بسم اللہ الرحمن الرحیم

EE213 COMPUTER ORGANIZATION AND ASSEMBLY LANGUAGE

Spring 2018

7

Integer Arithmetic

OUTLINES

- 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.

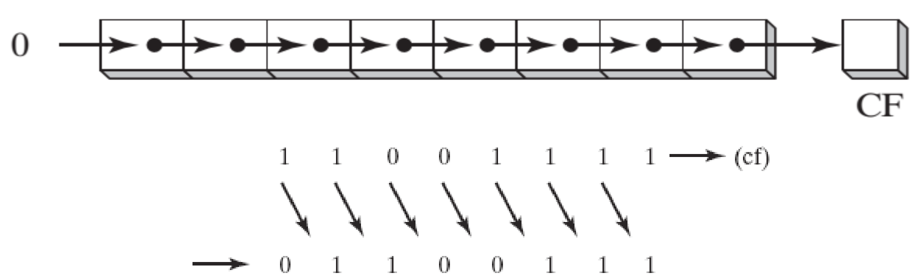
7.1 SHIFT AND ROTATE INSTRUCTIONS

- *Bit shifting* means to move bits right and left inside an operand.
- Shift and Rotate instructions affect the overflow and carry flags.

Logical Shifts and Arithmetic Shifts

- There are two ways to shift an operand's bits.

1. **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.



2. **Arithmetic Shift:** The newly created bit position is filled with a copy of the original number's sign bit:

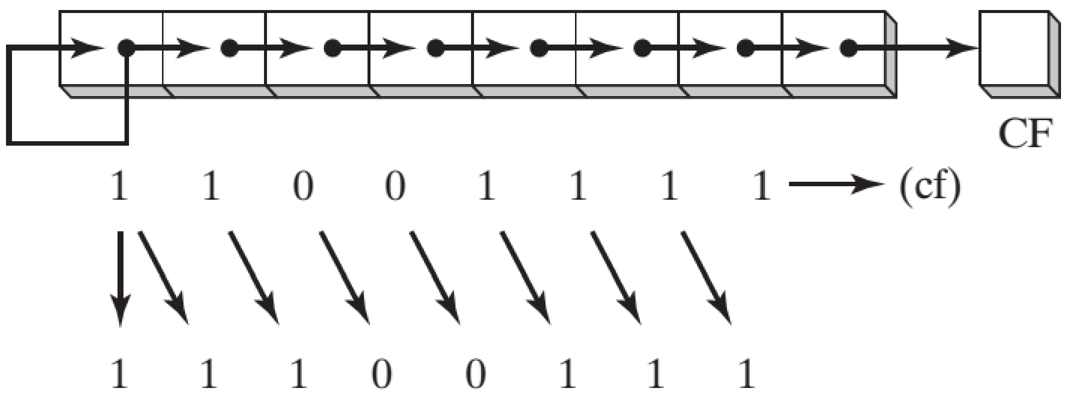


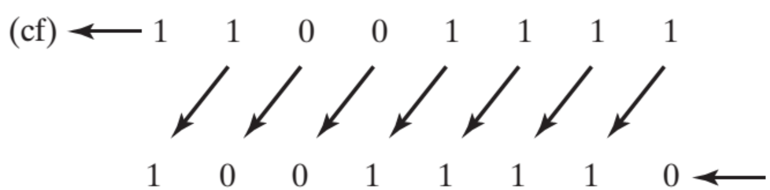
Table 7-1 Shift and Rotate Instructions.

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 INSTRUCTION

SHL Instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.

- The highest bit (msb) is moved to the Carry flag, and the bit that was in the Carry flag is discarded



Operand types for SHL:

```
SHL  reg,imm8
SHL  mem,imm8
SHL  reg,CL
SHL  mem,CL
```

E.g.

