 | 3 | | | 2 | 1 0 |

Did You Note!!

- There is an 100 % degradation in Memory access time – because every memory access is two accesses now, one for getting the base address and another for actually accessing the data.
- **A solution indeed:** Along with the segment registers, keep a shadow registers (also known as invisible cache registers) which stores additional necessary information.

| Visible Part | Invisible Cache Register | | |
|------------------|--------------------------|--------------|-------|
| Segment Selector | Access Rights | Base Address | LIMIT |
| CS | | | |
| SS | | | |
| DS | | | |
| ES | | | |
| FS | | | |
| GS | | | |

80386 Segment Check-List

- When an attempt is made to access a segment by loading a segment selector into the visible part of a segment Register, 386 automatically makes several checks.
- To access a data segment for example, the selector might be loaded into the visible part of the DS, ES, FS or GS register.
- **First** of all it checks to see if the descriptor table indexed by the selector contains a valid descriptor for that selector.
 - If the selector attempts to access a location outside the limit of the descriptor table or the location indexed by the selector in the descriptor table does not contain a valid descriptor, then an exception is produced.
- **Second** it also checks to see if the segment descriptor is of the right type to be loaded into the specified segment register.
 - E.g.: Descriptor for a read only data segment, cannot be loaded into SS register because stack must be able to be Written to.
 - A Selector for the code segment is “execute only” so can not be loaded in to the DS register.

80386 Segment Check-List (Continue...)

- If privilege level of the selector and the privilege level of the current code segment is not as high as the privilege level of the descriptor an exception will be produced and access will not be allowed.
- If all these protection conditions are met, the limit, base and access right byte of the segment descriptor are copied into the invisible part of the segment register.
- The 386 then checks the P bit of the access byte to see if the segment descriptor is present in physical memory.
 - If it is not present type 11 exception is produced the exception handler will read the segment from disk to physical memory.
- The address produced by program does not fall outside the limit.
- Attempts to write a code segment or read only data segment will cause an exception.

Example - 4

Describe the following descriptor in detail. Selector of this descriptor must be loaded into which register of 80386 in PM?

| | | |
|------------|------------|----------|
| 00h | 40h | 6 |
| 9Bh | 80h | 4 |
| 00h | 00h | 2 |
| 00h | 20h | 0 |

Which are all the checks 80386 will do and will there be exception(s) due to this checks ?

Example - 5

The following is valid data segment descriptor and already cached in invisible portion of the DS :

| | | |
|-----|-----|---|
| 00h | CFh | 6 |
| 91h | 00h | 4 |
| 00h | 80h | 2 |
| FFh | FFh | 0 |

Now if following instruction is executed in PM of 80386 :

MOV [000FFFF0h],12345678h

Will there be any exception ? Justify your answer.

If any exception, suggest the modification in the descriptor to avoid that exception.

IDT (Interrupt Descriptor Table) & LDT (Local Descriptor Table)

- **IDT (Interrupt Descriptor Table):** IDT is used to stores the descriptors for exception handles.
- **LDT (Local Descriptor Table):** LDT is local to a task and can be created per task basis, but it is optional.
- The 80386 protected mode defines **only one GDT and IDT whereas LDT can be multiple**.
- The protected mode registers GDTR, IDTR and LDTR are used to keep track of GDT ,IDT and LDT in memory respectively.
- The task either use all the descriptors from the GDT if it does not define its own LDT.
- **An LDT is not define as independent table**, rather its defined as an extension to GDT containing the Base and Limit of the LDT.
- That means the maximum number of LDT can be 8192 if all the descriptors in the GDT are LDT descriptors.

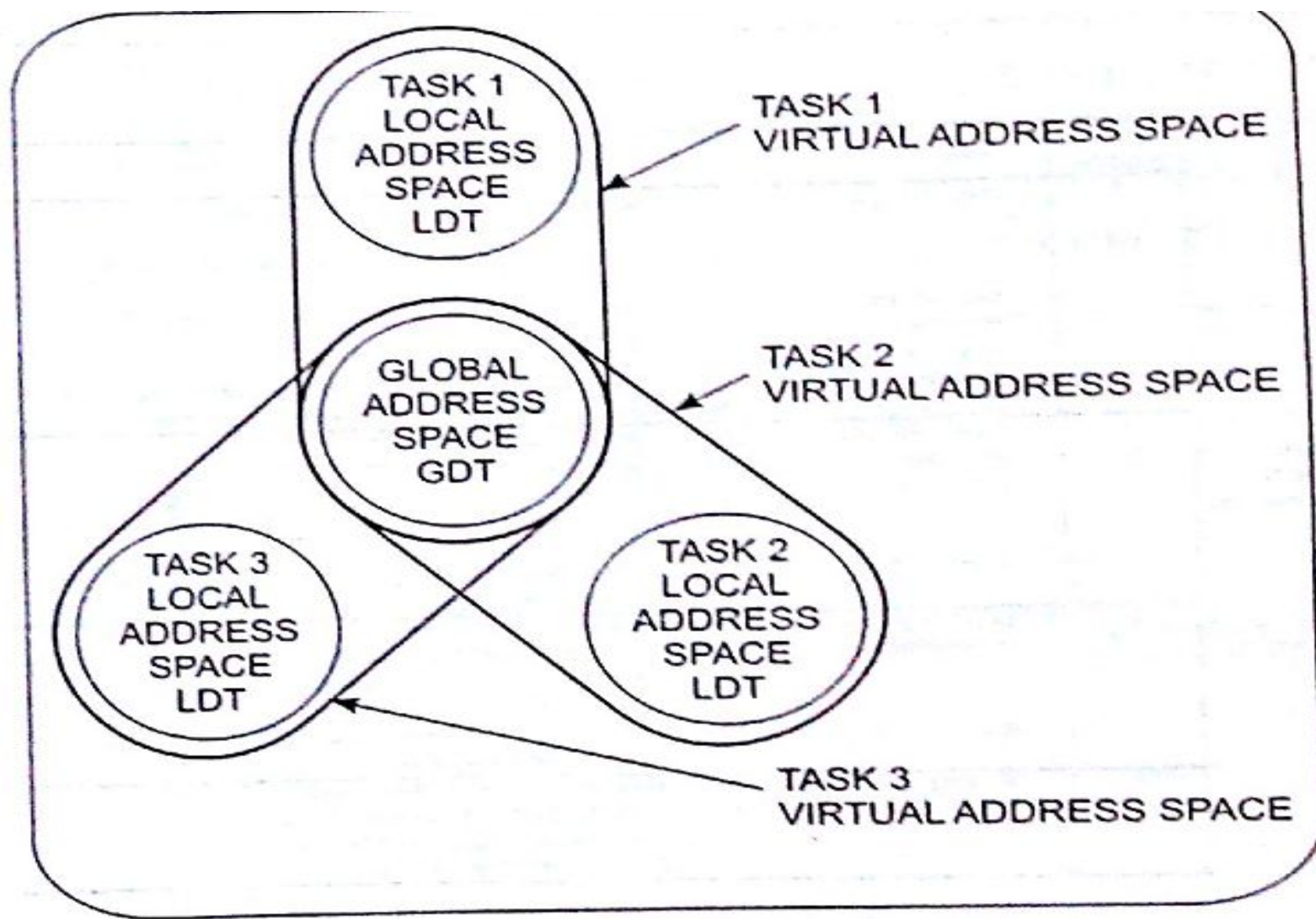


Fig. 15.22 Diagram showing how tasks can be isolated from each other by having separate local descriptor tables but can share a common global descriptor table. (Intel Corporation)

The system segment Descriptor of LDT

- This is the System Descriptor present in the GDT.
- Access bit is not required for System Descriptors.
- So, Type bits are of 4 bits.
- $2^4 = 16$ System Segment Descriptors are there.
- TYPE = 0010 indicates system descriptor for LDT table.
- This gives information about Base address and LIMIT of LDT.

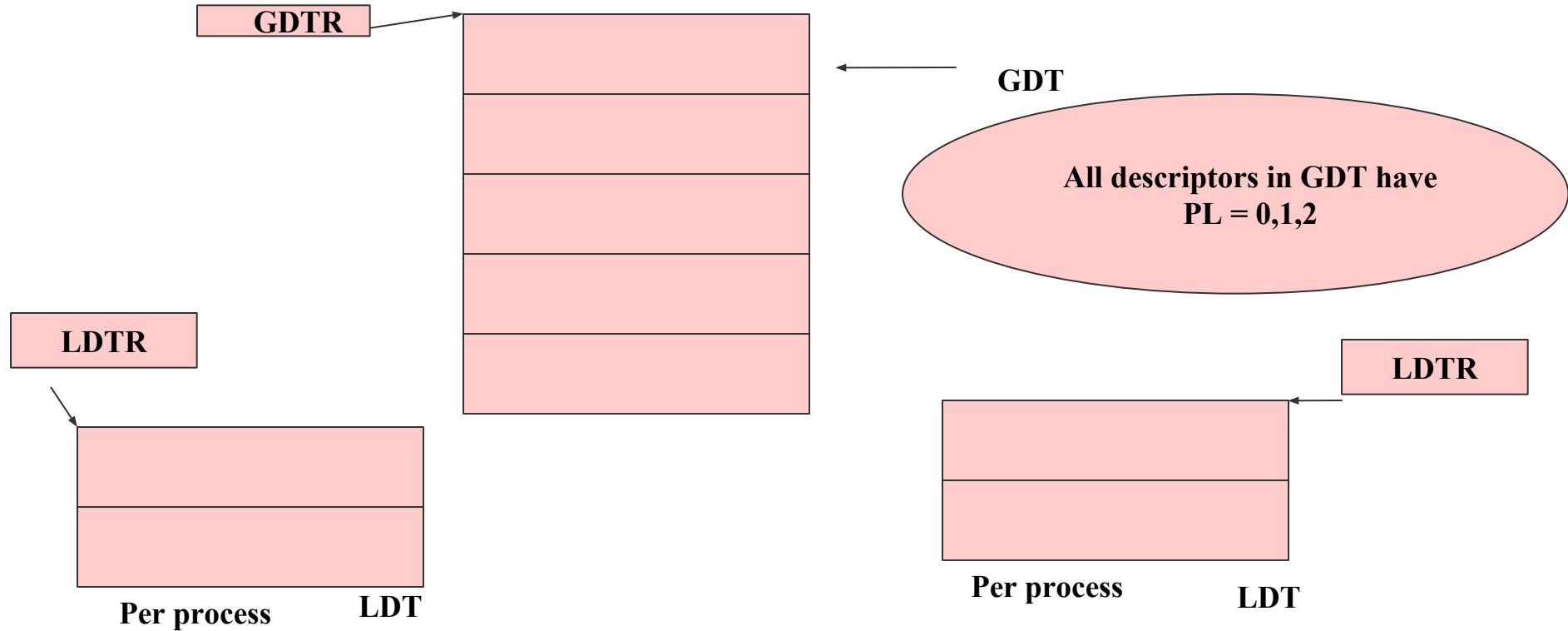
| 15 | | | | 8 | | | | 7 | | | | 0 | | | |
|-------------|---|-------|-----|---|---|------|---|-------------|---|---|---|------------------|---|---|--|
| Base(31-24) | | | | | | | | 0 | 0 | 0 | 0 | Limit (19-16) | | 6 | |
| P | 0 | 0 | (0) | 0 | 0 | 1 | 0 | Base(23-16) | | | | | | 4 | |
| | | (DPL) | | S | | TYPE | | | | | | | | | |
| Base(15-0) | | | | | | | | | | | | | 2 | | |
| Limit(15-0) | | | | | | | | | | | | | 0 | | |

LDT (Local Descriptor Table)

- User task can be protected from each other in a 386 system by giving each task its own local descriptor table.
- The LDT register which points to user's local descriptor table can only be changed with the **LLDT** instruction or by a task switch.
- The LLDT instruction can be executed only at the highest privileged level which is usually reserved for the operating system.
- Likewise a switch from one task to another task is done by the operating system at the highest privilege level.
- So, user task operating at lower privilege level cannot cause switch to other user tasks. Also because of the limit checking, a task cannot accidentally or intentionally access descriptor in the another task local descriptor table.

LDT (Local Descriptor Table)

- Two process each of PL = 3 should be allotted segments such that one should not access the segments of other.



- If at all each process should access memory, it has to use the descriptors in its LDTR only and it cannot change the LDTR/LDT/GDTR/GDT contents as they would be maintained in a higher privileged memory area.

Virtual Memory in 80386

- Virtual address is of 46 bit.

