

---

# **Assembly Language and Microcomputer Interface**

## **Chapter 6: Program Control Instructions**

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology  
Zhejiang University

# Introduction

- This chapter explains the program control instructions, including the jumps, calls, returns, interrupts, and machine control instructions.
- This chapter also presents the relational assembly language statements (.IF, .ELSE, .ELSEIF, .ENDIF, .WHILE, .ENDW, .REPEAT, and .UNTIL) that are available in version 6.xx and above of MASM or TASM, with version 5.xx set for MASM compatibility.

# Chapter Objectives

**Upon completion of this chapter, you will be able to:**

- Use both conditional and unconditional jump instructions to control the flow of a program.
- Use the relational assembly language statements `.IF`, `.REPEAT`, `.WHILE`, and so forth in programs.
- Use the call and return instructions to include procedures in the program structure.

# Chapter Objectives

(*cont.*)

**Upon completion of this chapter, you will be able to:**

- Explain the operation of the interrupts and interrupt control instructions.
- Use machine control instructions to modify the flag bits.
- Use ENTER and LEAVE to enter and leave programming structures.

# 6–1 THE JUMP GROUP

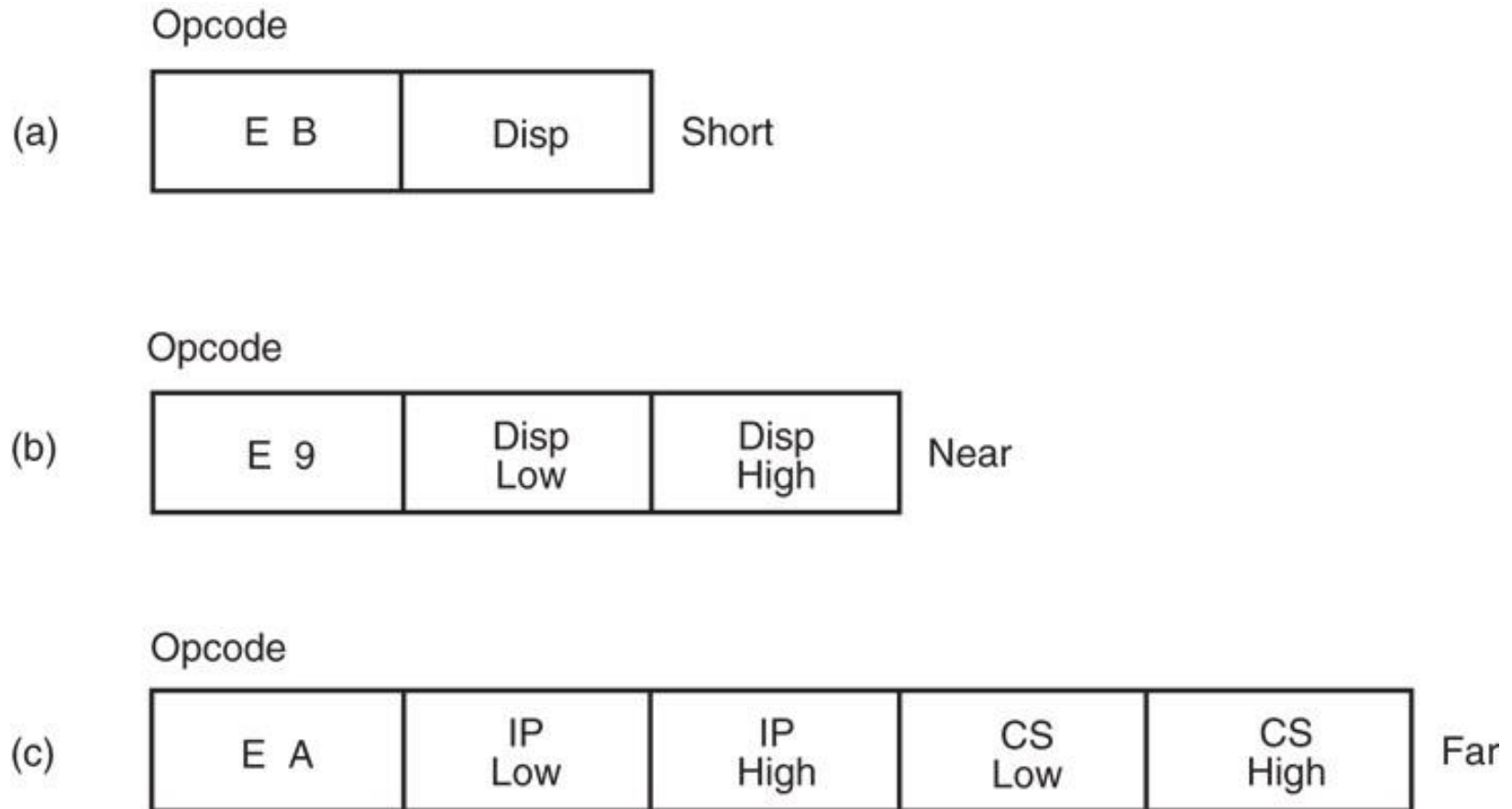
- Allows programmer to skip program sections and branch to any part of memory for the next instruction, by either unconditional or conditional jump.
- A **conditional jump instruction** allows decisions based upon numerical tests.
  - results are held in the flag bits, then tested by conditional jump instructions
- LOOP and conditional LOOP are also forms of the jump instruction.

# Unconditional Jump (JMP)

- Three types: **short jump**, **near jump**, **far jump**.
- **Short jump** is a 2-byte instruction that allows jumps or branches to memory locations within +127 and –128 bytes.
  - from the address following the jump
- 3-byte **near jump** allows a branch or jump within  $\pm 32K$  bytes from the instruction in the current code segment.

- 5-byte **far jump** allows a jump to any memory location within the real memory system.
- The short and near jumps are often called **intra-segment jumps**.
- Far jumps are called **inter-segment jumps**.

**Figure 6–1** The three main forms of the JMP instruction. Note that Disp is either an 8- or 16-bit signed displacement or distance.

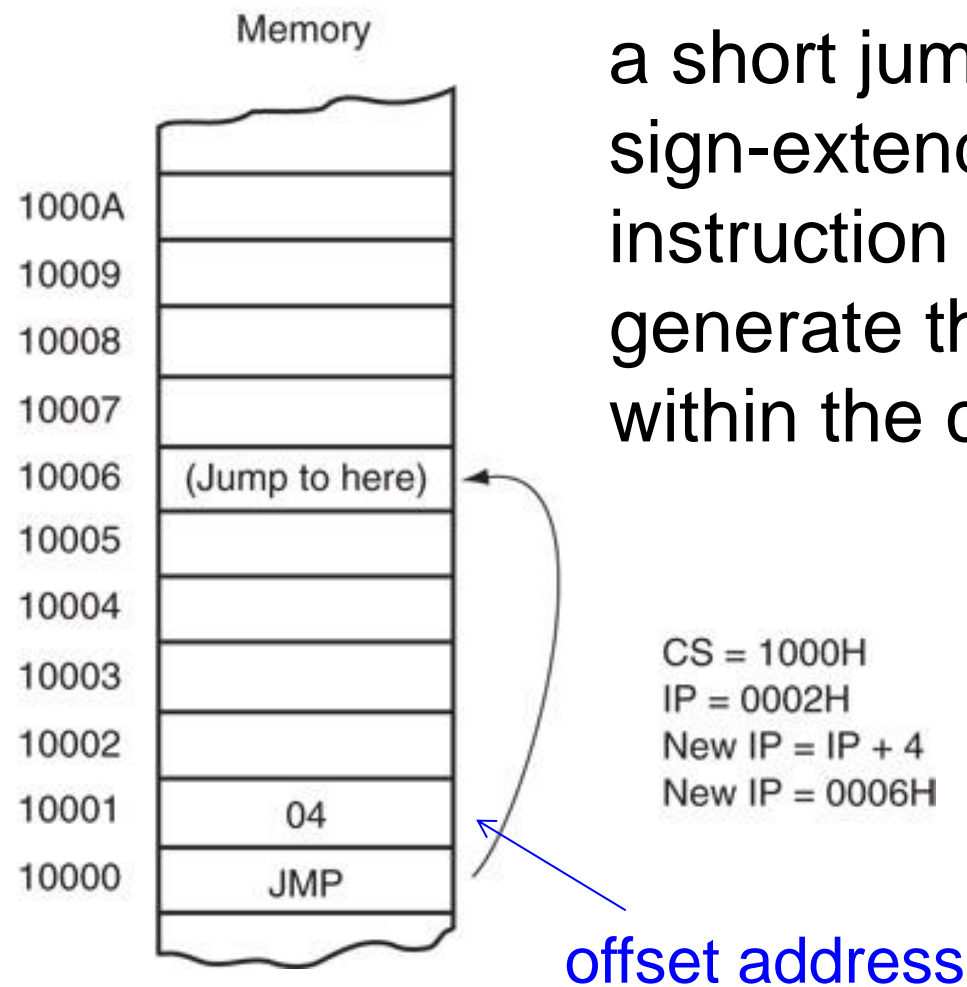




# *Short Jump*

- Called **relative jumps** because they can be moved, with related software, to any location in the current code segment without a change.
  - jump address is not stored with the opcode
  - a **distance**, or displacement, follows the opcode
- The short jump displacement is a distance represented by a 1-byte signed number whose value ranges between +127 and –128.
- Short jump instruction appears in Figure 6–2.

**Figure 6–2** A short jump to four memory locations beyond the address of the next instruction.



- when the microprocessor executes a short jump, the displacement is sign-extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment

- The instruction branches to this new address for the next instruction in the program

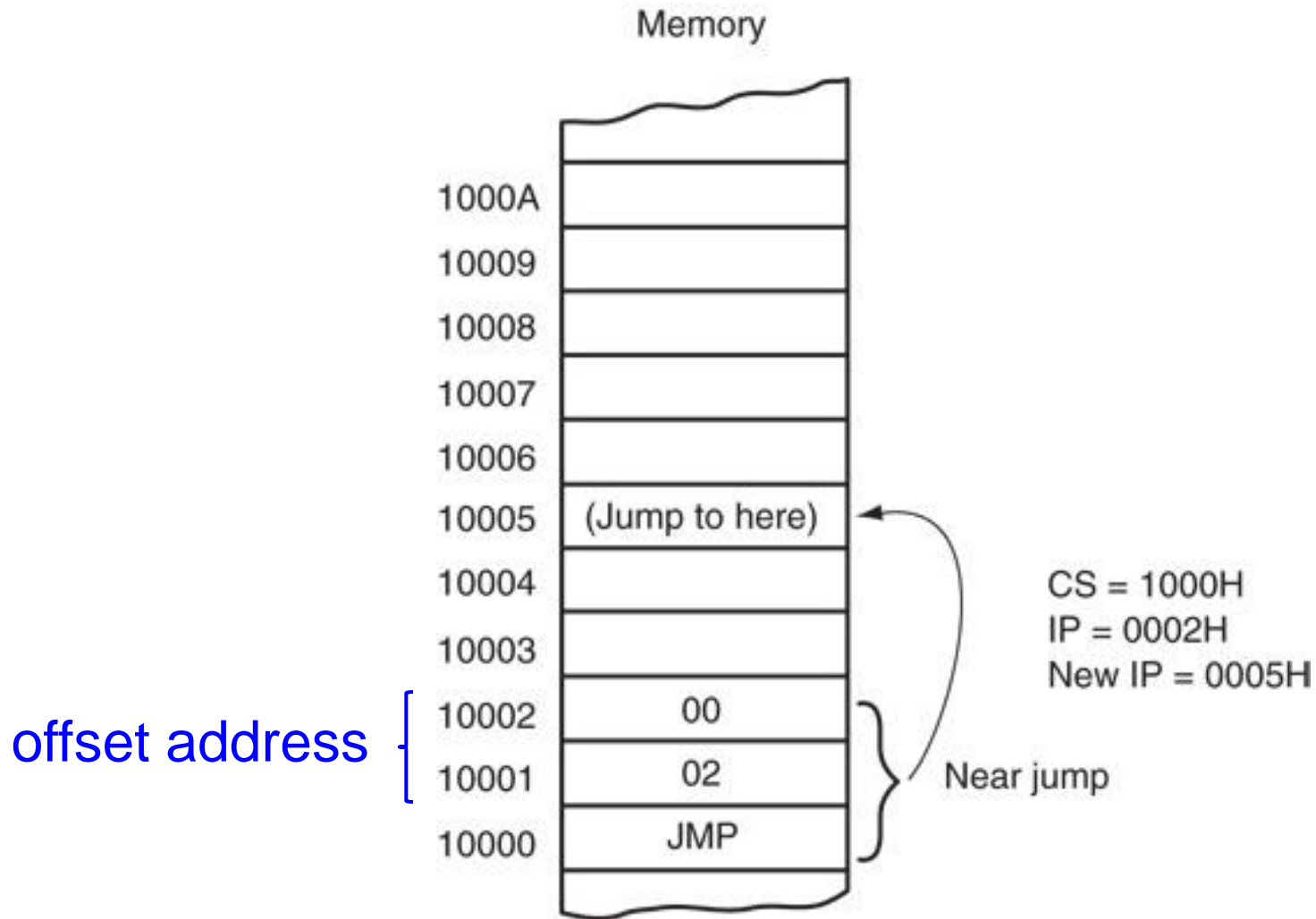
- When a jump references an address, a label normally identifies the address.
- The **JMP NEXT** instruction is an example.
  - it jumps to label NEXT for the next instruction
  - very rare to use an actual hexadecimal address with any jump instruction
- The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it
  - if a colon does not follow, you cannot jump to it
- The only time a colon is used is when the label is used with a jump or call instruction.

# *Near Jump*

- In real mode, near jump is a 3-byte instruction with opcode followed by a signed 16-bit displacement.
- A near jump passes control to an instruction in the current code segment located within  $\pm 32K$  bytes from the near jump instruction.
- 80386 - Pentium 4 displacement is 32 bits and the near jump is 5 bytes long.
- The distance is  $\pm 2G$  in 80386 and above when operated in protected mode.

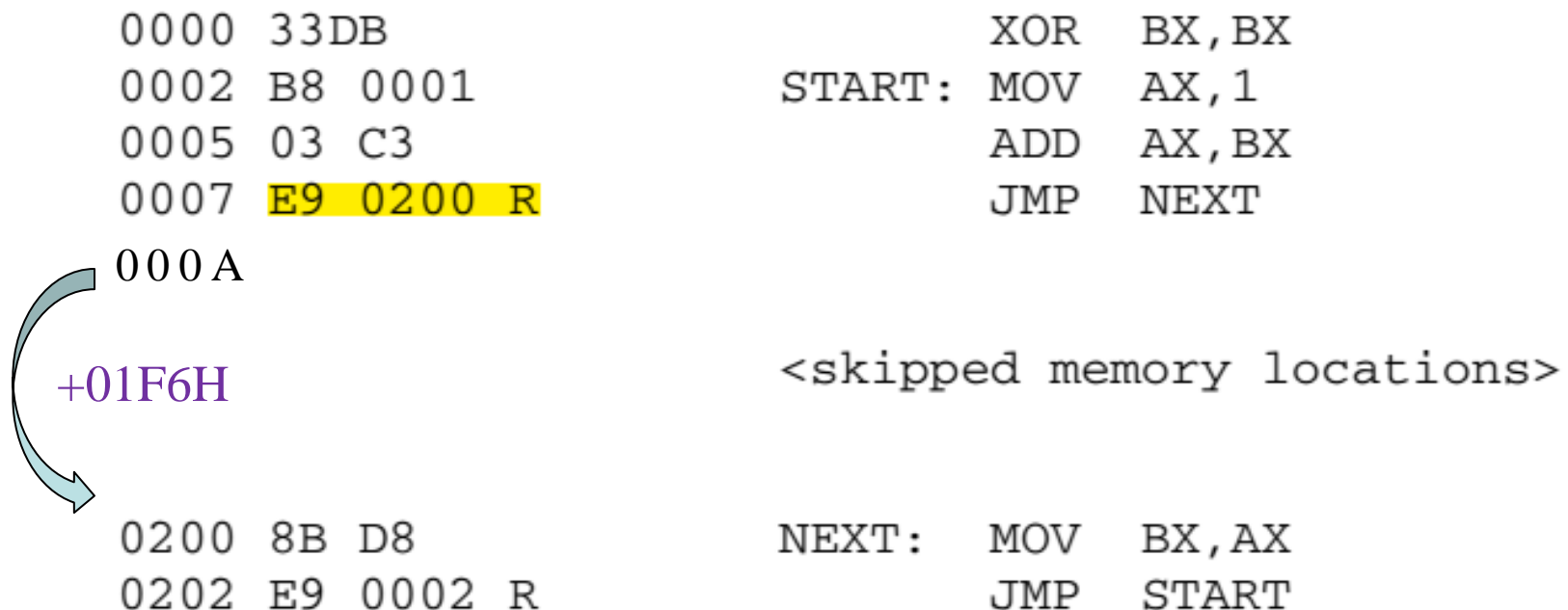
- Signed displacement adds to the instruction pointer (IP) to generate the jump address.
  - because signed displacement is  $\pm 32K$ , a near jump can jump to any memory location within the current real mode code segment
- The protected mode code segment in the 80386 and above can be 4G bytes long.
  - 32-bit displacement allows a near jump to any location within  $\pm 2G$  bytes
- Figure 6–3 illustrates the operation of the real mode near jump instruction.

**Figure 6–3** A near jump that adds the displacement (0002H) to the contents of IP.



- The near jump is also relocatable because it is also a relative jump.
- This feature, along with the relocatable data segments, Intel microprocessors ideal for use in a general-purpose computer system.
- Software can be written and loaded anywhere in the memory and function without modification because of the relative jumps and relocatable data segments.

- The following example shows a program with a near jump.
- The first jump (JMP NEXT) passes control to the instruction at offset memory location 0200H within the code segment.
- Notice that the instruction assembles as E9 0200 R. The letter R denotes a **relocatable jump address** of 0200H.
- After linking, the jump instruction appears as E9 F6 01 (01F6H is the actual displacement).

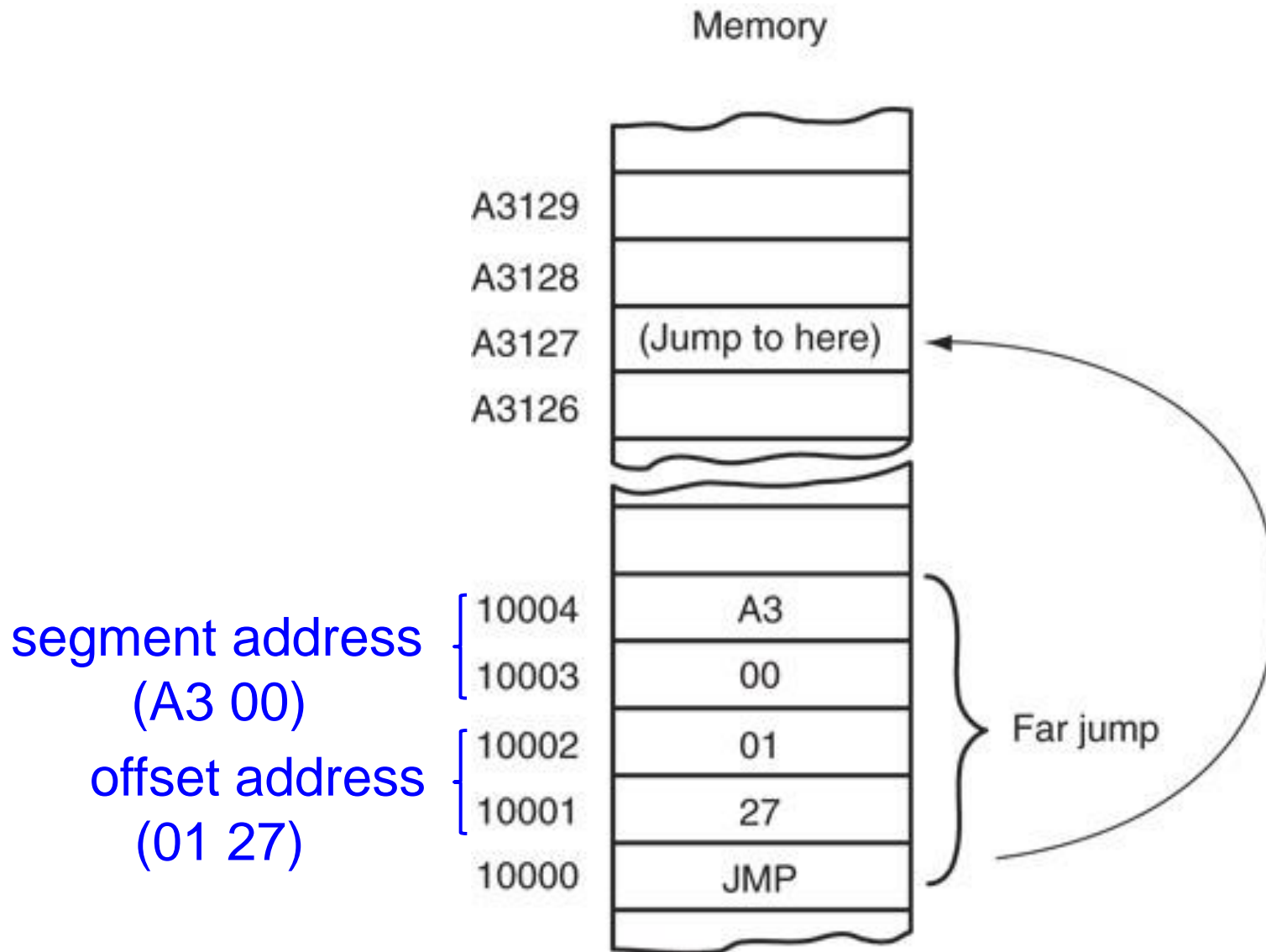




# ***Far Jump***



- Obtains a new segment and offset address to accomplish the jump.
- In real mode, it is a 5-byte instruction
  - bytes 2 and 3 contain the new offset address
  - bytes 4 and 5 contain the new segment address
- In protected mode
  - the segment address accesses a descriptor with the base address of the far jump segment
  - offset address, either 16 or 32 bits, contains the offset address within the new code segment


**Figure 6–4** A far jump instruction replaces the contents of both CS and IP with 4 bytes following the opcode.



- Two ways to obtain a far jump
  - uses a **FAR PTR** directive, e.g., **JMP FAR PTR START**.
  - defines a **far label**, e.g., **EXTRN START:FAR**.
- A label is far only if it is external to the current code segment or procedure. External labels appear in programs that contain more than one program file.
- Two ways of defining a far label that can be accessed from outside the procedure block
  - uses directive **EXTRN**, e.g., **EXTRN START:FAR**
  - uses a double colon, e.g., **START::**

- The **JMP UP** instruction references a far label.
  - Label UP is defined as a far label by the EXTRN UP:FAR directive
- When the program files are joined, the linker inserts the address for the UP label into the JMP UP instruction. Also inserts segment address in JMP START instruction.

