UNIT 4
Input-Output, Exceptions and Interrupts

Input-Output

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

i. Methods of addressing I/O ports

ii. Instructions that cause I/O operations

iii. Protection to the use of I/O instructions and I/O port addresses.

i. 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)

1. Separate I/O address space (An Isolated I/O)

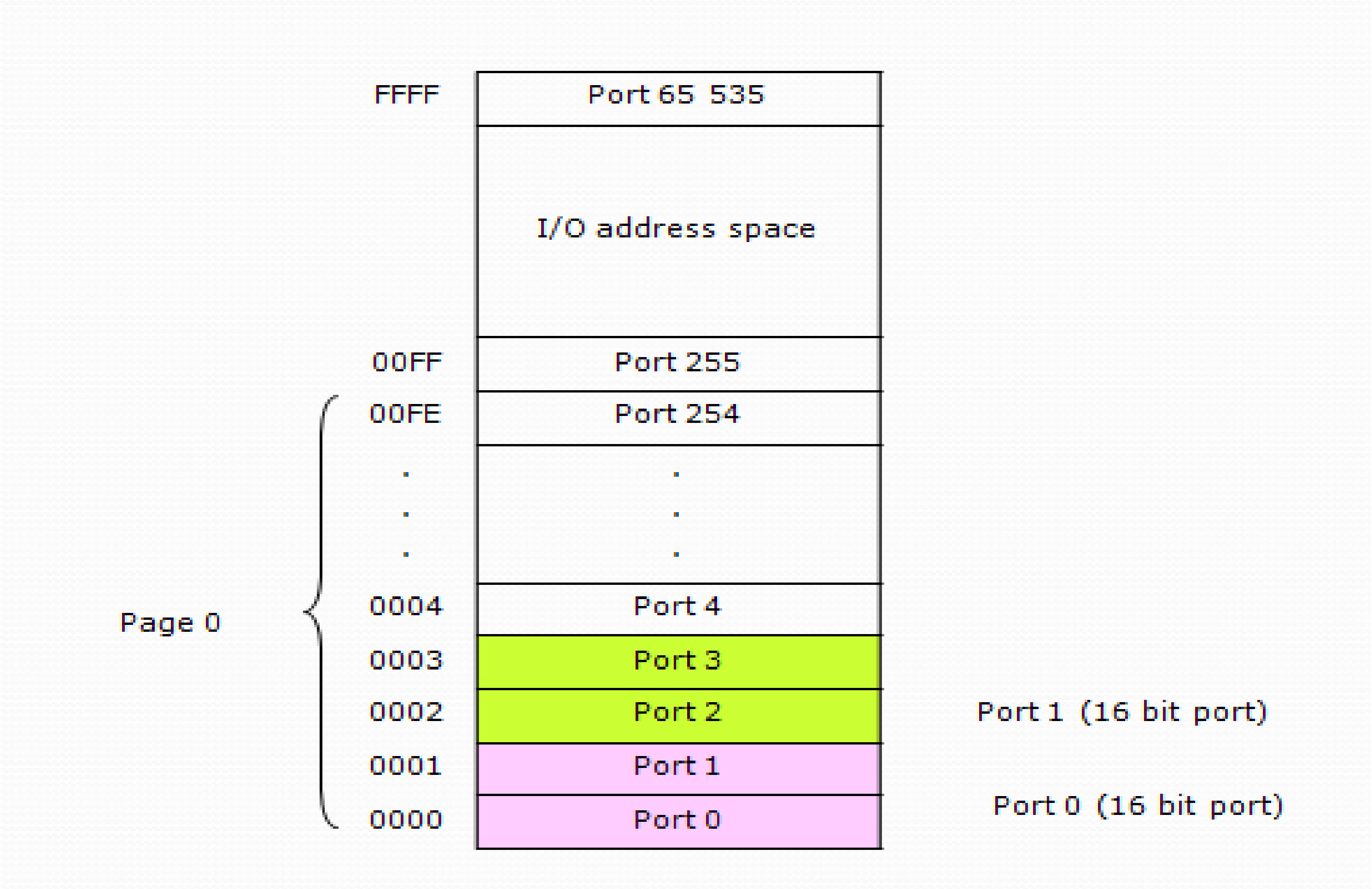
- **I/O devices treated separately from memory.**

- The 80386 provides a separate I/O address space, other from physical memory, that **can be used to address the input/output ports that are used for external 16 devices.**

- Can be accessed as either **byte-wide, word-wide, double word wide.**

- The I/O address space consists of
  - 2^{16} (64K) individually addressable 8-bit ports; **(64K 8-bit ports).**
  - Any two consecutive 8-bit ports can be treated as a 16-bit port; **(32K 16-bit ports).**
  - Any four consecutive 8-bit ports can be treated as a 32-bit port. **(16K 32-bit ports).**

The 80386 can transfer 32, 16, or 8 bits at a time to a device located in the I/O space.



- **Advantages:**

  - Provides a separate I/O address space, other from physical memory.
  - Special instructions have been provided in the instruction set to perform isolated I/O input and output operations.
  - These instructions have been tailored to maximize I/O performance.

- **Disadvantages:**

  - All input and output data transfers must take place between AL or AX register and the I/O port.

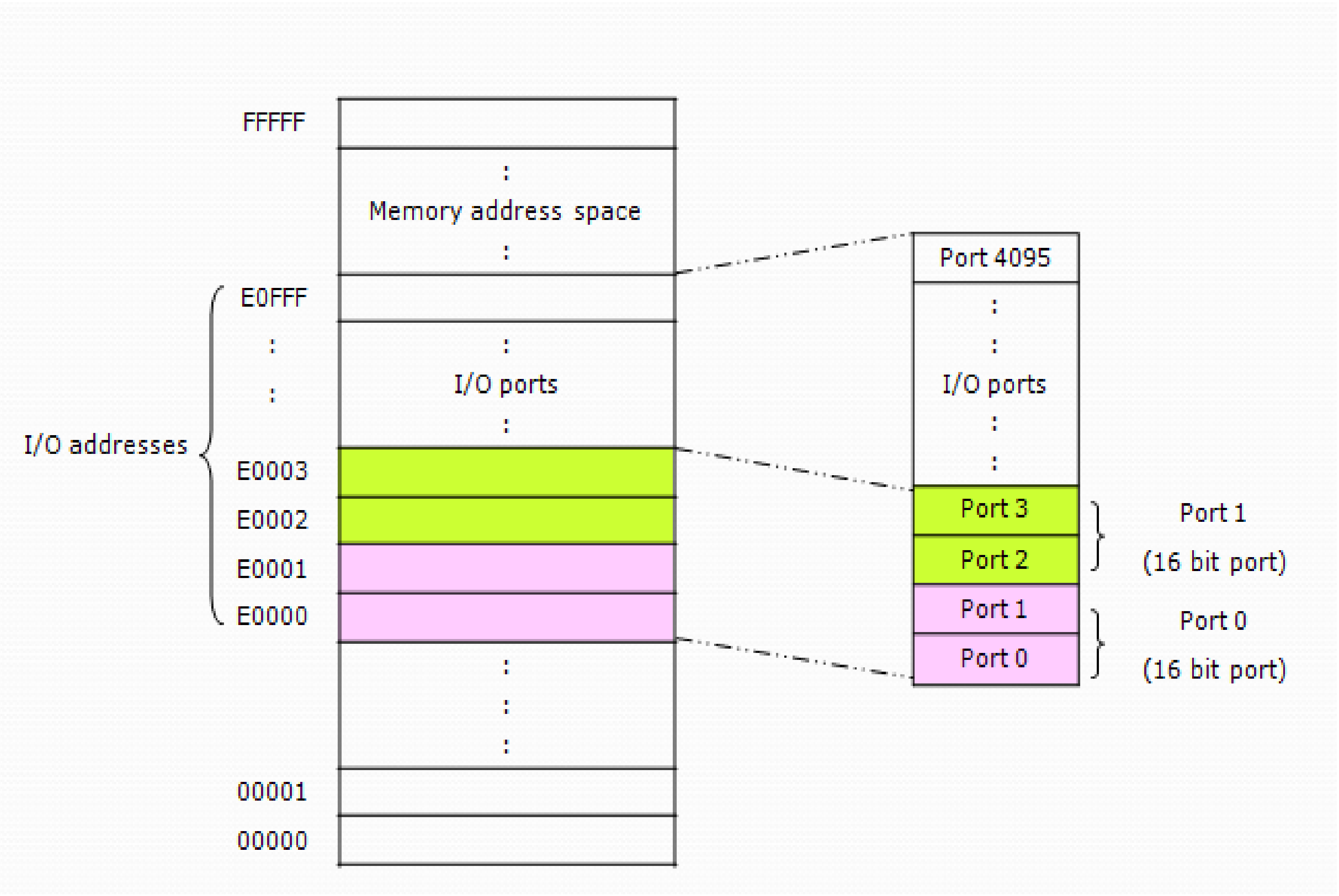
2. Memory-mapped I/O

- **I/O devices is placed in memory address space of the computer.**

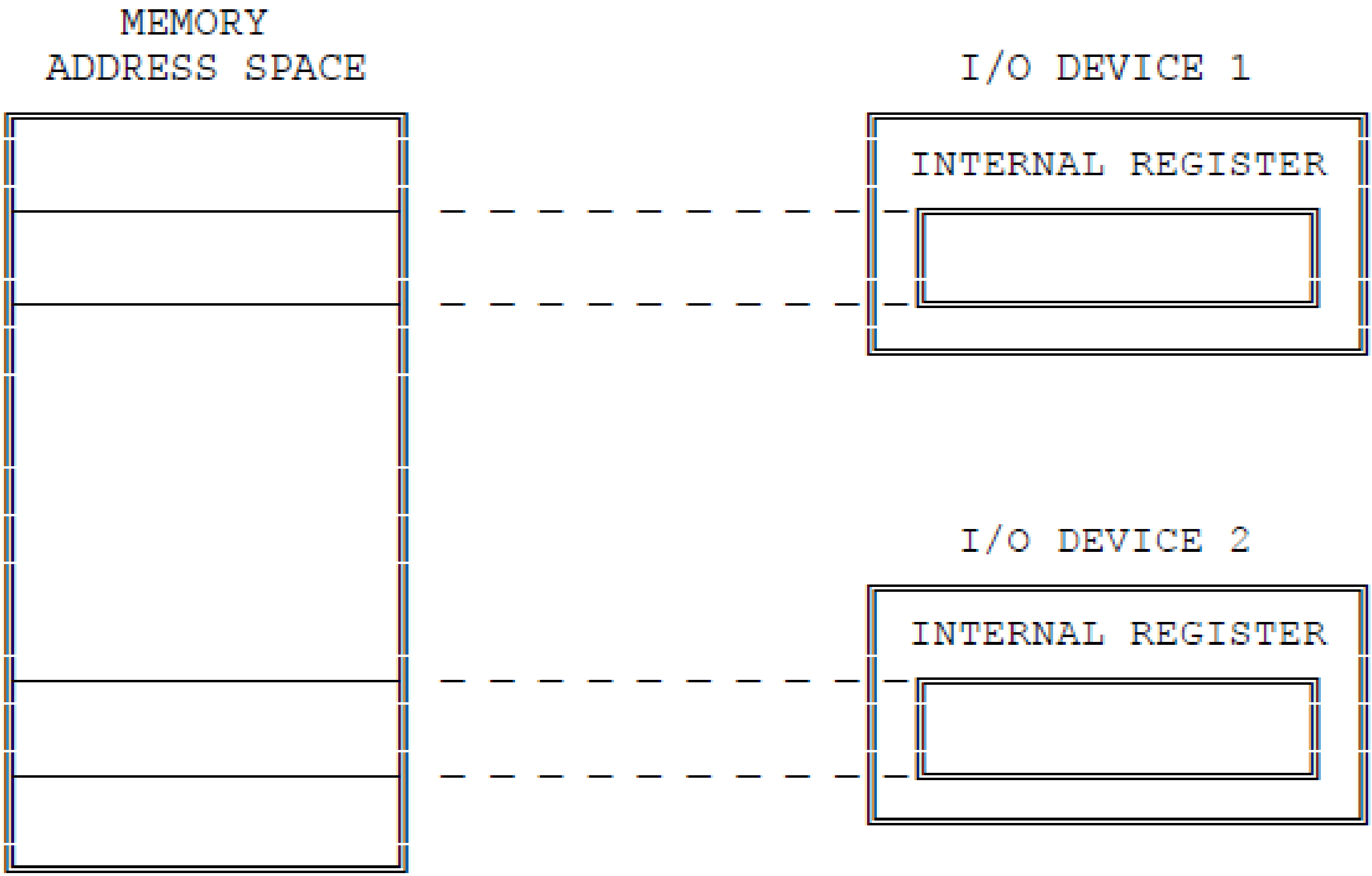
  - The memory address space is assigned to I/O devices.

