# CHAPTER 7

# LOGIC , SHIFT & ROTATE INSTRUCTIONS

# AND, OR, and XOR Instructions

AND destination, source

OR destination, source

XOR destination, source

**Effect on flags:**

SF, ZF, PF reflect the result

AF is undefined

CF, OF = 0

# Rules

- The destination must be a register or memory location.
- The source may be a constant, register, or memory location.
- The memory-to-memory operations are not allowed.

# Use of AND, OR, and XOR Instructions

- Selectively modify the bits in the destination
- Construct a source bit pattern, **mask**
- Only desired destination bits are modified

# AND

- can be used to **clear** specific destination bits while preserving the others.
- A 0 mask bit clears the corresponding destination bit.
- A 1 mask bit preserves the corresponding destination bit.

# OR

- can be used to **set** specific destination bits while preserving the others.
- A 1 mask bit sets the corresponding destination bit.
- A 0 mask bit preserves the corresponding destination bit.

# XOR

- can be used to **complement** specific destination bits while preserving the others.
- A 1 mask bit complements the corresponding destination bit.
- A 0 mask bit preserves the corresponding destination bit.

# Clear the sign bit of AL while leaving the other bits unchanged.

Use AND with 0111 1111b = 7Fh as the mask. Thus,

**AND AL, 7Fh**

# Set msb and lsb of AL while preserving the other bits.

Use OR with 1000 0001b = 81h as the mask. Thus,

**OR AL, 81h**

# Change the sign bit of DX.

Use XOR with a mask of 8000h. Thus,

```
XOR DX, 8000h
```

# Converting an ASCII Digit to a Number

If the “5” key is pressed, AL gets 35h instead of 5. To get 5 in AL, we could do this:

**SUB AL, 30h**

Another method is to use AND to clear the high nibble (high four bits) of AL:

**AND AL, 0FH**

Because the codes “0” to “9” are 30h to 39h.

# Converting a Lowercase Letter to Upper Case

The ASCII codes of “a” to “z” range from 61h to 7Ah; the codes “A” to “Z” go from 41h to 5Ah.

Thus for example, if DL contains the code of a lowercase letter, we could convert to upper case by executing

**SUB DL, 20h**

# Converting a Lowercase Letter to Upper Case

| Character | Code | Character | Code |
|---|---|---|---|
| a | 01**1**0 0001 | A | 01**0**0 0001 |
| b | 01**1**0 0010 | B | 01**0**0 0010 |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| z | 01**1**1 1010 | Z | 01**0**1 1010 |

# Converting a Lowercase Letter to Upper Case

We need only clear bit 5 by using AND with 1101 1111b or 0DFh.

So if the lowercase character to be converted is in DL, execute

**AND DL, 0DFh**

# Clearing a Register

To clear AX, we could execute

**MOV AX, 0**

**SUB AX, AX**

Using the fact that **1 XOR 1 = 0** and **0 XOR 0 = 0**, a third way is

**XOR AX, AX**

# Testing a Register for Zero

**OR CX, CX**

Because **1 OR 1 = 1** and **0 OR 0 = 0**, it leaves the content of CX unchanged; however, it affects ZF and SF, and in particular if CX contains 0 then ZF = 1.

So it can be used as an alternative to

**CMP CX, 0**

# NOT Instruction

**NOT destination**

# Complement the bits in AX.

```
NOT        AX
```

# TEST Instruction

TEST destination, source

**Effect on flags:**

SF, ZF, PF reflect the result

AF is undefined

CF, OF = 0

# TEST Instruction

- performs an AND operation but does not change the destination content.
- The purpose of TEST is to set the status flags.

# Examining Bits

TEST destination, mask

Because **1 AND b = b** and **0 AND b = 0**, the result will have 1’s in the tested bit positions if and only if the destination has 1’s in these positions; it will have 0’s elsewhere.

If destination has 0’s in all the tested position, the result will be 0 and so ZF = 1.

# Jump to label BELOW if AL contains an even number

Even numbers have a 0 in bit 0. Thus, the mask is 0000 0001b = 1.

```
TEST    AL, 1       ; is AL even?
JZ      BELOW       ; yes, go to BELOW
```

# Shift and Rotate Instructions

For a single shift or rotate, the form is

**Opcode destination, 1**

For a shift or rotate of N positions, the form is

**Opcode destination, CL**

Where CL contains N.

In both cases, destination is an 8- or 16-bit register or memory location.

