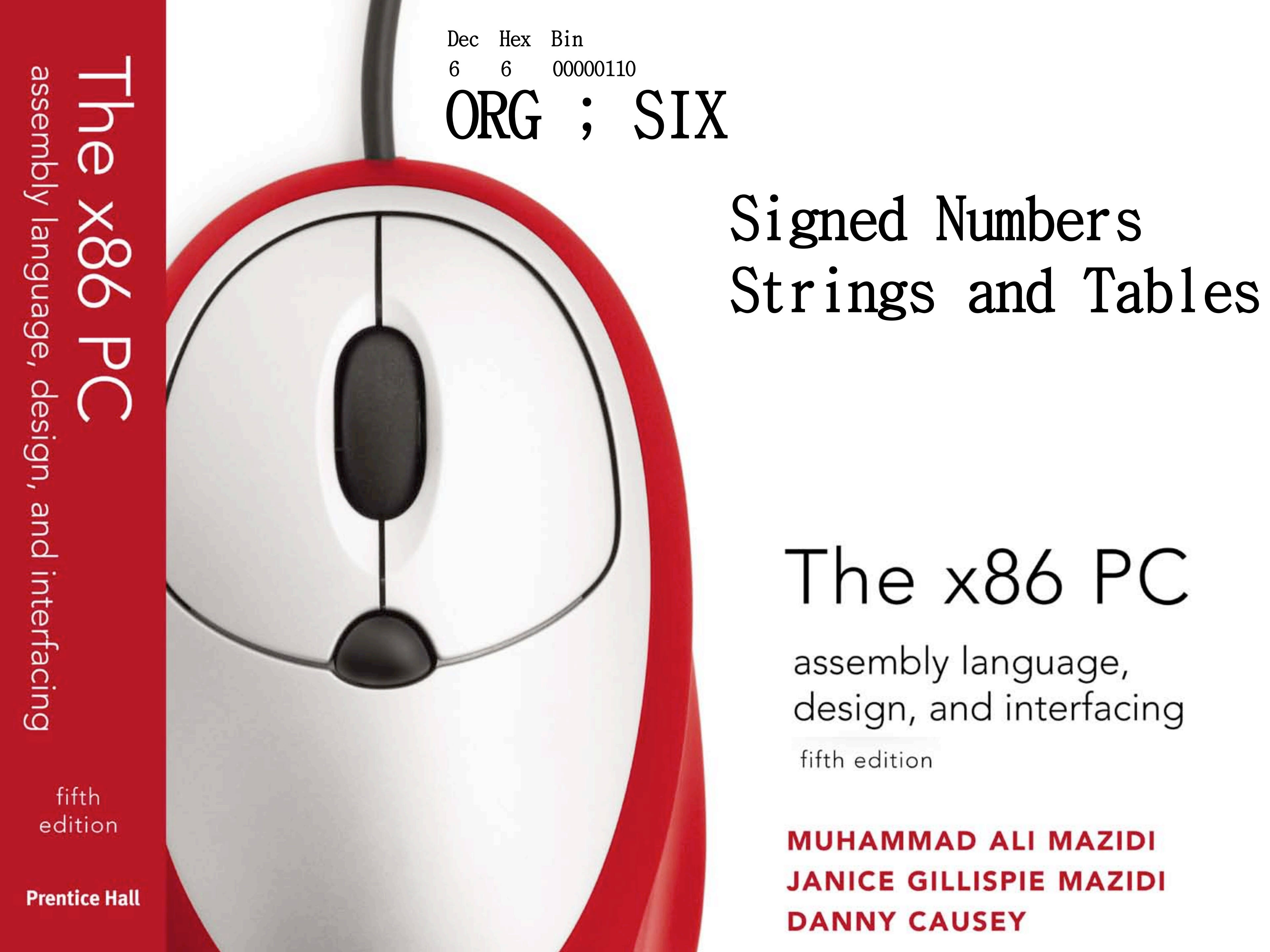
Dec Hex Bin

6 6 00000110

ORG ; SIX

Signed Numbers Strings and Tables



OBJECTIVES
this chapter enables the student to:

- Convert a number to its 2's complement.
- Code signed arithmetic instructions:
  - ADD, SUB, IMUL, and IDIV.
- Demonstrate how arithmetic instructions affect the sign flag.
- Explain the difference between a carry and an overflow.
- Prevent overflow errors by sign-extending data.
- Code signed shift instructions:
  - SAL and SAR.

