

EE-2003

Computer Organization & Assembly Language




CHAPTER No: 7

INTEGER ARITHMETIC

OUTLINE



- **Shift and Rotate Instructions**
- **Multiplication and Division Instructions**

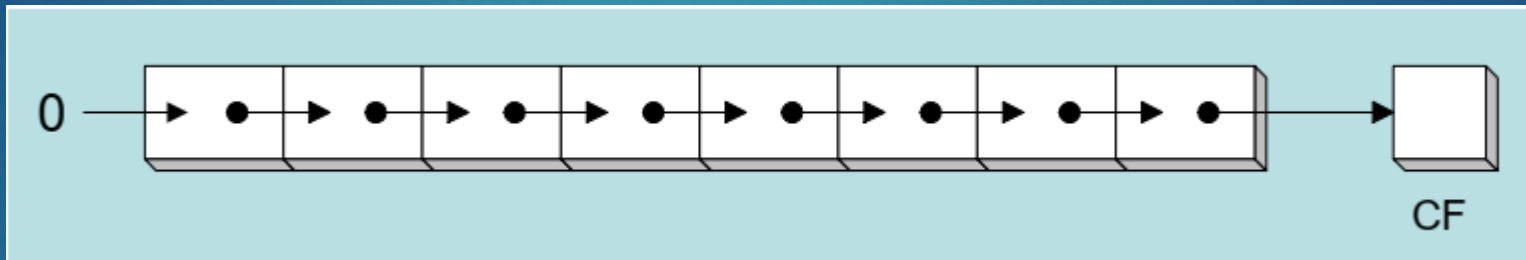
- 
- Bit manipulation is an intrinsic part of computer graphics, data encryption, and hardware manipulation.
 - Bit shifting means to move bits right and left inside an operand.
 - Shift and Rotate instructions affect the **overflow** and **carry** flags.

SHIFT AND ROTATE INSTRUCTIONS

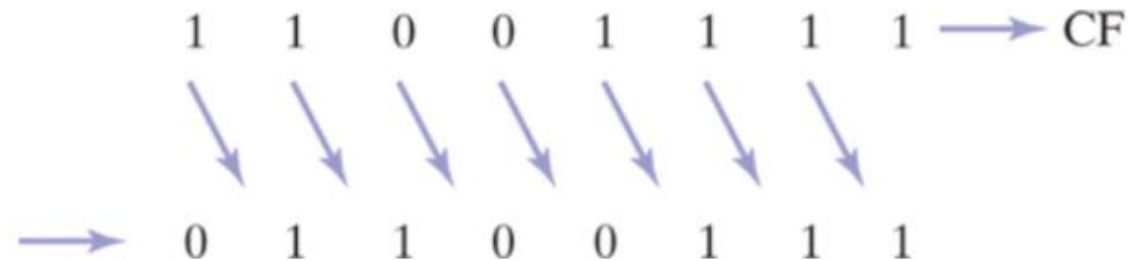
- ▶ Logical vs Arithmetic Shifts
- ▶ SHL and SHR Instruction
- ▶ SAL and SAR Instructions
- ▶ ROL and ROR Instruction
- ▶ RCL and RCR Instructions
- ▶ SHLD/SHRD Instructions

LOGICAL VS ARITHMETIC SHIFTS

- ▶ A logical shift fills the newly created bit position with zero.
- ▶ each bit is moved to the next lowest bit position, and the newly created bit position is filled with zero.

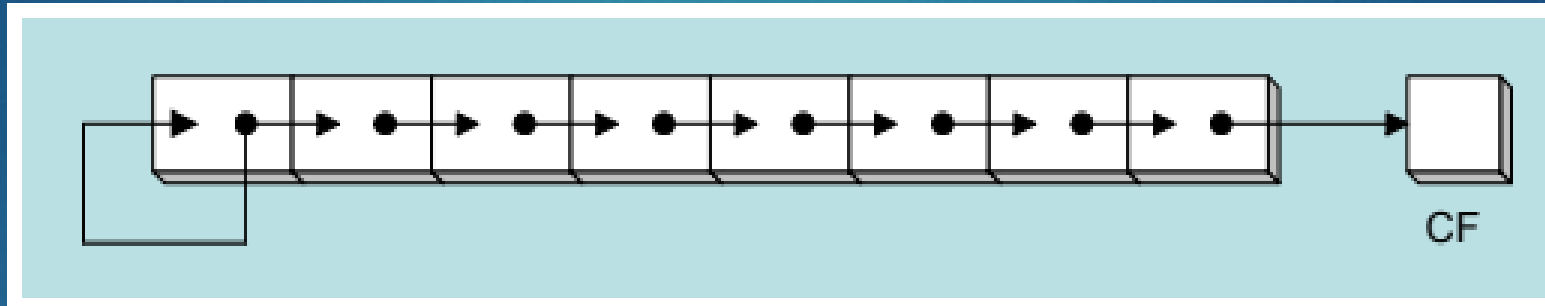


The following illustration shows a single logical right shift on the binary value 11001111, producing 01100111:

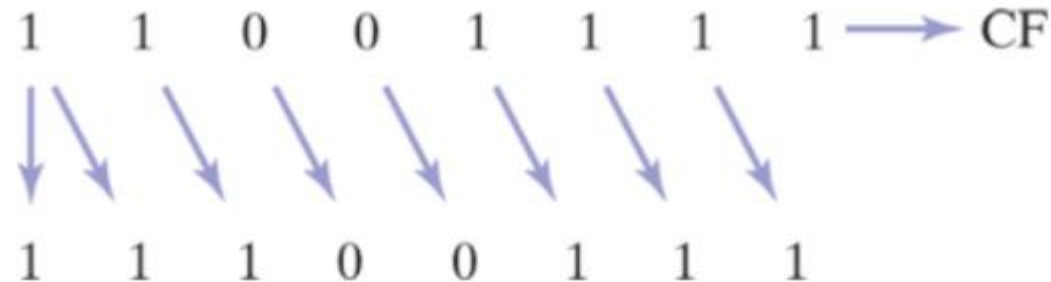


LOGICAL VS ARITHMETIC SHIFTS

- An arithmetic shift fills the newly created bit position with a copy of number's sign bit.