			<u>EXTRN UP:FAR</u>		far label
0000	33	DB		XOR BX,BX	
0002	B8	0001	START:	ADD AX,1	
0005	E9	0200	R	JMP NEXT	
				<skipped memory locations>	
0200	8B	D8	NEXT:	MOV BX,AX	
0202	EA	0002	— R	<u>JMP FAR PTR START</u>	
0207	EA	0000	— E	<u>JMP UP</u>	
					far jump

 segment address

# ***Jumps with Register Operands***

- Jump can also use a 16- or 32-bit register as an operand.
  - automatically sets up as an **absolute indirect jump**
  - address of the jump is in the register specified by the jump instruction
- Unlike displacement associated with the near jump, register contents are transferred directly into the instruction pointer.
- An absolute indirect jump does not add to the instruction pointer.

- For example, **JMP AX** copies the contents of the AX register into the IP.
  - allows a jump to any location within the current code segment
- In 80386 and above, JMP EAX also jumps to any location within the current code segment;
  - in protected mode the code segment can be 4G bytes long, so a 32-bit offset address is needed

# *Indirect Jumps Using an Index*

- Jump instruction may also use the [ ] form of addressing (e.g., **JMP TABLE [SI]**) to directly access the jump table.
- The jump table can contain offset addresses for **near indirect jumps**, or segment and offset addresses for **far indirect jumps**.
  - also known as a *double-indirect jump* if the register jump is called an *indirect jump*
- The assembler assumes that the jump is near unless the FAR PTR directive indicates a far jump instruction.

- Mechanism used to access the jump table is identical with a normal memory reference.
  - **JMP TABLE [SI]** instruction points to a jump address stored at the data segment offset location addressed by SI
- Both the register and indirect indexed jump instructions usually address a 16-bit offset.
  - both types of jumps are near jumps
- If **JMP FAR PTR [SI]** or **JMP TABLE [SI]**, with TABLE data defined with the DD directive:
  - microprocessor assumes the jump table contains doubleword, 32-bit addresses (IP and CS)



# Conditional Jumps and Conditional Sets

- In 8086 – 80286, conditional jump instructions are always short jumps.
  - limits range to within +127 and –128 bytes from the location following the conditional jump
- In 80386 and above, conditional jumps are either short or near jumps ( $\pm 32K$ ).
  - in 64-bit mode of the Pentium 4, the near jump distance is  $\pm 2G$  for the conditional jumps
- Allows a conditional jump to any location within the current code segment.

- Conditional jump instructions test flag bits:
  - sign (S), zero (Z), carry (C)
  - parity (P), overflow (O)
- If the condition under test is true, a branch to the label occurs.
- If false, next sequential step in program executes.
- For example, a **JC** will jump if the carry bit is set.
- Most conditional jump instructions are straightforward as they often test one flag bit.
  - although some test more than one

**TABLE 6–1** Conditional jump instructions.

	<i>Assembly Language</i>	<i>Tested Condition</i>	<i>Operation</i>
for unsigned numbers	JA	<u><math>Z = 0</math> and <math>C = 0</math></u>	Jump if above
	JAЕ	$C = 0$	Jump if above or equal
	JB	$C = 1$	Jump if below
	JBE	<u><math>Z = 1</math> or <math>C = 1</math></u>	Jump if below or equal
	JC	$C = 1$	Jump if carry
	JE or JZ	$Z = 1$	Jump if equal or jump if zero
for signed numbers	JG	<u><math>Z = 0</math> and <math>S = 0</math></u>	Jump if greater than
	JGE	$S = 0$	Jump if greater than or equal
	JL	$S \neq 0$	Jump if less than
	JLE	<u><math>Z = 1</math> or <math>S \neq 0</math></u>	Jump if less than or equal
	JNC	$C = 0$	Jump if no carry
	JNE or JNZ	$Z = 0$	Jump if not equal or jump if not zero
	JNO	$O = 0$	Jump if no overflow
	JNS	$S = 0$	Jump if no sign (positive)
	JNP or JPO	$P = 0$	Jump if no parity or jump if parity odd
	JO	$O = 1$	Jump if overflow
	JP or JPE	$P = 1$	Jump if parity or jump if parity even
	JS	$S = 1$	Jump if sign (negative)
	JCXZ	$CX = 0$	Jump if CX is zero
	JECXZ	$ECX = 0$	Jump if ECX equals zero
	JRCXZ	$RCX = 0$	Jump if RCX equals zero (64-bit mode)

- Because both signed and unsigned numbers are used in programming.
- Because the order of these numbers is different, there are two sets of conditional jump instructions for magnitude comparisons.
- 16- and 32-bit numbers follow the same order as 8-bit numbers, except that they are larger.
- Figure 6–5 shows the order of both signed and unsigned 8-bit numbers.

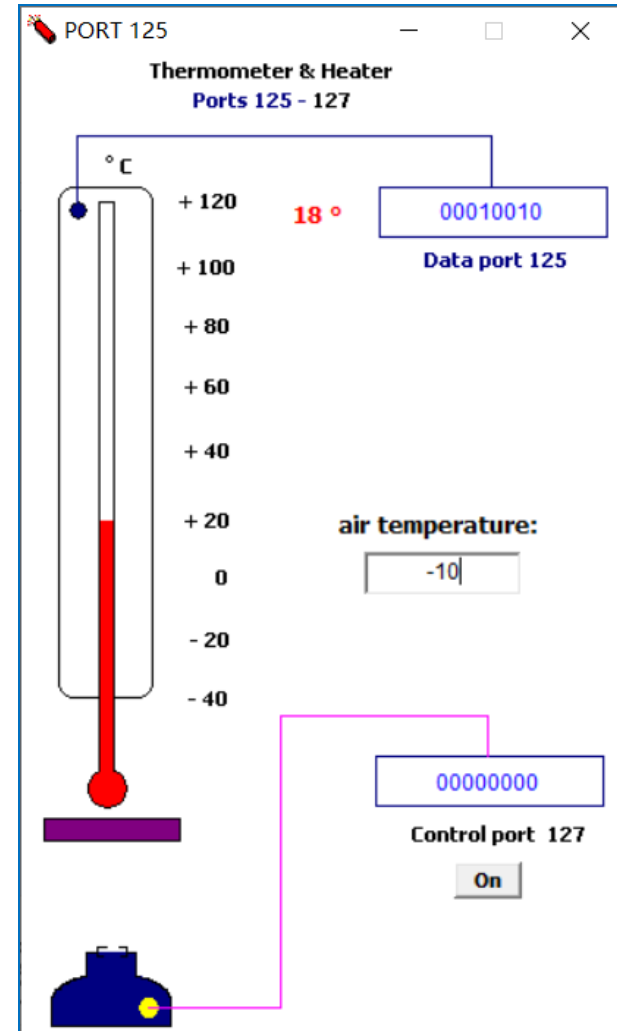
**Figure 6–5** Signed and unsigned numbers follow different orders.

Unsigned numbers		Signed numbers	
255	FFH	+127	7FH
254	FEH	+126	7EH
132	84H	+2	02H
131	83H	+1	01H
130	82H	+0	00H
129	81H	-1	FFH
128	80H	-2	FEH
4	04H	-124	84H
3	03H	-125	83H
2	02H	-126	82H
1	01H	-127	81H
0	00H	-128	80H

- When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions.
  - terms *greater than* and *less than* refer to signed numbers
- When unsigned numbers are compared, use the JA, JB, JAE, JBE, JE, and JNE instructions.
  - terms *above* and *below* refer to unsigned numbers

# An Example of Heater Control

- This short program shows how to keep constant temperature using heater and thermometer (between 15°C and 20°C).
- It is assumed that air temperature is lower 15°C.
- The air temperature can be read from I/O port 125.
- The heater can be turned on (“1”) and off (“0”) through I/O port 127.



- The following code is an example of a heater control.

.code

start:

in al, 125 ; read temperature

cmp al, 15

jb low ← jl low

cmp al, 20

ja high ← jg high

jmp start

low:

mov al, 1

out 127, al ; turn on heater

jmp start

high:

mov al, 0

out 127, al ; turn off heater

jmp start

hlt

.exit



- Remaining conditional jumps test individual flag bits, such as overflow and parity.
  - notice that JE has an alternative opcode JZ
- All instructions have alternates, but many aren't used in programming because they don't usually fit the condition under test.
- The conditional jump instructions all test flag bits except for **JCXZ** (jump if CX = 0) and **JECXZ** (jump if ECX = 0).

- Instead of testing flag bits, JCXZ directly tests the contents of the CX register without affecting the flag bits, and JECXZ tests the contents of the ECX register.
- For the JCXZ instruction
  - if  $CX = 0$ , a jump occurs
  - if  $CX \neq 0$ , no jump occurs.
- Likewise for the JECXZ instruction, if  $ECX = 0$ , a jump occurs; if  $ECX \neq 0$ , no jump occurs.

- The following program uses the SCASB instruction to search a table for 0AH. Following the search, a JCXZ instruction tests CX to see if the count has reached zero.

```
;Instructions that search a table of 100 bytes for 0AH
;The offset address of TABLE is assumed to be in SI
;
    MOV CX,100                ;load counter
    MOV AL,0AH                ;load AL with 0AH
    CLD                       ;auto-increment
    REPNE SCASB                ;search for 0AH
    JCXZ NOT_FOUND            ;if not found
NOT_FOUND
```

# ***The Conditional Set Instructions***

- 80386 - Core2 processors also contain conditional set instructions.
- Set Byte on Condition (**SETcc**) instructions check the status flags in the EFLAGS register
  - If the flags meet the condition specified in the mnemonic (cc), sets the value in the specified 8-bit memory location or register to 1.
  - If the flags do not meet the specified condition, SETcc clears the memory location or register to 0.

# ***The Conditional Set Instructions***

- For example, **SETC EAX**
  - if carry is set, EAX = 01H
  - if carry is cleared, EAX = 00H
- The contents of EAX can be tested at a later point in the program to determine if carry is cleared at the point where the SETC EAX instruction executed.
- The conditional set instructions are useful where a condition must be tested at a point much later in the program.

# The Conditional Set Instructions

**TABLE 6–2** Conditional set instructions.

<i>Assembly Language</i>	<i>Tested Condition</i>	<i>Operation</i>
SETA	$Z = 0$ and $C = 0$	Set if above
SETAE	$C = 0$	Set if above or equal
SETB	$C = 1$	Set if below
SETBE	$Z = 1$ or $C = 1$	Set if below or equal
SETC	$C = 1$	Set if carry
SETE or SETZ	$Z = 1$	Set if equal or set if zero
SETG	$Z = 0$ and $S = 0$	Set if greater than
SETGE	$S = 0$	Set if greater than or equal
SETL	$S \neq 0$	Set if less than
SETLE	$Z = 1$ or $S \neq 0$	Set if less than or equal
SETNC	$C = 0$	Set if no carry
SETNE or SETNZ	$Z = 0$	Set if not equal or set if not zero
SETNO	$O = 0$	Set if no overflow
SETNS	$S = 0$	Set if no sign (positive)
SETNP or SETPO	$P = 0$	Set if no parity or set if parity odd
SETO	$O = 1$	Set if overflow
SETP or SETPE	$P = 1$	Set if parity or set if parity even
SETS	$S = 1$	Set if sign (negative)

# ***The Conditional Set Instructions***

- Software typically uses the SETcc instructions to set logical indicators.
- Like the CMOVcc instructions, the SETcc instructions can replace two instructions—a conditional jump and a move. For example:

```
.....  
ret = square(scale);  
if (ret > 0)  
    return 0;  
else  
    return 1;  
.....
```

using Jcc  
→  
gcc -O0

using SETcc  
→  
gcc -O1

```
call    square(int)  
cmp     eax, 0  
jle     .L1  
mov     al, 0  
.....  
.L1:    mov     al, 1  
.....
```

```
call    square(int)  
test    eax, eax  
setle   al
```

# LOOP

- the **LOOP** instruction performs a loop operation using the RCX, ECX or CX register as a counter.
- Each time the LOOP instruction is executed, the count register is decremented, then checked for 0.
  - if CX  $\neq$  0, a near jump is performed to the address indicated by the label
  - if CX becomes 0, the next sequential instruction executes



- Typically, the LOOP contains a compare instruction to set or clear the ZF flag.
- In 16-bit instruction mode, LOOP uses CX; in the 32-bit mode, LOOP uses ECX.
- In 64-bit mode, the loop counter is in RCX and is 64 bits wide.
- The size of the count register used (CX, ECX, or RCX) depends on the address-size attribute of the LOOP instruction.

- The following program sums the contents of BLOCK1 and BLOCK2 and stores the results on top of the data in BLOCK2.

```
.MODEL SMALL                ;select SMALL model
.DATA                      ;start data segment
BLOCK1 DW 100 DUP(?)       ;100 words for BLOCK1

BLOCK2 DW 100 DUP(?)       ;100 words for BLOCK2

.CODE                      ;start code segment
.STARTUP                  ;start program
    MOV AX,DS              ;overlap DS and ES
    MOV ES,AX
    CLD                   ;select auto-increment
    MOV CX,100             ;load counter
    MOV SI,OFFSET BLOCK1   ;address BLOCK1
    MOV DI,OFFSET BLOCK2   ;address BLOCK2
L1:    LODSW               ;load AX with BLOCK1
    ADD AX,ES:[DI]         ;add BLOCK2
    STOSW                 ;save answer
    LOOP L1               ;repeat 100 times

.EXIT
END
```

# ***Conditional LOOPS***

- LOOP instruction also has conditional forms: **LOOPE** and **LOOPNE**
- Syntax:
  - LOOPE     destination
  - LOOPNE   destination
- LOOPE (**loop while equal**) instruction accepts the ZF flag as a condition for terminating the loop before the count reaches zero.
  - will exit loop if the condition is not equal or the CX register decrements to 0

- **LOOPNE (loop while not equal)** jumps if  $CX \neq 0$  while a not-equal condition exists.
  - will exit loop if the condition is equal or the CX register decrements to 0
- **LOOPE** is useful when scanning an array for the first element that **does not match** a given value.
- **LOOPNE** is useful when scanning an array for the first element that **matches** a given value.
- The **LOOPE** and **LOOPNE** instruction do not affect the state of the ZF flag.

- In 80386 - Core2 processors, conditional LOOP can use CX or ECX as the counter.
  - LOOPEW/LOOPED or LOOPNEW/LOOPNED override the instruction mode if needed
- Under 64-bit operation, the loop counter uses RCX and is 64 bits in width
- Alternates exist for LOOPE and LOOPNE.
  - LOOPE same as LOOPZ
  - LOOPNE instruction is the same as LOOPNZ
- In most programs, only the LOOPE and LOOPNE apply.

- The following code finds the first positive value in an array:

```
.data
```

```
array DW -3,-6,-1,-10,10,30,40,4
```

```
.code
```

```
    mov esi,OFFSET array
```

```
    mov ecx,LENGTHOF array
```

```
next:
```

```
    test WORD PTR [esi],8000h
```

```
; test sign bit
```

```
    pushf
```

```
; push flags on stack
```

```
    add esi, 2
```

```
; inc ESI to next value
```

```
    popf
```

```
; pop flags from stack
```

```
    loopnz next
```

```
; continue loop
```

```
    jnz quit
```

```
; ZF=0, none found
```

```
    sub esi,2
```

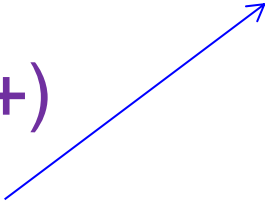
```
; ESI points to value
```

```
quit:
```

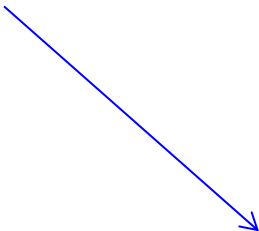
# For, Do-while, While Loops in Assembly Language

- For-loop in C:

```
for(int x = 0; x<=3; x++)  
{  
    //Do something!  
}
```



```
xor cx,cx  
begin: .....  
      inc cx  
      cmp cx,3  
      jle begin
```



```
mov cx,4  
start: .....  
      loop start
```

# For, Do-while, While Loops in Assembly Language

- Do-while-loop in C:

```
int x=1;
```

```
do{  
    //Do something!  
} while(x==1)
```



```
mov ax,1  
start: .....  
      cmp ax,1  
      je start
```



# For, Do-while, While Loops in Assembly Language

- While-loop in C:

```
while(x==1)
```

```
{
```

```
    //Do something
```

```
}
```



```
        jmp sloop
```

```
cloop:  .....
```

```
sloop:  cmp ax,1
```

```
        je cloop
```

# 6–2 CONTROLLING THE FLOW OF THE PROGRAM

- It is much easier to use assembly language directives `.IF`, `.ELSE`, `.ELSEIF`, and `.ENDIF` to control the flow of the program than to use the correct conditional jump statement.
  - these statements always indicate a special assembly language command to MASM
- Control flow assembly language statements beginning with a period available to MASM version 6.xx, and not to earlier versions.

- Other directives developed include .REPEAT–.UNTIL and .WHILE–.ENDW.
  - **the dot commands** do not function using the Visual C++ inline assembler
- Never use uppercase for assembly language commands with the inline assembler.
  - some of them are reserved by C++ and will cause problems

# Directives for Conditional Control Flow

- .WHILE
- .ENDW
- .REPEAT
- .UNTIL
- .UNTILCXZ
- .BREAK
- .CONTINUE
- .IF
- .ENDIF
- .ELSE
- .ELSEIF

# Operators for Conditional Control Flow

- ! (logical not)
- != (not equal)
- || (logical or)
- && (logical and)
- < (less than)
- <= (less or equal)
- == (equal)
- > (greater than)
- >= (greater or equal)
- & (bitwise and)
- CARRY? (carry test)
- PARITY? (parity test)
- SIGN? (sign test)
- ZERO? (zero test)
- OVERFLOW? (overflow test)

# .IF, .ELSE, .ELSEIF, and .ENDIF

- The example shows how .IF and .ENDIF statements are used to control the flow of a program by testing AL for the ASCII letters A through F. If the contents of AL are A through F, 7 is subtracted from AL.

- input: ASCII code '0'-'9', 'A'-'F'
- output: hexadecimal number 0h-fh

```
.IF AL >= 'A' && AL <= 'F'  
    SUB AL, 7  
.ENDIF  
SUB AL, 30H
```



```
char temp;  
_asm{  
    mov     al, temp  
    cmp     al, 41h  
    jnb     Later  
    cmp     al, 46h  
    jnb     Later  
    sub     al, 7  
Later:  
    sub     al, 30h  
    mov     temp, al  
}
```

- See Table 6–3 for a complete list of relational operators (such as &&) used with the .IF statement.

**TABLE 6–3** Relational operators used with the .IF statement in assembly language.

<i>Operator</i>	<i>Function</i>
==	Equal or the same as
!=	Not equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
&	Bit test
!	Logical inversion
&&	Logical AND
	Logical OR
	Or

# WHILE Loops

- The **.WHILE** statement is used with a condition to begin the loop.
  - the **.ENDW** statement ends the loop
- The **.BREAK** and **.CONTINUE** statements are available for use with the while loop.
  - **.BREAK** is often followed by **.IF** to select the break condition as in **.BREAK .IF AL == 0DH**
  - **.CONTINUE** can be used to allow a **DO–.WHILE** loop to continue if a certain condition is met
- The **.BREAK** and **.CONTINUE** commands function the same manner in C++.