OBJECTIVES (*cont*)

this chapter enables the student to:

- Code logic instruction CMP for signed numbers and explain its effect on the flag register.
- Code conditional jump instructions after CMP of signed data.
- Explain the function of registers SI and DI in string instructions.
- Describe the operation of the direction flag in string instructions.
- Code instructions CLD and STD to control the direction flag.

OBJECTIVES (*cont*)

this chapter enables the student to:

- Describe the operation of the REP prefix
- Code string instructions:
  - MOVSB and MOVSW for data transfer.
  - STOS, LODS to store and load the contents of AX.
  - CMPS to compare two strings of data.
  - SCAS to scan a string for data matching in AX.
  - XLAT for table processing .

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
Concept of signed numbers in computers

- Many applications require signed data, & computers must be able to accommodate such numbers.
  - The most significant bit (MSB) is set aside for the sign (+ or -) & the rest of the bits are used for the magnitude.
  - The sign is represented by 0 for positive (+) numbers and 1 for negative (-) numbers.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS signed byte operands

- In signed byte operands, **D7** (**MSB**) is the sign and **D0** to **D6** are set aside for the *magnitude* of the number.
  - If D7 = 0, the operand is *positive*.
  - If D7 = 1, it is *negative.*

D7	D6	D5	D4	D3	D2	D1	D0
sign	magnitude						

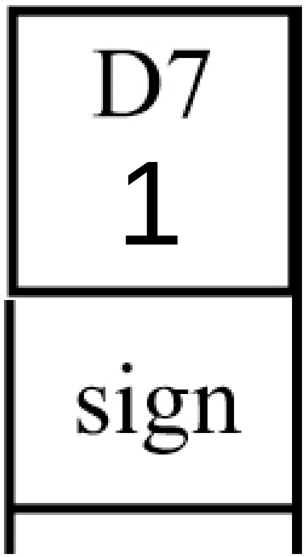
6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
positive numbers

- The range of positive numbers that can be represented by the format above is 0 to +127.
  - If a positive number is larger than +127, a word-sized operand must be used.

0	0000	0000
+1	0000	0001
+5	0000	0101
...	...	....
+127	0111	1111

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
negative numbers

- For negative numbers D7 is 1, but the magnitude is represented in 2's complement.



To convert to negative number representation (2's complement), follow these steps:

1. Write the magnitude of the number in 8-bit binary (no sign).

2. Invert each bit.

3. Add 1 to it.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
negative numbers

Example 6-1

Show how the computer would represent –5.

Solution:

```
1. 0000 0101      5 in 8-bit binary
2. 1111 1010      invert each bit
3. 1111 1011      add 1 (hex = FBH)
```

The signed number representation in 2's complement for –5.

The range of byte-sized negative numbers is -1 to -128.

Example 6-2

Show –34H as it is represented internally.

Solution:

```
1. 0011 0100
2. 1100 1011
3. 1100 1100        (which is CCH)
```

Example 6-3

Show the representation for -128_{10}.

Solution:

```
1.   1000 0000
2.   0111 1111
3.   1000 0000
```

Notice that this is not negative zero (–0).

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
negative numbers

Byte-sized signed number ranges.

Decimal	Binary	Hex
–128	1000 0000	80
–127	1000 0001	81
–126	1000 0010	82
...	....	..
–2	1111 1110	FE
–1	1111 1111	FF
0	0000 0000	00
+1	0000 0001	01
+2	0000 0010	02
...	... ...	...
+127	0111 1111	7F

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
word-sized signed numbers

- A word is 16 bits in length in x86 computers.
  - Setting aside the **MSB** (**D15**) for the sign leaves a total of 15 bits (**D14** – **D0**) for the magnitude.
    - A range of -32768 to +32767.

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
sign	magnitude														

- Larger numbers is must be treated as a multiword operand, processed chunk by chunk.
  - The same way as unsigned numbers.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
word-sized signed numbers

The range of signed word operands.

