UNIT VI

EXCEPTIONS ,

INTERRUPTS & INTRODUCTION

OF MICROCONTROLLER

Prof. B. F. More, Department of Computer Engg., MESCOE, Pune

Syllabus

Unit VI	Interrupts, Exceptions, and Introduction to Microcontrollers	(07 Hours)
Interrupts and Exceptions: Identifying Interrupts, Enabling and Disabling Interrupts, Priority among Simultaneous Interrupts and Exceptions, Interrupt Descriptor Table (IDT), IDT Descriptors, Interrupt Tasks and Interrupt Procedures, Error Code, and Exception Conditions. **Introduction to Microcontrollers:** Architecture of typical Microcontroller, Difference between Microprocessor and Microcontroller, Characteristics of microcontrollers, Application of Microcontrollers.		

- Exceptions and interrupts are forced transfers of execution to a task or a procedure.
- The task or procedure is called a *handler*.
- Interrupts occur at random times during the execution of a program, in response to signals from hardware.
- Exceptions occur when instructions are executed which provoke exceptions.
- Usually, the servicing of interrupts and exceptions is performed in a manner transparent to application programs.
- Interrupts are used to handle events external to the processor, such as requests to service peripheral devices.
- Exceptions handle conditions detected by the processor in the course of executing instructions, such as division by zero.

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

1. Interrupts

- Maskable interrupts, which are received on the CPU's INTR input pin. Maskable interrupts do not occur unless the interrupt-enable flag (IF) is set.
- Nonmaskable interrupts, which are received on the NMI (Non-Maskable Interrupt) input of the processor. The processor does not provide a mechanism to prevent nonmaskable interrupts.

2. Exceptions

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

EXCEPTION AND INTERRUPT VECTORS

- The processor associates an identifying number with each different type of interrupt or exception. This number is called a *vector*.
- The NMI interrupt and the exceptions are assigned vectors in the range 0 through 31. Not all of these vectors are currently used by the processor; unassigned vectors in this range are reserved for possible future uses. Do not use unassigned vectors.
- The vectors for maskable interrupts are determined by hardware.
- External interrupt controllers (such as Intel's 8259A Programmable Interrupt Controller) put the vector on the processor's bus during its interrupt-acknowledge cycle.
- Any vectors in the range 32 through 255 can be used. Table next shows the assignment of exception and interrupt vectors.

Table 14-1. Exception and Interrupt Vectors

Vector Number	Description
0	Divide Error
1	Debug Exception
2	NMI Interrupt
3	Breakpoint
4	INTO-detected Overflow
5	BOUND Range Exceeded
6	Invalid Opcode
7	Device Not Available
8	Double Fault
9	CoProcessor Segment Overrun (reserved)
10	Invalid Task State Segment
11	Segment Not Present
12	Stack Fault
13	General Protection
14	Page Fault
15	(Intel reserved. Do not use.)
16	Floating-Point Error
17	Alignment Check
18	Machine Check*
19-31	(Intel reserved. Do not use.)
32-255	Maskable Interrupts

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

1. Fault

- A fault is an exception which is reported at the instruction boundary prior to the instruction in which the exception was detected.
- The fault is reported with the machine restored to a state which permits the instruction to be restarted.
- The return address for the fault handler points to the instruction which generated the fault, rather than the instruction following the faulting instruction.

2. Traps

A trap is an exception which is reported at the instruction boundary immediately after the instruction in which the exception was detected.

3. Abort

- An abort is an exception which does not always report the location of the instruction causing the exception and does not allow restart of the program which caused the exception.
- Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

Table 14-2. Priority Among Simultaneous Exceptions and Interrupts

Priority	Class	Descriptions
Highest	Class 1	Traps on the Previous Instruction - Breakpoints - Debug Trap Exceptions (TF flag set, T bit in TSS set, or data/IO breakpoint)
	Class 2	External Interrupts - NMI Interrupts - Maskable Interrupts
	Class 3	Faults from fetching next instruction - Code Breakpoint Fault
	Class 4	Faults from fetching or decoding the next instruction - Code Segment Limit Violation - Page Fault on Prefetch - Illegal Opcode - Instruction length > 15 bytes - Coprocessor Not Available
Lowest	Class 5	Faults on Executing an Instruction - General Detection - FP error (from previous FP instruction) - Interrupt on Overflow - Bound - Invalid TSS - Segment Not Present - Stack Exception - General Protection - Data Page Fault - Alignment Check

- The IDT may reside anywhere in physical memory.
- Figure (Next) shows, the processor locates the IDT using the IDTR register.
- This register holds both a 32-bit base address and 16-bit limit for the IDT.
- The LIDT and SIDT instructions load and store the contents of the IDTR register.
- Both instructions have one operand, which is the address of six bytes in memory.

LIDT (Load IDT register) loads the IDTR register with the base address and limit held in the memory operand. This instruction can be executed only when the CPL is 0. It normally is used by the initialization code of an operating system when creating an IDT. An OS also may use it to change from one IDT to another.

- **SIDT (Store IDT register)** copies the base and limit value stored in IDTR to memory. This instruction can be executed at any privilege level.

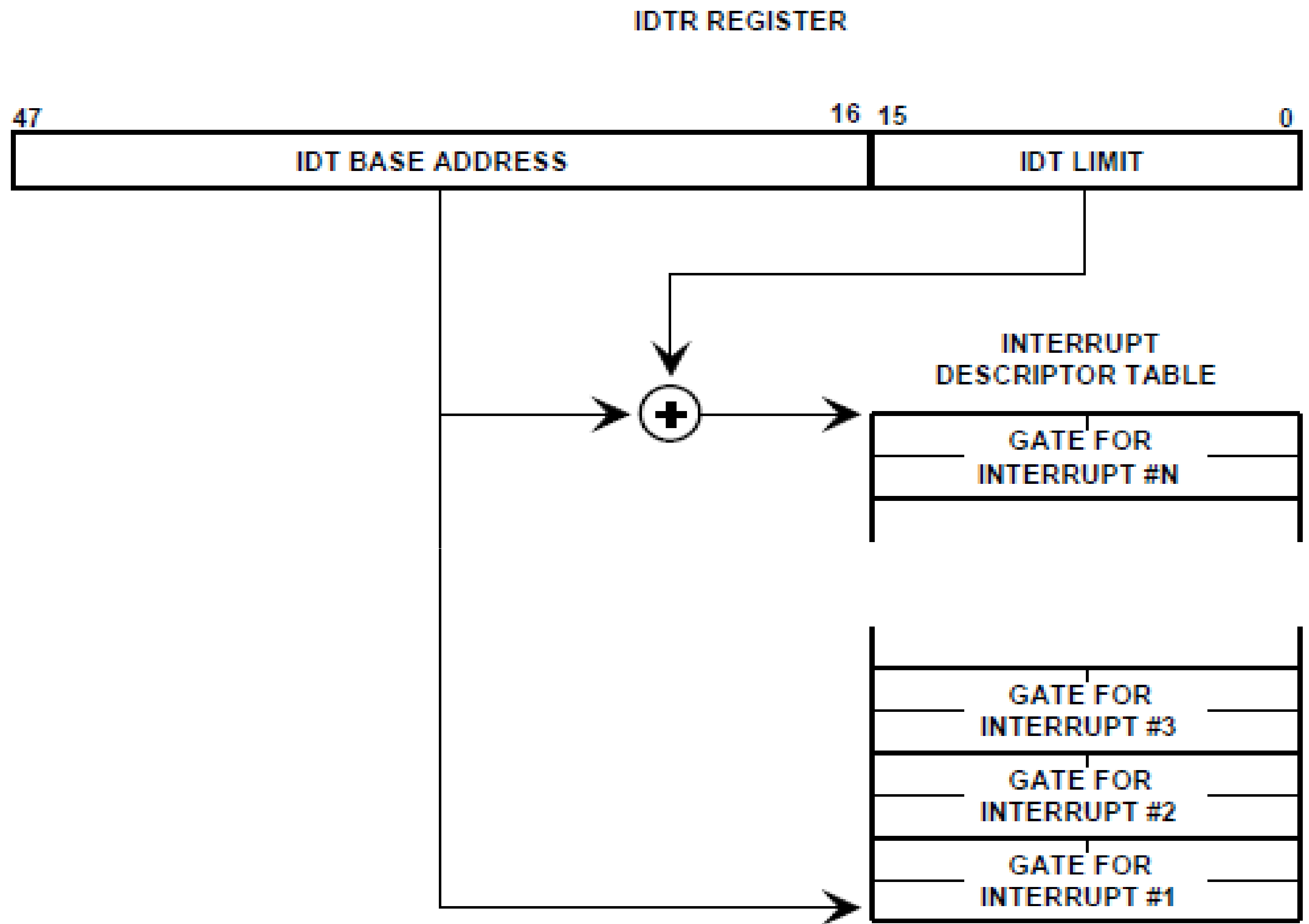
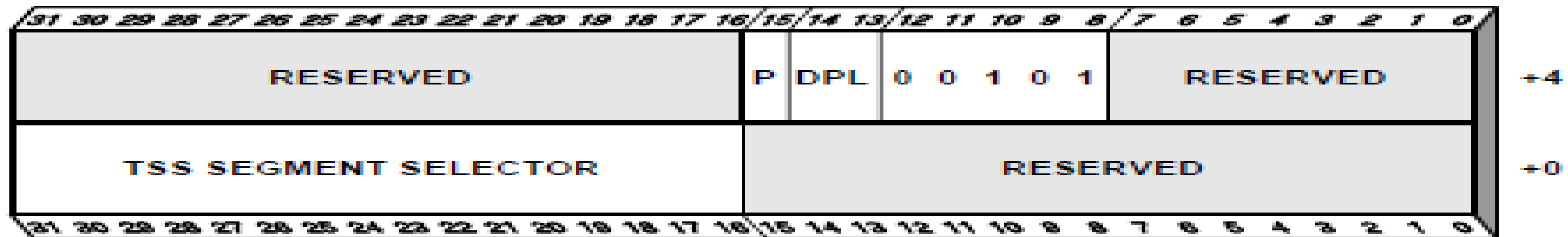