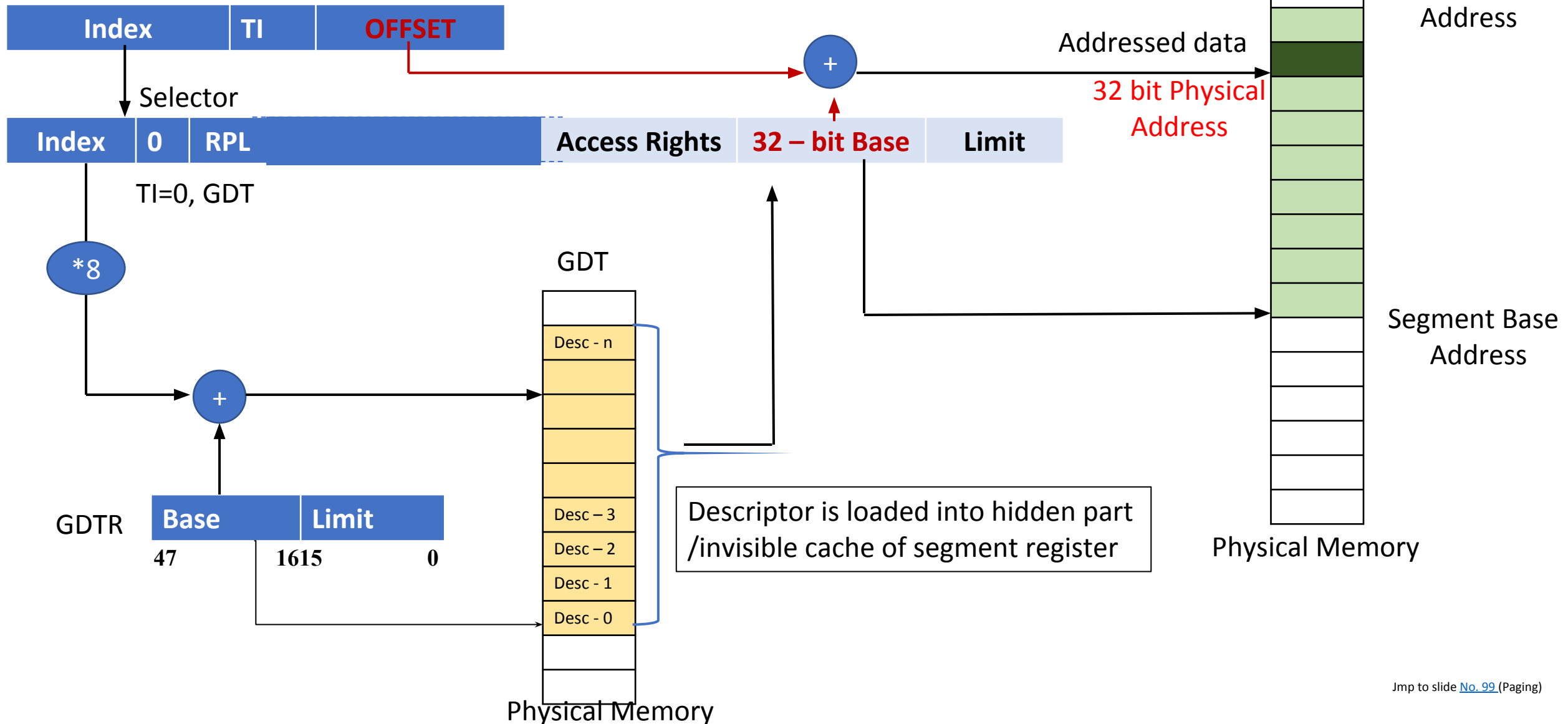


- 16-bit Selector: out of which only 14-bit is used for VA and 32 –bit offset.
- So, Virtual Memory Supported by 80386 is $2^{46} = 64\text{TB}$

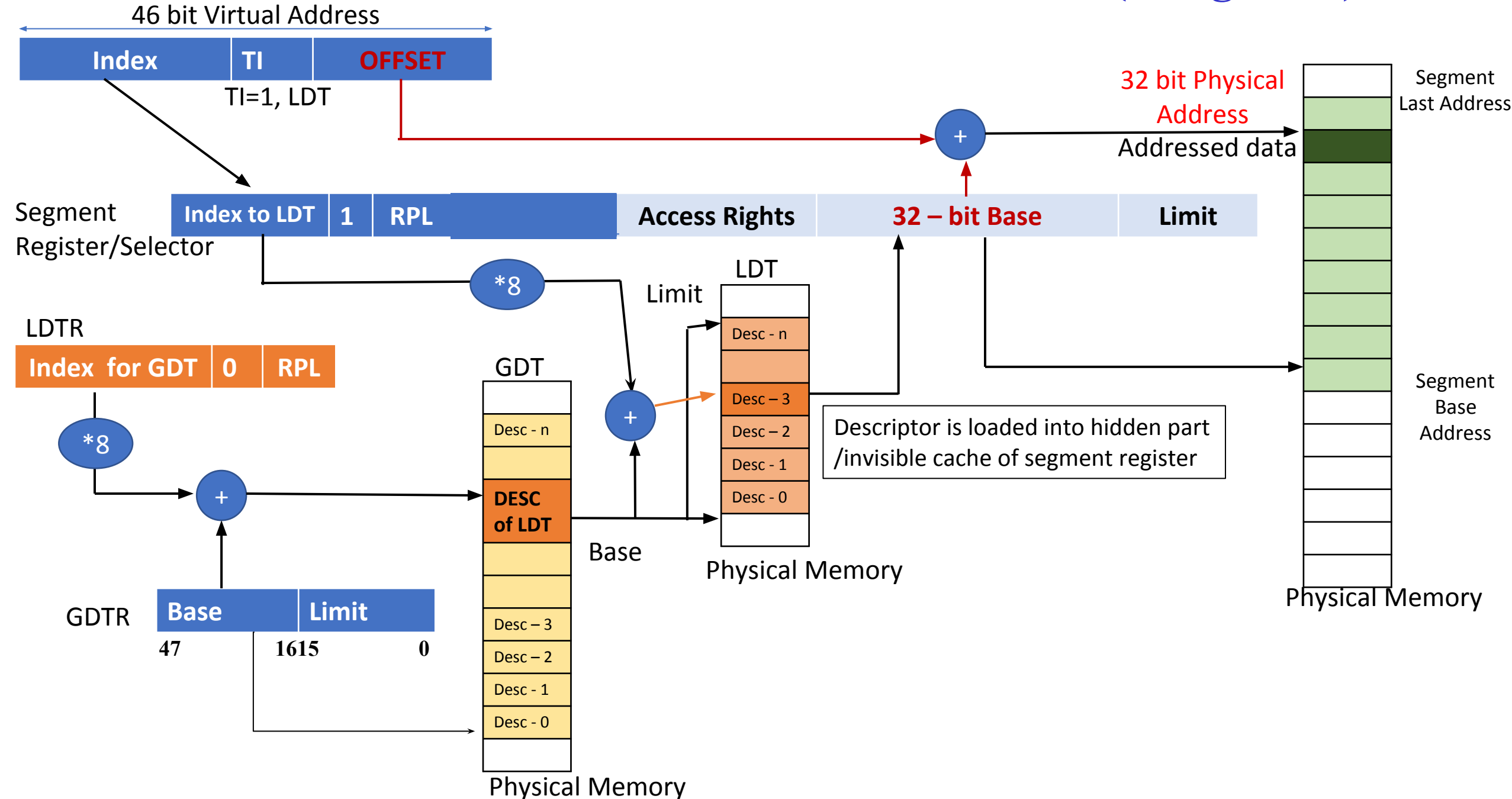
Virtual Address to Linear Address Translation (using GDT)

46 bit Virtual Address

(16-bit Selector (out of which only 14-bit is used for VA) + 32-bit offset)



Virtual Address to Linear Address Translation (using LDT)



Protection Model (Privilege levels) in 80386

- The memory Segmentation divides the system into chunks known as segments.
- Each segment is defined by its corresponding descriptor.
- The segment descriptors are stored in either GDT or LDTs.
- GDT defines global space and LDTs define local space for the tasks.
- A task can use descriptor from its own LDT and also from the GDT.
- A program under **execution is called the process** which performs a predefined function and is also called a **task**.
- 80386 can run a task at one of the four privilege levels – 0,1,2,3.
- Level 0 is the most privileged and level 3 is the least privileged.

Protection Model (Privilege levels) in 80386

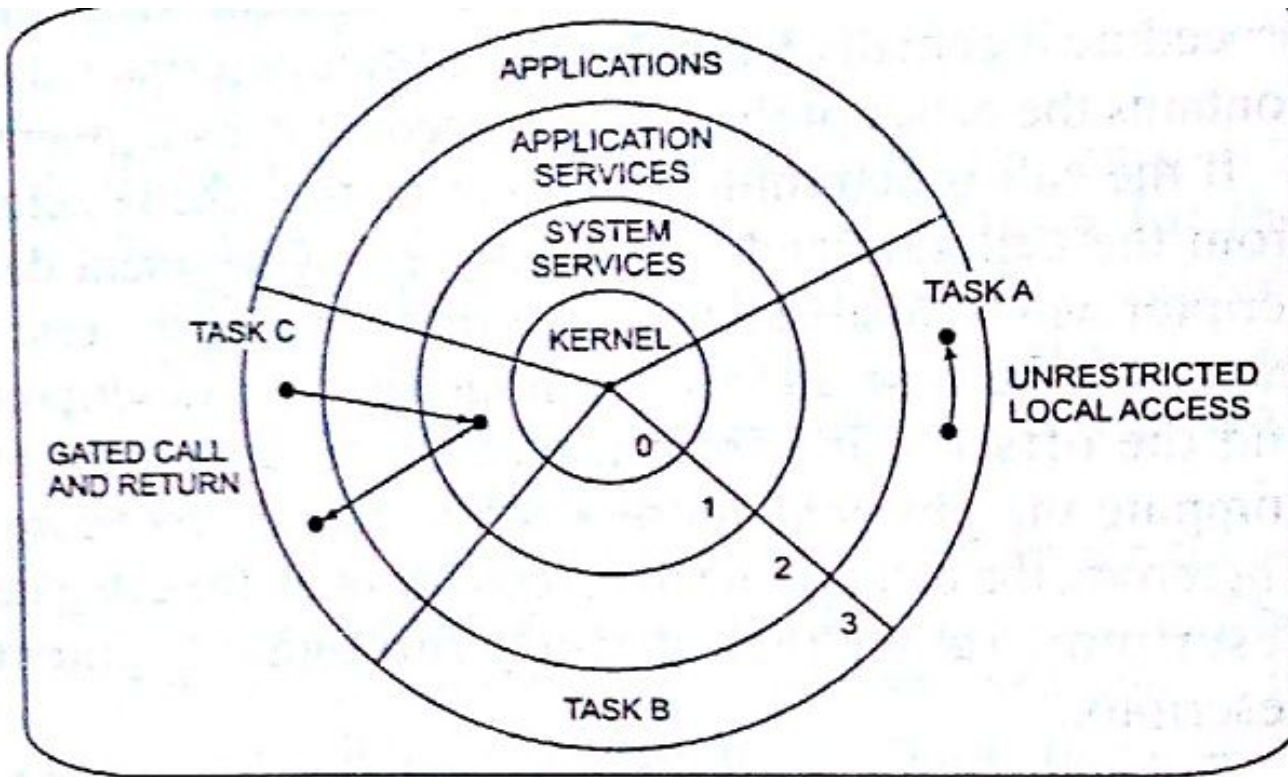


Fig. 15.24 Diagram showing how a 386 system can be set up with four privilege levels.

- Normally, operating system tasks accessed by all the programs run at level 0,1 and 2.
- The kernel – Process management, Memory management etc. - LEVEL 0.
- System services – LEVEL 1.
- Other applications services – LEVEL 2.
- User task – LEVEL 3.
- Level 0, 1 and 2 are known as **supervisor mode**.
- Level 3 is known as **user mode**.

DPL,CPL,RPL

- **DPL (Descriptor Privilege Level):** field of segment descriptor defines the privilege level of an entire segment.
- At any movement, the processor is fetching instructions from a code segment with a particular DPL.
- **The CPL (Current Privilege Level)** is set to DPL from where the processor is fetching a instruction.
- The DPL of the segment (CS, DS, SS) is defined at the time of creating the segment descriptor.
- CPL changes as the process moves from one code segment to other code segment.
- Lower 2-bits of a selector are defined as **RPL (Requested Privilege Level)**.
- When a selector is loaded for a code segment, the RPL is set to the CPL, that is DPL of the target code segment.

Data segment Access

- A currently executing task may need to transfer the control to the other code segment at the same or a different privilege level.
- The task may need to use data from the data segment or access a stack segment at different privilege levels. 80386 protection rules are strictly followed for such accesses of stack/data/code.
- **How task makes access to the data segment under the protection rules?**
- The code can access the data at the current privilege level or a less privilege level.
- E.g. If CPL of code is 2, then it can access the data at privilege level 2 and 3.
- Whenever a data segment register (DS,ES,FS or GS) is loaded, the DPL of the target data segment is compared to the CPL and RPL. It should met following condition.

$$\text{Target DPL} \geq \text{Maximum (CPL , RPL)}$$

- Performing security check using DPL,CPL and RPL is know as **privilege validation**.

Stack segment Access

- **How task makes access to the stack segment under the protection rules?**
- More strict for stack segment.
- The code can access the stack only at the same privilege level.
- E.g. If the code is running at CPL 2, then it can access the stack only at privilege level 2.
- Whenever a stack segment register (SS) is loaded, the following privilege validation is performed:

$$\text{Target DPL} = \text{CPL}$$

- If it is satisfied then access to stack segment is allowed otherwise the general protection fault is raised.
(Exception is raised)

Code segment Access

- **How task makes access to the another code segment under the protection rules?**
1. Near call/ near JMP instruction. (Intra segment /same segment)
 - No privilege validation is required as the same code segment is used.
 - Only the limit checked is performed to check whether the target instruction is within the addressable range of the segment.
 2. Far call/far JMP (Inter-segment /different segment)
 - The 80386 allows the transfer of the control only to the target code segment at the same privilege level.
 - DPL of the target code segment is equal to CPL. i.e **Target DPL = CPL**
 - Same is applicable for far RET instruction.

How far JUMP/CALL effected by privilege level?

- The FAR jump or FAR Call instruction can transfer the control directly by specifying the location within another code segment with the same privilege level as the CPL.
- **What if the target code segment at different privilege level?**
- There are two ways:
 1. Conforming code segment
 2. CALL gate

1. Confirming code segment

- The code segment can transfer a control to a target code segment with a **lower privilege level** provided the target code segment is defined as **confirming code segment**.
- It means if the target DPL is less privileged than the CPL and the target code segment is defined as a confirming code segment then the control transfer is possible.
- Type field of descriptor is 6 and 7 are confirming code segment.
- A confirming code segment is a code segment which does not have its own inherent privilege level.
- If a code segment with privilege level 2 makes a call to the confirming code segment, then it runs at CPL 2.
- The same confirming code segment at other time runs at CPL 1 if a code segment with privilege level 1 makes jump into it.
- Confirming code segments are very useful for developing the shared code, that is, common library routines which are normally called by the code running at different levels.
- We can not use privilege instructions into the confirming code segment.

2. CALL Gates

- The control transfer is possible directly to a target code segment if its privilege level is equal to the CPL.
- It is possible for a target code segment with a lower privilege level if it is a confirming code segment.
- The third possibility is to transfer the control to a target code segment with a higher privilege.
- This is possible through the special interface called gate.
- Four types of gates: call, trap, interrupt and task.
- A call gate implements the transfer control mechanism to a code segment at higher privilege level.
- A call gate is a system descriptor (S=0) with type value 1100.
- It acts as an interface between the code segments working at different privilege levels.