Decimal	Binary	Hex
−32,768	1000 0000 0000 0000	8000
−32,767	1000 0000 0000 0001	8001
−32,766	1000 0000 0000 0010	8002
...	...	...
...	...	...
−2	1111 1111 1111 1110	FFFE
−1	1111 1111 1111 1111	FFFF
0	0000 0000 0000 0000	0000
+1	0000 0000 0000 0001	0001
+2	0000 0000 0000 0010	0002
...	...	...
...	...	...
+32,766	0111 1111 1111 1110	7FFE
+32,767	0111 1111 1111 1111	7FFF

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
word-sized signed numbers

- To convert a negative number to its word operand representation, the same three steps discussed in negative byte operands are used:

1. Write the magnitude of the number in 8-bit binary (no sign).
2. Invert each bit.
3. Add 1 to it.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS signed number operations overflow problem

- When using signed numbers, a serious issue, the overflow problem, must be dealt with.
  - The CPU understands only 0s & 1s, ignoring the human convention of positive/negative numbers.
    - The CPU indicates the problem with the OF (overflow) flag.
- If the result of an operation on signed numbers is too large for the register, an overflow occurs.
  - Review Example 6-4.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS signed number operations overflow problem

Example 6-4

```
DATA1        DB    +96
DATA2        DB    +70
             ...   ....
             MOV   AL,DATA1    ;AL=0110 0000 (AL=60H)
             MOV   BL,DATA2    ;BL=0100 0110 (BL=46H)
             ADD   AL,BL       ;AL=1010 0110 (AL=A6H= 90 invalid!)
+ 96 0110 0000
+ 70 0100 0110
+166 1010 0110
```

According to the CPU, this is 90, which is wrong. (OF = 1, SF = 1, CF = 0)

+96 is added to **+70** and the result according to the CPU is -**90**.

Why? The result was more than **AL** could handle, because like all other 8-bit registers, **AL** could only contain up to +127.

The CPU designers created the overflow flag to inform the programmer that the result of the signed number operation is erroneous.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS the overflow flag in 8-bit operations

- In 8-bit signed number operations, OF is set to 1 if:
  - There is a carry from D6 to D7 but no carry out of D7.
    - (CF = 0)
  - There is a carry from D7 out, but no carry from D6 to D7.
    - (CF = 1)
- The overflow flag is set to 1 if there is a carry from D6 to D7 or from D7 out, but *not* both.
  - If there is a carry both from D6 to D7, and from D7 out, then OF = 0.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS the overflow flag in 8-bit operations

- In Example 6-4, since there is only a carry from D6 to D7 and no carry from D7 out, OF = 1.
  - Examples 6-5, 6-6, and 6-7 give further illustrations of the overflow flag in signed arithmetic.

Example 6-5 Observe the results of the following:

```
            MOV   DL,-128      ;DL=1000 0000 (DL=80H)
            MOV   CH,-2        ;CH=1111 1110 (CH=FEH)
            ADD   DL,CH        ;DL=0111 1110 (DL=7EH=+126 invalid!)

     -128   1000 0000
 +     -2   1111 1110
     -130   0111 1110 OF=1, SF=0 (positive), CF=1
```

According to the CPU, the result is +126, which is wrong. The error is indicated by OF = 1.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS the overflow flag in 8-bit operations

Example 6-6 Observe the results of the following:

```
      MOV   AL,-2        ;AL=1111 1110 (AL=FEH)
      MOV   CL,-5        ;CL=1111 1011 (CL=FBH)
      ADD   CL,AL        ;CL=1111 1001 (CL=F9H=7 which is correct)

   -2  1111 1110
 + -5  1111 1011
   -7  1111 1001
```

OF = 0, CF = 0 , and SF = 1 (negative); the result is correct since OF = 0.

Example 6-7 Observe the results of the following:

```
      MOV   DH,+7        ;DH=0000 0111     (DH=07H)
      MOV   BH,+18       ;BH=0001 0010     (BH=12H)
      ADD   BH,DH        ;BH=0001 1001     (BH=19H=+25, correct)

   +7        0000 0111
 + +18       0001 0010
   +25       0001 1001
```

OF = 0, CF = 0, and SF = 0 (positive).

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
overflow flag in 16-bit operations

- In a 16-bit operation, OF is set to 1 in two cases:
  - A carry from D14 to D15, but no carry out of D15.
    - (CF = 0).
  - A carry from D15 out but no carry from D14 to D15.
    - (CF = 1)
- Again, the overflow flag is low (*not* set) if there is a carry from both D14 to D15 and from D15 out.
  - See examples 6-8 & 6-9.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
overflow flag in 16-bit operations

OF is set to 1 only when there is carry from D14 to D15 or from D15 out, but *not* from both.