Binary 11001111, for example, has a 1 in the sign bit. When shifted arithmetically 1 bit to the right, it becomes 11100111:

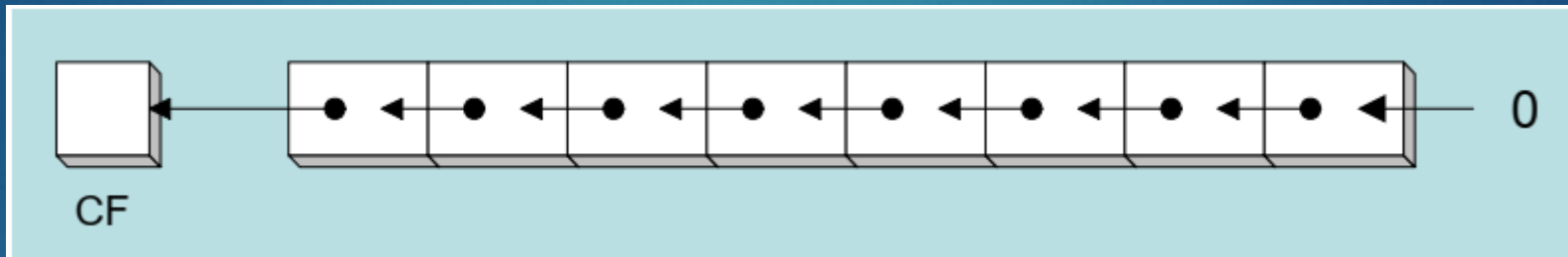


SHIFT & ROTATE INSTRUCTIONS

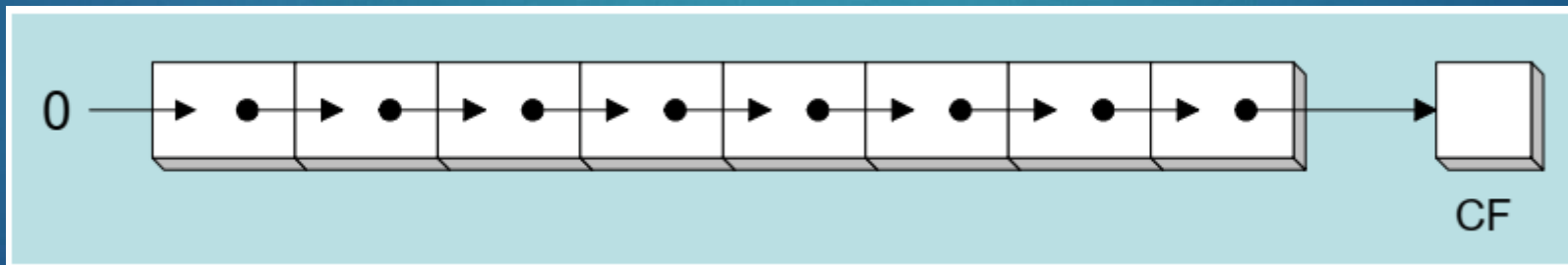
SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate carry left
RCR	Rotate carry right
SHLD	Double-precision shift left
SHRD	Double-precision shift right

SHL & SHR INSTRUCTION

- The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0



- The SHR (shift right) instruction performs a logical right shift on the destination operand, the highest bit position is filled with a zero



SHL & SHR INSTRUCTION

The first operand in **SHL** is the destination and the second is the shift count:

```
SHL  destination, count
```

The following lists the types of operands permitted by this instruction:

```
SHL  reg, imm8
```

```
SHL  mem, imm8
```

```
SHL  reg, CL
```

```
SHL  mem, CL
```

x86 processors permit *imm8* to be any integer between 0 and 255. Alternatively, the CL register can contain a shift count.

EXAMPLE

```
mov    bl, 8Fh                ; BL = 10001111b
shl    bl, 1                   ; CF = 1, BL = 00011110b
```

```
mov    al, 100000000b
shl    al, 2                   ; CF = 0, AL = 00000000b
```

APPLICATION

- ▶ Bitwise multiplication is performed when you shift a number's bits in leftward direction (toward the MSB).

- ▶ Example:

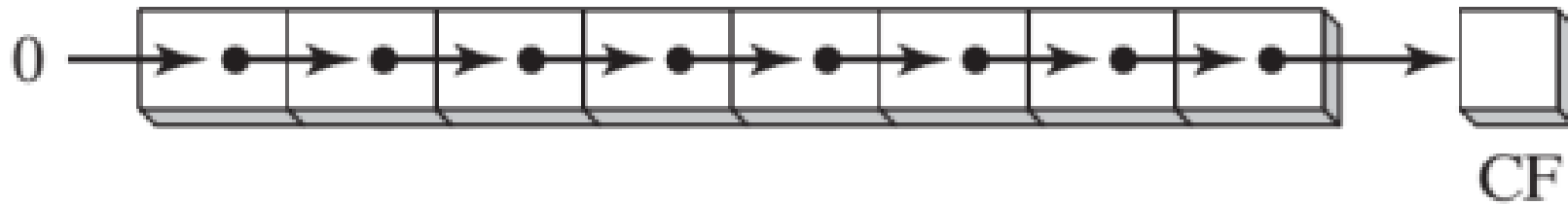
<code>mov dl,5</code>	Before:	<code>0 0 0 0 0 1 0 1</code>	= 5
<code>shl dl,1</code>	After:	<code>0 0 0 0 1 0 1 0</code>	= 10

- ▶ •SHL can perform multiplication by powers of 2.
- ▶ •Shifting any operand left by n bits multiplies the operand by 2^n .
- ▶ •For example, shifting the integer 5 left by 1 bit yields the product of $5 \times 2^1 = 10$.
- ▶ •If binary 00001010 (decimal 10) is shifted left by two bits, the result is the same as multiplying 10 by 2^2

<code>mov dl,10</code>	; before:	<code>00001010</code>
<code>shl dl,2</code>	; after:	<code>00101000</code>

SHL & SHR INSTRUCTION

SHR Instruction performs a logical right shift on the destination operand, replacing the highest bit with a 0. The lowest bit is copied into the Carry flag.



```
mov al,0D0h           ; AL = 11010000b
shr al,1               ; AL = 01101000b, CF = 0
```

```
mov al,00000010b
shr al,2               ; AL = 00000000b, CF = 1
```

APPLICATION

- ▶ Bitwise division is performed when you shift a number's bits in rightward direction (toward the LSB).
- ▶ Shifting an unsigned integer right by n bits divides the operand by 2^n .
- ▶ In the following statements, we divide 32 by 2^1 , producing 16.

```
mov dl,32
```

Before:

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 32

```
shr dl,1
```

After:

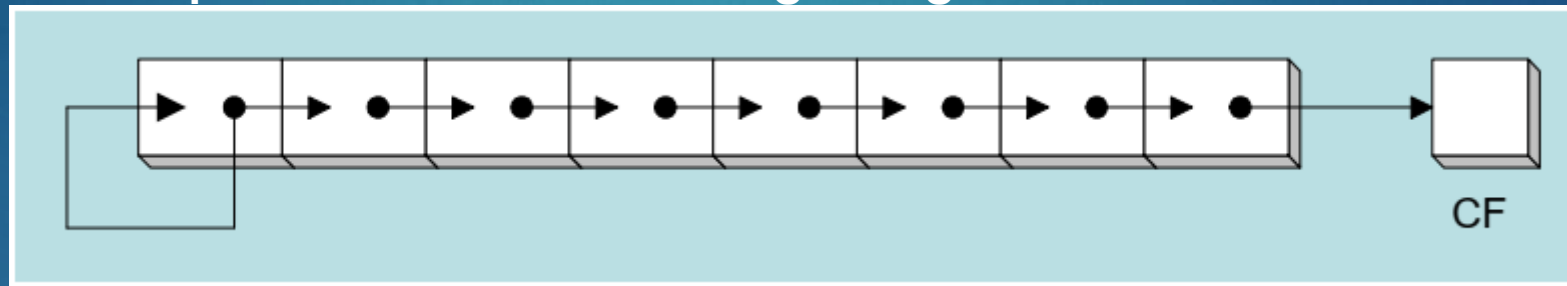
0	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---

