CSE213

Microcontroller Programming

Program Control Instructions

Introduction

- This chapter explains the program control instructions, including the jumps, loops, calls, and returns.

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 conditional and unconditional loops to control the flow of a program.
- Use the call and return instructions to include procedures in the program structure.

Program Flow Control

- Controlling the program flow is very important, where your program can make decisions based on particular conditions.
- Remember the **if** statement in C programming language.
- In the Assembly language, flow control is handled by several **Jump** instructions including conditional and unconditional jumps.

Unconditional Jumps

- The basic instruction that transfers control to another point in the program is **JMP**:

  JMP label

- To declare a **label** in your program, just type its name and add "**:**" to the end.
- A label can be any character combination but it cannot start with a number. Examples:

  label1:

  label2:

  a:

More About Labels

- Label can be declared on a separate line or before any other instruction
- Examples:

```
x1:
MOV AX, 1

x2: MOV AX, 2
```

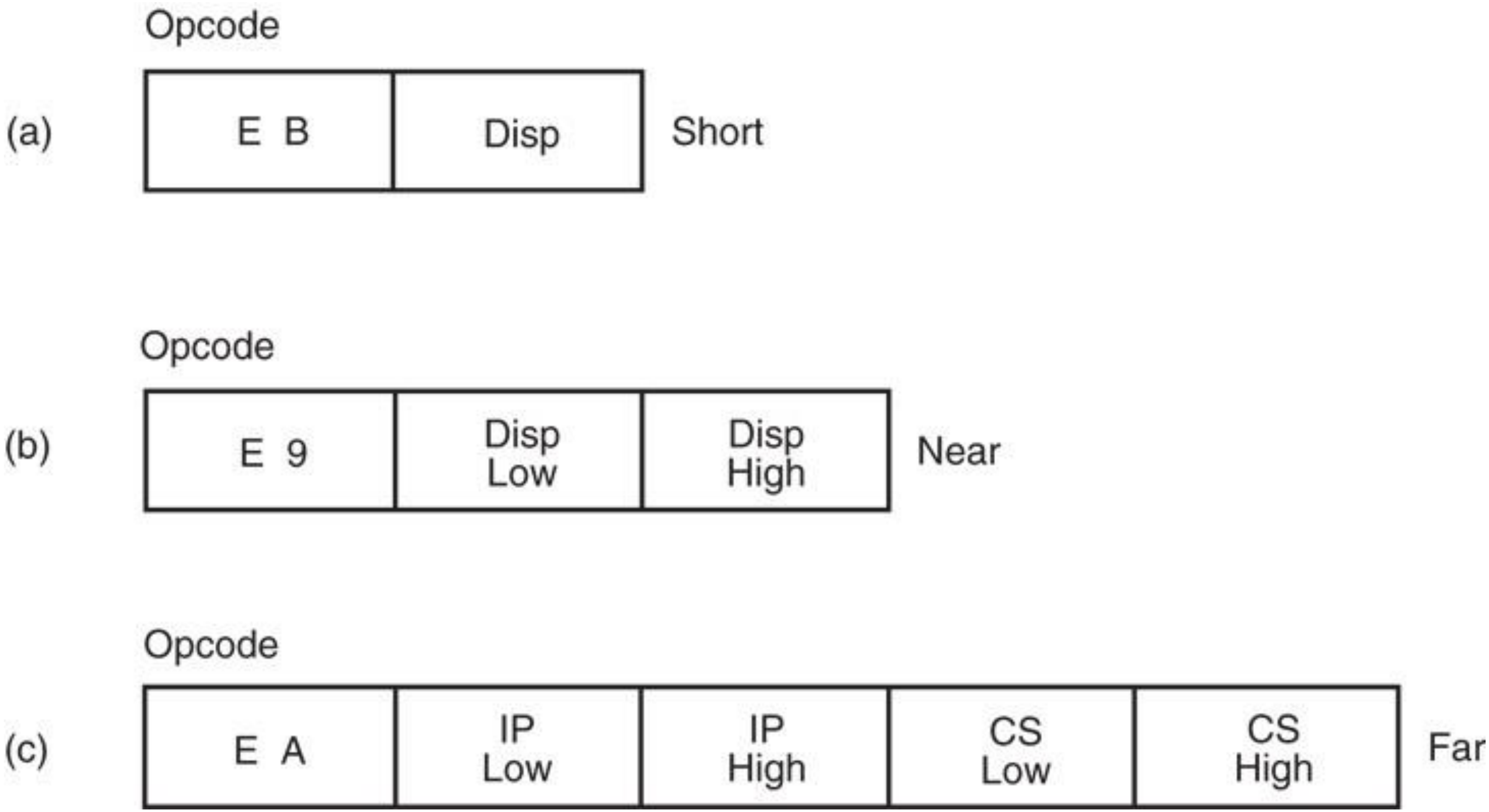
JMP Example

```
ORG 100H
MOV AX, 5      ; set AX to 5.
MOV BX, 2      ; set BX to 2.
JMP calc       ; go to 'calc'.
back:
JMP stop       ; go to 'stop'.
calc:
ADD AX, BX     ; add BX to AX.
JMP back       ; go 'back'.
stop:
RET            ; return to operating system.
```