Example 6-8

```
    MOV   AX,6E2FH   ; 28,207
    MOV   CX,13D4H   ;+ 5,076
    ADD   AX,CX      ;= 33,283 – expected answer

6E2F       0110  1110  0010  1111
+13D4      0001  0011  1101  0100
8203       1000  0010  0000  0011  = – 32,253 incorrect!
           OF = 1, CF = 0, SF = 1
```

Example 6-9

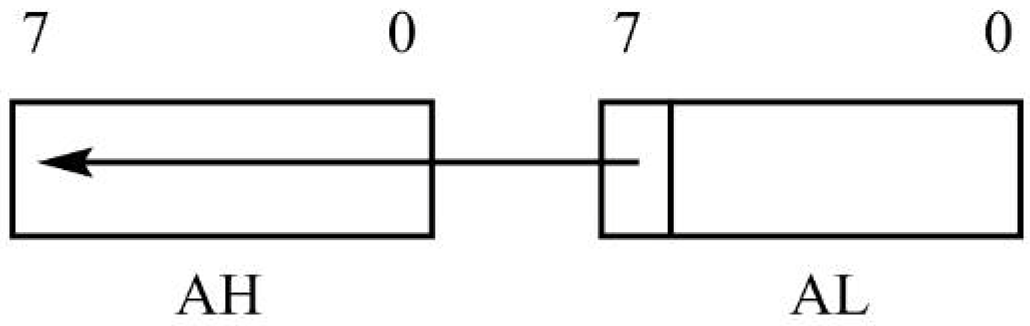
```
    MOV   DX,542FH   ; 21,551
    MOV   BX,12E0H   ; +4,832
    ADD   DX,BX      ;=26,383

543F       0101  0100  0010  1111
+12E0      0001  0010  1110  0000
670F       0110  0111  0000  1111 = 26,383 (correct answer); OF = 0, CF = 0, SF = 0
```

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
avoiding erroneous results

- Sign-extending the operand can avoid problems associated with signed number operations.
  - This copies the sign bit (D7) of the lower byte of a register into the upper bits of the register.
    - Or copies the sign bit of a 16-bit register into another register.
- Two directives are used to perform sign extension:
  - CBW (convert signed byte to signed word)
  - CWD (convert signed word to signed double word)

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
avoiding erroneous results

- CBW will copy D7 (the sign flag) to all bits of AH.
  - The operand is assumed to be AL, and the previous contents of AH are destroyed.



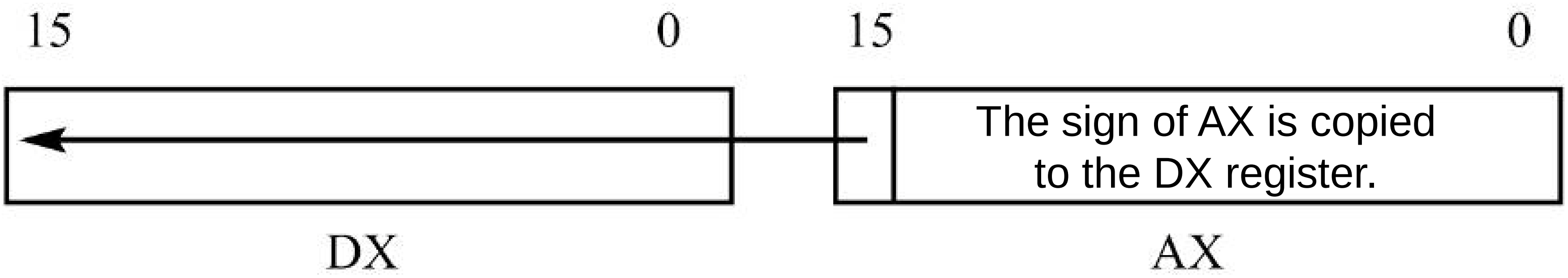

```
MOV    AL,+96        ;AL=0110 0000
CBW                  ;now AH=0000 0000 and AL=0110 0000
```

OR:

```
MOV    AL,-2         ;AL=1111 1110
CBW                  ;AH=1111 1111 and AL=1111 1110
```

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
avoiding erroneous results

- CWD sign-extends AX, copying D15 of AX to all bits of the DX register.
  - This is used for signed word operands.




```
MOV    AX,+260       ;AX=0000 0001 0000 0100 or AX=0104H
CWD                  ;DX=0000H and AX=0104H
```

Another example :

```
MOV    AX,-32766     ;AX=1000 0000 0000 0010B or AX=8002H
CWD                  ;DX=FFFF and AX=8002
```

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
avoiding erroneous results

Example 6-10 shows 6-4 rewritten to correct for overflow.

If the overflow flag is not raised, the result of the signed number is correct & **JNO** (jump if no overflow) will jump to **OVER**.