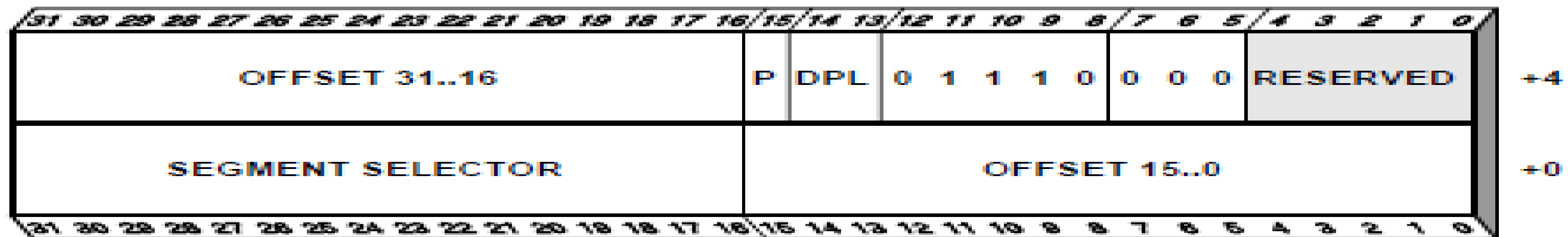
Figure 14-1. IDTR Locates IDT in Memory

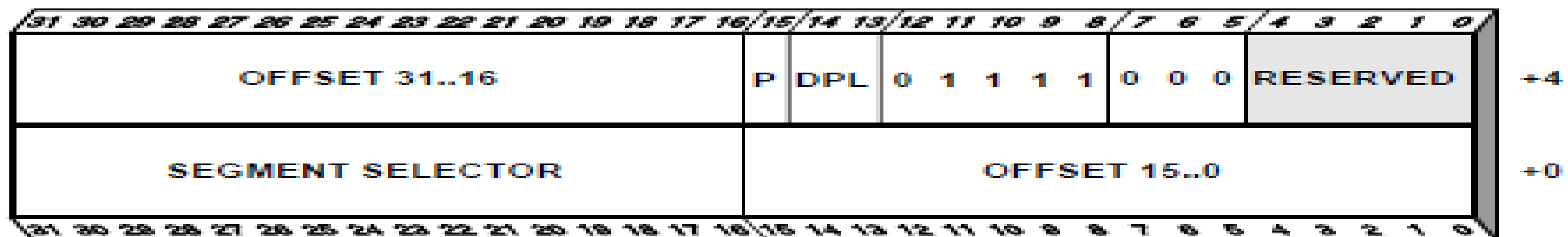
IDT DESCRIPTORS

- The IDT may contain any of three kinds of descriptors:

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







DPL	DESCRIPTOR PRIVILEGE LEVEL
OFFSET	OFFSET TO PROCEDURE ENTRY POINT
P	SEGMENT PRESENT BIT
RESERVED	DO NOT USE
SELECTOR	SEGMENT SELECTOR FOR DESTINATION CODE SEGMENT

APM122

Figure 14-2. IDT Gate Descriptors

Interrupt & Exception Descriptions

1. **Interrupt / Vector 0—Divide Error**

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

2.Interrupt 1—Debug Exceptions

- The processor generates a debug exception for a number of conditions; whether the exception is a fault or a trap depends on the condition

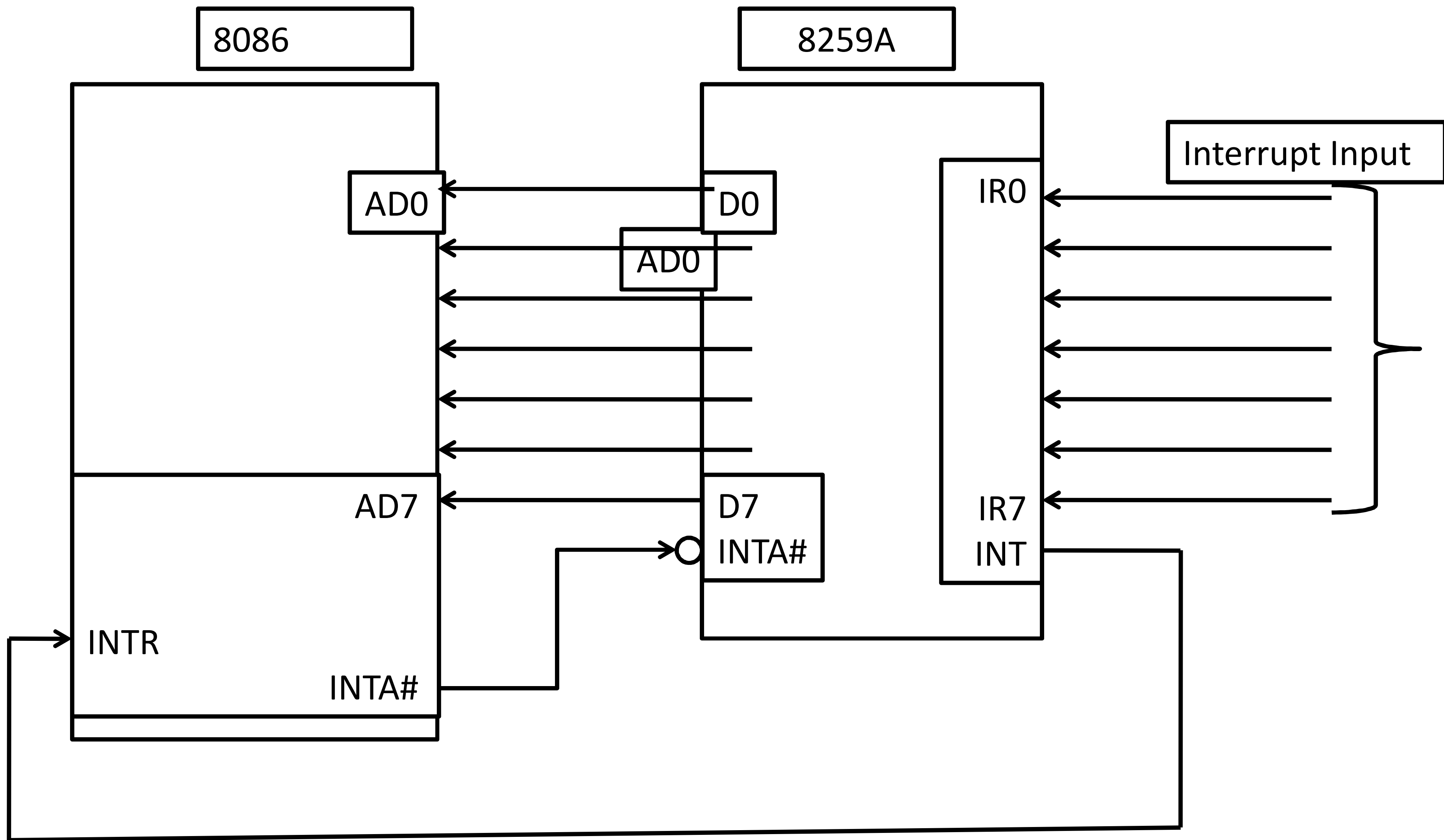
Instruction address breakpoint	fault
Data address breakpoint	trap
General detect	fault
Single-step	trap
Task-switch breakpoint	trap

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) 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. 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



8259A Connected To An 8086

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 signaled 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 and STI

- 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.

IF Flag

- The IF is also affected implicitly by the following operations:

1. The instruction PUSHF stores all flags, including IF, in the stack where they can be examined.
2. Task switches and the instructions POPF and IRET load the flags register; therefore, they can be used to modify IF.
3. Interrupts through interrupt gates automatically reset IF, disabling interrupts.

- **RF Masks Debug Faults**

1. The RF bit in EFLAGS controls the recognition of debug faults.
2. This permits debug faults to be raised for a given instruction at most once, no matter how many times the instruction is restarted.

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.

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 next.
- 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.

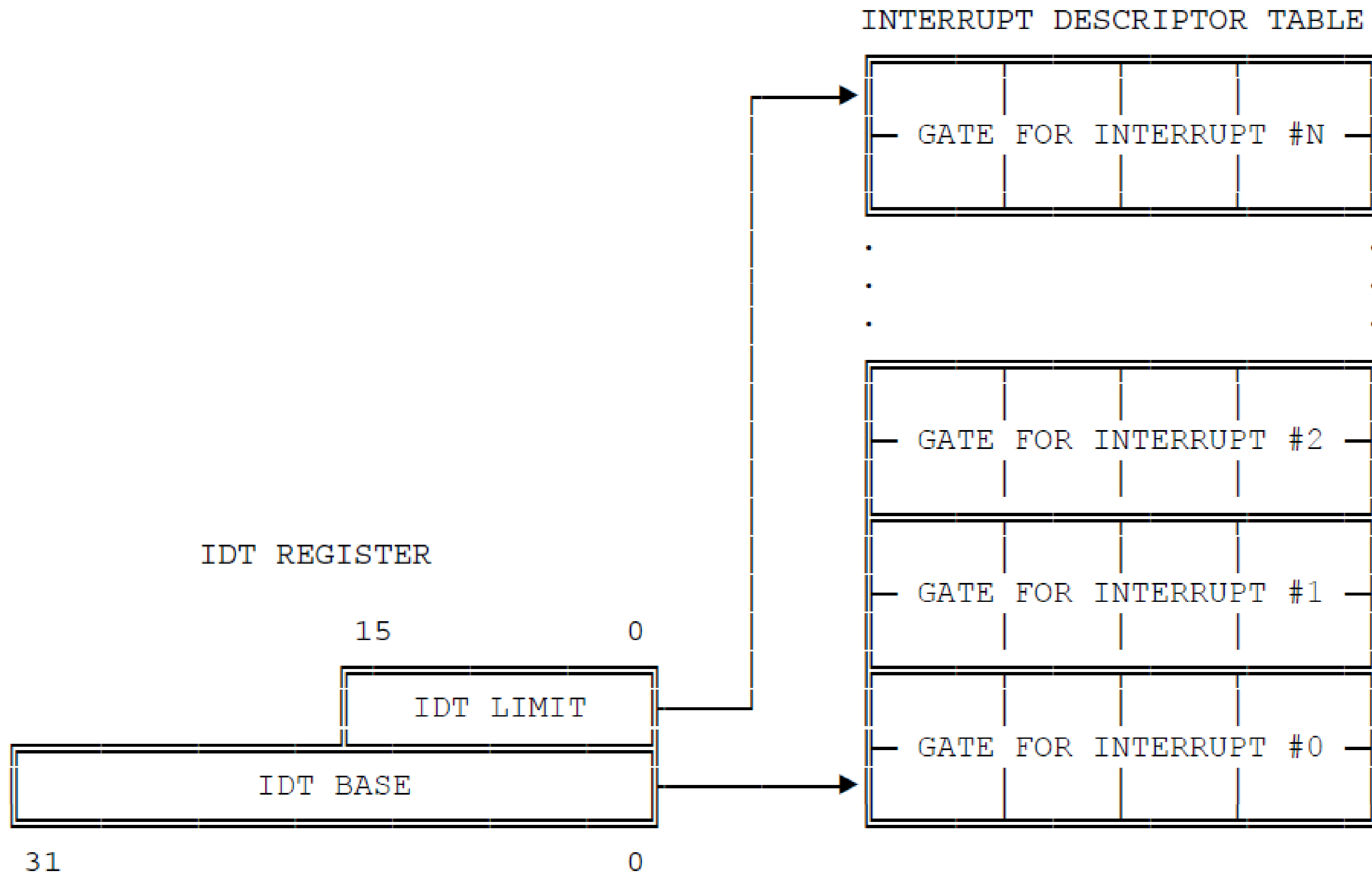
Table 9-2. Priority Among Simultaneous Interrupts and Exceptions

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 exception or interrupt vector with a descriptor for the procedure or task which services the associated event.
- Like the GDT and LDTs, the IDT is an array of 8-byte descriptors.
- Unlike the GDT, the first entry of the IDT may contain a descriptor.
- To form an index into the IDT, the processor scales the exception or interrupt vector by eight, the number of bytes in a descriptor.
- Because there are only 256 vectors, the IDT need not contain more than 256 descriptors.
- It can contain fewer than 256 descriptors; descriptors are required only for the interrupt vectors which may occur.

- The IDT may reside anywhere in physical memory.