JMP in Detail

- There are three types of the JMP instruction:
  1. **Short Jump**
  2. **Near Jump**
  3. **Far Jump**
- **Short** and **near** jumps are **intrasegment** jumps and **far** jump is **intersegment** jump.

Jump Types



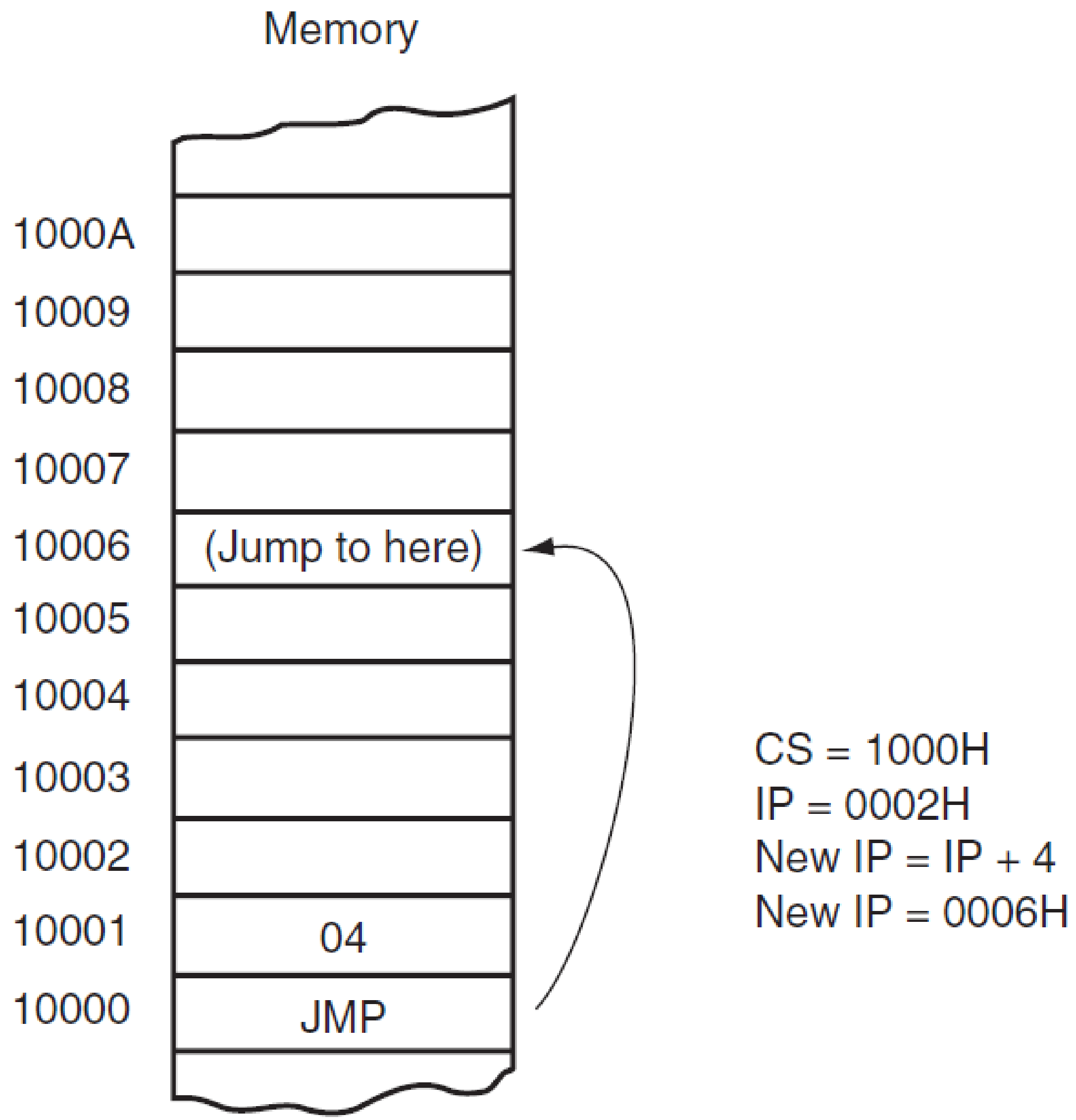
Opcode
(a)
E B
Disp
Short
Opcode
(b)
E 9
Disp Low
Disp High
Near
Opcode
(c)
E A
IP Low
IP High
CS Low
CS High
Far

Short Jump

- The jump address is not stored in the opcode.
- Instead, a **distance**, or **displacement**, follows the opcode.
- Therefore, short jumps are also called **relative** jumps.
- The displacement takes a value in the range between **-128** and **+127**.

