Chapter 5 Arithmetic and Logic Instructions

Assoc. Prof. Dr. Gazi Erkan BOSTANCI

Slides are mainly based on The Intel Microprocessors by Barry B. Brey, 2008

- ADD: Addition instruction for 8, 16 and 32 bit binary addition.

```
ADD AL, BL ; AL = AL + BL
ADD CL, 44H; CL = CL + 44H
ADD [BX], AL ; AL adds to the byte contents of the
data segment memory location addressed by BX with the
sum stored in the same location
```

- Register Addition

```
ADD AX, BX
ADD AX, CX
ADD AX, DX
```

16 bit addition

❖ Whenever artihmetic and logic instructions execute, the contents of the flag register change (only the rightmost 8 bits and the overflow change, interrupt, trap and other flags donot change with these instructions.)

- Immediate Addition

```
MOV DL, 12H
ADD DL, 33H
```

- After the addition, the sum (45h) moves into register DL and flags change as
  - Z=0 (result is not zero)
  - C=0 (no carry)
  - A=0 (no half-carry) ★Half carry is only for the least significant 4 bits.
  - S=0 (result positive)
  - P=0 (odd parity)
  - O=0 (no overflow)

- Memory to register addition

```
MOV DI, OFFSET NUMB; address NUMB
MOV AL, 0; clear sum
ADD AL, [DI]; add NUMB
ADD AL, [DI+1]; add NUMB+1
```

- Addition of two consecutive bytes stored at data segment offset locations NUMB, NUMB+1.

- Add Sample

- Increment Addition (INC): Adds 1 to a register or memory location.
  - INC BL; BL = BL+1
  - INC BYTE PTR[BX]; Adds 1 to the byte contents of the data segment memory location addressed by BX
  - INC DATA1; Adds 1 to the contents of data segment memory location DATA1

❖INC does not affect the carry flag bit

- Further to the example in the previous slide

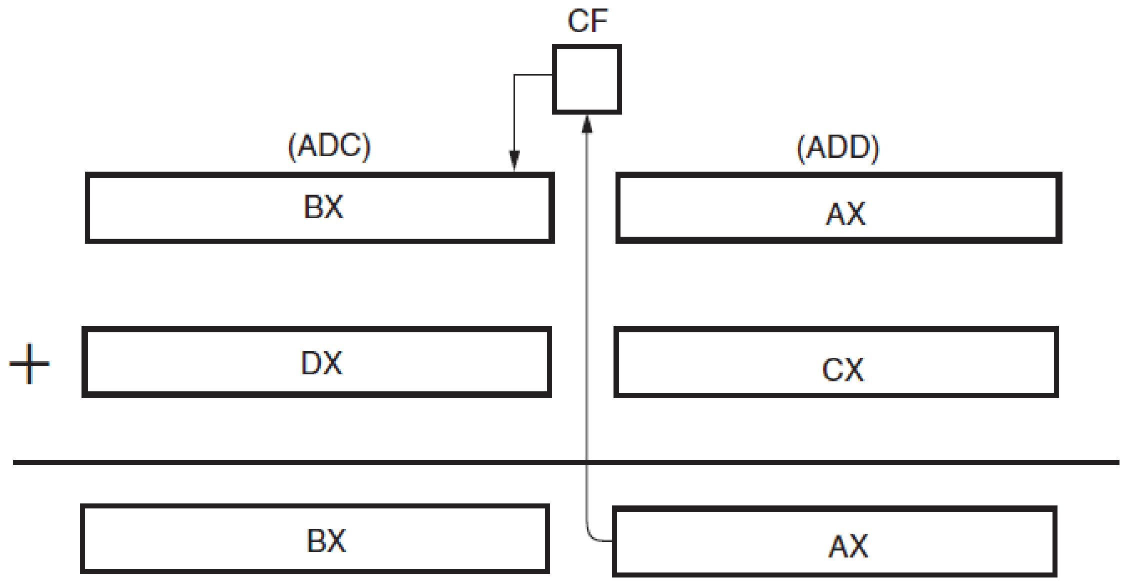
```
MOV DI, OFFSET NUMB; address NUMB
MOV AL, 0; clear sum
ADD AL, [DI]; add NUMB
INC DI; increment DI
ADD AL, [DI]; add NUMB+1
```

- INC Sample

- ADC: Addition-with-carry adds the bit in the carry flag (C) to the operand data.
  - This operation is used to add numbers wider than 16 bits in 8086.

```
ADC AL, AH; AL = AL + AH+ carry
ADC CX, BX; CX = CX + BX + carry
```

- ADC is used to perform 32 bit addition from two 16 bit additions using the carry flag.
  - `ADD AX, CX`
  - `ADC BX, DX`



- ADC Sample

- SUB: Subtraction instruction for 8, 16 or 32 bit data.

```
SUB CL, BL; CL = CL – BL

SUB [DI], CH; Subtracts CH from tge byte contents of
the data segment addressed by DI and stores the
difference in the same memory location.

SUB DH, 6FH; DH = DH- 6FH
```