# Shift Left

**SHL** shifts the bits in the destination to the left.

**Effect on flags:**

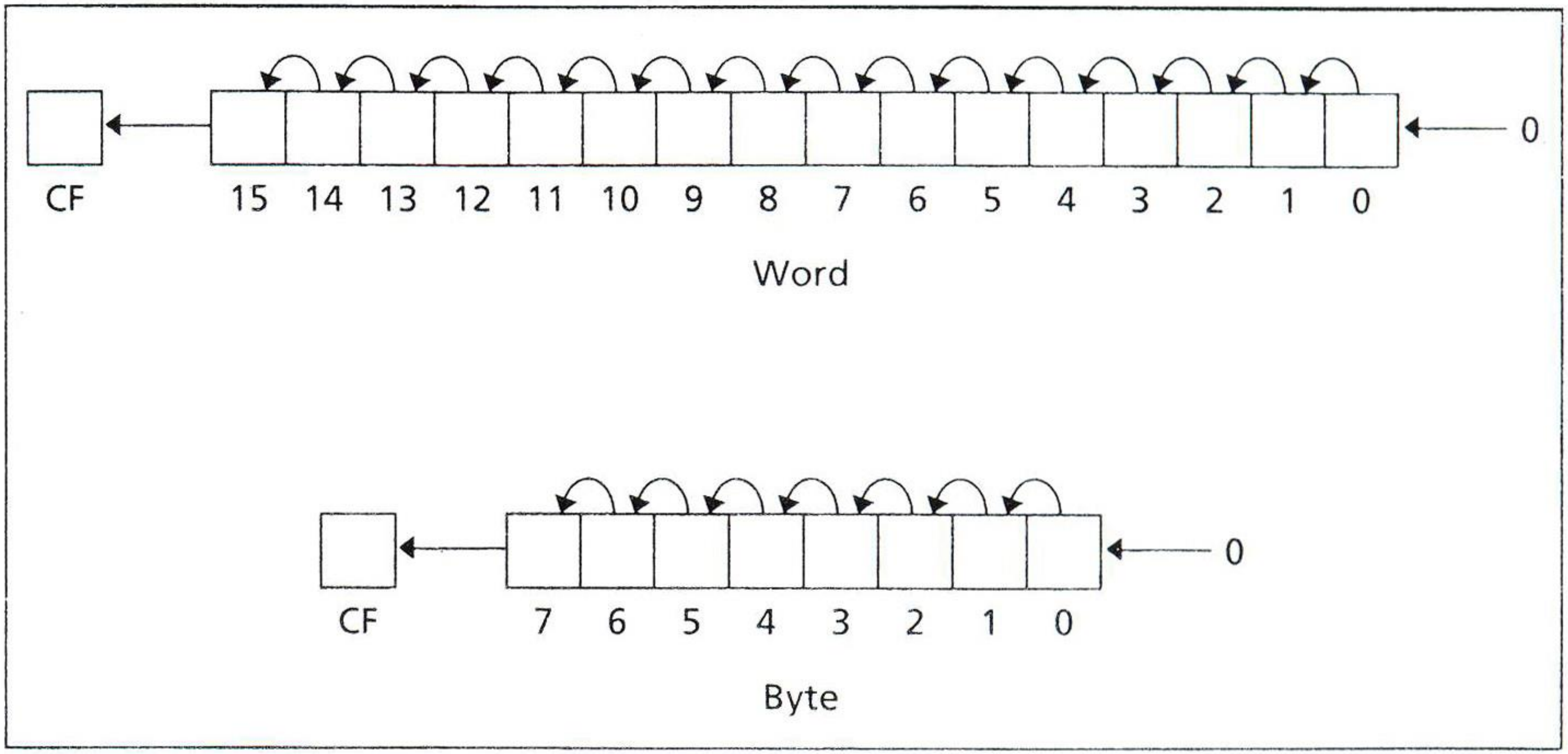
SF, ZF, PF reflect the result

AF is undefined

CF = last bit shifted out

OF = 1 if result changes sign on last shift

# SHL and SAL



# Multiplication by Left Shift

A left shift on a binary number multiplies it by 2.

Suppose that AL contains 5 = 00000101b. A left shift gives 00001010b = 10d, thus doubling its value.

If AX is FFFFh (–1), then shifting three times will yield AX = FFF8h (–8).