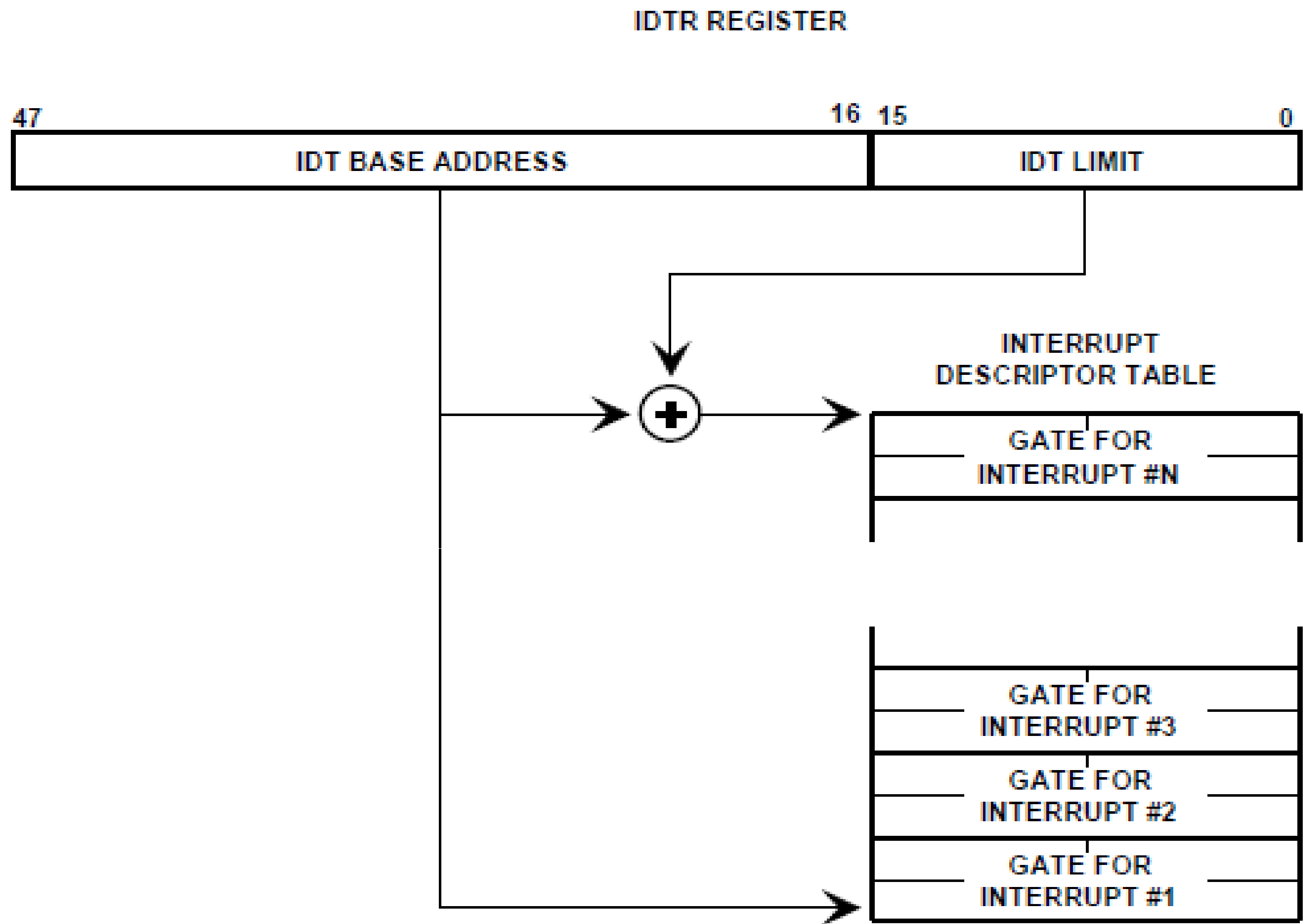
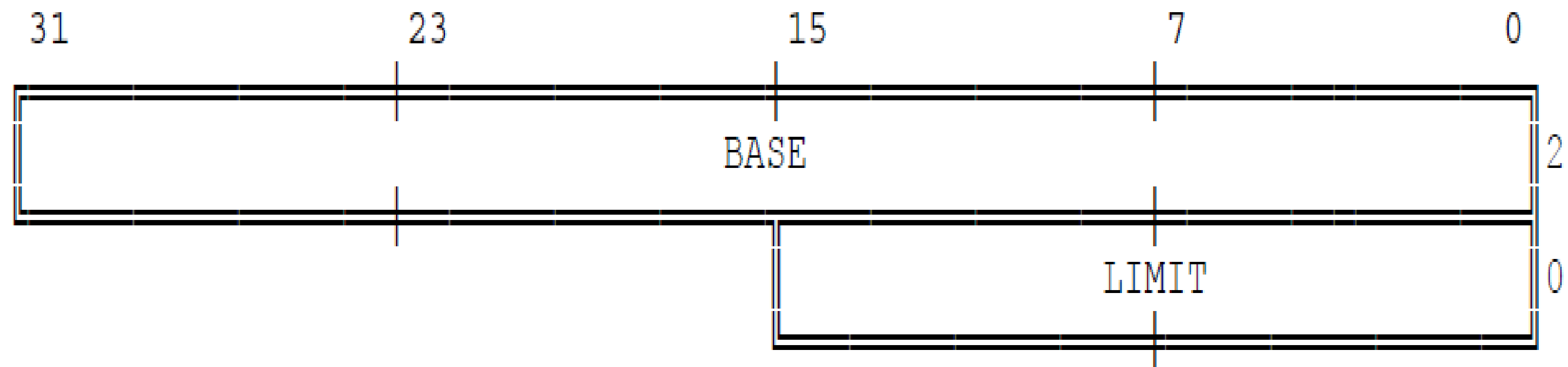


Figure 14-1. IDTR Locates IDT in Memory

Conti…

- 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. Pseudo-Descriptor Format for LIDT and SIDT



LIDT :Load IDT register

- loads the IDT register with the linear base address and limit values contained in the memory operand.
- 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.

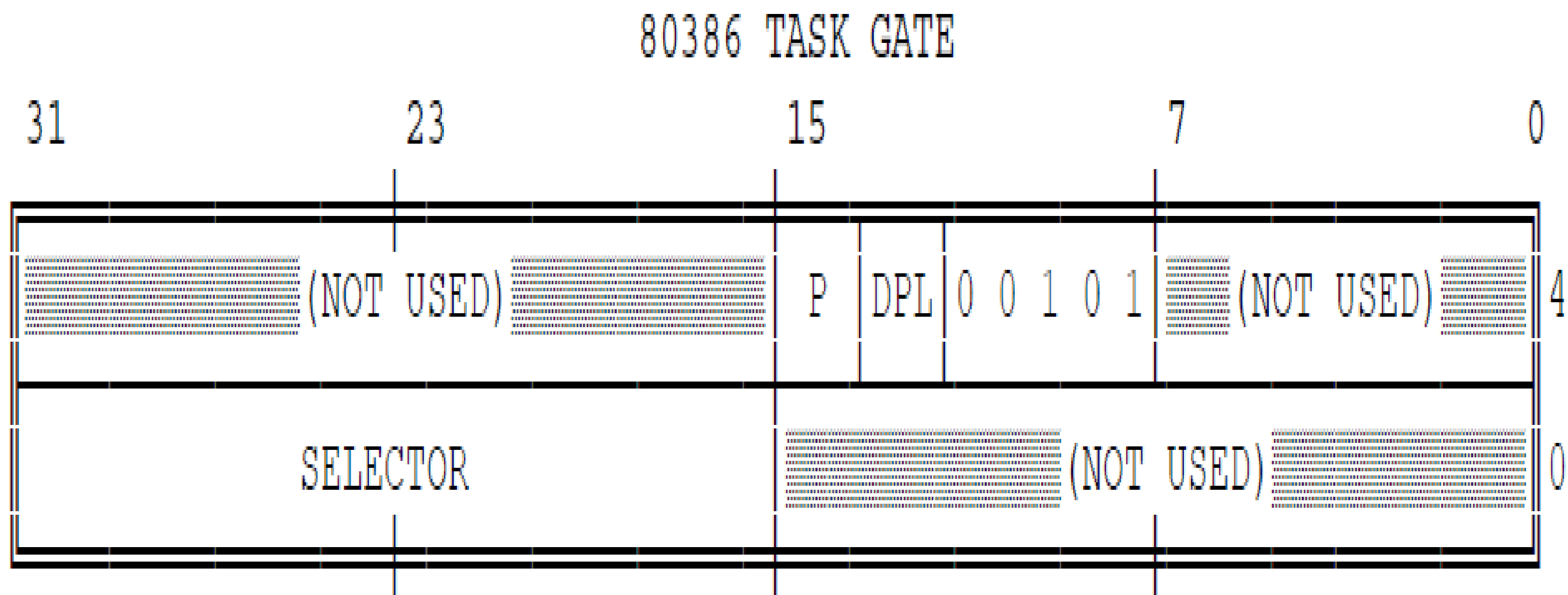
SIDT : Store IDT register

- copies the base and limit value stored in IDTR to a memory location.
- This instruction can be executed at any privilege level.

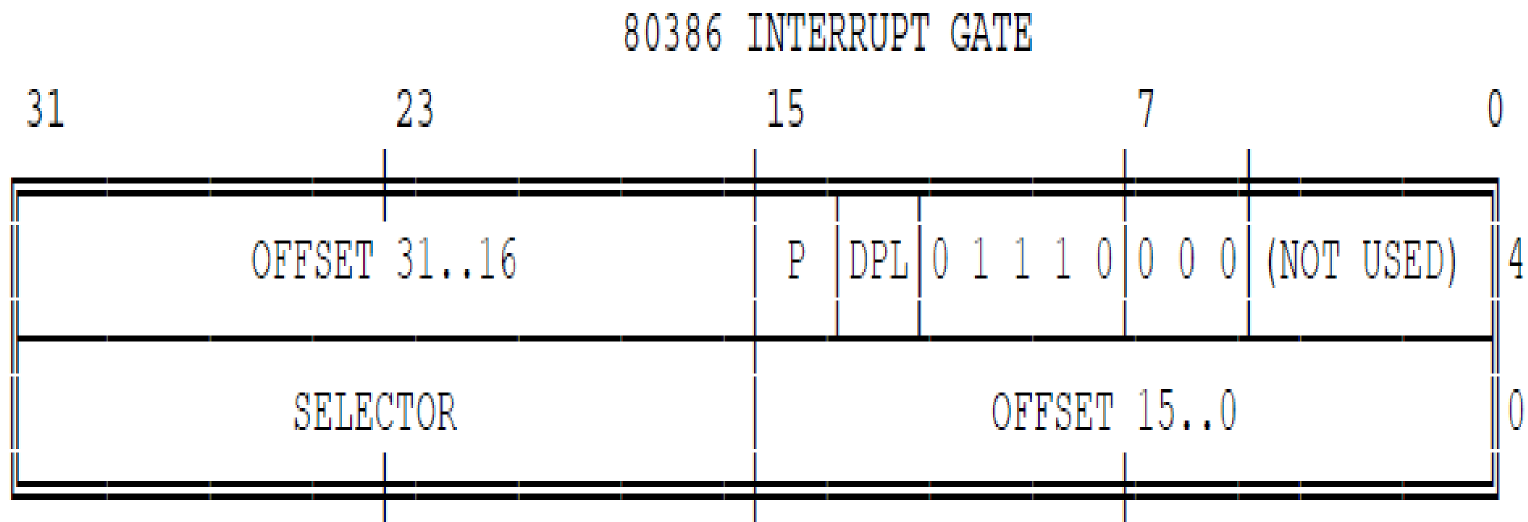
IDT Descriptors

- The IDT may contain any of three kinds of descriptor:

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

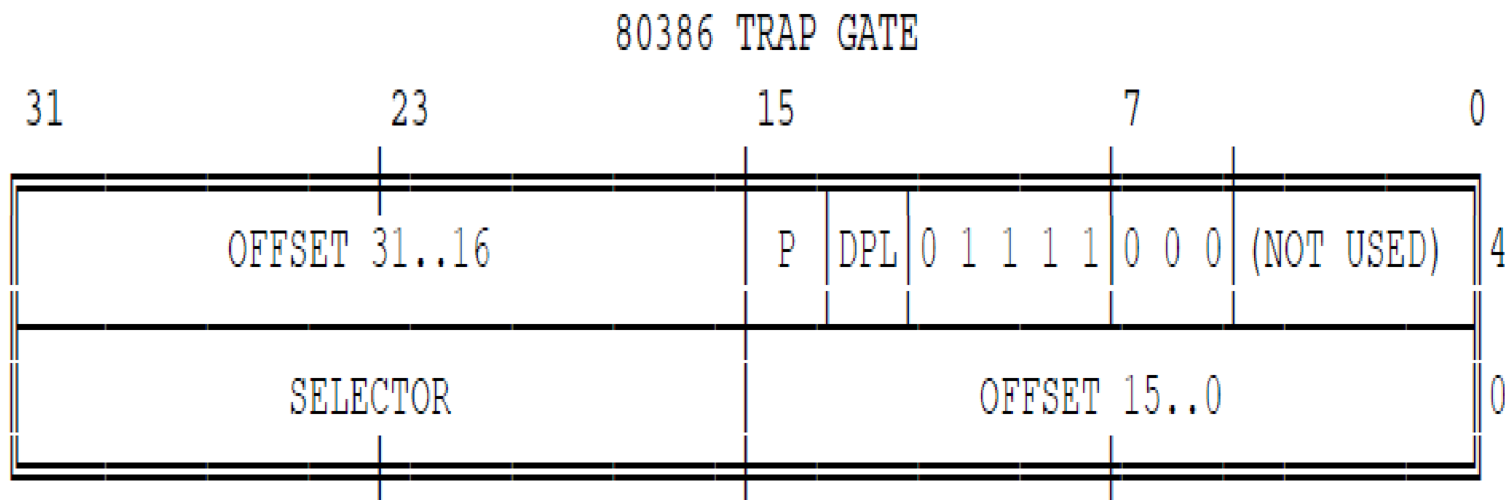


- The Interrupt Gate is used to specify a interrupt service routine
- When you do INT 50 in assembly, running in protected mode, the CPU looks up the 50th entry (located at 50 * 8) in the IDT. Then the Interrupt Gates selector and offset value is loaded.
- The selector and offset is used to call the interrupt service routine. When the IRET instruction is read, it returns.
- If running in 32 bit mode and the specified selector is a 16 bit selector, then the CPU will go in 16 bit protected mode after calling the interrupt service routine.



Conti…

- To return you need to do 32 IRET, else the CPU doesn't know that it should do a 32 bit return
- Here are some pre-cooked type_attr values people are likely to use (assuming DPL=0):
- 32-bit Interrupt gate: 0x8E (P=1, DPL=00b, S=0, type=1110b => type_attr=1000_1110b=0x8E)



- When an interrupt/exception occurs that corresponds to a Trap or Interrupt Gate, the CPU places the return info on the stack (EFLAGS, CS, EIP), so the interrupt handler can resume the interrupted code by IRET.
- Then, execution is transferred to the given selector:offset from the gate descriptor.
- For some exceptions, an error code is also pushed on the stack, which must be POPped before doing IRET.
- Trap and Interrupt gates are similar, and their descriptors are structurally the same, they differ only in the "type" field.
- The difference is that for interrupt gates, interrupts are automatically disabled upon entry and reenabled upon IRET which restores the saved EFLAGS.
- 32-bit Trap gate: 0x8F (P=1, DPL=00b, S=0, type=1111b => type_attr=1000_1111b=0x8F)

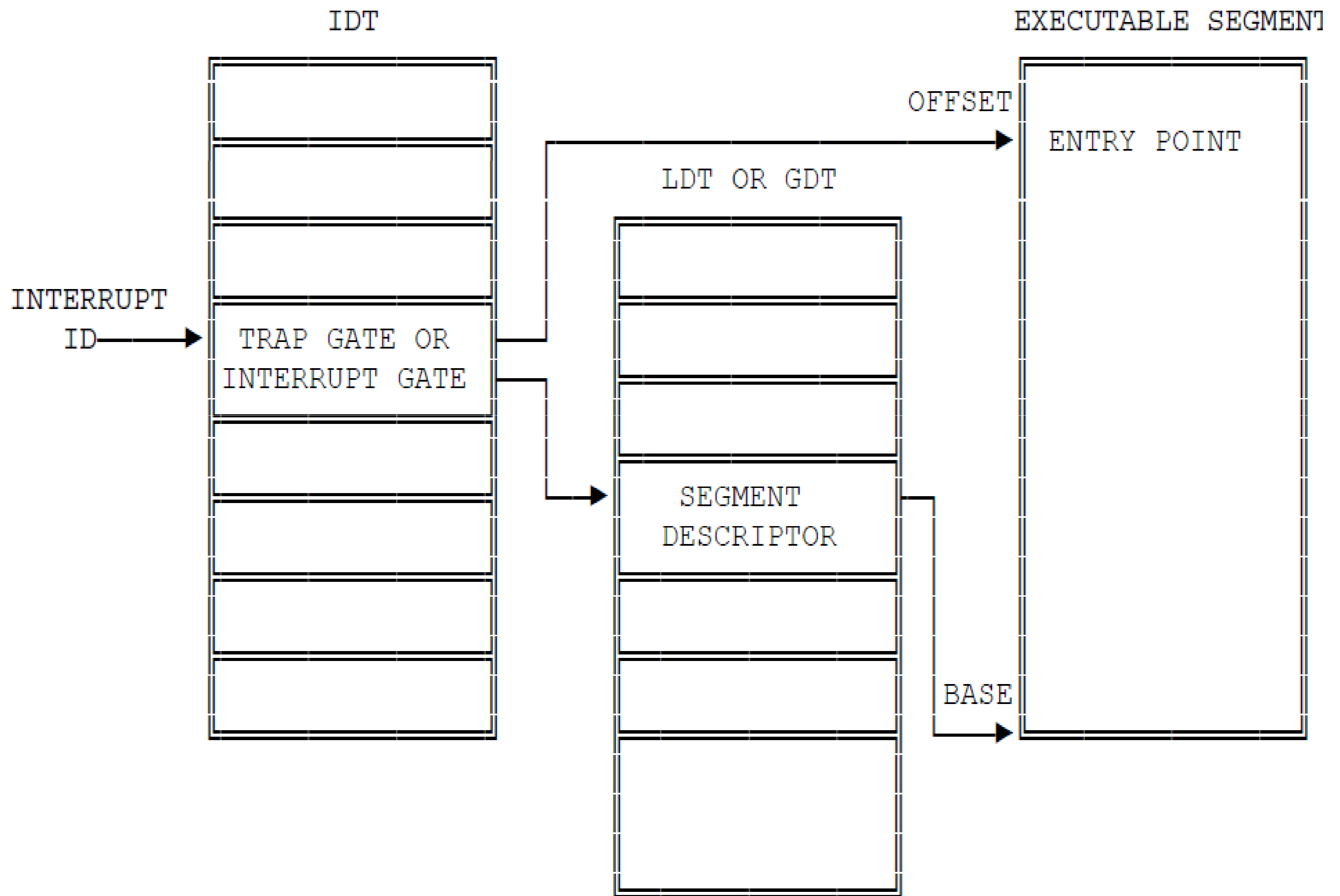
Interrupt Tasks and Interrupt Procedures

- 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

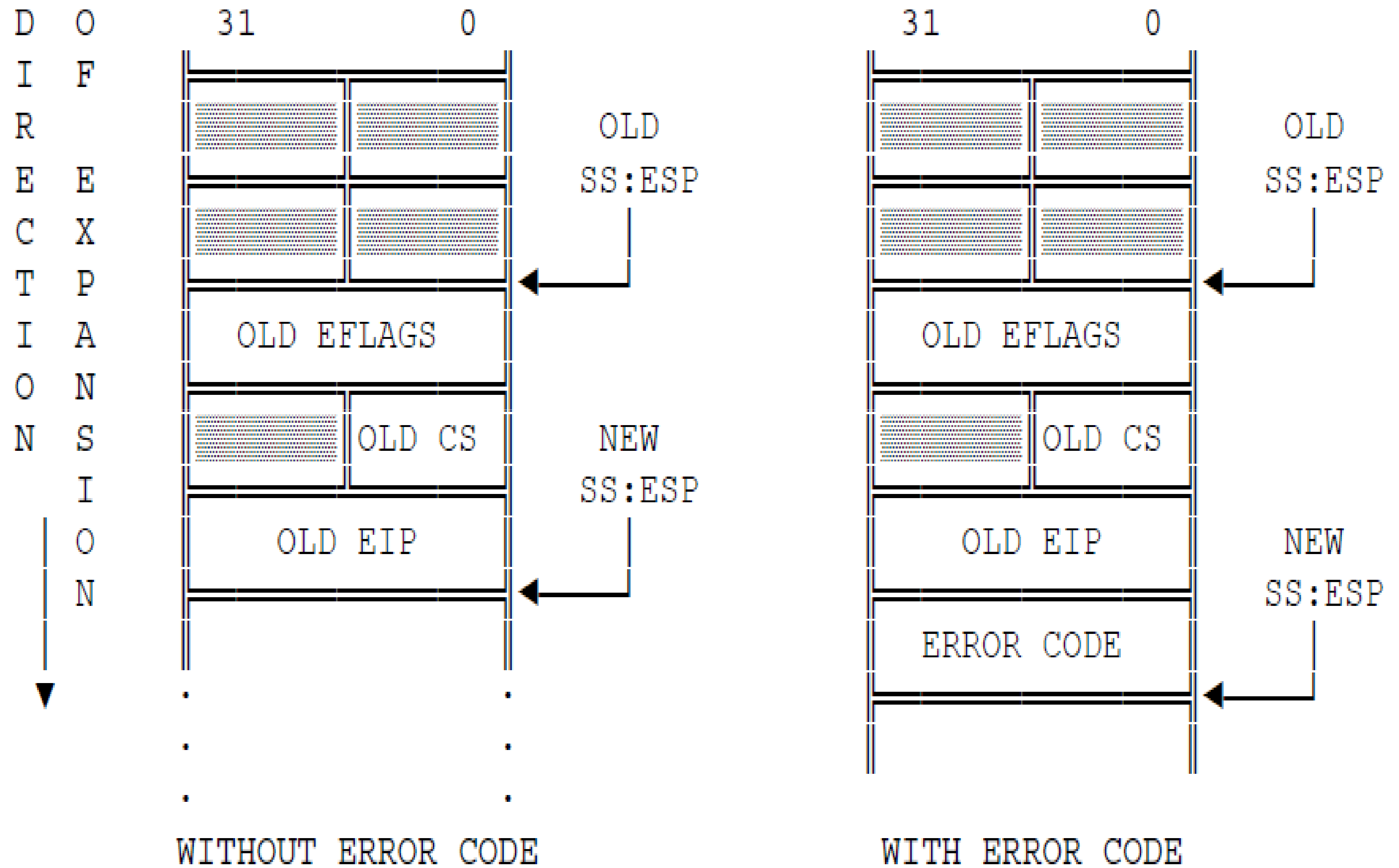
- An interrupt gate or trap gate points indirectly to a procedure which will execute in the context of the currently executing task
- Figure
- The selector of the gate points to an executable-segment descriptor in either the GDT or the current LDT.
- The offset field of the gate points to the beginning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