Example 6-10

Rewrite Example 6-4 to provide for handling the overflow problem.

Solution:

```
DATA1       DB     +96
DATA2       DB     +70
RESULT      DW     ?
            ......
            SUB    AH,AH         ;AH=0
            MOV    AL,DATA1      ;GET OPERAND 1
            MOV    BL,DATA2      ;GET OPERAND 2
            ADD    AL,BL         ;ADD THEM
            JNO    OVER          ;IF OF=0 THEN GO TO OVER
            MOV    AL,DATA2      ;OTHERWISE GET OPERAND 2 TO
            CBW                  ;SIGN EXTEND IT
            MOV    BX,AX         ;SAVE IT IN BX
            MOV    AL,DATA1      ;GET BACK OPERAND 1 TO
            CBW                  ;SIGN EXTEND IT
            ADD    AX,BX         ;ADD THEM AND
OVER:       MOV    RESULT,AX     ;SAVE IT
```

avoiding erroneous results

Example 6-10 shows 6-4 rewritten to correct for overflow.

If OF = 1, result is erroneous & each operand must be sign-extended, then added.

This works for addition of any two signed bytes.

Example 6-10

Rewrite Example 6-4 to provide for handling the overflow problem.

Solution:

```
DATA1      DB     +96
DATA2      DB     +70
RESULT     DW     ?
           ......
           SUB    AH,AH         ;AH=0
           MOV    AL,DATA1      ;GET OPERAND 1
           MOV    BL,DATA2      ;GET OPERAND 2
           ADD    AL,BL         ;ADD THEM
           JNO    OVER          ;IF OF=0 THEN GO TO OVER
           MOV    AL,DATA2      ;OTHERWISE GET OPERAND 2 TO
           CBW                  ;SIGN EXTEND IT
           MOV    BX,AX         ;SAVE IT IN BX
           MOV    AL,DATA1      ;GET BACK OPERAND 1 TO
           CBW                  ;SIGN EXTEND IT
           ADD    AX,BX         ;ADD THEM AND
OVER:      MOV    RESULT,AX     ;SAVE IT
```

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
avoiding erroneous results

- Analysis of the values in Example 6-10.
  - Each is sign-extended and added as follows:

```
S      AH            AL
0      000 0000      0110 0000    +96    after sign extension
0      000 0000      0100 0110    +70    after sign extension
0      000 0000      1010 0110    +166
```

  - If the possibility of overflow exists, byte-sized signed numbers should be sign-extended into a word,
  - Word-sized signed operands should be sign-extended before they are processed

IDIV signed number division

- IDIV means "*integer division*", used for signed number division.
  - All 8088/86 arithmetic instructions are for integer numbers, regardless of whether the operands are signed or unsigned.
    - Real numbers operations are done by the 8087 coprocessor.

Table 6-1: Signed Division Summary

Division	Numerator	Denominator	Quotient	Rem.
byte/byte	AL = byte CBW	register or memory	AL	AH
word/word	AX = word CWD	register or memory	AX	DX
word/byte	AX = word	register or memory	AL[1]	AH
doubleword/word	DXAX = doubleword	register or memory	AX[2]	DX

Notes: 1. Divide error interrupt if –127 > AL > +127. 2. Divide error interrupt if –32,767 > AL > +32,767.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
IMUL signed number multiplication

- Similar in operation to the unsigned multiplication.
  - Operands in signed number operations can be positive or negative, and the result must indicate the sign.

Table 6-2: Signed Multiplication Summary

Multiplication	Operand 1	Operand 2	Result
byte × byte	AL	register or memory	AX[1]
word × word	AX	register or memory	DX AX[2]
word × byte	AL = byte CBW	register or memory	DX AX[2]

Notes: 1. CF = 1 and OF = 1 if AH has part of the result, but if the result is not large enough to need the AH, the sign bit is copied to the unused bits and the CPU makes CF = 0 and OF = 0 to indicate that.
2. CF = 1 and OF = 1 if DX has part of the result, but if the result is not large enough to need the DX, the sign bit is copied to the unused bits and the CPU makes CF = 0 and OF = 0 to indicate that. One can use the J condition to find out which of the conditions above has occurred. The rest of the flags are undefined.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
IMUL signed number multiplication

Program 6-1 computes the average of temperatures:
+13, -10, +19, +14, -18, -9, +12, -19, +16, Celsius.

```
TITLE       PROG 6-1 FIND THE AVERAGE TEMPERATURE
PAGE        60,132
            .MODEL STMALL
            .STACK 64
;-----------------------
            .DATA
SIGN_DAT    DB +13,-10,+19,+14,-18,-9,+12,-1
            ORG 0010H
AVERAGE     DW ?
REMAINDER   DW ?
;-----------------------
      .CODE
MAIN  PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      MOV   CX,9            ;LOAD COUNTER
      SUB   BX,BX           ;CLEAR BX, USED A
      MOV   SI,OFFSET SIGN
```

Each data byte was sign-extended and added to BX, computing the sum, which is a signed word.