  - MP looks at the I/O port as though it is a storage location in memory.

  - No special instructions are used.


FFFFF
:
Memory address space
:
E0FFF
:
:
I/O addresses
E0003
E0002
E0001
E0000
:
I/O ports
:
:
:
:
00001
00000
Port 4095
:
:
I/O ports
:
:
Port 3
Port 2
Port 1
Port 0
Port 1
(16 bit port)
Port 0
(16 bit port)

Memory Mapped I/O



- **Advantages**:

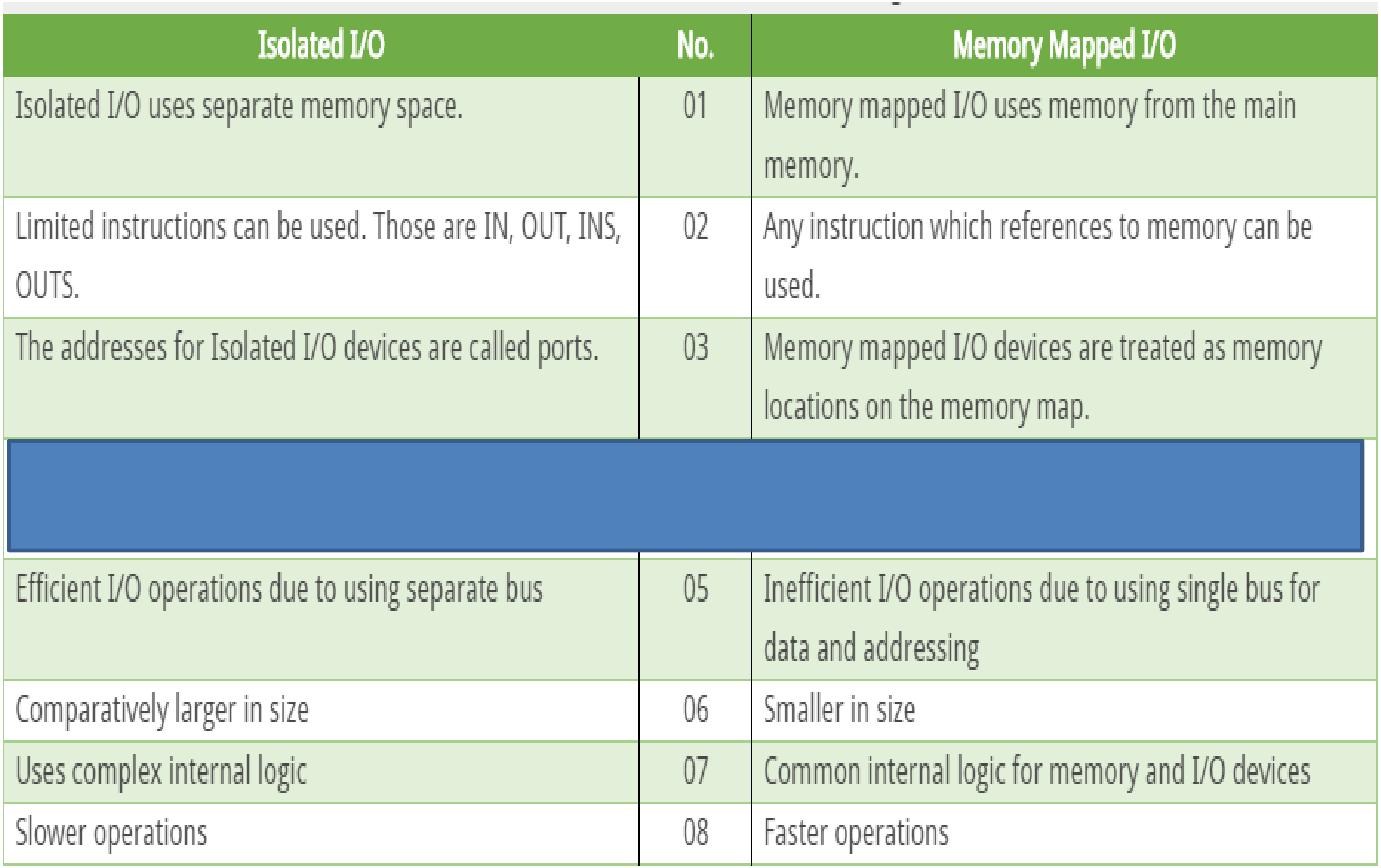
  - Many more instructions and addressing modes are available to perform I/O operations.

  - I/O transfers can now take place between I/O port and internal registers other than just AL/AX.

- **Disadvantages:**

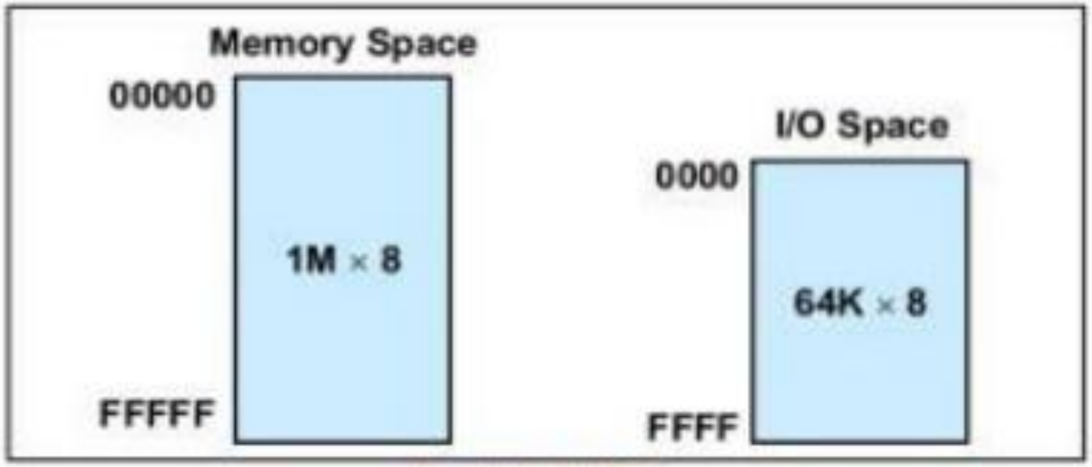
  - Memory instructions tend to execute slower than those specifically designed for isolated I/O.

  - Part of the memory address space is lost.

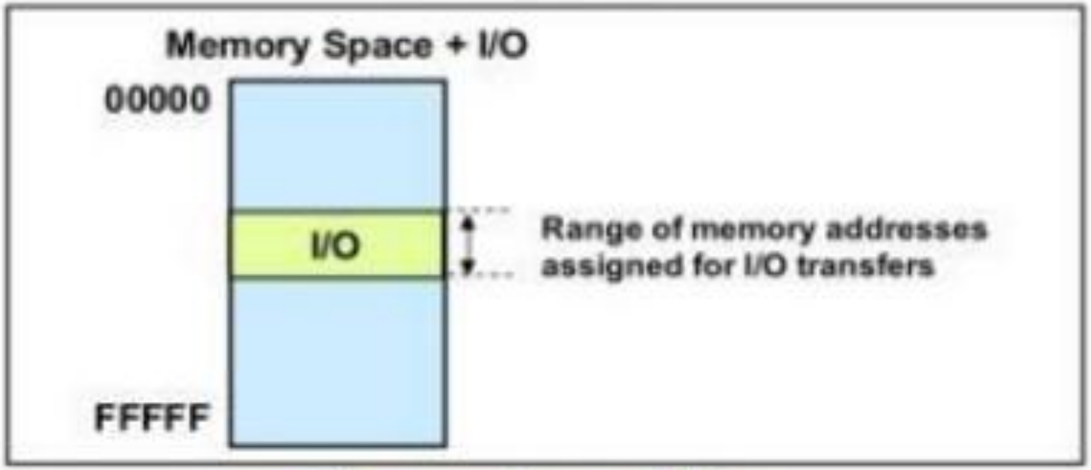


Isolated I/O	No.	Memory Mapped I/O
Isolated I/O uses separate memory space.	01	Memory mapped I/O uses memory from the main memory.
Limited instructions can be used. Those are IN, OUT, INS, OUTS.	02	Any instruction which references to memory can be used.
The addresses for Isolated I/O devices are called ports.	03	Memory mapped I/O devices are treated as memory locations on the memory map.
[redacted]	[redacted]	[redacted]
Efficient I/O operations due to using separate bus	05	Inefficient I/O operations due to using single bus for data and addressing
Comparatively larger in size	06	Smaller in size
Uses complex internal logic	07	Common internal logic for memory and I/O devices
Slower operations	08	Faster operations

Isolated vs. Memory Mapped I/O



Isolated I/O



Memory-Mapped I/O

ii. I/O Instructions

There are two classes of I/O instruction:

1. Those that transfer a single item (byte, word, or doubleword) located in a register. Known as **“Register I/O instructions”.**