Short Jump Example

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

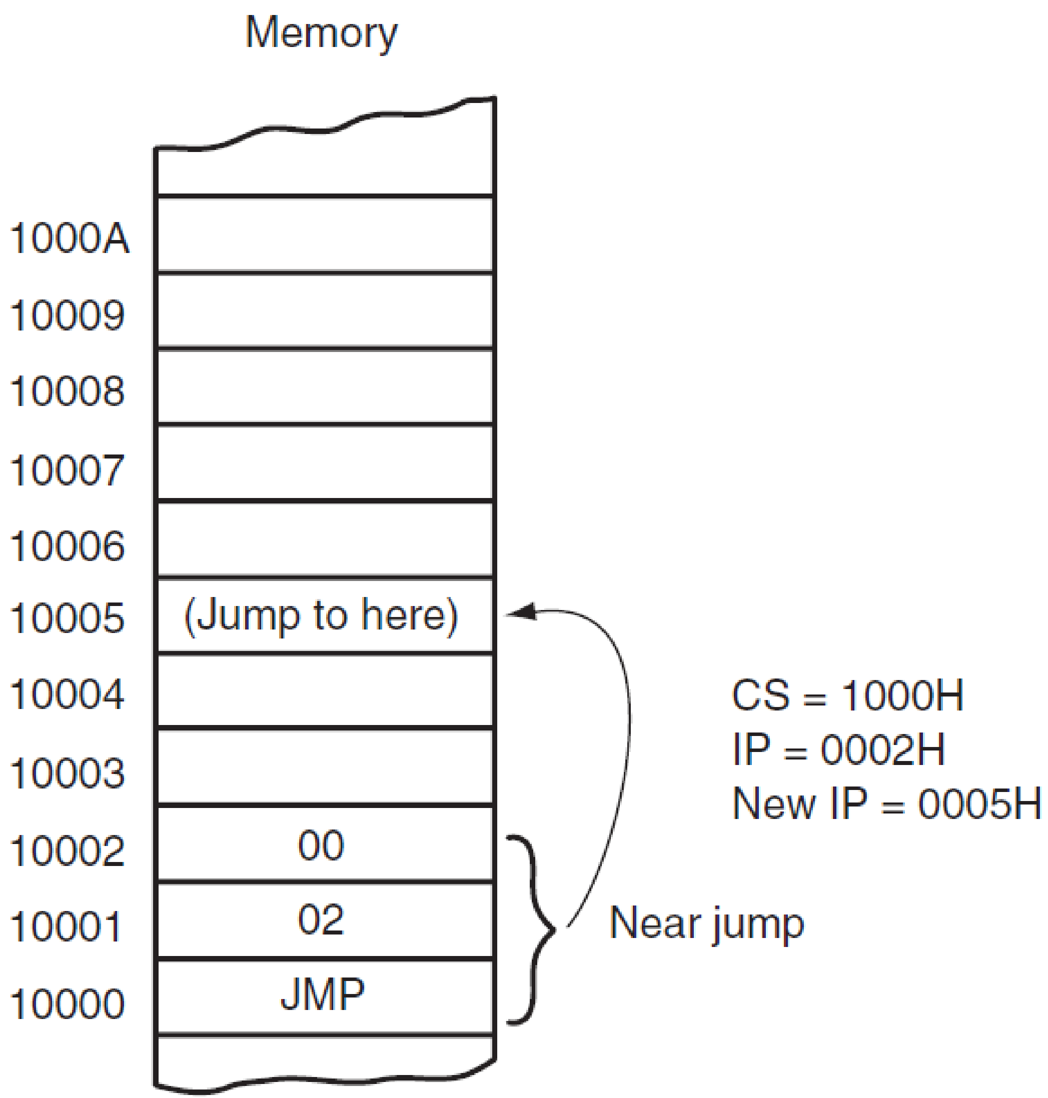


Near Jump

- The near jump is similar to the short jump, except that the distance is farther.
- A **near jump** passes control to an instruction in the current code segment located within **±32K bytes** from the near jump instruction.
- Near jump is also a **relative** jump.