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

mov  al,10000000b
shl  al,2                    ; CF = 0, AL = 00000000b
```

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

```
mov  dl,5        Before:  | 0 0 0 0 0 1 0 1 |  = 5

shl  dl,1        After:   | 0 0 0 0 1 0 1 0 |  = 10
```

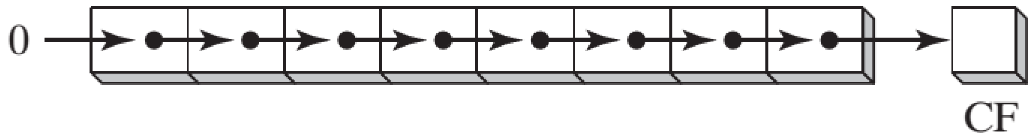
- SHL can perform multiplication by powers of 2.
- Shifting any operand left by *n* bits multiplies the operand by 2^n.
- For example, shifting the integer 5 left by 1 bit yields the product of 5 X 2^1 = 10.
- If binary `00001010` (decimal `10`) is shifted left by two bits, the result is the same as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

```
mov dl,10                        ; before: 00001010
shl dl,2                         ; after:  00101000
```

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
```

- *Bitwise division* is accomplished when you shift a number's bits in a 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
shr  dl,1
```

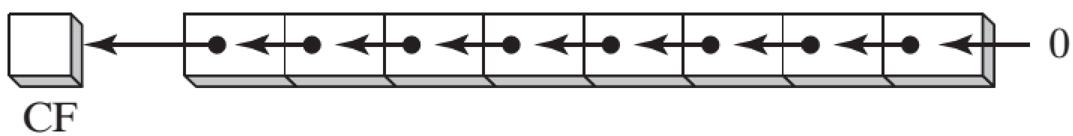
Before: $\boxed{0\ 0\ 1\ 0\ 0\ 0\ 0\ 0} = 32$

After: $\boxed{0\ 0\ 0\ 1\ 0\ 0\ 0\ 0} = 16$

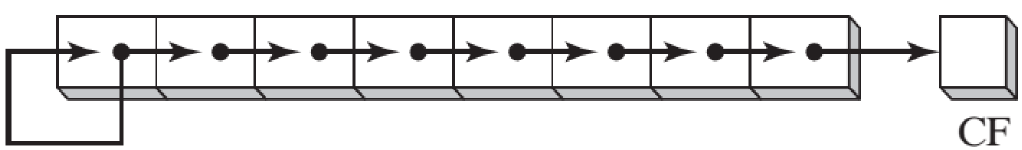
```
mov  al,01000000b               ; AL = 64
shr  al,3                       ; divide by 8, AL = 00001000b
```

SAL AND SAR INSTRUCTION

- The **SAL (shift arithmetic left)** Instruction works the same as the SHL instruction.



- The **SAR (shift arithmetic right)** instruction performs a right arithmetic shift on its destination operand:



- The operands for SAL and SAR are identical to those for SHL and SHR.

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

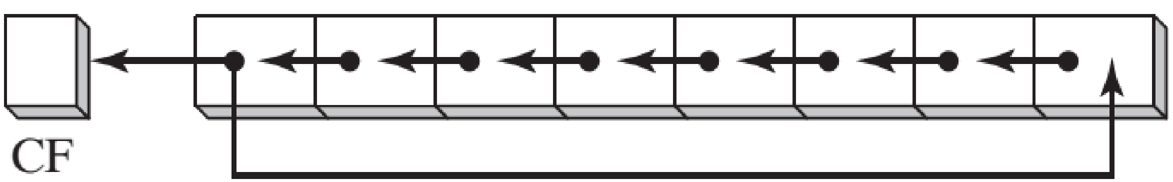
- You can divide a signed operand by a power of 2

```
mov dl,-128                             ; DL = 10000000b
sar dl,3                                ; DL = 11110000b (-16)
```

ROL INSTRUCTION

- *Bitwise rotation* occurs when you move the bits in a circular fashion.

- The **ROL** (rotate left) instruction shifts each bit to the left. The highest bit is copied into the Carry flag and the lowest bit position.



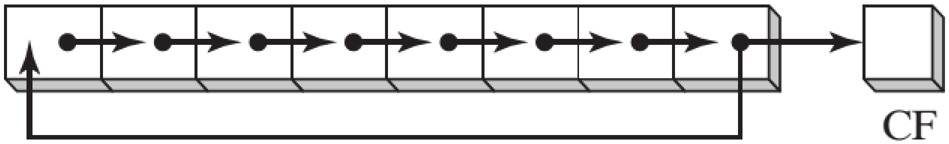
```
mov al,40h              ; AL = 01000000b
rol al,1                ; AL = 10000000b, CF = 0
rol al,1                ; AL = 00000001b, CF = 1
rol al,1                ; AL = 00000010b, CF = 0
```

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