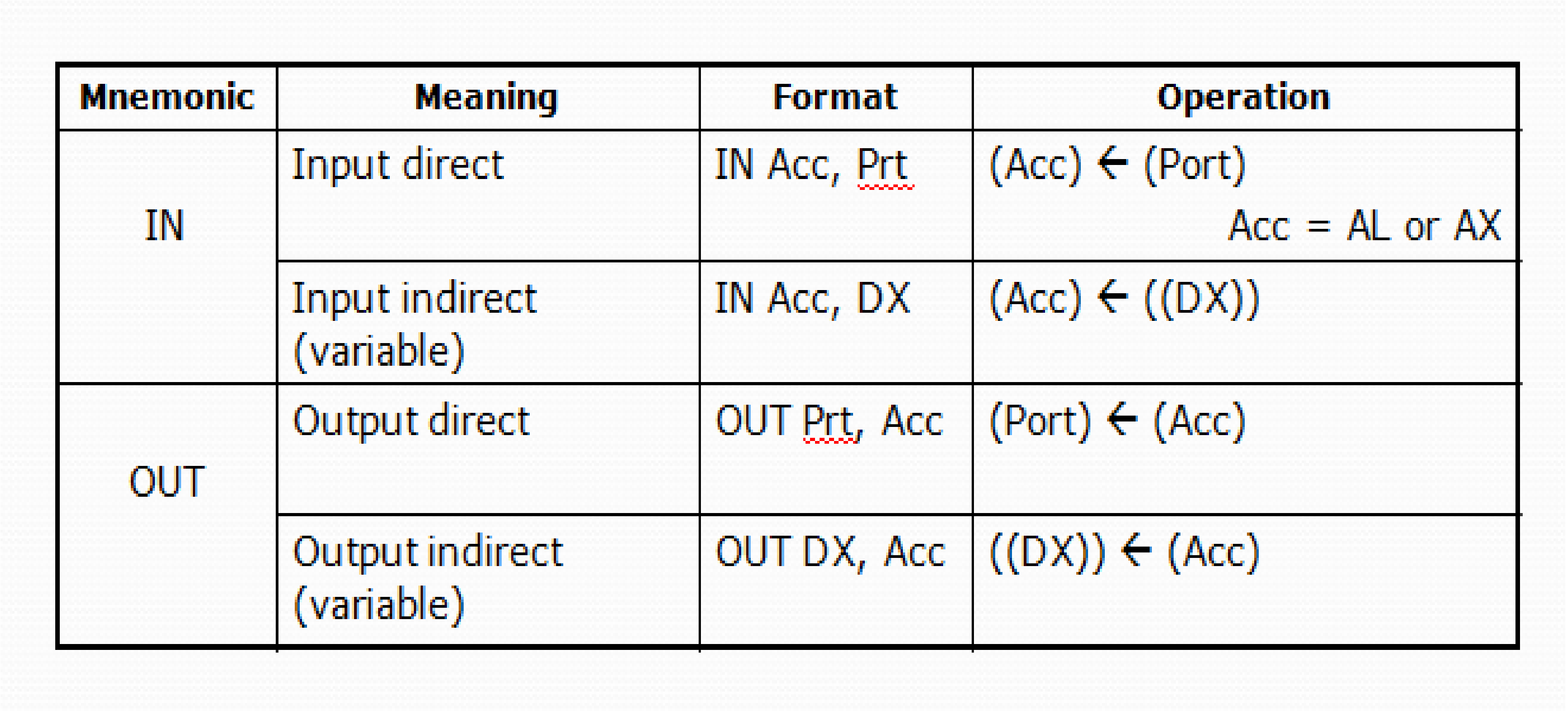
2. Those that transfer strings of items (strings of bytes, words, or doublewords) located in memory.

   These are known as **"String I/O Instructions" or "Block I/O Instructions".**

1. Register I/O Instructions

- The I/O instructions **IN and OUT** are provided to move data between I/O ports and the EAX (32-bit I/O), the AX (I6-bit I/O), or AL (8-bit I/O) general registers.
- IN and OUT instructions addresses 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.

Input Output Instructions



Mnemonic	Meaning	Format	Operation
IN	Input direct	IN Acc, Prt	(Acc) ← (Port) Acc = AL or AX
	Input indirect (variable)	IN Acc, DX	(Acc) ← ((DX))
OUT	Output direct	OUT Prt, Acc	(Port) ← (Acc)
	Output indirect (variable)	OUT DX, Acc	((DX)) ← (Acc)

2. 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

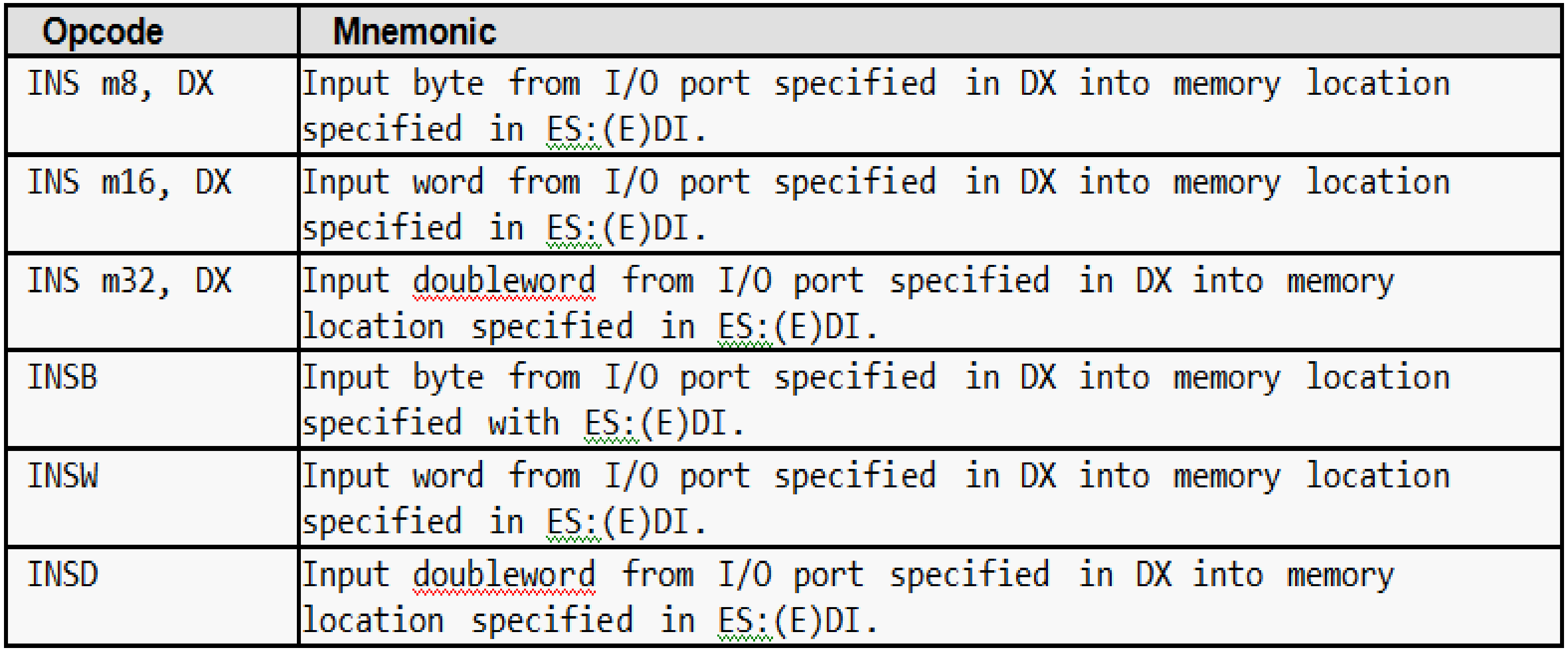
  - 16-bit ports numbered 0, 2,4, ... , 65532, 65534

  - 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.

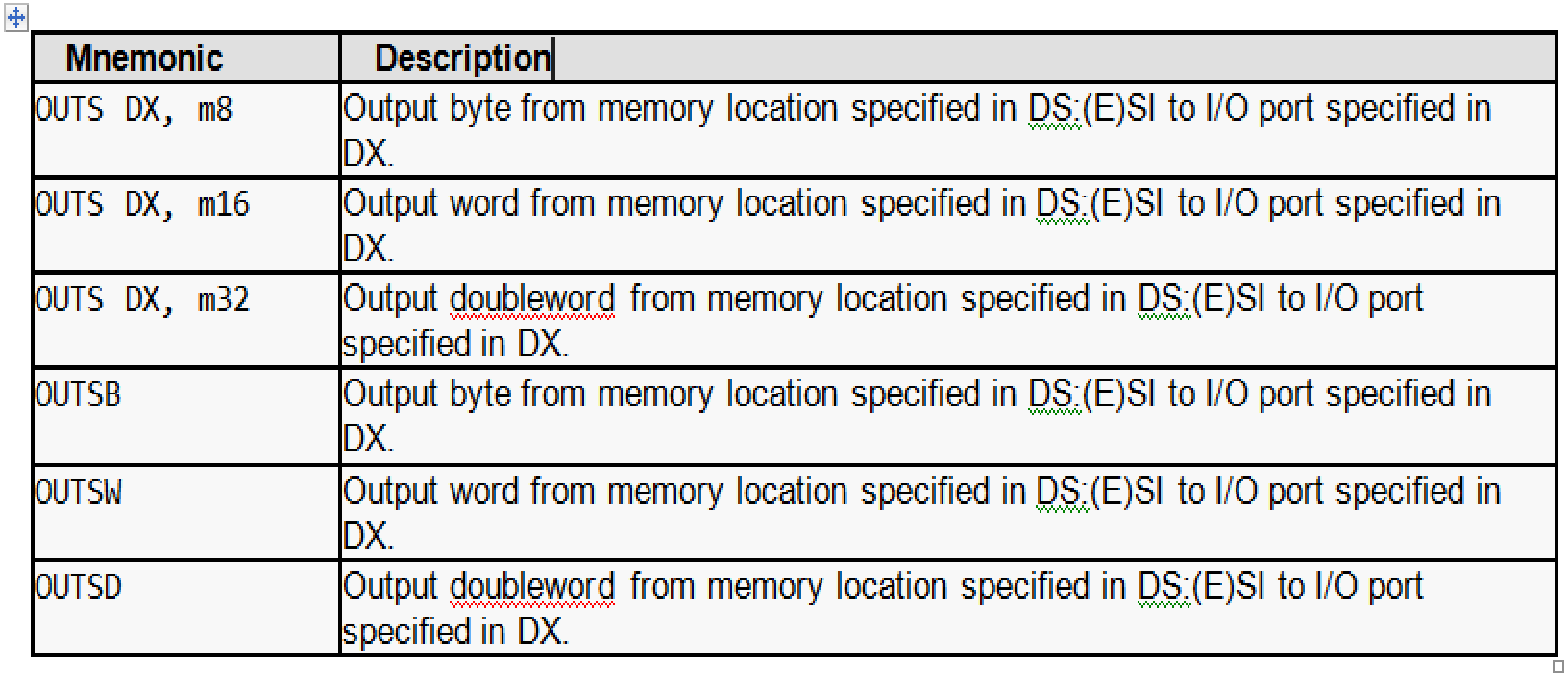
- For each transfer, **SI or DI are automatically either incremented or decremented** as specified by the direction bit in the flags register.

INS Instruction



Opcode	Mnemonic
INS m8, DX	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI.
INS m16, DX	Input word from I/O port specified in DX into memory location specified in ES:(E)DI.
INS m32, DX	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI.
INSB	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI.
INSW	Input word from I/O port specified in DX into memory location specified in ES:(E)DI.
INSD	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI.