Format of CALL Gate

| | | | | | |
|----------------------------|-----|----------|--------------|-----|--------------------|
| 15870 | | | | | |
| Destination offset(31-16) | | | | | |
| P | DPL | (0) S | TYPE 1100 | 000 | Word Count (WC) |
| Destination Selector(15-0) | | | | | |
| Destination offset(15-0) | | | | | |

| S | TYPE in binary | TYPE in HEX | System - Descriptor information |
|---|----------------|-------------|---------------------------------|
| 0 | 0 0 0 0 | 0 | |
| 0 | 0 0 0 1 | 1 | |
| 0 | 0 0 1 0 | 2 | Descriptor for LDT |
| 0 | 0 0 1 1 | 3 | |
| 0 | 0 1 0 0 | 4 | |
| 0 | 0 1 0 1 | 5 | |
| 0 | 0 1 1 0 | 6 | |
| 0 | 0 1 1 1 | 7 | |
| 0 | 1 0 0 0 | 8 | |
| 0 | 1 0 0 1 | 9 | |
| 0 | 1 0 1 0 | A | |
| 0 | 1 0 1 1 | B | |
| 0 | 1 1 0 0 | C | CALL gate |
| 0 | 1 1 0 1 | D | |
| 0 | 1 1 1 0 | E | |
| 0 | 1 1 1 1 | F | |

CALL Gate

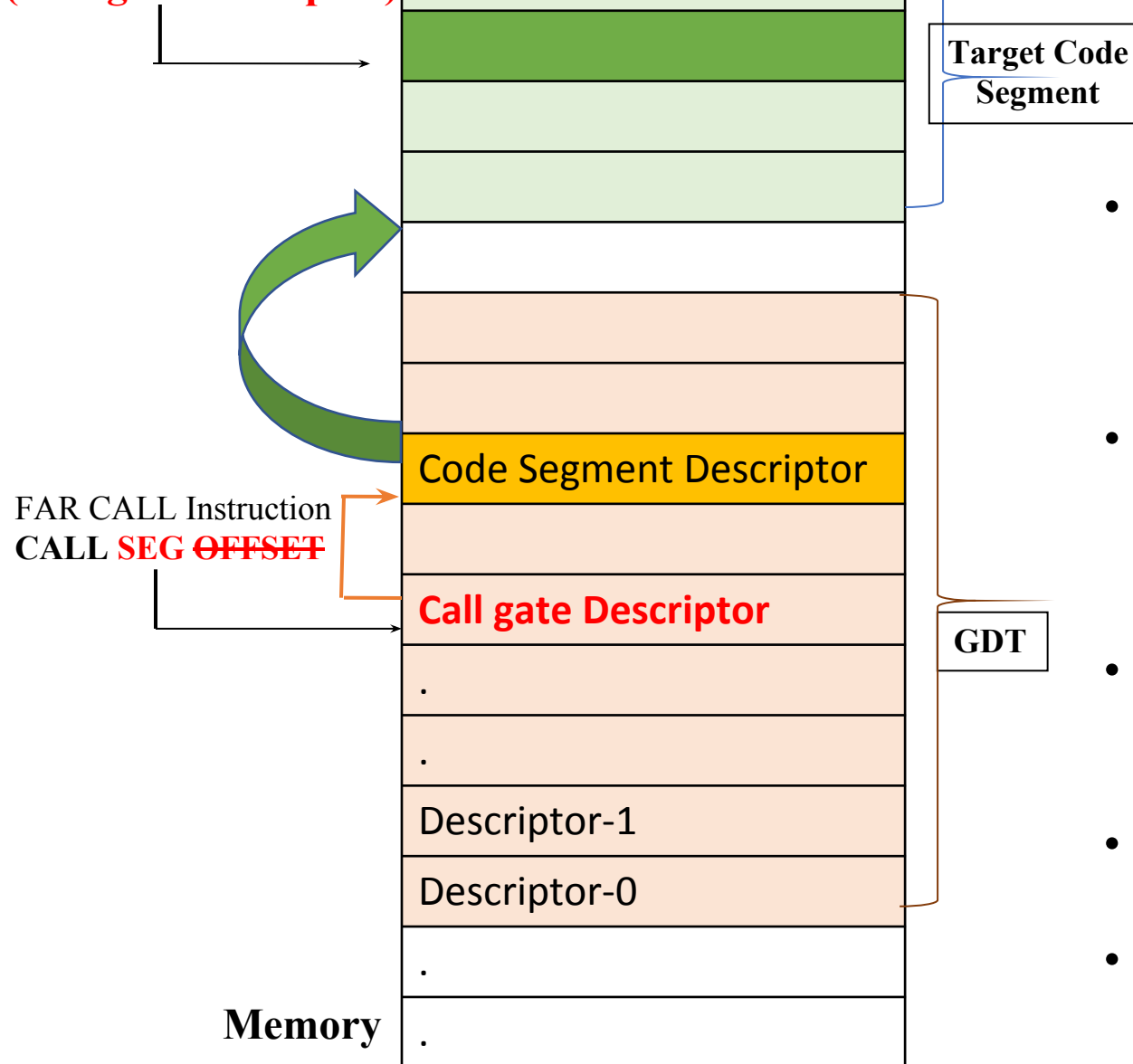
| | | | | | |
|----------------------------|-----|----------|--------------|-----|-------------------------------------|
| 15 | | 8 7 | | 0 | |
| Destination offset(31-16) | | | | | |
| P | DPL | (0) S | TYPE 1100 | 000 | Word Count (WC) (4 – bits) |
| Destination Selector(15-0) | | | | | |
| Destination offset(15-0) | | | | | |

- Call gate does not define any memory space.
- It provides the entry point to the code segment at higher privilege level by defining the virtual address of the higher privilege level code segment.
- Call Gate descriptor is put into the GDT or LDT.
- **Destination selector** is used to identify the descriptor of the target code segment. (Out of 16 bits, 13 –bits are used as index in GDT or LDT to get target code segment descriptor).

- **Destination offset** provides the offset if the instruction into the target code segment from where execution is to be resumed.
- If we have transferred the control to higher privilege level then we can not access old stack
- **WORD COUNT:** number of double word parameters to be copied from the OLD stack segment to the new stack segment.

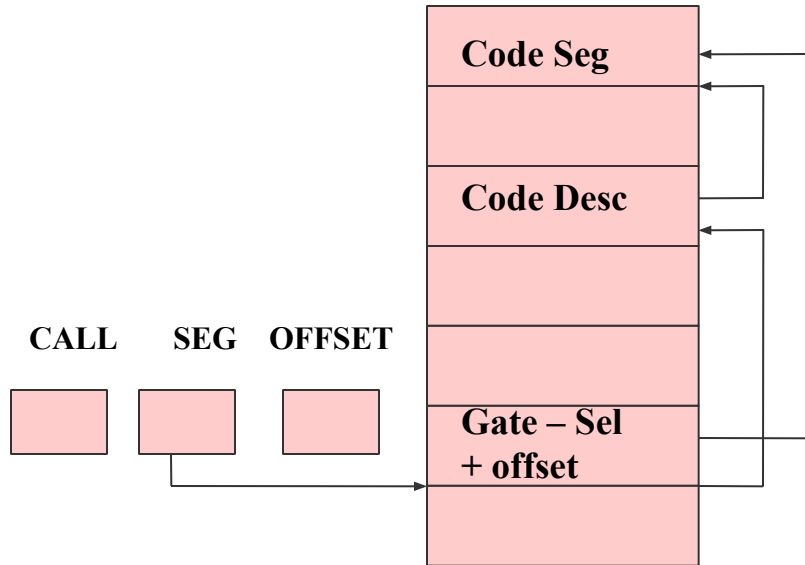
Calling With the call gate

(EIP) – 32 bit
Destination offset
(Call gate descriptor)

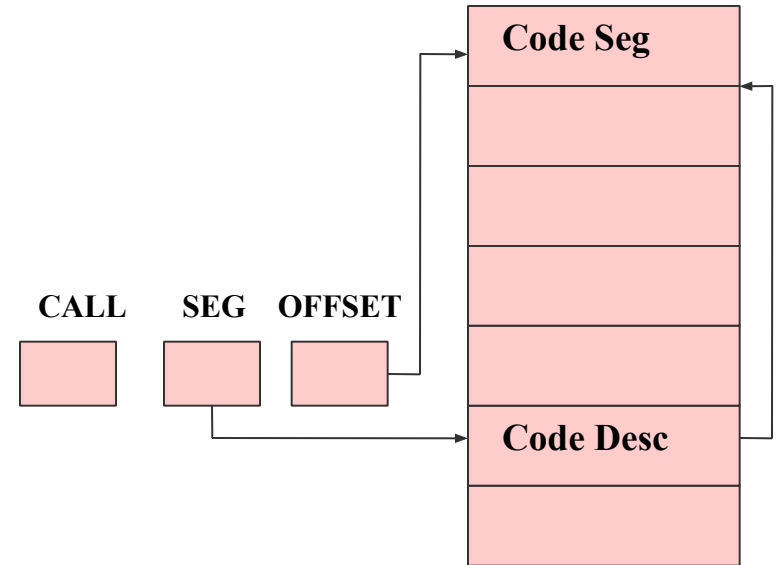


- If the selector/segment in the **FAR CALL** instruction **points to the call gate** then the transfer happens as below.
- The **destination selector field** of the call gate gives the **pointer to the descriptor of the target code segment** having higher privilege.
- The **CS** is loaded with the destination selector value from call gate and pointing to the descriptor of the target code segment.
- The destination offset field from call gate gives the offset (EIP) of the entry point in the new code segment.
- OFSSET provided by FAR call is ignored.
- Thus call gate acts as an intermediately between two code segments at different privilege levels.

Calling Higher privileged code



Correct



Incorrect

CALL Gate

- When a call gate is used to transfer the control, the privilege fields involved are the CPL (or RPL) of the current code segment, the DPL of the call gate and the DPL of the target code segment.
- Rule is as below:
Target DPL \leq maximum (CPL, RPL) \leq call gate DPL
- e.g. if CPL of the current code segment is 1, then it can use the call gate with DPL 1,2 or 3 and transfer the control to the target code segment with DPL 0 or 1.
- NOTE: If calling code segment (let's say level 1) and target code segment (level 1) are at same privilege level then no need to use call gate.
- When we use call gate the control is transferred to **higher privilege level so now we can not use the OLD stack** because stack can be used only from same privilege level.
- The TSS (Task State Segment) of the task maintains stacks for each privilege level.
- So new stack with more privilege is to be used now.
 - Processor **pushes** the **OLD SS, ESP** and number of double word parameter equal to **WC field** , **OLD CS** and **EIP** on new stack.
- After Completing procedure at higher level, the **FAR RET** instruction **POPS** all the values from the stack and resumes from **the next instruction after FAR CALL** instruction.

80386 Multitasking

Multitasking: The multitasking system allow execution of multiple tasks from one or more users at the same time using a concept called time sharing.

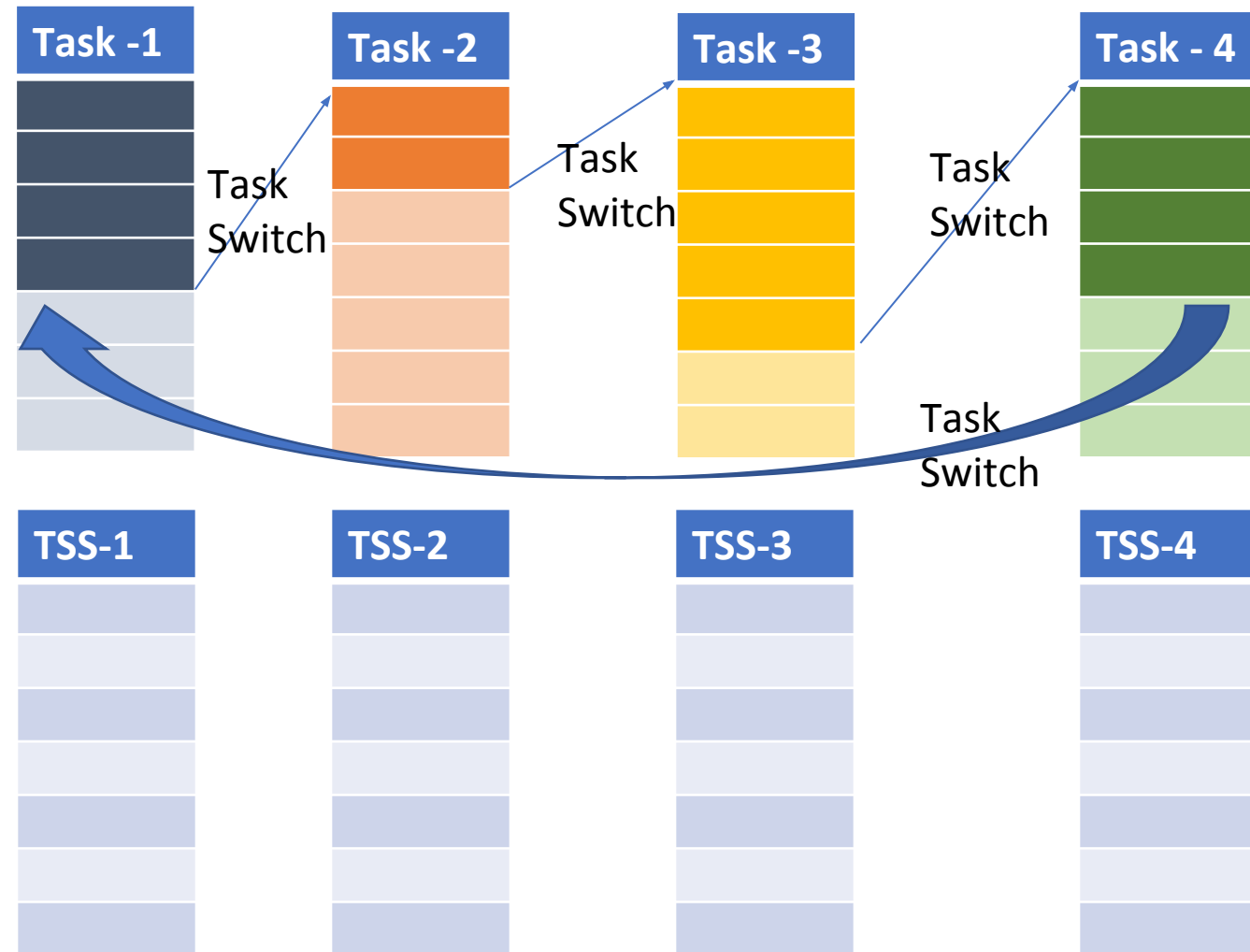
- The processor runs a task for a period of a time slice and switches to the next task waiting for its turn. After execution of last task processor comes again to the first task.
- This make it mandatory to save the **state of the task** somehow so that it can be started next time with the last state, not from the beginning, and thus the task moves forward towards finishing.
- The 80386 multitasking environment provides all the support for executing multiple task without any problem.
- The concept of multitasking looks very simple raises many issues, solving which makes the system complex

Context Switching

- The processor needs to remember the **correct point to resume** the task when task switches.
- Not only remember the correct point but also remember the **last state** including **registers, flags** and **other state information** when the task was suspended.
- The entire state of a task including the execution point, register flags and other necessary status information is called **context of the task**.
- Whenever the time slice for the current task is over, the context of the task is saved and the context of the next task to be executed is restored to resume it correctly.
- The process of saving the context of an outgoing task and loading the context of an incoming task is known as **context switching** or **task switching**.