- The following example shows how the .WHILE statement is used to read data from the keyboard and store it into an array (BUF) until the enter key (0DH).

```

.CODE                                ;start code segment
.STARTUP                             ;start program
    MOV     AX,DX                     ;overlap DS with ES
    MOV     ES,AX
    CLD                               ;select auto-increment
    MOV     DI,OFFSET BUF             ;address buffer

    .WHILE AL != 0DH                  ;loop while not enter
*      jmp   @C0001
* @C0002:
    MOV     AH,1                      ;read key
    INT     21H
    STOSB                               ;store key code
    .ENDW
* @C0001:
*      cmp   al,0dh
*      jne   @C0002
    MOV     BYTE PTR[DI-1], '$' ← '$' is the terminated string for DOS
    MOV     DX,OFFSET BUF              INT 21H function 9
    MOV     AH,9
    INT     21H                        ;display BUF
.EXIT
END

```

# REPEAT-UNTIL Loops

- A series of instructions is repeated until some condition occurs.
- The **.REPEAT** statement defines the start of the loop.
- The end is defined with the **.UNTIL** statement, which contains a condition.
- An **.UNTILCXZ** instruction uses the LOOP instruction to check CX for a repeat loop.
  - **.UNTILCXZ** uses the CX register as a counter to repeat a loop a fixed number of times

- The following example shows how the REPEAT-UNTIL is used to read data from the keyboard and store it into an array (BUF) until the enter key (0DH).

```

.CODE                                ;start code segment
.STARTUP                             ;start program
    MOV     AX,DX                    ;overlap DS with ES
    MOV     ES,AX
    CLD                               ;select auto-increment
    MOV     DI,OFFSET BUF           ;address buffer

    .REPEAT                           ;repeat until enter

*   @C0001:
    MOV     AH,1                     ;read key
    INT     21H
    STOSB                             ;store key code

    .UNTIL AL == 0DH

*   cmp     al,0dh
*   jne     @C0001
    MOV     BYTE PTR[DI-1], '$'
    MOV     DX,OFFSET BUF
    MOV     AH,9
    INT     21H                      ;display BUF

.EXIT
END

```

- The following code uses the .UNTILCXZ to add the contents of byte-sized array ONE to byte-sized array TWO. The sums are stored in array THREE.

```
MOV    CX,100                      ;set count
MOV    DI,OFFSET THREE            ;address arrays
MOV    SI,OFFSET ONE
MOV    BX,OFFSET TWO
```

```
.REPEAT
```

```
@C0001:
```

```
LODSB
ADD    AL,[BX]
STOSB
INC    BX
```

```
.UNTILCXZ
```

```
LOOP  @C0001
```

## 6–3 PROCEDURES

- A procedure is a group of instructions that usually performs one task.
  - subroutine, method, or **function** is an important part of any system's architecture
- A procedure is a reusable section of the software stored in memory once, used as often as necessary.
  - saves memory space and makes it easier to develop software

- Disadvantage of procedure is time it takes the computer to link to, and return from it.
  - CALL links to the procedure; the RET (**return**) instruction returns from the procedure
- CALL pushes the address of the instruction following the CALL (**return address**) on the stack.
  - the stack stores the return address when a procedure is called during a program
- RET instruction removes an address from the stack so the program returns to the instruction following the CALL.

- A procedure begins with the PROC directive and ends with the ENDP directive.
  - each directive appears with the procedure name
- PROC is followed by the type of procedure:
  - NEAR or FAR
- In MASM version 6.x, the NEAR or FAR type can be followed by the **USES** statement.
  - USES allows any number of registers to be automatically pushed to the stack and popped from the stack within the procedure

0000		
0000	03	C3
0002	03	C1
0004	03	C2
0006		C3
0007		

near return  
opcode

0007		
0007	03	C3
0009	03	C1
000B	03	C2
000D		CB
000E		

far return  
opcode

000E		
0011	03	C3
0013	03	C1
0015	03	C2
001B		

SUMS	PROC	NEAR
	ADD	AX, BX
	ADD	AX, CX
	ADD	AX, DX
	RET	
SUMS	ENDP	

SUMS1	PROC	FAR
	ADD	AX, BX
	ADD	AX, CX
	ADD	AX, DX
	RET	
SUMS1	ENDP	

SUMS3	PROC	NEAR	<u>USE</u>	<u>BX</u>	<u>CX</u>	<u>DX</u>
	ADD	AX, BX				
	ADD	AX, CX				
	ADD	AX, DX				
	RET					
SUMS	ENDP					

USES  
statement



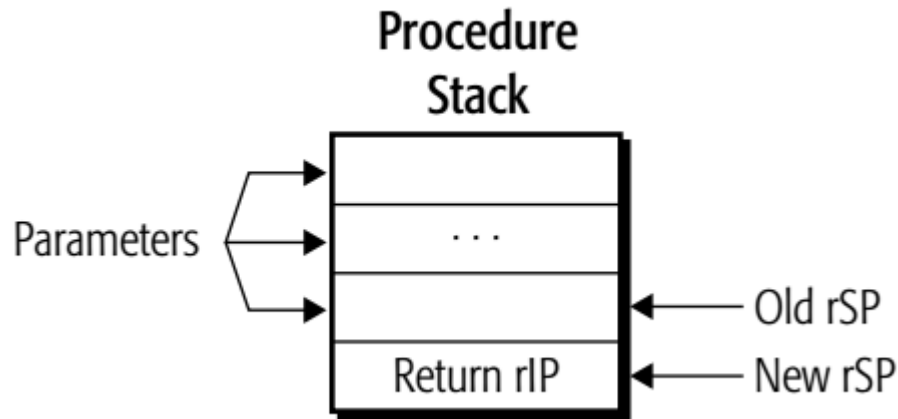
- Procedures that are to be used by all software (global) should be written as far procedures.
- Procedures that are used by a given task (local) are normally defined as near procedures.
- Most procedures are near procedures.

# CALL

- Transfers the flow of the program to the procedure.
- CALL instruction differs from the jump instruction because a CALL saves a return address on the stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.

# Near CALL

- When executing a near call
  - push **parameters** onto the stack prior to executing the CALL instruction.
  - push the value of the **EIP** register on the stack.
- Then branches to the address in the current code segment specified by the target operand.
- The CS register is not changed on near calls.



# *Near CALL*

- The target operand specifies two types
  - a **relative offset** (a signed displacement relative to the current value of the instruction pointer pointing to the instruction following the CALL instruction).
  - an **absolute offset** in the code segment (an offset from the base of the code segment)
- For a “**near absolute indirect**” call, an absolute offset is specified indirectly in a register or a memory location. For example
  - MOV AX, 64h
  - CALL AX ; jump to CS:64h

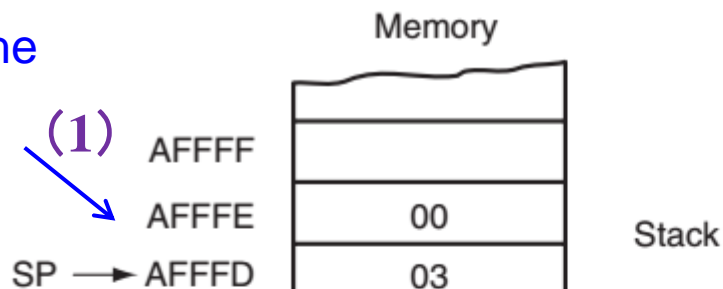
# ***Near CALL***

- For a “near relative direct” call, it is 3 bytes long.
  - the first byte contains the opcode; the second and third bytes contain the displacement
- When the near relative CALL executes
  - It first pushes the offset address of the next instruction onto the stack. Offset address of the next instruction appears in the instruction pointer (IP or EIP)
  - It then adds displacement from bytes 2 & 3 to the IP to transfer control to the procedure.

- Why save the IP or EIP on the stack?
  - the instruction pointer always points to the next instruction in the program
- For the CALL instruction, the contents of IP/EIP are pushed onto the stack.
  - program control passes to the instruction following the CALL after a procedure ends
- Figure 6–6 shows the return address (IP) stored on the stack and the call to the procedure.

**Figure 6–6** The effect of a near relative CALL on the stack and the instruction pointer.

push the offset address of the  
next instruction onto the  
stack: 0003



procedure entry:  
 $10003H + 0FFFH = 11002H$

(2)

11003  
11002  
11001  
11000

(Procedure)

SP before CALL = FFFF  
SS before CALL = A000  
IP before CALL = 0003

10004  
10003  
10002  
10001  
10000

0F

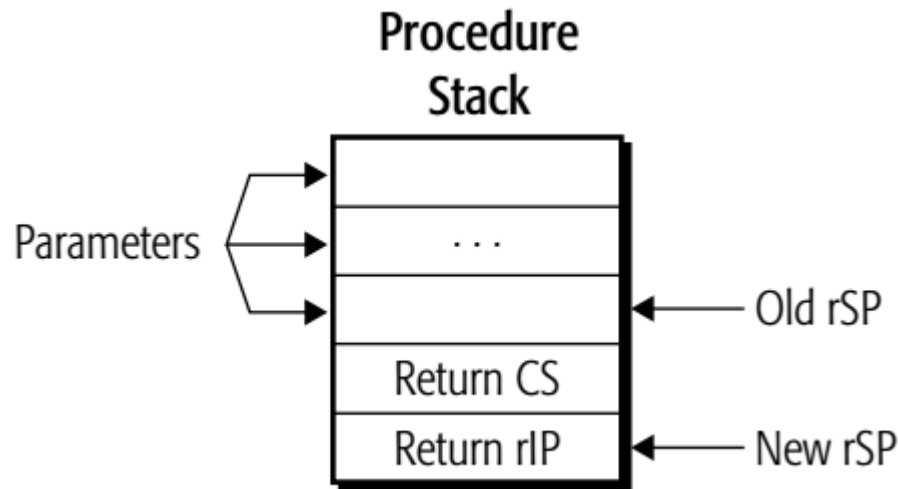
FF

CALL

} Near CALL

# Far CALL

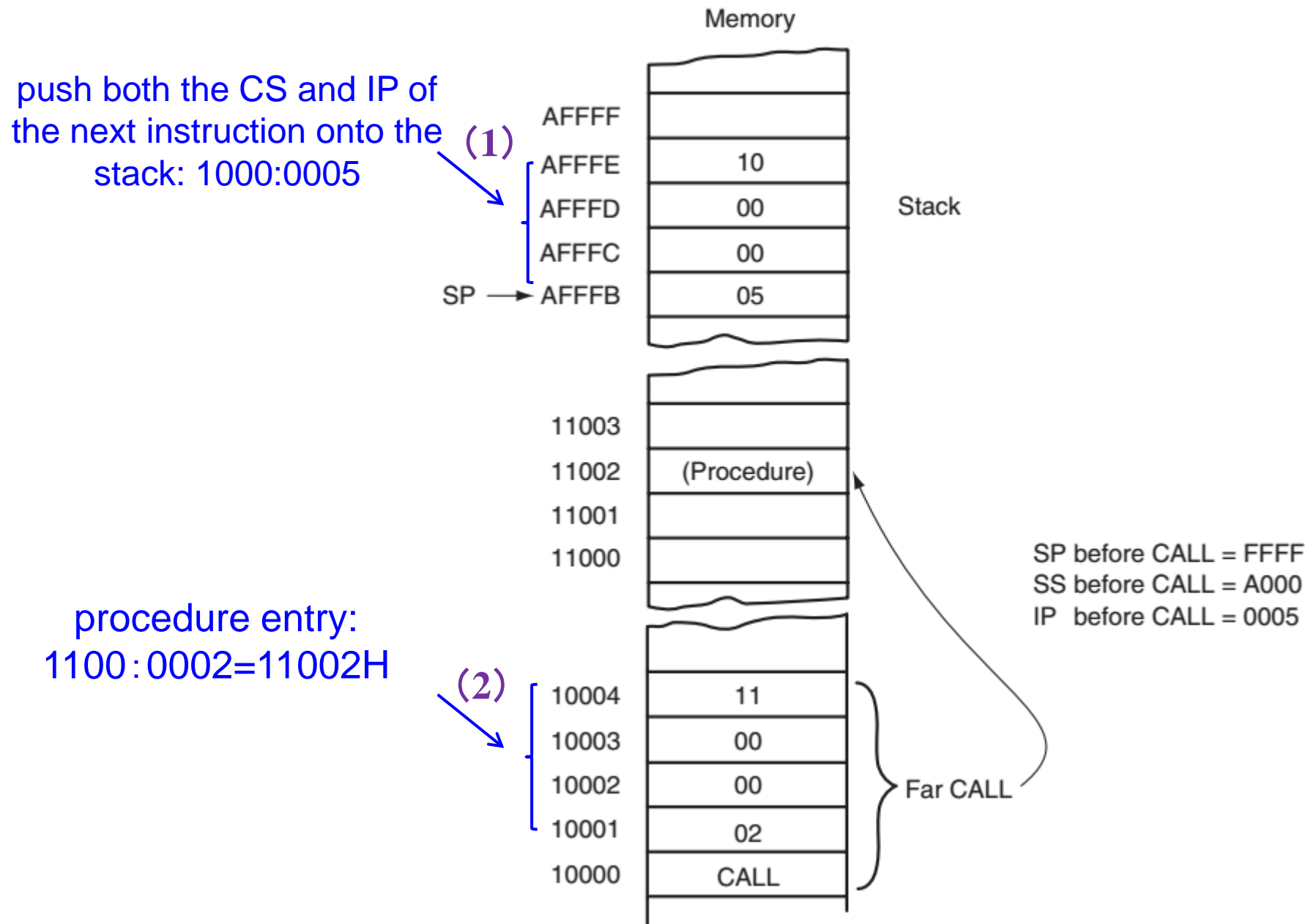
- 5-byte instruction contains an opcode followed by the next value for the IP and CS registers.
  - bytes 2 and 3 contain new contents of the IP
  - bytes 4 and 5 contain the new contents for CS
- Far CALL places **parameters** and the contents of both **IP and CS** on the stack before jumping to the address indicated by bytes 2 through 5.





- This allows far CALL to call a procedure located anywhere in the memory and return from that procedure.
- In 64-bit mode a far call is to any memory location and information placed onto the stack is an 8-byte number.
  - the far return instruction retrieves an 8-byte return address from the stack and places it into RIP
- Figure 6–7 shows how far CALL calls a far procedure.
  - contents of CS and IP are pushed onto the stack

**Figure 6-7** The effect of a far CALL instruction.



# ***CALLs with Register Operands***

- The example illustrates the use of the CALL register instruction to call a procedure that begins at offset address DISP in the current code segment to display 'OK'.

```
        MOV    BX,OFFSET DISP    ;load BX with offset DISP
        MOV    DL,'O'            ;display O
        CALL   BX
        MOV    DL,'K'            ;display K
        CALL   BX
.EXIT
;
;Procedure that displays the ASCII character in DL
;
DISP    PROC    NEAR
        MOV    AH,2              ;select function 2
        INT    21H              ;execute DOS function 2
        RET
DISP    ENDP
```

# ***CALLs with Indirect Memory Addresses***

- Particularly useful when different subroutines need to be chosen in a program.
- Essentially the same as the indirect jump that used a lookup table for a jump address.

```
        ;Instruction that calls procedure ZERO, ONE, or TWO
        ;depending on the value in EBX
        ;
TABLE    DW      ZERO          ;address of procedure ZERO
        DW      ONE           ;address of procedure ONE
        DW      TWO           ;address of procedure TWO

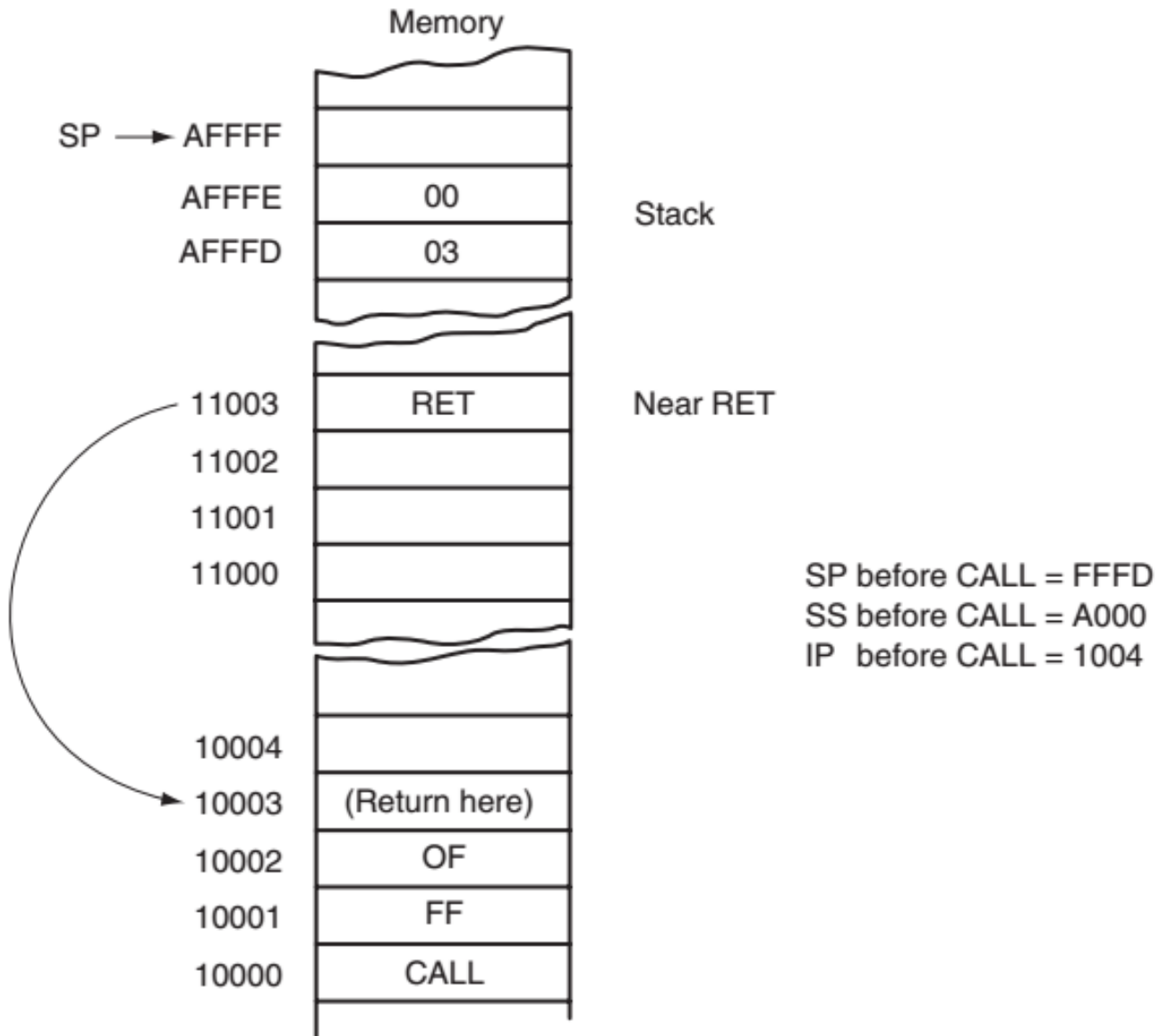
CALL    TABLE[2*EBX]
```

- The CALL instruction also can reference far pointers if the instruction appears as CALL FAR PTR [4\*EBX] or as CALL TABLE [4\*EBX], if the data in the table are defined as doubleword.

# RET

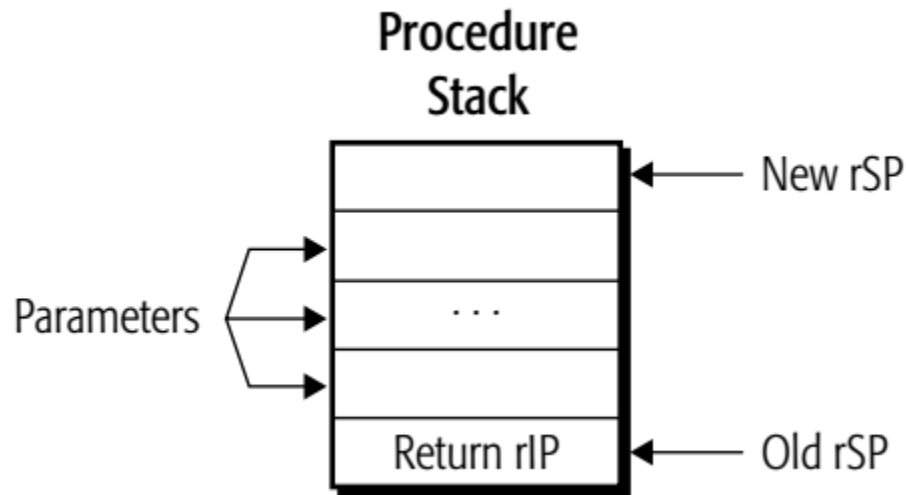
- Removes a 16-bit number (**near return**) from the stack placing it in IP, or removes a 32-bit number (**far return**) and places it in IP & CS.
  - near and far return instructions in procedure's PROC directive
  - automatically selects the proper return instruction
- Figure 6–8 shows how the CALL instruction links to a procedure and how RET returns in the 8086–Core2 operating in the real mode.

**Figure 6–8** The effect of a near return instruction on the stack and instruction pointer.



- With the 80386 through the Pentium 4 processors operating in the protected mode, the far return removes 6 bytes from the stack.
  - the first 4 bytes contain the new value for EIP and
  - the last 2 contain the new value for CS
- In the 80386 and above, a protected mode near return removes 4 bytes from the stack and places them into EIP.

- Another form of return (**RET n**) adds a number to the contents of the stack pointer (SP) after the return address is removed from the stack.
- If an immediate operand is included with the RET instruction, the stack pointer is adjusted by the number of bytes indicated.