OUTS Instruction



Mnemonic	Description
OUTS DX, m8	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX.
OUTS DX, m16	Output word from memory location specified in DS:(E)SI to I/O port specified in DX.
OUTS DX, m32	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX.
OUTSB	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX.
OUTSW	Output word from memory location specified in DS:(E)SI to I/O port specified in DX.
OUTSD	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX.

iii. 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 TSS segment** defines the right to use ports in the I/O address space.

- These **mechanisms operate only in protected mode and Virtual 8086 mode**; they do not operate in real mode.

- Since in real mode, **there is no protection of the I/O space.**

- So any procedure can execute I/O instructions, and any I/O port can be addressed by the I/O instructions.

1. I/O Privilege Level

- The IOPL from EFLAG register defines the privilege level needed to execute I/O-related instructions.

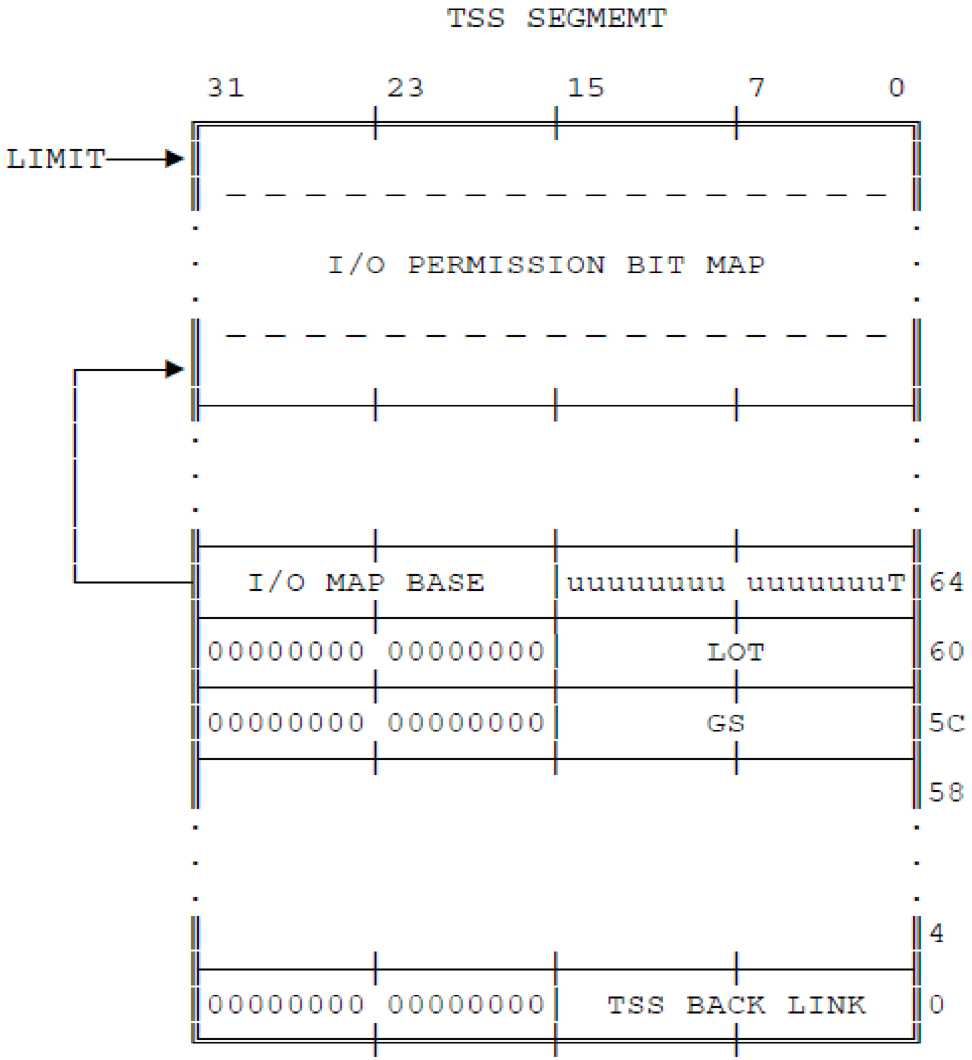
- The following instructions can be executed only if IOPL is highest privileged than CPL:
  - IN — Input
  - INS — Input String
  - OUT — Output
  - OUTS — Output String
  - CLI — Clear Interrupt-Enable Flag
  - STI — Set Interrupt-Enable

- These instructions are called "**Sensitive**" instructions, because they are sensitive to IOPL.

- To use sensitive instructions, IOPL must have a high privilege than CPL (Numerically less value).

- Any **attempt by a less privileged procedure to use a sensitive instruction results in a general protection exception.**

2. I/O Permission Bit Map



TSS SEGMEMT

31 23 15 7 0

LIMIT

I/O PERMISSION BIT MAP

I/O MAP BASE | uuuuuuuu uuuuuuuT | 64

00000000 00000000 | LOT | 60

00000000 00000000 | GS | 5C

58

4

00000000 00000000 | TSS BACK LINK | 0

- In protected mode, when it encounters an I/O instruction (IN, INS, OUT, or OUTS), the processor first checks whether CPL ≤ IOPL.

- If this condition is true, the I/O operation may proceed. If not true, the processor checks the I/O permission map.

- Each bit in the map are mapped to an I/O port byte address.

- The processor checks all bits from requested address with bits in I/O map, If all tested OK, then operation is continued, otherwise General Protection Exception is generated.

Exceptions and Interrupts

- Interrupts and exceptions are special kinds of control transfer; they work somewhat like unprogrammed CALLs.

- They alter the normal program flow to handle external events or to report errors or exceptional conditions.

- **Interrupts** are used to handle asynchronous events external to the processor.

- **Exceptions** handle conditions detected by the processor itself in the course of executing instructions.

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

1. Interrupts

- Maskable interrupts, which are signalled via the INTR pin.
- Nonmaskable interrupts, which are signalled via the NMI (Non-Maskable Interrupt) pin.

2. Exceptions

- Processor detected. These are further classified as **Faults, Traps and Aborts.**
- Programmed. The instructions **INT 0, INT 3, INT n, and BOUND** can trigger exceptions. These instructions are often called "Software Interrupts", but the processor handles them as exceptions.

Identifying Interrupts

- Each different type of **interrupt or exception have given a unique identification number.**

- 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; they are reserved for future.

- The **identifiers of the maskable interrupts are determined by external interrupt controllers (eg. 8259A 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.**

- **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.

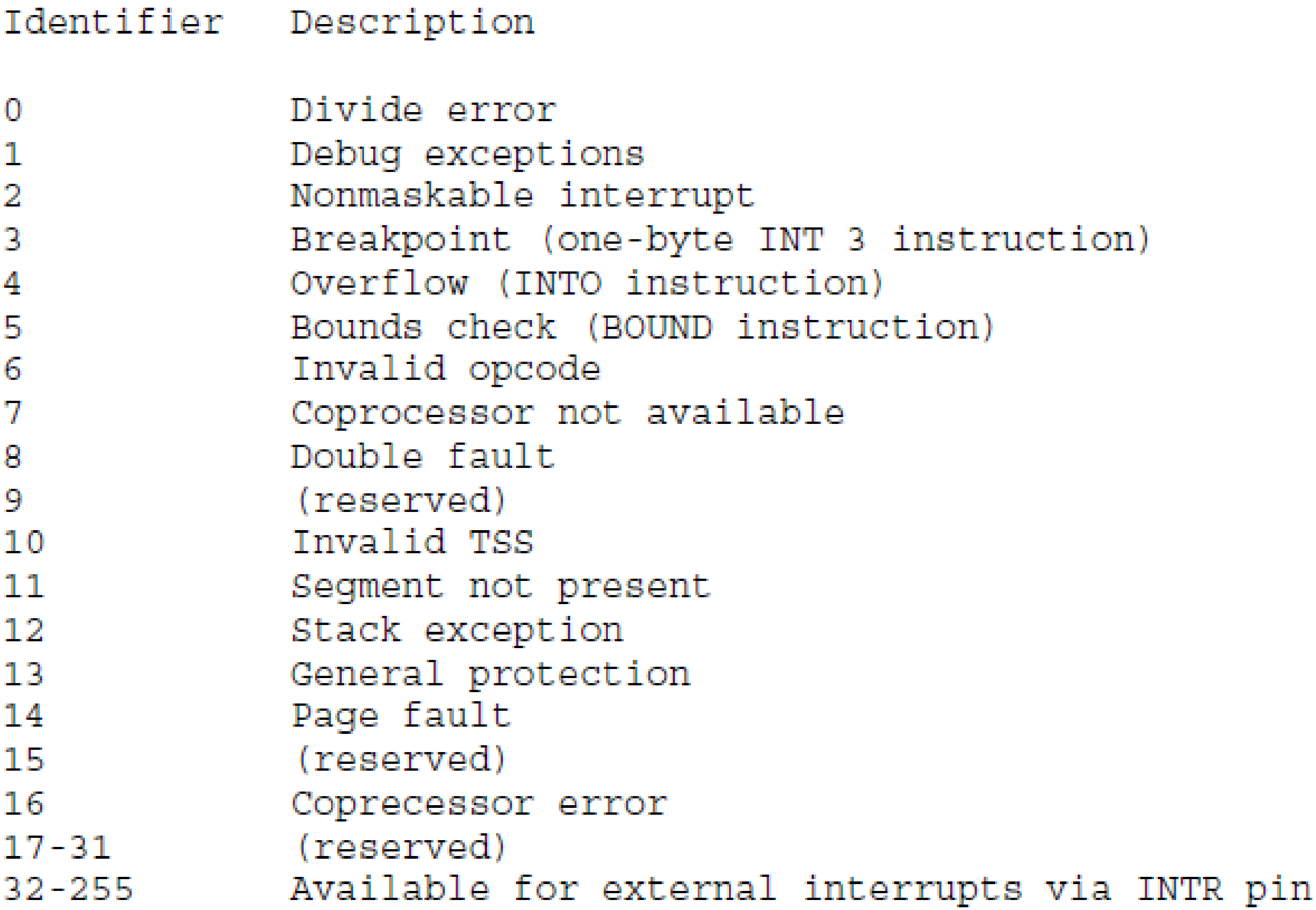
  - Faults are either detected before the instruction begins to execute, or during execution of the instruction.

  - If detected during the instruction, the instruction is automatically restarted by the system.

- **Traps**
  - A trap is an exception that is reported at the instruction boundary immediately after the instruction in which the exception was detected.
  - Trap happens after or during the execution of instruction.
  - Eg. Debugger breakpoint, Overflow situation, Division by zero or Invalid memory access.

- **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.
  - Eg. Blue screen error because of RAM or ROM.

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

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.

1. **NMI Masks Further NMIs**
2. **IF (Interrupt-enable Flag) Masks INTR**
3. **RF Masks Debug Faults**
4. **MOV or POP to SS Masks Some Interrupts and Exceptions**

1. 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.