Figure 9-4. Interrupt Vectoring for Procedures



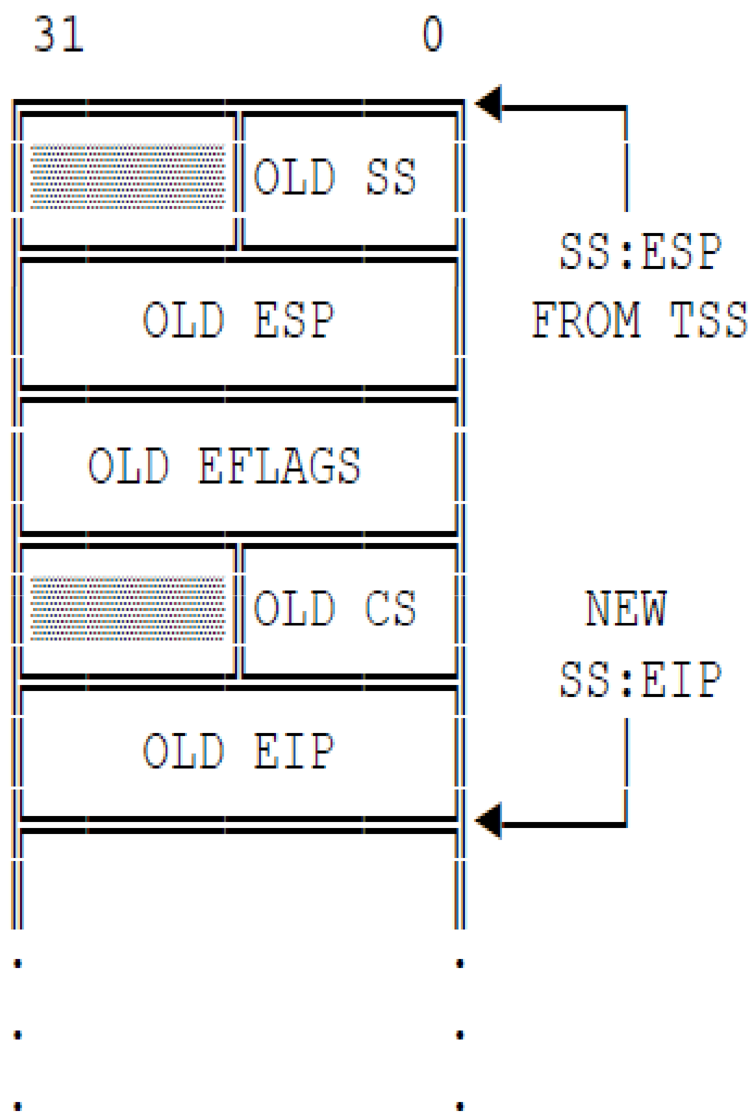
Stack of Interrupt Procedure

- 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.
- 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.

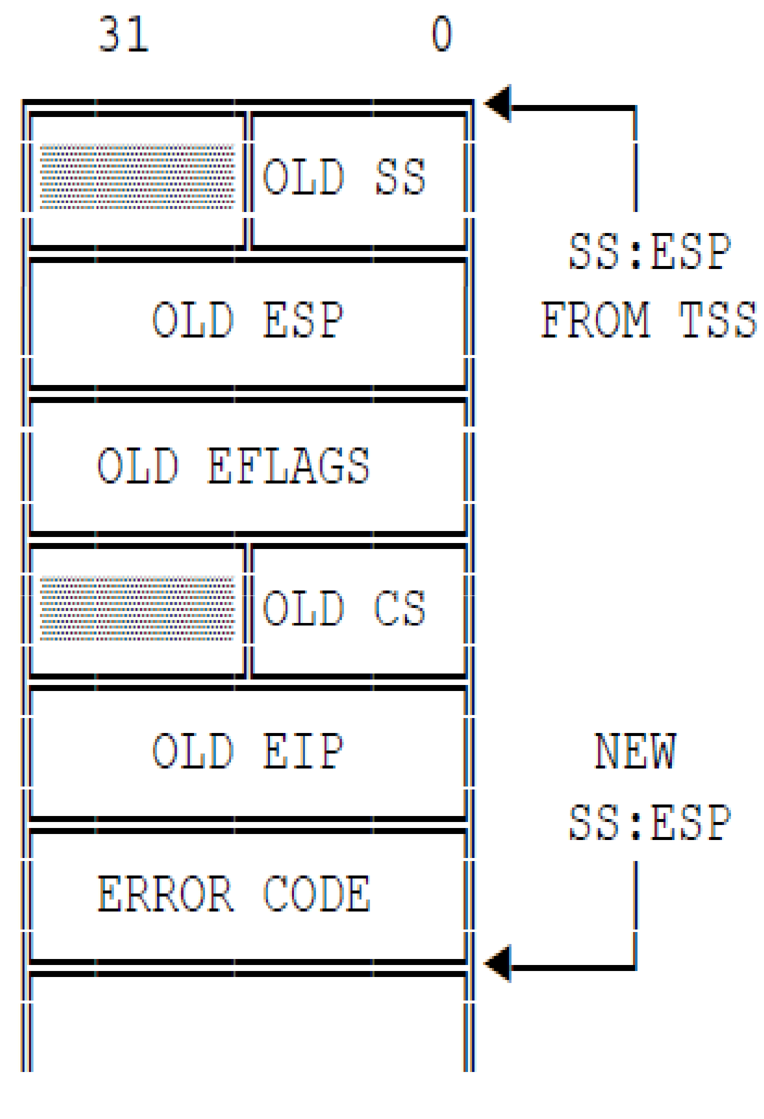

WITHOUT PRIVILEGE TRANSITION
DIRECTION OF EXPANSION
31
0
OLD EFLAGS
OLD CS
OLD EIP
OLD SS:ESP
NEW SS:ESP
WITHOUT ERROR CODE
31
0
OLD EFLAGS
OLD CS
OLD EIP
ERROR CODE
OLD SS:ESP
NEW SS:ESP
WITH ERROR CODE

WITH PRIVILEGE TRANSITION

DIRECTION OF EXPANSION



WITHOUT ERROR CODE



WITH ERROR CODE

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 IRET increments EIP by an extra four bytes (because of the flags on the stack) and moves the saved flags into the EFLAGS register.
- The IOPL field of EFLAGS is changed only if the CPL is zero.
- The IF flag is changed only if CPL $\leq$ 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
- 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.

Protection in Interrupt Procedures

- The privilege rule that governs interrupt procedures is similar to that for procedure calls:
  - the CPU does not permit an interrupt to transfer control to a procedure in a segment of lesser privilege (numerically greater privilege level) than the current privilege level.
- An attempt to violate this rule results in a general protection exception
- Because occurrence of interrupts is not generally predictable, this privilege rule effectively imposes restrictions on the privilege levels at which interrupt and exception handling procedures can execute.

Conti…

- Either of the following strategies can be employed to ensure that the privilege rule is never violated.

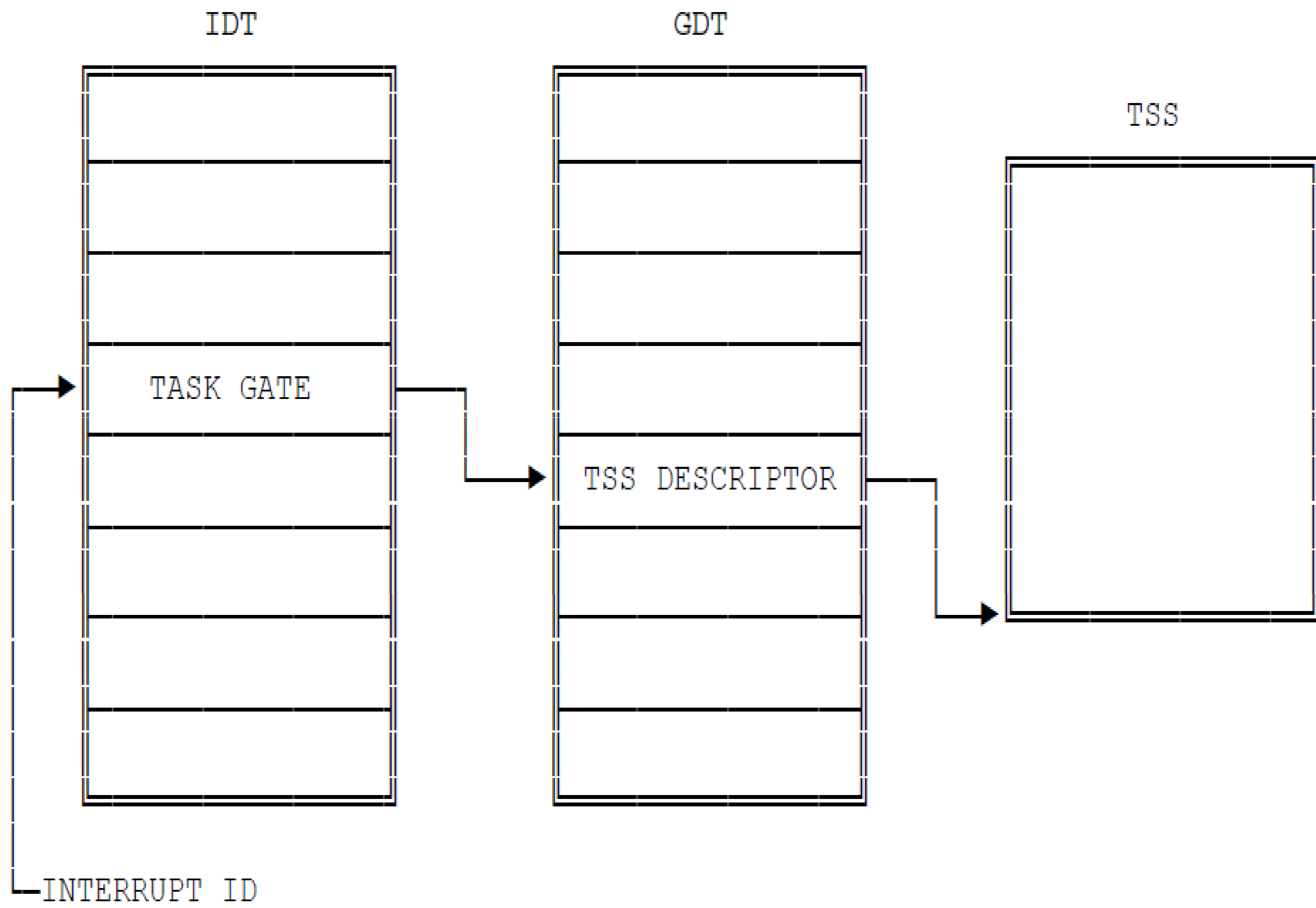
1. Place the handler in a conforming segment. This strategy suits the handlers for certain exceptions (divide error, for example). Such a handler must use only the data available to it from the stack. If it needed data from a data segment, the data segment would have to have privilege level three, thereby making it unprotected.
2. Place the handler procedure in a privilege level zero segment.