- Parameters are addressed on the stack by using the BP register, which addresses the stack segment.
- The example shows how this type of return erases the data placed on the stack by a few pushes. The RET 4 adds a 4 to SP after removing the return address from the stack.

```

MOV    AX,30
MOV    BX,40
PUSH   AX                ;stack parameter 1
PUSH   BX                ;stack parameter 2
CALL   ADDM              ;add stack parameters

```

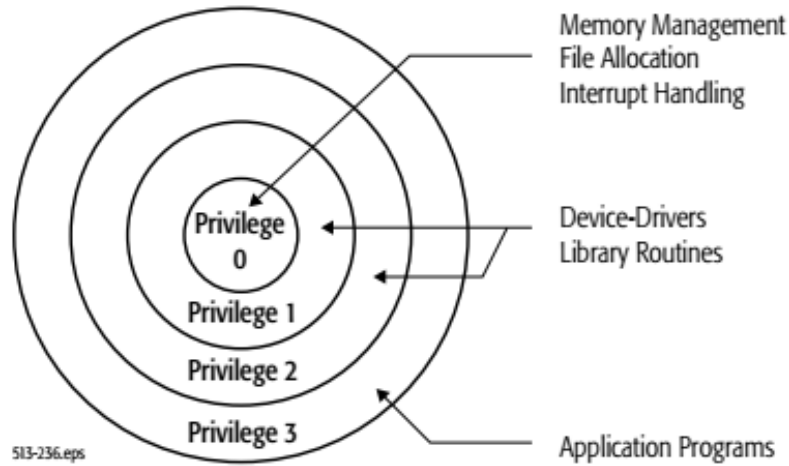
```

ADDM   PROC    NEAR
        PUSH   BP                ;save BP
        MOV    BP,SP              ;address stack with BP
        MOV    AX,[BP+4]          ;get parameter 2
        ADD    AX,[BP+6]          ;add parameter 1
        POP    BP                ;restore BP
        RET    4                  ;return, dump parameters
ADDM   ENDP

```

# Transfer Control Between Privilege Levels

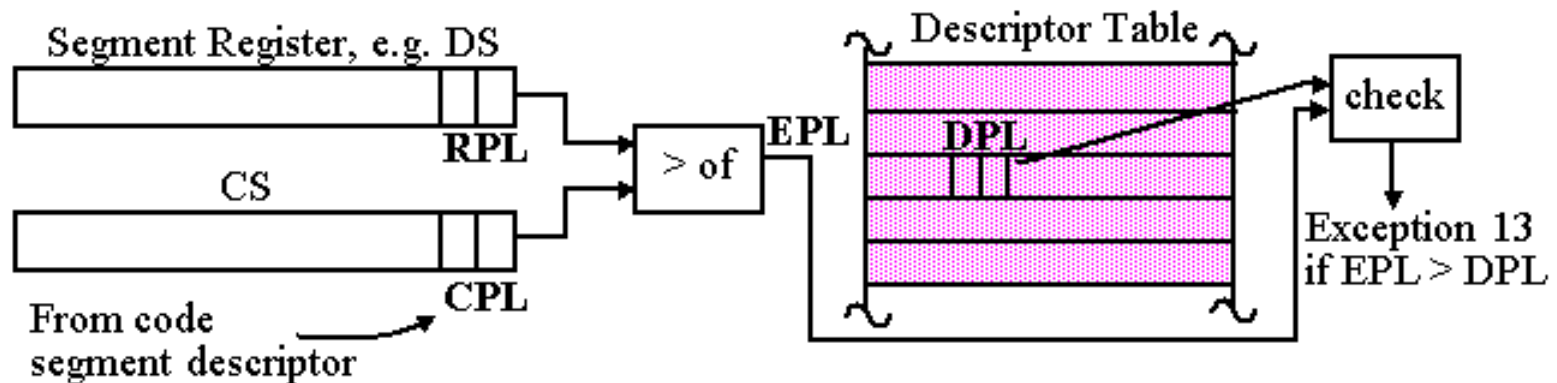
- Branches can also be used to transfer control to other code running at a different privilege level.
- In such cases, the processor automatically checks the source program and target program privileges to ensure that the transfer is allowed before performing control transfer operations.



**Figure 3-9. Privilege-Level Relationships**

# Privilege Levels

- Three different privilege levels for privilege check:
  - The **CPL** (current privilege level).
  - The **RPL** (requestor's privilege level) of the selector used to specify the target segment.
  - The **DPL** of the descriptor of the target segment.
- Instructions may use the target segment only if the DPL of the target segment is greater than or equal to the maximum of the CPL and the selector's RPL.



# Requested privilege level (RPL)

- The RPL is an override privilege level that is assigned to segment selectors. It is stored in bits 0 and 1 of the segment selector.
- The processor checks the RPL along with the CPL to determine if access to a segment is allowed. Even if the program requesting access to a segment has sufficient privilege to access the segment, access is denied if the RPL is not of sufficient privilege level.
- The RPL can be used to insure that privileged code does not access a segment on behalf of an application program unless the program itself has access privileges for that segment.

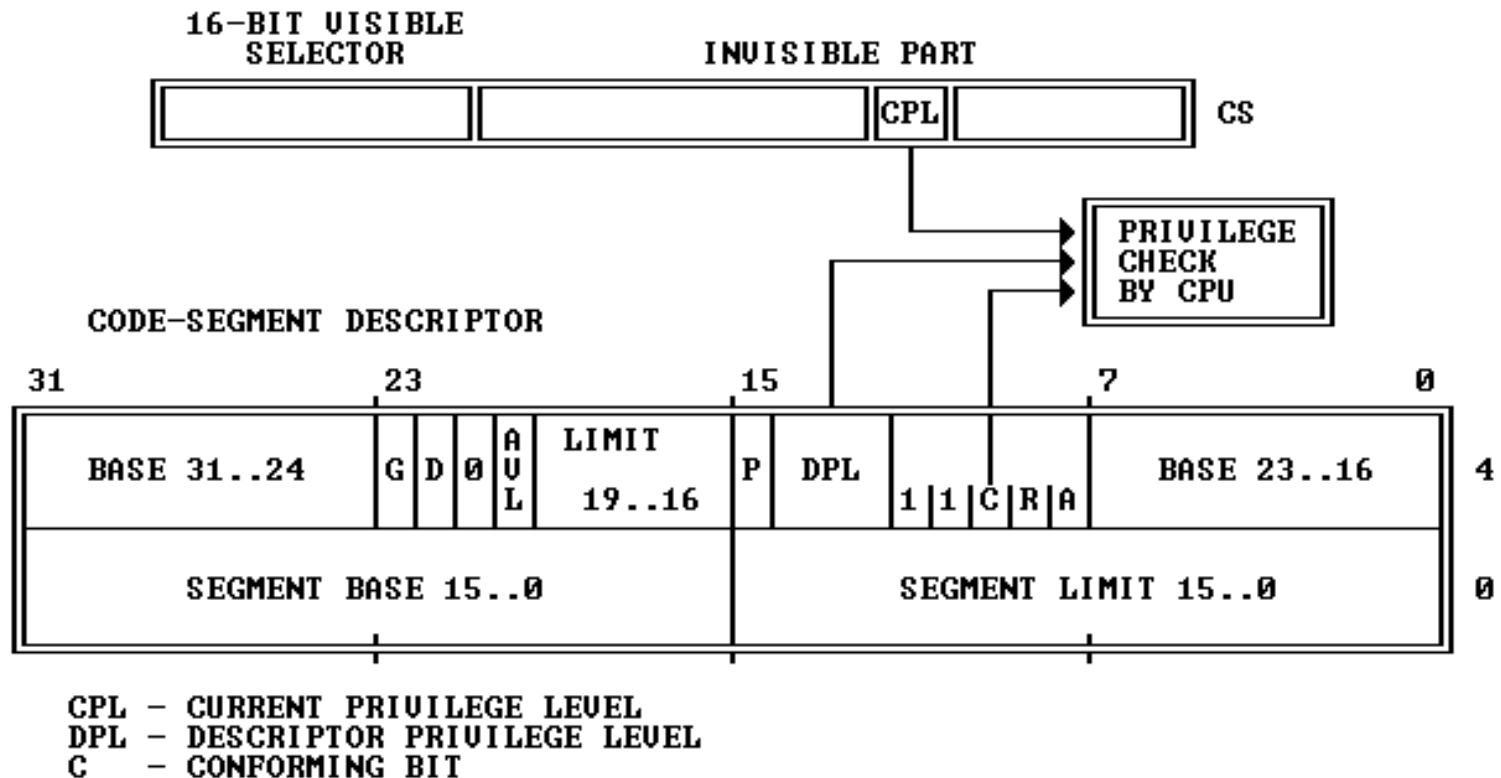
# Control Transfer Between Privilege Levels

- There are four ways to make an inter-privilege-level far call:
  - defines **Conforming Code segments**
  - through special segment descriptors called **Gates**
  - utilizes fast **system call instructions** (SYSCALL/SYSRET or SYSENTER/SYSEXIT) to access privilege level 0

# Conforming-Segment Mechanism

- The processor permits a JMP or CALL directly to another segment only if one of the following privilege rules is satisfied:
  - DPL of the target is equal to CPL.
  - The conforming bit of the target code-segment descriptor is set, and the DPL of the target is less than or equal to CPL.
- An executable segment whose descriptor has the conforming bit set is called a conforming segment.

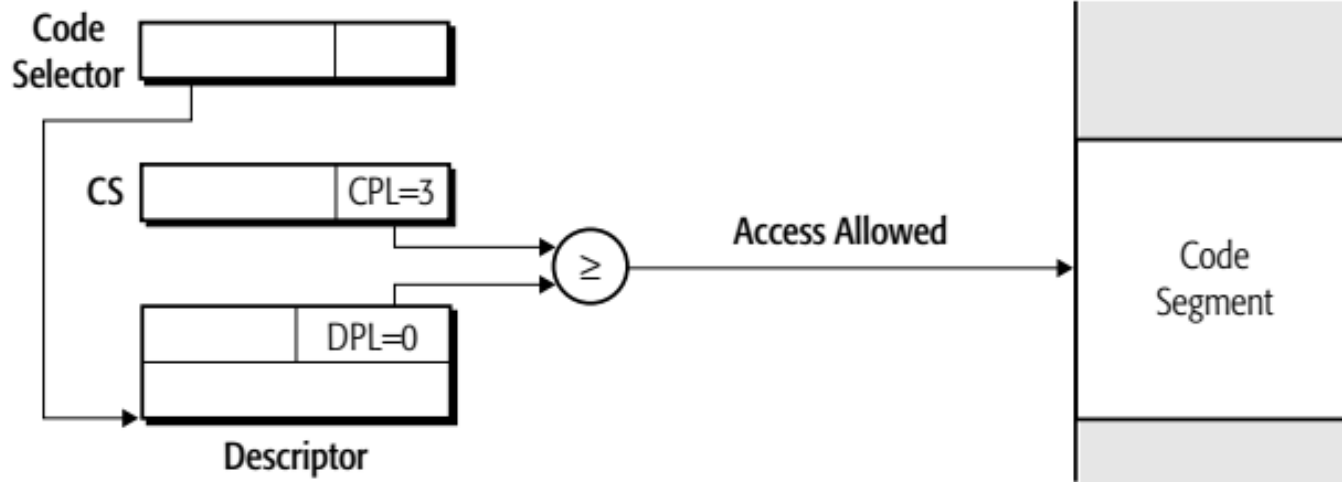
- The segment selector RPL for the destination code segment is not checked if the segment is a conforming code segment.
- The conforming-segment mechanism permits sharing of procedures that may be called from various privilege levels.



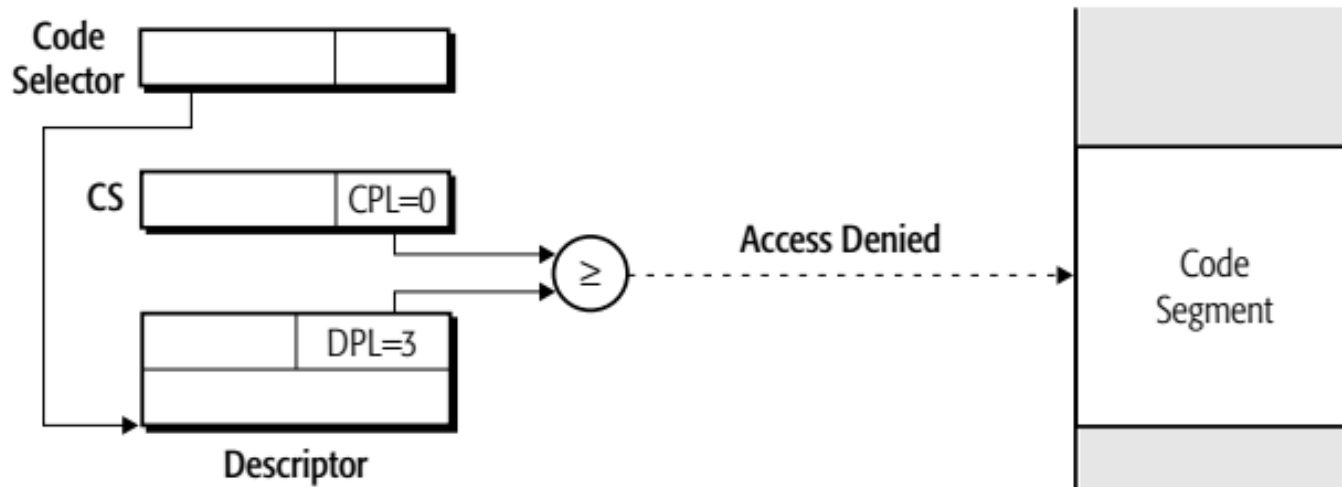
- When control is transferred to a conforming segment, the **CPL does not change**. This is the only case when CPL may be unequal to the DPL of the current executable segment.
- This mechanism is well suited to handle programs that share code but run at different privilege levels, e.g., shared math libraries.
- **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.



# Conforming Code-Segment Privilege-Check Examples



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

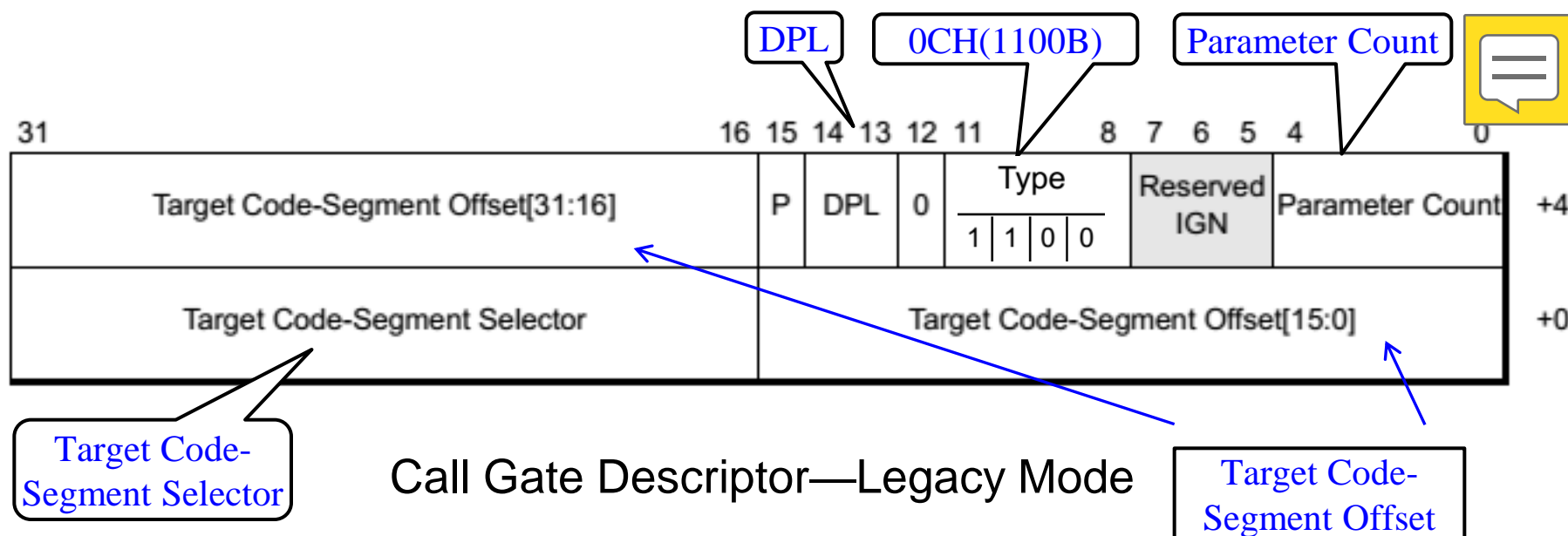
# Gate Descriptors

- To provide protection for control transfers among executable segments at different privilege levels, the processor uses **gate descriptors**.
- Gate descriptors do not describe segments, but instead **hold pointers to code-segment descriptors**.
- Gate descriptors are used for protected-mode control transfers between less-privileged and more-privileged software.

- There are four kinds of gate descriptors:
  - Call gates
  - Trap gates
  - Interrupt gates
  - Task gates
- Call gates act as an interface layer between code segments at different privilege levels.
- Trap and interrupt gates are special kinds of call gates used for calling exception and interrupt handlers.
- Task gates are used for task switching.

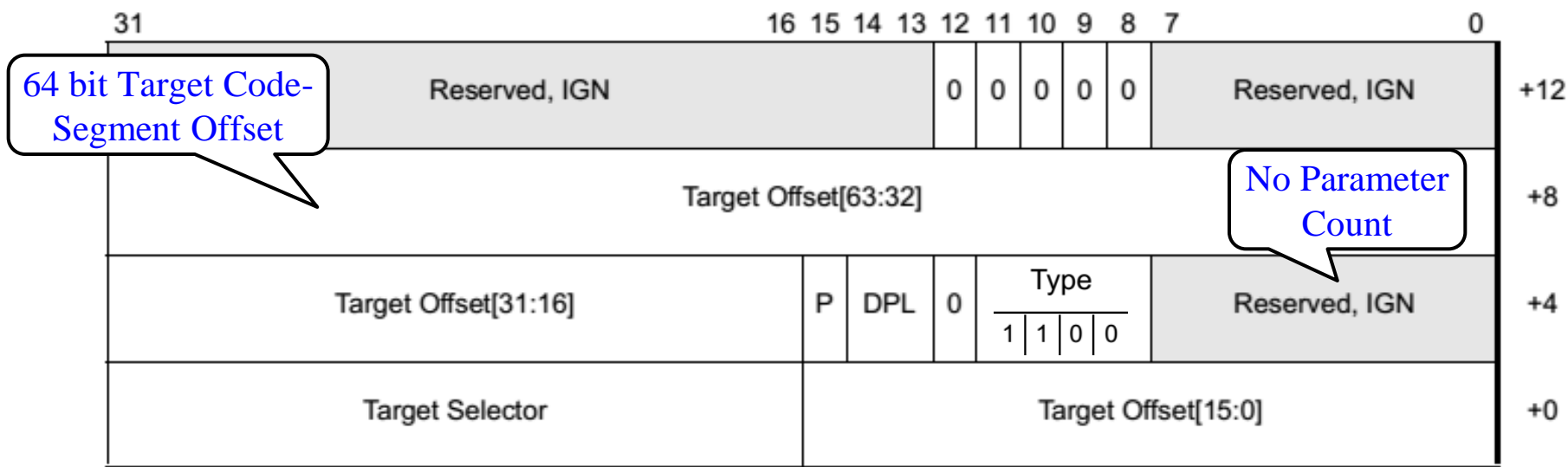
# Legacy Mode Call Gate Descriptor

- Call Gates are located in the GDT or LDT and are used to control access between code segments in the same or different tasks.
- System software uses call gates to establish protected entry points into system service routines.

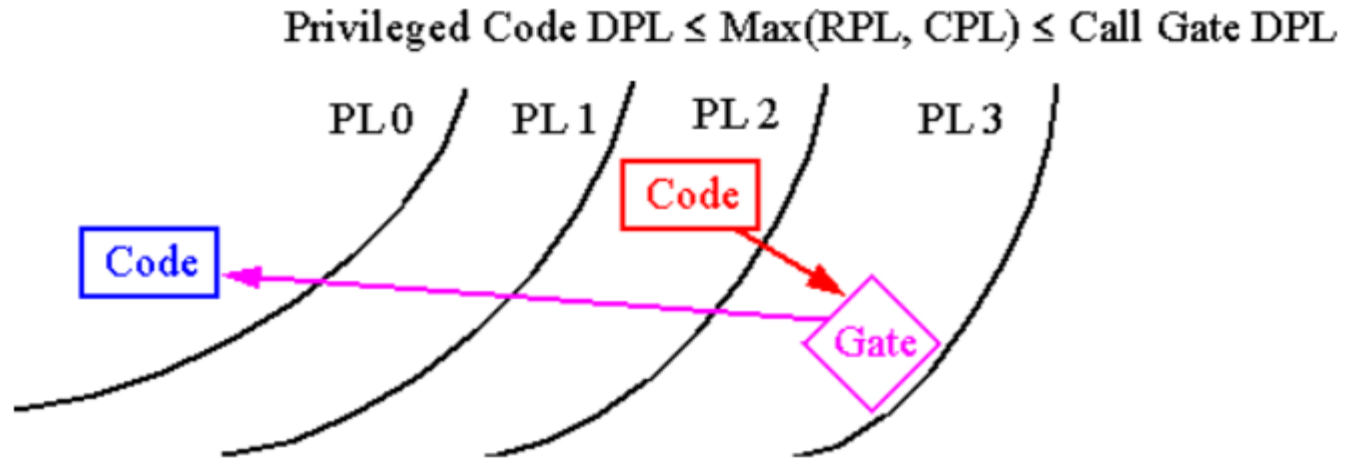


# Long Mode Call Gate Descriptor

- The legacy 32-bit mode call gate type (0CH) is redefined in long mode as a 64-bit call gate type.
- In long mode, gate descriptors are expanded by 64 bits, allowing them to hold 64-bit offsets.
- The parameter-count field has been removed.