2. IF Masks INTR

– The IF (Interrupt-enable Flag) controls the acceptance of external interrupts signalled via the INTR pin.

– 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 explicitly alter IF. These instructions may be executed only if 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 Service Sequence

1. External interface sends an interrupt signal, to the Interrupt Request (**INTR**) pin, or an internal interrupt occurs.

2. The CPU finishes the present instruction and sends Interrupt Acknowledge (**INTA**) to hardware interface.

3. The interrupt type N is sent to the Central Processor Unit (CPU) via the Data bus from the hardware interface.

4. The contents of the flag registers are pushed onto the stack.

5. Both the interrupt (IF) and (TF) flags are cleared. This disables the INTR pin and the trap or single-step feature.

6. The contents of the code segment register (CS) are pushed onto the Stack.

7. The contents of the instruction pointer (IP) are pushed onto the Stack.

8. The interrupt vector contents are fetched, and then placed into the IP and into the CS so that the next instruction executes at the interrupt service procedure addressed by the interrupt vector.

9. While returning from the interrupt-service routine by the Interrupt Return (IRET) instruction, the IP, CS and Flag registers are popped from the Stack and return to their state prior to the interrupt.

3. 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.

Note: **RF** (Resume Flag):

- The RF flag is used in conjunction with the debug register breakpoints.

- When RF is set, it causes any debug fault to be ignored on the next instruction.

4. 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, then 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.

- So, its better to use the 80386 LSS instruction, so 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.

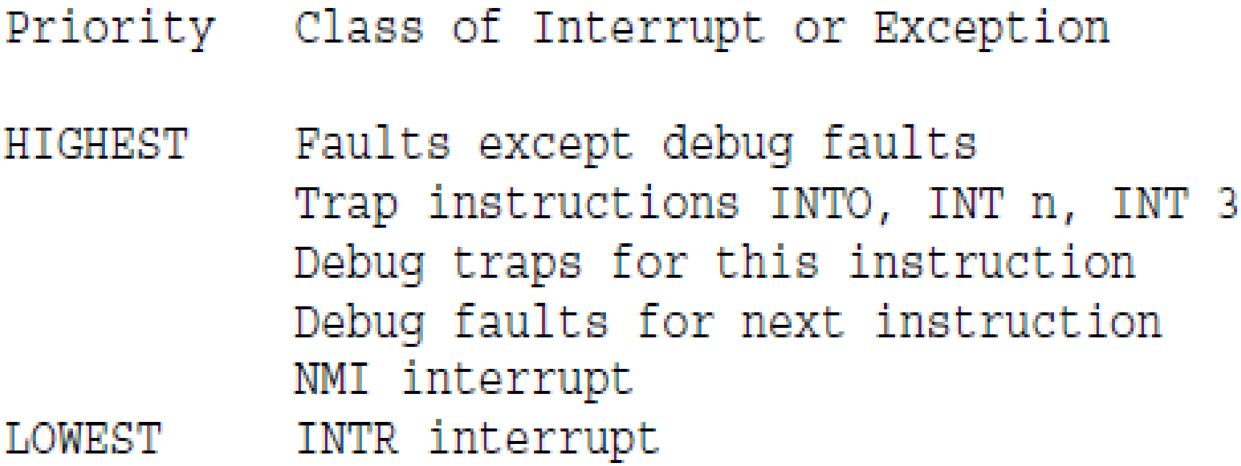
- The priority is assigned to classes of interrupt and exception sources.

- 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, or lower priority interrupts are held pending.

- Discarded exceptions will be rediscovered when the interrupt handler returns control to the point of interruption.

- **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.

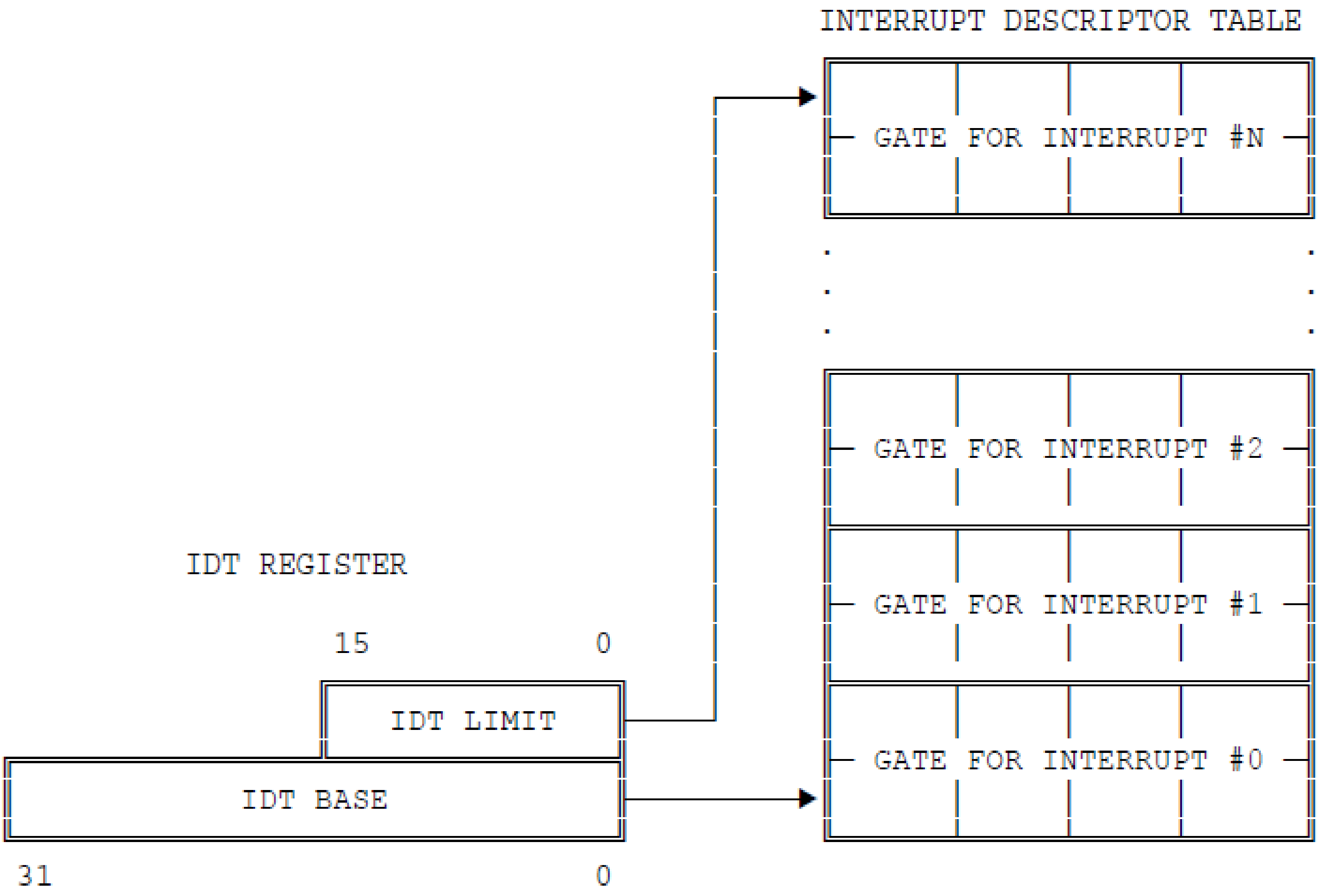
- Because there are only 256 identifiers, the IDT need not contain more than 256 descriptors.

- 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. 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.

- **IDT Register and Table**



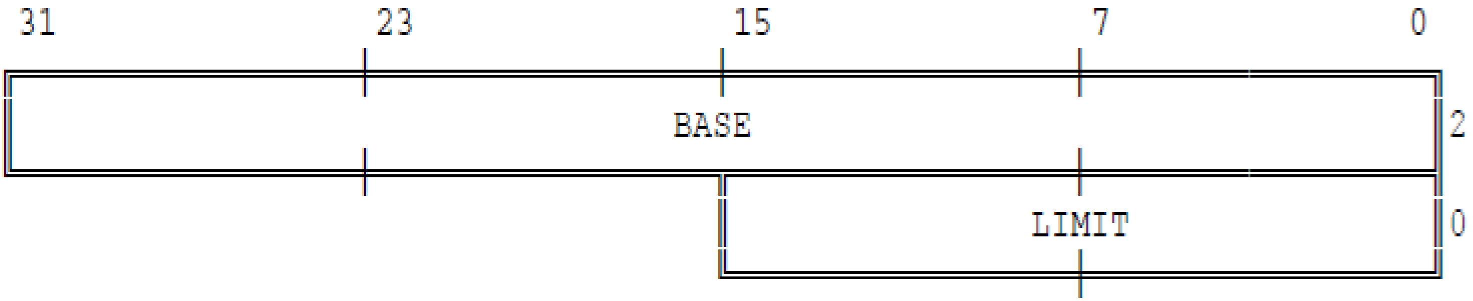
- LIDT (Load IDT register) loads the IDT register with the linear base address and limit values contained in the memory operand.

- 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.

- 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.

- **Pseudo-Descriptor Format for LIDT and SIDT**

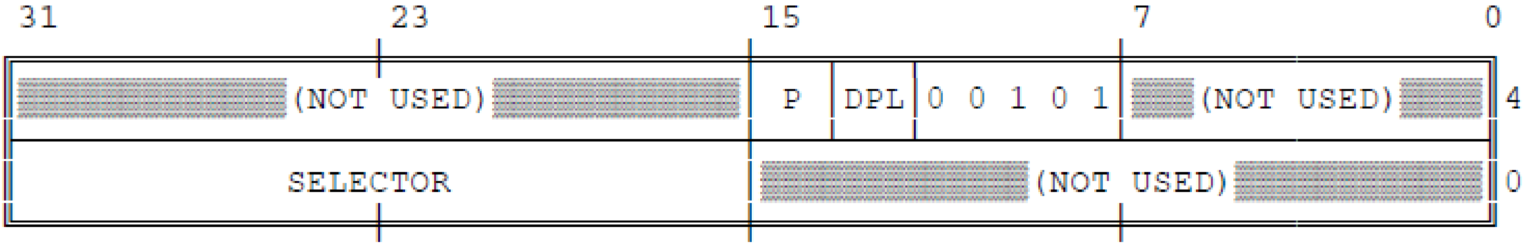


IDT Descriptors

- The IDT may contain any of three kinds of descriptor:
  - **Task gates**
  - **Interrupt gates**
  - **Trap gates**