The sum & count were sign-extended, and by dividing the total sum by the count (9), the average was calculated.

See the entire program listing on page 182 of your textbook.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS

SAR/SAL

- SAR destination, count
  - As the bits of the destination are shifted to the right into CF, the empty bits are filled with the sign bit.

```
MOV    AL,-10        ;AL=-10=F6H=1111 0110
SAR    AL,1          ;AL is arithmetic shifted right once
                     ;AL=1111 1011=FDH=-5
```

Example 6-11 Using DEBUG, evaluate the results of the following:

```
MOV    AX,-9
MOV    BL,2
IDIV   BL            ;divide -9 by 2 results in FCH
MOV    AX,-9
SAR    AX,1          ;divide -9 by 2 with arithmetic shift
                     ;results in FBH
```

Solution:

The DEBUG trace demonstrates that an IDIV of –9 by 2 gives FCH (– 4), whereas SAR –9 gives FBH (–5). This is because SAR rounds negative numbers down but IDIV rounds up.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
SAR/SAL

- SAL (shift arithmetic left) and SHL (shift left) do exactly the same thing.
  - Basically the same instruction with two mnemonics.
  - As far as signed numbers are concerned, there is no need for SAL.

6.1: SIGNED NUMBER ARITHMETIC OPERATIONS
CMP signed number comparison

- CMP dest, source
  - The same for both signed and unsigned numbers.
    - The J condition instruction used to make a decision is different from that used for the unsigned numbers.
- In unsigned number comparisons CF and ZF are checked for conditions of larger, equal, and smaller.
  - In signed number comparison, OF, ZF, SF are checked.

```
destination > source    OF=SF or ZF=0
destination = source    ZF=1
destination < source    OF=negation of SF
```

CMP signed number comparison

Mnemonics used to detect the conditions above:

```
JG    Jump Greater              jump if OF=SF or ZF=0
JGE   Jump Greater or Equal     jump if OF=SF
JL    Jump Less                 jump if OF=inverse of SF
JLE   Jump Less or Equal        jump if OF=inverse of SF or ZF=1
JE    Jump if Equal             jump of ZF = 1
```

– Example 6-12 on page 185 should help clarify how the condition flags are affected by the compare instruction.

– Program 6-2 on page 186 is an example of the application of the signed number comparison.

- It uses the data in Program 6-1, and finds the lowest temperature.

6.2: STRING AND TABLE OPERATIONS

- String instructions in the x86 family are capable of operations on a series of operands located in consecutive memory locations.
- While CMP can compare only 2 bytes (or words) of data, CMPS (compare string) compares two arrays of data located in memory locations.
  - Pointed at by the SI and DI registers.

6.2: STRING AND TABLE OPERATIONS
SI & DI, DS & ES in string instructions

- For string operations to work, CPU designers set aside certain registers for specific functions, to permanently provide source/destination operands.
  - In 8088/86 processors, SI & DI registers always point to the source and destination operands, respectively.
- To generate the physical address, 8088/86 always uses SI as the offset of the DS (data segment) register. and DI as the offset of ES (extra segment).
  - The ES register must be initialized for the string operation to work.

6.2: STRING AND TABLE OPERATIONS
byte/word operands in string instructions

The operand can be a byte or a word, distinguished by letters B (byte) & W (word) in the mnemonic.