Performing A task switching

- **How does 80386 performs a task switch?**
- Normally a task switch happens as result of executing the FAR JMP or FAR CALL instruction.
- Another way is the occurrence of the **interrupt or exception**.
- CALL gate is used to transfer control to the code at a higher privilege level using FAR CALL instruction.
- The task switch by executing FAR JMP or FAR CALL happens almost similar manner.
- **The only difference is that in case of task switch two tasks are totally independent and have no inherent relation (generally).**



Multi Tasking

- To provide efficient, protected multitasking, the 80386 employs several special data structures.
- It does not use special instructions to control multitasking; instead, it interprets ordinary control-transfer instructions differently when they refer to the special data structures.
- The registers and data structures that support multitasking are:
 1. Task state segment
 2. Task state segment descriptor
 3. Task register
 4. Task gate descriptor

Task State Segment

- The 80386 uses a special segment called **task state segment (TSS)** to save the context of a task.
- The TSS is described by a system segment descriptor called **TSS descriptor**.
- **The TSS descriptor can stored only in GDT.**
- The Value of TYPE field is 1001 which indicates the TSS is available.
- When task represented by TSS is executing value of TYPE field is 1011.

| | | | | | | | | | |
|--------------|-----|----------|-----------------|-------------|---|---|---|------------------|---|
| Base(31-24) | | | | G | D | X | U | Limit (19-16) | 6 |
| P | DPL | S (0) | TYPE 1 0 B 1 | Base(23-16) | | | | | 4 |
| Base (15-0) | | | | | | | | | 2 |
| Limit (15-0) | | | | | | | | | 0 |

The format of TSS descriptor

| S | TYPE in binary | TYPE in HEX | System - Descriptor information |
|---|----------------|-------------|---------------------------------|
| 0 | 0 0 1 0 | 2 | Descriptor for LDT |
| 0 | 1 0 0 1 | 9 | TSS is available |
| 0 | 1 0 1 0 | A | |
| 0 | 1 0 1 1 | B | TSS is busy (executing task) |
| 0 | 1 1 0 0 | C | CALL gate |

Task State Segment

- What is context of task?
- The 80386 needs minimum **104 bytes** to store the state of the task.
- This indicates that limit field in the TSS descriptor can never be less than the value **00067h**.
- The TSS stores all the general purpose registers including eight extended registers, extended instruction pointer, EFLAG and six segment register.
- Storing the **CR3** register in the TSS indicates that the **page directory and the page table** can also be changed from task to task.
- The **back link field** is used to store the selector of the previous task in a task chain specifically when the task are nested so that it can return to the calling task.

| 31 | 16 | 15 | 0 | |
|-----------------|----|-----------|---|-------|
| Bit_Map offeset | | - | T | 0064h |
| - | | LDT | | |
| - | | GS | | |
| - | | FS | | |
| - | | DS | | |
| - | | SS | | |
| - | | CS | | |
| - | | ES | | |
| EDI | | | | |
| ESI | | | | |
| EBP | | | | |
| ESP | | | | |
| EBX | | | | |
| EDX | | | | |
| ECX | | | | |
| EAX | | | | |
| EFLAG | | | | |
| EIP | | | | |
| CR3 | | | | |
| - | | SS2 | | |
| ESP2 | | | | |
| - | | SS1 | | |
| ESP1 | | | | |
| - | | SS0 | | 0008h |
| ESP0 | | | | 0004h |
| - | | Back Link | | 0000h |

Task State Segment

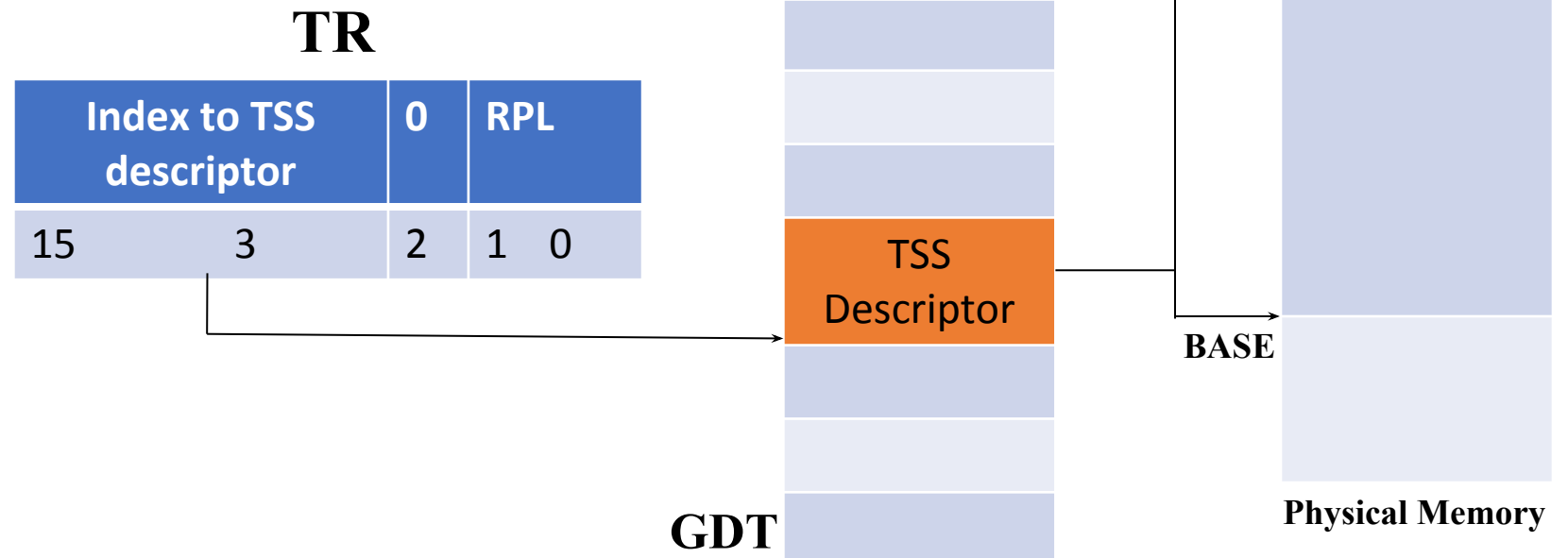
- The **Bit_map field** is used for the I/O permission.
- **T-bit** is used for debugging.
- The TSS also store the details of stack (SS and ESP) at privileged level 0, 1 and 2.

1. For each task one TSS is required to store its context.
2. TSS initialized only once when the task is created first time.
3. The SS0, ESP0, SS1, ESP1, SS2, ESP2, LDT, T and BIT_MAP fields remain the same across the time slice for a task and not updated each time.
4. The general purpose registers change every time and needs to be save for each task switch.

| 31 | 16 | 15 | 0 | |
|----------------|----|-----------|---|-------|
| Bit_Map offset | | - | T | 0064h |
| - | | LDT | | |
| - | | GS | | |
| - | | FS | | |
| - | | DS | | |
| - | | SS | | |
| - | | CS | | |
| - | | ES | | |
| EDI | | | | |
| ESI | | | | |
| EBP | | | | |
| ESP | | | | |
| EBX | | | | |
| EDX | | | | |
| ECX | | | | |
| EAX | | | | |
| EFLAG | | | | |
| EIP | | | | |
| CR3 | | | | |
| - | | SS2 | | |
| ESP2 | | | | |
| - | | SS1 | | |
| ESP1 | | | | |
| - | | SS0 | | 0008h |
| ESP0 | | | | 0004h |
| - | | Back Link | | 0000h |

TR Register

- The task register plays an important role in task switching
- The TR always hold the selector value of TSS descriptor of the current task
- The TR is updated automatically when a task switch happens to keep it pointing to the current task.
- The instructions LTR and STR are used to modify and read the visible portion of the task register. (LTR is privileged instructions).



Task Switching

- Let us see how the execution of the FAR CALL or FAR JMP instruction causes a task switch.
- One of the operand in the FAR JMP or FAR CALL instruction is selector value.
- The 80386 provides two different ways for a task switch,

1. Direct Method

- By directly refereeing to [TSS descriptor](#) using it selector value.

2. Task gate

1. Direct Method of task switching

- In this method the task switching instruction (FAR JMP and FAR CALL) contains directly the selector value of TSS descriptor of the new task to which the context switch is going to happen.
- The CPL or RPL of the current task, whichever is higher, must be privileged enough to refer to the TSS, that is, it must have equal or higher privilege than that of the TSS.
- The violation of this rule result in a general protection fault.
- **Maximum (CPL,RPL) <= TSS DPL (Numerically)**

2. Task switching using Task Gate

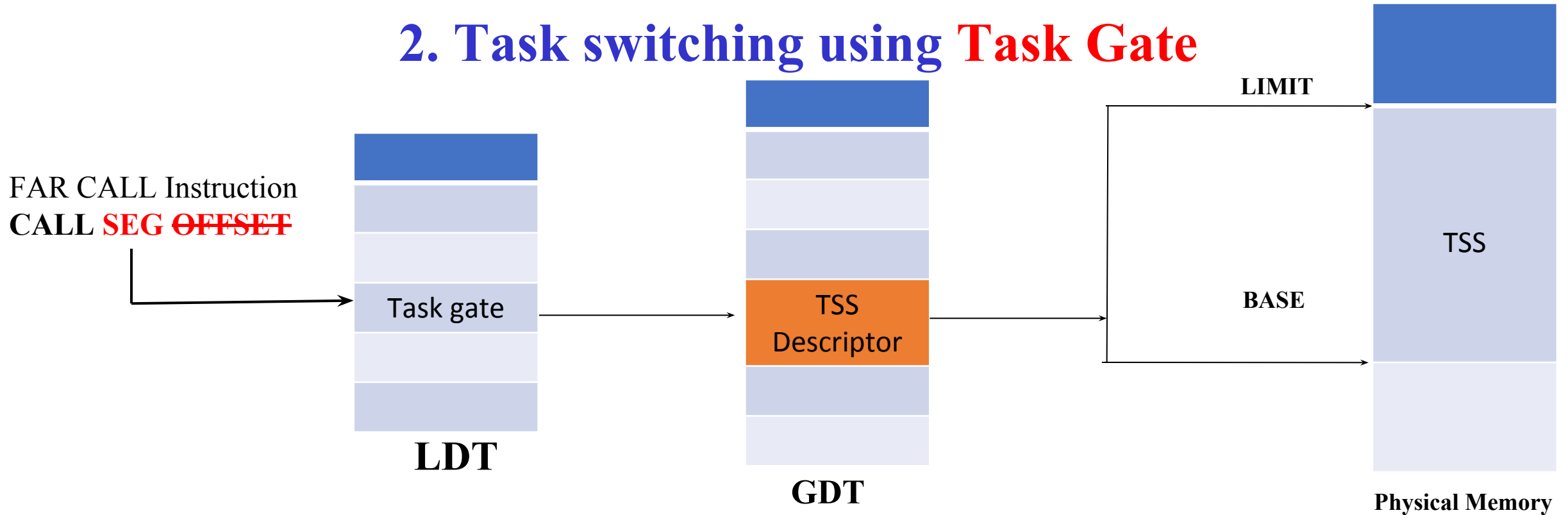
Format of a Task Gate

| | | | |
|----------------------------|-----|----------|--------------|
| 15 | 8 | 7 | 0 |
| - | | | |
| P | DPL | (0) S | TYPE 0101 |
| Destination Selector(15-0) | | | |
| - | | | |

| S | TYPE in binary | TYPE in HEX | System - Descriptor information |
|---|----------------|-------------|---------------------------------|
| 0 | 0 0 1 0 | 2 | Descriptor for LDT |
| 0 | 0 1 0 1 | 5 | Task Gate |
| 0 | 1 0 0 1 | 9 | TSS is available |
| 0 | 1 0 1 1 | B | TSS is busy (executing task) |
| 0 | 1 1 0 0 | C | CALL gate |

- In this method the task switching instruction (FAR JMP and FAR CALL) contains the selector value of the task gate which is very similar to the call gate.
- It is a system segment descriptor with **TYPE=0101**.
- The task gate contains only the selector field which points to the TSS descriptor of the task to which task switching is to happen.
- The violation of the following rule result in a general protection fault.
Maximum (CPL,RPL) <= Task gate DPL (Numerically)

2. Task switching using Task Gate



- If the condition of privilege level is satisfied, task switch operation is performed by saving the context of the current (outgoing) task, loading the context of the new (incoming) task and updating the TR with the new selector value.
- Both methods are same almost same only one difference is there.
- **TSS descriptor is stored only in GDT**, it means any task having sufficient privilege level can cause the switch to a task represented by the TSS in **direct Method**.
- In 2nd method we create **task gate in LDTs** of the task. In this case suppose you make DPL of TSS 0 so that the task at **privilege level 0** and the **task having the task gate in its LDTs** can switch to a task represented by the TSS.