Interrupt Tasks

- A task gate in the IDT points indirectly to a task
- The selector of the gate points to a TSS descriptor in the GDT.
- When an interrupt or exception vectors to a task gate in the IDT, a task switch results.
- Handling an interrupt with a separate task offers two advantages:

1. The entire context is saved automatically.
2. The interrupt handler can be isolated from other tasks by giving it a separate address space, either via its LDT or via its page directory.

Figure 9-6. Interrupt Vectoring for Tasks



Conti…

- The interrupt task returns to the interrupted task by executing an IRET instruction.
- If the task switch is caused by an exception that has an error code, the processor automatically pushes the error code onto the stack that corresponds to the privilege level of the first instruction to be executed in the interrupt task
- When interrupt tasks are used in an operating system for the 80386, there are actually two schedulers:

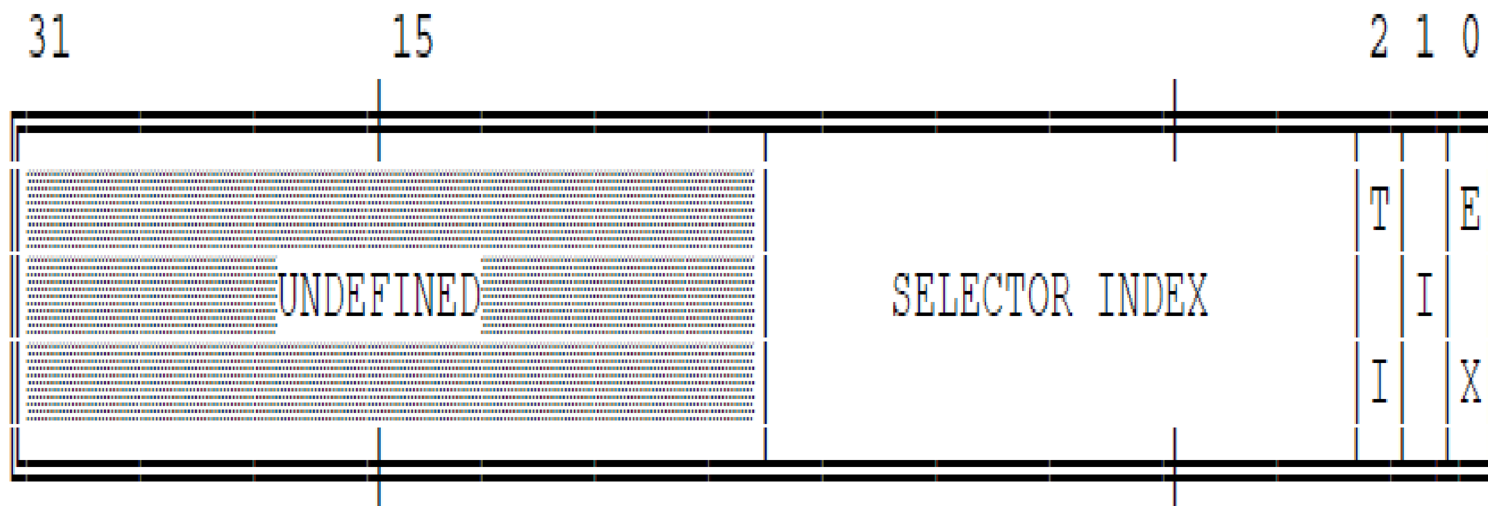
1. the software scheduler (part of the operating system) and
2. the hardware scheduler (part of the processor's interrupt mechanism).

- The design of the software scheduler should account for the fact that the hardware scheduler may dispatch an interrupt task whenever interrupts are enabled.

Error Code

- With exceptions that relate to a specific segment, the processor pushes an error code onto the stack of the exception handler (whether procedure or task).
- The format of the error code resembles that of a selector

Figure 9-7. Error Code Format



Conti…

- Instead of an RPL field, the error code contains two one-bit items:

1. The processor sets the EXT bit if an event external to the program caused the exception.
2. The processor sets the I-bit (IDT-bit) if the index portion of the error code refers to a gate descriptor in the IDT.

Change in format

- If the I-bit is not set, the TI bit indicates whether the error code refers to the GDT (value 0) or to the LDT (value 1).
- The remaining 14 bits are the upper 14 bits of the segment selector involved.
- In some cases the error code on the stack is null, i.e., all bits in the low-order word are zero.

Exception Conditions

- Each possible exception condition classifies the exception as :

1. Faults
2. Traps
3. Aborts

- This classification provides information needed by systems programmers for restarting the procedure in which the exception occurred

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:

1. Instruction address breakpoint fault.
2. Data address breakpoint trap.
3. General detect fault.
4. Single-step trap.
5. 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.

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 OS 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.

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:

1. The processor encounters an ESC (escape) instruction, and the EM (emulate) bit of CR0 (control register zero) is set.
2. 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.

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 signaled as a double fault, the 80386 divides the exceptions into three classes:
- Benign Exceptions,
- Contributory Exceptions, and
- Page Faults.

Table 9-3. Double-Fault Detection Classes

Class	ID	Description
Benign Exceptions	1	Debug exceptions
	2	NMI
	3	Breakpoint
	4	Overflow
	5	Bounds check
	6	Invalid opcode
	7	Coprocessor not available
	16	Coprocessor error
Contributory Exceptions	0	Divide error
	9	Coprocessor Segment Overrun
	10	Invalid TSS
	11	Segment not present
	12	Stack exception
	13	General protection
Page Faults	14	Page fault

Conti…

- 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
- 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.

Table 9-4. Double-Fault Definition

		SECOND EXCEPTION		
		Benign Exception	Contributory Exception	Page Fault
	Benign Exception	OK	OK	OK
FIRST EXCEPTION	Contributory Exception	OK	DOUBLE	OK
	Page Fault	OK	DOUBLE	DOUBLE

Interrupt 9 : Coprocessor Segment Overrun

- This exception is raised in protected mode if the 80386 detects a page or segment violation while transferring the middle portion of a coprocessor operand to the NPX.
- This exception is avoidable

Interrupt 10 — Invalid TSS

- Interrupt 10 occurs if during a task switch the new TSS is invalid.
- Table
- An error code is pushed onto the stack to help identify the cause of the fault.
- The EXT bit indicates whether the exception was caused by a condition outside the control of the program
- Example: an external interrupt via a task gate triggered a switch to an invalid TSS.

Table 9-5. Conditions That Invalidate the TSS

Error Code	Condition
TSS id + EXT	The limit in the TSS descriptor is less than 103
LTD id + EXT	Invalid LDT selector or LDT not present
SS id + EXT	Stack segment selector is outside table limit
SS id + EXT	Stack segment is not a writable segment
SS id + EXT	Stack segment DPL does not match new CPL
SS id + EXT	Stack segment selector RPL < > CPL
CS id + EXT	Code segment selector is outside table limit
CS id + EXT	Code segment selector does not refer to code segment
CS id + EXT	DPL of non-conforming code segment < > new CPL
CS id + EXT	DPL of conforming code segment > new CPL
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS segment selector is outside table limits
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS is not readable segment

Interrupt 11 — Segment Not Present

- Exception 11 occurs when the processor detects that the present bit of a descriptor is zero.
- The processor can trigger this fault in any of these cases:

1. While attempting to load the CS, DS, ES, FS, or GS registers; loading the SS register, however, causes a stack fault.
2. While attempting loading the LDT register with an LLDT instruction; loading the LDT register during a task switch operation, however, causes the "invalid TSS" exception.
3. While attempting to use a gate descriptor that is marked not-present.

Conti…

- If a not-present exception occurs during a task switch, not all the steps of the task switch are complete.
- During a task switch, the processor first loads all the segment registers, then checks their contents for validity.
- If a not-present exception is discovered, the remaining segment registers have not been checked and therefore may not be usable for referencing memory.
- The not-present handler should not rely on being able to use the values found in CS, SS, DS, ES, FS, and GS without causing another exception.
- The exception handler should check all segment registers before trying to resume the new task;

Conti…

- otherwise, general protection faults may result later under conditions that make diagnosis more difficult. There are three ways to handle this case:

1. Handle the not-present fault with a task. The task switch back to the interrupted task will cause the processor to check the registers as it loads them from the TSS.
2. PUSH and POP all segment registers. The task switch back to the interrupted task will cause the processor to check the registers as it loads them from the TSS.
3. Scrutinize the contents of each segment-register image in the TSS, simulating the test that the processor makes when it loads a segment register.

Conti…

- This exception pushes an error code onto the stack. The EXT bit of the error code is set if an event external to the program caused an interrupt that subsequently referenced a not-present segment.
- The I-bit is set if the error code refers to an IDT entry, e.g., an INT instruction referencing a not-present gate.
- An operating system typically uses the "segment not present" exception to implement virtual memory at the segment level.
- A not-present indication in a gate descriptor, however, usually does not indicate that a segment is not present (because gates do not necessarily correspond to segments).
- Not-present gates may be used by an operating system to trigger exceptions of special significance to the operating system.

Interrupt 12 — Stack Exception

- A stack fault occurs in either of two general conditions:
- As a result of a limit violation in any operation that refers to the SS register. This includes stack-oriented instructions such as POP,PUSH, ENTER, and LEAVE, as well as other memory references that implicitly use SS (for example, MOV AX, [BP+6]). ENTER causes this exception when the stack is too small for the indicated local-variable space.
- When attempting to load the SS register with a descriptor that is marked not-present but is otherwise valid. This can occur in a task switch, an interlevel CALL, an interlevel return, an LSS instruction, or a MOV or POP instruction to SS.

Conti…

- When the processor detects a stack exception, it pushes an error code onto the stack of the exception handler.
- If the exception is due to a not-present stack segment or to overflow of the new stack during an interlevel CALL, the error code contains a selector to the segment in question (the exception handler can test the present bit in the descriptor to determine which exception occurred); otherwise the error code is zero.
- An instruction that causes this fault is restartable in all cases.
- The return pointer pushed onto the exception handler's stack points to the instruction that needs to be restarted.
- This instruction is usually the one that caused the exception; however, in the case of a stack exception due to loading of a not-present stack-segment descriptor during a task switch, the indicated instruction is the first instruction of the new task.

11.13 Interrupt 13 — General Protection Exception

Interrupt 13 — General Protection Exception

- All protection violations that do not cause another exception cause a general protection exception.
- This includes (but is not limited to):

1. Exceeding segment limit when using CS, DS, ES, FS, or GS
2. Exceeding segment limit when referencing a descriptor table
3. Transferring control to a segment that is not executable
4. Writing into a read-only data segment or into a code segment
5. Reading from an execute-only segment
6. Loading the SS register with a read-only descriptor (unless the selector comes from the TSS during a task switch, in which case a TSS exception occurs
7. Loading SS, DS, ES, FS, or GS with the descriptor of a system segment

Conti…

8. Loading DS, ES, FS, or GS with the descriptor of an executable segment that is not also readable
9. Loading SS with the descriptor of an executable segment
10. Accessing memory via DS, ES, FS, or GS when the segment register contains a null selector
11. Switching to a busy task
12. Violating privilege rules
13. Loading CR0 with PG=1 and PE=0.
14. Interrupt or exception via trap or interrupt gate from V86 mode to privilege level other than zero.
15. Exceeding the instruction length limit of 15 bytes (this can occur only if redundant prefixes are placed before an instruction)

Conti…

- The general protection exception is a fault.
- In response to a general protection exception, the processor pushes an error code onto the exception handler's stack.
- If loading a descriptor causes the exception, the error code contains a selector to the descriptor; otherwise, the error code is null.
- The source of the selector in an error code may be any of the following:

1. An operand of the instruction.
2. A selector from a gate that is the operand of the instruction.
3. A selector from a TSS involved in a task switch.