Call-Gate Descriptor—Long Mode



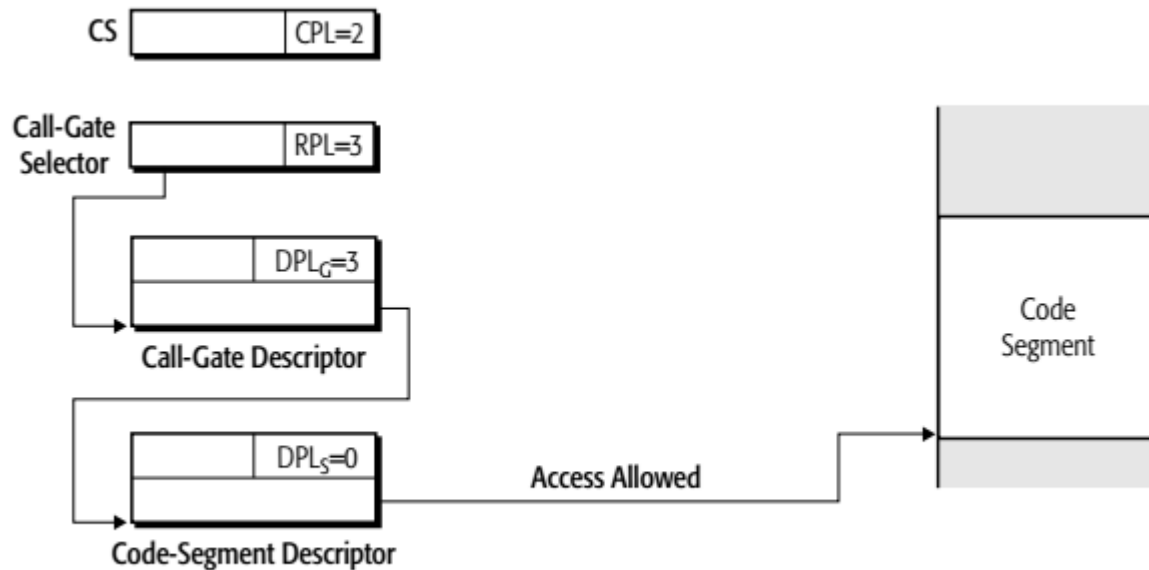
- Key points in a call gate control transfer
  - How to check the validity of a control transfer via a call gate?
  - How to find an entry point for a caller trying to access the more privileged code?
  - How to make a stack switching (from the calling procedures stack to the new stack)?

# Privilege Checking Rules

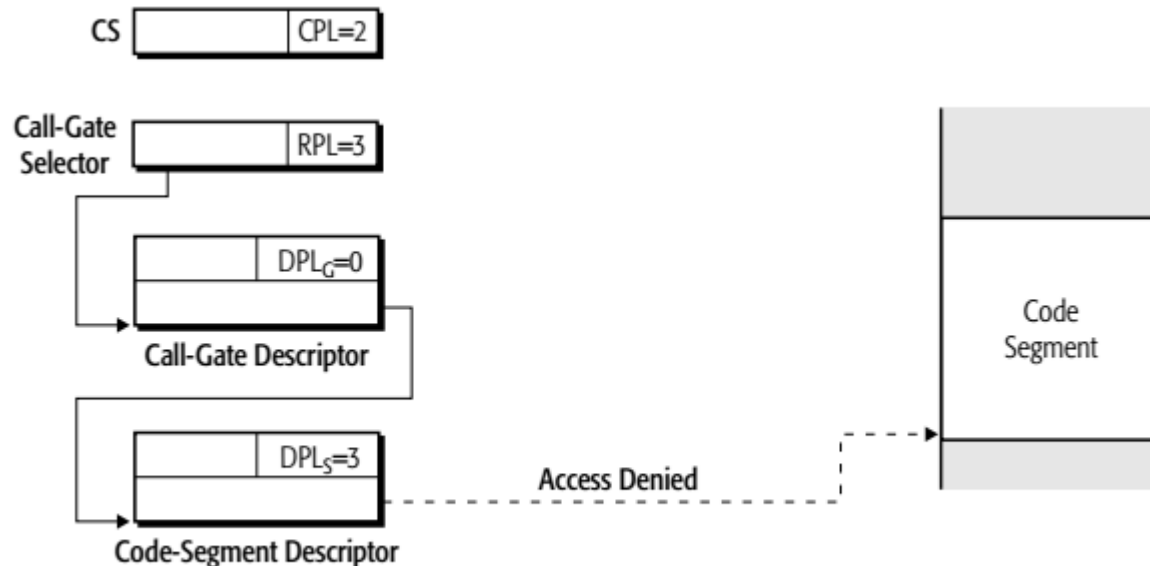
- The privilege checking rules are different depending on whether the control transfer was initiated with a CALL or a JMP instruction.

Instruction	Privilege Check Rules
CALL	$CPL \leq \text{call gate DPL}; RPL \leq \text{call gate DPL}$ Destination conforming code segment $DPL \leq CPL$ Destination nonconforming code segment $DPL \leq CPL$
JMP	$CPL \leq \text{call gate DPL}; RPL \leq \text{call gate DPL}$ Destination conforming code segment $DPL \leq CPL$ Destination nonconforming code segment $DPL = CPL$

# Privilege-Check Examples for Call Gates



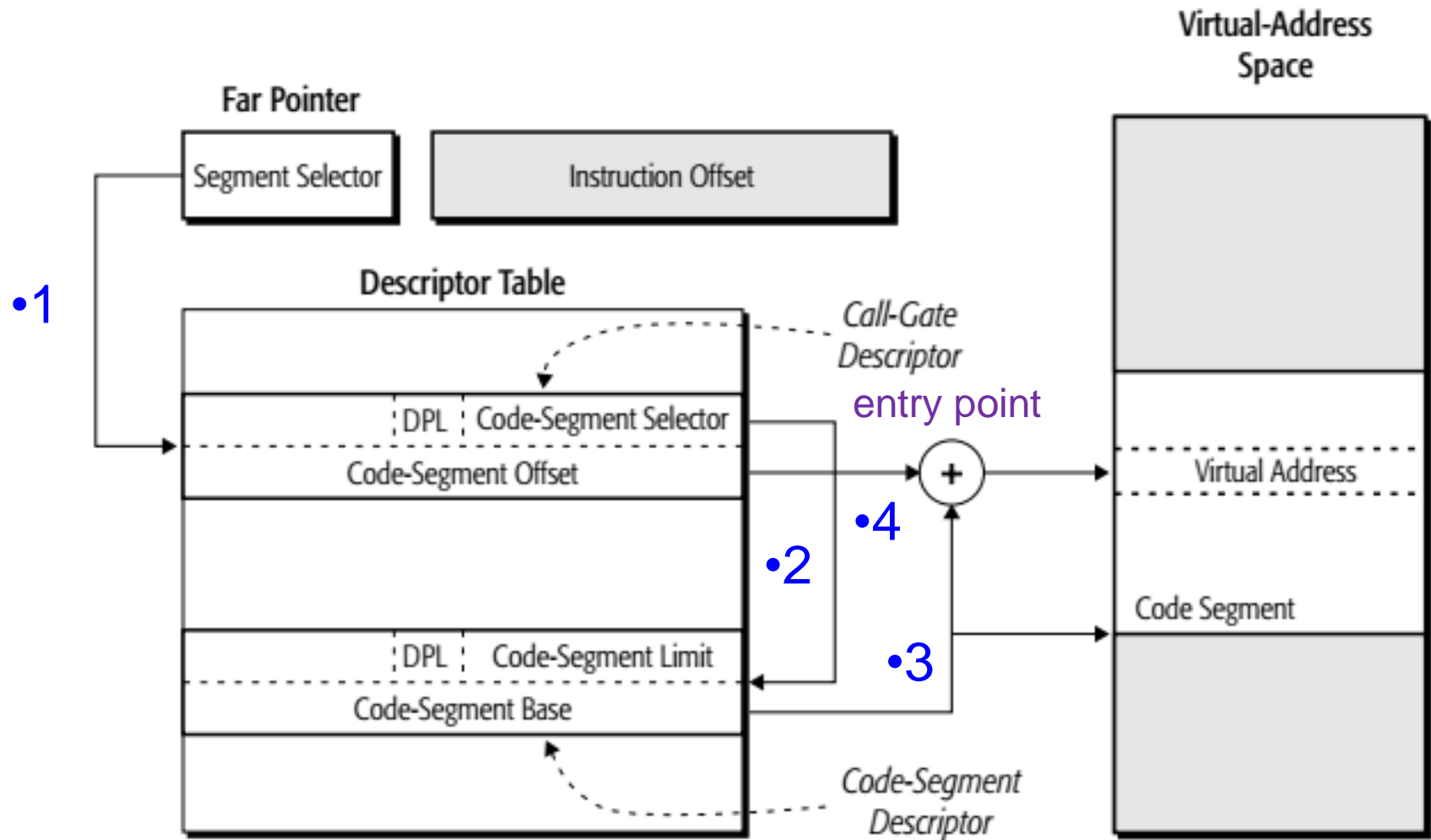
Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

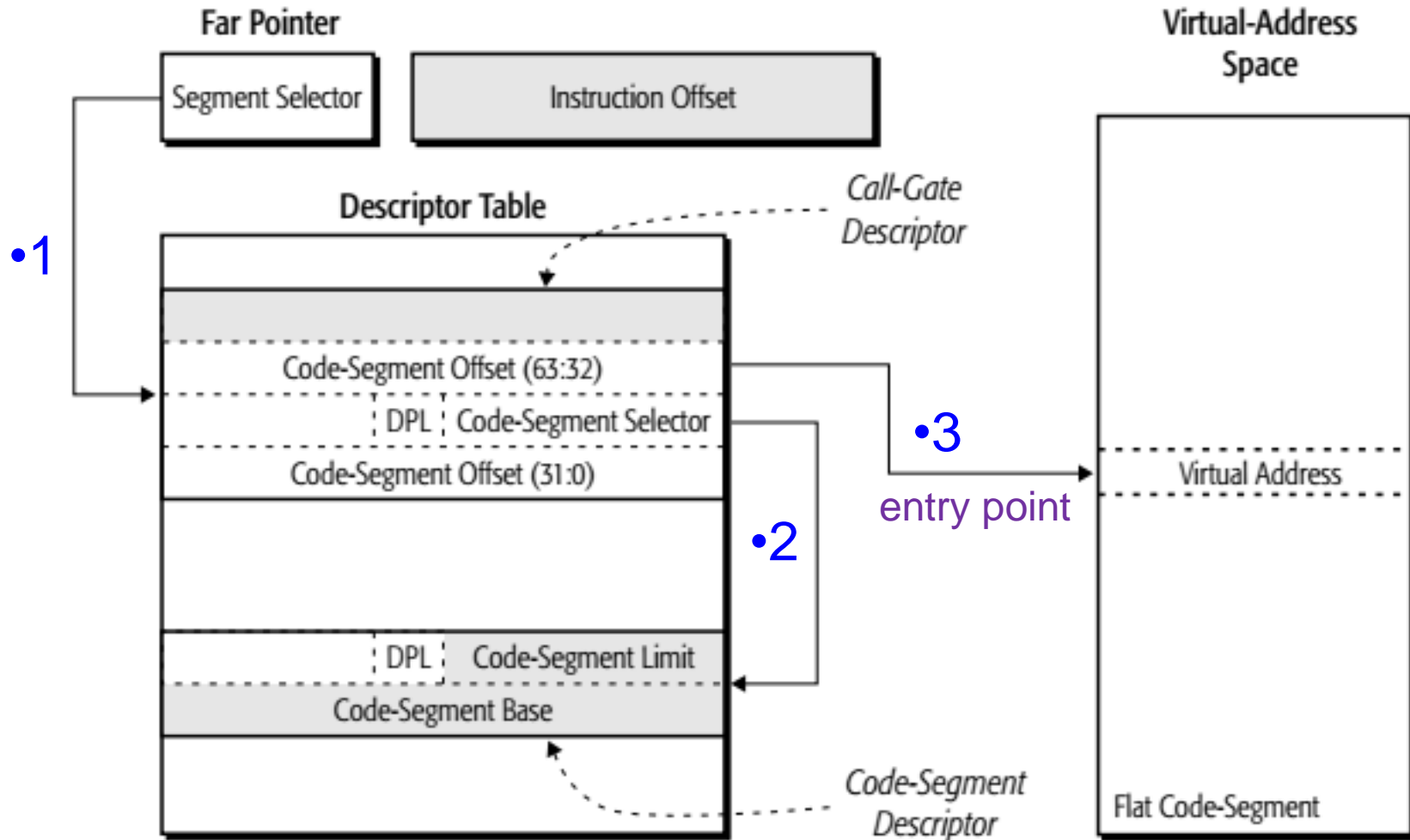


# Call Gate Transfer Mechanism



Legacy-Mode Call Gate Transfer Mechanism

# Call-Gate Transfer Mechanism



Long-Mode Call Gate Transfer Mechanism

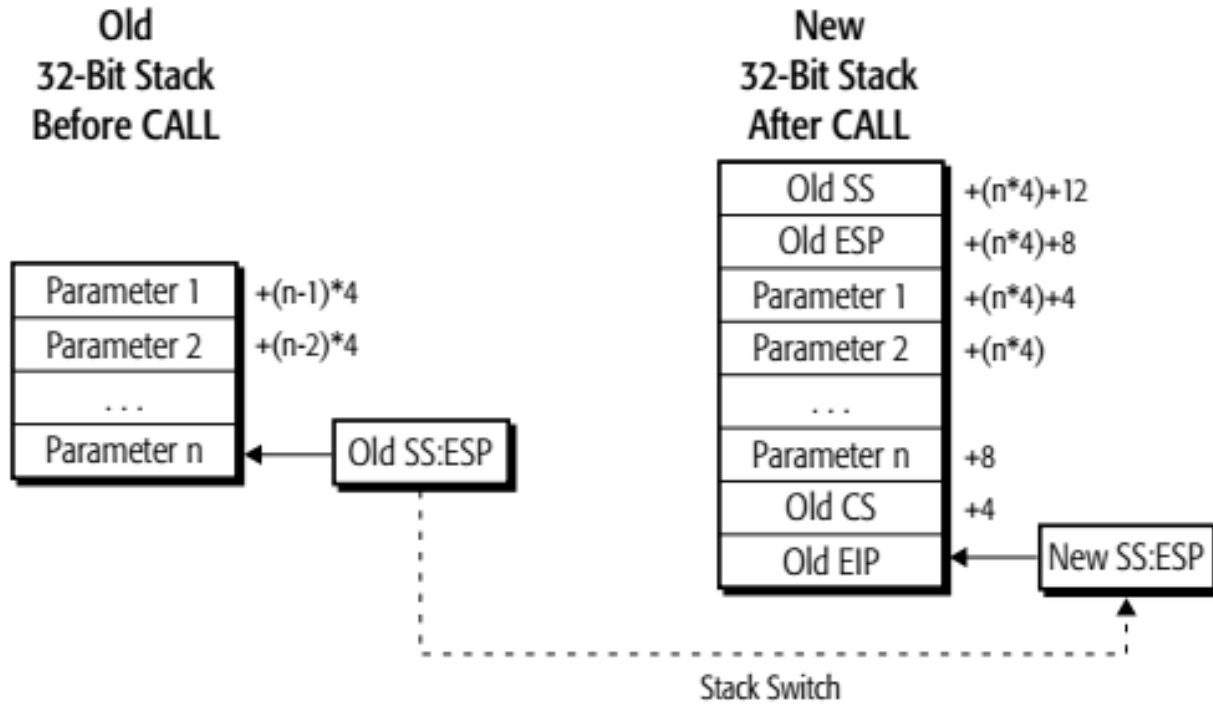
# Stack Switching

- Whenever a call gate is used to transfer program control to a more privileged code segment, the processor automatically switches to the stack for the destination code segment's privilege level.
- The purpose of stack switching is:
  - to prevent more privileged procedures from crashing due to insufficient stack space.
  - to prevent less privileged procedures from interfering (by accident or intent) with more privileged procedures through a shared stack.
- This happens transparently to the program code on both sides of the call gate.

# Legacy-Mode 32-Bit Stack Switch

- The processor performs the following stack-related steps in executing an interlevel CALL.
  - Temporarily save the current SS and ESP registers.
  - Load the new SS:ESP from the **task-state segment (TSS)**.
  - Push the temporarily saved SS:ESP onto the new stack.
  - **Copy the number of parameter specified in the parameter count field of the call gate** from the calling procedure's stack to the new stack. If the count is 0, no parameters are copied.
  - Push the return instruction pointer (the current contents of the CS and EIP registers) onto the new stack.
  - Load the segment selector for the new CS and the new EIP from the call gate into the CS and EIP, and begins execution of the called procedure.

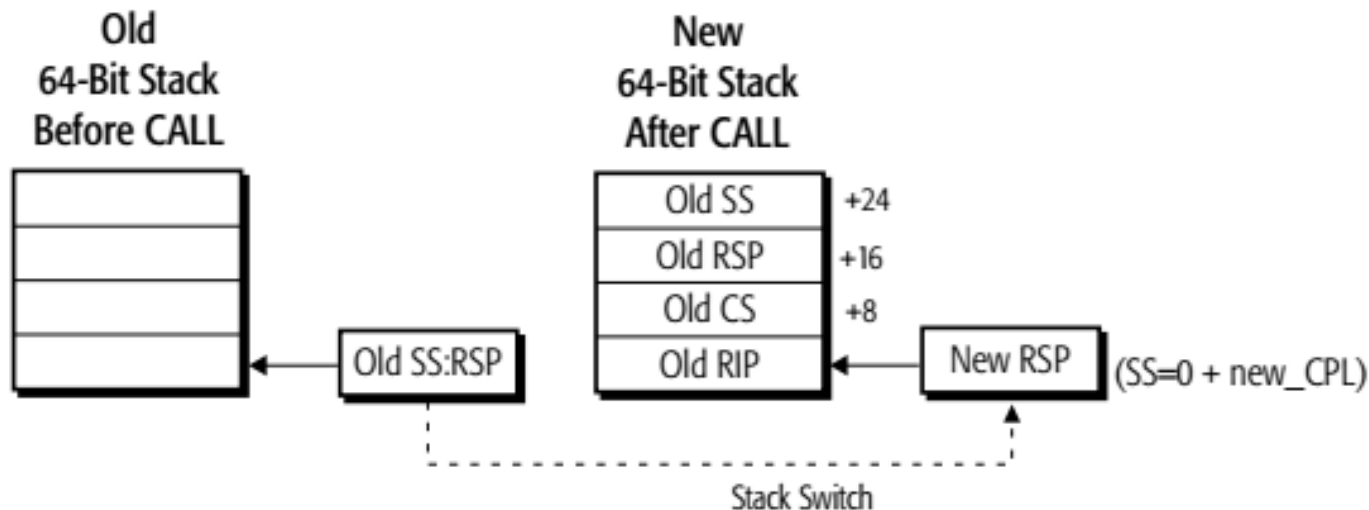
# Legacy-Mode Stack Switch



Legacy-Mode 32-Bit Stack Switch, with Parameters

- Most software does not use the call-gate descriptor count-field to pass parameters.
- System software typically defines linkage mechanisms that do not rely on automatic parameter copying.

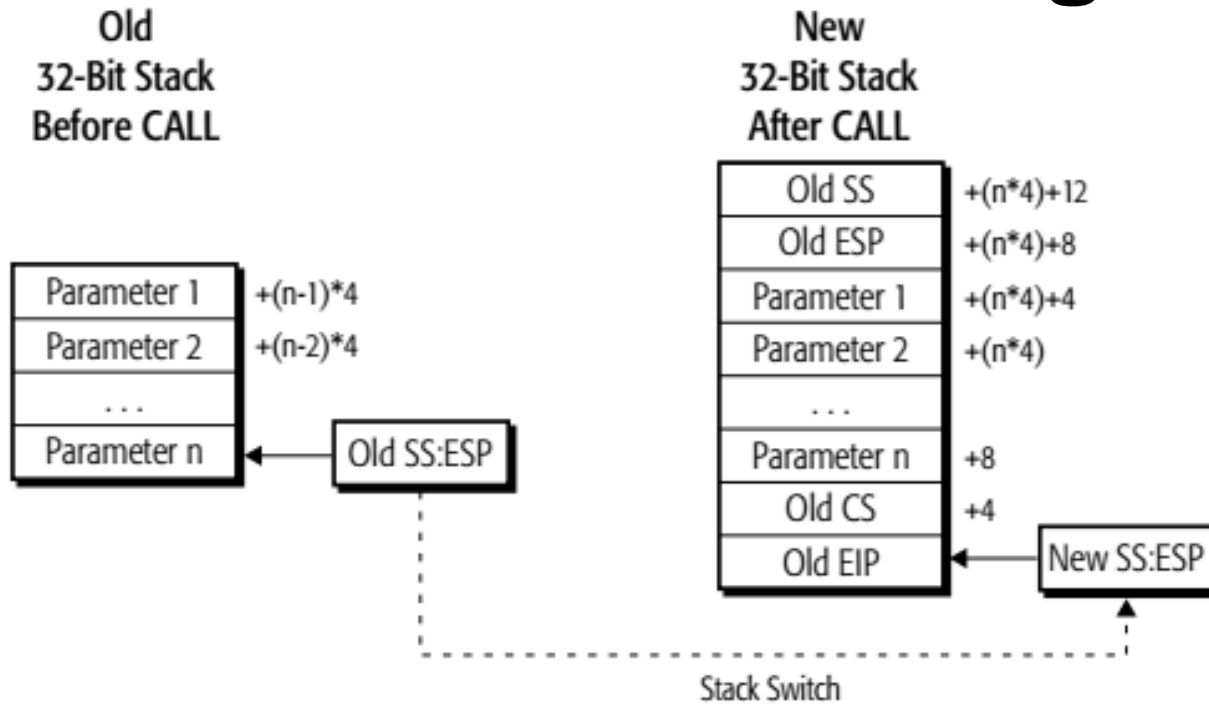
# Long Mode Stack Switch



## Long Mode Stack Switch

- The RSP is loaded with the new RSP value read from the TSS. The SS is loaded with a null selector (SS=0).
- In long mode, the **null selector acts as a flag indicating the existence of nested control transfers.**
- The automatic parameter-copy feature is no longer supported. The call-gate count field is ignored.