Near Jump Example

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

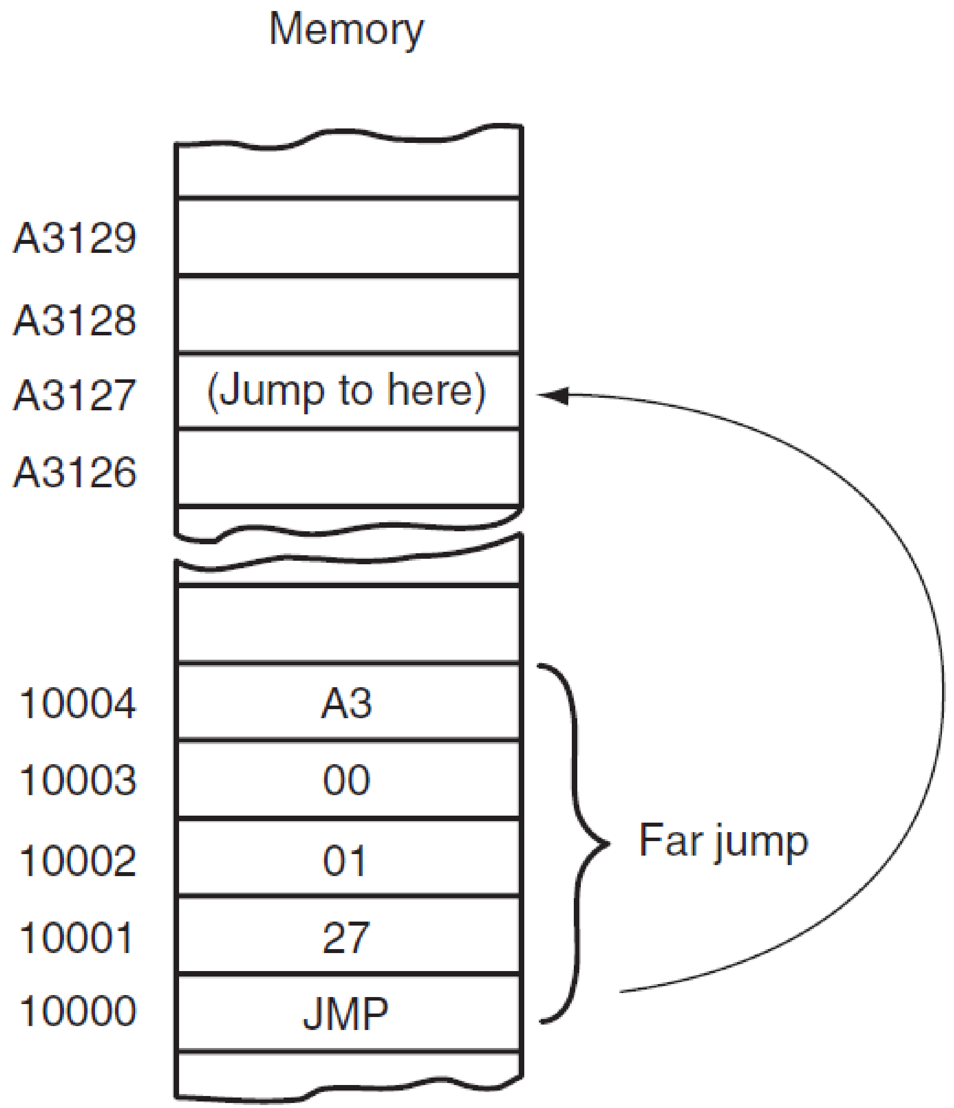


Far Jump

- A **far jump** instruction obtains a new segment and offset address to accomplish the jump.
- The far jump instruction sometimes appears with the **FAR PTR** directive.
- Another way to obtain a far jump is to define a label as a **far label**. A label is far only if it is external to the current code segment or procedure (i.e. the compiler sets it automatically).

Far Jump Example

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



Conditional Jumps

- Unlike **JMP** instruction that does an unconditional jump, there are instructions that perform conditional jumps (jump only when some conditions are true)
- Conditional jumps are always **short jumps** in 8086 - 80286.
- These instructions are divided in three groups:
  - First group just tests single flag
  - Second group compares numbers as signed
  - Third group compares numbers as unsigned.

General Syntax

- The general syntax for conditional jumps:

  <Conditional jump instruction> <label>

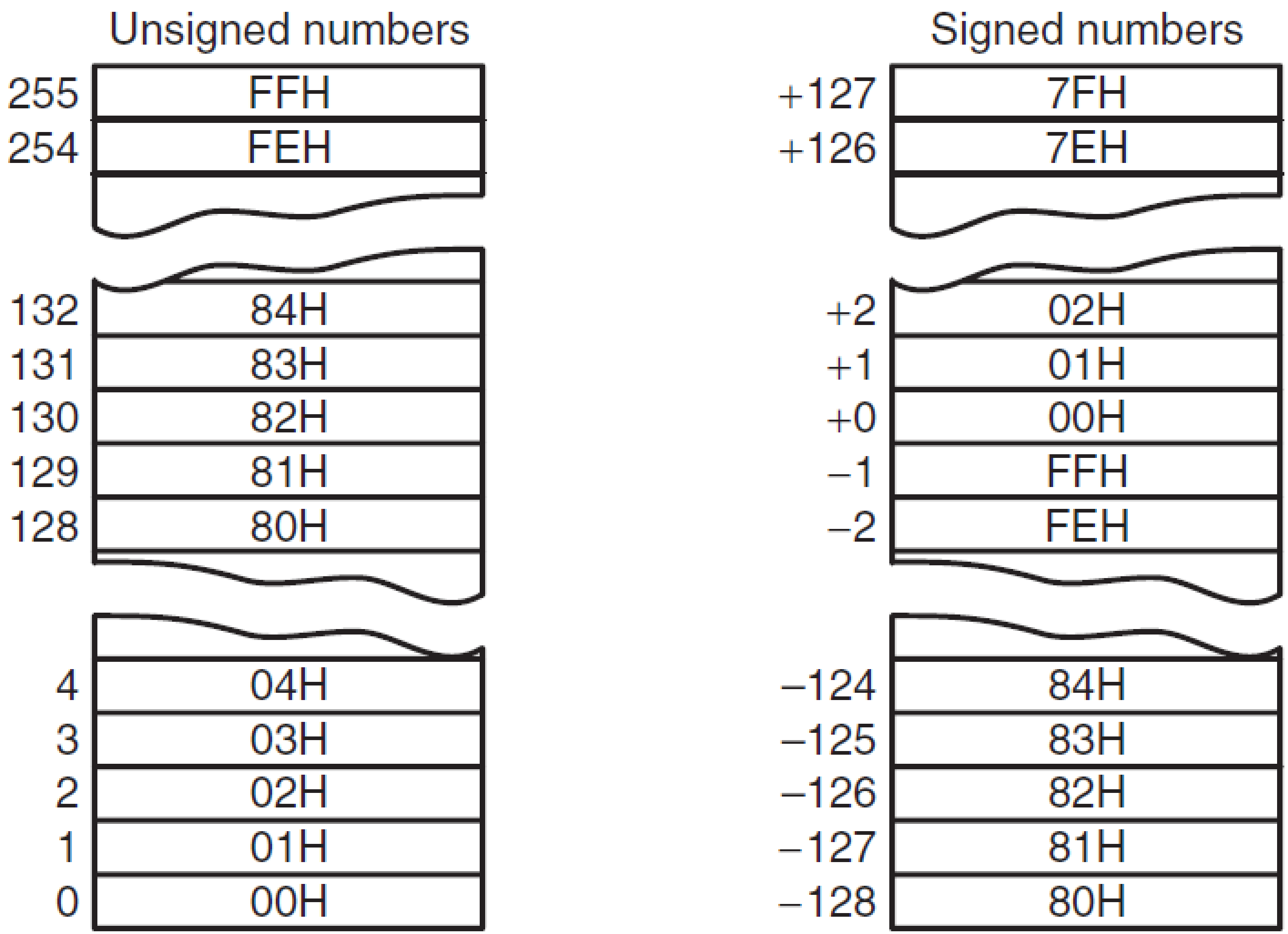
  Example: **JC label1**
- If the condition under the test is true, a branch to the label occurs.
- If the condition is false, the next sequential step in the program executes.

Jump Instructions That Test Single Flag

Instruction	Description	Condition	Opposite
JZ, JE	Jump if Zero (Equal)	Z = 1	JNZ, JNE
JC, JB, JNAE	Jump if Carry (Below, Not Above Equal)	C = 1	JNC, JNB, JAE
JS	Jump if Sign	S = 1	JNS
JO	Jump if Overflow	O = 1	JNO
JPE, JP	Jump if Parity Even	P = 1	JPO, JNP
JNZ, JNE	Jump if Not Zero (Not Equal)	Z = 0	JZ, JE
JNC, JNB, JAE	Jump if Not Carry (Not Below, Above Equal)	C = 0	JC, JB, JNAE
JNS	Jump if Not Sign	S = 0	JS
JNO	Jump if Not Overflow	O = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity)	P = 0	JPE, JP

* Different names are used to make programs easier to understand.