 = 16

```
mov al,01000000b           ; AL = 64
shr al,3                    ; divide by 8, AL = 00001000b
```


SAL & SAR INSTRUCTION

- ▶ SAL (shift arithmetic left) is identical to SHL.
- ▶ SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand
 - ▶ An arithmetic shift preserves the number's sign



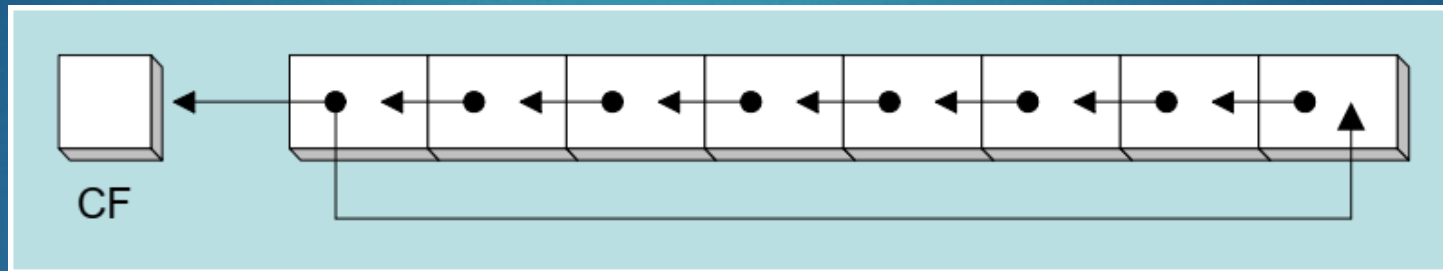
```
mov al,0F0h          ; AL = 11110000b (-16)
sar al,1              ; AL = 11111000b (-8), CF = 0
```

- An arithmetic shift preserves the number's sign.

```
mov dl,-80
sar dl,1              ; DL = -40
sar dl,2              ; DL = -10
```

ROL & ROR INSTRUCTION

- ▶ ROL (rotate) shifts each bit to the left
 - ▶ The highest bit is copied into both the Carry flag and into the lowest bit
 - ▶ No bits are lost shift preserves the number's sign

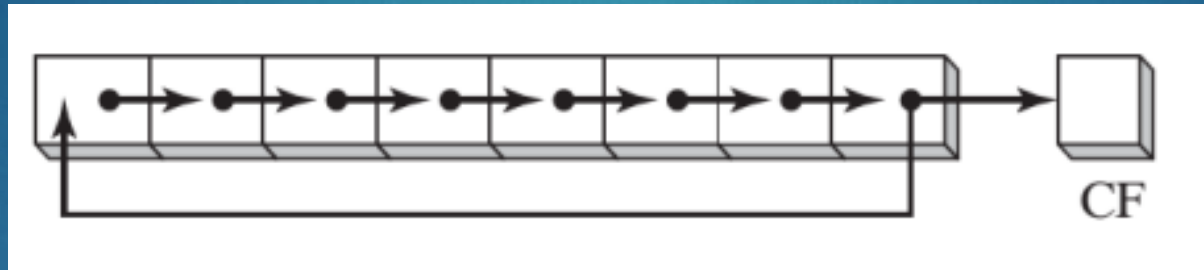


```
mov al,40h          ; AL = 01000000b
rol al,1             ; AL = 10000000b, CF = 0
rol al,1             ; AL = 00000001b, CF = 1
rol al,1             ; AL = 00000010b, CF = 0
```

- When using a rotation count greater than 1, the Carry flag contains the last bit rotated out of the MSB position.

ROL & ROR INSTRUCTION

- ROR Instruction shifts each bit to the right and copies the lowest bit into the Carry flag and the highest bit position (MSB).

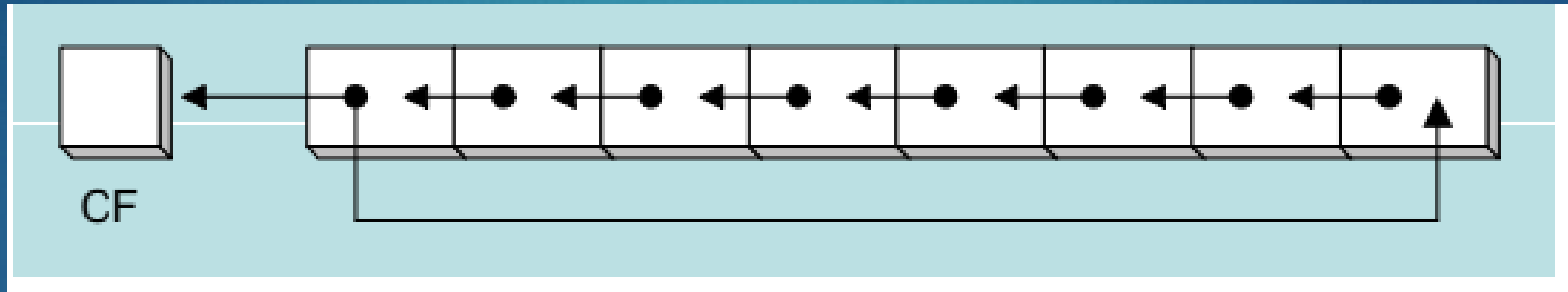


```
mov al,01h          ; AL = 00000001b
ror al,1             ; AL = 10000000b, CF = 1
ror al,1             ; AL = 01000000b, CF = 0
```

```
mov dl,3Fh
ror dl,4             ; DL = F3h
```

ROL & ROR INSTRUCTION

- ▶ ROL (rotate) shifts each bit to the left.
- ▶ The highest bit is copied into both the Carry flag and into the lowest bit.
- ▶ No bits are lost.

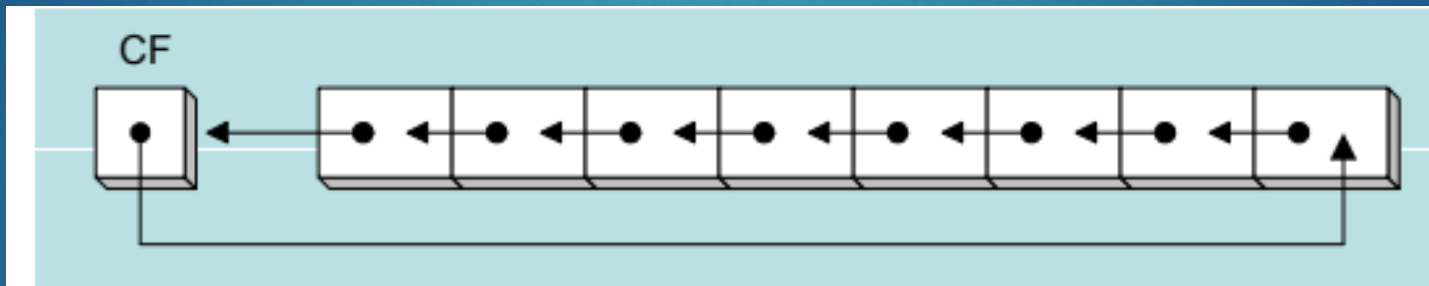


```
mov al,11110000b
rol al,1           ; AL = 11100001b
```

```
mov dl,3Fh
rol dl,4           ; DL = F3h
```