Interrupt 14 — Page Fault

- This exception occurs when paging is enabled (PG=1) and the processor detects one of the following conditions while translating a linear address to a physical address:

1. The page-directory or page-table entry needed for the address translation has zero in its present bit.
2. The current procedure does not have sufficient privilege to access the indicated page.

- The processor makes available to the page fault handler two items of information that aid in diagnosing the exception and recovering from it:

1. An error code on the stack.
2. CR2 (control register two).

An error code on the stack

- The error code for a page fault has a format different from that for other exceptions
- Figure
- The error code tells the exception handler three things:

1. Whether the exception was due to a not present page or to an access rights violation.
2. Whether the processor was executing at user or supervisor level at the time of the exception.
3. Whether the memory access that caused the exception was a read or write.

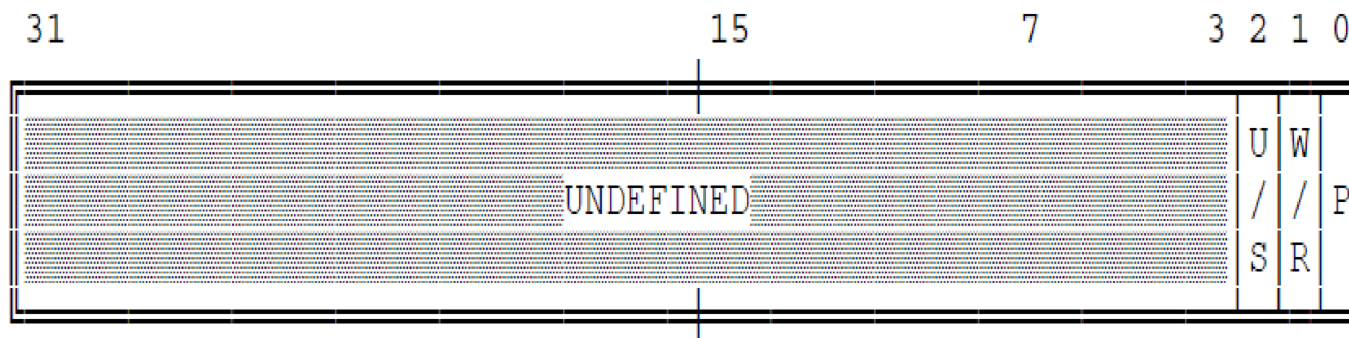


Figure 9-8. Page-Fault Error Code Format

Field	Value	Description
U/S	0	The access causing the fault originated when the processor was executing in supervisor mode.
	1	The access causing the fault originated when the processor was executing in user mode.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
P	0	The fault was caused by a not-present page.
	1	The fault was caused by a page-level protection violation.

CR2 (control register two)

- The processor stores in CR2 the linear address used in the access that caused the exception
- Figure
- The exception handler can use this address to locate the corresponding page directory and page table entries.
- If another page fault can occur during execution of the page fault handler, the handler should push CR2 onto the stack.

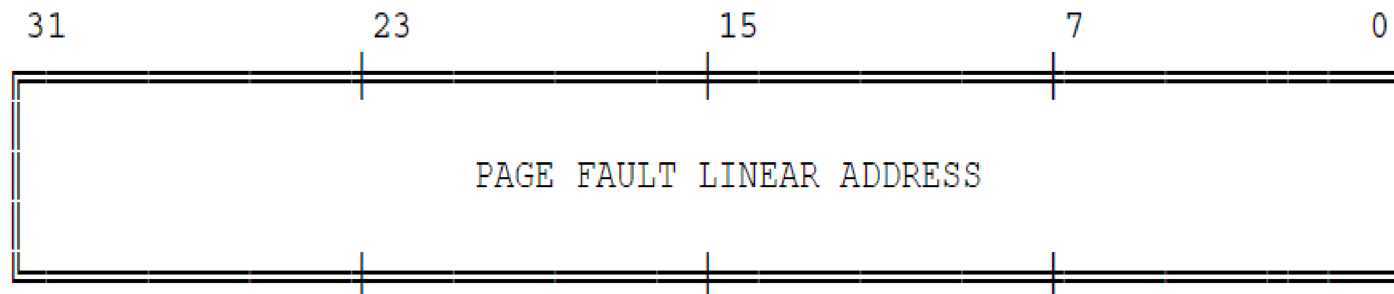


Figure 9-9. CR2 Format

Page Fault During Task Switch

- The processor may access any of four segments during a task switch:

1. Writes the state of the original task in the TSS of that task.
2. Reads the GDT to locate the TSS descriptor of the new task.
3. Reads the TSS of the new task to check the types of segment descriptors from the TSS.
4. May read the LDT of the new task in order to verify the segment registers stored in the new TSS.

- A page fault can result from accessing any of these segments.
- In the latter two cases the exception occurs in the context of the new task.
- The instruction pointer refers to the next instruction of the new task, not to the instruction that caused the task switch.
- If the design of the operating system permits page faults to occur during task-switches, the page-fault handler should be invoked via a task gate.

Page Fault with Inconsistent Stack Pointer

- Special care should be taken to ensure that a page fault does not cause the processor to use an invalid stack pointer (SS:ESP).
- Software written for earlier processors in the 8086 family often uses a pair of instructions to change to a new stack:

MOV SS, AX

MOV SP, Stack Top

- With the 80386, because the second instruction accesses memory, it is possible to get a page fault after SS has been changed but before SP has received the corresponding change.
- At this point, the two parts of the stack pointer SS:SP (or, for 32-bit programs, SS:ESP) are inconsistent.

Conti…

- The processor does not use the inconsistent stack pointer if the handling of the page fault causes a stack switch to a well defined stack
- The handler is a task or a more privileged procedure
- However, if the page fault handler is invoked by a trap or interrupt gate and the page fault occurs at the same privilege level as the page fault handler, the processor will attempt to use the stack indicated by the current (invalid) stack pointer.
- In systems that implement paging and that handle page faults within the faulting task (with trap or interrupt gates), software that executes at the same privilege level as the page fault handler should initialize a new stack by using the new LSS instruction rather than an instruction pair shown.
- When the page fault handler executes at privilege level zero (the normal case), the scope of the problem is limited to privilege-level zero code, typically the kernel of the operating system.

Interrupt 16 — Coprocessor Error

- The 80386 reports this exception when it detects a signal from the 80287 or 80387 on the 80386's ERROR# input pin.
- The 80386 tests this pin only at the beginning of certain ESC instructions and when it encounters a WAIT instruction while the EM bit of the MSW is zero (no emulation).

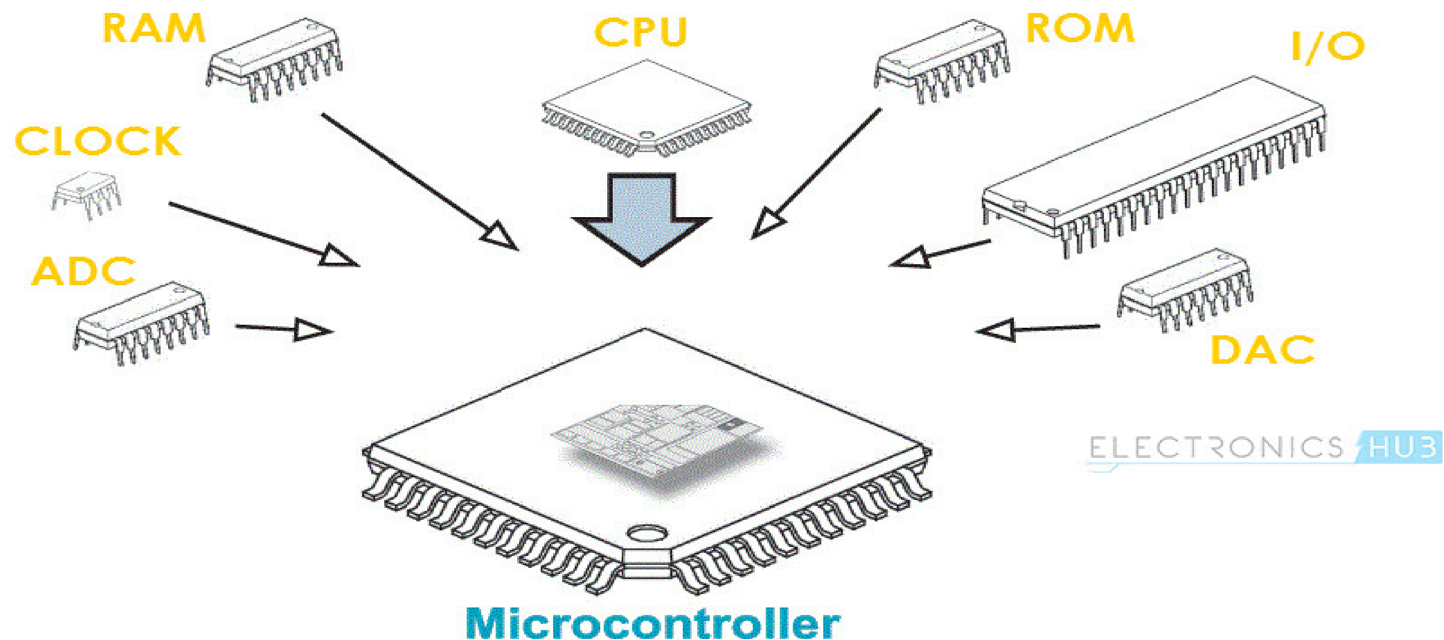
LGS/LSS/LDS/LES/LFS — Load Full Pointer

- These instructions read a full pointer from memory and store it in the selected segment register : register pair.
- The full pointer loads 16 bits into the segment register SS, DS, ES, FS, or GS.
- The other register loads 32 bits if the operand-size attribute is 32 bits, or loads 16 bits if the operand-size attribute is 16 bits.
- The other 16- or 32-bit register to be loaded is determined by the r16 or r32 register operand specified.
- When an assignment is made to one of the segment registers, the descriptor is also loaded into the segment register.
- The data for the register is obtained from the descriptor table entry for the selector given.
- A null selector (values 0000-0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception.

Microcontroller

Microcontroller

- A **microcontroller** is a small and low-cost microcomputer, which is designed to perform the specific tasks of embedded systems like displaying microwave's information, receiving remote signals, etc.
- The general microcontroller consists of the processor, the memory (RAM, ROM, EPROM), Serial ports, peripherals (timers, counters), etc.