Nested Task (Task Linking)

- If task switch happens due to execution of a **FAR CALL** instruction or an exception or interrupt, the new task or exception runs as the nested task within the current task.
- The nesting of more level creates a **task chain**.
- The 80386 set the **NT flag** in the EFLAG register if the new task is nested inside the current task.
- The TSS contains the **16 bit Back Link field** to store the selector of the TSS of the caller task if the task is nested task.
- That means if the NT=1 the back link field is valid and it store the selector value of the TSS of the caller task.
- The NT=1 indicates the current task is child of the parent task having parent selector in the Back Link field.

| Affected Field | Effect of JMP instruction | Effect of CALL instruction | Effect of IRET instruction |
|----------------------------------|---------------------------|------------------------------|----------------------------|
| Busy bit of Incoming Task | Set, must be 0 before | Set, must be 0 before | Unchanged, Must be set |
| Busy bit of outgoing Task | Cleared | Unchanged (Already set) | Cleared |
| | | | |
| NT bit of incoming Task | Cleared | Set | Unchanged |
| NT bit of outgoing Task | Unchanged | Unchanged | Cleared |
| | | | |
| Back-link field of incoming task | Unchanged | Set to outgoing TSS Selector | Unchanged |
| Back-link field of outgoing task | Unchanged | Unchanged | Unchanged |

Task Scheduling

- Multitasking operating systems allow the multiple tasks to run at same time using the time sharing concept.
- When a time slice is complete, task switching is performed.
- Who will perform the task switching?
- Which pending task is taken next?
- The operating system contains the software module called scheduler.
- The scheduler has well defined policy to select the next task for the execution.
- You will learn about it in detail in Operating System subject.

I/O Privilege or I/O protection

- When 80386 operating in protected mode it has **two mechanism for protecting I/O ports**.
- **The first mechanism involves the I/O privilege level bits (2-bits) in the 80386 EFLAGS.**
- Only the OS or the procedure which is running at higher privilege level can set these IOPL bits.
- In order to execute Following instructions **the CPL of the task is must be same or lower number then IOPL. (CPL <= IOPL)**
- IN -- Input, INS -- Input String, OUT -- Output, OUTS -- Output String, CLI -- Clear Interrupt-Enable Flag, STI -- Set Interrupt-Enable
- If it does not satisfy this then privilege level exception is generated.

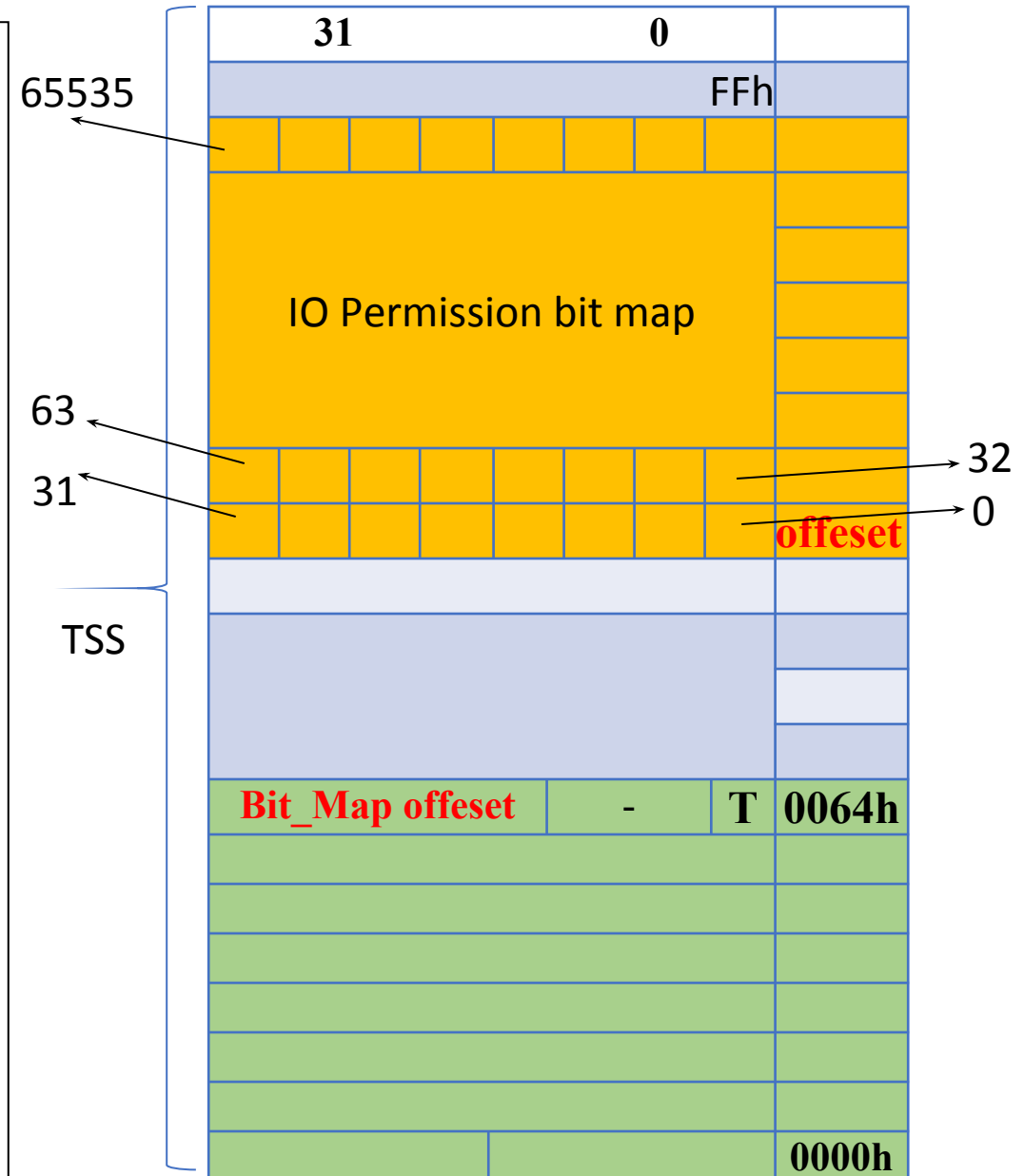
I/O permission bit map

- **Second mechanism** to protect I/O ports from unauthorized access is an I/O permission **Bit map**.
- The minimum size of the TSS for a task is 104bytes.
- However the TSS is like any other segment and can be as large as 4GB.
- We can Extend the TSS beyond the 104bytes to store some useful information.
- One such optional information is the **I/O permission bit map** for the task.
- If the **I/O permission bit map** is defined for the task, we can execute the I/O instructions even though the standard privileged check fails.
- The field at the offset **0066h in the TSS** is the **bit map offset**.
- It store the offset at which I/O permission bit map starts.

| 31 | 16 | 15 | 0 | |
|----------------|----|-----------|---|-------|
| Bit_Map offset | | - | T | 0064h |
| - | | LDT | | |
| - | | GS | | |
| - | | FS | | |
| - | | DS | | |
| - | | SS | | |
| - | | CS | | |
| - | | ES | | |
| EDI | | | | |
| ESI | | | | |
| EBP | | | | |
| ESP | | | | |
| EBX | | | | |
| EDX | | | | |
| ECX | | | | |
| EAX | | | | |
| EFLAG | | | | |
| EIP | | | | |
| CR3 | | | | |
| - | | SS2 | | |
| ESP2 | | | | |
| - | | SS1 | | |
| ESP1 | | | | |
| - | | SS0 | | 0008h |
| ESP0 | | | | 0004h |
| - | | Back Link | | 0000h |

I/O permission bit map

- The I/O permission bit map is sequence of bits, one for each byte I/O port address.
- In 80386 the I/O port address can be 16-bit so $2^{16} = 64\text{KB}$ port address space.
- Each bit in the I/O permission bit map correspond to one byte I/O port address.
- 0 means granted and 1 means access not allowed.
- For example, the least significant bit of the first byte is the I/O permission for the byte 0, next for address 1 and so on.
- Maximum size of bit map is $64\text{KB}/8 = 8\text{KB}$.
- It's terminating using FFh byte.
- This bit map should be included in size calculation of TSS. (LIMIT field)



80386 Interrupt

- The 8086 categorizes interrupts as hardware interrupts, **exceptions** and software interrupts.
 - The 80386 interrupt and interrupt handling in the protected mode is also similar to 8086 processor.
 - The 80386 exceptions are nothing but the interrupts in the 80386 system including **hardware interrupts** (maskable and non maskable), **software exceptions** and **software interrupts** (INT n).
1. The source of the **hardware interrupts** are the signals from the external devices received either on INTR or on NMI pins. The interrupt received on INTR are maskable, where as on NMI are non maskable.
 2. The **software interrupt** are caused by executing the INT n instruction, where n is the interrupt number or vector number and ranges from 0 to 255.
 3. The **software exceptions** occur due to the error condition like privilege violation, accessing address outside the LIMIT.

80386 Exception

There are three types of software exception in 80386.

1. TRAP

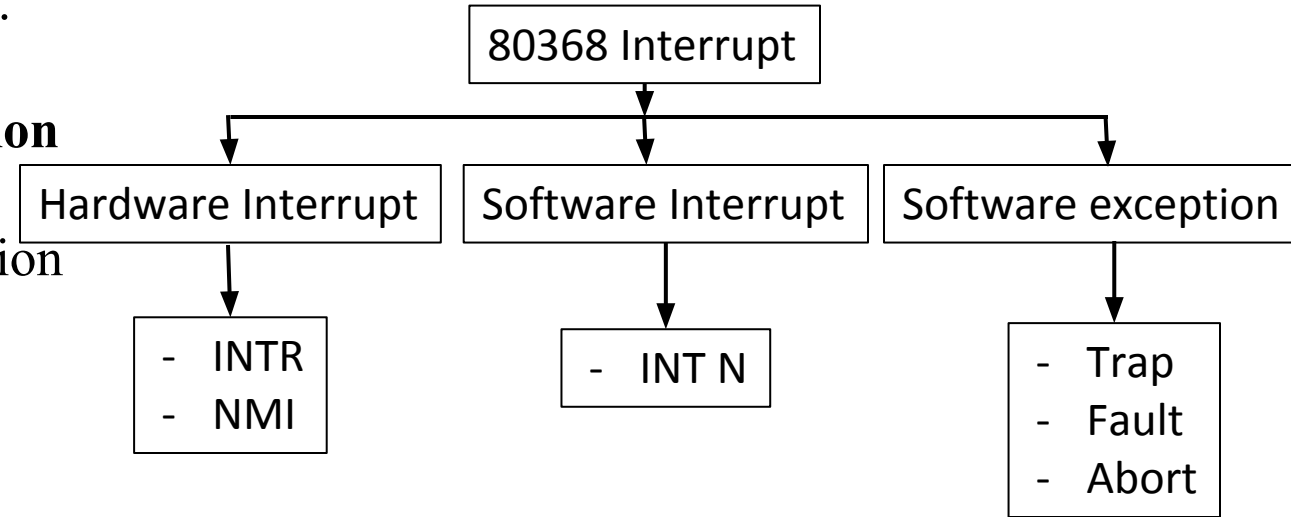
- The software traps are caught **after an exception occurs**.
- The execution is resumed from the next instruction after completing the exception handler.
- They occurs in **debugging or breakpoints**.

2. Fault

- The software faults are caught **before they occur**.
- Execution is resumed from the instruction that caused the fault.
- **Privilege violation and segment limit overrun are example of faults.**

3. Abort

- The software abort are caught after an error occurs.
- The program causing abort is terminated.



| Vector Number | category |
|---------------|---------------------------------------|
| 0-16 | The 80386 internal interrupts |
| 17-31 | Reserved |
| 32-255 | Maskable interrupts and INT n vectors |

The 80386 protected mode exception

Well know exception

| Interrupt No. | Description |
|---------------|-----------------------------|
| 0 | Divide Error |
| 1 | Debug Exception |
| 3 | Break point |
| 4 | Overflow |
| 5 | Bounds Check |
| 6 | Invalid opcode |
| 7 | Coprocessor not available |
| 8 | System error |
| 9 | Coprocessor segment Overrun |
| 10 | Invalid TSS |
| 11 | Segment not present |
| 13 | General Protection |
| 14 | Page Fault |

The IDT and Exception handling

- We know that 80386 protected mode uses three types of descriptor tables: GDT, LDT and **IDT**.
- The **IDT** is like the GDT and 80386 needs **only one**.
- The IDT stores the descriptors which act as an entry point (gate) for the corresponding exception handler (ISRs).
- The IDT stores the 256 gate descriptor, one for each vector, as the 80386 recognize a total 256 exception in the protected mode identified by vectors 0 to 255.
- Consider 8 bytes for each descriptor, the size of IDT is $256 * 8 = 2\text{KB}$. That means limit of the IDT is 07FFh.
- In that sense IDT looks very similar to vector table.
- **The only difference is that it store the descriptor gates rather than directly the addresses.**

IDT – Interrupt Descriptor Table

- When an exception occurs, the 80386 gets the vector number which act as an index into the IDT and the gate descriptor at that index into the IDT is used to transfer the control to the respective handler.
- The IDTR register store the base and limit of the IDT.
- An interrupt through a interrupt gate change IF but an interrupt through an trap gate does not resets IF.
- The IDT stores three different types of descriptor
 1. Interrupt Gate
 2. Task gate
 3. Trap Gate