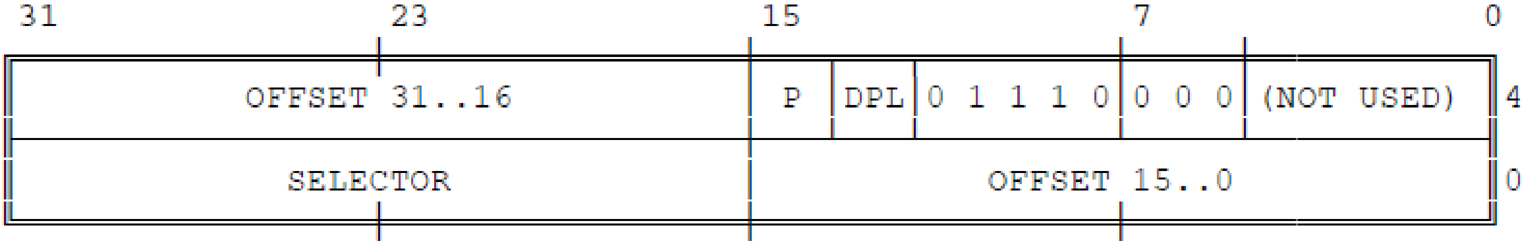
80386 IDT Gate Descriptors

80386 TASK GATE



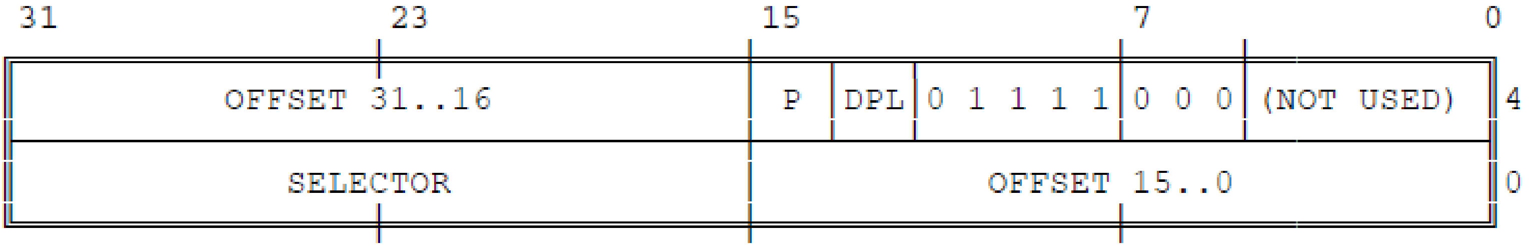
```
31              23              15              7               0
|▒▒▒▒▒(NOT USED)▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒| P |DPL|0 0 1 0 1|▒(NOT USED)▒▒| 4
|           SELECTOR            |▒▒▒▒▒▒▒▒▒(NOT USED)▒▒▒▒▒▒▒▒▒▒▒| 0
```

80386 INTERRUPT GATE



```
31              23              15              7               0
|          OFFSET 31..16        | P |DPL|0 1 1 1 0|0 0 0|(NOT USED)| 4
|           SELECTOR            |          OFFSET 15..0            | 0
```

80386 TRAP GATE



```
31              23              15              7               0
|          OFFSET 31..16        | P |DPL|0 1 1 1 1|0 0 0|(NOT USED)| 4
|           SELECTOR            |          OFFSET 15..0            | 0
```

Interrupt Tasks and Interrupt Procedures

- Using “CALL” instruction, interrupt or exception call a interrupt handler.

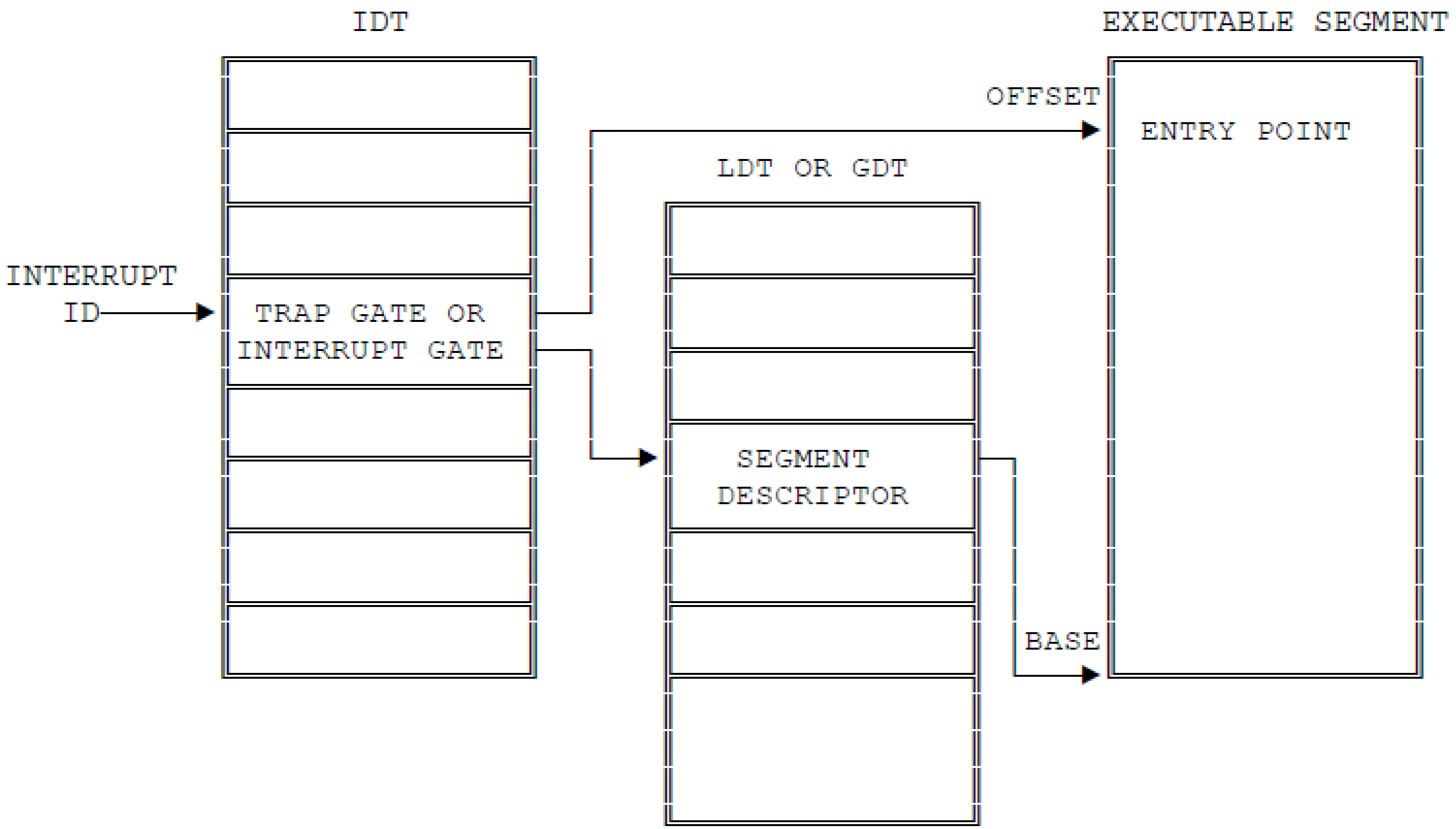
- The processor uses the interrupt or exception identifier to index a descriptor in the IDT.

- 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.

1. Interrupt Procedures

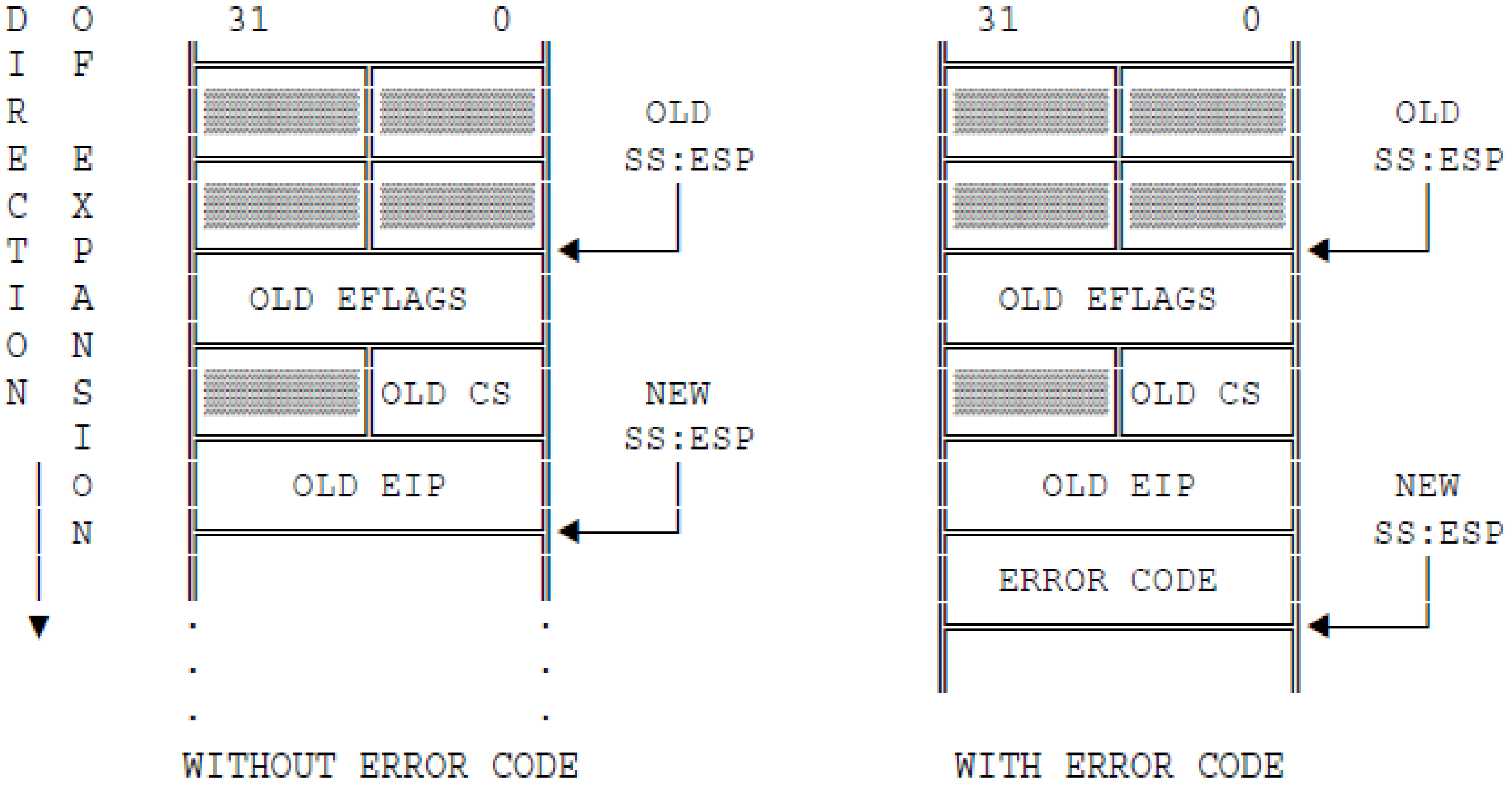
- **Interrupt Vectoring for Procedures**