# Shift Arithmetic Left

**SAL** is often used in instances where numeric multiplication is intended.

Both instructions generate the same machine code.

# Overflow

The overflow flags are not reliable indicators for a multiple left shift because it is really a series of single shifts, and CF, OF only reflect the result of the last shift.

# Overflow

If BL contains 80h, CL contains 2and we execute SHL BL, CL, then CF = OF = 0 even though both signed and unsigned overflow occur.

Write some code to multiply the value of AX by 8. Assume that overflow will not occur.

```
MOV     CL, 3       ; number of shifts to do
SAL     AX, CL      ; multiply by 8
```

# Shift Right

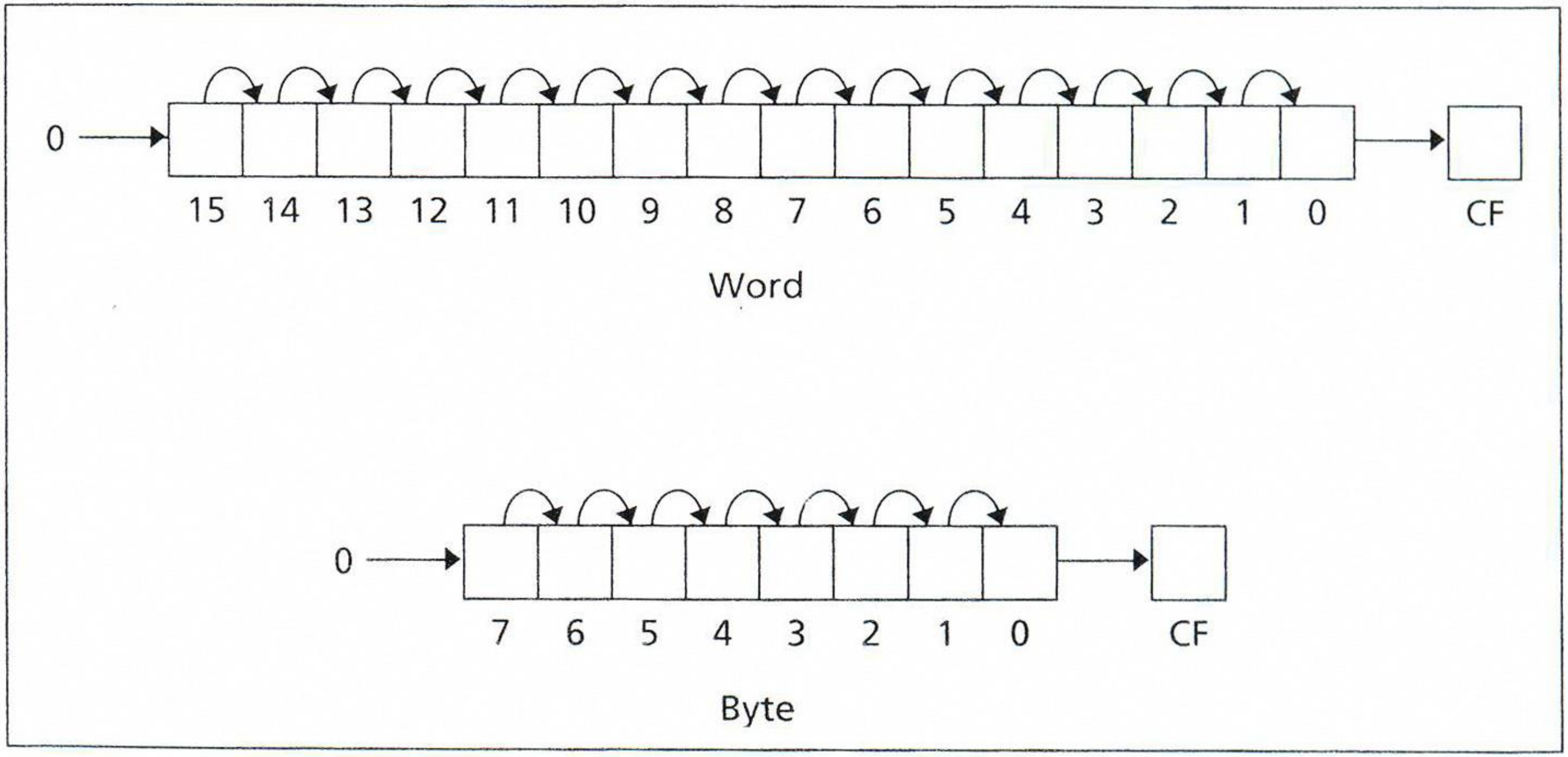
**SHR** performs right shifts on the destination operand.