# Far Return to Less Privileged Code



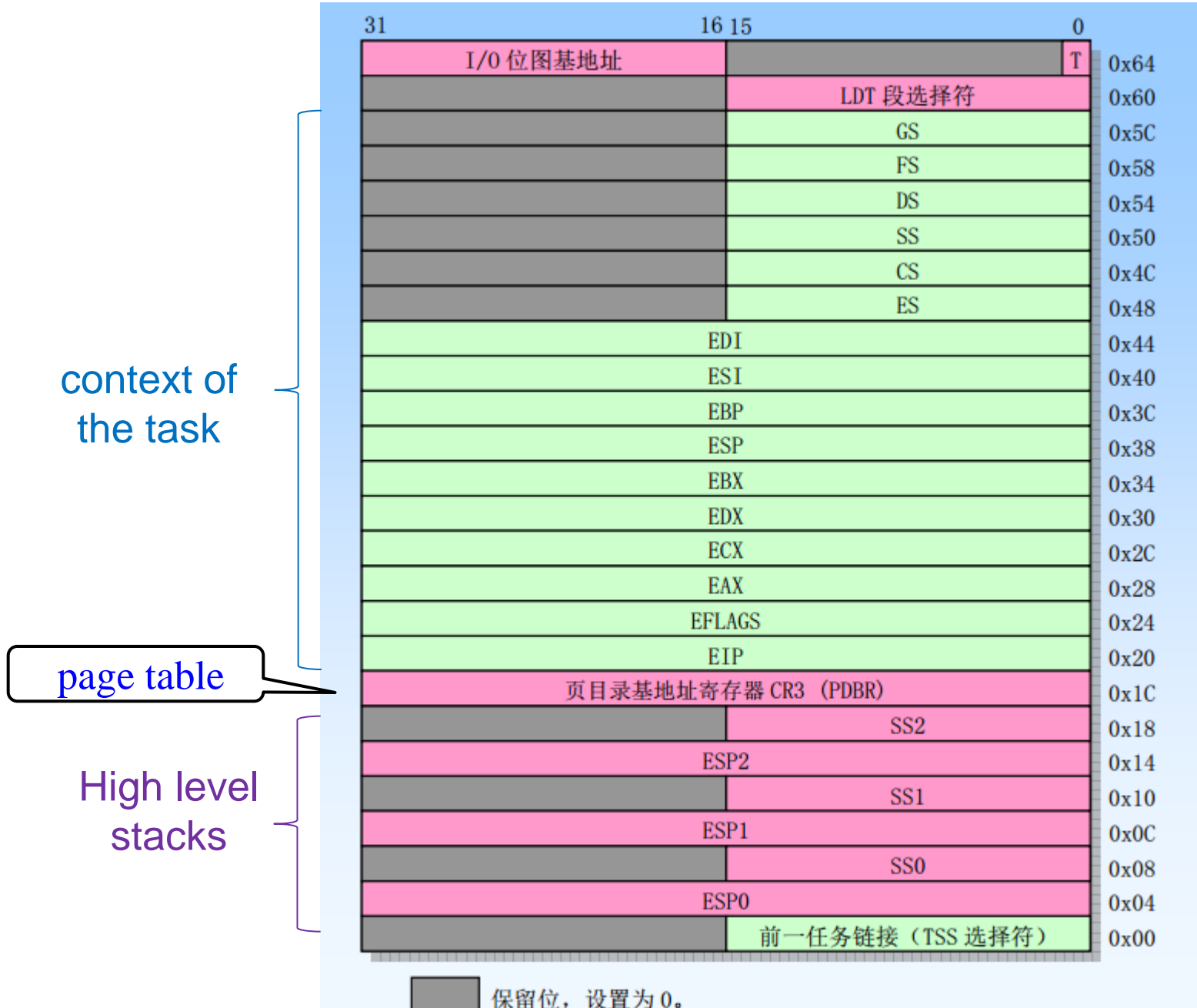
- Loads the old CS:EIP on the called procedure's stack.
- Adds the parameter count to the current ESP to step past the parameters on the called procedure's stack.
- Loads the SS and ESP registers with the saved SS:ESP values and switches back to the calling procedure's stack.
- Adds the parameter count to the current ESP to step past the parameters on the calling procedure's stack.

# Task-State Segment (TSS)

- The processor locates privilege level 0, 1, and 2 stack pointer via the task state segment (TSS).
- Each task has a separate TSS, thereby permitting tasks to have separate stacks.
- System software is responsible for creating TSSs and placing stack pointers in them.
- The initial stack pointers in the TSS are strictly read-only values.
- TSSs store all information prior to the task switch that is needed for the task restore.



# 32-Bit Task-State Segment (TSS)



# Fast System Call Instructions

- However, x86 operating systems are transitioning away from call gates. Because most other architectures do not support call gates, software interrupts or traps were preferred for portability, even though call gates are significantly faster than interrupts.
- With the introduction of system call instructions (SYSENTER/SYSEXIT by Intel and SYSCALL/SYSRET by AMD), a new faster mechanism was provided for control transfers.

# Fast System Call Instructions

- Call gates are more flexible than the SYS\* instructions since unlike the latter two, call gates allow for changing from an arbitrary privilege level to an arbitrary (albeit higher or equal) privilege level.
- The fast SYS\* instructions only allow control transfers from ring 3 to 0 and vice versa to reduce the privilege checking overhead.

- an example of system call "write" in Linux

### Linux 32-bit x86 Syscalls

```

mov eax,4    ; the system call
              ; number of "write"
mov ebx,1    ; first parameter: 1,
              ; the file descriptor
mov ecx,Str  ; data to write
mov edx,3    ; bytes to write
int 0x80     ; issue the system call

```

ret

### Linux 64-bit x86 System Calls

```

mov rax,1    ; the system call
              ; number of "write"
mov rdi,1    ; first parameter: 1,
              ; the file descriptor
mov rsi,Str  ; data to write
mov rdx,3    ; bytes to write
syscall      ; issue the system call

```

ret

Architecture	Syscall number	arg0	arg1	arg2	arg3	arg4	arg5
x86_64	rax	rdi	rsi	rdx	r10	r8	r9
x86	eax	ebx	ecx	edx	esi	edi	ebp

register-based parameter passing convention

- an example of system call “ZwClose” in Windows 10

```
NtClose      proc      near
    mov     r10, rcx
    mov     eax, 0Fh
    test    byte ptr ds:7FFE0308h, 1    ; test for 32-bit or 64-bit
    jnz     short loc_a
    syscall                                ; 64-bit system call
    ret

loc_a:
    int     2eh                          ; legacy way of system call
    ret
NtClose      endp
```

from ntdll.dll in Windows 10 X86\_64

# **SYSENTER/SYSEXIT**

- SYSENTER executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT.
- These instruction are optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system running at privilege level 0.
- The SYSENTER and SYSEXIT can be invoked from all operating modes except real mode.

# SYSENTER

- Prior to executing the SYSENTER, software must specify the privilege level 0 CS:EIP, and the privilege level 0 SS:ESP by writing values to the following **MSRs**:
  - IA32\_SYSENTER\_CS: The lower 16 bits of MSR are the segment selector for the privilege **level 0 code selector (CS)** and **stack selector (SS)**.
  - IA32\_SYSENTER\_EIP
  - IA32\_SYSENTER\_ESP
- These MSRs can be read from and written to using **RDMSR/WRMSR**.

# SYSENTER

- The following conventions must be followed:
  - The segment descriptors for the privilege level 0 CS and SS must be contiguous in a descriptor table. This convention allows the processor to compute the SS from CS ( $SS.Sel = CS.Sel + 8$ ).
  - The fast system call routines executed by user code must save the required return IP and processor state information if a return to the calling procedure is required.



# SYSEXIT

- SYSEXIT executes a fast return to privilege level 3 user code.
- Prior to executing SYSEXIT, software must specify the privilege level 3 CS:EIP, and the privilege level 3 SS:ESP by writing values into the following MSR and general-purpose registers:
  - IA32\_SYSENTER\_CS: CS, SS
  - RDX: EIP
  - ECX: ESP

# **SYSCALL/SYSRET**

- Like SYSENTER and SYSEXIT, SYSCALL and SYSRET are low-latency system call and return instructions designed for use by system and application software.
- The STAR, LSTAR, and CSTAR registers are model-specific registers (MSRs) used to specify the target address of a SYSCALL instruction as well as the CS and SS selectors of the called and returned procedures.

# SYSCALL/SYSRET

		63	48	47	32	31	0
<b>STAR</b>	C000_0081h	SYSRET CS and SS		SYSCALL CS and SS		32-bit SYSCALL Target EIP	
<b>LSTAR</b>	C000_0082h	Target RIP for 64-Bit-Mode Calling Software					
<b>CSTAR</b>	C000_0083h	Target RIP for Compatibility-Mode Calling Software					
<b>SFMASK</b>	C000_0084h	Reserved, RAZ				SYSCALL Flag Mask	

- **STAR**: target CS, EIP, SS, return CS, SS
- **LSTAR** and **CSTAR**: target RIP for long and legacy mode
- **SFMASK**: RFLAGS mask

# SYSCALL/SYSRET

- The SYSCALL instruction does not save the stack pointer (RSP). If the OS will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer.
- There is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read.

# SWAPGS Instruction

- The **SWAPGS** instruction provides a fast method for system software to load a pointer to system data structures.
- SWAPGS can be used upon entering system-software routines as a result of a SYSCALL instruction, an interrupt or an exception to obtain a pointer to kernel data structures.

# SWAPGS Instruction

- Before returning to application software, SWAPGS can restore an application data-structure pointer that was replaced by the system data structure pointer.
- SWAPGS exchanges the base-address value located in the **KernelGSbase** MSR register with the base-address value located in the GS selector register (**GS.base**).

- SWAPGS Instruction example
- At a kernel entry point, the OS uses SWAPGS to obtain a pointer to kernel data structures and simultaneously save the user's GS base. Upon exit, it uses SWAPGS to restore the user's GS base.

SystemCallEntryPoint:

SWAPGS	; get kernel pointer, save user GS.base
mov gs:[SavedUserRSP], rsp	; save user's stack pointer
mov rsp, gs:[KernelStackPtr]	; set up kernel stack
push rax	; now save user GPRs on kernel stack
.....	
	; perform system service
SWAPGS	; restore user GS, save kernel pointer
.....	

# 6-4 INTRO TO INTERRUPTS

- **Exceptions** and **interrupts** force control transfers from the currently-executing program to a system software service routine that handles the interrupting event.
- These routines are referred to as exception handlers and interrupt handlers , or **interrupt service procedure (ISP)**.
- During the control transfer to the ISP, the processor stops executing the interrupted program and saves its return pointer.



- The system-software service routine that handles the exception or interrupt is responsible for saving the state of the interrupted program.
- This allows the processor to restart the interrupted program after system has handled the event.
- The processor uses the **vector number** assigned to an exception or interrupt as an index into the **interrupt descriptor table (IDT)**. The IDT provides the entry point to an exception or interrupt handler.
- An IDT is used in all processor operating modes (real mode, protected mode, and long mode).

# Sources of Exceptions and Interrupts

- Exceptions and interrupts come from three general sources:
  - An **external interrupts** is an **asynchronous** event that is typically triggered by an I/O device.
  - An **exception** is a **synchronous** event that is generated when the processor detects predefined conditions, e.g., divide-by-zero-error, page fault.
  - A **software interrupt** occurs as a result of executing interrupt instructions, e.g. breakpoint trap (int3). Software interrupts are **synchronous** events.

# Precise Exceptions and Imprecise Exceptions

- **Precise exceptions** are reported on an instruction boundary.
  - Some report the boundary **before** the instruction causing the exception, while others report the boundary **after** the instruction causing the exception.
  - When the event handler returns to the interrupted program, it **can be restarted** at the interrupted-instruction boundary.
- **Imprecise exceptions** are not guaranteed to be reported on a predictable instruction boundary.
  - Imprecise events can be considered **asynchronous**.
  - The interrupted program is **not restartable**.

# Three Types of Exceptions

- **Faults** are **precise exceptions** reported on the boundary **before** the faulting instruction.
  - can be corrected and restarted with no loss of continuity.
  - the return address points to the faulting instruction.
- **Traps** are **precise exceptions** reported on the boundary **following** the trapping instruction.
  - can be continued without loss of program continuity.
  - the return address points to the trapping instruction.
- **Aborts** are imprecise exceptions and do not allow reliable program restart.

# Masking External Interrupts

- Software can mask the occurrence of certain exceptions and interrupts. **Masking** can delay or even prevent triggering of the exception-handling or interrupt-handling mechanism.
- External interrupts are classified as **maskable** or **nonmaskable**:
  - **Maskable interrupts** are triggered through the **INTR pin** by the interrupt-handling mechanism only when **FLAGS.IF=1**. Otherwise they are held pending for as long as the **FLAGS.IF** bit is cleared to 0.
  - **Nonmaskable interrupts (NMI)** are unaffected by the value of the **FLAGS.IF** bit.

# Interrupt Vectors

- Specific exception and interrupt sources are assigned a fixed vector-identification number (called “interrupt vector” or simply “vector”).
- The **interrupt vector** is used by the interrupt-handling mechanism to locate the service routine assigned to the exception or interrupt.
- Up to 256 unique interrupt vectors are available. Each vector contains the address of an ISP.
- Intel reserved the first 32 vectors for predefined exception and interrupt conditions.

- Interrupt vectors (32–255) are available to users.
- In real mode
  - a 4-byte number stored in the first 1024 bytes of memory (00000H–003FFH).
  - Each vector contains a value for IP and CS that forms the address of the ISP.
  - the first 2 bytes contain IP; the last 2 bytes CS.
- In protected mode, the vector table is replaced by an interrupt descriptor table that uses 8-byte descriptors to describe each of the interrupts.

# Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction. <sup>1</sup>
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.



# Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory. <sup>2</sup>
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent. <sup>3</sup>
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions <sup>4</sup>
20	#VE	Virtualization Exception	Fault	No	EPT violations <sup>5</sup>
21	#CP	Control Protection Exception	Fault	Yes	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.
22-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT <i>n</i> instruction.

# Double Fault Exception

- A **double fault exception** can occur when a second exception occurs during the handling of a prior (first) exception or interrupt handler.
  - refer to the AMD64 manual (Section 8.2.9)
- For example, a double fault occurs when a page fault is triggered but there is no page fault handler registered in the Interrupt Descriptor Table (IDT).

- Usually, the first and second exceptions can be handled sequentially without resulting in a double fault exception.
- In some cases, however, the first exception adversely affects the ability of the processor to handle the second exception, it signals the double fault exception instead.
- These exceptions contribute to the occurrence of a double fault exception, and are called **contributory exceptions**.

- Only very specific combinations of exceptions lead to a double fault. These combinations are:

### Double-Fault Exception Conditions

First Interrupting Event	Second Interrupting Event
Contributory Exceptions <ul style="list-style-type: none"> <li>• Divide-by-Zero-Error Exception</li> <li>• Invalid-TSS Exception</li> <li>• Segment-Not-Present Exception</li> <li>• Stack Exception</li> <li>• General-Protection Exception</li> </ul>	Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception
Page Fault Exception	Page Fault Exception Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception

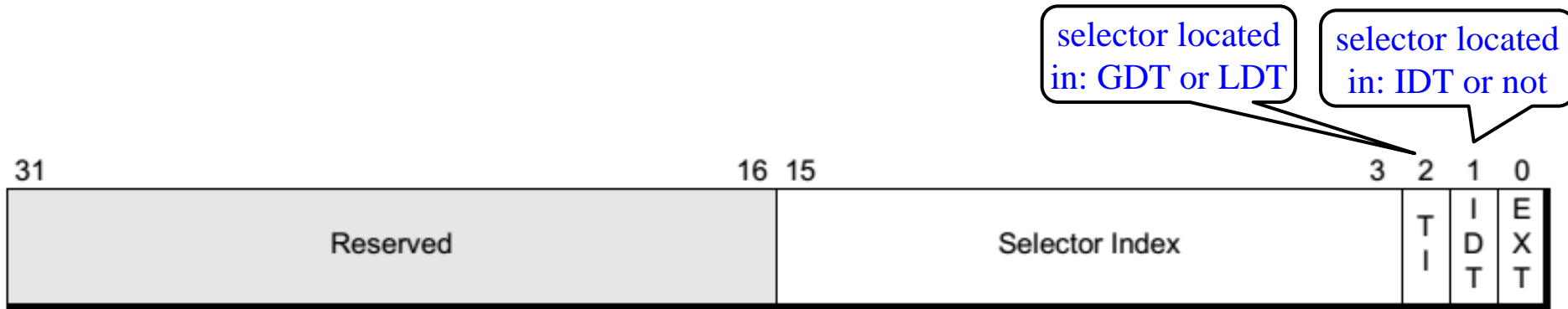
- For example, a divide-by-zero fault followed by a page fault is fine, but a divide-by-zero fault followed by a general-protection fault leads to a double fault.

- If another contributory or page fault exception occurs while attempting to call the double-fault handler, the processor enters shutdown mode.
- It is really important to provide a double fault handler, because if a double fault is unhandled, a fatal **triple fault** occurs, which in turn initiates a CPU reset.

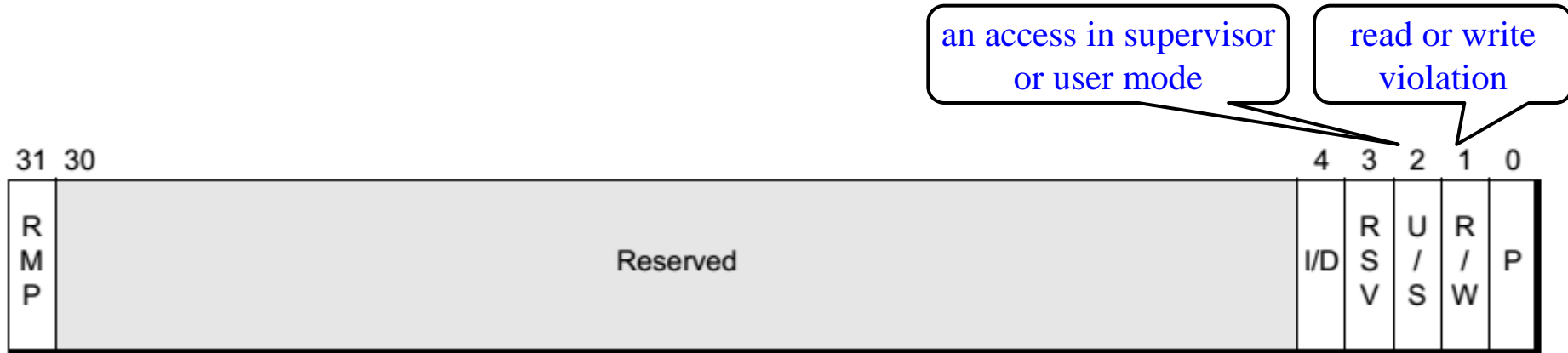
# Error Codes

- The processor exception-handling mechanism reports error and status information for some exceptions using an error code.
- The **error code** is pushed onto the stack by the exception-mechanism during the control transfer into the exception handler.
- The error code has two formats:
  - a **selector format** for error-reporting exceptions
  - a **page-fault format** for page faults

# Error Codes



## Selector-Error Code



## Page-Fault Error Code

# Priority among Simultaneous Exceptions and Interrupts

- When simultaneous interrupts occur, the processor transfers control to the highest-priority interrupt handler.
- Lower-priority interrupts from external sources are held pending by the processor, and they are handled after the higher-priority interrupt.
- The x86 architecture defines priorities between groups of interrupts, and interrupt prioritization within a group is implementation dependent.