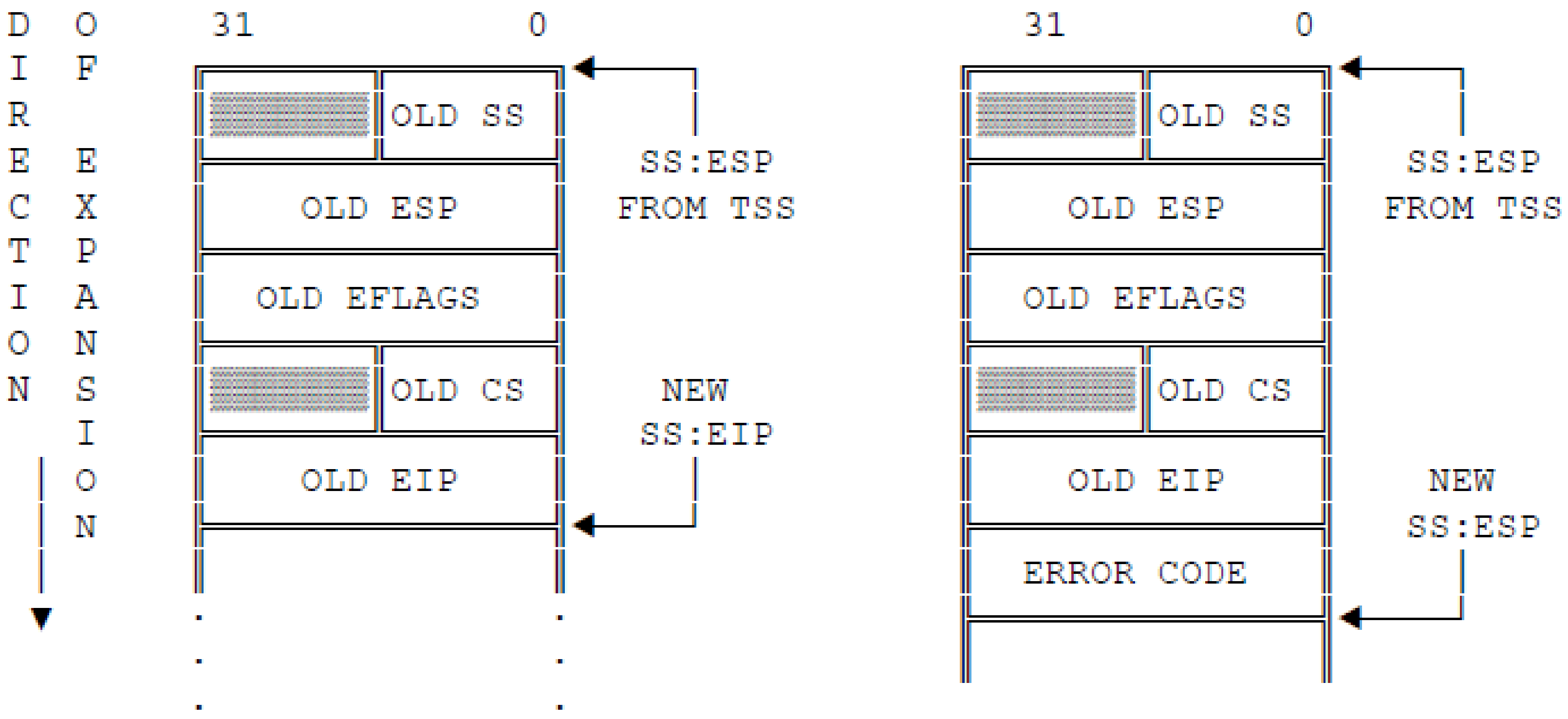
1.1 Stack of Interrupt Procedure

- 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.**
- Interrupt pushes the EFLAGS register onto the stack before the pointer to the interrupted instruction.
- **If any error code is generated, it is also pushed on the stack.**
- An exception handler can use the error code to help diagnose the exception.

WITHOUT PRIVILEGE TRANSITION



WITH PRIVILEGE TRANSITION



DIRECTION OF EXPANSION

31 0

OLD SS

OLD ESP

OLD EFLAGS

OLD CS

OLD EIP

SS:ESP FROM TSS

NEW SS:EIP

WITHOUT ERROR CODE

31 0

OLD SS

OLD ESP

OLD EFLAGS

OLD CS

OLD EIP

ERROR CODE

SS:ESP FROM TSS

NEW SS:ESP

WITH ERROR CODE

1.2 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.

1.3 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.

- 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.**

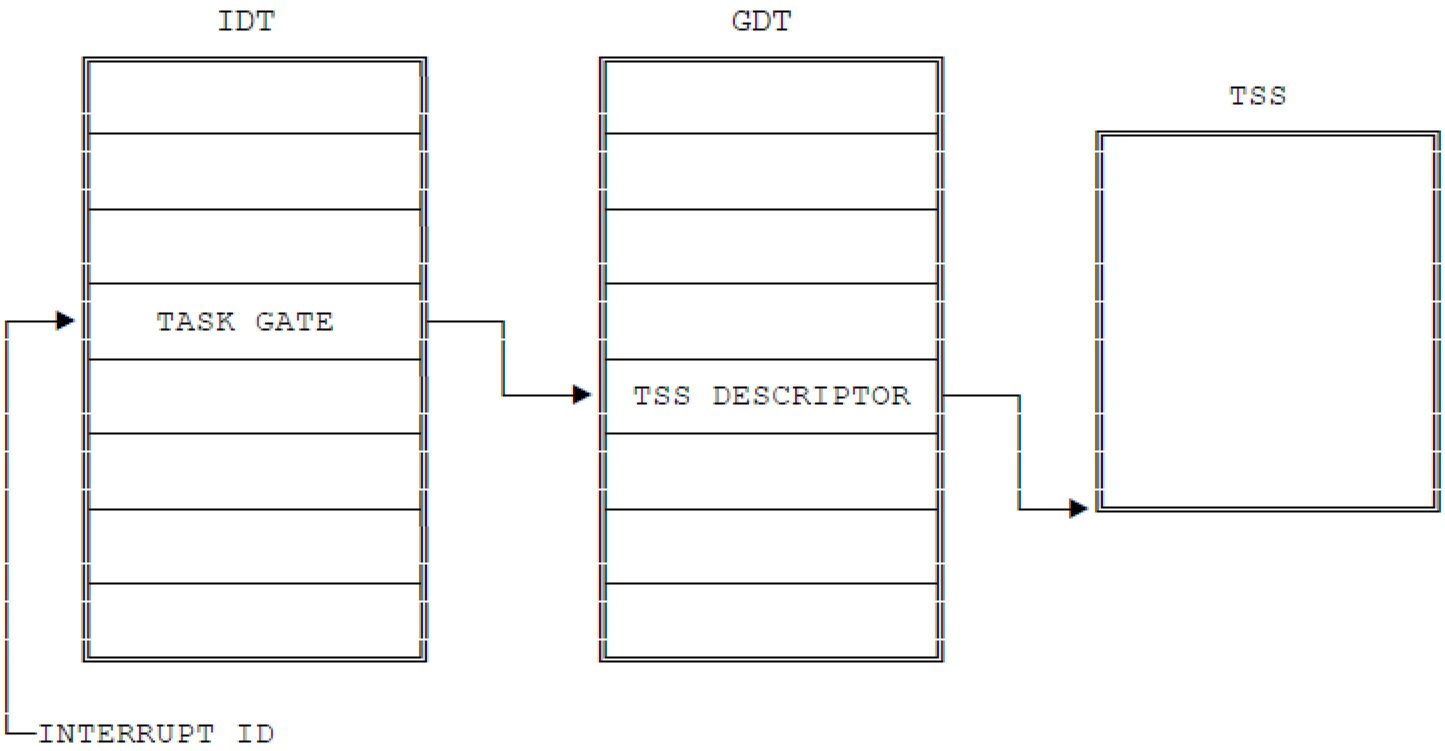
1.4 Protection in Interrupt Procedures

- The privilege rules for **interrupt procedures is similar to that for procedure calls.**
- **CPU does not permit an interrupt to transfer control to a procedure in a segment of lesser privilege** 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.

- Either of the following strategies can be employed to ensure that the privilege rule is never violated.
    - **Place the handler in a conforming segment.** This strategy suits the handlers for certain exceptions (eg. divide error). 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.
    - Place the handler procedure in a privilege level zero segment.

2. 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:

  – The entire context is saved automatically.

  – 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.