A 0 is shifted into the msb position, and the rightmost bit is shifted into CF.

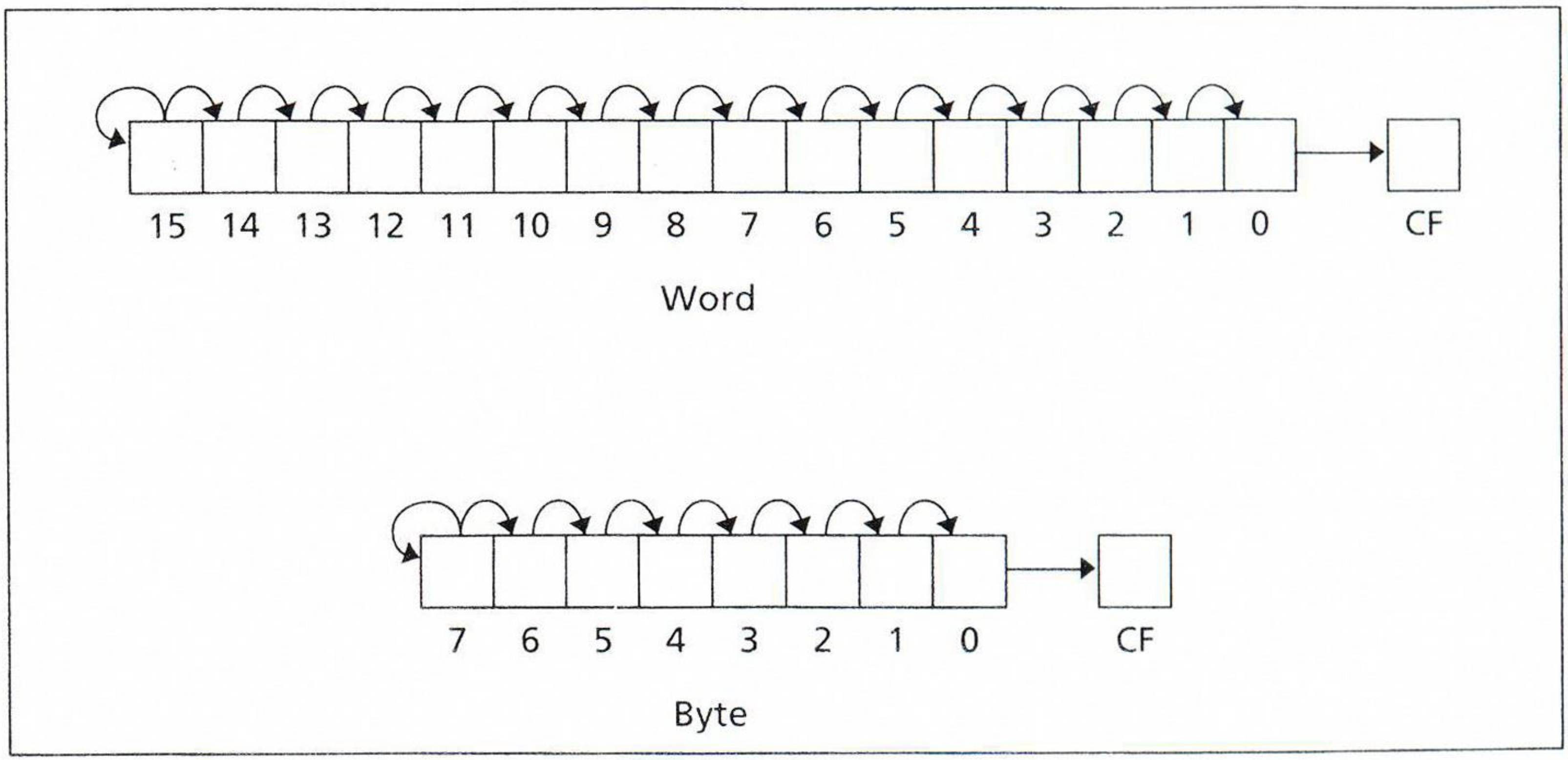
# Shift Arithmetic Right

**SAR** operates like SHR, with one difference: the msb retains its original value.

# SHR



# SAR



# Division by Right Shift

For even numbers, a right shift divides the destination's value by 2.