RCL & RCR INSTRUCTION

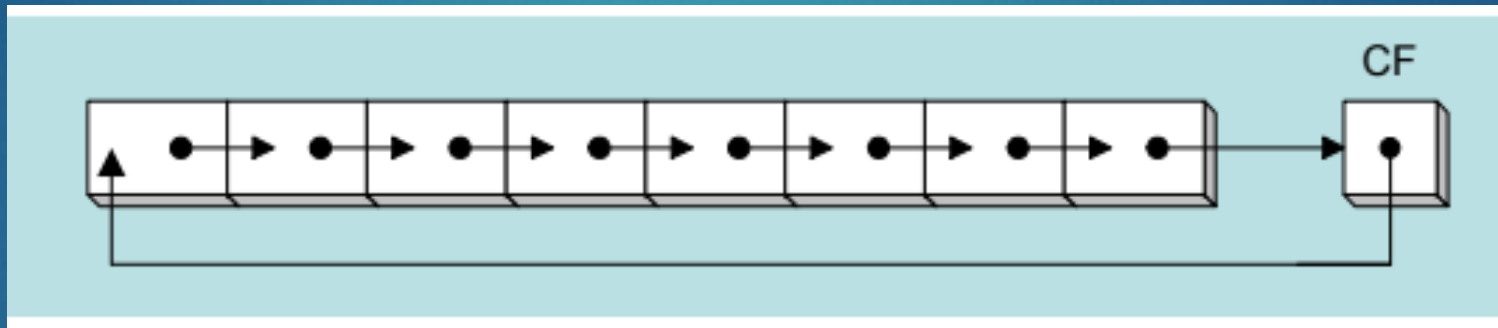
- ▶ RCL (rotate carry left) shifts each bit to the left
- ▶ Copies the Carry flag to the least significant bit
- ▶ Copies the most significant bit to the Carry flag



<code>clc</code>	<code>; CF = 0</code>
<code>mov bl,88h</code>	<code>; CF,BL = 0 10001000b</code>
<code>rcl bl,1</code>	<code>; CF,BL = 1 00010000b</code>
<code>rcl bl,1</code>	<code>; CF,BL = 0 00100001b</code>

RCL & RCR INSTRUCTION

- ▶ RCR (rotate carry right) shifts each bit to the right
- ▶ Copies the Carry flag to the most significant bit
- ▶ Copies the least significant bit to the Carry flag

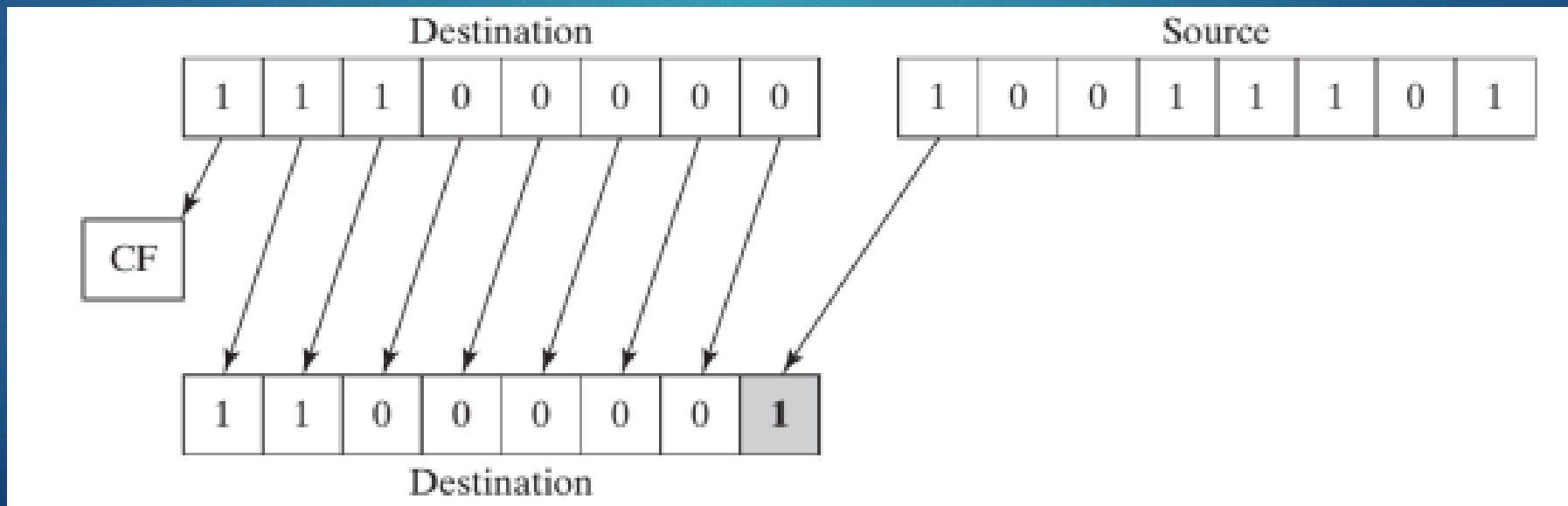


```
stc                ; CF = 1
mov ah,10h         ; CF,AH = 00010000 1
rcr ah,1           ; CF,AH = 10001000 0
```


SHLD AND SHRD INSTRUCTIONS

- ▶ The SHLD (shift left double) instruction shifts a destination operand a given number of bits to the left.
- ▶ The bit positions opened up by the shift are filled by the most significant bits of the source operand.