# Simultaneous Interrupt Priorities

Interrupt Priority	Interrupt Condition	Interrupt Vector
(High) 0	Processor Reset	—
	Machine-Check Exception	18
1	External Processor Initialization (INIT)	—
	SMI Interrupt	
	External Clock Stop (Stpclk)	
2	Data, and I/O Breakpoint (Debug Register)	1
	Single-Step Execution Instruction Trap (RFLAGS.TF=1)	
3	Non-Maskable Interrupt	2
4	Maskable External Interrupt (INTR)	32–255
5	Instruction Breakpoint (Debug Register)	1
	Code-Segment-Limit Violation <sup>1</sup>	13
	Instruction-Fetch Page Fault <sup>1</sup>	14

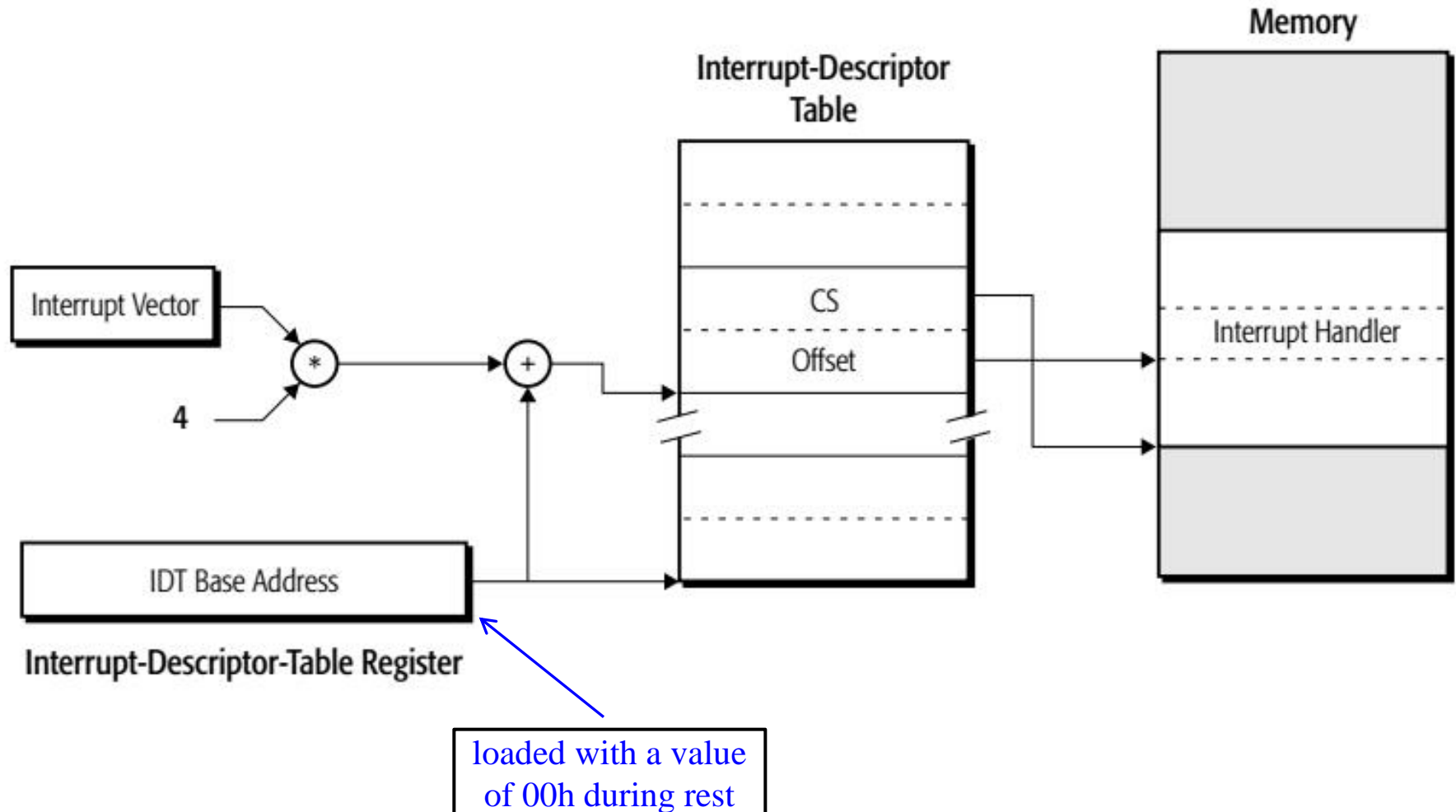
# Simultaneous Interrupt Priorities

Interrupt Priority	Interrupt Condition	Interrupt Vector
6	Invalid Opcode Exception <sup>1</sup>	6
	Device-Not-Available Exception	7
	Instruction-Length Violation (> 15 Bytes)	13
7	Divide-by-zero Exception	0
	Invalid-TSS Exception	10
	Segment-Not-Present Exception	11
	Stack Exception	12
	General-Protection Exception	13
	Data-Access Page Fault	14
	Floating-Point Exception-Pending Exception	16
	Alignment-Check Exception	17
	SIMD Floating-Point Exception	19
	Hypervisor Injection Exception	28
	VMM Communication Exception	29

# Real-Mode Interrupt Control Transfers

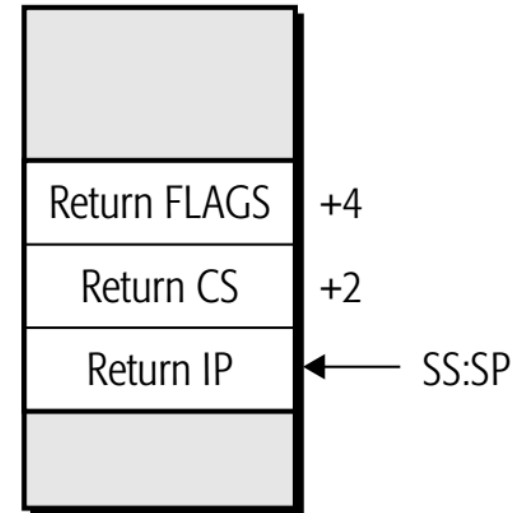
- In real mode, the IDT is a table of 4-byte entries, one entry for each of the 256 possible interrupts implemented by the system. The real mode IDT is often referred to as an **interrupt vector table (IVT)**.
- IVT table entries contain a far pointer (CS:IP pair) to an exception or interrupt handler. The base of the IDT is stored in the IDTR register, which is loaded with **a value of 00h** during a processor reset.

# Real-Mode Interrupt Control Transfers



# Real-Mode Interrupt Control Transfers

- When an exception or interrupt occurs in real mode, the processor performs the following:
  - Pushes the FLAGS onto the stack
  - Clears EFLAGS.IF to 0
  - Pushes the CS and IP onto the stack
  - Locates the ISP in the IDT by scaling the interrupt vector by four
  - Transfers control to the ISP referenced by the CS:IP in the IDT



Stack After Interrupt  
in Real Mode

# Real-Mode Interrupt Control Transfers

- An **IRET** instruction is used to return to the interrupted program.
- When an IRET is executed, the processor performs the following:
  - Pops the saved CS:IP off the stack
  - Pops the FLAGS value off of the stack
  - Execution begins at the saved CS.IP location

# Protected-Mode Interrupt Control Transfers

- In protected mode, the interrupt mechanism transfers control to an exception or interrupt handler through gate descriptors.
- The IDT is a table of 8-byte gate entries, one for each of the 256 possible interrupt vectors implemented by the system.
- Three gate types are allowed in the IDT:
  - Interrupt gates
  - Trap gates
  - Task gates

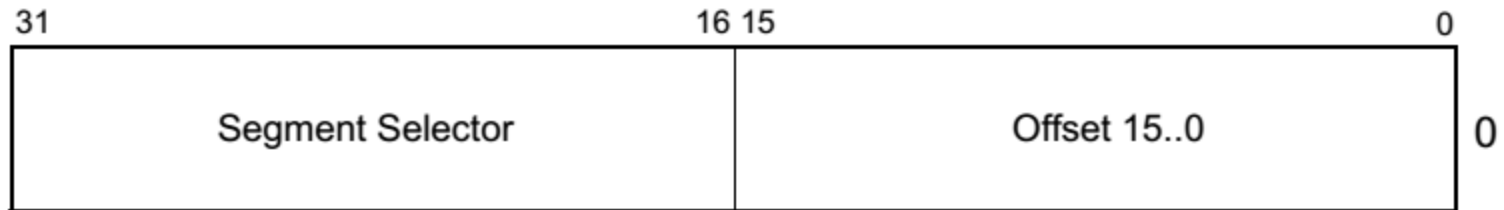
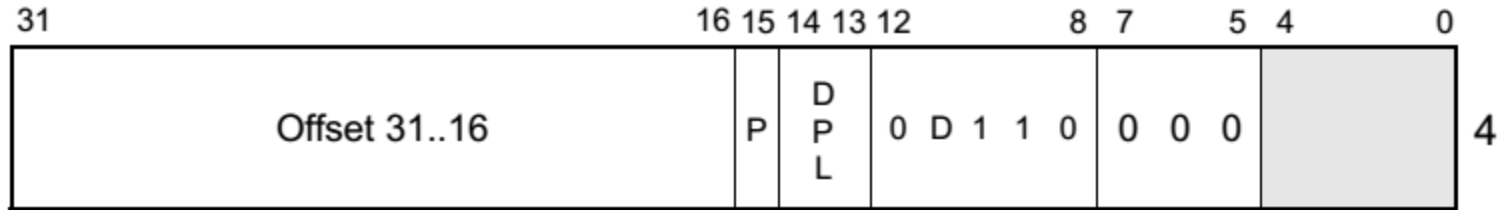
# Protected-Mode Interrupt Control Transfers

- Interrupt and trap gates are very similar to call gates. They contain a far pointer (segment selector and offset) to transfer program execution to an ISP.
- The only difference between an interrupt gate and a trap gate is:
  - Accessing an exception through an interrupt gate, the processor clears the IF flag to prevent other interrupts from interfering with the current ISP.
  - Accessing a handler procedure through a trap gate does not affect the IF flag.

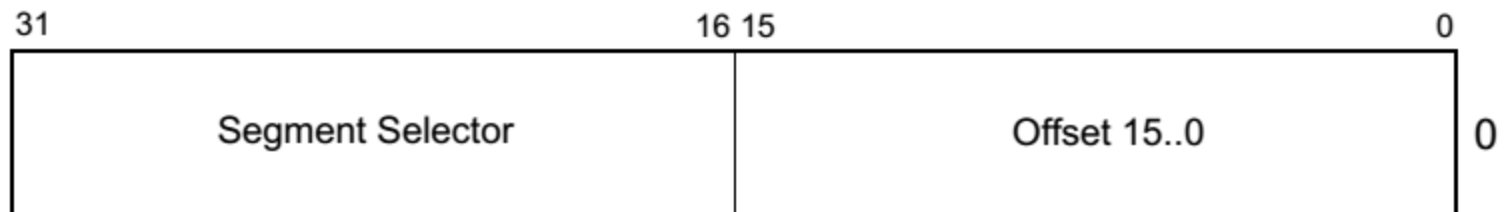
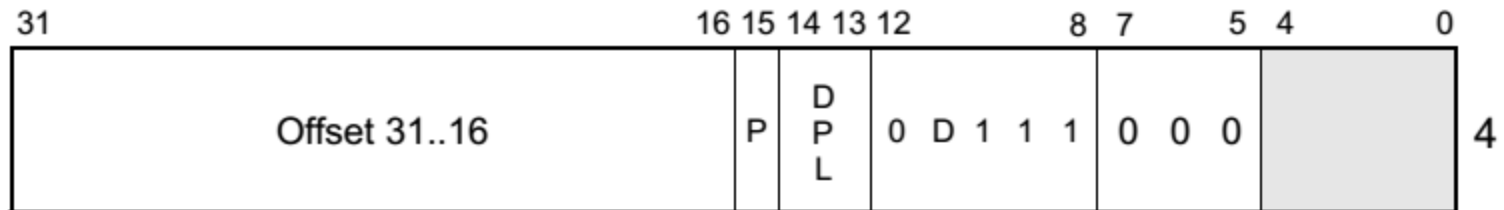


# Protected-Mode Interrupt Control Transfers

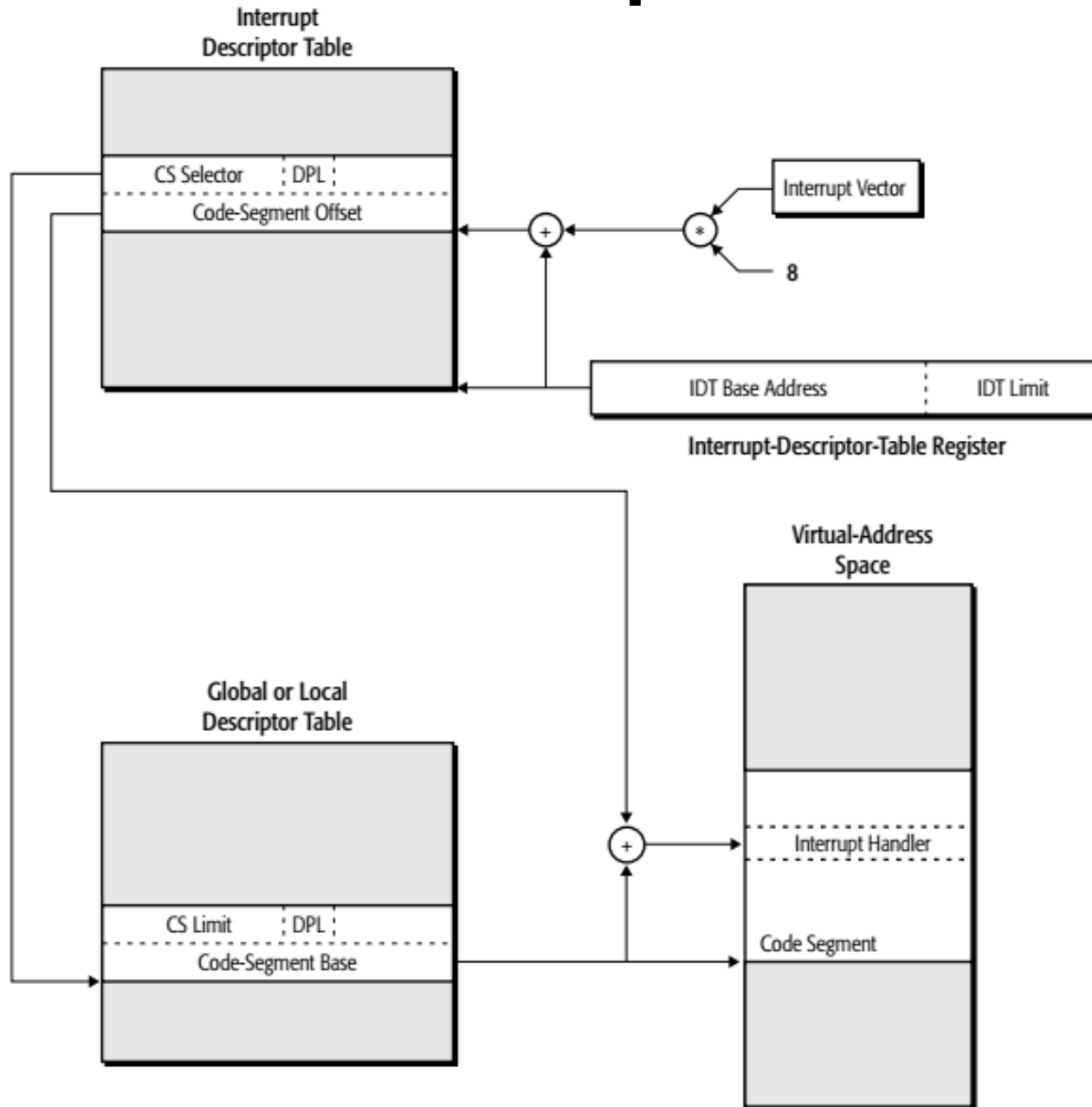
## Interrupt Gate



## Trap Gate

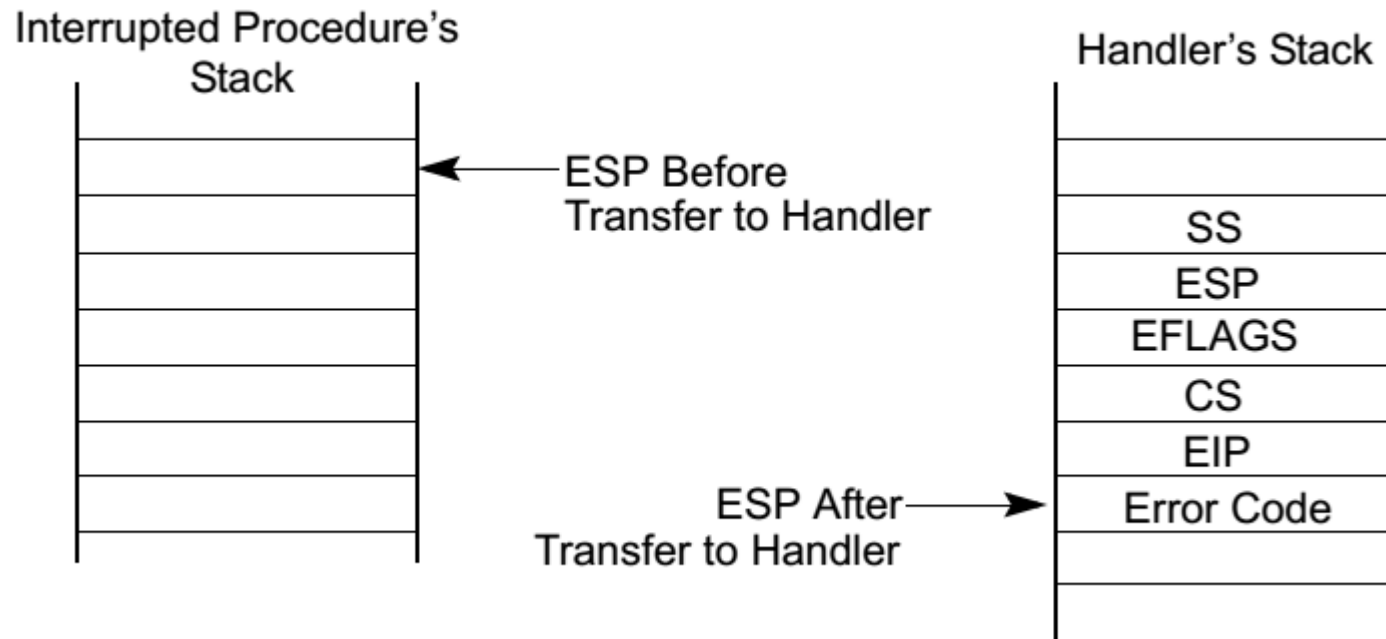


# Protected-Mode Interrupt Control Transfers



# Protected-Mode Interrupt Control Transfers

- If the ISP has the same privilege level as the current program, the ISP uses the current stack.
- If the ISP executes at a more privileged level, the processor performs a **stack switch** by loading the new SS:ESP from the **TSS**.



Stack Usage with Privilege-Level Change

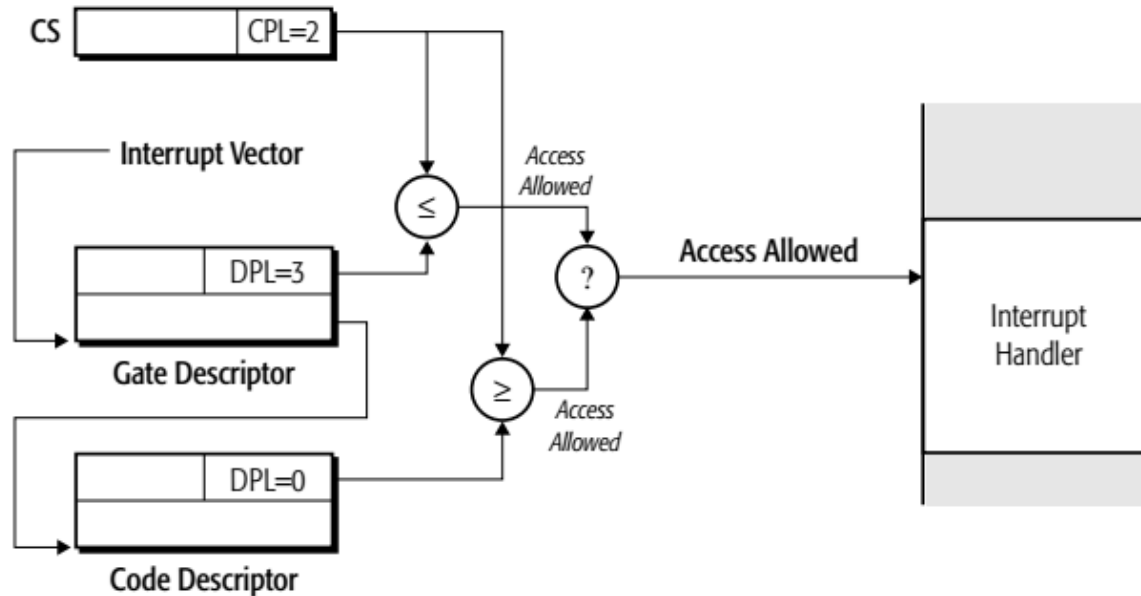
# Protected-Mode Interrupt Control Transfers

- The processor performs **privilege checks** similar to those performed on call gates. The checks are performed when either conforming or nonconforming interrupt handlers are referenced.
- The exception or interrupt handler can be placed **in a conforming code segment**. This technique can be used for handlers that only need to access data available on the stack (e.g., divide error exceptions).

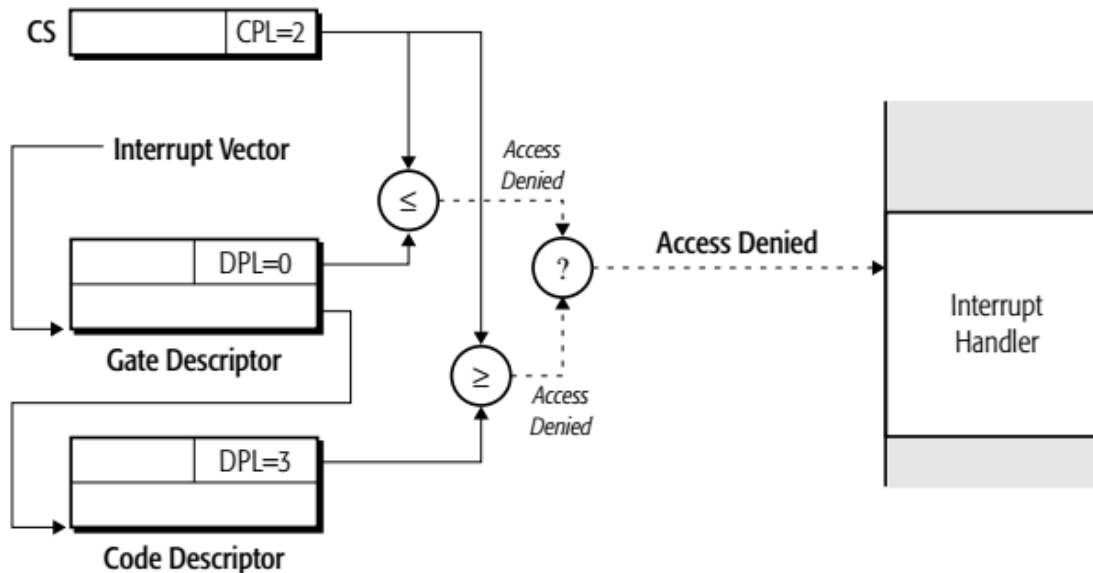
# Protected-Mode Interrupt Control Transfers

- The handler can be placed in a nonconforming code segment with high privilege level. The following check rules are performed to avoid privilege-level violations:
  - The CPL must be numerically less-than or equal-to the gate DPL.
  - The CPL must be numerically greater-than or equal-to the ISP code-segment DPL.
- Unlike call gates, no RPL comparison takes place. Because in an ISP, the only referenced data is an interrupt vector rather than a selector and no RPL field exists.