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



Jump Instructions for Signed Numbers

Instruction	Description	Condition	Opposite
JE, JZ	Jump if Equal (=) Jump if Zero	Z = 1	JNE, JNZ
JNE, JNZ	Jump if Not Equal (≠) Jump if Not Zero	Z = 0	JE, JZ
JG, JNLE	Jump if Greater (>) Jump if Not Less or Equal (not <=)	Z = 0 and S = O	JNG, JLE
JL, JNGE	Jump if Less (<) Jump if Not Greater or Equal	S ≠ O	JNL, JGE
JGE, JNL	Jump if Greater or Equal (>=) Jump if Not Less	S = O	JNGE, JL
JLE, JNG	Jump if Less or Equal (<=) Jump if Not Greater	Z = 1 or S ≠ O	JNLE, JG

Jump Instructions for Unsigned Numbers

Instruction	Description	Condition	Opposite
JE, JZ	Jump if Equal (=) Jump if Zero	Z = 1	JNE, JNZ
JNE, JNZ	Jump if Not Equal (≠) Jump if Not Zero	Z = 0	JE, JZ
JA, JNBE	Jump if Above (>) Jump if Not Below or Equal	C = 0 and Z = O	JNA, JBE
JBE, JNA	Jump if Below or Equal (<=) Jump if Not Above	C = 1 or Z = 1	JNBE, JA
JB, JNAE, JC	Jump if Below (<) Jump if Not Above or Equal Jump if Carry	C = 1	JNB, JAE, JNC
JAE, JNB, JNC	Jump if Above or Equal (>=) Jump if Not Below Jump if Not Carry	C = 0	JB, JNAE, JC

Signed or Unsigned?

- The numbers in a comparison may be signed or unsigned, according to your selection.
- When signed numbers are compared, use the instructions with the terms **“greater than”**, **“less than”**, etc.
- When unsigned numbers are compared, use the instructions with the terms **“above”** and **“below”**.

Conditional Jump Example

```
ORG 100H
MOV AL, 25 ; set AL to 25.
MOV BL, 10 ; set BL to 10.
CMP AL, BL ; compare AL - BL.
JE equal   ; jump if AL = BL (Z = 1).
MOV CX, 1  ; if it gets here, then AL <> BL
JMP stop   ; so set CX, and jump to stop.
equal:     ; if gets here,
MOV CX, 0  ; then AL = BL, so clear CX.
stop:
RET        ; gets here no matter what.
```

Limitation

- All conditional jumps have one big limitation, unlike **JMP** instruction they can only jump **127** bytes forward and **128** bytes backward.

- We can easily avoid this limitation using a cute trick:
  - Get an opposite conditional jump instruction from the table above, make it jump to *label_x*.
  - Use **JMP** instruction to jump to desired location.
  - Define *label_x:* just after the **JMP** instruction.
  - *label_x:* - can be any valid label name, but there must not be two or more labels with the same name.

Example

```
include "emu8086.inc"

org     100h

mov     al, 5
mov     bl, 5

cmp     al, bl          ; compare al - bl.


; je equal              ; there is only 1 byte


jne     not_equal       ; jump if al <> bl (zf = 0).
jmp     equal
not_equal:


add     bl, al
sub     al, 10
xor     al, bl

jmp skip_data
db 256 dup(0)           ; 256 bytes
skip_data:


putc    'n'             ; if it gets here, then al <> bl,
jmp     stop            ; so print 'n', and jump to stop.

equal:                  ; if gets here,
putc    'y'             ; then al = bl, so print 'y'.

stop:

ret
```

LOOP

- LOOP is similar to JMP.
- It is a combination of a decrement CX and the JNZ conditional jump.
- LOOP decrements CX.
  - if CX != 0, it jumps to the address indicated by the label
  - If CX becomes 0, the next sequential instruction executes

Conditional LOOPs

- LOOP instruction also has conditional forms:
  LOOPE and LOOPNE
- LOOPE (**loop while equal**) instruction jumps if CX != 0 while an equal condition exists.
  - will exit loop if the condition is not equal or the CX register decrements to 0
- LOOPNE (**loop while not equal**) jumps if CX != 0 while a not-equal condition exists.
  - will exit loop if the condition is equal or the CX register decrements to 0

Loops

Instruction	Operation and Jump Condition	Opposite Direction
LOOP	Decrease CX, jump to label if CX not zero	DEC CX and JCXZ
LOOPE	Decrease CX, jump to label if CX not zero and equal (Z = 1)	LOOPNE
LOOPNE	Decrease CX, jump to label if CX not zero and not equal (Z = 0)	LOOPE
LOOPNZ	Decrease CX, jump to label if CX not zero and Z = 0	LOOPZ
LOOPZ	Decrease CX, jump to label if CX not zero and Z = 1	LOOPNZ
JCXZ	Jump to label if CX is zero	OR CX, CX and JNZ

Loop Example: C Code

- Write a program that sums the contents of two arrays and stores the result over the second array.

```
short arr1[100];
short arr2[100];
int count = 100;
int idx = 0;
while (count > 0)
{
    arr2[idx] = arr1[idx] + arr2[idx];
    idx++;
    count--;
}
```

Loop Example: Assembly Code

```
org 100h
mov cx, 100 ; Number of elements in the blocks
mov bx, 0   ; Start index
L1:
mov ax, BLOCK1[bx] ; read next number from BLOCK1
add ax, BLOCK2[bx] ; add next number from BLOCK2
mov BLOCK2[bx], ax ; store the result
add bx, 2          ; skip to next element
loop L1
Ret
BLOCK1 DW 100 DUP (1)
BLOCK2 DW 100 DUP (2)
```

Nested Loops

- All loop instructions use **CX** register to count steps, as you know CX register has 16 bits and the maximum value it can hold is 65535 or FFFF

- However with some agility it is possible to put one loop into another, and receive a nice value of 65535 x 65535 x 65535 ....till infinity.... or the end of ram or stack memory.

