Microprocessors & Microcontrollers : Unit-2

Prof. Prajay P. Raikar
Visiting Faculty
Department of Computer Engineering
Goa College of Engineering, Farmagudi-Ponda, Goa
prajayraikar@gmail.com

Table of Contents

Introduction to Memory Management

Memory Management techniques are implemented using both hardware and software. Processor hardware contains MMU (Memory Management Unit) that aids in memory management while teaming up with OS (Operating System).

Windows does use segmentation, but not in the way people typically imagine from early computing days. Modern Windows operating systems (like Windows NT and its successors) run on processors where segmentation is available, yet they configure it in a "flat" model, relying primarily on paging for memory management and protection. Here's a deeper look:

1. **The Legacy and the Hardware Reality**

x86 Architecture Essentials:

The x86 architecture was originally built around segmentation. Processors have segment registers (CS, DS, ES, FS, GS, SS) that define base addresses and limits for memory segments.

Legacy Requirements:

Early systems (and older operating systems) actively used segmentation to divide memory into distinct regions. This approach was integral in systems like DOS.

2. **How Modern Windows Uses Segmentation**

Flat Segmentation Model:

Modern Windows operating systems configure the segment registers so that they effectively cover the entire address space. For example, most segments are set with a base of zero and a limit that spans the full 4 GB (or more in 64bit mode) address range—this creates a "flat" memory view.

Primary Reliance on Paging:

The real work of memory protection, process isolation, and virtual memory management in Windows is done by the paging system. Paging maps virtual addresses to physical memory, allowing Windows to implement features like virtual memory, process isolation, and efficient multitasking.

3. **Special Uses of Segmentation in Windows**

ThreadLocal Storage (TLS):

Even though most segments are flat, Windows makes use of segmentation for special purposes. For instance, on 32bit Windows systems, the FS register is often used to point to a threadspecific data structure. In 64bit Windows, the GS register frequently holds such information. This mechanism helps the system quickly access threadlocal data without the overhead of additional memory lookups.

System and Compatibility Purposes:

Although the segmentation mechanism is not actively used to separate all application memory, its existence is essential for running legacy software and ensuring hardware compatibility.

4. **Summary**

Yes, Windows uses segmentation—but in a limited, largely "invisible" way:

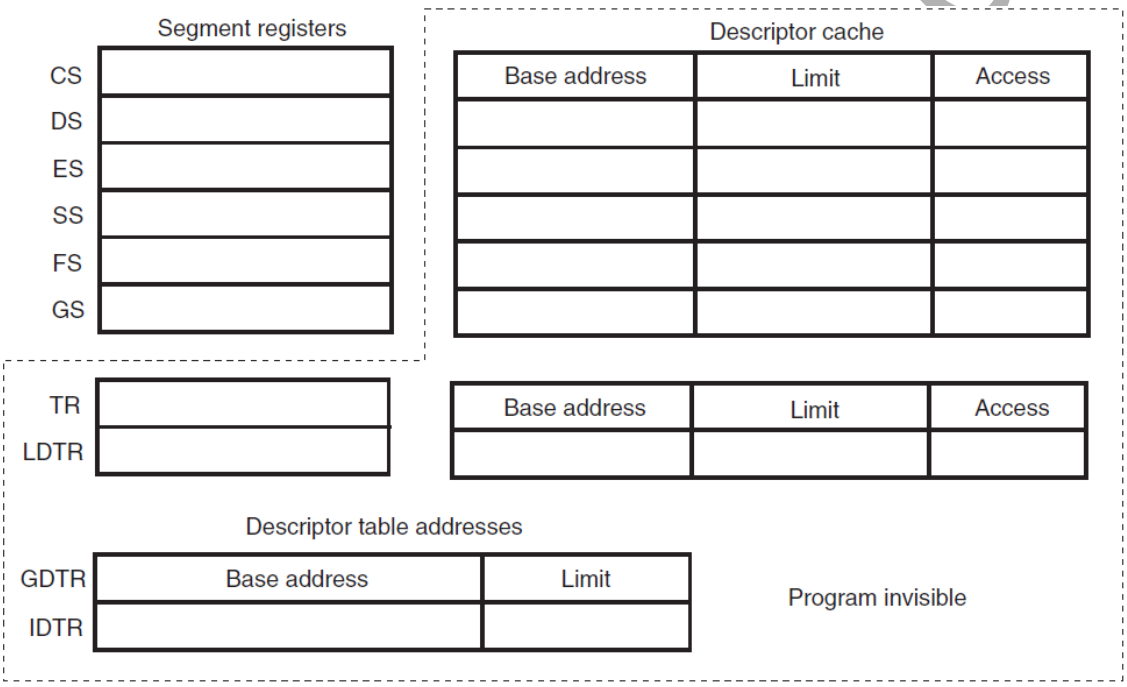
Visible Setup: The OS sets up segment registers to provide a flat memory model.

Behind the Scenes: Paging takes care of virtual memory and fine-grained access control.

Special Cases: Registers like FS and GS are used to efficiently manage threadlocal storage.

Thus, while segmentation is a built-in feature of the CPU that Windows relies on for technical and compatibility reasons, modern memory management in Windows is dominated by paging rather than heavy segmentation-based isolation or protection.

Program Invisible Register Portions



Notes:
1. The 80286 does not contain FS and GS nor the program-invisible portions of these registers.
2. The 80286 contains a base address that is 24-bits and a limit that is 16-bits.
3. The 80386/80486/Pentium/Pentium Pro contain a base address that is 32-bits and a limit that is 20-bits
4. The access rights are 8-bits in the 80286 and 12-bits in the 80386/80486/Pentium–Core2.

FIGURE 2–10 The program-invisible register within the 80286–Core2 microprocessors.

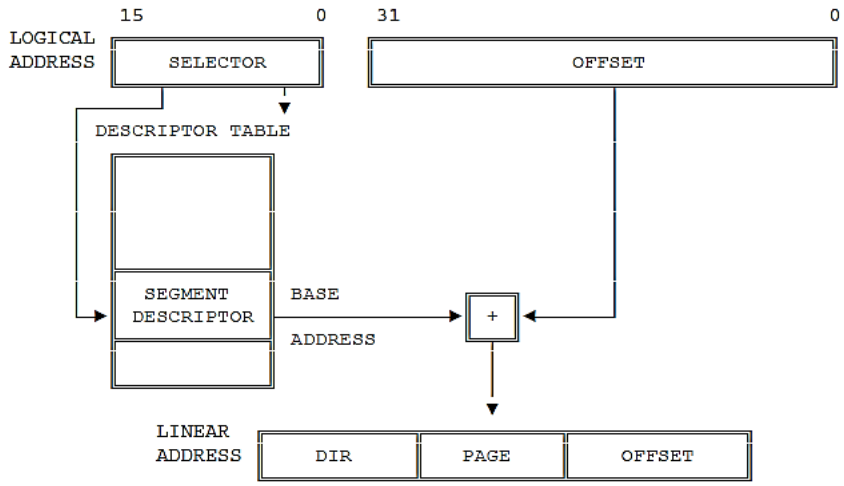
Memory Management in 80386

Segment Translation

- Figure 5-2 shows in more detail how the processor converts a logical address into a linear address.
- To perform this translation, the processor uses the following data structures:

1. Descriptors (Discussed in Unit-1)
2. Descriptor Tables (Discussed in Unit-1)
3. Selectors (Discussed in Unit-1)
4. Segment Registers

Figure 5-2. Segment Translation



Segment Registers

- The 80386 stores information from descriptors in segment registers, thereby avoiding the need to consult a descriptor table every time it accesses memory.
- Every segment register has a **"visible"** portion and an **"invisible"** portion, as Figure 5-7 illustrates.
- The visible portions of these segment address registers are **manipulated by programs** as if they were simply 16-bit registers.
- The invisible portions are **manipulated by the processor**.
- The operations that load these registers are normal program instructions (previously described in Chapter 3). These instructions are of two classes:
  1. Direct load instructions; for example, MOV, POP, LDS, LSS, LGS, LFS. These instructions explicitly reference the segment registers.
  2. Implied load instructions; for example, far CALL and JMP. These instructions implicitly reference the CS register, and load it with a new value.
- Using these instructions, a program loads the visible part of the segment register with a 16-bit selector.
- The processor automatically fetches the base address, limit, type, and other information from a descriptor table and loads them into the invisible part of the segment register.
- This invisible portion is sometimes referred to as **segment descriptor cache registers**.

Figure 5-7. Segment Registers

	16-BIT VISIBLE SELECTOR	HIDDEN DESCRIPTOR
CS		
SS		
DS		
ES		
FS		
GS		

Introduction to 80386 Control Registers

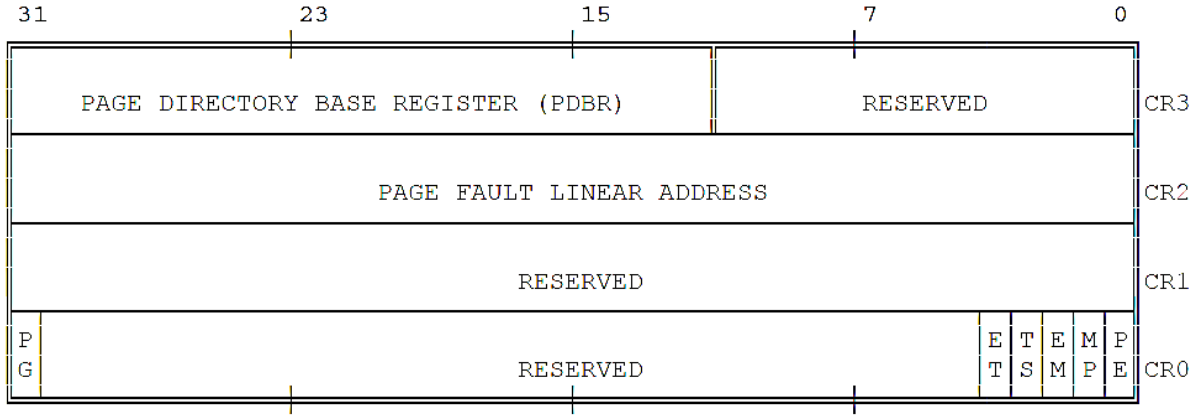
- There are 4 control registers CR0, CR1, CR2 & CR3.
- These registers are accessible to systems programmers only via variants of the MOV instruction, which allow them to be loaded from or stored in general registers; for example:

  MOV EAX, CR0

  MOV CR3, EBX

- **CR0** contains system control flags, which control or indicate conditions that apply to the system as a whole, not to an individual task.
  - PE (Protection Enable, bit 0)
    - Setting PE causes the processor to begin executing in protected mode.
    - Resetting PE returns to real-address mode.
    - Refer to Chapter 14 and Chapter 10 for more information on changing processor modes.
  - MP (Math Present, bit 1)
    - MP controls the function of the WAIT instruction, which is used to coordinate a coprocessor.
    - Refer to Chapter 11 for details.
  - EM (Emulation, bit 2)
    - EM indicates whether coprocessor functions are to be emulated.
    - Refer to Chapter 11 for details.
  - TS (Task Switched, bit 3)
    - The processor sets TS with every task switch and tests TS when interpreting coprocessor instructions.
    - Refer to Chapter 11 for details.
  - ET (Extension Type, bit 4)
    - ET indicates the type of coprocessor present in the system (80287 or 80387).
    - Refer to Chapter 11 and Chapter 10 for details.
  - PG (Paging, bit 31)
    - PG indicates whether the processor uses page tables to translate linear addresses into physical addresses.
    - Refer to Chapter 5 for a description of page translation; refer to Chapter 10 for a discussion of how to set PG.
- **CR2** is used for handling page faults when PG is set. The processor stores in CR2 the linear address that triggers the fault. Refer to Chapter 9 for a description of page-fault handling.
- **CR3** is used when PG is set. CR3 enables the processor to locate the page table directory for the current task. Refer to Chapter 5 for a description of page tables and page translation.

Figure 4-2. Control Registers



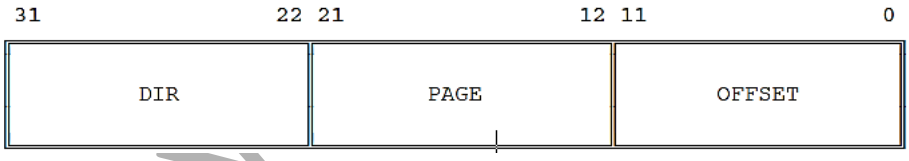
Page Translation

- In the second phase of address transformation, the 80386 transforms a linear address into a physical address.
- This phase of address transformation implements the basic features needed for **page-oriented virtual-memory** systems and **page-level protection**.
- The page-translation step is optional. Page translation is in effect only when the PG bit of CR0 is set.
- This bit is typically set by the operating system during software initialization.
- The PG bit must be set if the operating system is to implement multiple virtual 8086 tasks, page-oriented protection, or page-oriented virtual memory.

Linear Address

- A linear address refers indirectly to a physical address by specifying a page table, a page within that table, and an offset within that page.
- Figure 5-8 shows the format of a linear address.

Figure 5-8. Format of a Linear Address



- Figure 21-10 shows how the processor converts the DIR, PAGE, and OFFSET fields of a linear address into the physical address by consulting two levels of page tables.
- The addressing mechanism uses the DIR field as an index into a page directory, uses the PAGE field as an index into the page table determined by the page directory, and uses the OFFSET field to address a byte within the page determined by the page table.

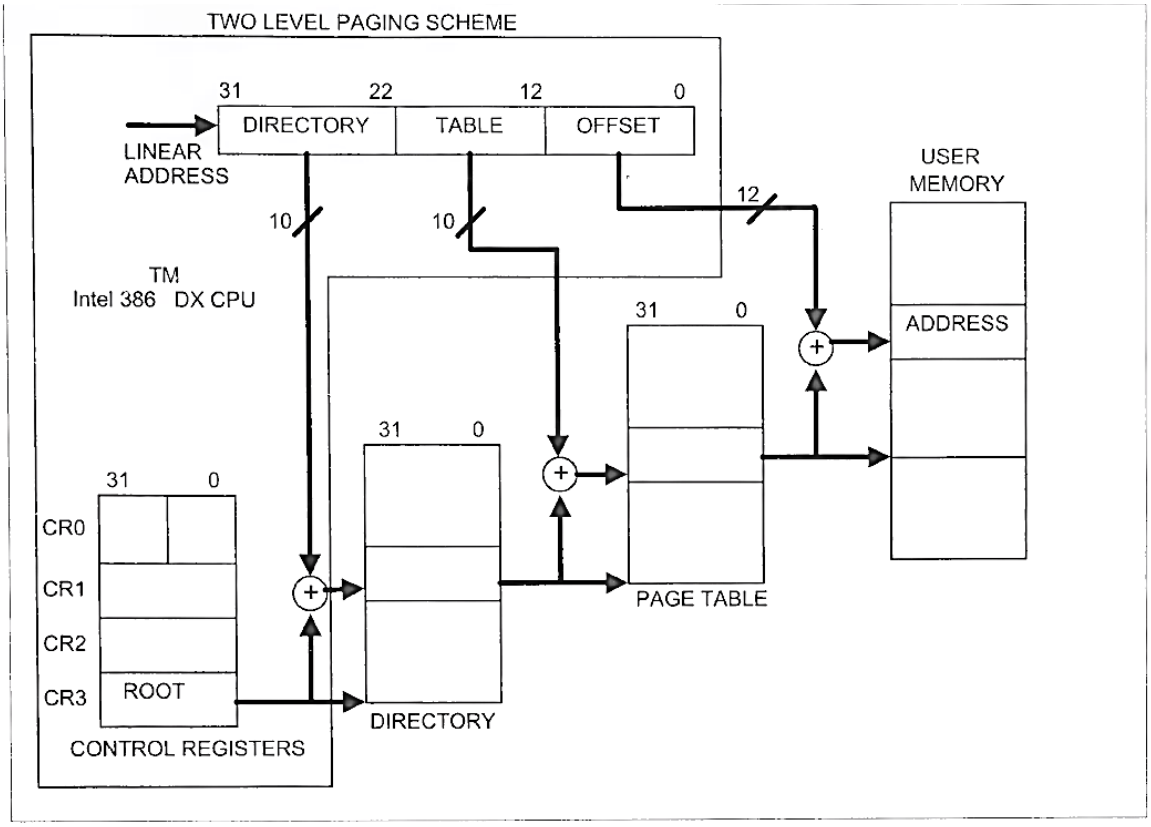
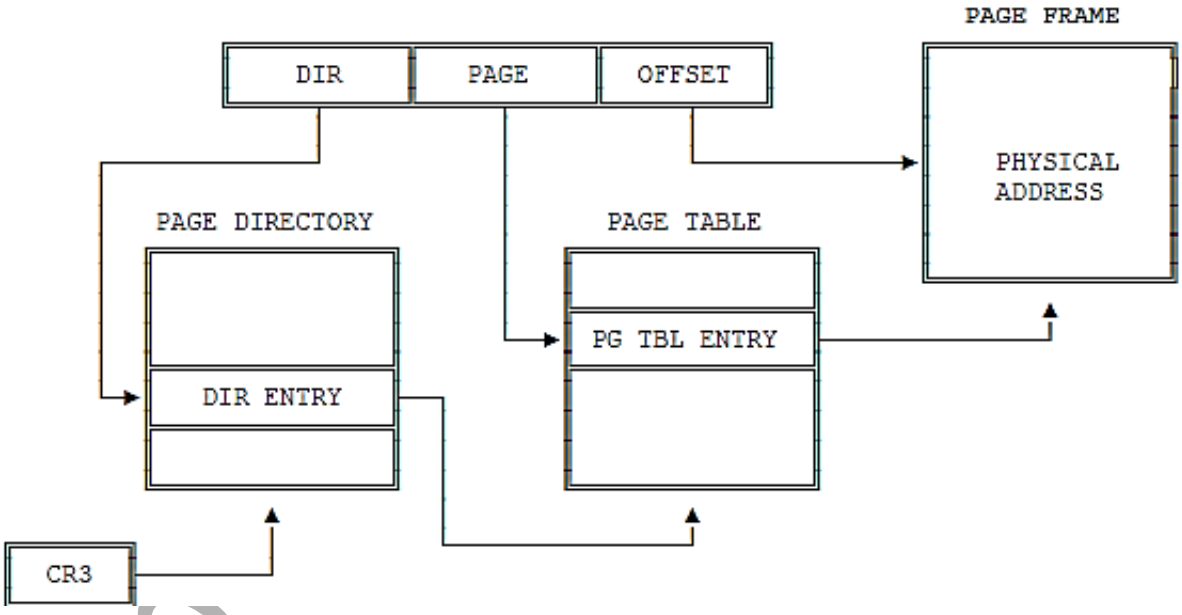


Figure 21-10. Paging Mechanism

Both diagrams convey same paging mechanism. Use anyone as per your convenience.