# Privilege-Check Examples for Interrupts



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

# Interrupt Instructions

- Three different interrupt instructions available:
  - INT N, INT3 and INTO and INT1 (INT0 is exactly INT4)
- In real mode, each fetches a vector from the vector table, and then calls the procedure stored at the location addressed by the vector.
- In protected mode, each fetches an interrupt descriptor from the interrupt descriptor table.
- Similar to a far CALL instruction because it places the return address (IP/EIP and CS) on the stack.

# INT N

- 256 different software interrupt instructions (INT N) available to the programmer.
  - each INT instruction has a numeric operand whose range is 0 to 255 (00H–FFH)
- In real mode, address of the interrupt vector is determined by multiplying the interrupt type number by 4.
- For example, INT 10H instruction calls the interrupt service procedure whose address is stored beginning at memory location 40H ( $10H \times 4$ ).



- For example, INT 100 uses interrupt vector 100, which appears at memory address 190H–193H ( $64H \times 4$ ).
- In protected mode, the interrupt descriptor is located by multiplying the type number by 8
  - because each descriptor is 8 bytes long
- Each INT N instruction is 2 bytes long.
  - the first byte contains the opcode
  - the second byte contains the vector type number

- When a software interrupt executes, it:
  - pushes the flags onto the stack
  - clears the IF flag bits
  - pushes CS onto the stack
  - fetches the new value for CS from the interrupt vector
  - pushes IP/EIP onto the stack
  - fetches the new value for IP/EIP from the vector
  - jumps to the new location addressed by CS and IP/EIP

- INT N performs as a far CALL
  - not only pushes CS & IP onto the stack, also pushes the flags onto the stack
- The INT N instruction performs the operation of a PUSHF, followed by a far CALL instruction.
- Software interrupts are most commonly used to call system procedures because the address of the function need not be known.
- The interrupts often control printers, video displays, and disk drives.

- INT N replaces a far CALL that would otherwise be used to call a system function.
  - INT N instruction is 2 bytes long, whereas the far CALL is 5 bytes long
- Each time that the INT N instruction replaces a far CALL, it saves 3 bytes of memory.
- This can amount to a sizable saving if INT N often appears in a program, as it does for system calls.

# ***IRET/IRETD***

- Used only with software or hardware interrupt service procedures.
- IRET instruction will
  - pop stack data back into the IP
  - pop stack data back into CS
  - pop stack data back into the flag register
- Accomplishes the same tasks as the POPF followed by a far RET instruction.

- When IRET executes, it restores the contents of IF from the stack.
  - preserves the state of these flag bits
- If interrupts were enabled before an interrupt service procedure, they are automatically re-enabled by the IRET instruction.
  - because it restores the flag register
- IRET is used in real mode and IRETD in the protected mode.

# An Interrupt Service Procedure Example

- Suppose a procedure is required to add the contents of DI, SI, BP, and BX and save the sum in AX.
- As a common task, it may be worthwhile to develop the task as a software interrupt.
- It is also important to save all registers that are changed by the procedure using USES.

```
INTS    PROC    FAR  USES  AX
        ADD     AX, BX
        ADD     AX, BP
        ADD     AX, DI
        ADD     AX, SI
        IRET
INTS    ENDP
```

employ 'USES' statement to save AX

ends with IRET instead of RET

# INT N versus Far Call

- The action of the INT N instruction (including the INTO, INT3 instructions) is similar to that of a far call made with the CALL instruction.
- The primary differences are:
  - INT N instruction is 2 bytes long, whereas the far CALL is 5 bytes long.
  - With the INT N instruction, the EFLAGS register is pushed onto the stack before the return address.
  - Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

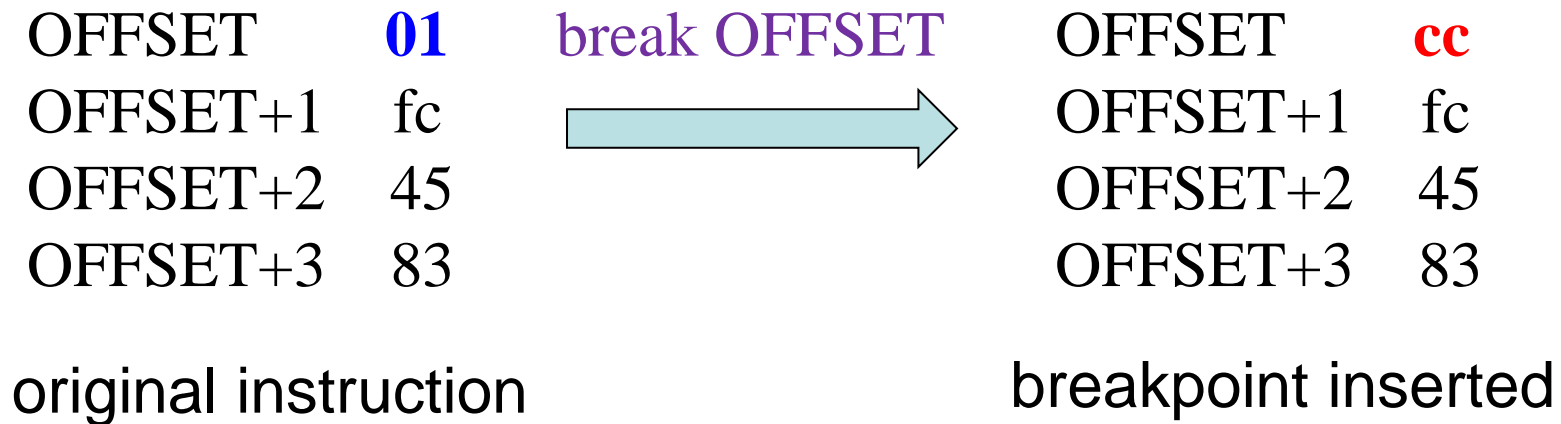


# INT3

- A special software interrupt designed to function as a breakpoint.
  - a 1-byte instruction (0xCC), while others are 2-byte
- Common to insert an INT3 in software to interrupt or break the flow of the software.
  - function is called a breakpoint
  - breakpoints help to debug faulty software
- A breakpoint occurs for any software interrupt, but because INT3 is 1 byte long, it is easier to use for this function.

# How GDB Implements Breakpoint

- Suppose the original instruction at OFFSET is 0x8345fc**01** (add [ebp-0x8], 1).
- When we type "break OFFSET" in GDB, it will memorize last byte (0x01) of this word, and change the word to (0x8345fc**cc**).



- When this program hits the INT3 instruction GDB just inserted, the debugged program will trap into the kernel, and the kernel will in turn signal GDB.
- GDB will then restore the byte at OFFSET using the original value it memorized, and move the instruction pointer EIP back to OFFSET to restart the instruction at OFFSET.

after breakpoint  
EIP points here

OFFSET	<b>cc</b>
OFFSET+1	fc
OFFSET+2	45
OFFSET+3	83

breakpoint inserted

restart the  
instruction



OFFSET	<b>01</b>
OFFSET+1	fc
OFFSET+2	45
OFFSET+3	83

instruction restored

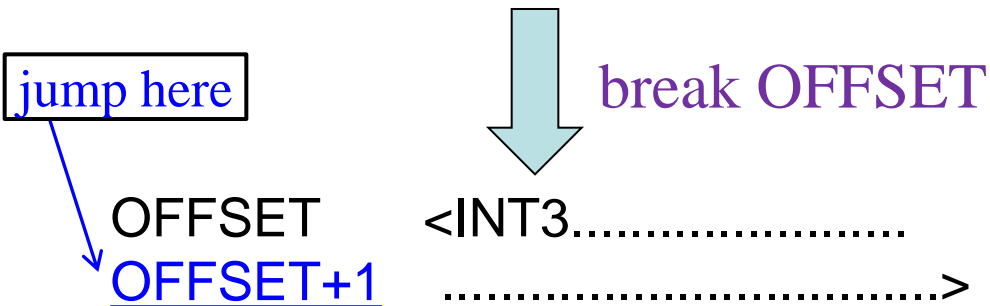
GDB restores  
instruction and  
moves EIP here



# Why INT3 should be one byte?

- Suppose INT3 is longer than some x86 instructions. When GDB insert a breakpoint, it may overwrite more than one instructions, which can cause problems.
- Consider the following example with two one-byte instructions:

OFFSET     <instruction 1, 1 byte>  
OFFSET+1   <instruction 2, 1 byte>



- If a code wants to jump to “OFFSET+1”, it will jump to the middle of INT3 instruction, which can create undefined behaviors.

- We don't have this problem if INT3 is one byte, the minimum length of all x86 instructions.

# INTO

- Interrupt on overflow (INTO) is a conditional software interrupt that tests overflow flag (O).
  - If  $O = 0$ , INTO performs no operation
  - if  $O = 1$  and an INTO executes, an interrupt occurs via vector type number 4
- The INTO instruction appears in software that adds or subtracts signed binary numbers.
  - with these operations, it is possible to have an overflow
- JO or INTO instructions detect the overflow.

# Interrupt Control

- Two instructions control the maskable interrupts (connect to the CPU INTR pin).
- The **set interrupt flag instruction (STI)** places 1 in the I flag bit.
  - which enables the INTR pin
- The **clear interrupt flag instruction (CLI)** places a 0 into the I flag bit.
  - which disables the INTR pin
- The STI instruction enables INTR and the CLI instruction disables INTR.

- In software interrupt service procedure, hardware interrupts are enabled as one of the first steps.
  - accomplished by the STI instruction
- Interrupts are enabled early because just about all of the I/O devices in the personal computer are interrupt-processed.
  - if interrupts are disabled too long, severe system problems result

# Interrupts in the Personal Computer

- Interrupts found in the personal computer only contained Intel-specified interrupts 0–4.
- Access to protected mode interrupt structure in use by Windows is accomplished through kernel functions Microsoft provides.
  - and cannot be directly addressed
- Protected mode interrupts use an interrupt descriptor table.



# 64-Bit Mode Interrupts

- The 64-bit system expands the legacy interrupt-mechanism. These change include:
  - The size of an interrupt-stack push is fixed at 8 bytes and the stack frame is aligned on a 16-byte boundary.
  - The stack pointer, SS:RSP, is pushed unconditionally on interrupts, rather than conditionally based on a change in CPL.
  - The SS selector register is loaded with a null selector as a result of an interrupt.
  - The IRETQ instruction is used to return from an interrupt service procedure.
  - The IRETQ instruction unconditionally pops SS:RSP.

# 6–5 MACHINE CONTROL AND MISCELLANEOUS INSTRUCTIONS

- These instructions provide control of the carry bit, stop instruction execution, and perform various other functions.

# Controlling the Carry Flag Bit

- The carry flag (C) propagates the carry or borrow in multiple-word/doubleword addition and subtraction.
  - can indicate errors in assembly language procedures
- Three instructions control the contents of the carry flag:
  - STC (set carry), CLC (clear carry), and CMC (complement carry)

# HLT

- HLT stops instruction execution and places the processor in a HALT state.
- Several ways to exit a halt
  - an enabled interrupt (NMI and SMI)
  - a debug exception
  - a hardware reset (BINIT#, INIT# or RESET# signal)
- The saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

# NOP

- In early years, before software development tools were available, a NOP, which performs absolutely no operation, was often used to pad software with space for future machine language instructions.
- When the microprocessor encounters a NOP, it takes a short time to execute.

- If you are developing machine language programs, it is recommended that you place 10 or so NOPS in your program at 50-byte intervals.
  - in case you need to add instructions at some future point
- A NOP may also find application in time delays to waste time.
- A NOP used for timing is not very accurate because of the cache and pipelines in modern microprocessors.

# LOCK Prefix

- Appends an instruction and causes the `pin` to become a logic 0.
- *LOCK* pin often disables external bus masters or other system components
  - causes pin to activate for duration of instruction
- If more than one sequential instruction is locked, *LOCK* pin remains logic 0 for duration of the sequence of instructions.
- The `LOCK:MOV AL,[SI]` instruction is an example of a locked instruction.

# BOUND

- A comparison instruction that may cause an interrupt (vector type number 5).
- Compares the contents of any 16-bit or 32-bit register against the contents of two words or doublewords of memory
  - an upper and a lower boundary
- If register value compared with memory is not within the boundary, a type 5 interrupt ensues.
- If it is within the boundary, the next instruction in the program executes.



# BOUND

- For example, if the **BOUND SI,DATA** instruction executes, word-sized location DATA contains the lower boundary, and word-sized location DATA+2 bytes contains the upper boundary.
- If the number contained in SI is less than memory location DATA or greater than memory location DATA+2 bytes, a type 5 interrupt occurs.

$SI < [DATA]$  or  $SI > [DATA+2]$ : raise a BOUND RANGE exceeded exception

# ENTER and LEAVE

- **ENTER** and **LEAVE** instructions create and release stack frames for called procedures.
- The stack frames have predefined spaces for local variables and the necessary pointers to allow coherent returns from called procedures.
- They also provide a called procedure with access points to other variables located in **nested stack frames**.

- Syntax: **ENTER** stack space, nesting levels
  - The first operand specifies the size of the dynamic storage in the stack frame.
  - The second operand specifies the nesting level (0 to 31—the value is automatically masked to 5 bits).
- For example:

SUM PROC

push ebp ; Save old frame pointer  
 mov ebp, esp ; Point ebp to top-of-stack  
 sub esp, 10 ; Reserve 10 bytes of locals

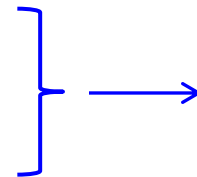
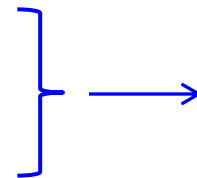
...

mov esp, ebp ; Restore ebp and remove  
                   ; stack space for locals.

pop ebp

ret

SUM ENDP



SUM PROC

enter 10, 0

...

...

...

leave

ret

SUM ENDP

- In the following example **ENTER 2048,3** allocates 2 Kbytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame.
- The **nesting level** determines the number of stack frame pointers to copy into the new stack frame from the preceding frame before adjusting the stack pointer, which allows multiple parent frames to be accessed from the called function.

- Nested functions are not supported by standard C; local variables have only the scope of the function they are declared in.

```
void func_a(void)
{
    int a1 = 7;

    void func_b(void)
    {
        printf("a1 = %d\n", a1);
    }

    func_b();
}
```

nested function declaration

a1 is inherited from func\_a()

```
def func_a():
    a1 = 7
    def func_b():
        print 'a1 = %d' % a1
    func_b()
```

nested function in Python

- Why Intel invests ENTER to support nested functions?

- Note that the x86 was originally designed as a Pascal machine, which is why there are special instructions to support its features:
  - nested functions using **enter** and **leave**
  - the **pascal calling convention** where the callee pops numbers of arguments from the stack using **ret N**
  - **bounds checking** using **bound**

#### 7. 8086 High-Level-Language Programming (PL/M)

*Who Needs High-Level Languages?*, 171  
*Structure of PL/M-86 Programs*, 173  
*Tokens*, 175  
*Expressions*, 177  
*Statements*, 179  
*Executable Statements*, 180  
*Declarative Statements*, 185  
*Procedures*, 192  
*Block Structure and Scope*, 198  
*Input and Output*, 200  
*Modular Programming*, 201  
*Tying It All Together*, 203  
*In Conclusion*, 205

#### 8. 8086 High-Level-Language Programming (Pascal)

*Who Needs High-Level Languages?*, 207  
*Structure of Pascal Programs*, 208

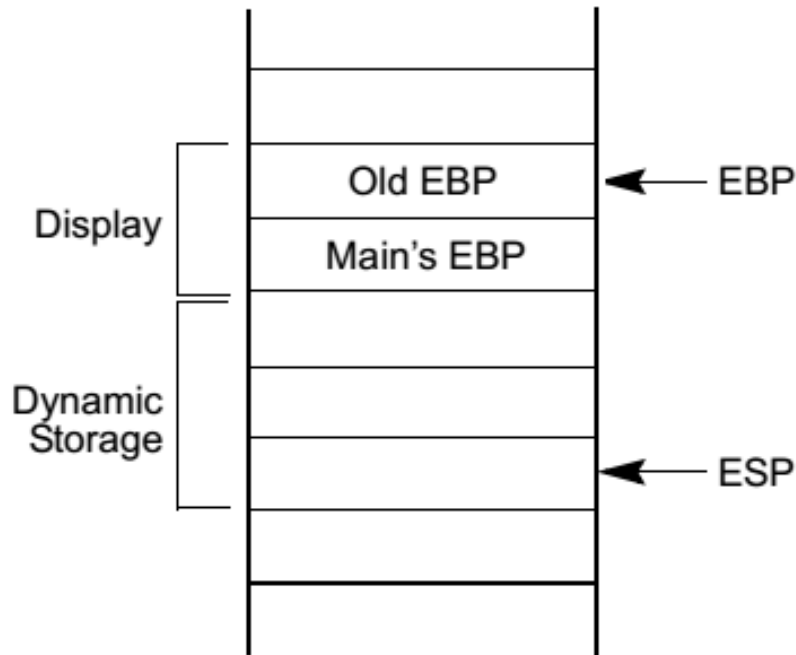
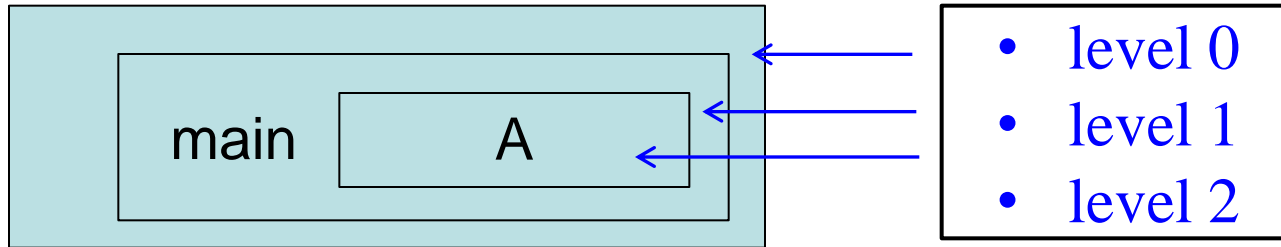
from 《The 8086/8088 Primer》  
([www.stevemorse.org/8086/index.html](http://www.stevemorse.org/8086/index.html))

written by **Stephen P. Morse**:  
**Father of the 8086 Processor**

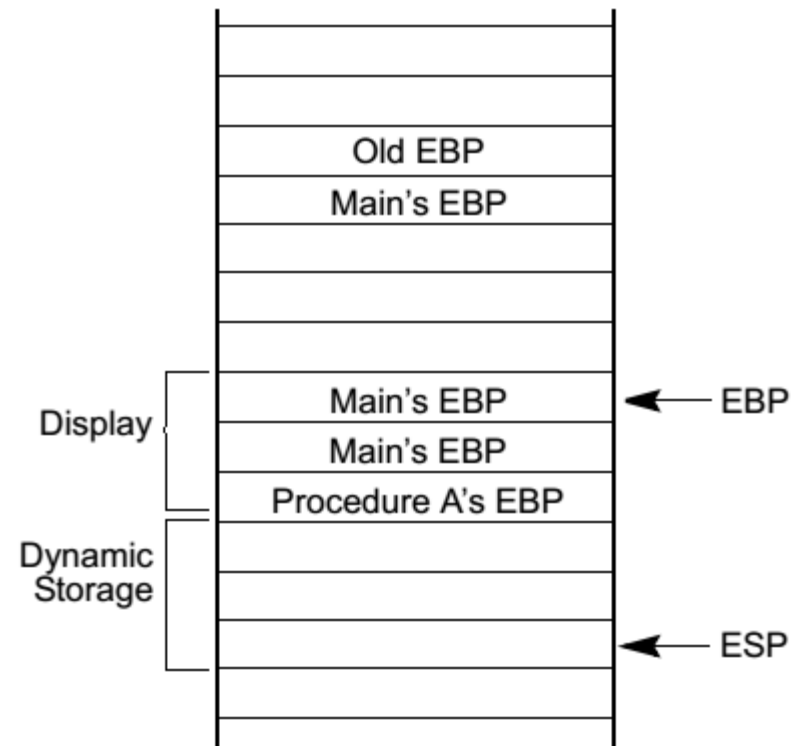
- The following pseudo code shows the definition of the ENTER instruction.

```
PUSH EBP;
FRAME_PTR := ESP;
IF LEVEL > 0
    THEN
        DO (LEVEL – 1) times
            EBP := EBP – 4;
            PUSH Pointer(EBP); (* doubleword pointed to by EBP *)
        OD;
    PUSH FRAME_PTR;
FI;
EBP := FRAME_PTR;
ESP := ESP – STORAGE;
```

- Stack frame for the following nested functions



Stack frame after entering the function MAIN using `ENTER 3,1`



Stack Frame After Entering function A using `ENTER 3,2`



# Assignment

6-6, 10-12, 24, 26-27, 47, 51-52, 54-56