For odd numbers, a right shift halves the destination's value and round down to the nearest integer.

If BL contains 00000101b = 5, then after a right shift BL will contain 00000010b = 2.

# Signed and Unsigned Division

If an unsigned interpretation is being given, SHR should be used.

For a signed interpretation, SAR must be used, because it preserves the sign.

# Rotate Left

**ROL** shifts bits to the left.

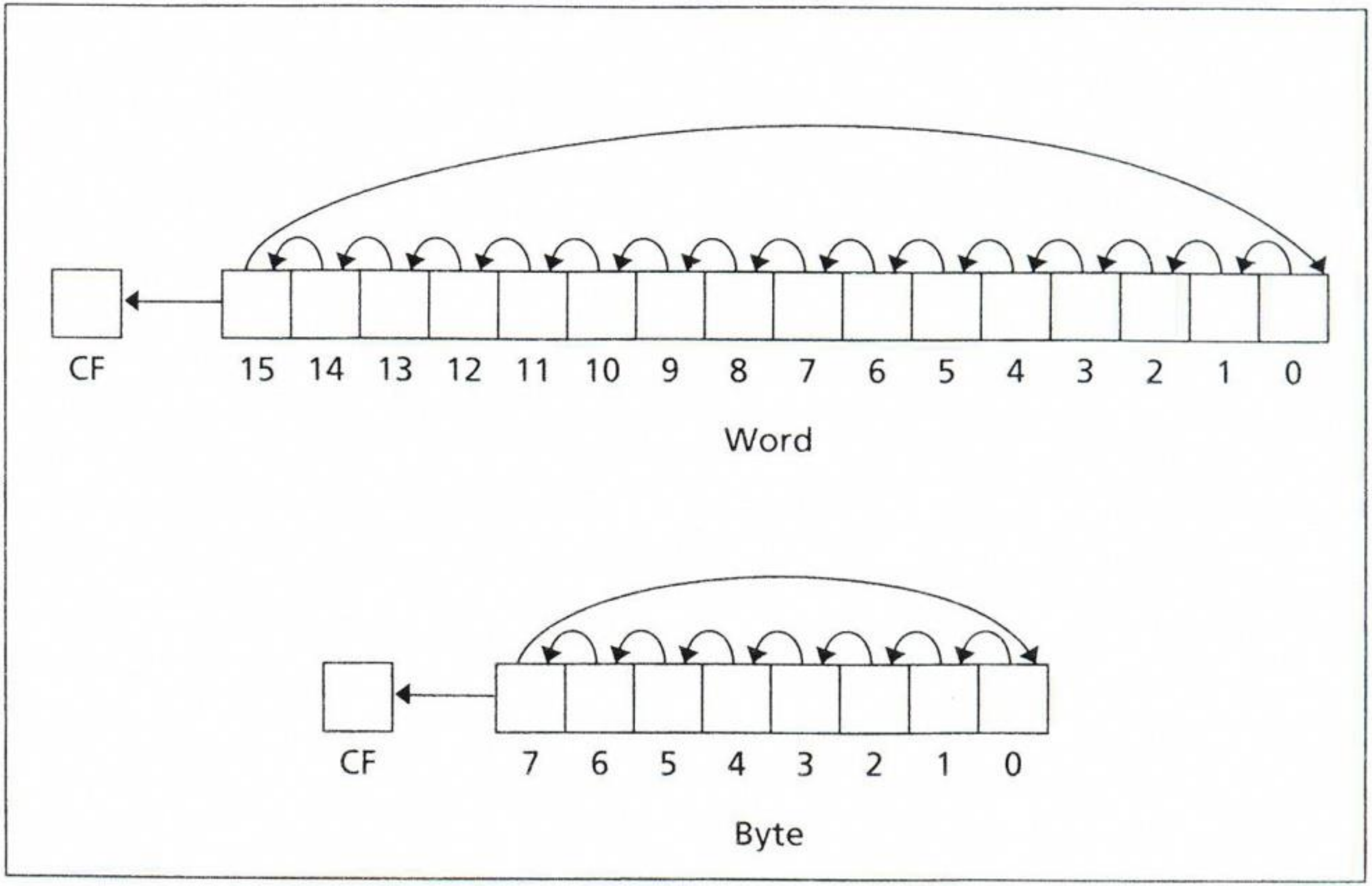
The msb is shifted into the rightmost bit.