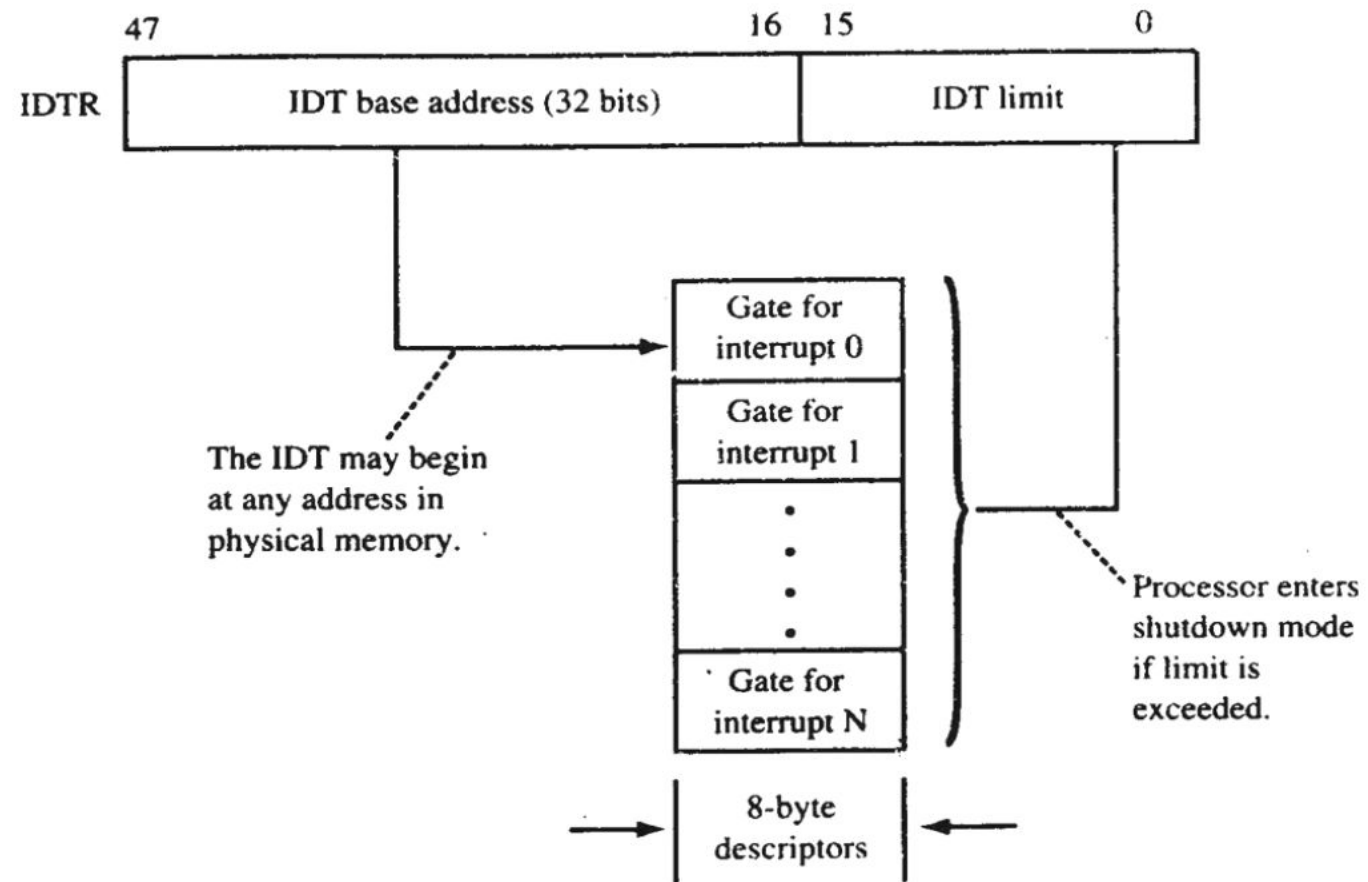


Figure: Using IDTR to access IDT

The format of a Interrupt Gate & Trap Gate Descriptor

Interrupt Gate

| | | | |
|--|-----|----------|--------------|
| 15 8 7 0 | | | |
| Destination offset(31-16) | | | |
| P | DPL | (0) S | TYPE 1110 |
| Reserved | | | |
| Destination Selector(15-0) | | | |
| Destination offset(15-0) | | | |

Trap Gate

| | | | |
|--|-----|----------|--------------|
| 15 8 7 0 | | | |
| Destination offset(31-16) | | | |
| P | DPL | (0) S | TYPE 1111 |
| Reserved | | | |
| Destination Selector(15-0) | | | |
| Destination offset(15-0) | | | |

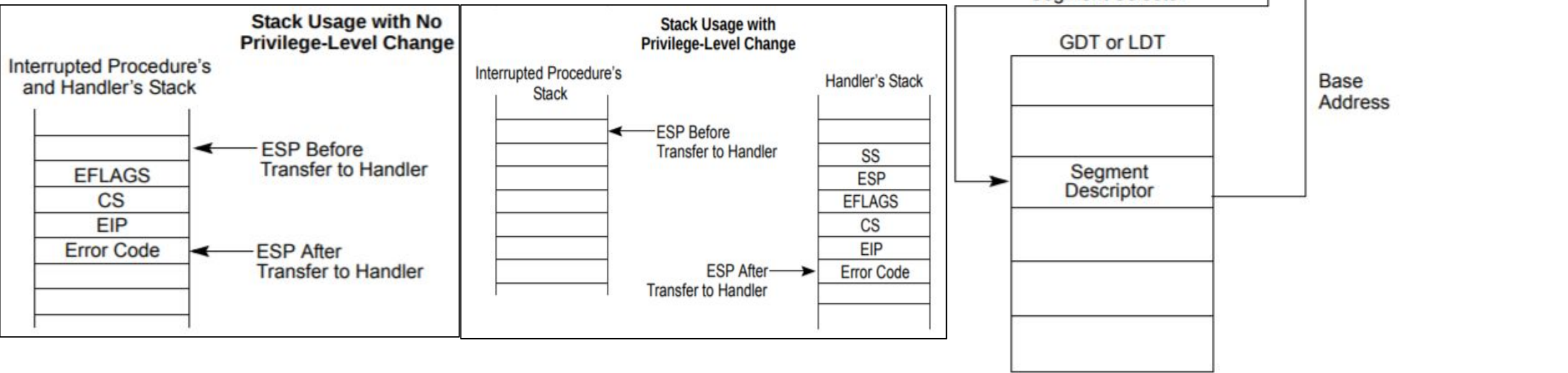
Task Gate

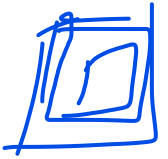
| | | | |
|--|-----|----------|--------------|
| 15 8 7 0 | | | |
| - | | | |
| P | DPL | (0) S | TYPE 0101 |
| - | | | |
| Destination Selector(15-0) | | | |
| - | | | |

| S | TYPE in binary | TYPE in HEX | System - Descriptor information |
|---|----------------|-------------|---------------------------------|
| 0 | 0 0 1 0 | 2 | Descriptor for LDT |
| 0 | 0 1 0 1 | 5 | Task Gate |
| 0 | 1 0 0 1 | 9 | TSS is available |
| 0 | 1 0 1 1 | B | TSS is busy (executing task) |
| 0 | 1 1 0 0 | C | CALL Gate |
| 0 | 1 1 1 0 | D | Interrupt Gate |
| 0 | 1 1 1 1 | F | Trap Gate |

Accessing an exception code using an interrupt gate

- The selector and offset in the gate are used to transfer control to the code segment of the exception handler.
- Before transferring control, 80386 pushes the EFLAG, CS, and EIP if there is **no privilege change**.
- If there is privilege change it pushes SS, ESP, EFLAG, CS and EIP.
- Difference between trap and interrupt gate is that the interrupt gate clears the IF before executing ISR while TRAP gate does not.





Example - 6

- If an interrupt **type** $(16)_d$ comes and **IDTR=0000 1000 008F**,
 1. Show that it will point to a valid interrupt gate descriptor in IDT.
The Interrupt Gate descriptor is given as beside.
 2. Prove that it is a valid 80386 interrupt gate descriptor.
 3. Now if **GDTR=0000 8000 007F**, selector of interrupt gate descriptor will generate any exception?
 4. If yes, do the correction in the selector of the above descriptor such that there would not be any exception.
 5. If no ,why?

| | | |
|-----|-----|---|
| 80h | 00h | 6 |
| EEd | 08h | 4 |
| 00h | 80h | 2 |
| 00h | 00h | 0 |

Interrupt Gate descriptor

- First check the Limit field base on the interrupt type number of IDT.
- For this example interrupt type= $(16)_d$ $16*8=(128)_{10} = 80 \text{ h (offset in IDT)}$
- $\text{Base} + \text{offset} \leq \text{LIMIT}$ i.e $00001000 + 0080 = 00001080 < 0000108F$
- Then check **S bit** and **TYPE field** to prove it valid or not.
- Then check the **selector value** from the given **gate descriptor** it should not be more than the limit specified in GDTR otherwise it will generate exception.

Example - 7

If an interrupt comes on IR0 pin of 8259 and upper five bits of ICW2 contains 00100_b and IDTR=0000 0000 007fh, will there be any exception? If any exception, correct the content of IDTR. The following is the interrupt Gate descriptor whose content is as follows :

| | | |
|-----|-----|---|
| 00h | 0Fh | 6 |
| EEh | 00h | 4 |
| 00h | 80h | 2 |
| 00h | 00h | 0 |

Prove that it is a valid interrupt gate descriptor.

IR0 pin of 8259 will set ICW2 register with interrupt type number. upper 5 bits, lower 3 bits are always 0 for ICW2.



ICW2

Paging

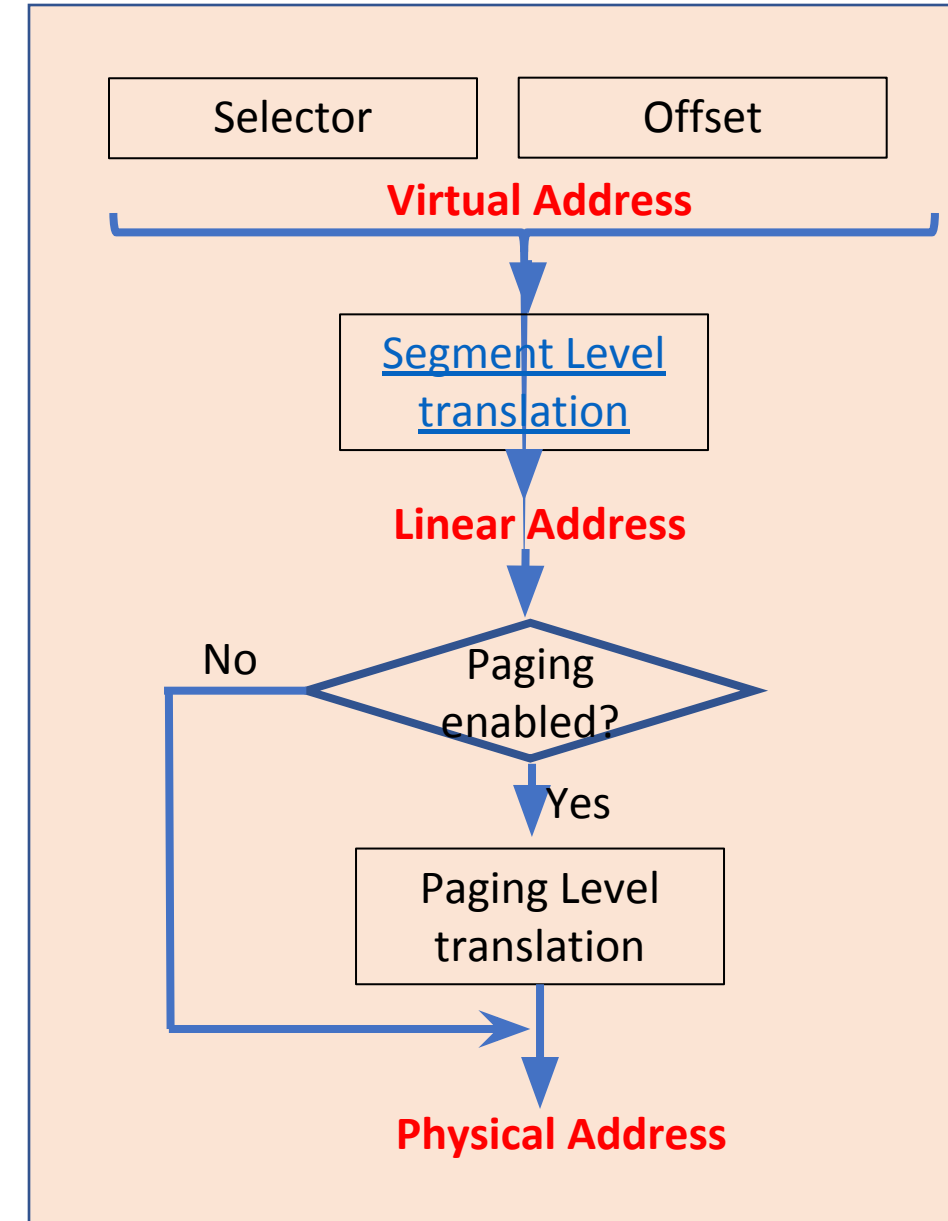
- Paging is one of the memory management techniques used for **virtual memory multitasking** operating system.
- The segmentation scheme may divide the physical memory into a **variable size segments** but the **paging divides the memory into a fixed size pages**.
- The pages are just **fixed size portions** of the program module or data.
- Major Objectives of paging system are:
 - Address Translation
 - Protection
 - Demand paging

Paging

- The advantage of paging scheme is that the complete segment of a task need not be in the physical memory at any time.
- Only a few pages of the segments, which are required currently for the execution need to be available in the physical memory.
- Whenever the other pages of task are required for execution, they may be fetched from the secondary storage.
- The previous page which are executed, need not be available in the memory, and hence the space occupied by them may be relinquished for other tasks.
- Thus paging mechanism provides an effective technique to manage the physical memory for multitasking systems.