Figure 5-9. Page Translation



Page Tables

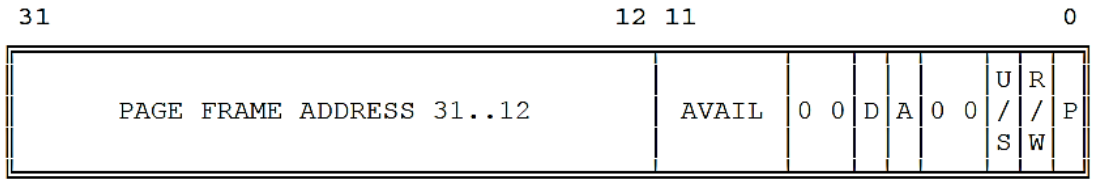
- A page table is simply an array of 32-bit page specifiers.

- A page table is itself a page, and therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.
- Two levels of tables are used to address a page of memory.
- At the higher level is a page directory.
- The page directory addresses up to **1K page tables** of the second level.
- A page table of the second level addresses up to **1K pages**.
- All the tables addressed by one page directory, therefore, can address 1M pages (2^{20}).
- Because each page contains 4K bytes (2^{12} bytes), the tables of one page directory can span the entire physical address space of the 80386 (2^{20} times $2^{12} = 2^{32}$).
- The physical address of the current page directory is stored in the CPU register CR3, also called the page directory base register (PDBR).
- The lower 12-bits of CR3 is zero (reserved) to align with 4K page size.
- **Memory management software has the option of using one page directory for all tasks, one page directory for each task, or some combination of the two.**
- Refer to Chapter 10 for information on initialization of CR3. Refer to Chapter 7 to see how CR3 can change for each task.

Page Table Entries

- Entries in either level of page tables have the same format. Figure 5-10 illustrates this format. Hence, format is applicable to page table & page directory entry.

Figure 5-10. Format of a Page Table Entry




```
P       - PRESENT
R/W     - READ/WRITE
U/S     - USER/SUPERVISOR
D       - DIRTY
AVAIL   - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

Page Frame Address

- The page frame address specifies the physical starting address of a page.
- Because pages are located on 4K boundaries, the low-order 12 bits are always zero.
- In a page directory, the page frame address is the address of a page table.
- In a second-level page table, the page frame address is the address of the page frame that contains the desired memory operand.

Present Bit

- This bit indicates whether the page is currently present in physical memory. If the present bit is set (1), the page is in physical memory. If it is clear (0), the page is not in memory, and a page fault will occur if accessed.
- The Present bit indicates whether a page table entry can be used in address translation.
- P=1 indicates that the entry can be used.

- When P=0 in either level of page tables, the entry is not valid for address translation, and the rest of the entry is available for software use; none of the other bits in the entry is tested by the hardware.
- Figure 5-11 illustrates the format of a page-table entry when P=0.
- If P=0 in either level of page tables when an attempt is made to use a page-table entry for address translation, the processor signals a page exception.
- In software systems that support paged virtual memory, the page-not-present exception handler can bring the required page into physical memory. The instruction that caused the exception can then be re-executed.
- Refer to Chapter 9 for more information on exception handlers.
- Note that there is **no present bit for the page directory** itself.
- The page directory may be not-present while the associated task is suspended, but the operating system must ensure that the page directory indicated by the CR3 image in the TSS is present in physical memory before the task is dispatched. Refer to Chapter 7 for an explanation of the TSS and task dispatching.

Figure 5-11. Invalid Page Table Entry

Accessed and Dirty Bits

- Accessed Bit (A): This bit is set (1) by the hardware whenever the page is accessed (read or written). It helps the operating system keep track of page usage.
- Dirty Bit (D): This bit is set (1) by the hardware when the page is written to. It indicates that the page has been modified and needs to be written back to disk if it is swapped out.
- These bits provide data about page usage in both levels of the page tables.
- With the exception of the dirty bit in a page directory entry, these **bits are set by the hardware; however, the processor does not clear any of these bits.**
- The **processor sets** the corresponding accessed bits in both levels of page tables to one before a read or write operation to a page.
- The processor sets the dirty bit in the second-level page table to one before a write to an address covered by that page table entry.
- The dirty bit in directory entries is undefined.
- An operating system that supports paged virtual memory can use these bits to determine what pages to eliminate from physical memory when the demand for memory exceeds the physical memory available.
- **The operating system is responsible for testing and clearing these bits.**
- Refer to Chapter 11 for how the 80386 coordinates updates to the accessed and dirty bits in multiprocessor systems.

Read/Write and User/Supervisor Bits

- These bits are not used for address translation, but are used for page-level protection, which the processor performs at the same time as address translation.
- When the processor is executing at supervisor level, all pages are both readable and writable.

- When the processor is executing at user level, only pages that belong to user level and are marked for read/write access are writable; pages that belong to supervisor level are neither readable nor writable from user level.
    - **Supervisor level (U/S=0)** — for the operating system and other systems software and related data.
    - **User level (U/S=1)** — for applications procedures and data.
- The supervisor privilege level is equivalent to levels 0, 1, and 2 in the segmentation method.
- Refer to Chapter 6 where protection is discussed in detail.
- In the segmentation method there were 2 bits for privilege level, giving rise to four levels of protection of 0, 1, 2, and 3, where level 3 was assigned to the lowest level and level 0 to the highest level.
- However, in the paging method, there is only 1 bit for privilege level, which is called U/S (user/supervisor).

Table 21-7: Paging and Segmentation Comparison

Feature	Paging	Segmentation
Size	4K bytes	Any size
Levels of privilege	2	4
Base address	4K-byte aligned	Any address
Dirty bit	Yes	No
Access bit	Yes	Yes
Present bit	Yes	Yes
Read/write protection	Yes	Yes

Page Translation Cache / Translation Lookaside Buffer (TLB)

Since the TLB in the 386 keeps the list of addresses for the 32 most recently used pages, it allows the CPU to have access to 128K bytes (32 × 4 = 128) of code and data at any time without going through the time-consuming process of converting the linear address to a physical address (two-stage table translation). See Figure 21-13. Therefore, one way to enhance the processor is to increase the

number of pages held by the TLB.

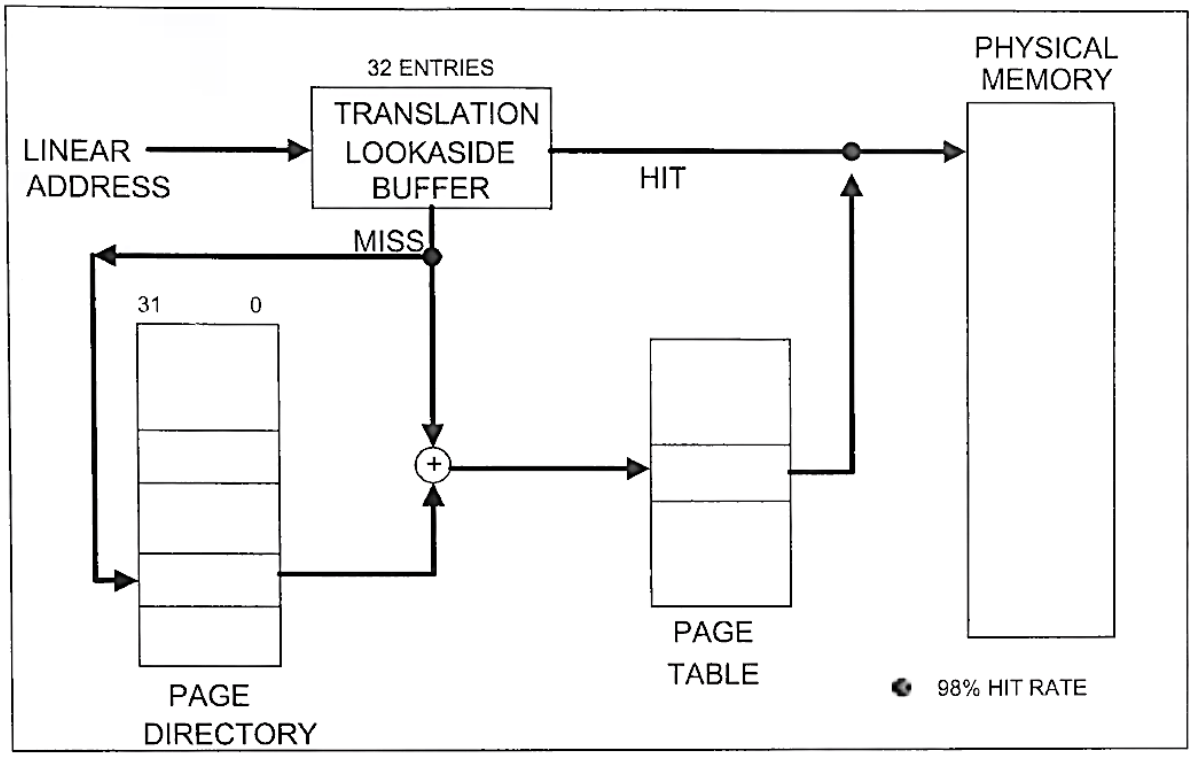


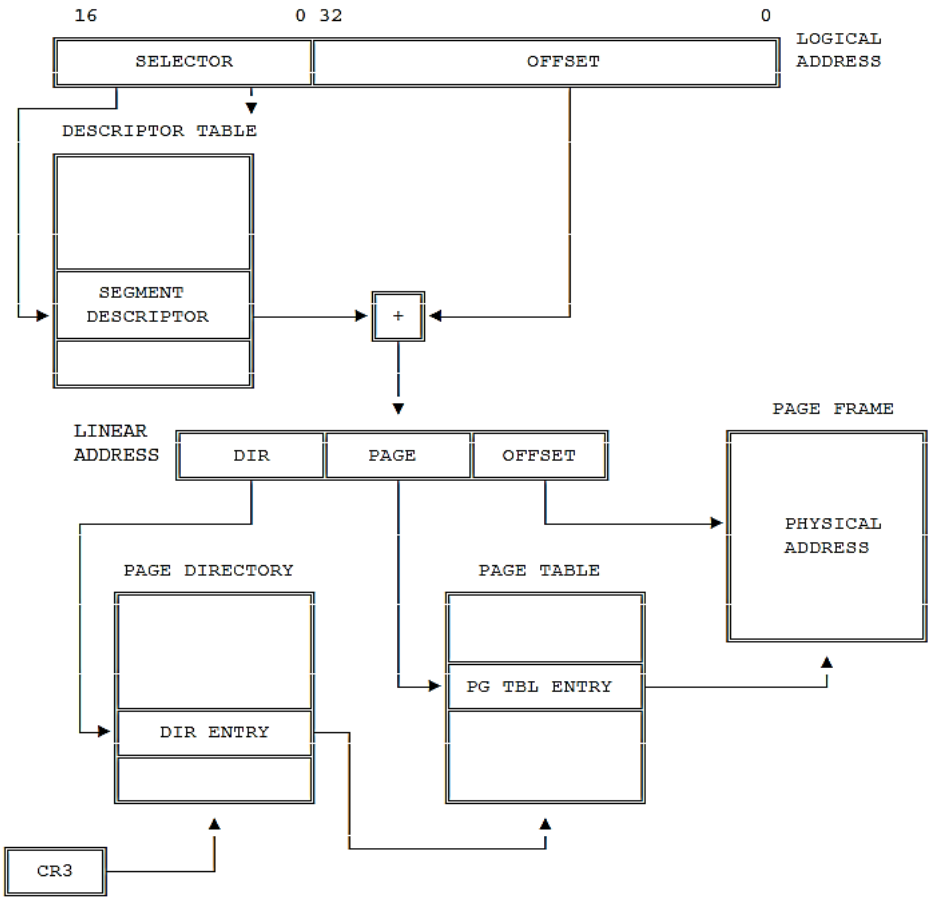
Figure 21-13. Translation Lookaside Buffer

- The page-translation cache can be flushed by either of two methods:
  - By reloading CR3 with a MOV instruction; for example: MOV CR3, EAX
  - By performing a task switch to a TSS that has a different CR3 image than the current TSS. (Refer to Chapter 7 for more information on task switching.)

Combining Segment & Page Translation

- Figure 5-12 combines Figure 5-2 and Figure 21-10 to summarize both phases of the transformation from a logical address to a physical address when paging is enabled.
- By appropriate choice of options and parameters to both phases, memory-management software can implement several different styles of memory management.

Figure 5-12. 80306 Addressing Machanism



Flat Architecture or Disabling Segmentation

- When the 80386 is used to execute software designed for architectures that don't have segments, it may be convenient to effectively "turn off" the segmentation features of the 80386.
- The 80386 **does not have a mode that disables segmentation**, but the same effect can be achieved by initially loading the segment registers with selectors for descriptors that covers the entire 32-bit linear address space.
- Once loaded, the segment registers don't need to be changed. The 32-bit offsets used by 80386 instructions are adequate to address the entire linear-address space.

----------EXTRA INFORMATION (NOT PART OF SYLLABUS)----------

Setting up a flat memory model in an operating system involves configuring the memory segments so that they effectively cover the entire addressable memory space, bypassing segmentation. Here's a concise guide:

1. Initialize Segment Descriptors:

- In a flat memory model, all segments (like code, data, and stack) start at address 0 and span the entire addressable space.

- For x86 architectures, you'll need to configure the Global Descriptor Table (GDT) with segment descriptors that:

- Have a base address of `0x00000000`.

- Have a limit of `0xFFFFFFFF` (for 32-bit systems) or `0xFFFFFFFFFFFFFFFF` (for 64-bit systems).

2. Set Up the GDT:

- Define segment descriptors for the code and data segments in the GDT, with the settings above.
- Load the GDT into the `GDTR` register using the `lgdt` assembly instruction.

3. Load Segment Registers:

- Update the segment registers (CS, DS, ES, FS, GS, SS) to point to the flat segments defined in the GDT.
- For example:

```asm
mov ax, flat_data_segment_selector
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax
mov ss, ax
```

4. Switch to Protected Mode (if not already in it):

- The flat memory model is typically used in protected mode. Ensure that the Protected Mode Enable (PE) bit in the `CR0` register is set.

Once set up, the flat memory model allows programs to access the entire memory space directly, as segmentation no longer imposes any effective boundaries.

Segments Spanning Several Pages

- The architecture of the 80386 permits segments to be larger or smaller than the size of a page (4 Kilobytes).
- For example, suppose a segment is used to address and protect a large data structure that spans 132 Kilobytes.
- In a software system that supports paged virtual memory, it is not necessary for the entire structure to be in physical memory at once.
- The structure is divided into 33 pages, any number of which may not be present.
- The applications programmer does not need to be aware that the virtual memory subsystem is paging the structure in this manner.

Pages Spanning Several Segments

- On the other hand, segments may be smaller than the size of a page.

- For example, consider a small data structure such as a semaphore.
- Because of the protection and sharing provided by segments (refer to Chapter 6), it may be useful to create a separate segment for each semaphore.
- But, because a system may need many semaphores, it is not efficient to allocate a page for each.
- Therefore, it may be useful to cluster many related segments within a page.

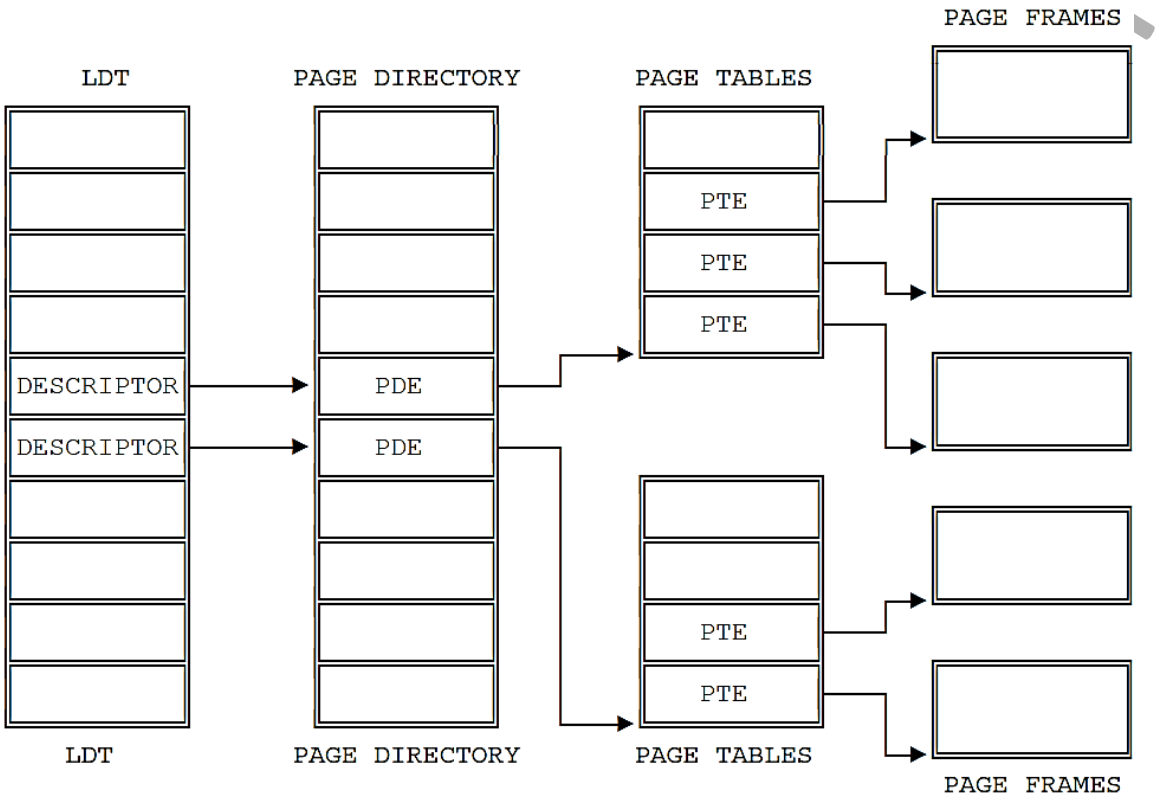
Aligned Page and Segment Boundaries – Combining Segmentation & Paging

- Memory-management software may be simpler, however, if it enforces some correspondence between page and segment boundaries.
- For example, if segments are allocated only in units of one page, the logic for segment and page allocation can be combined.
- There is no need for logic to account for partially used pages.

Page-Table per Segment

- An approach to space management that provides even further simplification of space-management software is to maintain a one-to-one correspondence between segment descriptors and page-directory entries, as Figure 5-13 illustrates.
- Each descriptor has a base address in which the low-order 22 bits are zero; in other words, the base address is mapped by the first entry of a page table.
- A segment may have any limit from 1 to 4 megabytes.
- Depending on the limit, the segment is contained in from 1 to 1K page frames.
- A task is thus limited to 1K segments (a sufficient number for many applications), each containing up to 4 Mbytes.
- The descriptor, the corresponding page-directory entry, and the corresponding page table can be allocated and deallocated simultaneously.

Figure 5-13. Descriptor per Page Table



----------EXTRA INFORMATION (NOT PART OF SYLLABUS)----------

Introduction to GDT & LDT

Global Descriptor Table (GDT)

- **Purpose**: The GDT is used by the operating system to define the characteristics of various memory segments for all processes in the system.
- **Scope**: Global. It is shared by all processes and provides a common set of segment descriptors.
- **Descriptors**: The GDT contains descriptors that define the base address, size, and access privileges of memory segments. These descriptors can define code segments, data segments, task state segments (TSS), and more.
- **Access**: The GDT is accessed using the GDTR (Global Descriptor Table Register), which holds the base address and limit of the GDT.

Local Descriptor Table (LDT)

- **Purpose**: The LDT is used to define memory segments for individual processes. Each process can have its own LDT, allowing for more granular control over memory segmentation.
- **Scope**: Local. Each process can have its own LDT, which provides segment descriptors specific to that process.
- **Descriptors**: Similar to the GDT, the LDT contains descriptors that define memory segments. However, these descriptors are specific to the process that owns the LDT.

- **Access**: The LDT is accessed using the LDTR (Local Descriptor Table Register), which holds the base address and limit of the current process's LDT.

GDT vs LDT

- **Scope**: GDT is global and shared by all processes, while LDT is local and specific to individual processes.
- **Use Case**: GDT is used for defining system-wide segments, while LDT is used for process-specific segments.
- **Access**: GDT is accessed via the GDTR, and LDT is accessed via the LDTR.

Memory Protection

- The purpose of the protection features of the 80386 is to help detect and identify bugs.
- To help debug applications faster and make them more robust in production, the 80386 contains mechanisms to verify memory accesses and instruction execution for conformance to protection criteria.

. The lack of protection of the operating system or users' programs is one of the weaknesses of 8088/86-based MS-DOS. This weakness is due to the inability of the 8088/86 to block general instructions from accessing the core (kernel) of the operating system. In the 8088/86, any program can go from any code segment to any code segment, so it is easy to crash the system. In contrast, the 80386 provides resources to the operating system that prevent the user from either accidentally or maliciously taking over the core (kernel) of the operating system and forcing the system to crash. Of course, this idea of protection is nothing new; it is commonly used in mainframes and minicomputers, where it is often referred to as user and supervisor mode. The 386 provides protection by allowing any data or code to be assigned a privilege level. The four privilege levels are 0, 1, 2, and 3, where the privilege level of 0 is the highest and level 3 is the lowest. While operating systems are always assigned the highest privilege level (level 0), the user and applications such as word processors are assigned the lowest privilege level (level 3). Since the user is assigned the lowest privilege level, any attempt by the user to take over the operating system is blocked. Higher privilege levels can access lower levels, but not the other way around. Again, it must be emphasized that the protection mechanism can be used only when the 80386 is switched to protected mode.

Overview of 80386 Protection Mechanisms

- Protection in the 80386 has five aspects:
  1. Type checking
  2. Limit checking
  3. Restriction of addressable domain
  4. Restriction of procedure entry points
  5. Restriction of instruction set
- Protection is the integral part of memory management and hence plays role during segment and page translation.
- Each reference to memory is checked by the hardware to verify that it satisfies the protection criteria. All these checks are made before the memory cycle is started; any violation prevents that cycle from starting and results in an exception.
- Since the **checks are performed concurrently** with address formation, there is no performance penalty.

Segment-Level Protection

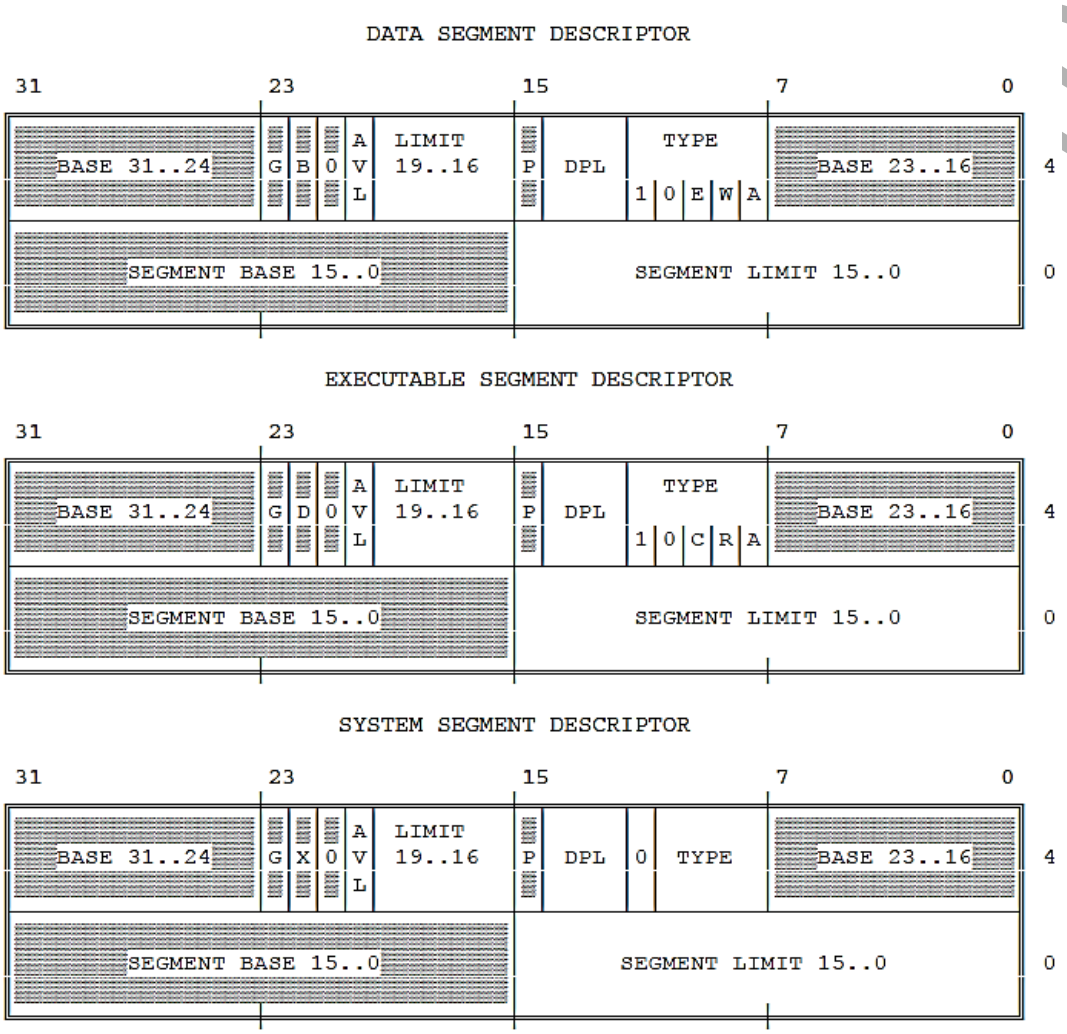
- All five aspects of protection apply to segment translation:
  1. Type checking
  2. Limit checking
  3. Restriction of addressable domain
  4. Restriction of procedure entry points
  5. Restriction of instruction set
- The segment is the unit of protection, and segment descriptors store protection parameters.

- Protection checks are performed automatically by the CPU when the selector of a segment descriptor is loaded into a segment register and with every segment access.
- Segment registers hold the protection parameters of the currently addressable segments.

Descriptors Store Protection Parameters

- Figure 6-1 highlights the protection-related fields of segment descriptors.
- The protection parameters are placed in the descriptor by systems software (OS) at the time a descriptor is created.
- In general, applications programmers do not need to be concerned about protection parameters.
- When a program loads a selector into a segment register, the processor loads not only the base address of the segment but also protection information.
- Each segment register has bits in the invisible portion for storing base, limit, type, and privilege level; therefore, subsequent protection checks on the same segment do not consume additional clock cycles.

Figure 6-1. Protection Fields of Segment Descriptors



A - ACCESSED
AVL - AVAILABLE FOR PROGRAMMERS USE
B - BIG
C - CONFORMING
D - DEFAULT
DPL - DESCRIPTOR PRIVILEGE LEVEL
E - EXPAND-DOWN
G - GRANULARITY
P - SEGMENT PRESENT
R - READABLE
W - WRITABLE

Type Checking

The TYPE field of a descriptor has two functions:

1. It distinguishes among different descriptor formats.
2. It specifies the intended usage of a segment.

Besides the descriptors for data and executable segments commonly used by applications programs, the 80386 has descriptors for special segments used by the operating system and for gates. Table 6-1 lists all the types defined for system segments and gates. Note that not all descriptors define segments; gate descriptors have a different purpose that is discussed later in this chapter.

Executable segment means code segment (CS).

The type fields of data and executable segment descriptors include bits which further define the purpose of the segment (refer to Figure 6-1):

- The writable bit in a data-segment descriptor specifies whether instructions can write into the segment.
- The readable bit in an executable-segment descriptor specifies whether instructions are allowed to read from the segment (for example, to access constants that are stored with instructions). A readable, executable segment may be read in two ways:
    - Via the CS register.
    - By loading a selector of the descriptor into a data-segment register (DS, ES, FS,or GS).

Type checking can be used to detect programming errors that would attempt to use segments in ways not intended by the programmer. The processor examines type information on two kinds of occasions:

1. When a selector of a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types; for example:
    a. The CS register can be loaded only with a selector of an executable segment.
    b. Selectors of executable segments that are **not readable** cannot be loaded into data-segment registers.
    c. Only selectors of writable data segments can be loaded into SS.
2. When an instruction refers (implicitly or explicitly) to a segment register. Certain segments can be used by instructions only in certain predefined ways; for example:
    a. No instruction may write into an executable segment.
    b. No instruction may write into a data segment if the writable bit is not set.