Table 6-3: String Operation Summary

Instruction	Mnemonic	Destination	Source	Prefix
move string byte	MOVSB	ES:DI	DS:SI	REP
move string word	MOVSW	ES:DI	DS:SI	REP
store string byte	STOSB	ES:DI	AL	
store string word	STOSW	ES:DI	AX	REP
load string byte	LODSB	AL	DS:SI	none
load string word	LODSW	AX	DS:SI	none
compare string byte	CMPSB	ES:DI	DS:SI	REPE/REPNE
compare string word	CMPSW	ES:DI	DS:SI	REPE/REPNE
scan string byte	SCASB	ES:DI	AL	REPE/REPNE
scan string byte	SCASW	ES:DI	AX	REPE/REPNE

6.2: STRING AND TABLE OPERATIONS
DF, the direction flag

- To process operands in consecutive locations requires the pointer to be incremented/decremented.
  - In string operations, achieved by the direction flag.
    - Flag register bit 11 (D10) is set aside for the direction flag (DF).
- The programmer to specifes the choice of increment or decrement by setting the direction flag high or low.
  - CLD (clear direction flag) will reset (zeroes) DF, telling the string instruction to increment the pointers automatically.
    - Sometimes is referred to as *autoincrement*.
  - STD (set the direction flag) performs the opposite function.
    - Sets DF to 1, indicating that the pointers SI and DI should be decremented automatically.

6.2: STRING AND TABLE OPERATIONS
REP prefix

- The REP (repeat) prefix allows a string instruction to perform the operation repeatedly.
  - REP assumes that CX holds the number of times the instruction should be repeated. (until CX becomes zero)
- In Example 6-13, after transfer of every byte by the MOVSB instruction, both the SI and DI registers are incremented automatically once only (notice CLD).
  - The REP prefix causes the CX counter to decrement, and MOVSB is repeated until CX becomes zero.
    - Both DS and ES are set to the same value.

6.2: STRING AND TABLE OPERATIONS
REP prefix

Example 6-13 Using string instructions, write a program to transfer a block of 20 bytes of data.

Solution:

```
;in the data segment:
DATA1 DB          'ABCDEFGHIJKLMNOPQRST'
      ORG  30H
DATA2 DB          20 DUP (?)
;in the code segment:
     MOV   AX,@DATA
     MOV   DS,AX              ;INITIALIZE THE DATA SEGMENT
     MOV   ES,AX              ;INITIALIZE THE EXTRA SEGMENT
     CLD                      ;CLEAR DIRECTION FLAG FOR AUTOINCREMENT
     MOV   SI,OFFSET DATA1    ;LOAD THE SOURCE POINTER
     MOV   DI,OFFSET DATA2    ;LOAD THE DESTINATION POINTER
     MOV   CX,20              ;LOAD THE COUNTER
     REP   MOVSB              ;REPEAT UNTIL CX BECOMES ZERO
```

6.2: STRING AND TABLE OPERATIONS
REP prefix

- An alternative solution would change only two lines of code:

```
MOV   CX,10
REP   MOVSW
```

  - MOVSW will transfer a word (2 bytes) at a time and increment SI & DI registers each twice.
    - REP will repeat that process until CX becomes zero.
    - CX has the value of 10 in it, as 10 words is equal to 20 bytes.

6.2: STRING AND TABLE OPERATIONS
STOS instruction

- STOSB stores the byte in register AL into memory locations pointed at by ES:DI.
  - If DF = 0, then DI is *incremented once*.
  - If DF = 1, then DI is decremented.
- STOSW stores the contents of AX in ES:DI and ES:DI+1 (AL into ES:DI and AH into ES:DI+1)
  - If DF = 0, then DI is *incremented twice*.
  - If DF = 1, then DI is decremented twice

6.2: STRING AND TABLE OPERATIONS
LODS instructions

- LODSB loads the contents of memory locations pointed at by DS:SI into AL.
  - Increments SI once, if DF = 0.
  - Decrements SI once, if DF = 1.
- LODSW loads the contents of memory locations pointed at by DS:SI into AL, and DS:SI+1 into AH.
  - SI is incremented *twice* if DF = 0, else it is decremented twice.
- LODS is *never* used with a REP prefix.

6.2: STRING AND TABLE OPERATIONS
testing memory using STOSB and LODSB

- Example 6-14 on page 189 uses string instructions STOSB & LODSB to test an area of RAM memory.
  - First AAH is written into 100 locations by using word-sized operand AAAAH, and a count of 50.
  - In the test, LODSB brings the contents of memory locations into AL, one by one, and each is eXclusive-ORed with AAH. (register AH has hex value AA).
    - If they are the same, ZF = 1 and the process is continued.
    - Otherwise, the pattern written there by the previous routine is not there and the program will exit.