- It is possible store original value of CX register using **PUSH CX** instruction and return it to original when the internal loop ends with **POP CX**

Nested Loop Example

```
org 100h

mov bx, 0  ; total step counter.

mov cx, 5
k1: add bx, 1
    mov al, '1'
    mov ah, 0eh
    int 10h
    push cx
    mov cx, 5
      k2: add bx, 1
      mov al, '2'
      mov ah, 0eh
      int 10h
      push cx
         mov cx, 5
         k3: add bx, 1
         mov al, '3'
         mov ah, 0eh
         int 10h
         loop k3    ; internal in internal loop.
      pop  cx
      loop  k2      ; internal loop.
    pop cx
loop k1             ; external loop.

ret
```

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

Procedure Syntax

- name PROC

  ; here goes the code

  ; of the procedure ...

  RET

  name ENDP

- name - is the procedure name
- the same name should be in the top and the bottom, this is used to check correct closing of procedures.

Procedure Example

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

Calling a Procedure

- **CALL** instruction is used to call a procedure.
- Example:

```
        ORG 100H
        CALL m1
        MOV AX, 2
        RET ; return to operating system.
  m1        PROC
        MOV BX, 5
        RET ; return to caller.
  m1        ENDP
        END
```

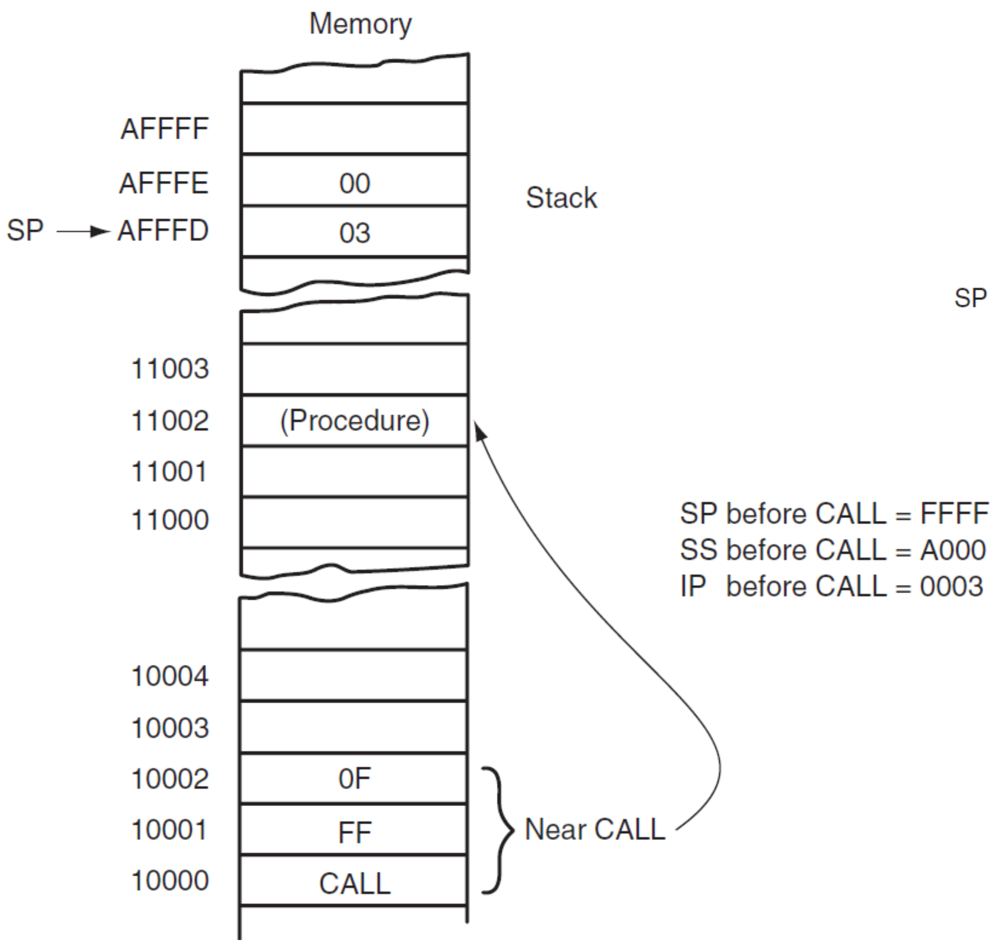
CALL

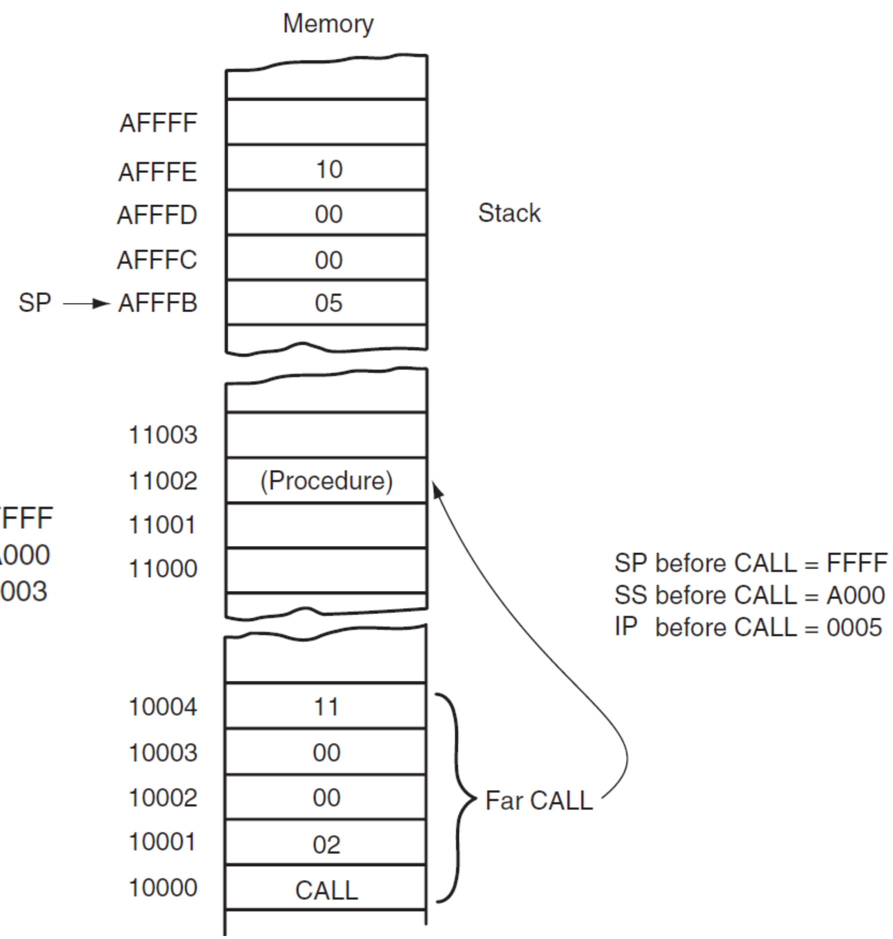
- Transfers the flow of the program to the procedure.
- CALL instruction differs from the jump instruction because a CALL saves a return address on the stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes at the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

CALL *(cont.)*

- The CALL construction is a combination of a PUSH and a JMP instruction.