```
mov al,00100000b
rol al,3                ; CF = 1, AL = 00000001b
```

ROR INSTRUCTION

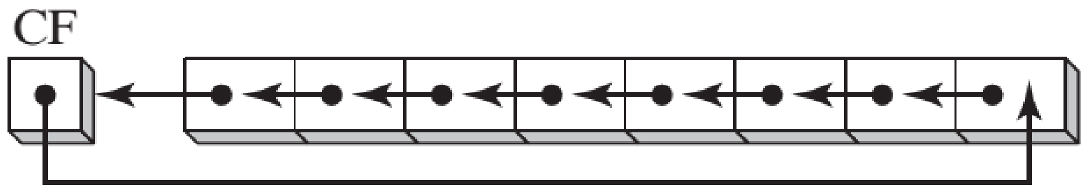
ROR Instruction shifts each bit to the right and copies the lowest bit into the Carry flag and the highest bit position (MSB).




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

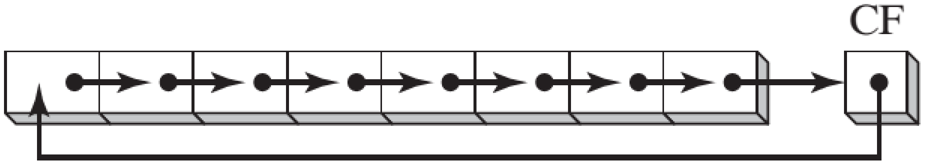
RCL AND RCR INSTRUCTIONS

- The **RCL** (**rotate carry left**) instruction shifts each bit to the left, copies the Carry flag to the LSB, and copies the MSB into the Carry flag.




```
clc                     ; CF = 0
mov bl,88h              ; CF,BL = 0 10001000b
rcl bl,1                ; CF,BL = 1 00010000b
rcl bl,1                ; CF,BL = 0 00100001b
```

RCR Instruction: The RCR (rotate carry right) instruction shifts each bit to the right, copies the Carry flag into the MSB, and copies the LSB into the Carry flag:

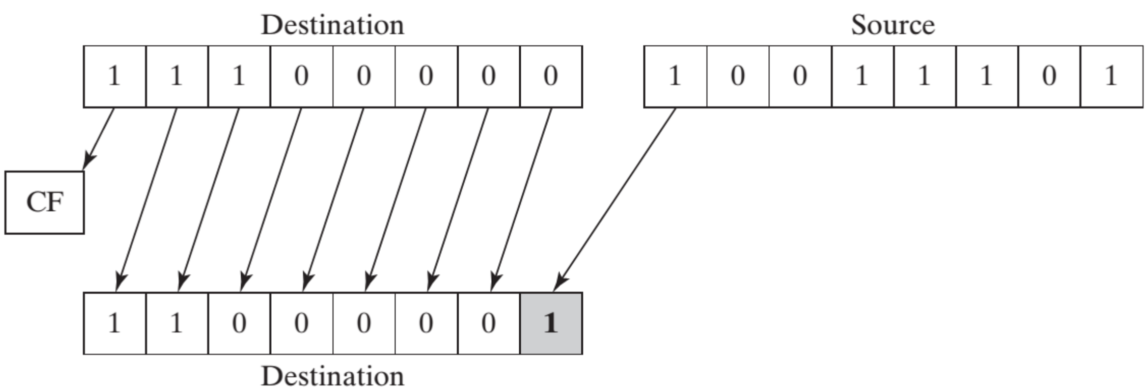