The CF also gets the bit shifted out of the msb.

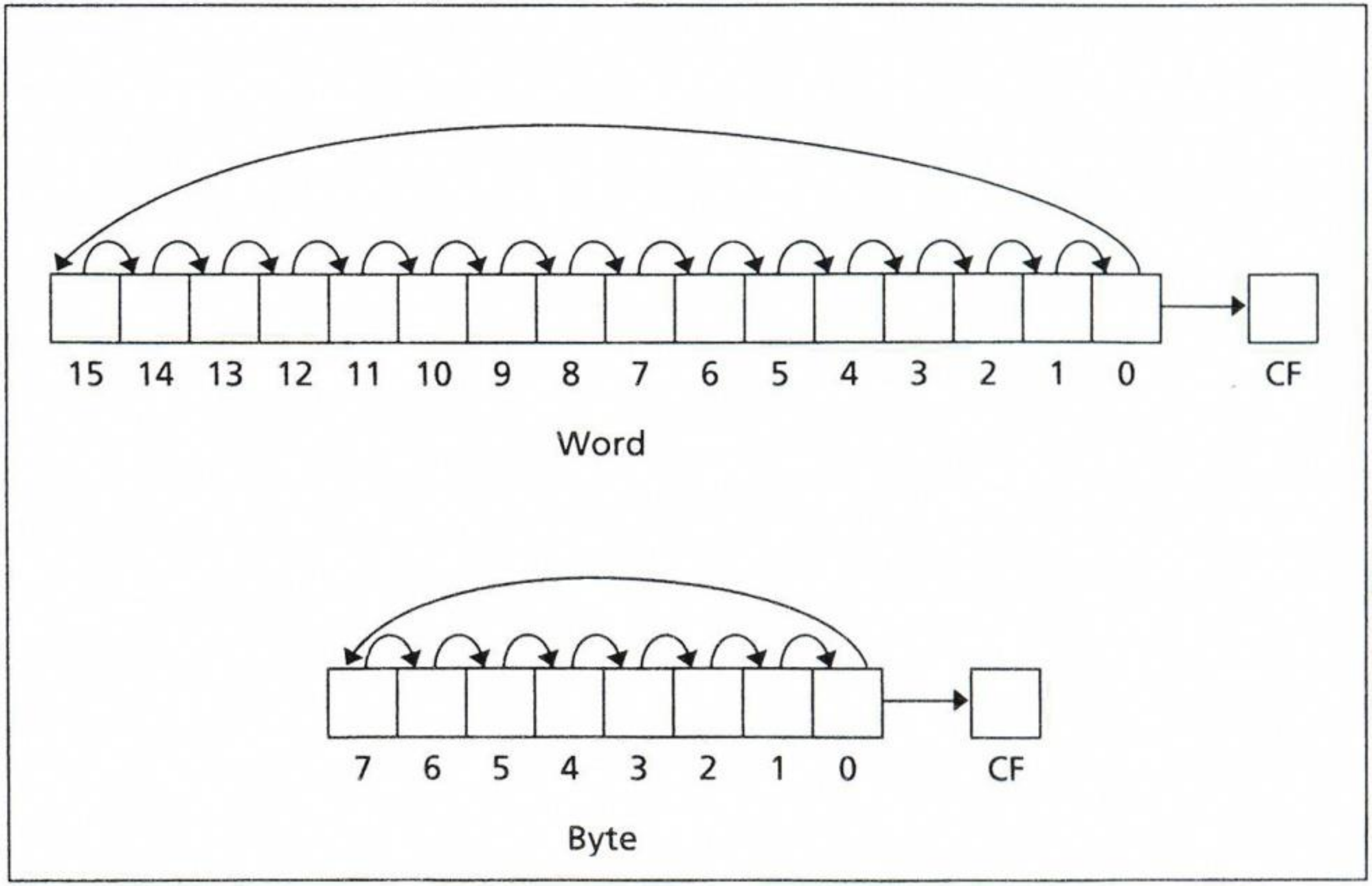
# Rotate Right

**ROR** works just like ROL, except that the bits are rotated to the right.

# ROL



# ROR



Use ROL to count the number of 1 bits in BX, without changing BX. Put the answer in AX.

```
        XOR   AX, AX      ; AX counts bits
        MOV   CX, 16      ; loop counter
TOP:
        ROL   BX, 1       ; CF = bit rotated out
        JNC   NEXT        ; 0 bit
        INC   AX          ; 1 bit, increment total
NEXT:
        LOOP  TOP         ; loop until done
```

Use ROL to count the number of 1 bits in BX, without changing BX. Put the answer in AX.