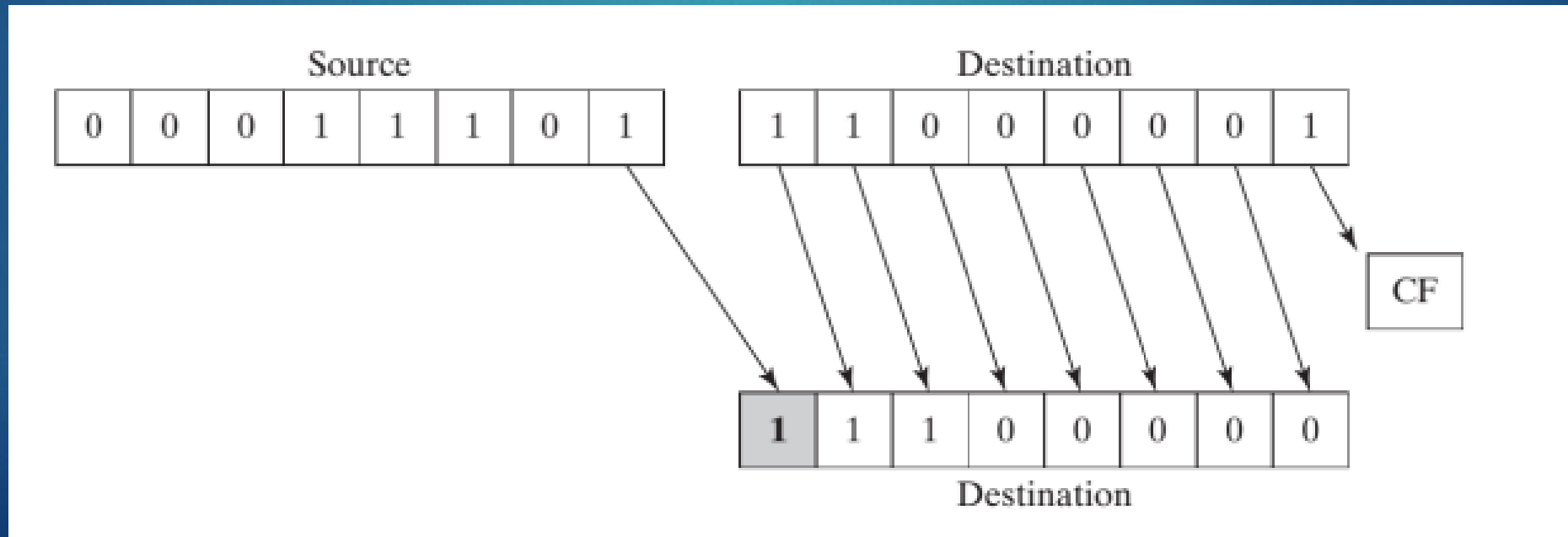
SHLD *dest, source, count*



SHLD AND SHRD INSTRUCTIONS

- ▶ The SHRD (shift right double) instruction shifts a destination operand a given number of bits to the right.
- ▶ The bit positions opened up by the shift are filled by the least significant bits of the source operand:

SHRD *dest, source, count*



SHLD AND SHRD INSTRUCTIONS

The source operand is not affected, but the Sign, Zero, Auxiliary, Parity, and Carry flags are affected.

The following instruction formats apply to both SHLD and SHRD:

SHLD *reg16, reg16, CL/imm8*

SHLD *mem16, reg16, CL/imm8*

SHLD *reg32, reg32, CL/imm8*

SHLD *mem32, reg32, CL/imm8*

SHLD example

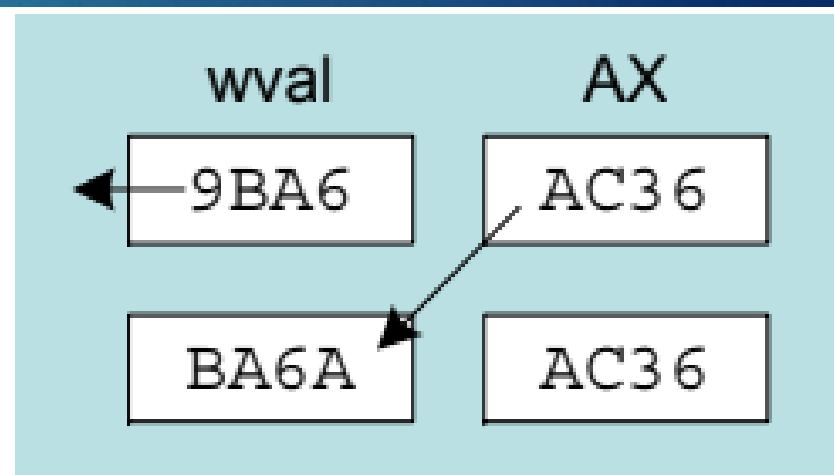
Shift wval 4 bits to the left and replace its lowest 4 bits with the high 4 bits of AX:

```
.data
wval WORD 9BA6h
.code
mov ax, 0AC36h
shld wval, ax, 4
```



Before:

After:



SHRD example

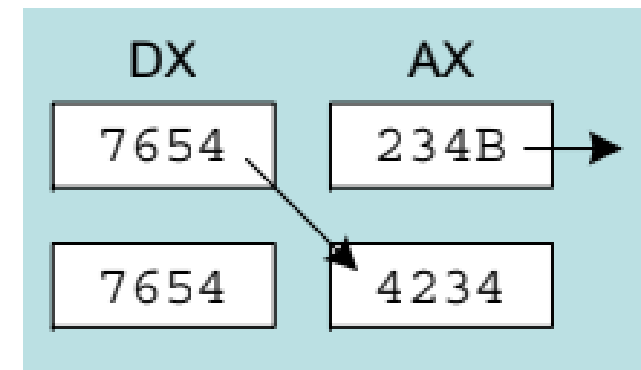
Shift AX 4 bits to the right and replace its highest 4 bits with the low 4 bits of DX:

```
mov ax,234Bh  
mov dx,7654h  
shrd ax,dx,4
```



Before:

After:



Shift and rotate applications

- ▶ Shifting Multiple Doublewords
- ▶ Binary Multiplication
- ▶ Displaying Binary Bits
- ▶ Isolating a Bit String

Binary multiplication

We already know that SHL performs unsigned multiplication efficiently when the multiplier is a power of 2.

Factor any binary number into powers of 2.

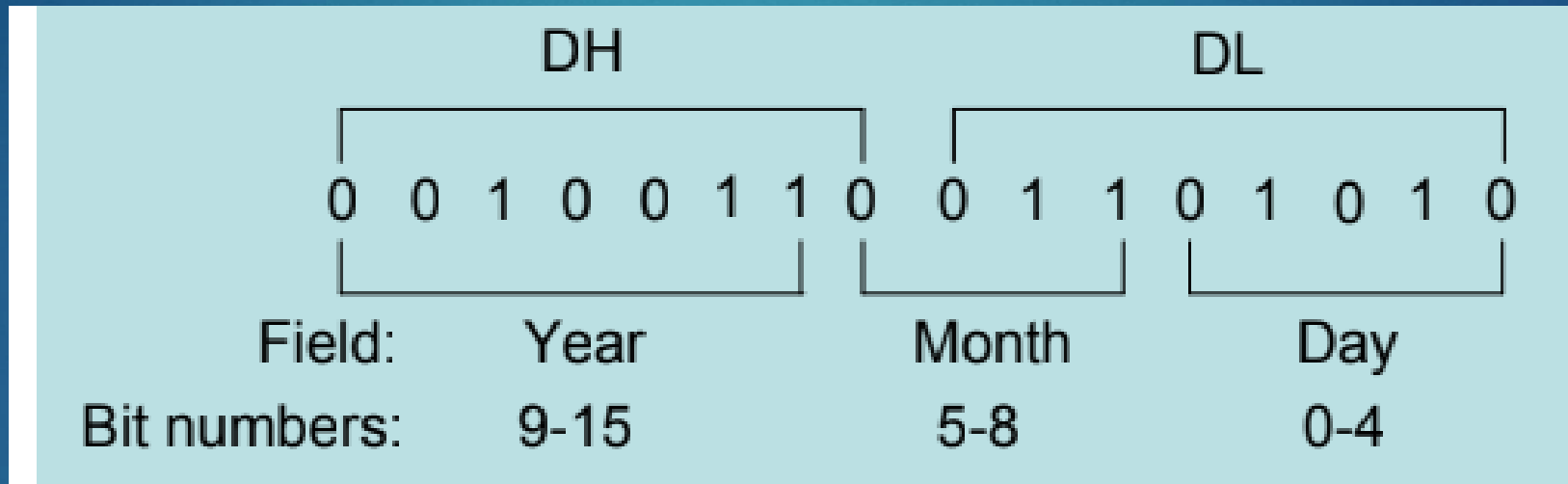
- For example, to multiply $EAX * 36$, factor 36 into $32 + 4$ and use the distributive property of multiplication to carry out the operation:

```
EAX * 36
= EAX * (32 + 4)
= (EAX * 32) + (EAX * 4)
```

```
mov eax,123
mov ebx,eax
shl eax,5
shl ebx,2
add eax,ebx
```