```
stc                     ; CF = 1
mov ah,10h              ; AH, CF = 00010000 1
rcr ah,1                ; AH, CF = 10001000 0
```

SHLD AND SHRD INSTRUCTIONS

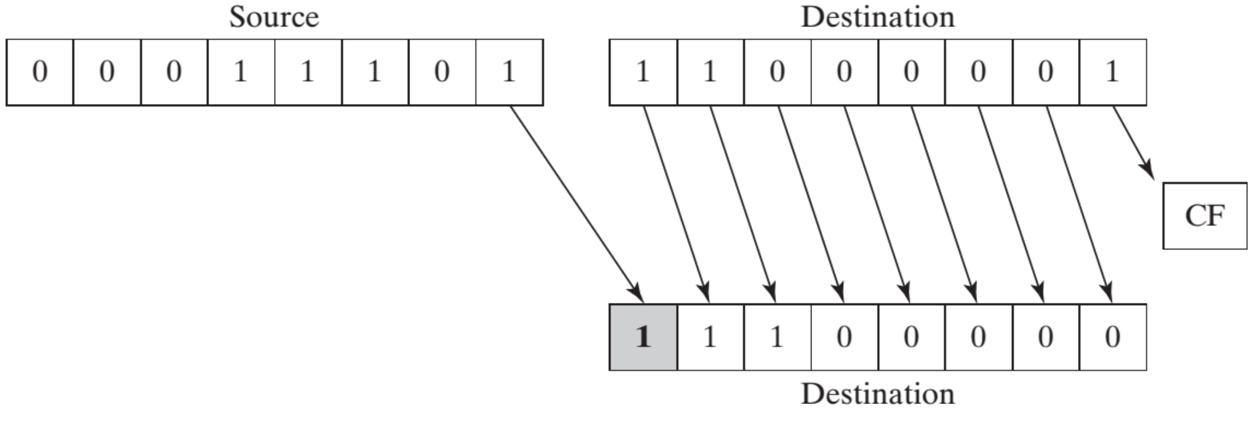
- The **SHLD** (shift left double) instruction shifts a destination operand a given number of bits to the left.
- The bit positions opened up by the shift are filled by the most significant bits of the source operand.

SHLD *dest, source, count*



- 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*



- 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:

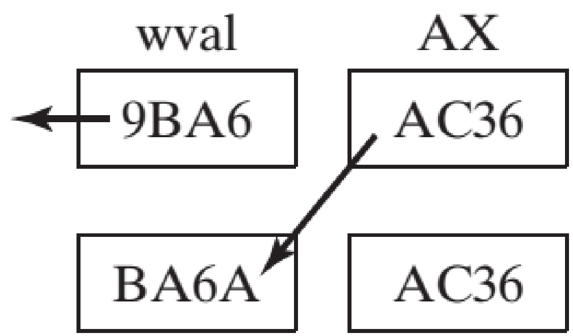
SHLD *reg16, reg16, CL/imm8*

SHLD *mem16, reg16, CL/imm8*

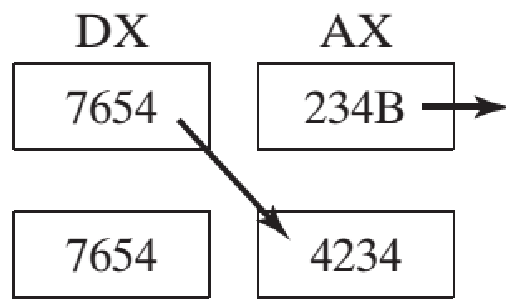
SHLD *reg32, reg32, CL/imm8*

SHLD *mem32, reg32, CL/imm8*

```
.data
        wval WORD 9BA6h
.code
        mov ax,0AC36h
        shld wval,ax,4              ; wval = BA6Ah
```



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



7.3 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
```

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.

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.

SINGLE OPERAND IMUL

- 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 IMUL

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
```

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.

THREE OPERAND IMUL

- 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
```