In this example, we used **JNC** (jump if no carry), which causes a jump if CF = 0.

# Rotate Carry Left

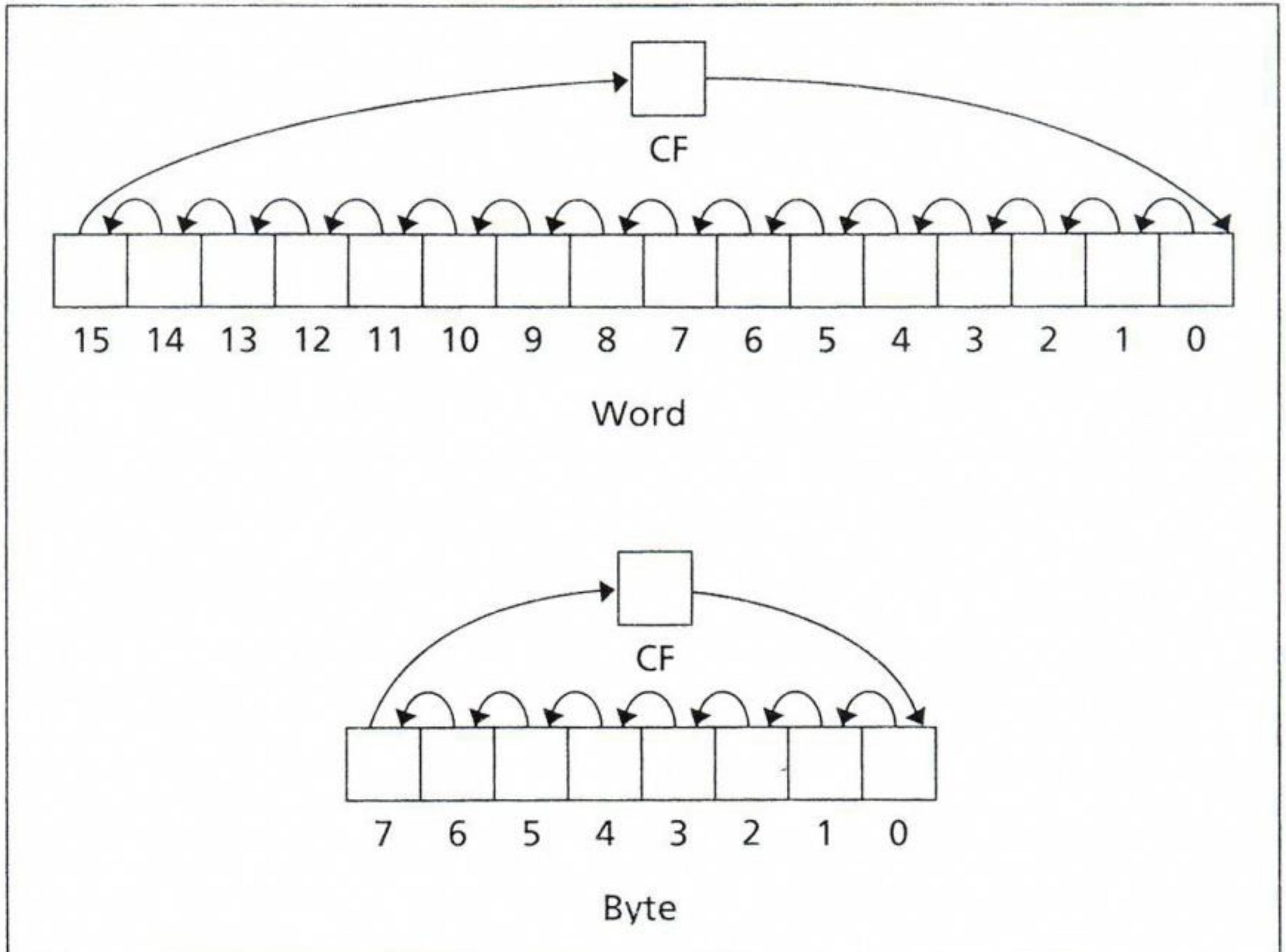
**RCL** shifts bits of the destination to the left.

The msb is shifted into the CF, and the previous value of CF is shifted into the rightmost bit.