Isolating a bit string

The MS-DOS file date field packs the year (relative to 1980), month and , a day into 16 bits:



Isolating a bit string

```
mov al,dl          ; make a copy of DL
and al,00011111b   ; clear bits 5-7
mov day,al         ; save in day variable

mov ax,dx          ; make a copy of DX
shr ax,5           ; shift right 5 bits
and al,00001111b   ; clear bits 4-7
mov month,al       ; save in month variable

mov al,dh          ; make a copy of DX
shr al,1           ; shift right 1 bit
mov ah,0           ; clear AH to 0
add ax,1980        ; year is relative to 1980
mov year,ax        ; save in year
```

MULTIPLICATION AND DIVISION INSTRUCTIONS

In 32-bit mode, integer multiplication can be performed as a 32-bit, 16-bit, or 8-bit operation.

The process of multiplication and division is different for signed and unsigned numbers, so there are different Instructions for signed and unsigned multiplication and division.

The MUL and IMUL instructions perform unsigned and signed integer multiplication, respectively.

The DIV instruction performs unsigned integer division, and IDIV performs signed integer division.

Signed Vs Unsigned Multiplication

Suppose we want to multiply the eight-bit numbers 10000000 and 11111111.

Interpreted as unsigned numbers, they represent 128 and 255; respectively. The product is 32,640.

However, taken as signed numbers, they represent -128 and -1, respectively; and the product is 128.

Thus signed and unsigned numbers must be treated differently.

MUL INSTRUCTION

- In 32-bit mode, the MUL (unsigned multiply) instruction comes in three versions:

1. The first version multiplies an 8-bit operand by the AL register.
2. The second version multiplies a 16-bit operand by the AX register
3. Third version multiplies a 32-bit operand by the EAX register.

- The multiplier and multiplicand must always be the same size, and the product is twice their size.

```
MUL reg/mem8 ; byte form
```

```
MUL reg/mem16 ; word form
```

```
MUL reg/mem32 ; DWORD form
```


MUL INSTRUCTION

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

When AX is multiplied by a 16-bit operand, for example, the product is stored in the combined DX and AX registers.

- the high 16 bits of the product are stored in DX, and the low 16 bits are stored in AX.

The Carry flag is set if DX is not equal to zero, which lets us know that the product will not fit into the lower half of the implied destination operand.

- After MUL, CF/OF=0; if upper half of the result is zero; 1 otherwise.

EXAMPLES

100h * 2000h, using 16-bit operands:

```
.data
val1 WORD 2000h
val2 WORD 100h
.code
mov ax,val1
mul val2    ; DX:AX=00200000h, CF=1
```

The Carry flag indicates whether or not the upper half of the product contains significant digits.

12345h * 1000h, using 32-bit operands:

```
mov eax,12345h
mov ebx,1000h
mul ebx     ; EDX:EAX=0000000012345000h, CF=0
```

IMUL INSTRUCTION

- The IMUL (signed multiply) instruction performs signed integer multiplication.
- Unlike the MUL instruction, IMUL preserves the sign of the product.
 - It does this by sign extending the highest bit of the lower half of the product into the upper bits of the product.
- The x86 instruction set supports three formats for the IMUL instruction: one operand, two operands, and three operands.

IMUL INSTRUCTION

- The one-operand formats store the product in AX, DX:AX, or EDX : EAX

```
IMUL reg/mem8      ; AX = AL * reg/mem8
```

```
IMUL reg/mem16     ; DX:AX = AX * reg/mem16
```

```
IMUL reg/mem32     ; EDX:EAX = EAX * reg/mem32
```

. Two-Operand Formats (32-Bit Mode): stores the product in the first operand, which must be a register. The second operand (the multiplier) can be a register, a memory operand, or an immediate value:

```
IMUL reg16, reg/mem16
```

```
IMUL reg16, imm8
```

```
IMUL reg16, imm16
```

```
IMUL reg32, reg/mem32
```

```
IMUL reg32, imm8
```

```
IMUL reg32, imm32
```

IMUL INSTRUCTION

The two-operand formats truncate the product to the length of the destination. If significant digits are lost, the Overflow and Carry flags are set.

- The three-operand formats in 32-bit mode store the product in the first operand. The second operand can be a 16-bit register or memory operand, which is multiplied by the third operand, an 8- or 16-bit immediate value:

```
IMUL reg16, reg/mem16, imm8
```

```
IMUL reg16, reg/mem16, imm16
```

```
IMUL reg32, reg/mem32, imm8
```

```
IMUL reg32, reg/mem32, imm32
```


IMUL INSTRUCTION

- After IMUL, CF/OF = 0, if the upper half of the result is the sign extension of the lower half; CF/OF = 1 otherwise
- This means that the bits of the upper half are the same as the sign bit of the lower half.

```
mov al,48  
mov bl,4  
imul bl                      ; AX = 00C0h, OF = 1
```

```
mov al,-4  
mov bl,4  
imul bl                      ; AX = FFF0h, OF = 0
```


IMUL INSTRUCTION

- The following instructions multiply 48 by 4, producing +192 in DX:AX. DX is a sign extension of AX, so the Overflow flag is clear:

```
mov ax,48  
mov bx,4  
imul bx           ; DX:AX = 000000C0h, OF = 0
```

DIV INSTRUCTION

- In 32-bit mode, the DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit unsigned integer division.

DIV *reg/mem*8

DIV *reg/mem*16

DIV *reg/mem*32

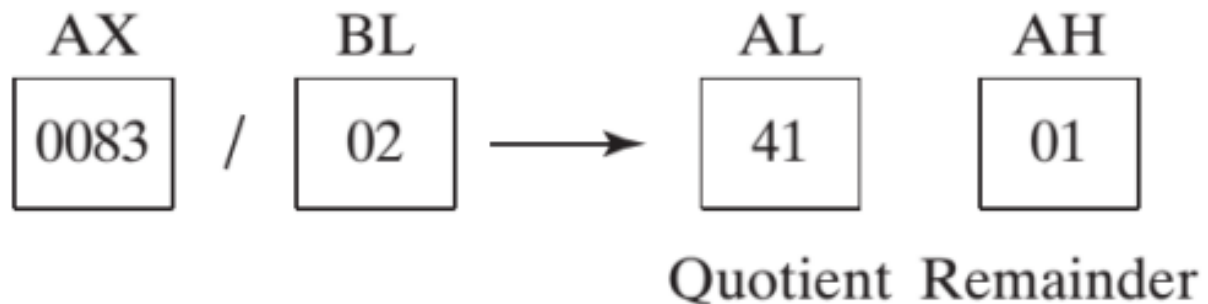
- Where the single *reg/mem* operand is divisor.
- When division is performed, we obtain two results, the quotient and the remainder.
- Quotient and remainder have the same size as the divisor.