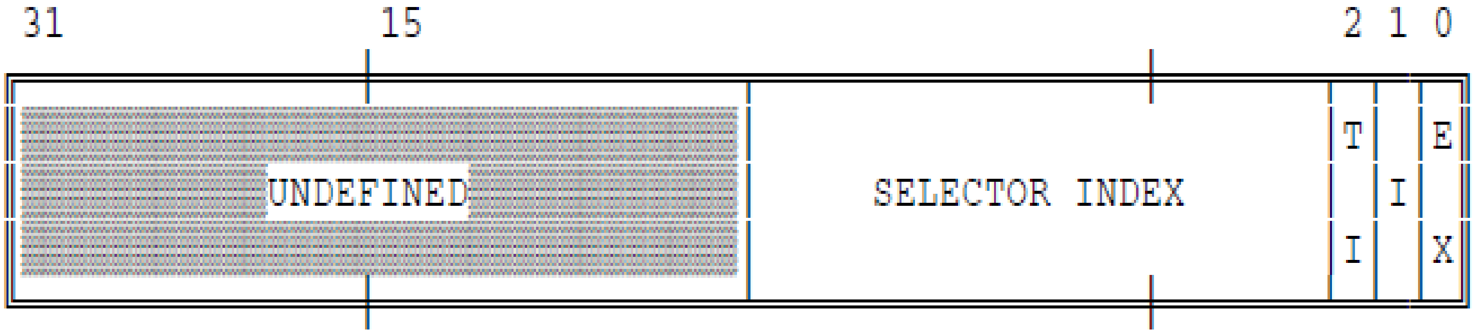
Error Code

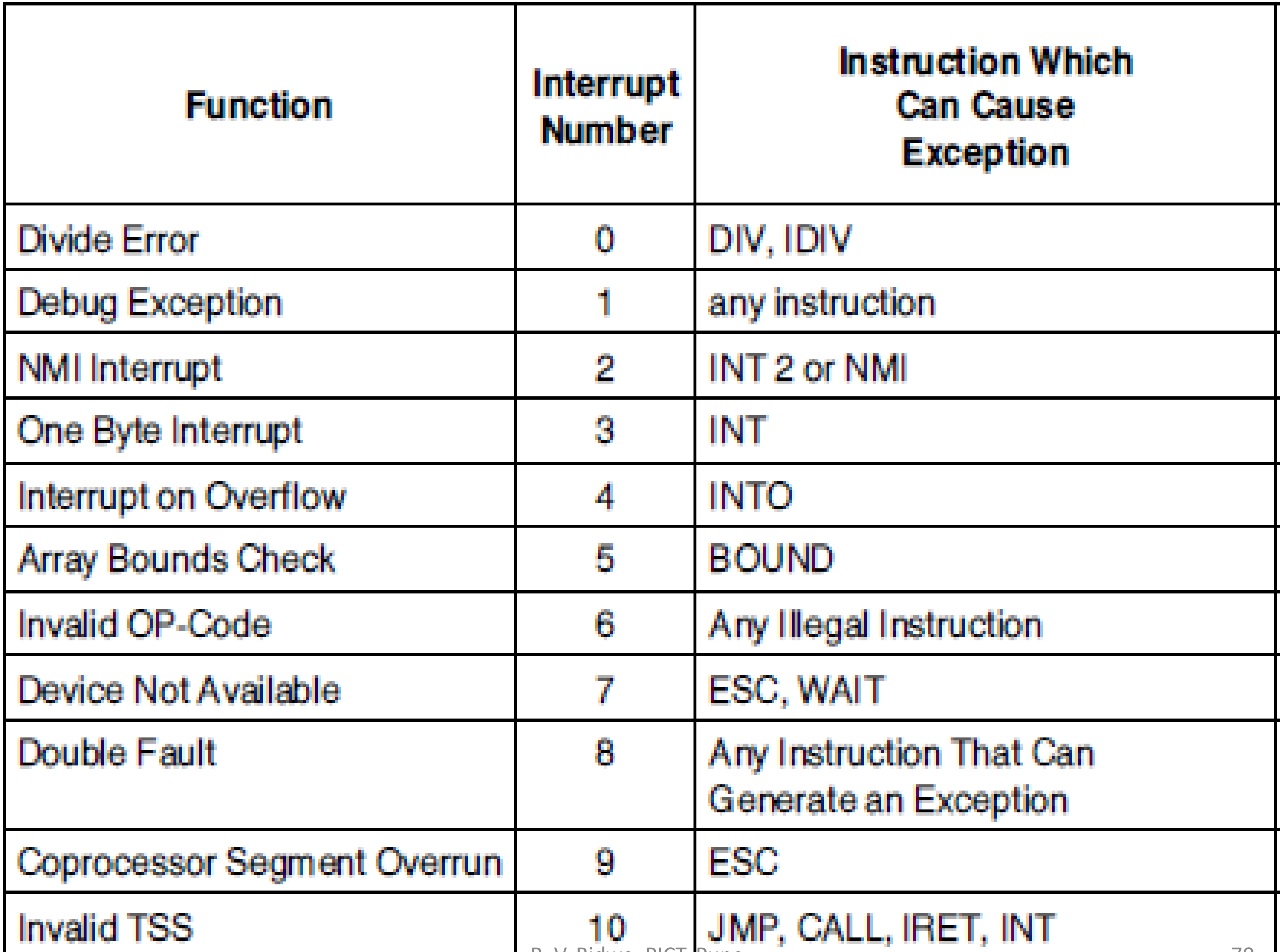
- With exceptions that relate to a specific segment, the processor pushes an error code onto the stack of the exception handler (whether procedure or task).

- The format of the error code resembles that of a selector; however, instead of an RPL field, the error code contains two one-bit items:

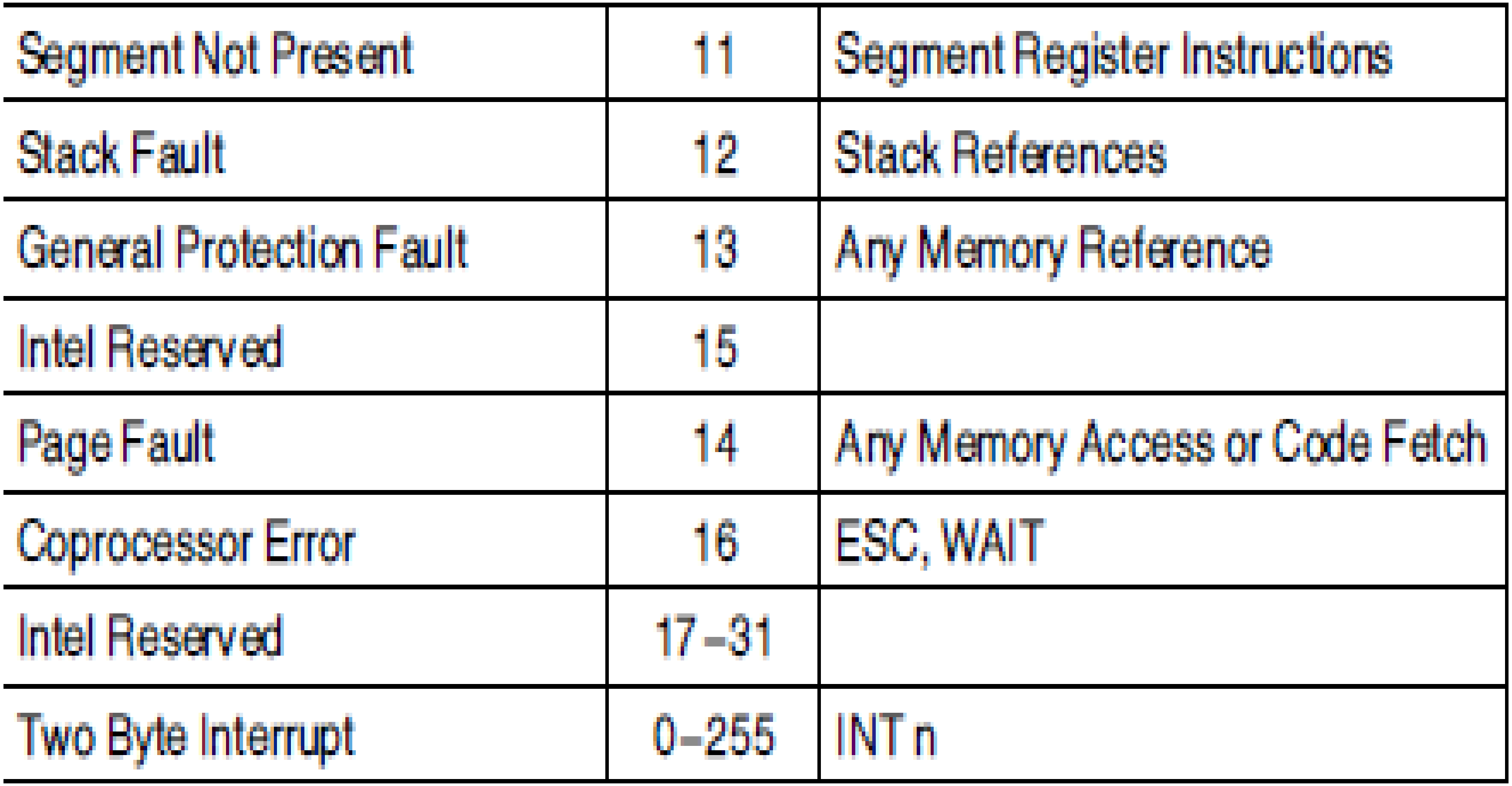
1. The processor sets the **EXT bit** if an event external to the program caused the exception.
2. The processor sets the **I-bit (IDT-bit)** if the index portion of the error code refers to a gate descriptor in the IDT.

- 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.





Function	Interrupt Number	Instruction Which Can Cause Exception
Divide Error	0	DIV, IDIV
Debug Exception	1	any instruction
NMI Interrupt	2	INT 2 or NMI
One Byte Interrupt	3	INT
Interrupt on Overflow	4	INTO
Array Bounds Check	5	BOUND
Invalid OP-Code	6	Any Illegal Instruction
Device Not Available	7	ESC, WAIT
Double Fault	8	Any Instruction That Can Generate an Exception
Coprocessor Segment Overrun	9	ESC
Invalid TSS	10	JMP, CALL, IRET, INT



Segment Not Present	11	Segment Register Instructions
Stack Fault	12	Stack References
General Protection Fault	13	Any Memory Reference
Intel Reserved	15	
Page Fault	14	Any Memory Access or Code Fetch
Coprocessor Error	16	ESC, WAIT
Intel Reserved	17–31	
Two Byte Interrupt	0–255	INT n

Divide Error (INT 0)

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

Debug Exceptions (INT 1)

- 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. (attempt is made to use the debug registers at the same time that 80386 is using them) (BD)
  - Single-step Trap. (BS)
  - Task-switch Breakpoint Trap. (BT)

- 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.**

Breakpoint (INT 3)

- Exceptions generated because of breakpoints.

Overflow (INT 4)

- This trap occurs when the processor encounters an INTO instruction and the OF (overflow) flag is set. {**INTO – Used to interrupt the program during execution if OF = 1**}

- 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.

Bounds Check (INT 5)

- This fault occurs when the processor, while **executing a BOUND instruction**, finds that the operand exceeds the specified limits.

- **BOUND ensures that a signed array index is within the limits** specified by a block of memory consisting of an upper and a lower bound.

Invalid op-code (INT 6)

- The Invalid Opcode exception occurs when the **processor tries to execute an invalid or undefined opcode**, or an instruction with invalid prefixes.

- This exception also occurs when the **type of operand is invalid for the given opcode.** Example include an intersegment JMP referencing a register operand.

No Math Unit Available (INT 7)

- It is triggered whenever a floating point instruction is being executed with the **EM (Emulate Processor Extension)** and **TS bit of MSW are set.**

- It can also be generated when a WAIT instruction is detected with both **MP (Monitor Processor Extension)** and **TS bits of MSW are set.**

- Wait instruction causes a processor to wait till coprocessor completes its task.

Double Fault Exception (INT 8)

- A **Double Fault** exception occurs if the processor encounters a problem while trying to service a pending interrupt or exception.

- An example situation when a double fault would occur is **when an interrupt is triggered but the segment in which the interrupt handler resides is invalid.**

- If the processor encounters a problem when calling the double fault handler, a **Triple Fault** is generated and the processor shuts down

Coprocessor Segment Overrun (INT 9)

- 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.

- **Math Coprocessor** is also known as **NPX, NDP, FPU**. Numeric Processor Extension (**NPX**), Numeric Data Processor (**NDP**), Floating Point Unit (**FPU**).

Invalid Task State Segment (INT 10)

- Its generated by the following conditions:

I. A **illegal back link** in a Task State Segment.
II. The TSS containing an **illegal CS,DS,ES or FS** value.
III. The TSS indicating an **invalid privileged stack** is not valid during inter-level call.
IV. The **TSS is too small**.
V. An **invalid or not present LDT** in a TSS.

Not Present (INT 11)

- It is generated by trying to load the CS,DS,ES,SS,FS,GS or a task register with an operand that is valid except for **being marked Not Present**. (P bit from descriptor is reset)

Stack Exception (INT 12)

- 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.
  - When attempting to load the SS register with a **descriptor that is marked not-present** but is otherwise valid.

General Protection (INT 13)

- It is **activated for all protection exceptions that are not specifically covered by other exceptions.**

A. A jump to data segment with high privilege level.
B. Writing to **read only segment**.
C. Attempting to address a memory location with an offset address that exceed the limit for the specified segment (**Invalid Offset**).
D. Putting an address into SS for a read only segment when the address come from a Task State segment.

Page Fault (INT 14)

- It is triggered only in **Protected or Virtual** mode, is generated when page fault occurs.

- The returned error code contains following information:

a) **If bit 0 is reset**, the exception was generated by a <u>Page that is not present</u>.

b) **If bit 0 set**, the exception was generated by a <u>Page Level Violation</u>.

c) **If bit 1 is reset**, the exception was caused by an illegal read access.

d) **If bit 1 is set**, the exception was caused by an illegal write access.

e) **If bit 2 is reset**, the exception was generated at user level.

f) **If bit 2 is set**, the exception was generated at supervisor level.