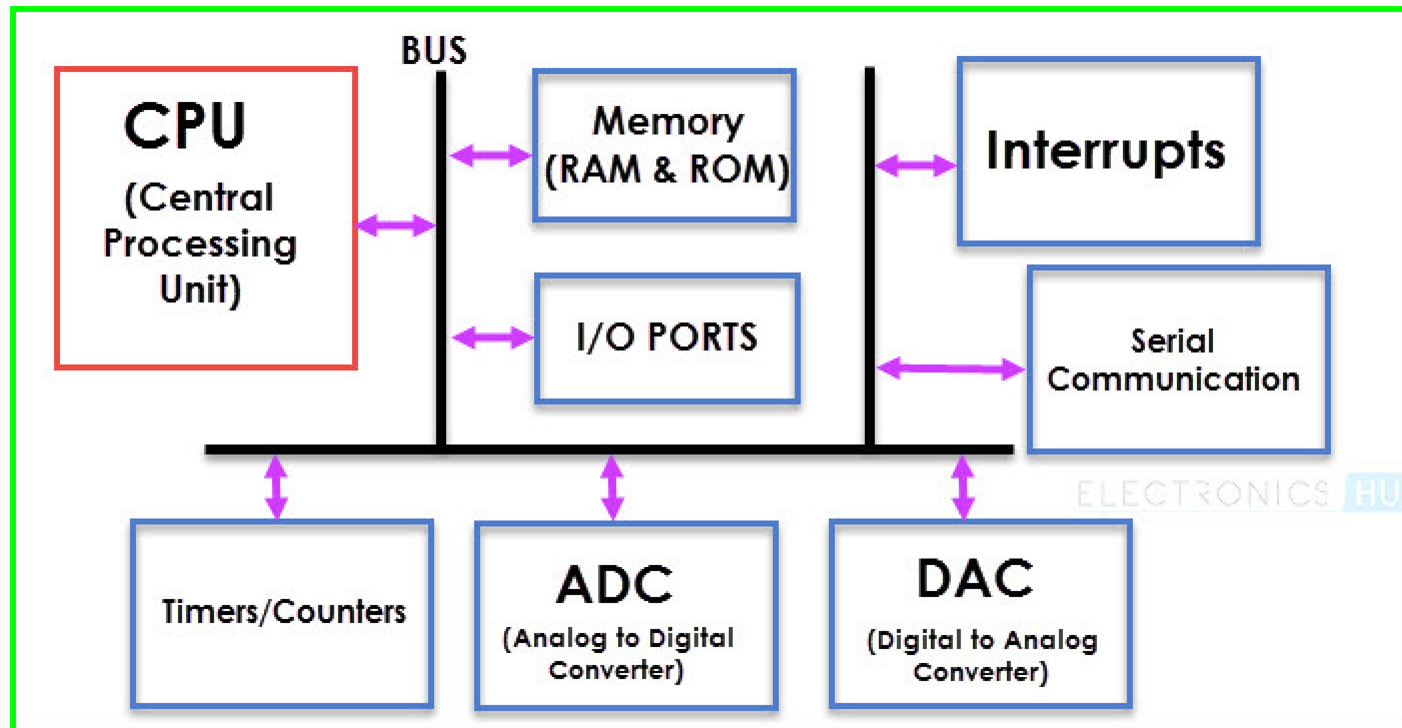


Microcontroller Components

- Basically, a Microcontroller consists of the following components.

1. Central Processing Unit (CPU)
2. Program Memory (ROM – Read Only Memory)
3. Data Memory (RAM – Random Access Memory)
4. Timers and Counters
5. I/O Ports (I/O – Input/Output)
6. Serial Communication Interface
7. Clock Circuit (Oscillator Circuit)
8. Interrupt Mechanism

Architecture of Microcontroller



Architecture of Microcontroller

- The Basic Components of a Microcontroller,

1. **CPU** Central Processing Unit or CPU is the brain of the Microcontroller. It consists of an Arithmetic Logic Unit (ALU) and a Control Unit (CU). A CPU reads, decodes and executes instructions to perform Arithmetic, Logic and Data Transfer operations.
2. **Memory** Any Computational System requires two types of Memory: Program Memory and Data Memory. Program Memory, as the name suggests, contains the program i.e. the instructions to be executed by the CPU. Data Memory on the other hand, is required to store temporary data while executing the instructions. Usually, Program Memory is a Read Only Memory or ROM and the Data Memory is a Random Access Memory or RAM. Data Memory is sometimes called as Read Write Memory (R/W M).
3. **I/O Ports** The interface for the Microcontroller to the external world is provided by the I/O Ports or Input/Output Ports. Inputs device like Switches, Keypads, etc. provide information from the user to the CPU in the form of Binary Data. The CPU, upon receiving the data from the input devices, executes appropriate instructions and gives response through Output Devices like LEDs, Displays, Printers, etc.

Architecture of Microcontroller

4. **Bus**

Another important component of a Microcontroller, but rarely discussed is the System Bus. A System bus is a group of connecting wire that connect the CPU with other peripherals like Memory, I/O Ports and other supporting components.

5. Timers/Counters

One of the important components of a Microcontroller are the Timers and Counters. They provide the operations of Time Delays and counting external events. Additionally, Timers and Counters can provide Function Generation, Pulse Width Modulation, Clock Control, etc.

6. Serial Port

One of the important requirement of a Microcontroller is to communicate with other device and peripherals (external). Serial Port proves such interface through serial communication. Most common serial communication implemented in Microcontrollers is UART.

7. Interrupts

A very important feature of a Microcontroller is Interrupts and its Interrupt Handling Mechanism. Interrupts can be external, internal, hardware related or software related.

Architecture of Microcontroller

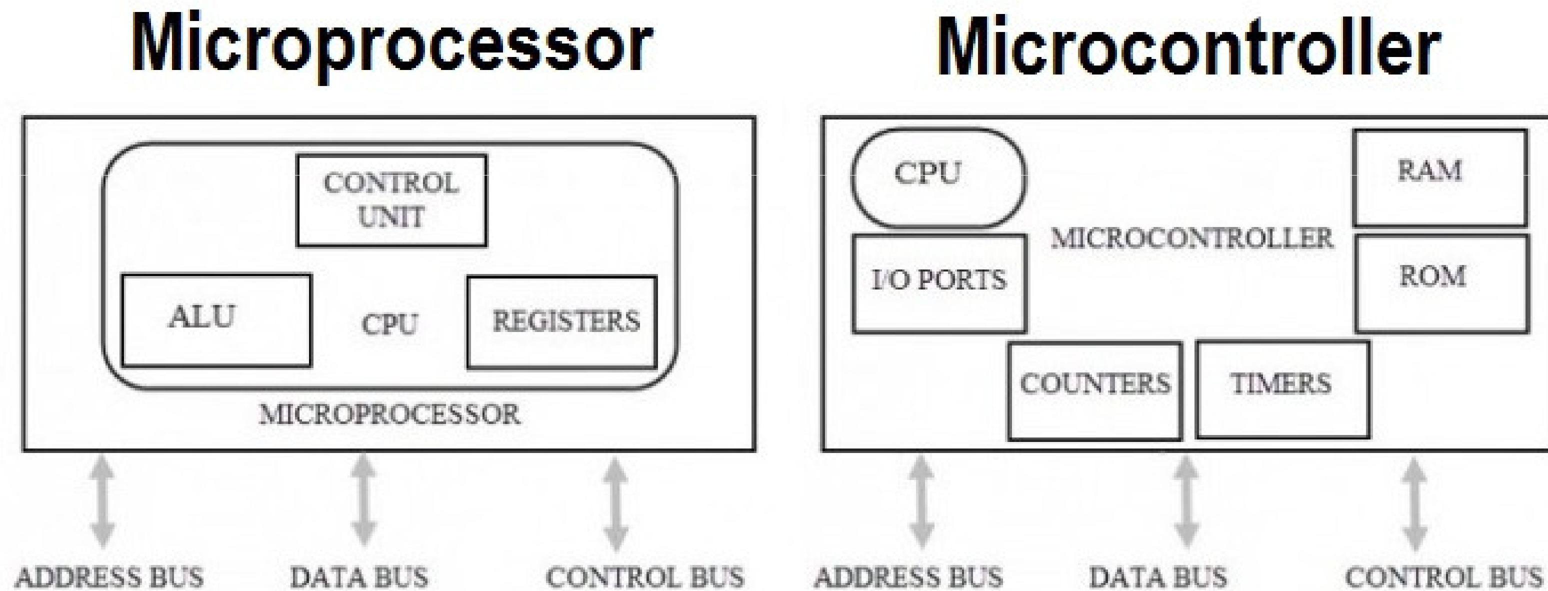
8. ADC (Analog to Digital Converter)

Analog to Digital Converter or ADC is a circuit that converts Analog signals to Digital Signals. The ADC Circuit forms the interface between the external Analog Input devices and the CPU of the Microcontroller. Almost all sensors are analog devices and the analog data from these sensors must be converted in to digital data for the CPU to understand.

9. DAC (Digital to Analog Converter)

DIgital to Analog Converter or DAC is a circuit, that works in contrast to an ADC i.e. it converts Digital Signals to Analog Signals. DAC forms the bridge between the CPU of the Microcontroller and the external analog devices.

Difference between Microprocessor and Microcontroller



Difference between Microprocessor and Microcontroller

Microprocessors		Microcontrollers
1	It is only a general purpose computer CPU	It is a micro computer itself
2	Memory, I/O ports, timers, interrupts are not available inside the chip	All are integrated inside the microcontroller chip
3	This must have many additional digital components to perform its operation	Can function as a micro computer without any additional components.
4	Systems become bulkier and expensive.	Make the system simple, economic and compact
5	Not capable for handling Boolean functions	Handling Boolean functions
6	Higher accessing time required	Low accessing time
7	Very few pins are programmable	Most of the pins are programmable
8	Very few number of bit handling instructions	Many bit handling instructions
9	Widely Used in modern PC and laptops	widely in small control systems
E.g.	INTEL 8086,INTEL Pentium series	INTEL8051,89960,PIC16F877

Characteristics of Microcontroller

- A micro-controller is a single integrated circuit, commonly with the following features:
- central processing unit – ranging from small and simple 4-bit processors to complex 32-bit or 64-bit processors
- volatile memory (RAM) for data storage
- ROM, EPROM, EEPROM or Flash memory for program and operating parameter storage
- discrete input and output bits, allowing control or detection of the logic state of an individual package pin
- serial input/output such as serial ports (UARTs)
- other serial communications interfaces like I^2C, Serial Peripheral Interface and Controller Area Network for system interconnect
- peripherals such as timers, event counters, PWM generators, and watchdog
- clock generator – often an oscillator for a quartz timing crystal, resonator or RC circuit
- many include analog-to-digital converters, some include digital-to-analog converters
- in-circuit programming and in-circuit debugging support

Microcontroller

- **Advantages of Microcontrollers**

1. A Microcontroller is a true device that fits the computer-on-a-chip idea.
2. No need for any external interfacing of basic components like Memory, I/O Ports, etc.
3. Microcontrollers doesn’t require complex operating systems as all the instructions must be written and stored in the memory. (RTOS is an exception).
4. All the Input/Output Ports are programmable.
5. Integration of all the essential components reduces the cost, design time and area of the product (or application).

- **Disadvantages of Microcontrollers**

1. Microcontrollers are not known for their computation power.
2. The amount of memory limits the instructions that a microcontroller can execute.
3. No Operating System and hence, all the instruction must be written.

Applications of Microcontrollers

- There are huge number of applications of Microcontrollers. In fact, the entire embedded systems industry is dependent on Microcontrollers. The following are few applications of Microcontrollers.

1. Front Panel Controls in devices like Oven, washing Machine etc.
2. Function Generators
3. Smoke and Fire Alarms
4. Home Automation Systems
5. Automatic Headlamp ON in Cars
6. Speed Sensed Door Locking System