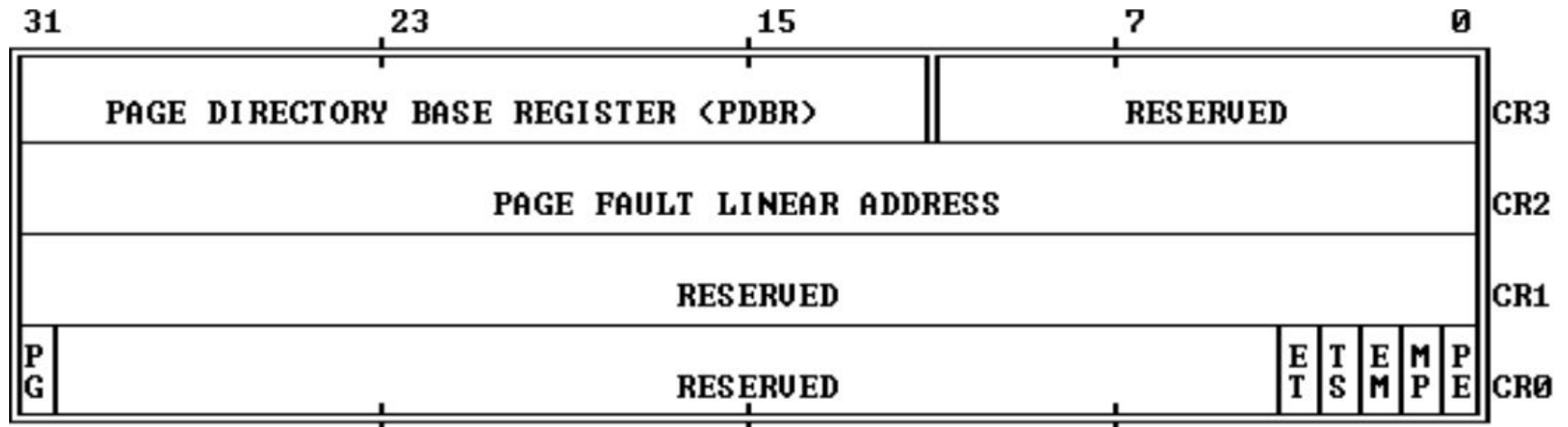
Virtual address to physical address translation

- Segments can be too large to be swapped in and swapped out of memory, so 80386 allows paging mechanism to be switched in after the segmentation unit.
- The paging unit divides segments into 4KB pages for swapped in and out from the memory.
- Segmentation translates the virtual address into the linear address.
- If paging is disable this linear address will become physical address.
- If paging is enable this linear address will be converted into physical address by paging unit.
- To convert Linear address in to physical paging unit uses two tables
 1. Page Directory Table
 2. Page Table



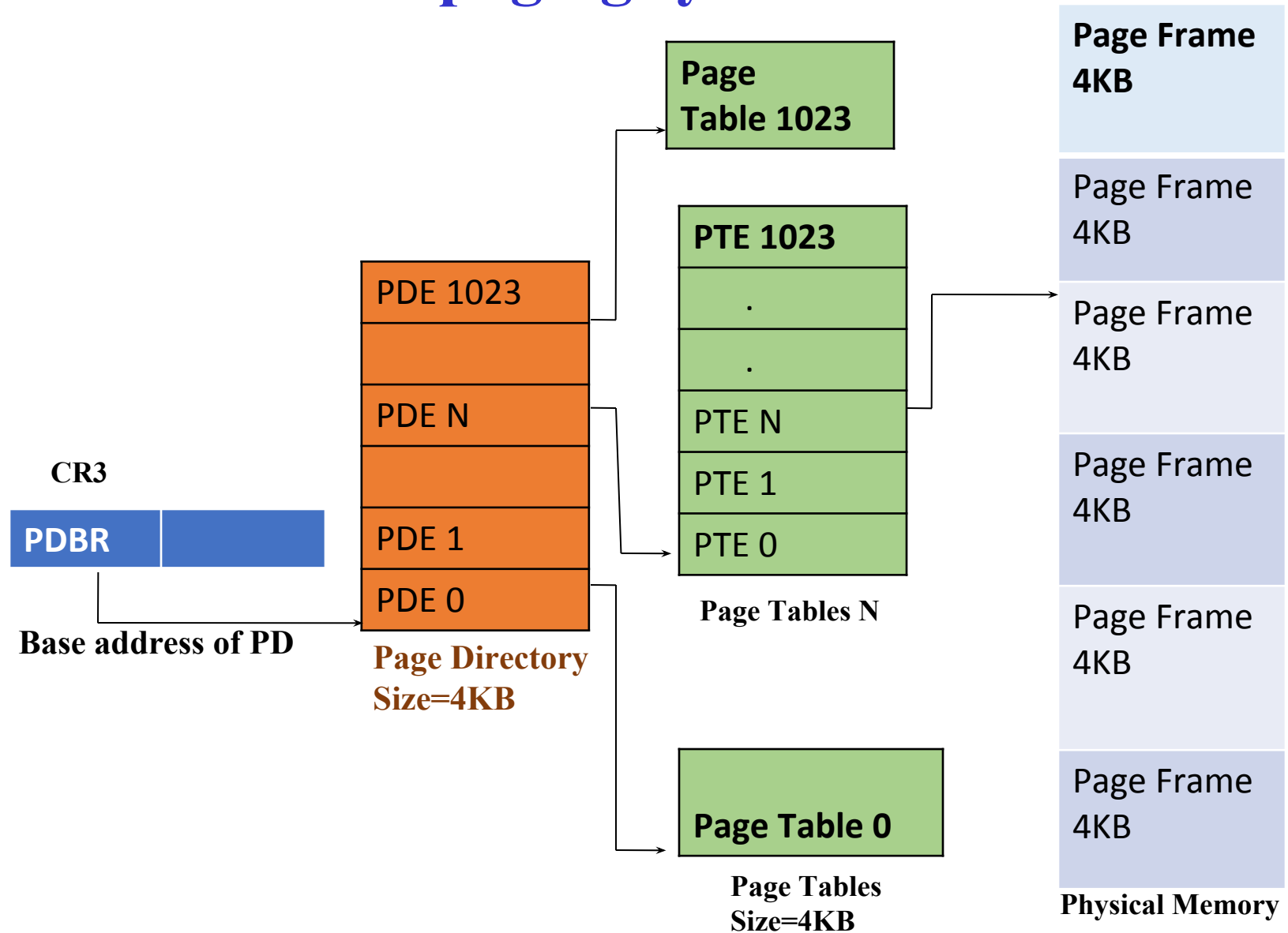
Use of Control Register in Paging

- The **CR3** is used as page directory physical base address register, to store the physical starting address of the page directory. The lower 12 bit of the CR3 are always zero to ensure the page size aligned directory.
- The control register **CR2** is used to store the 32-bit linear address at which the previous page fault was detected. (When a page is not present in main memory it needs to be load into main memory).
- PG bit of CR0 should be 1.



Overall structure of the paging system

- The paging unit of 80386 uses a two level table mechanism to convert a linear address provided by segmentation unit into physical addresses.
- The paging unit converts the complete map of a task into pages, each of size 4K. The task is further handled in terms of its page, rather than segments.
- The paging unit handles every task in terms of three components namely **page directory**, **page tables** and page itself.



Page Directory and Page Table



- **Page Directory**: This is at the most 4Kbytes in size.
- Each directory entry is of 4 bytes, thus a total of 1024 entries are allowed in a directory.
- The upper 10 bits of the linear address are used as an index to the corresponding page directory entry.
- The page directory entries point to page tables.
- **Page Tables**: Each page table is of 4Kbytes in size and contain a maximum of 1024 entries.
- The page table entries contain the starting address of the page and the statistical information about the page.
- The address bits A12- A21 are used to select the 1024 page table entries.
- The upper 20 bit page frame address is combined with the lower 12 bit of the linear address.

Page Directory Table & Page Directory Entry

| | | | | | | | | | | | | |
|--------------------|----|-------------|---|---|---|---|---|---|---|-----|-----|---|
| 31 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PAGE TABLE ADDRESS | | OS RESERVED | | 0 | 0 | 0 | A | 0 | 0 | U/S | R/W | P |

Page Directory Entry



Base address

of PD

Page Directory Size 4KB

| |
|----------|
| PDE 1023 |
| |
| PDE 1 |
| PDE 0 |

| Field | Position | Meaning |
|------------------------------|------------|---|
| Page Table address | Bits 31-12 | The most significant 20-bits of the starting physical address of a page table. The address is 4KB aligned so, the lower 12 bits are 0. E.g: If it is 12A3Dh , it means the page table starts from the physical address 12A3D000h . |
| User bits / OS reserved bits | Bits 11-9 | They are used by OS. |
| Accessed bit | Bit 5 | This bit is set whenever this PDE is used for address translation. |
| User/Supervisor bit | Bit 2 | If this bit is 0, pages covered by PDE are accessible only to supervisor mode. (i.e. Privilege level 0,1, and 2.) If it is 1, pages are accessible from all privilege levels. |
| Read/Write bit | Bit 1 | If U/S=0, this bit has no meaning, as the supervisor mode is never restricted. IF U/S=1, & R/W=0, only reading is allowed from all pages of this PDE. If U/S=1, & R/W=1, both the read and write operations are allowed. |
| Present bit | Bit 0 | If P=1, the page table pointed by this PDE is present in the memory. IF P=0, it is not present in the memory. |

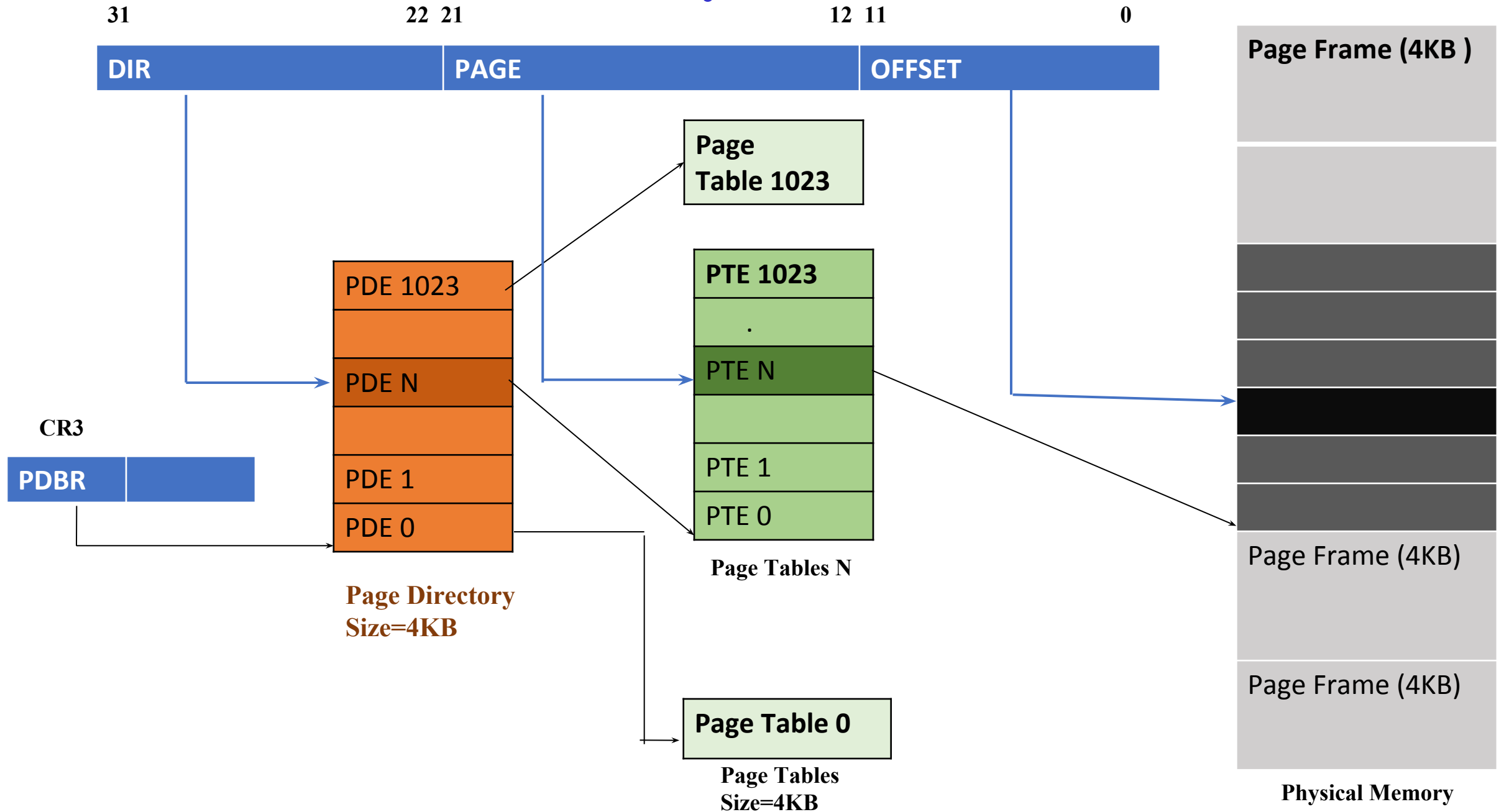
Page Table & Page Table Entry

| | | | | | | | | | | | | |
|--------------------|----|-------------|---|---|---|---|---|---|---|-----|-----|---|
| 31 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PAGE Frame ADDRESS | | OS RESERVED | | 0 | 0 | D | A | 0 | 0 | U/S | R/W | P |

Page Table Entry

| Field | Position | Meaning |
|------------------------------|------------|--|
| Page Frame address | Bits 31-12 | The most significant 20-bits of the starting physical address of a page frame. The address is 4KB aligned so, the lower 12 bits are 0. |
| User bits / OS reserved bits | Bits 11-9 | They are used by OS. |
| Dirty bit | Bit 6 | This bit is set to one whenever the write operation is performed on the page pointed by PTE. |
| Accessed bit | Bit 5 | This bit is set whenever this PTE is used for address translation. |
| User/Supervisor bit | Bit 2 | If this bit is 0, pages covered by PTE are accessible only to supervisor mode. (i.e. Privilege level 0,1, and 2.) If it is 1, pages are accessible from all privilege levels. |
| Read/Write bit | Bit 1 | If U/S=0, this bit has no meaning, as the supervisor mode is never restricted. If U/S=1, & R/W=0, only reading is allowed from all pages of this PTE. If U/S=1, & R/W=1, both the read and write operations are allowed. |
| Present bit | Bit 0 | If P=1, the page frame pointed by this PTE is present in the memory. If P=0, it is not present in the memory. |

Linear Address to Physical Address Translation



Example-8



Assuming PDBR=23455h, the page table address in the PDE-5 =45345h, the page frame address in the PTE-32 =67345h, What is the physical address for a linear address 014202CAh?