    c. No instruction may read an executable segment unless the readable bit is set.

Table 6-1. System and Gate Descriptor Types

Code	Type of Segment or Gate
0	-reserved
1	Available 286 TSS
2	LDT
3	Busy 286 TSS
4	Call Gate
5	Task Gate
6	286 Interrupt Gate
7	286 Trap Gate
8	-reserved
9	Available 386 TSS
A	-reserved
B	Busy 386 TSS
C	386 Call Gate
D	-reserved
E	386 Interrupt Gate
F	386 Trap Gate

Limit Checking

The limit field of a segment descriptor is used by the processor to prevent programs from addressing outside the segment. The processor's interpretation of the limit depends on the setting of the G (granularity) bit. For data segments, the processor's interpretation of the limit depends also on the E-bit (expansion-direction bit) and the B-bit (big bit) (refer to Table 6-2).

- When G=0, the actual limit is the value of the 20-bit limit field as it appears in the descriptor. In this case, the limit may range from 0 to 0FFFFFH (2^{20}-1 or 1 megabyte).

- When G=1, the processor appends 12 low-order one-bits to the value in the limit field.
- In this case the actual limit may range from 0FFFH (2^{12}-1 or 4 kilobytes) to 0FFFFFFFFH (2^{32}-1 or 4) gigabytes).

For all types of segments except expand-down data segments, the value of the limit is one less than the size (expressed in bytes) of the segment. The processor causes a general-protection exception in any of these cases:

- Attempt to access a memory byte at an address > limit.
- Attempt to access a memory word at an address ≥limit.
- Attempt to access a memory doubleword at an address ≥(limit-2).

For **expand-down data segments (Stack),** the limit has the same function but is interpreted differently. In these cases, the range of valid addresses is from limit + 1 to either 64K or 2^{32}-1 (4 Gbytes) depending on the B-bit. An expand-down segment has maximum size when the limit is zero.

----------EXTRA INFORMATION (NOT PART OF SYLLABUS)----------

When the "E" bit is set, the segment behaves differently, particularly useful for stack segments where the stack grows downwards (from higher addresses to lower addresses). Here's how it works:

1. **Base Address and Limit**: In an expand-down segment, the base address remains the lower bound, but the segment limit specifies the upper bound of the segment.

For example:

   - Base Address: 0x3000
   - Segment Limit: 0x2000

The addressable range is from 0x2000 to 0x3000.

2. **Effective Address Calculations**: The effective address (offset) within the segment is calculated by subtracting the offset from the segment limit. This ensures the segment grows downwards.
3. **Limit Check**: The effective address must be within the specified range. In this case, it should be between 0x2000 and 0x3000.

The **"B" bit**, also known as the "Big" bit or "D/B" (Default/Big) bit, is used in data segment descriptors to control the default operand size and address size for the segment. This bit is particularly important in the context of 32-bit and 16-bit segments in the x86 architecture.

The "B" bit ensures that the instructions use the correct operand and address sizes by default, helping to maintain compatibility and optimize performance.

Details:

1. **32-bit Segment (B bit = 1)**:
   - When the "B" bit is set to 1, the segment is considered a 32-bit segment.
   - Default operand size is 32 bits.
   - Default address size is 32 bits.

- This means that instructions operating on this segment will, by default, use 32-bit operands and addresses unless overridden.

2. **16-bit Segment (B bit = 0)**:
    - When the "B" bit is set to 0, the segment is considered a 16-bit segment.
    - Default operand size is 16 bits.
    - Default address size is 16 bits.
    - Instructions will use 16-bit operands and addresses unless overridden.

Example:

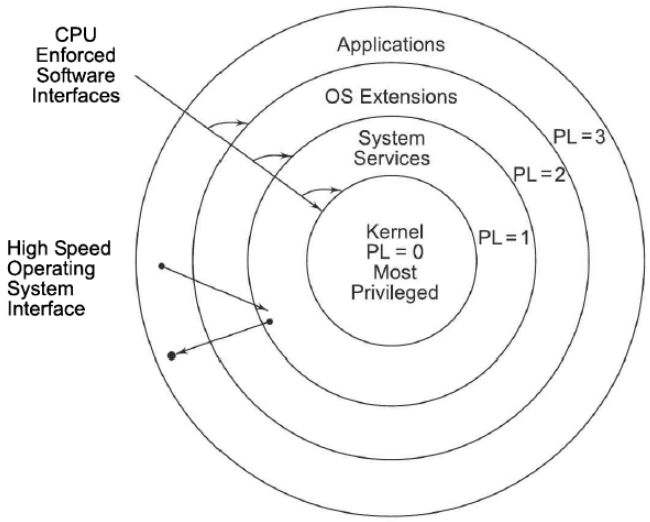
If you have a data segment descriptor with the following settings:

- Base Address: 0x0000
- Limit: 0xFFFF
- B bit: Set to 1 (32-bit segment)

Privilege Levels

- The concept of privilege is implemented by assigning a value from zero to three to key objects recognized by the processor. This value is called the privilege level.
- The following processor-recognized objects contain privilege levels:
    - **Descriptors** contain a field called the descriptor privilege level (**DPL**).
    - **Selectors** contain a field called the requestor's privilege level (**RPL**). The RPL is intended to represent the privilege level of the procedure (program) that originates a selector.
    - An internal processor register records the current privilege level (**CPL**). Normally the **CPL is equal to the DPL** of the segment that the processor is currently executing. CPL changes as control is transferred to segments with differing DPLs.

The processor automatically evaluates the right of a procedure to access another segment by comparing the CPL to one or more other privilege levels. The evaluation is performed at the time the selector of a descriptor is loaded into a segment register.



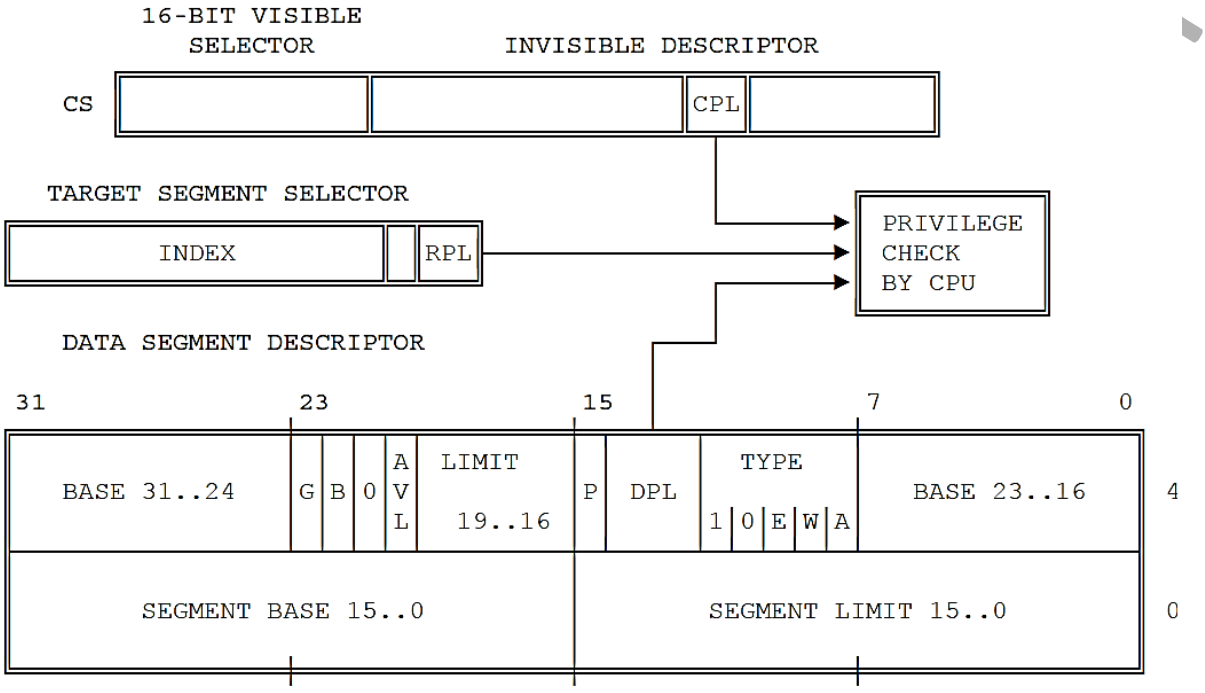
Note: PL Becomes Numerically Lower as Privilege Level Increases

Fig. 9.15 *Four Level Privilege Mechanism*

Restricting Access to Data

- To address operands in memory, an 80386 program must load the selector of a data segment into a data-segment register (DS, ES, FS, GS, SS).
- The processor automatically evaluates access to a data segment by comparing privilege levels.
- Instructions may load a data-segment register (and subsequently use the target segment) only if the DPL of the target segment is numerically greater than or equal to the maximum of the CPL and the selector's RPL.
- In other words, a procedure can only access data that is at the same or less privileged level.

Figure 6-3. Privilege Check for Data Access



CPL - CURRENT PRIVILEGE LEVEL
RPL - REQUESTOR'S PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL

Restricting Access to Code

- It is not possible to write to a segment described as a code segment.
- The following methods of accessing data in code segments are possible:
  - Load a data-segment register with a selector of a nonconforming, readable, executable segment.
  - Load a data-segment register with a selector of a conforming, readable, executable segment.
  - Use a CS override prefix to read a readable, executable segment whose selector is already loaded in the CS register.

Restricting Control Transfers

Near Control Transfer

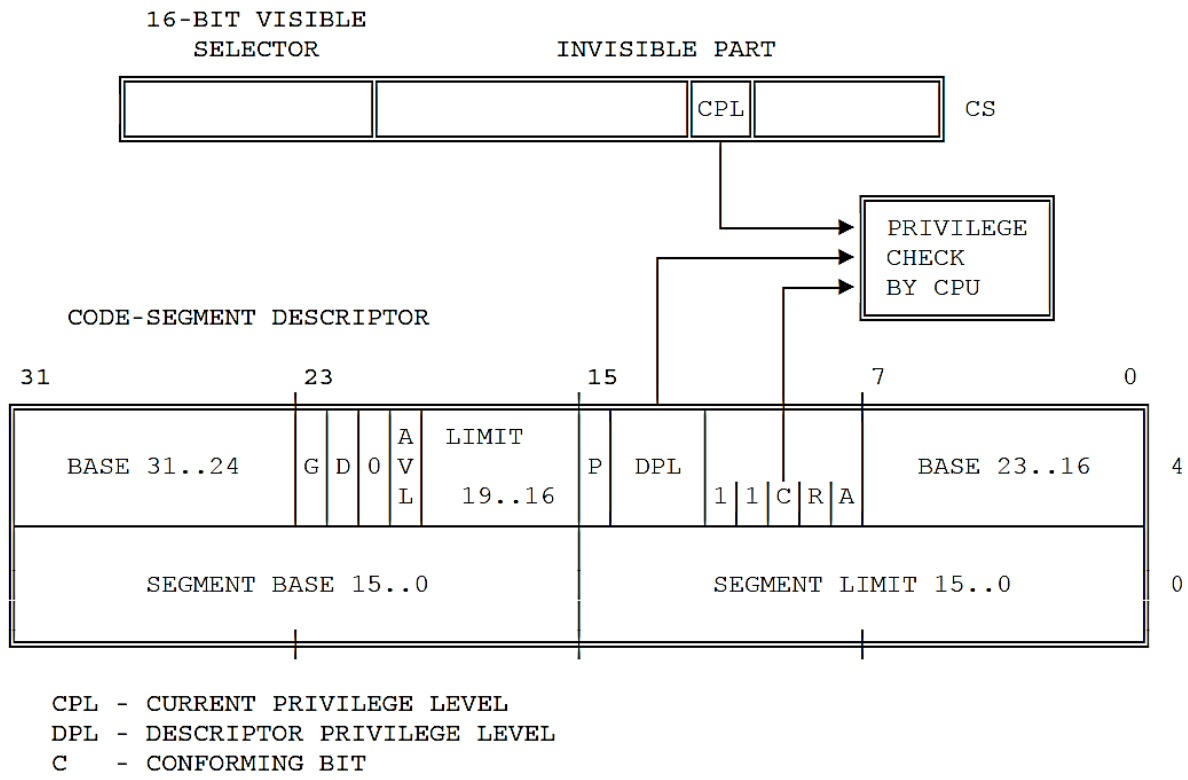
- The **"near" forms** of JMP, CALL, and RET transfer within the current code segment, and therefore are subject only to limit checking.
- The processor ensures that the destination of the JMP, CALL, or RET instruction does not exceed the limit of the current executable segment.
- This limit is cached in the CS register; therefore, protection checks for near transfers require no extra clock cycles.

Far Control Transfer

- The operands of the **"far" forms** of JMP and CALL refer to other segments; therefore, the processor performs privilege checking. There are two ways a JMP or CALL can refer to another segment:

- The operand selects the descriptor of another executable (code) segment.
- The operand selects a call gate descriptor. This gated form of transfer is discussed in a later section on call gates.

Figure 6-4. Privilege Check for Control Transfer without Gate



The processor permits a JMP or CALL directly to another segment only if one of the following privilege rules is satisfied:

1. DPL of the target is equal to CPL.

   target segment DPL = CPL

2. The conforming bit of the target code-segment descriptor is set, and the DPL of the target is less than or equal to CPL.

   target segment DPL ≤ CPL

- An executable segment whose descriptor has the conforming bit set is called a **conforming segment**.
- The conforming-segment mechanism permits sharing of procedures (codes) that may be called from various privilege levels but should execute at the privilege level of the calling procedure.
- For example, if you have a conforming code segment, it can be called from code running at a lower privilege level, but it will execute with the same privilege level as the calling code. This ensures that the code can be shared without elevating the privilege level, which could potentially lead to security vulnerabilities.

- **Most code segments are not conforming**. The basic rules of privilege above mean that, for nonconforming segments, control can be transferred without a gate only to executable segments at the same level of privilege.
- **There is a need, however, to transfer control to (numerically) smaller privilege levels; this need is met by the CALL instruction when used with call-gate descriptors**, which are explained in the next section. The JMP instruction may never transfer control to a nonconforming segment whose DPL does not equal CPL.

Restricting Control Transfer using Call Gates

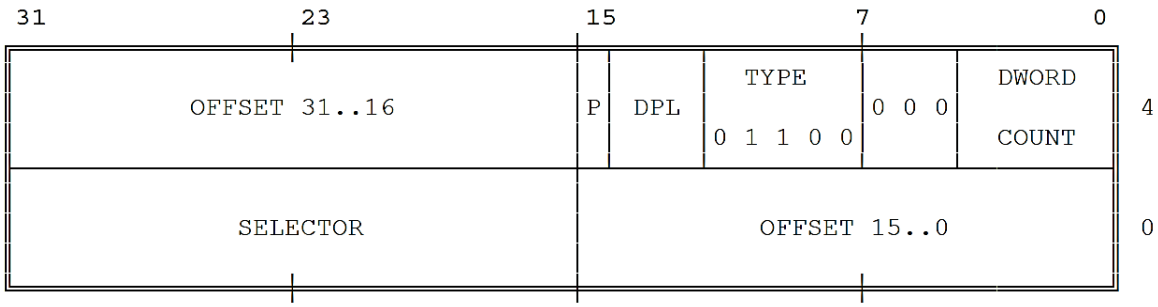
To provide protection for control transfers among executable segments at different privilege levels, the 80386 uses gate descriptors. There are four kinds of gate descriptors:

1. Call gates
2. Trap gates
3. Interrupt gates
4. Task gates

This chapter is concerned only with **call gates**. Figure 6-5 illustrates the format of a call gate.

- A call gate descriptor may reside in the GDT or in an LDT, but not in the IDT.

Figure 6-5. Format of 80386 Call Gate



A call gate has two primary functions:

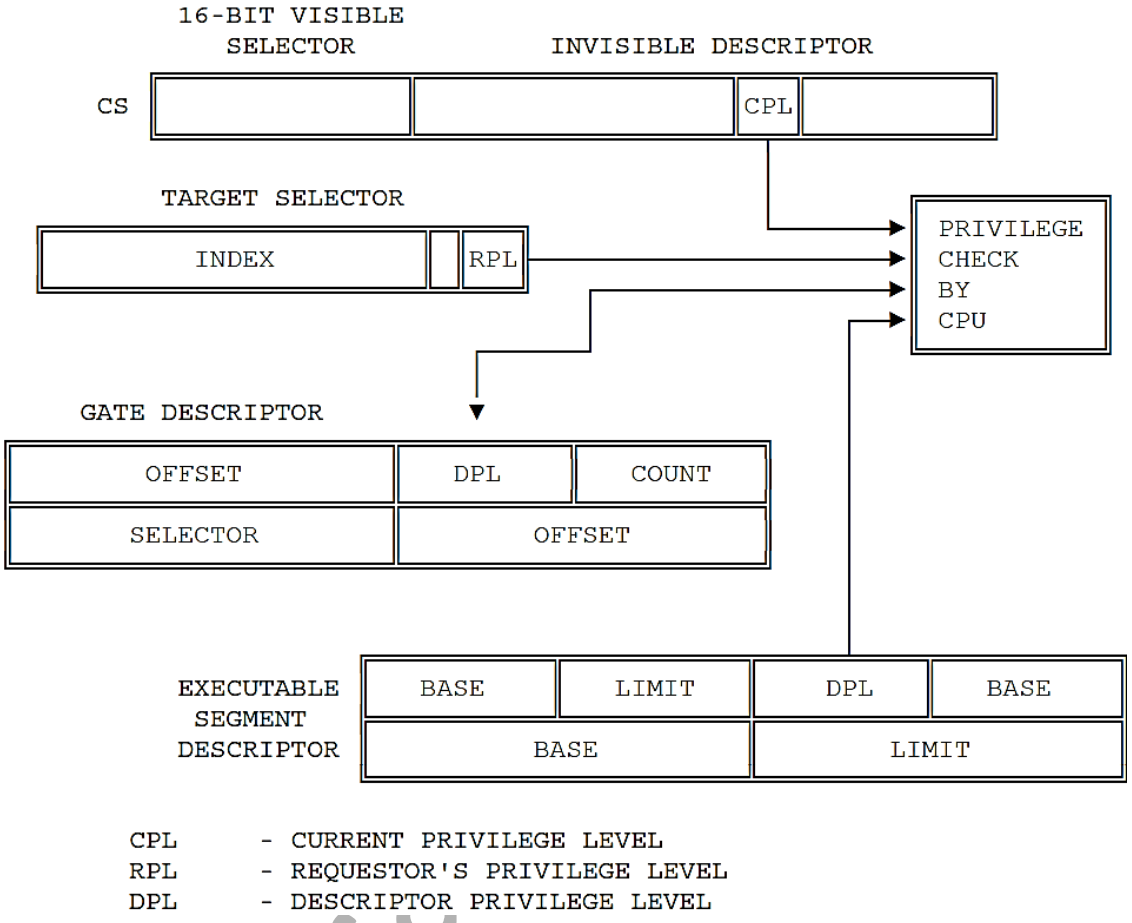
- To define an entry point of a procedure.
- To specify the privilege level of the entry point.

Call gate descriptors are used by call and jump instructions in the same manner as code segment descriptors. When the hardware recognizes that the destination selector refers to a gate descriptor, the operation of the instruction is expanded as determined by the contents of the call gate.

- Gates can be used for control transfers to **numerically smaller** privilege levels or to the same privilege level (though they are not necessary for transfers to the same level).
- Only CALL instructions can use gates to transfer to **smaller privilege levels**.
- A gate may be used by a JMP instruction only to transfer to an executable segment with the same privilege level or to a conforming segment.
- For a **JMP instruction** to a nonconforming segment, both of the following privilege rules must be satisfied; otherwise, a general protection exception results.
  - **MAX (CPL,RPL) ≤ gate DPL**
  - **target segment DPL = CPL**

- For a **CALL instruction** (or for a JMP instruction to a conforming segment), both of the following privilege rules must be satisfied; otherwise, a general protection exception results.
  - **MAX (CPL,RPL) ≤ gate DPL**