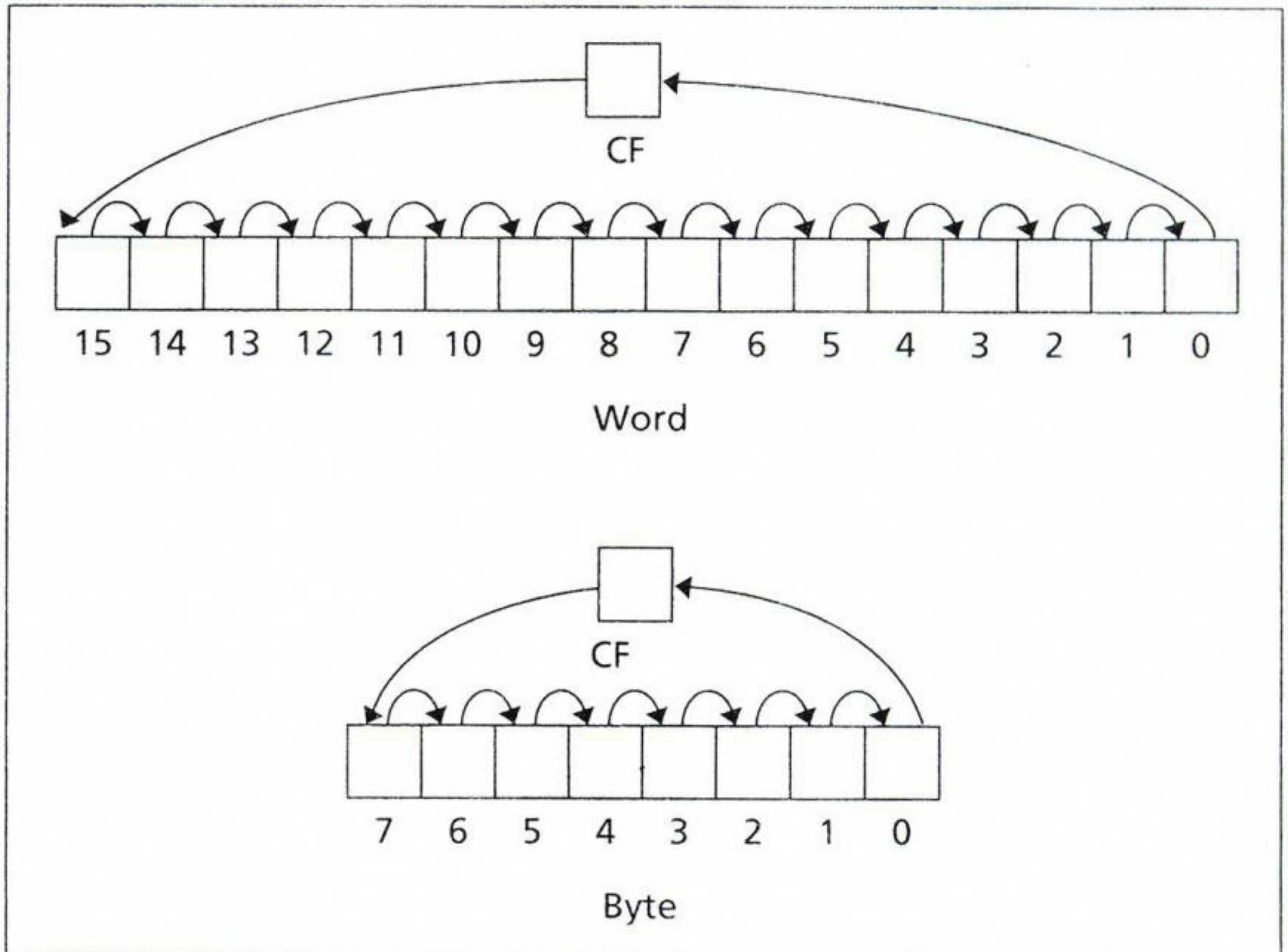
# Rotate Carry Right

**RCR** works just like RCL, except that the bits are rotated to the right.

# RCL



# RCR



# An application: Reversing a Bit Pattern

```
        MOV CX, 8          ; number of operations to do
REVERSE:
        SHL   AL, 1        ; get a bit into CF
        RCR   BL, 1        ; rotate it to BL
        LOOP REVERSE       ; loop until done
        MOV AL, BL         ; AL gets reversed pattern
```