DIV INSTRUCTION

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

```
mov ax,0083h
mov bl,2
div bl
```

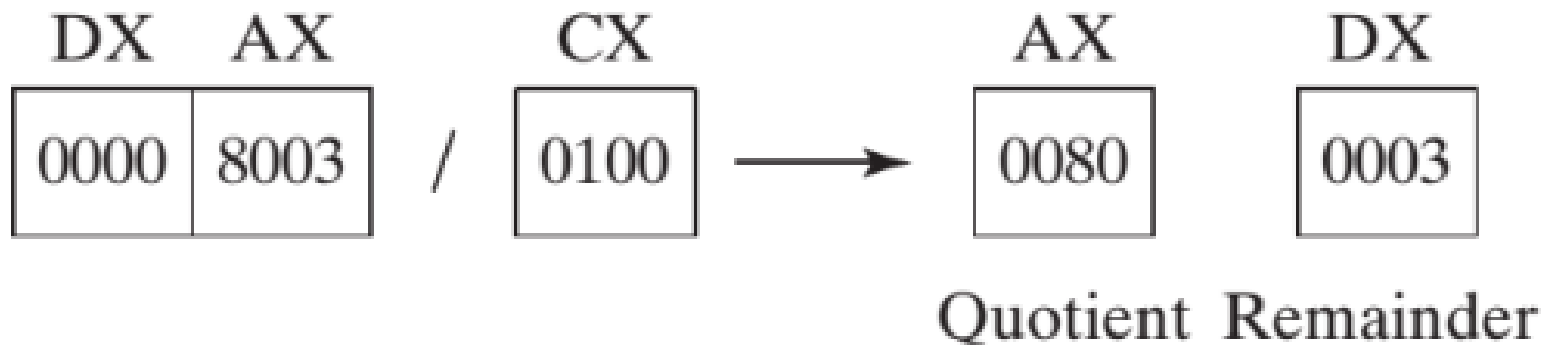
```
; dividend
; divisor
; AL = 41h, AH = 01h
```



DIV INSTRUCTION

- In word (16-bit) division, the dividend is in DX:AX even if the actual dividend will fit in AX. In this case DX should be cleared:

```
mov dx,0           ; clear dividend, high
mov ax,8003h        ; dividend, low
mov cx,100h         ; divisor
div cx              ; AX = 0080h, DX = 0003h
```

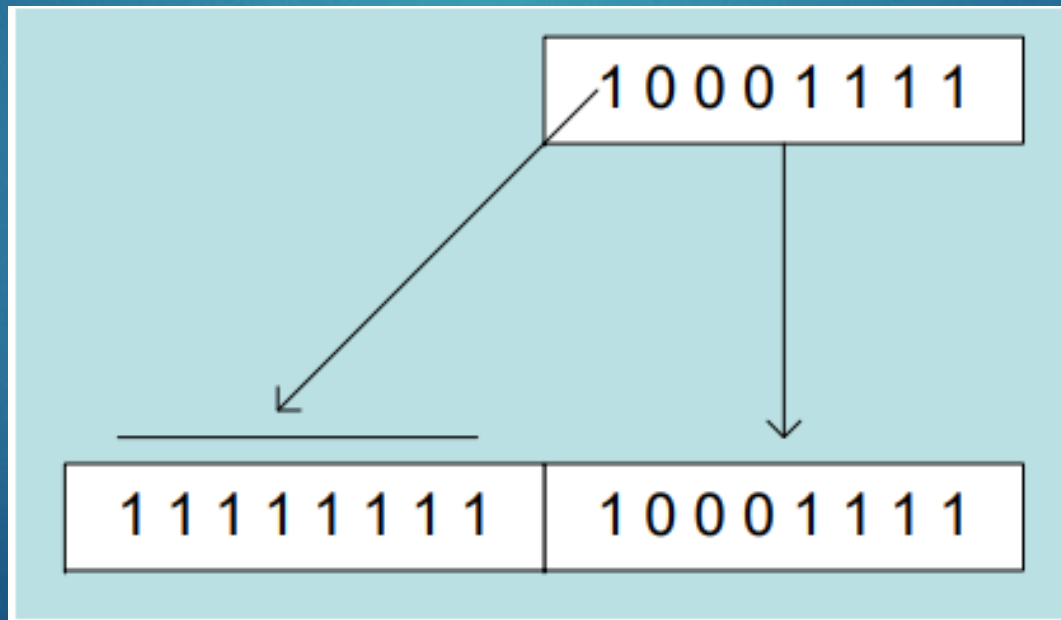


SIGNED INTEGER DIVISION

Signed integers must be sign-extended before division takes place.

fill high byte/word/doubleword with a copy of the low byte/word/doubleword's sign bit.

- For example, the high byte contains a copy of the sign bit from the low byte:



CBW, CWD, CDQ instructions

The CBW, CWD, and CDQ instructions provide important sign-extension operations:

- The CBW instruction (convert byte to word) extends the sign bit of AL into AH.
- The CWD (convert word to doubleword) instruction extends the sign bit of AX into DX.
- The CDQ (convert doubleword to quadword) instruction extends the sign bit of EAX into EDX.

```
mov  eax,0FFFFFF9Bh      ; -101 (32 bits)
cdq                      ; EDX:EAX = FFFFFFFF9Bh
                           ; -101 (64 bits)
```

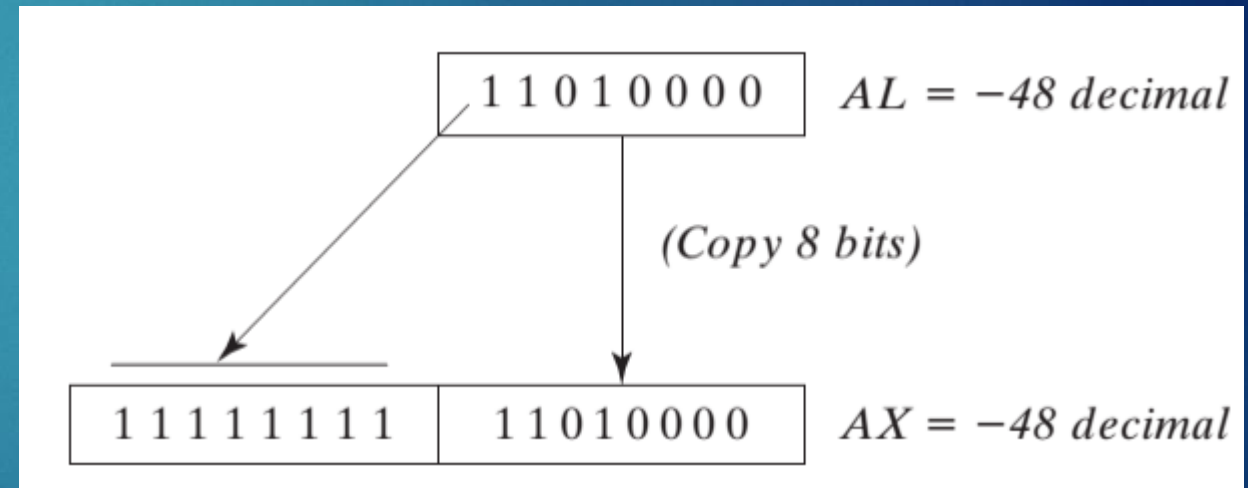

IDIV INSTRUCTION

The IDIV (signed divide) instruction performs signed integer division, using the same operands as DIV.