  - **target segment DPL ≤ CPL**

Figure 6-7. Privilege Check via Call Gate



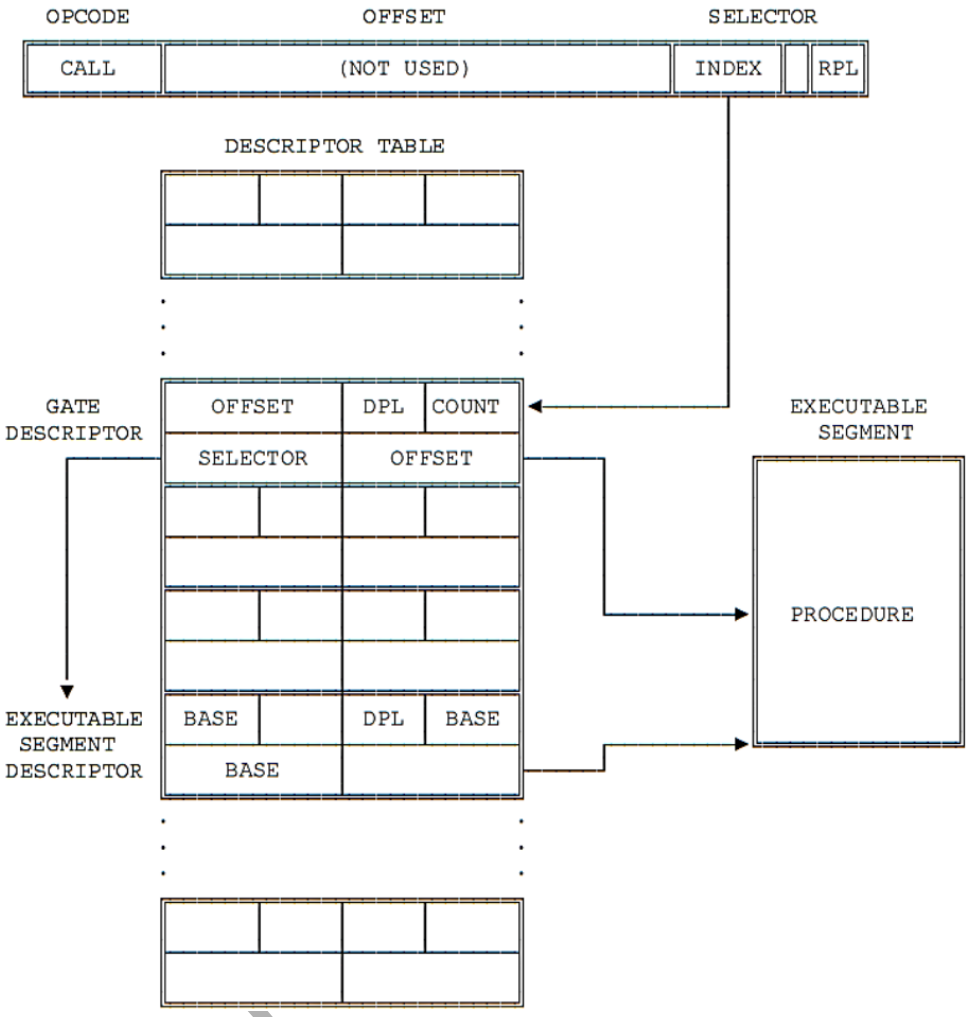
Here are the key components and features of a call gate:

- **Offset**: The offset is the address of the entry point of the procedure being called. It is divided into two parts: the lower 16 bits and the upper 16 bits.
- **Selector**: This is a segment selector that points to a segment descriptor in the Global Descriptor Table (GDT) or Local Descriptor Table (LDT). The segment descriptor provides information about the code segment where the procedure is located.
- **Parameter Count**: This field specifies the number of parameters to be copied from the caller's stack to the called procedure's stack. It is used to facilitate parameter passing between different privilege levels.
- **DPL (Descriptor Privilege Level)**: This field indicates the privilege level required to access the call gate. It ranges from 0 to 3, with 0 being the highest privilege level and 3 being the lowest.

- **P (Present) Bit**: This bit indicates whether the call gate descriptor is valid and present in memory. If the bit is set to 1, the gate is present; if it is 0, the gate is not present.

The selector and offset fields of a gate form a pointer to the entry point of a procedure. A call gate guarantees that all transitions to another segment go to a valid entry point. the selector part of the pointer selects a gate, and the offset is not used. Figure 6-6 illustrates this style of addressing.

Figure 6-6. Indirect Transfer via Call Gate



Page-Level Protection

Two kinds of protection are related to pages:

- Restriction of addressable domain.
- Type checking.

Restricting Addressable Domain

- The concept of privilege for pages is implemented by assigning each page to one of two levels:
  - **Supervisor level (U/S=0)** — for the operating system and other systems software and related data.

  - **User level (U/S=1)** — for applications procedures and data.
- The current level (U or S) is related to CPL.
- If CPL is 0, 1, or 2, the processor is executing at supervisor level.
- If CPL is 3, the processor is executing at user level.
- When the processor is executing at supervisor level, all pages are addressable (modifiable).
- But, when the processor is executing at user level, only pages that belong to the user level are addressable (modifiable).

Type Checking

At the level of page addressing, two types are defined:

1. **Read-only access (R/W=0)**
2. **Read/write access (R/W=1)**

- When the processor is executing at supervisor level, all pages are both readable and writable.
- When the processor is executing at user level, only pages that belong to user level and are marked for read/write access are writable; pages that belong to supervisor level are neither readable nor writable from user level.

Table 6-5. Combining Directory and Page Protection

Page Directory Entry		Page Table Entry		Combined Protection	
U/S	R/W	U/S	R/W	U/S	R/W
S-0	R-0	S-0	R-0	S	x
S-0	R-0	S-0	W-1	S	x
S-0	R-0	U-1	R-0	S	x
S-0	R-0	U-1	W-1	S	x
S-0	W-1	S-0	R-0	S	x
S-0	W-1	S-0	W-1	S	x
S-0	W-1	U-1	R-0	S	x
S-0	W-1	U-1	W-1	S	x
U-1	R-0	S-0	R-0	S	x
U-1	R-0	S-0	W-1	S	x
U-1	R-0	U-1	R-0	U	R
U-1	R-0	U-1	W-1	U	R
U-1	W-1	S-0	R-0	S	x
U-1	W-1	S-0	W-1	S	x
U-1	W-1	U-1	R-0	U	R
U-1	W-1	U-1	W-1	U	W

NOTE
S — Supervisor
R — Read only
U — User
W — Read and Write
x indicates that when the combined U/S attribute is S, the R/W attribute is not checked.

Combining Page & Segment Protection

- When paging is enabled, the 80386 first evaluates segment protection, then evaluates page protection.
- If the processor detects a protection violation at either the segment or the page level, the requested operation cannot proceed; a protection exception occurs instead.

Multitasking in 80386

- To provide efficient, protected multitasking, the 80386 employs several special data structures.
- It does not, however, 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:
    - Task state segment
    - Task state segment descriptor
    - Task register
    - Task gate descriptor
- With these structures the 80386 can rapidly switch execution from one task to another, saving the context of the original task so that the task can be restarted later.
- In addition to the simple task switch, the 80386 offers two other task-management features:
    1. Interrupts and exceptions can cause task switches (if needed in the system design). The processor not only switches automatically to the task that handles the interrupt or exception, but it automatically switches back to the interrupted task when the interrupt or exception has been serviced. Interrupt tasks may interrupt lower-priority interrupt tasks to any depth.
    2. With each switch to another task, the 80386 can also switch to another LDT and to another page directory. Thus, each task can have a different logical-to-linear mapping and a different linear-to-physical mapping. This is yet another protection feature, because tasks can be isolated and prevented from interfering with one another.

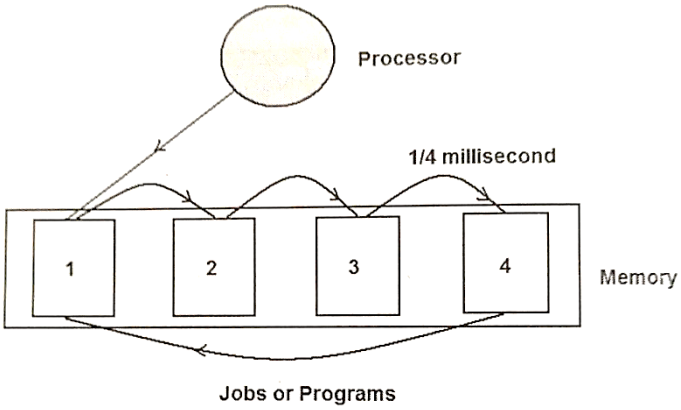
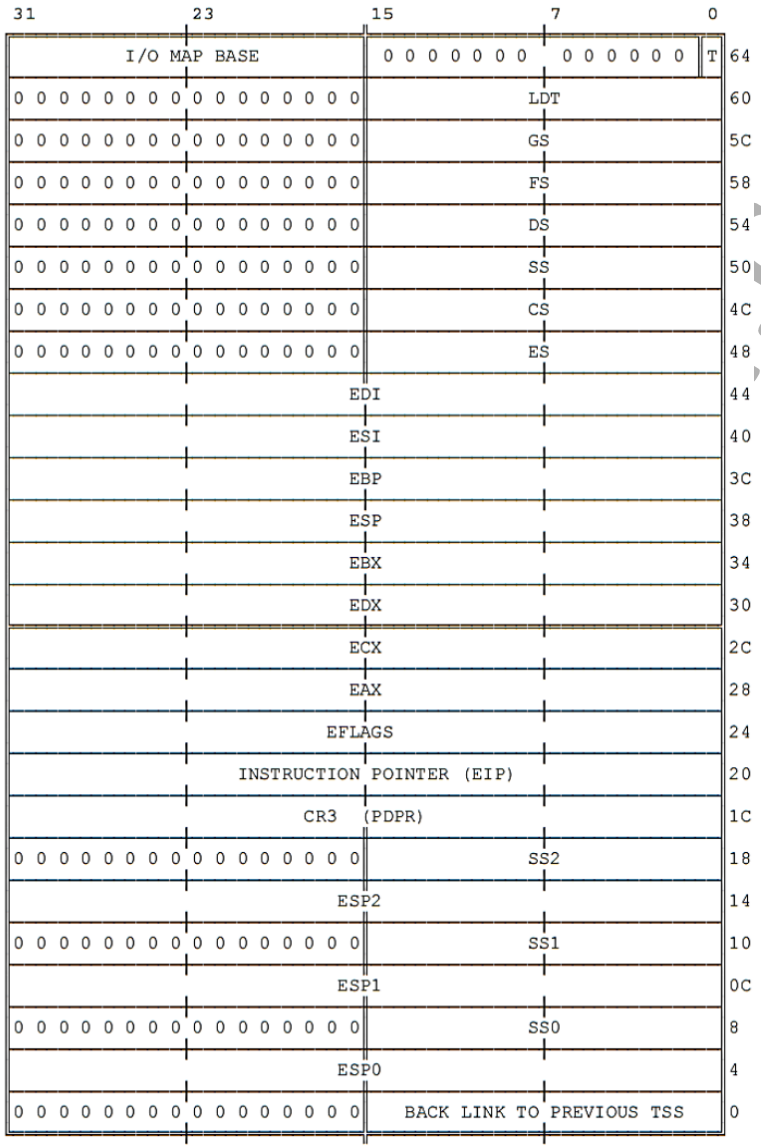


Figure 21.2: Process-based multitasking

Task State Segment (TSS)

- All the information the processor needs in order to manage a task is stored in a special type of segment, a task state segment (TSS).
- Figure 7-1 shows the format of a TSS for executing 80386 tasks.

Figure 7-1. 80386 32-Bit Task State Segment



- The fields of a TSS belong to two classes:

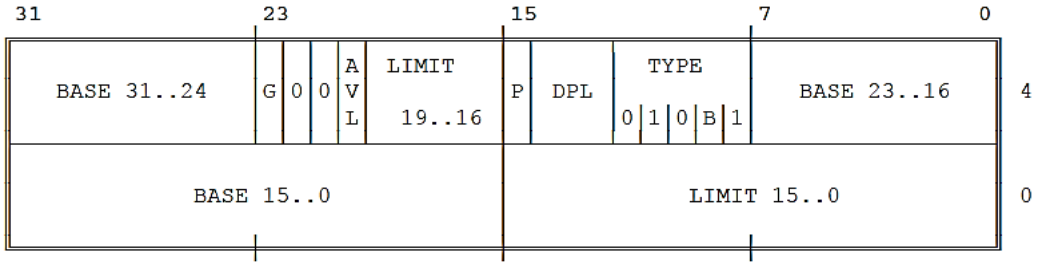
1. A dynamic set that the processor updates with each switch from the task. This set includes the fields that store:
    - The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI).
    - The segment registers (ES, CS, SS, DS, FS, GS).
    - The flags register (EFLAGS).
    - The instruction pointer (EIP).
    - The selector of the TSS of the previously executing task (updated only when a return is expected).
2. A static set that the processor reads but does not change. This set includes the fields that store:
    - The selector of the task's LDT.
    - The register (PDBR) that contains the base address of the task's page directory (read only when paging is enabled).

- Pointers to the stacks for privilege levels 0-2.
- The T-bit (debug trap bit) which causes the processor to raise a debug exception when a task switch occurs. (Refer to Chapter 12 for more information on debugging.)
- The I/O map base is used to enable or disable I/O ports. (refer to Chapter 8 for more information on the use of the I/O map).

TSS Descriptor

- The task state segment, like all other segments, is defined by a descriptor. Figure 7-2 shows the format of a TSS descriptor.

Figure 7-2. TSS Descriptor for 32-bit TSS

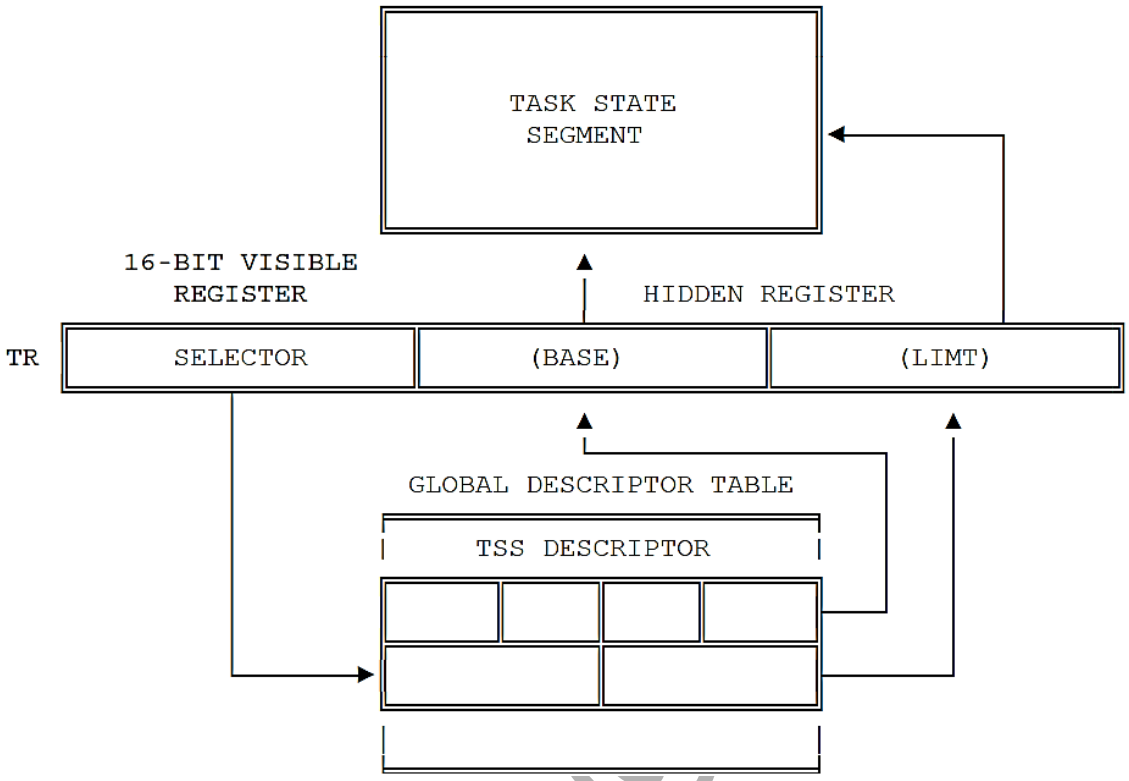


- The **B-bit** in the type field indicates whether the task is busy. A type code of 9 indicates a non-busy task; a type code of 11 indicates a busy task.
- The B-bit allows the processor to detect an attempt to switch to a task that is already busy.
- The **BASE, LIMIT, and DPL** fields and the G-bit and P-bit have functions similar to their counterparts in data-segment descriptors.
- The LIMIT field, however, must have a value equal to or greater than 103.
- An attempt to switch to a task whose TSS descriptor has a limit less than 103 causes an exception.
- A procedure that has access to a TSS descriptor can cause a task switch. In most systems the DPL fields of TSS descriptors should be set to zero, so that only trusted software has the right to perform task switching.
- Having access to a TSS-descriptor does not give a procedure the right to read or modify a TSS. Reading and modification can be accomplished only with another descriptor that redefines the TSS as a data segment. An attempt to load a TSS descriptor into any of the segment registers (CS, SS, DS, ES, FS, GS) causes an exception.
- **TSS descriptors may reside only in the GDT.** An attempt to identify a TSS with a selector that has TI=1 (indicating the current LDT) results in an exception.

Task Register

- The task register (TR) identifies the currently executing task by pointing to the TSS. Figure 7-3 shows the path by which the processor accesses the current TSS.

Figure 7-3. Task Register

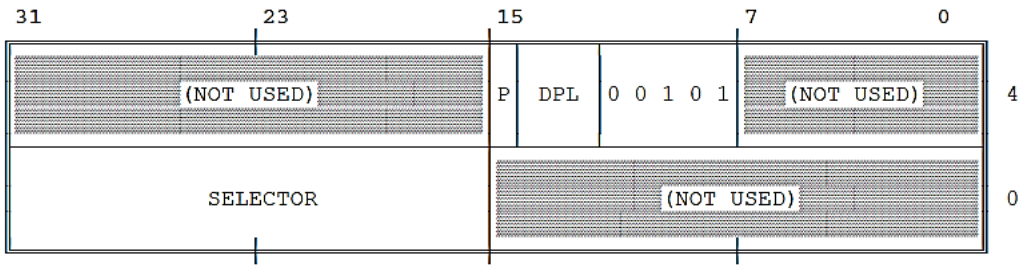


- The task register has both a "visible" portion (i.e., can be read and changed by instructions) and an "invisible" portion (maintained by the processor to correspond to the visible portion; cannot be read by any instruction).
- The selector in the visible portion selects a TSS descriptor in the GDT.
- The processor uses the invisible portion to cache the base and limit values from the TSS descriptor.
- Holding the base and limit in a register makes execution of the task more efficient, because the processor does not need to repeatedly fetch these values from memory when it references the TSS of the current task.
- The instructions LTR and STR are used to modify and read the visible portion of the task register. Both instructions take one operand, a 16-bit selector located in memory or in a general register.
- **LTR (Load task register)** loads the visible portion of the task register with the selector operand, which must select a TSS descriptor in the GDT. LTR also loads the invisible portion with information from the TSS descriptor selected by the operand.
- **LTR is a privileged instruction;** it may be executed only when CPL is zero. LTR is generally used during system initialization to give an initial value to the task register; thereafter, the contents of TR are changed by task switch operations.
- **STR (Store task register)** stores the visible portion of the task register in a general register or memory word. STR is not privileged.

Task Gate Descriptor

- A task gate descriptor provides an indirect, protected reference to a TSS.
- Figure 7-4 illustrates the format of a task gate.

Figure 7-4. Task Gate Descriptor



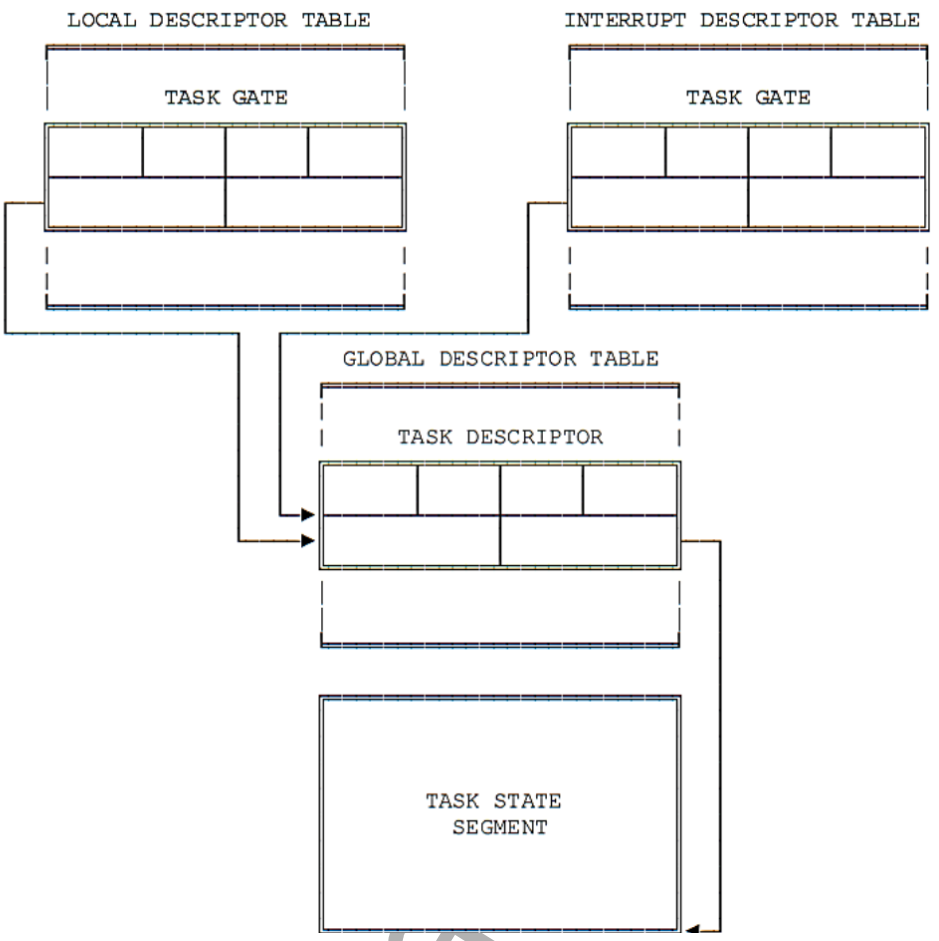
- The SELECTOR field of a task gate must refer to a TSS descriptor. **The value of the RPL in this selector is not used by the processor.**
- The DPL field of a task gate controls the right to use the descriptor to cause a task switch. A procedure may not select a task gate descriptor unless the maximum of the selector's RPL and the CPL of the procedure is numerically less than or equal to the DPL of the descriptor.
- This constraint prevents untrusted procedures from causing a task switch. **(Note that when a task gate is used, the DPL of the target TSS descriptor is not used for privilege checking.)**
- A procedure that has access to a task gate has the power to cause a task switch, just as a procedure that has access to a TSS descriptor.