- When CALL executes, it pushes the return address on the stack and then jumps to the procedure.
- A near CALL places the contents of IP on the stack, and a far CALL places both IP and CS on the stack.

Near and Far CALL Examples





RET

- The RET instruction returns from a procedure by removing the return address from the stack and placing it into IP (near return), or IP and CS (far return).

Passing Parameters to a Procedure

- There are several ways to pass parameters to a procedure
- The easiest way is by using registers
  - Put parameters to the registers and call the procedure
  - The procedure should be designed so that it uses those registers
- Another way is by using stack
  - Push the parameters to the stack and call the procedure
  - The procedure should be designed so that it receives parameters from the stack
- In either way, the programmer should know the structure of the procedure

Example

```
        ORG 100h
        MOV AL, 1
        MOV BL, 2
        CALL m2
        CALL m2
        CALL m2
        CALL m2
        RET ; return to operating system.

m2      PROC
        MUL BL ; AX = AL * BL.
        RET ; return to caller.
m2      ENDP
        END
```

About the Example

- The program calculates 2^4
- The procedure takes its parameters from **AL** and **BL**; multiplies them and stores the result into **AX**
- The C equivalent can be thought as the following:

```
short m2(char al, char bl)
{
   short ax = al * bl;
   return ax;
}
```

The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, Pentium Pro Processor, Pentium II, Pentium, 4, and Core2 with 64-bit Extensions Architecture, Programming, and Interfacing, Eighth Edition
Barry B. Brey