- Before executing 8-bit division, the dividend (AX) must be completely sign-extended.

Example: 8-bit division of -48 by 5

```
mov al,-48
cbw          ; extend AL into AH
mov bl,5
idiv bl      ; AL = -9, AH = -3
```



IDIV INSTRUCTION

Example: 16-bit division of -48 by 5

```
mov    ax, -48
cwd                    ; extend AX into DX
mov    bx, 5
idiv   bx              ; AX = -9,   DX = -3
```

Example: 32-bit division of -48 by 5

```
mov    eax, -48
cdq                    ; extend EAX into EDX
mov    ebx, 5
idiv   ebx             ; EAX = -9,   EDX = -3
```

WHY SIGN EXTENSION IS NECESSARY

```
.code
mov     dx,0
mov     ax,wordVal           ; DX:AX = 0000FF9Bh
mov     bx,2                 ; BX is the divisor
idiv    bx                   ; divide DX:AX by BX (signed
                             operation)
```

```
.data
wordVal SWORD -101           ; FF9Bh
.code
mov     dx,0
mov     ax,wordVal           ; DX:AX = 0000FF9Bh
cwd     ; DX:AX = FFFFFFF9Bh (-101)
mov     bx,2
idiv    bx                   ; AX = FFCEh (-50)
```

Divide overflow

- Divide overflow happens when the quotient is too large to fit into the destination.

```
mov ax, 1000h  
mov bl, 10h  
div bl
```

It causes a CPU interrupt and halts the program. (divided by zero cause similar results).

Extended addition and subtraction

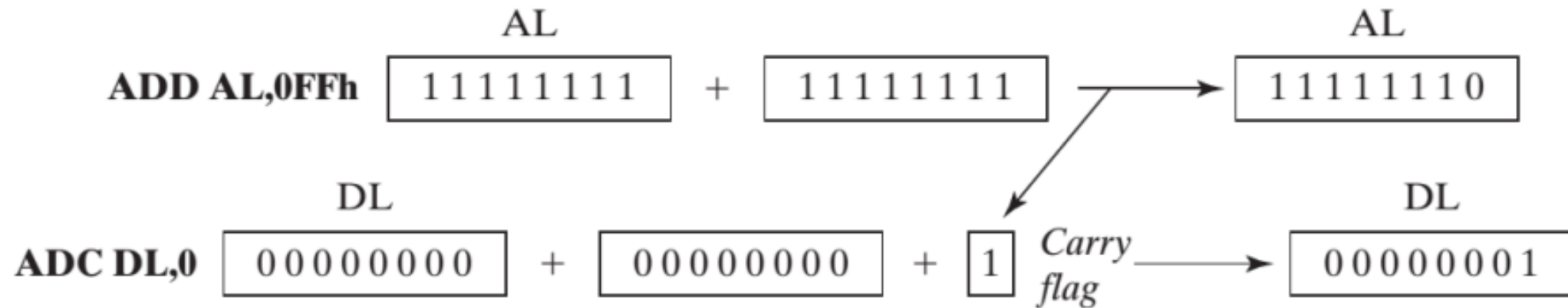
ADC Instruction

- The ADC (add with carry) instruction adds both a source operand and the contents of the Carry flag to a destination operand.
- The instruction formats are the same as for the ADD instruction, and the operands must be the same size:

```
ADC    reg, reg  
ADC    mem, reg  
ADC    reg, mem  
ADC    mem, imm  
ADC    reg, imm
```

ADC Instruction

```
mov  dl,0
mov  al,0FFh
add  al,0FFh                ; AL = FEh
adc  dl,0                   ; DL/AL = 01FEh
```



```
mov  edx,0
mov  eax,0FFFFFFFFh
add  eax,0FFFFFFFFh
adc  edx,0 ;EDX:EAX = 00000001FFFFFFFFh
```


SBB INSTRUCTION

- The SBB (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag from a destination operand.
- The possible operands are the same as for the ADC instruction

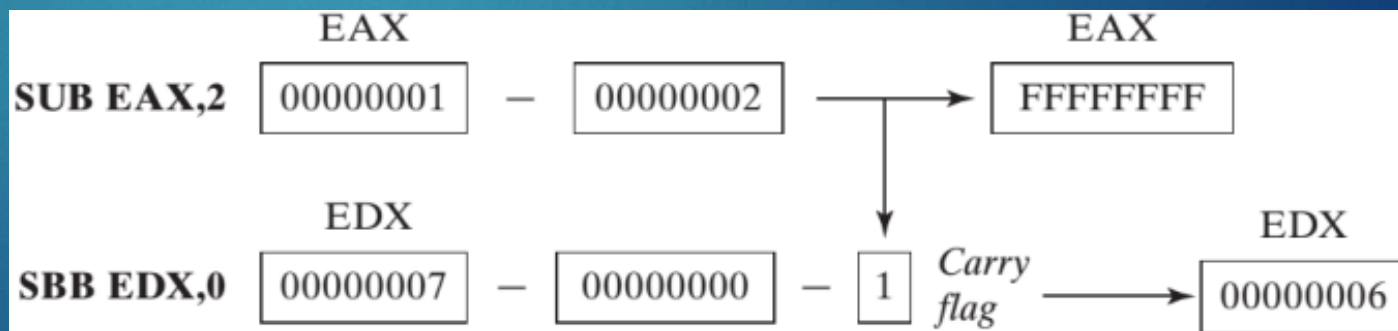
```
mov    edx, 7           ; upper half
mov    eax, 1           ; lower half
sub    eax, 2           ; subtract 2
sbb    edx, 0           ; subtract upper half
```

Before:

EDX	EAX
00000007	00000001

After:

EDX	EAX
00000006	FFFFFFFF



Extended addition example

```
.data
op1 QWORD 0A2B2A40674981234h
op2 QWORD 08010870000234502h
sum DWORD 3 dup(?)
      ; = 0000000122C32B0674BB5736
.code
...
mov  esi,OFFSET op1 ; first operand
mov  edi,OFFSET op2 ; second operand
mov  ebx,OFFSET sum ; sum operand
mov  ecx,2           ; number of doublewords
call Extended_Add
...
```

Extended addition example

```
Extended_Add PROC
```

```
Pushad
```

```
clc
```

```
L1:
```

```
    mov eax,[esi] ; get the first integer
```

```
    adc eax,[edi] ; add the second integer
```

```
    pushfd          ; save the Carry flag
```

```
    mov [ebx],eax   ; store partial sum
```

```
    add esi,4       ; advance all 3 pointers
```

```
    add edi,4
```

```
    add ebx,4
```

```
    popfd           ; restore the Carry flag
```

```
    loop L1         ; repeat the loop
```

```
    adc word ptr [ebx],0 ; add leftover carry
```

```
    popad
```

```
    ret
```

```
Extended_Add ENDP
```