The 80386 has task gates in addition to TSS descriptors to satisfy three needs:

1. The need for a task to have a single busy bit. Because the busy-bit is stored in the TSS descriptor, each task should have only one such descriptor. There may, however, be several task gates that select the single TSS descriptor.
2. The need to provide selective access to tasks. Task gates fulfil this need, because they can reside in LDTs and can have a DPL that is different from the TSS descriptor's DPL. A procedure that does not have sufficient privilege to use the TSS descriptor in the GDT (which usually has a DPL of 0) can still switch to another task if it has access to a **task gate for that task in its LDT**. With task gates, systems software can limit the right to cause task switches to specific tasks.
3. The need for an interrupt or exception to cause a task switch. Task gates may also reside in the IDT, making it possible for interrupts and exceptions to cause task switching. When interrupt or exception vectors to an IDT entry that contains a task gate, the 80386 switches to the indicated task. Thus, all tasks in the system can benefit from the protection afforded by isolation from interrupt tasks.

Figure 7-5 illustrates how both a task gate in an LDT and a task gate in the IDT can identify the same task.

Figure 7-5. Task Gate Indirectly Identifies Task



Task Switching

A task switching operation involves these steps:

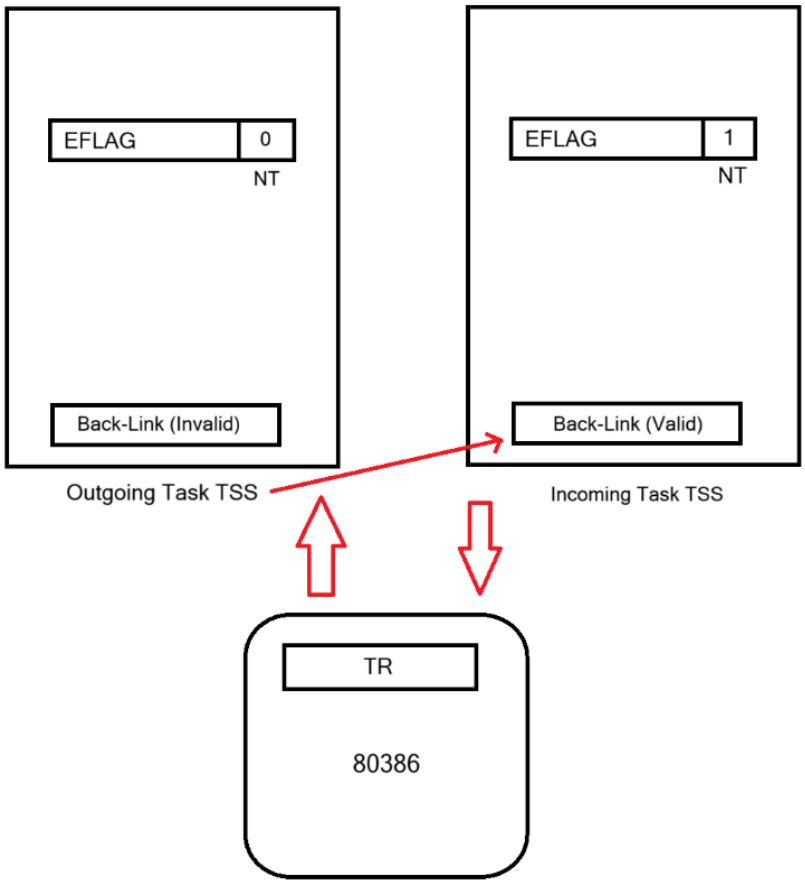
1. Checking that the current task is allowed to switch to the designated task. Data-access privilege rules apply in the case of JMP or CALL instructions. The DPL of the TSS descriptor or task gate must be less than or equal to the maximum of CPL and the RPL of the gate selector. Exceptions, interrupts, and IRETs are permitted to switch tasks regardless of the DPL of the target task gate or TSS descriptor.
2. Checking that the TSS descriptor of the new task is marked present and has a valid limit. Any errors up to this point occur in the context of the outgoing task. Errors are restartable and can be handled in a way that is transparent to applications procedures.
3. Saving the state of the current task. The processor finds the base address of the current TSS cached in the task register. It copies the registers into the current TSS (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, SS, DS, FS, GS, and the flag register). The EIP field of the TSS points to the instruction after the one that caused the task switch.
4. Loading the task register with the selector of the incoming task's TSS descriptor, marking the incoming task's TSS descriptor as busy, and setting the TS (task switched) bit of the MSW. The selector is either the operand of a control transfer instruction or is taken from a task gate.
5. Loading the incoming task's state from its TSS and resuming execution. The registers loaded are the LDT register; the flag register; the general registers EIP, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; the segment registers ES, CS, SS, DS, FS, and GS; and PDBR. Any errors detected in this

step occur in the context of the incoming task. To an exception handler, it appears that the first instruction of the new task has not yet executed.

The privilege level at which execution resumes in the incoming task is neither restricted nor affected by the privilege level at which the outgoing task was executing. Because the tasks are isolated by their separate address spaces and TSSs and because privilege rules can be used to prevent improper access to a TSS, no privilege rules are needed to constrain the relation between the CPLs of the tasks. The new task begins executing at the privilege level indicated by the RPL of the CS selector value that is loaded from the TSS.

Task Linking

- The back-link field of the TSS and the NT (nested task) bit of the flag word together allow the 80386 to automatically return to a task that CALLed another task or was interrupted by another task.
- When a CALL instruction, an interrupt instruction, an external interrupt, or an exception causes a switch to a new task, the 80386 automatically fills the back-link of the new TSS with the selector of the outgoing task's TSS and, at the same time, sets the NT bit in the new task's flag register.
- The NT flag indicates whether the back-link field is valid.
- The new task releases control by executing an IRET instruction. When interpreting an IRET, the 80386 examines the NT flag.
- If NT is set, the 80386 switches back to the task selected by the back-link field. Table 7-2 summarizes the uses of these fields.



Busy Bit

The B-bit (busy bit) of the TSS descriptor ensures the integrity of the back-link. A chain of back-links may grow to any length as interrupt tasks interrupt other interrupt tasks or as called tasks call other tasks. The busy bit ensures that the CPU can detect any attempt to create a loop. A loop would indicate an attempt to re-enter a task that is already busy; however, the TSS is not a re-entrable resource.

The processor uses the busy bit as follows:

- When switching **to a task**, the processor automatically sets the busy bit of the new task.
- When switching **from a task**, the processor automatically clears the busy bit of the old task if that task is not to be placed on the back-link chain (i.e., the instruction causing the task switch is JMP or IRET). If the task is placed on the back-link chain, its busy bit remains set.
- When switching to a task, the processor signals an exception if the busy bit of the new task is already set.

By these actions, the processor prevents a task from switching to itself or to any task that is on a back-link chain, thereby preventing invalid re-entry into a task.

Table 7-2. Effect of Task Switch on BUSY, NT, and Back-Link

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 of incoming task	Unchanged	Set to outgoing TSS selector	Unchanged
Back-link of outgoing task	Unchanged	Unchanged	Unchanged

Task Address Space

The LDT selector and PDBR fields of the TSS give software systems designers flexibility in utilization of segment and page mapping features of the 80386. By appropriate choice of the segment and page mappings for each task, tasks may share address spaces, may have address spaces that are largely distinct from one another, or may have any degree of sharing between these two extremes.

Task Linear-to-Physical Space Mapping

The choices for arranging the linear-to-physical mappings of tasks fall into two general classes:

1. One linear-to-physical mapping shared among all tasks.
   When paging is not enabled, this is the only possibility. Without page tables, all linear addresses map to the same physical addresses. When paging is enabled, this style of linear-to-physical mapping results from using one page directory for all tasks. The linear space utilized

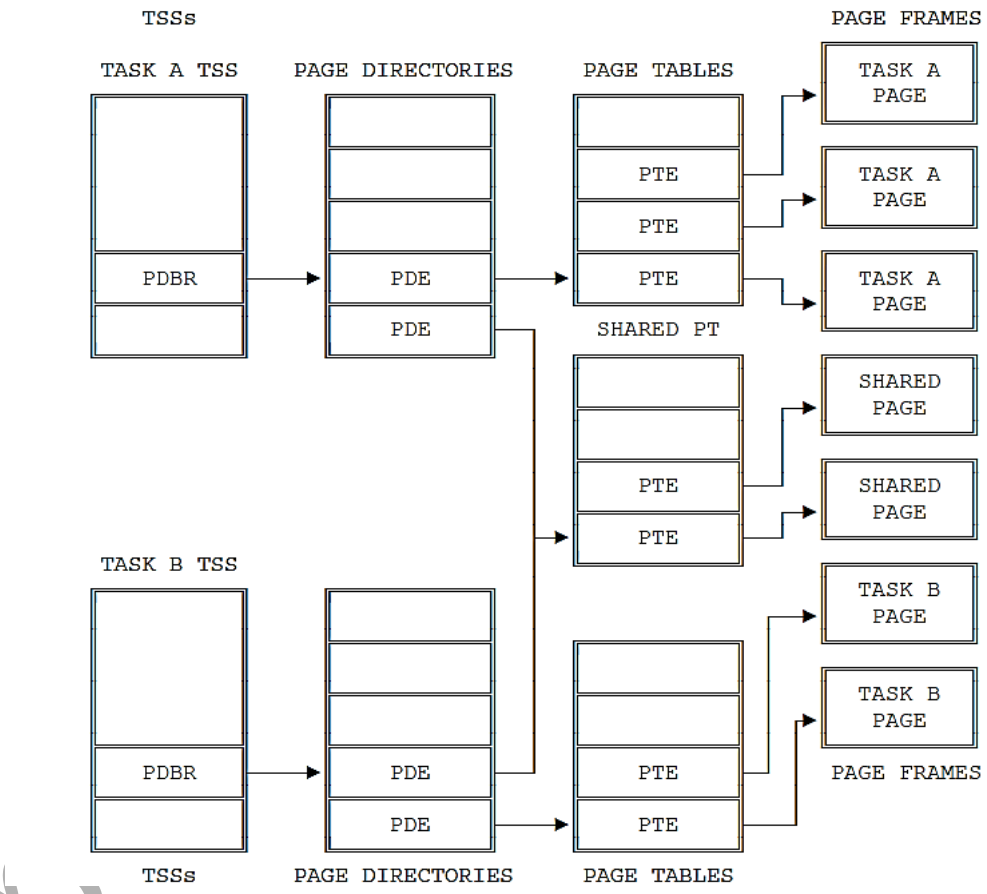
may exceed the physical space available if the operating system also implements **page-level virtual memory**.

2. Several partially overlapping linear-to-physical mappings.
   This style is implemented by using a different page directory for each task. Because the PDBR (page directory base register) is loaded from the TSS with each task switch, each task may have a different page directory.

In theory, the linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share any physical addresses.

In practice, some portion of the linear address spaces of all tasks must map to the same physical addresses. The task state segments must lie in a common space so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear space mapped by the GDT should also be mapped to a common physical space; otherwise, the purpose of the GDT is defeated. Figure 7-6 shows how the linear spaces of two tasks can overlap in the physical space by sharing page tables.

Figure 7-6. Partially-Overlapping Linear Spaces



Task Logical Address Space

By itself, a common linear-to-physical space mapping does not enable sharing of data among tasks. To share data, tasks must also have a common logical-to-linear space mapping; i.e., they must also have access to descriptors that point into a shared linear address space.

There are three ways to create common logical-to-physical address-space mappings:

1. **Via the GDT.** All tasks have access to the descriptors in the GDT. If those descriptors point into a linear-address space that is mapped to a common physical-address space for all tasks, then the tasks can share data and instructions.
2. **By sharing LDTs.** Two or more tasks can use the same LDT if the LDT selectors in their TSSs select the same LDT segment. Those LDT-resident descriptors that point into a linear space that is mapped to a common physical space permit the tasks to share physical memory. This method of sharing is more selective than sharing by the GDT; the sharing can be limited to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared areas.
3. **By descriptor aliases in LDTs.** It is possible for certain descriptors of different LDTs to point to the same linear address space. If that linear address space is mapped to the same physical space by the page mapping of the tasks involved, these descriptors permit the tasks to share the common space. Such descriptors are commonly called "aliases". This method of sharing is even more selective than the prior two; other descriptors in the LDTs may point to distinct linear addresses or to linear addresses that are not shared.

Input / Output

This chapter presents the I/O features of the 80386 from the following perspectives:

- Methods of addressing I/O ports
- Instructions that cause I/O operations
- Protection as it applies to the use of I/O instructions and I/O port

I/O Addressing

The 80386 allows input/output to be performed in either of two ways:

- By means of a separate **I/O address space** (using specific I/O instructions)
- By means of **memory-mapped I/O** (using general-purpose operand manipulation instructions).

I/O Address Space

- I/O address space uses 16-bit addressing that means it can address 2^{16} ports.
- The I/O address space consists of 65536 individually addressable 8-bit ports;
- any two consecutive 8-bit ports can be treated as a 16-bit port;
- and four consecutive 8-bit ports can be treated as a 32-bit port.
- Thus, the I/O address space can accommodate up to 64K 8-bit ports, up to 32K 16-bit ports, or up to 16K 32-bit ports.
- The program can specify the address of the port in two ways. Using an **immediate byte** constant, the program can specify:
  - 256 8-bit ports numbered 0 through 255.
  - 128 16-bit ports numbered 0, 2, 4, . . . , 252, 254.
  - 64 32-bit ports numbered 0, 4, 8, . . . , 248, 252.