6.2: STRING AND TABLE OPERATIONS
testing memory using STOSB and LODSB

Example 6-14

Solution:

Assuming that ES and DS have been assigned in the ASSUME directive, the following is from the code segment:

```
      ;PUT PATTERN AAAAH IN TO 50 WORD LOCATIONS
      MOV   AX,DTSEG              ;INITIALIZE
      MOV   DS,AX                 ;DS REG
      MOV   ES,AX                 ;AND ES REG
      CLD                         ;CLEAR DF FOR INCREMENT
      MOV   CX,50                 ;LOAD THE COUNTER (50 WORDS)
      MOV   DI,OFFSET MEM_AREA    ;LOAD THE POINTER FOR DESTINATION
      MOV   AX,0AAAAH             ;LOAD THE PATTERN
      REP   STOSW                 ;REPEAT UNTIL CX=0
      ;BRING IN THE PATTERN AND TEST IT ONE BY ONE
      MOV   SI,OFFSET MEM_AREA    ;LOAD THE POINTER FOR SOURCE
      MOV   CX,100                ;LOAD THE COUNT (COUNT 100 BYTES)
```

See the entire program listing on page 189 of your textbook.

6.2: STRING AND TABLE OPERATIONS
REPZ and REPNZ prefixes

- Used with CMPS & SCAS for testing purposes.
  - **REPZ (repeat zero)** - the same as REPE (repeat equal), will repeat the string operation as long as the source and destination operands are equal (ZF = 1) or until CX becomes zero.
  - **REPNZ (repeat not zero**) - the same as REPNE (repeat not equal), will repeat the string operation as long as the source and destination operands are not equal (ZF = 0) or until CX becomes zero.
  - **CMPS (compare string)** - allows the comparison of two arrays of data pointed at by registers SI & DI.
    - CMPS can test inequality of two arrays using "REPNE CMPSB".

6.2: STRING AND TABLE OPERATIONS
SCASB (scan string)

- SCASB compares each byte of the array pointed at by ES:DI with the contents of the AL register.
  - Depending on which prefix, REPE or REPNE, is used, a decision is made for equality or inequality.
- In Example 6-16, on page 191, the letter "G" is compared with "M".
  - Since they are not equal, DI increments, CX decrements
    - Scanning is repeated until letter "G" is found or the CX register is zero.
- SCASB can search for a character in an array & if found, it will be replaced with the desired character.

6.2: STRING AND TABLE OPERATIONS
XLAT instruction and look-up tables

- A table is commonly referred to as a *look-up table*.
  - To access elements of a table, 8088/86 processors provide the XLAT (translate) instruction.
- Assume a need for a table for the values of x^2, where x is between 0 and 9.
  - First the table is generated and stored in memory:

```
SQUR_TABLE   DB     0,1,4,9,16,25,36,49,64,81
```

  - To access the square of any number from 0 to 9, by use of XLAT, register BX must have the offset address of the look-up table, and the number whose square is sought must be in the register AL.
    - After XLAT execution, AL will have the square of the number.

6.2: STRING AND TABLE OPERATIONS
XLAT instruction and look-up tables

- To get the square of 5 from the table:

```
MOV   BX,OFFSET SQUR_TABLE ;load the offset address of table
MOV   AL,05          ;AL=05 will retrieve 6th element
XLAT                 ;pull the element out of table
                     ;and put in AL
```

  - After execution AL will have 25 (19H), the square of 5.

- XLAT is one instruction, equivalent to the following:

```
SUB   AH,AH        ;AH=0
MOV   SI,AX        ;SI=000X
MOV   AL,[ BX+SI]  ;GET THE SIth ENTRY FROM BEGINNING
                   ;OF THE TABLE POINTED AT BY BX
```

6.2: STRING AND TABLE OPERATIONS
code conversion using XLAT

- XLAT can translate the hex keys of non-ASCII keyboards to ASCII.
  - Assuming keys are 0–F, the following is the program to convert the hex digits of 0–F to their ASCII equivalents.

```
;data segment:
ASC_TABL     DB     '0','1','2','3','4','5','6','7','8'
             DB     '9','A','B','C','D','E','F'
HEX_VALU     DB     ?
ASC_VALU     DB     ?
;code segment:
             MOV    BX,OFFSET ASC_TABL ;BX= TABLE OFFSET
             MOV    AL,HEX_VALU        ;AL=THE HEX DATA
             XLAT                 ;GET THE ASCII EQUIVALENT
             MOV    ASC_VALU,AL        ;MOVE IT TO MEMORY
```

Dec Hex Bin
6 6 00000110

ENDS ; SIX

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY

The x86 PC
assembly language, design, and interfacing
fifth edition

Prentice Hall