- 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
```

- 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/mem8*

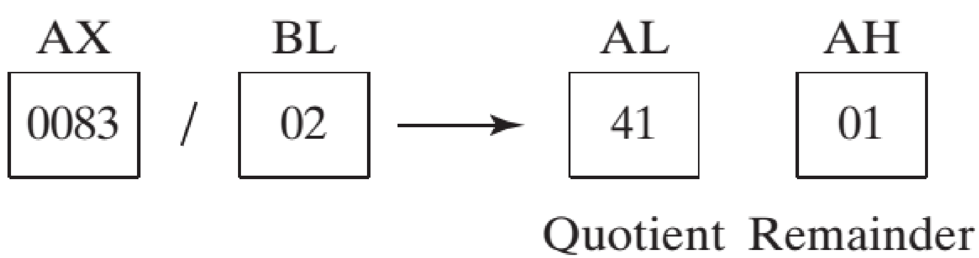
  `DIV` *reg/mem16*

  `DIV` *reg/mem32*

- 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.

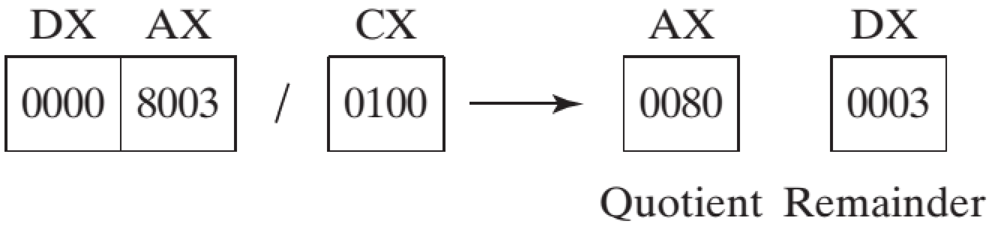
Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

```
mov ax,0083h                    ; dividend
mov bl,2                        ; divisor
div bl                          ; AL = 41h, AH = 01h
```



- 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 integer division is nearly identical to unsigned division, with one important difference: The dividend must be **sign-extended** before the division takes place.
  - *Sign extension* is the term used for copying the highest bit of a number into all of the upper bits of its enclosing variable or register.

- **Sign Extension Instructions (CBW, CWD, CDQ)**

- 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.

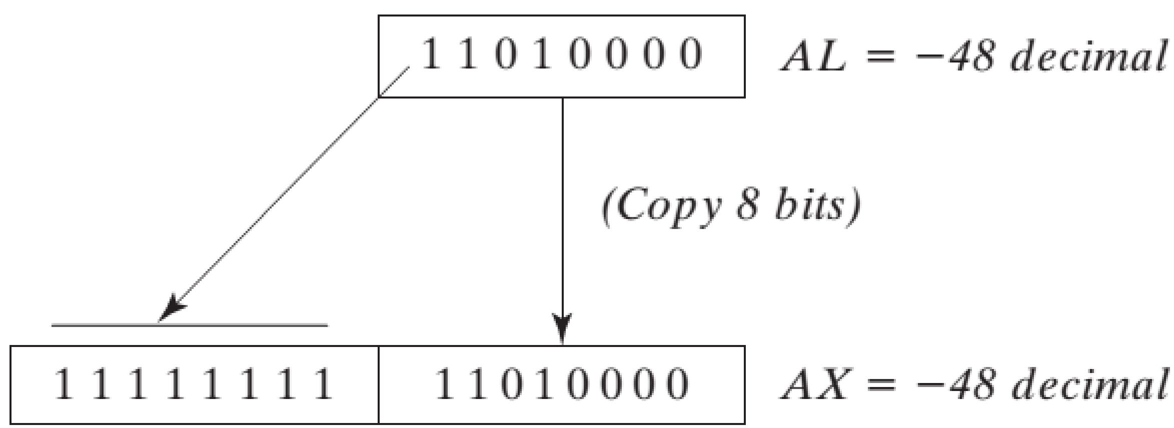
- The `CDQ` (convert doubleword to quadword) instruction extends the sign bit of EAX into EDX.

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.

```
.data
      byteVal SBYTE -48                  ; D0 hexadecimal
.code
      mov al,byteVal                     ; lower half of dividend
      cbw                                ; extend AL into AH
      mov bl,+5                          ; divisor
      idiv bl                            ; AL = -9, AH = -3
```



WHY SIGN EXTENSION IS NECESSARY

```
.data
byteVal SBYTE -48               ; D0 hexadecimal
.code
mov  ah,0                       ; upper half of dividend
mov  al,byteVal                 ; lower half of dividend
mov  bl,+5                      ; divisor
idiv bl                         ; AL = 41, AH = 3
```

SUMMARY

- Shift and Rotate Instructions
  - SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR, SHLD, SHRD

- Multiplication and Division Instructions
  - MUL
  - IMUL
  - DIV
  - IDIV

- Sign Extension Instructions (CBW, CWD, CDQ)