```
Example 1:
      IN    AL,99H       ;BRING A BYTE INTO AL FROM PORT 99H

Example 2:
      IN    AX,78H       ;BRING A WORD FROM PORT ADDRESSES 78H
                         ;AND 79H.  THE BYTE FROM PORT 78 GOES
                         ;TO AL AND BYTE FROM PORT 79H TO AH.
```

- Using a value in **DX**, the program can specify:
  - 8-bit ports numbered 0 through 65535
  - 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534
  - 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532

```
Example 3:
      MOV   DX,481H      ;DX=481H
      IN          AL,DX        ;BRING THE BYTE TO AL FROM THE
PORT
                               ;WHOSE ADDRESS IS POINTED BY DX

Example 4:
      IN    AX,DX        ;BRING A WORD FROM PORT ADDRESS POINTED
                         ;TO BY DX. THE BYTE FROM PORT AT
                         ;DX GOES TO AL AND BYTE FROM PORT AT
```

- The 80386 can transfer 32, 16, or 8 bits at a time to a device located in the I/O space. Like doublewords in memory, 32-bit ports should be aligned at addresses evenly divisible by four

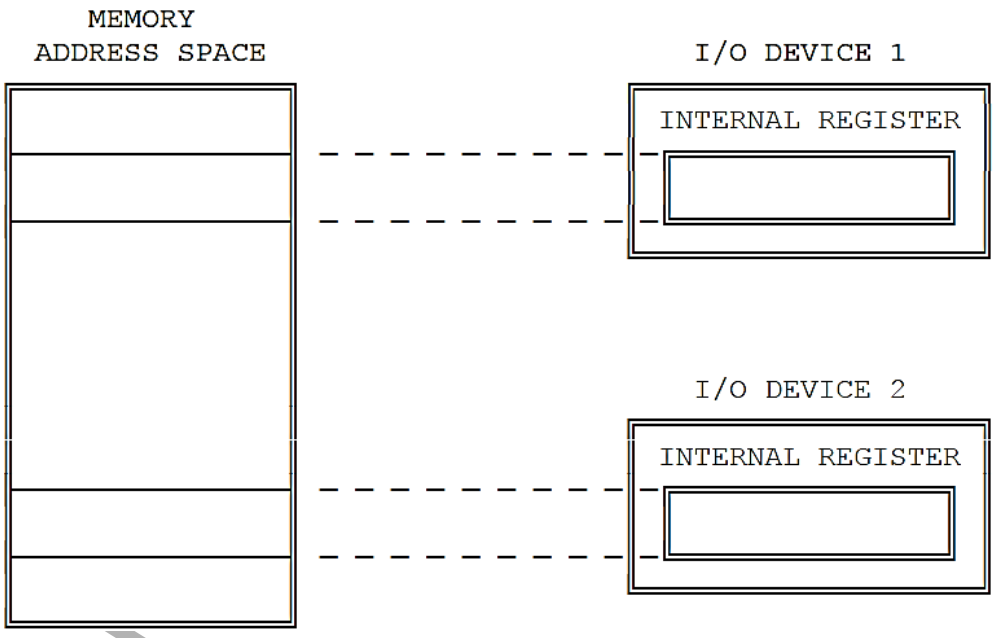
so that the 32 bits can be transferred in a single bus access. An 8-bit port may be located at either an even or odd address.

- The instructions IN and OUT move data between a register and a port in the I/O address space.
- The instructions INS and OUTS move strings of data between the memory address space and ports in the I/O address space.

Memory-Mapped I/O

- Memory-mapped I/O provides additional programming flexibility.
- Any instruction that references memory may be used to access an I/O port located in the memory space.
  For example, the MOV instruction can transfer data between any register and a port; and the AND, OR, and TEST instructions may be used to manipulate bits in the internal registers of a device (see Figure 8-1).
- Memory-mapped I/O performed via the full instruction set maintains the full complement of addressing modes for selecting the desired I/O device (e.g., direct address, indirect address, base register, index register, scaling).
- Memory-mapped I/O, like any other memory reference, is subject to access protection and control when executing in protected mode.

Figure 8-1. Memory-Mapped I/O



I/O Instructions

Register I/O Instructions

- The I/O instructions IN and OUT are provided to move data between I/O port and the **EAX** (32-bit I/O), the AX (16-bit I/O), or AL (8-bit I/O) general registers.
- IN and OUT instructions address I/O ports either directly, with the address of one of up to 256 port addresses coded in the instruction, or indirectly via the DX register to one of up to 64K port addresses.
- **IN** (Input from Port) transfers a byte, word, or doubleword from an **input port to AL, AX, or EAX**.

- If a program specifies AL with the IN instruction, the processor transfers 8 bits from the selected port to AL.
- If a program specifies AX with the IN instruction, the processor transfers 16 bits from the port to AX.
- If a program specifies EAX with the IN instruction, the processor transfers 32 bits from the port to EAX.

- **OUT** (Output to Port) transfers a byte, word, or doubleword **to an output port from AL, AX, or EAX**.
  - The program can specify the number of the port using the same methods as the IN instruction.

```
Example 1:
      OUT    68H,AL       ;SEND OUT A BYTE FROM AL TO PORT 68H

             or

      OUT    34H,AX       ;SEND OUT A WORD FROM AX TO PORT
                          ;ADDRESSES 34H AND 35H. THE BYTE
                          ;FROM AL GOES TO PORT 34H AND
                          ;THE BYTE FROM AH GOES TO PORT 35H

Example 2:
      MOV    DX,64B1H     ;DX=64B1H
      OUT    DX,AL        ;SENT OUT THE BYTE IN AL TO THE PORT
                          ;WHOSE ADDRESS IS POINTED TO BY DX
             or