- (1) The Linear address 014202CAh (0000 0001 0100 0010 0000 0010 1100 1010) means
DIR = 0000 0001 01 = PDE index 5,
PAGE = 00 0010 0000 = PTE index 32
and OFFSET = 0010 1100 1010 = 2CAh.
- (2) The Physical Address for the page directory is 23455000h (PDBR with lower 12 bits 0s).
- (3) The PDE is located at the physical address 23455000h + 5*4=23455014h. It is used to refer the page table address from the PDE 5 which is 45345h.
- (4) The physical address of the page table is 45345000h.
- (5) The PTE is located at the physical address 45345000h+32*4=45345080h. It is used to refer the page frame address from the PTE 32 which is 67345h.
- (6) The physical address of the page frame is 67345000h.
- (7) The memory location referred to is 67345000h+2CAh=673452CAh

Translation Lookaside Buffer

- The translation from Linear address to Physical address we require two extra memory references, one for page directory entry and second for page table entry. This makes system too slow.
- 80386 uses Translation Lookaside Buffer – TLB. (on chip cache memory used for paging)
- TLB stores Last 32 conversions on the first-come-first-serve basis.
- Most of the time the conversion is available in the TLB store (TLB hit).
- If it's not available (TLB miss), the convert Linear address to Physical address by making two memory references and access PDE and PTE and store this conversion on TLB for future reference.
- If TLB is full, the least recently used entry is removed to make room for the last conversion.

| Linear Address | Physical Address |
|----------------|------------------|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

Translation Lookaside Buffer – TLB

Memory Model Of 80386

- The memory in the 386 and 486 system can be used in any of the below memory model.
- 1. Segment-only-model: Memory Segmentation without paging
- 2. Segmented-paged-model: Memory Segmentation with paging
- 3. Simple flat model:
 - 80386 does not have a way to turnoff segmentation, but we can eliminate segmentation by initializing all the segment registers with the same base address and segment limits for 4GB.
 - Useful in small systems where we require fast execution of program with out protection.
- 4. Paged flat model
 - In simple flat model if paging is enable then page level protection can be implemented.

Virtual 8086 Mode

- There are 3 operation mode in 80386 processor.
 1. Real mode
 2. Protected mode
 3. Virtual 8086 Mode
- Once the 80386 enters the protected mode from the real mode, it cannot return back to the real mode without a reset operation.
- In multiuser / multitasking / multiprogramming environment it might be possible one user wants to execute program in real mode (Segment-offset addressing) and other user wants to execute program in protected mode.
- 80286 fails here but 80386 allows it using Virtual 8086 mode.
- Thus, the virtual 8086 mode of operation of 80386, offers an advantage of executing 8086 programs while in protected mode.

Virtual 8086 Mode

- When 80386 is in protected mode and perform task switching it examines VM bit.
- If VM bit of EFLAG is set then it enter into the virtual 8086 mode. This is done by Operating System functions.
- In Virtual 8086 mode, 386 provides a virtual 8086 operating environment to execute the 8086 programs.
- The address forming mechanism in virtual 8086 mode is exactly identical with that of 8086 real mode (Segment-offset). So address range is 1MB.
- In virtual mode, 8086 can address 1Mbytes of physical memory that may be anywhere in the 4Gbytes address space.
- In virtual mode, the paging mechanism and protection capabilities are available at the service of the programmers.

Virtual 8086 Mode

- To return to the protected mode from the virtual mode, any interrupt or execution may be used.
- As a part of interrupt service routine, the VM bit may be reset to zero to pull back the 80386 into protected mode.
- The virtual 8086 mode executes all the programs at privilege level 3. Any of the other programmes may deny access to the virtual mode programs or data.

80386 system descriptors

| S | TYPE in binary | TYPE in HEX | System - Descriptor information |
|---|----------------|-------------|---------------------------------|
| 0 | 0 0 1 0 | 2 | LDT |
| 0 | 0 1 0 1 | 5 | 386 Task Gate |
| 0 | 1 0 0 1 | 9 | Available 386 TSS |
| 0 | 1 0 1 1 | B | Busy 386 TSS |
| 0 | 1 1 0 0 | C | 386 CALL gate |
| 0 | 1 1 1 0 | E | 386 Interrupt Gate |
| 0 | 1 1 1 1 | F | 386 Trap Gate |

| | | | | | | | | |
|-------------|-------------|----------|--------------|-------------|---|---|---|------------------|
| Base(31-24) | | | | 0 | 0 | 0 | 0 | Limit (19-16) |
| P | 00 (DPL) | (0) S | 0010 TYPE | Base(23-16) | | | | |
| Base(15-0) | | | | | | | | |
| Limit(15-0) | | | | | | | | |

| | | | | |
|----------------------------|-----|-------|-----------|---|
| - | | | | |
| P | DPL | (0) S | TYPE 0101 | - |
| Destination Selector(15-0) | | | | |
| - | | | | |

| | | | | | | | | |
|--------------|-----|----------|--------------|-------------|---|---|---|------------------|
| Base(31-24) | | | | G | D | X | U | Limit (19-16) |
| P | DPL | S (0) | TYPE 10B1 | Base(23-16) | | | | |
| Base (15-0) | | | | | | | | |
| Limit (15-0) | | | | | | | | |

| | | | | |
|----------------------------|-----|-------|-----------|----------|
| Destination offset(31-16) | | | | |
| P | DPL | (0) S | TYPE 1110 | Reserved |
| Destination Selector(15-0) | | | | |
| Destination offset(15-0) | | | | |

| | | | | |
|----------------------------|-----|-------|-----------|----------|
| Destination offset(31-16) | | | | |
| P | DPL | (0) S | TYPE 1111 | Reserved |
| Destination Selector(15-0) | | | | |
| Destination offset(15-0) | | | | |

| | | | | | |
|----------------------------|-----|-------|-----------|-----|----------------------------|
| Destination offset(31-16) | | | | | |
| P | DPL | (0) S | TYPE 1100 | 000 | Word Count (WC) (4 – bits) |
| Destination Selector(15-0) | | | | | |
| Destination offset(15-0) | | | | | |

80286 and 80386 Descriptor Format (code/data/stack)

| 15 | | 8 | | 7 | | 0 | |
|----------------|-----|---|------|---|-------------|---|---|
| Intel Reserved | | | | | | | 6 |
| P | DPL | S | TYPE | A | Base(23-16) | | 4 |
| Base(15-0) | | | | | | | 2 |
| Limit(15-0) | | | | | | | 0 |

80286 descriptor format

- 80286 descriptor is of 8 bytes (**Upper two bytes are reserved**).
- The base field is 24 - bits and LIMIT field is 16 – bit.
- The largest size of segment is 64KB only.

| 15 | | | | | 8 | | 7 | | | 0 | | |
|-------------|-----|--|---|------|---|-------------|---|---|---|---|------------------|---|
| Base(31-24) | | | | | | | G | D | X | U | Limit (19-16) | 6 |
| P | DPL | | S | TYPE | A | Base(23-16) | | | | | 4 | |
| Base(15-0) | | | | | | | | | | | 2 | |
| Limit(15-0) | | | | | | | | | | | 0 | |

80386 descriptor format

- 80386 descriptor is of 8 bytes.
- It is extension of 80286 descriptor.
- The base field is 32 - bits and LIMIT field is 20 – bit.
- Segment can be of 1 byte to 4GB in size.

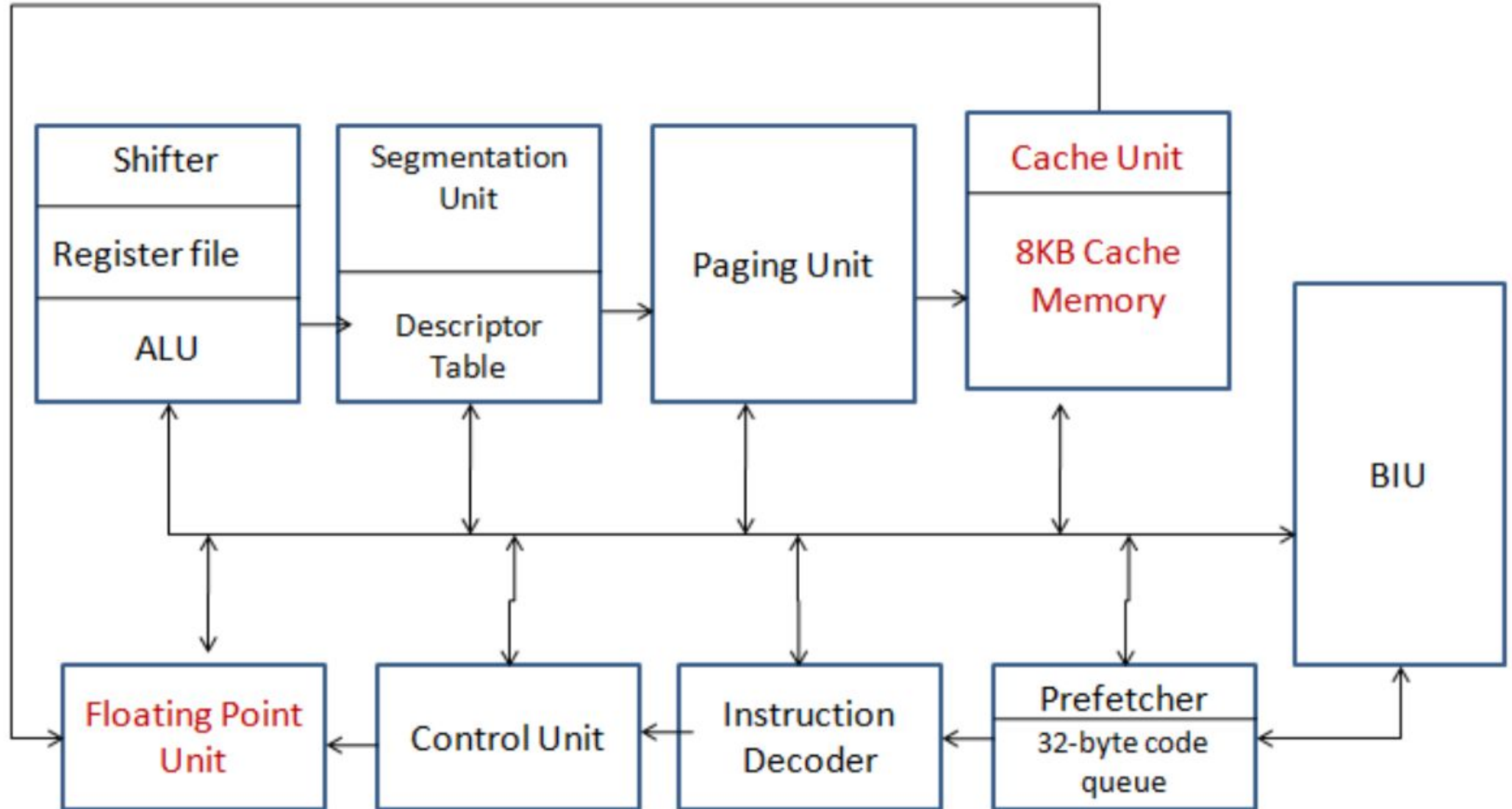
Intel family of microprocessor, bus and memory sizes

| Microprocessor | Data bus width | Address bus width | Memory size |
|--------------------|----------------|-------------------|-------------|
| 8086 | 16 | 20 | 1MB |
| 80186 | 16 | 20 | 1MB |
| 80286 | 16 | 24 | 16MB |
| 80386 | 32 | 32 | 4GB |
| 80486 | 32 | 32 | 4GB |
| Pentium 4 & core 2 | 64 | 40 | 1TB |

80486 Microprocessor

- The 32-bit 80486 is the next evolutionary step up from the 80386.
- One of the most obvious feature included in a 80486 is a built in **math coprocessor**. This coprocessor is essentially the same as the 80387 processor used with a 80386, but being integrated on the chip allows it to **execute math instructions about three times as fast as a 80386/387 combination**.
- 80486 is an **8Kbyte** code and data cache this improve the speed of 80486.
- It consists of parity generator/checker unit in order to implement parity detection and generation for memory reads and writes

80486 Architecture



80486 Architecture

- **BIU:**

- BIU generates address, data and control signals for a bus cycle it is supported with an additional parity detection/generation for memory reads and writes.
- During memory write operation, the 486 generates even parity bit for each byte outputs these bits. These bits will be stored in a separate parity memory bank.
- During read operation, stored parity bits will be read from the parity memory.
- 80486 checks the parities of data bytes read and compare them with the DP0 – DP3 signals and generates parity check error, if it occurs

80486 Architecture

- **Instruction Pre-fetch Unit** : It pre-fetches the instruction bytes in advance and holds them in a 32–byte code queue.
- **Instruction Decoder** : Decodes the instructions in the queue and passes the control and protection test unit.
- **Execution Unit**: Executes the instruction with the help of Shifter, ALU and Register bank.
- **Segmentation Unit and Paging Unit** : They are part of MMU (which manages virtual memory of system). Helpful in generation of Physical Address.
- Work same as they work in 80386.

80486 Architecture

- **Floating Point Unit :**
 - Responsible for performing the floating point operations.
 - The 80486 has a built in math co-processor. It performs the floating point operations.
 - It executes math instructions 3 times faster than the 386/387 combination.
- **Cache Unit :**
 - 8KB cache
 - Additional high speed cache memory provides a way of improving overall system performance.
 - It Contains the recently used instructions, data or both.
 - The main aim is that the microprocessor unit access code and data in the cache most of time, instead from the main memory.

References

- **BOOKS:**

1. Microprocessors and interfacing by: Dauglas V. Hall
2. 8086 Programming and Advanced Processor Architecture by: M. T. Savaliya

- **WEBSITE:**

- <https://pdos.csail.mit.edu/6.828/2005/readings/i386/toc.htm>