      OUT    DX,AX        ;SEND OUT A WORD FROM AX TO PORT
                          ;ADDRESS POINTED TO DX.   THE BYTE
                          ;FROM AL GOES TO PORT DX AND AND BYTE
                          ;FROM AH GOES TO PORT DX+1.
```

Block I/O Instructions

- The block (or string) I/O instructions INS and OUTS move blocks of data between I/O ports and memory space. **Block I/O instructions use the DX register** to specify the address of a port in the I/O address space.
- INS and OUTS use **DX** to specify:
  - 8-bit ports numbered 0 through 65535
  - 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534
  - 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532
- Block I/O instructions use either SI or DI to designate the source or destination memory address.
- For each transfer, SI or DI are automatically either incremented or decremented as specified by the direction bit in the flags register.
- **INS** (Input String from Port) transfers a byte or a word string element from an input port to memory.
  - The mnemonics INSB, INSW, and INSD are variants that explicitly specify the size of the operand.
  - If a program specifies **INSB**, the processor transfers 8 bits from the selected port to the memory location indicated by ES:EDI.

  - If a program specifies **INSW**, the processor transfers 16 bits from the port to the memory location indicated by ES:EDI.
  - If a program specifies **INSD**, the processor transfers 32 bits from the port to the memory location indicated by ES:EDI.
  - **The destination segment register choice (ES) cannot be changed for the INS instruction.**
  - Combined with the **REP** prefix, INS moves a block of information from an input port to a series of consecutive memory locations.
- **OUTS** (Output String to Port) transfers a byte, word, or doubleword string element to an output port from memory.
  - The mnemonics OUTSB, OUTSW, and OUTSD are variants that explicitly specify the size of the operand. If a program specifies OUTSB, the processor transfers 8 bits from the memory location indicated by ES:EDI to the selected port.
  - If a program specifies OUTSW, the processor transfers 16 bits from the memory location indicated by ES:EDI to the selected port.
  - If a program specifies OUTSD, the processor transfers 32 bits from the memory location indicated by ES:EDI to the selected port.
  - Combined with the **REP** prefix, OUTS moves a block of information from a series of consecutive memory locations indicated by DS:ESI to an output port.

Protection and I/O

Two mechanisms provide protection for I/O functions:

1. The IOPL field in the EFLAGS register defines the right to use I/O-related instructions.
2. The I/O permission bit map of a 80386 TSS segment defines the right to use ports in the I/O address space.

These mechanisms operate only in protected mode, including virtual 8086 mode; they do not operate in real mode. In real mode, there is no protection.

I/O Privilege Level

- Instructions that deal with I/O need to be restricted but also need to be executed by procedures executing at privilege levels other than zero.
- For this reason, the processor uses two bits of the flags register to store the I/O privilege level (IOPL).
- The IOPL defines the privilege level needed to execute I/O-related instructions.
- **The following instructions can be executed only if CPL ≤ IOPL:**
  - IN — Input
  - INS — Input String
  - OUT — Output
  - OUTS — Output String
  - CLI — Clear Interrupt-Enable Flag
  - STI — Set Interrupt-Enable
- These instructions are called **"sensitive" instructions**, because they are sensitive to IOPL.
- To use sensitive instructions, a procedure must execute at a privilege level at least as privileged as that specified by the IOPL (CPL ≤ IOPL). Any attempt by a less privileged procedure to use a sensitive instruction results in a general protection exception.

- Because each task has its own unique copy of the flags register, each task can have a different IOPL.
- A task whose primary function is to perform I/O (a device driver) can benefit from having an IOPL of three, thereby permitting all procedures of the task to perform I/O. Other tasks typically have IOPL set to zero or one, reserving the right to perform I/O instructions for the most privileged procedures.
- A task can change IOPL only with the POPF instruction; however, such changes are privileged. No procedure may alter IOPL (the I/O privilege level in the flag register) unless the procedure is executing at **privilege level 0**. An attempt by a less privileged procedure to alter IOPL does not result in an exception; IOPL simply remains unaltered.
- The POPF instruction may be used in addition to CLI and STI to alter the interrupt-enable flag (IF); however, changes to IF by POPF are IOPL-sensitive.
- A procedure may alter IF with a POPF instruction only when executing at a level that is at least as privileged as IOPL. An attempt by a less privileged procedure to alter IF in this manner does not result in an exception; IF simply remains unaltered.

Interrupts & Exceptions

The 80386 has two mechanisms for interrupting program execution:

1. **Exceptions** are synchronous events that are the responses of the CPU to certain conditions detected during the execution of an instruction.
2. **Interrupts** are asynchronous events typically triggered by external devices needing attention.

Interrupts and exceptions are alike in that both cause the processor to temporarily suspend its present program execution in order to execute a program of higher priority. The major distinction between these two kinds of interrupts is their origin. An exception is always reproducible by re-executing with the program and data that caused the exception, whereas an interrupt is generally independent of the currently executing program.

There are two sources for external interrupts and two sources for exceptions:

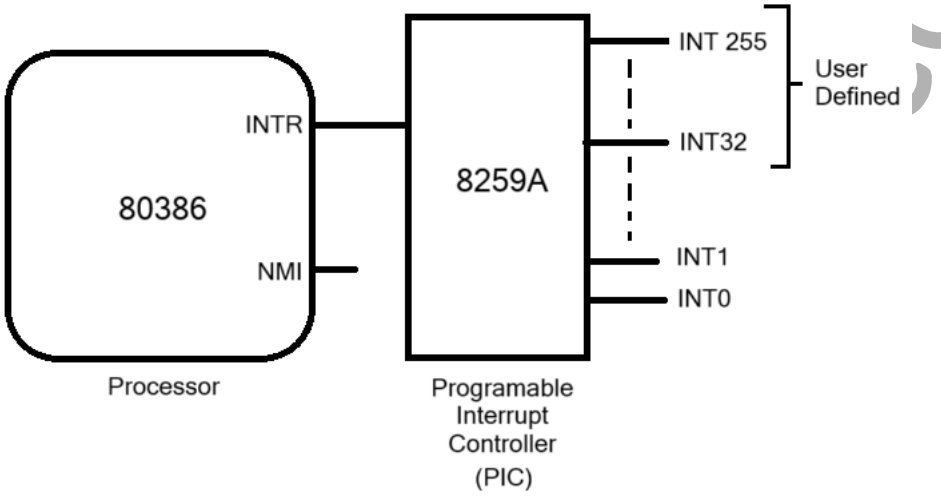
1. **Interrupts**
    - **Maskable interrupts:** which are signalled via the INTR pin.
    - **Non-maskable interrupts:** which are signalled via the NMI (Non-Maskable Interrupt) pin.
    - Also, these interrupts are known as **Hardware Interrupts.**

2. **Exceptions**
    - **Processor detected:** These are further classified as **faults, traps, and aborts**.
    - **Programmed:** The instructions INTO, INT 3, **INT 21H**, INT n, and BOUND can trigger exceptions. These instructions are often called **"software interrupts"**, but the processor handles them as exceptions.

Identifying Interrupts

- The processor associates an identifying number with each different type of interrupt or exception.
- The NMI and the exceptions recognized by the processor are assigned predetermined identifiers in the range 0 through 31. Not all of these numbers are currently used by the 80386; unassigned identifiers in this range are reserved by Intel for possible future expansion.

- The identifiers of the maskable interrupts are determined by external interrupt controllers (such as Intel's 8259A Programmable Interrupt Controller (PIC)) and communicated to the processor during the processor's interrupt-acknowledge sequence. The numbers assigned by an **8259A PIC** can be specified by software. Any numbers in the range 32 through 255 can be used. Table 9-1 shows the assignment of interrupt and exception identifiers.



Exceptions are classified as faults, traps, or aborts depending on the way they are reported and whether restart of the instruction that caused the exception is supported.

- **Faults**: Faults are exceptions that are reported "before" the instruction causing the exception. Faults are either detected before the instruction begins to execute, or during execution of the instruction. If detected during the instruction, the fault is reported with the machine restored to a state that permits the instruction to be restarted.
- **Traps**: A trap is an exception that is reported at the instruction boundary immediately after the instruction in which the exception was detected.
- **Aborts**: An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

Table 9-1. Interrupt and Exception ID Assignments

Identifier	Description
0	Divide error
1	Debug exceptions
2	Nonmaskable interrupt
3	Breakpoint (one-byte INT 3 instruction)
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
14	Page fault
15	(reserved)
16	Coprecessor error
17-31	(reserved)
32-255	Available for external interrupts via INTR pin

----------EXTRA INFORMATION (NOT PART OF SYLLABUS)----------

In the **Intel 80386** processor, exceptions are classified into **faults, traps, and aborts**, each with distinct characteristics:

- **Faults**: These occur **before** the instruction causing the exception. The system can recover and restart the instruction.
    - Example: **Divide Error Fault** – Happens when a **DIV** or **IDIV** instruction encounters a **zero divisor**.
    - Example: **Invalid Opcode Fault** – Occurs when the processor encounters an **undefined instruction**.
- **Traps**: These are reported **after** the instruction execution, allowing debugging or monitoring.
    - Example: **Breakpoint Trap** – Triggered by the **INT 3** instruction, commonly used in debuggers to halt execution.
    - Example: **Overflow Trap** – Occurs when an **INTO** instruction detects an overflow condition.
- **Aborts**: These are severe errors that prevent precise identification of the faulty instruction and do not allow recovery.
    - Example: **Double Fault Abort** – Happens when an exception occurs while handling another exception, leading to system failure.
    - Example: **Machine Check Abort** – Caused by **hardware errors**, such as memory corruption or CPU faults.

Enabling and Disabling Interrupts

The processor services interrupts and exceptions only between the end of one instruction and the beginning of the next. When the repeat prefix is used to repeat a string instruction, interrupts and

exceptions may occur between repetitions. Thus, operations on long strings do not delay interrupt response.

Certain conditions and flag settings cause the processor to inhibit certain interrupts and exceptions at instruction boundaries.

NMI Masks Further NMIs

While an NMI handler is executing, the processor ignores further interrupt signals at the NMI pin until the next IRET instruction is executed.

IF Masks INTR

- The **IF (interrupt-enable flag)** controls the acceptance of external interrupts signalled via the INTR pin.
- When **IF=0**, INTR interrupts are inhibited; when **IF=1**, INTR interrupts are enabled. As with the other flag bits, the processor clears IF in response to a RESET signal. The instructions CLI and STI alter the setting of IF.
- **CLI (Clear Interrupt-Enable Flag)** and **STI (Set Interrupt-Enable Flag)** explicitly alter IF (bit 9 in the flag register). These instructions may be executed only if CPL ≤ IOPL. A protection exception occurs if they are executed when CPL > IOPL.

The IF is also affected implicitly by the following operations:

- The instruction PUSHF stores all flags, including IF, in the stack where they can be examined.
- Task switches and the instructions POPF and IRET load the flags register; therefore, they can be used to modify IF.
- Interrupts through interrupt gates automatically reset IF, disabling interrupts. (Interrupt gates are explained later in this chapter.)

RF Masks Debug Faults

The **RF bit in EFLAGS** controls the recognition of debug faults. This permits debug faults to be raised for a given instruction at most once, no matter how many times the instruction is restarted. (Refer to Chapter 12 for more information on debugging.)

MOV or POP to SS Masks Some Interrupts and Exceptions

Software that needs to change stack segments often uses a pair of instructions; for example:

MOV SS, AX

MOV ESP, StackTop

- If an interrupt or exception is processed after SS has been changed but before ESP has received the corresponding change, the two parts of the stack pointer SS:ESP are inconsistent for the duration of the interrupt handler or exception handler.
- To prevent this situation, the 80386, after both a MOV to SS and a POP to SS instruction, inhibits NMI, INTR, debug exceptions, and single-step traps at the instruction boundary following the instruction that changes SS. Some exceptions may still occur; namely, page fault and general protection fault. **Always use the 80386 LSS instruction, and the problem will not occur.**

Priority Among Simultaneous Interrupts and Exceptions

- If more than one interrupt or exception is pending at an instruction boundary, the processor services one of them at a time.

- The priority among classes of interrupt and exception sources is shown in Table 9-2.
- The processor first services a pending interrupt or exception from the class that has the highest priority, transferring control to the first instruction of the interrupt handler.
- Lower priority **exceptions are discarded**; lower priority **interrupts are held pending**.
- Discarded **exceptions will be rediscovered** when the interrupt handler returns control to the point of interruption.

Table 9-2. Priority Among Simultaneous Interrupts and Exceptions

Priority	Class of Interrupt or Exception
HIGHEST	Faults except debug faults
	Trap instructions INTO, INT n, INT 3
	Debug traps for this instruction
	Debug faults for next instruction
	NMI interrupt
LOWEST	INTR interrupt

Interrupt Descriptor Table

- The interrupt descriptor table (IDT) associates each interrupt or exception identifier with a descriptor for the instructions that service the associated event.
- Like the GDT and LDTs, the IDT is an array of 8-byte descriptors.
- Unlike the GDT and LDTs, the first entry of the IDT may contain a descriptor.
- To form an index into the IDT, the processor **multiplies the interrupt or exception identifier by eight**.
- Because there are only 256 identifiers, the IDT need not contain more than 256 descriptors. It can contain fewer than 256 entries; entries are required only for interrupt identifiers that are actually used.

The IDT may reside anywhere in physical memory. As Figure 9-1 shows, the processor locates the IDT by means of the IDT register (IDTR). The instructions LIDT and SIDT operate on the IDTR. Both instructions have one explicit operand: the address in memory of a 6-byte area. Figure 9-2 shows the format of this area.

- **LIDT (Load IDT register)** loads the IDT register with the linear base address and limit values contained in the memory operand. This instruction can be **executed only when the CPL is zero**. It is normally used by the initialization logic of an operating system when creating an IDT. An operating system may also use it to change from one IDT to another.
- **SIDT (Store IDT register)** copies the base and limit value stored in IDTR to a memory location. This instruction can be executed at any privilege level.

Note: There is **no concept of selector** for Interrupt Descriptor Table (IDT).

Figure 9-1. IDT Register and Table

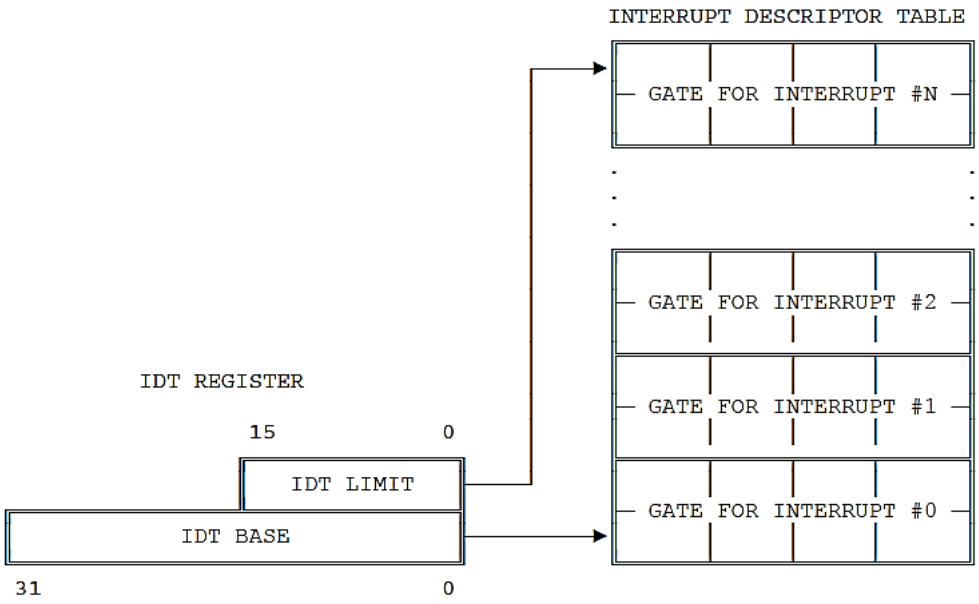
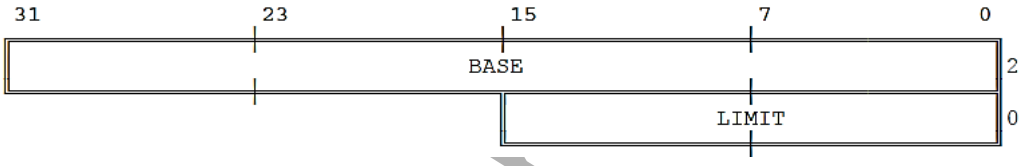


Figure 9-2. Pseudo-Descriptor Format for LIDT and SIDT



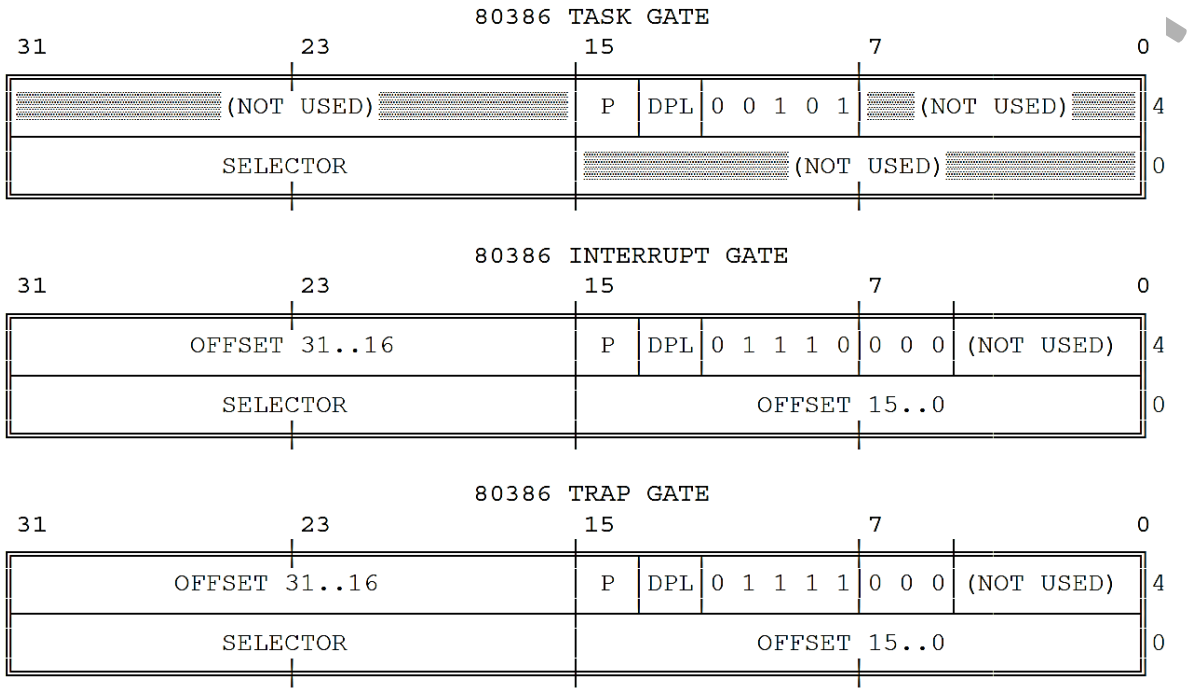
IDT Descriptors

The IDT may contain any of three kinds of descriptor:

- Task gates
- Interrupt gates
- Trap gates

Figure 9-3 illustrates the format of task gates and 80386 interrupt gates and trap gates. (The task gate in an IDT is the same as the task gate already discussed in Chapter 7.)

Figure 9-3. 80306 IDT Gate Descriptors



Interrupt Tasks and Interrupt Procedures

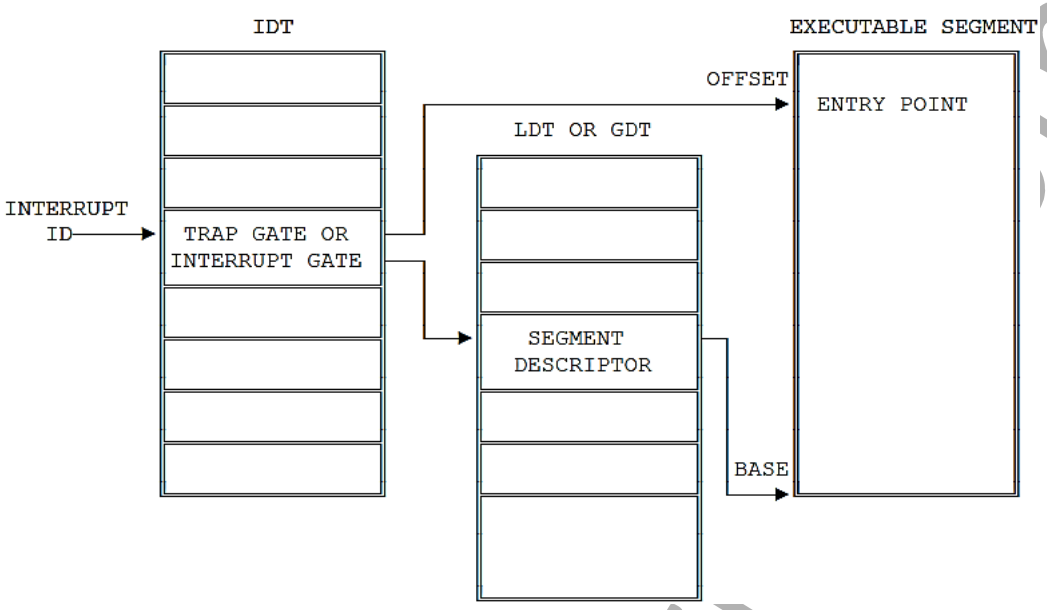
Just as a CALL instruction can call either a procedure or a task, so an interrupt or exception can "call" an interrupt handler that is either a procedure or a task. When responding to an interrupt or exception, the processor uses the interrupt or exception identifier to index a descriptor in the IDT. If the processor indexes to an interrupt gate or trap gate, it invokes the handler in a manner similar to a CALL to a call gate. If the processor finds a task gate, it causes a task switch in a manner similar to a CALL to a task gate.

Interrupt Procedures

- An interrupt gate or trap gate points indirectly to a procedure which will execute in the context of the currently executing task as illustrated by Figure 9-4.
- The selector of the gate points to an executable-segment descriptor in either the GDT or the current LDT.
- The offset field of the gate points to the beginning of the interrupt or exception handling procedure.

The 80386 invokes an interrupt or exception handling procedure in much the same manner as it CALLs a procedure; the differences are explained in the following sections.

Figure 9-4. Interrupt Vectoring for Procedures

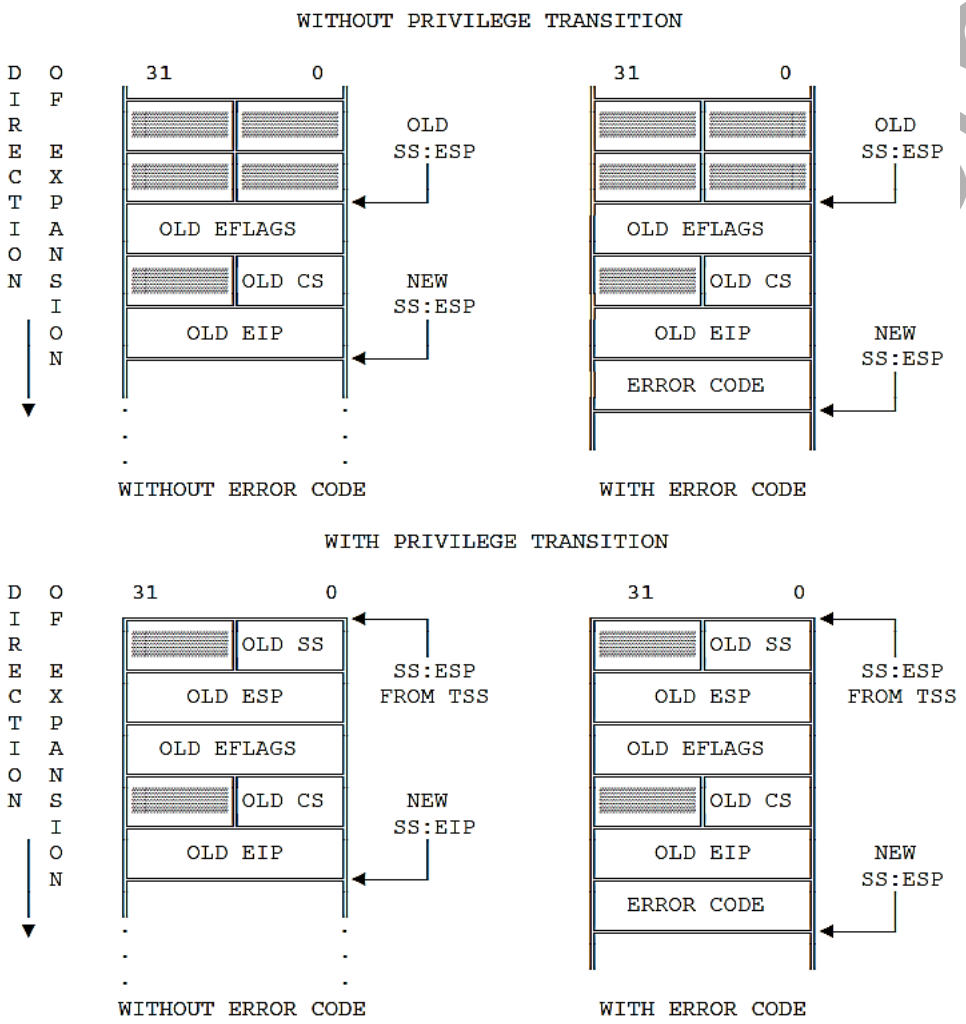


Stack of Interrupted Procedure

Just as with a control transfer due to a CALL instruction, a control transfer to an interrupt or exception handling procedure uses the stack to store the information needed for returning to the original procedure. As Figure 9-5 shows, an interrupt pushes the EFLAGS register onto the stack before the pointer to the interrupted instruction.

Certain types of exceptions also cause an **error code to be pushed on the stack**. An exception handler can use the error code to help diagnose the exception.

Figure 9-5. Stack Layout after Exception of Interrupt



Returning from an Interrupt Procedure

An interrupt procedure also differs from a normal procedure in the method of leaving the procedure. The IRET instruction is used to exit from an interrupt procedure. IRET is similar to RET except that it moves the saved flags into the EFLAGS register along with CS & EIP. The IOPL field of EFLAGS is changed only if the CPL is zero. The IF flag is changed only if **CPL ≤ IOPL.**

Flags Usage by Interrupt Procedure

Interrupts that vector through either interrupt gates or trap gates cause TF (the trap flag) to be reset after the current value of TF is saved on the stack as part of EFLAGS. By this action the processor prevents debugging activity that uses single-stepping from affecting interrupt response. A subsequent IRET instruction restores TF to the value in the EFLAGS image on the stack.

The difference between an interrupt gate and a trap gate is in the effect on IF (the interrupt-enable flag). An interrupt that vectors through an interrupt gate resets IF, thereby preventing other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores IF to the value in the EFLAGS image on the stack. An interrupt through a trap gate does not change IF.

Exception Conditions

The following sections describe each of the possible exception conditions in detail. Each description classifies the exception as a fault, trap, or abort. This classification provides information needed by systems programmers for restarting the procedure in which the exception occurred:

- **Faults:** The CS and EIP values saved when a fault is reported point to the instruction causing the fault.
- **Traps:** The CS and EIP values stored when the trap is reported point to the instruction dynamically after the instruction causing the trap. If a trap is detected during an instruction that alters program flow, the reported values of CS and EIP reflect the alteration of program flow. For example, if a trap is detected in a JMP instruction, the CS and EIP values pushed onto the stack point to the target of the JMP, not to the instruction after the JMP.
- **Aborts:** An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

Interrupt 0 — Divide Error

The divide-error fault occurs during a DIV or an IDIV instruction when the divisor is zero.

Interrupt 1 — Debug Exceptions

The processor triggers this interrupt for any of a number of conditions; whether the exception is a fault or a trap depends on the condition:

- Instruction address breakpoint fault.
- Data address breakpoint trap.
- General detect fault.
- Single-step trap.
- Task-switch breakpoint trap.

The processor does not push an error code for this exception. An exception handler can examine the debug registers to determine which condition caused the exception. Refer to Chapter 12 for more detailed information about debugging and the debug registers.

Interrupt 3 — Breakpoint

The INT 3 instruction causes this trap. The INT 3 instruction is one byte long, which makes it easy to replace an opcode in an executable segment with the breakpoint opcode. The operating system or a debugging subsystem can use a data-segment alias for an executable segment to place an INT 3 anywhere it is convenient to arrest normal execution so that some sort of special processing can be performed. Debuggers typically use breakpoints as a way of displaying registers, variables, etc., at crucial points in a task. The saved CS:EIP value points to the byte following the breakpoint. If a debugger replaces a planted breakpoint with a valid opcode, it must subtract one from the saved EIP value before returning. Refer also to Chapter 12 for more information on debugging.

Interrupt 4 — Overflow

This trap occurs when the processor encounters an INTO instruction and the OF (overflow) flag is set. Since signed arithmetic and unsigned arithmetic both use the same arithmetic instructions, the processor cannot determine which is intended and therefore does not cause overflow exceptions automatically. Instead it merely sets OF when the results, if interpreted as signed numbers, would be

out of range. When doing arithmetic on signed operands, careful programmers and compilers either test OF directly or use the INTO instruction.

Interrupt 5 — Bounds Check

This fault occurs when the processor, while executing a BOUND instruction, finds that the operand exceeds the specified limits. A program can use the BOUND instruction to check a signed array index against signed limits defined in a block of memory.

Interrupt 6 — Invalid Opcode

This fault occurs when an invalid opcode is detected by the execution unit. (The exception is not detected until an attempt is made to execute the invalid opcode; i.e., prefetching an invalid opcode does not cause this exception.) No error code is pushed on the stack. The exception can be handled within the same task.

This exception also occurs when the type of operand is invalid for the given opcode. Examples include an intersegment JMP referencing a register operand, or an LES instruction with a register source operand.

Interrupt 7 — Coprocessor Not Available

This exception occurs in either of two conditions:

- The processor encounters an ESC (escape) instruction, and the EM (emulate) bit ofCR0 (control register zero) is set.
- The processor encounters either the WAIT instruction or an ESC instruction, and both the MP (monitor coprocessor) and TS (task switched) bits of CR0 are set.

Refer to Chapter 11 for information about the coprocessor interface.

Interrupt 8 — Double Fault

Normally, when the processor detects an exception while trying to invoke the handler for a prior exception, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception instead. To determine when two faults are to be signalled as a double fault, the 80386 divides the exceptions into three classes: benign exceptions, contributory exceptions, and page faults. Table 9-3 shows this classification.

Table 9-4 shows which combinations of exceptions cause a double fault and which do not.

The processor always pushes an error code onto the stack of the double-fault handler; however, the error code is always zero. The faulting instruction may not be restarted. If any other exception occurs while attempting to invoke the double-fault handler, the processor shuts down.

Note: For additional exception conditions, refer to the Intel 80386 Programmer's Manual