- Register Subtraction

```
SUB BX, CX
SUB BX, DX
```

- Immediate Subtraction

```
MOV CH, 22H
SUB CH, 44H
```

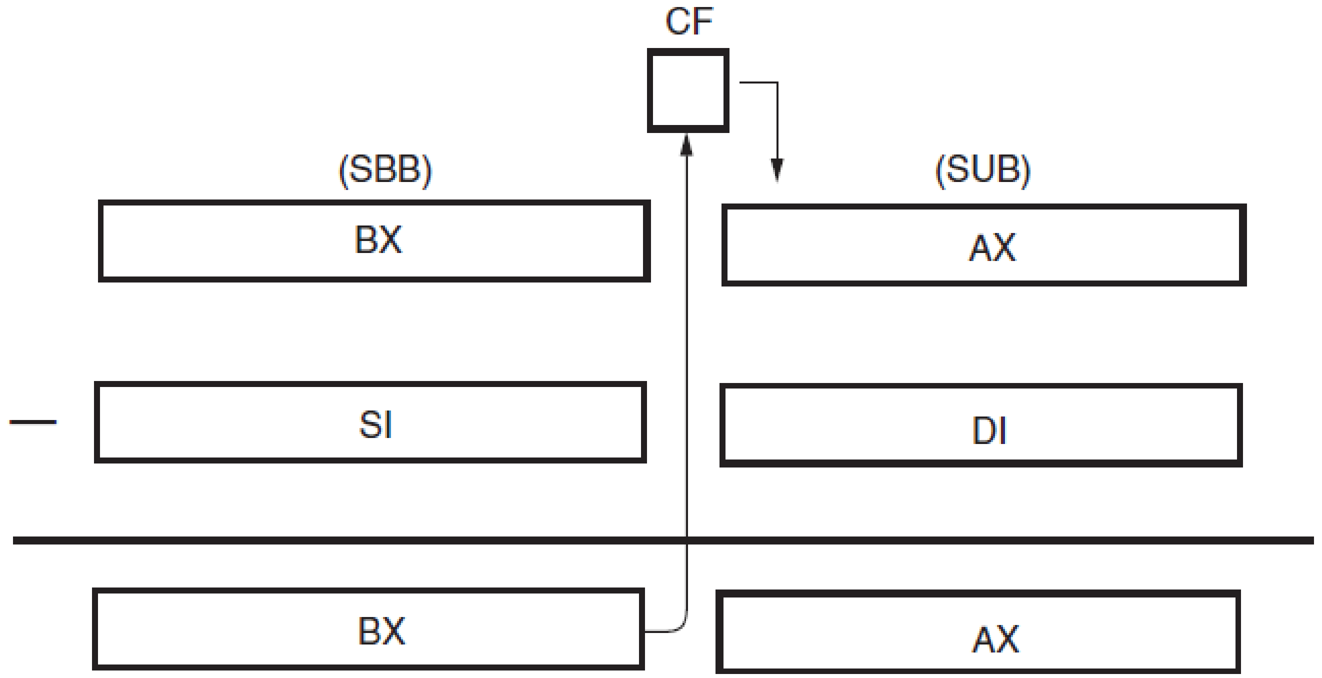
- After the subtraction (DEH) moves into CH register and the flags are as follows:
  - Z=0 (result is not zero)
  - C=1 (borrow)
  - A=1 (half-borrow)
  - S=1 (result negative)
  - P=1 (even parity)
  - O=0 (no overflow)

- Decrement Subtraction (DEC): Subtracts 1 from a register or contents of a memory location.
  - DEC BH; BH = BH – 1

- SBB: Subtraction with borrow functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference.

```
SBB AH, AL; AH = AH - AL – carry
```

```
SUB AX, DI
SBB BX, SI
```



- Comparison (CMP): This is a subraction tha changesonly the flag bits; the destination operand is never changed. It is normally followed by a conditional jump, which tests the condition of the flag bits.
  - JA (Jump above), JB (Jump below),
  - JAE (Jump above or equal), JBE (Jump below or equal)

```
CMP CL, BL; CL-BL
```

Ex.

```
CMP  AL, 10H; compare AL against 10H
JAE FUNC; if AL is 10H or above, jump to FUNC.
```

- Multiplication: This operation can be performed on bytes, words or double words and can be signed integer (IMUL) or unsigned integer (MUL).
- Flag bits change after a multiplication, but we only use C and O bits.
  - The rest of the bits are unpredictable.
- In 8 bit multiplication , if the most significant 8 bits of the result are zero, both C and O flag bits equal to zero.
  - These bits show that the result is 8 bits wide (C=0) or 16 bits wide (C=1)

- 8 bit multiplication

```
MUL CL; AL is multiplied by CL, the unsigned product
is in AX
IMUL DH; AL is multiplied by DH, the signed product is
in AX
```

- Ex.

```
MOV BL, 5; load data
MOV CL, 10
MOV AL, CL; position data
MUL BL; multiply
MOV DX, AX; position product
```

- 16 bit multiplication
  - This is very similar to 8 bit multiplication. The difference is AX stores the multiplicand instead of AL and the 32 bit product is stored in DX-AX.
    - DX stores the most significant 16 bits and AX stores the least significant 16 bits.

```
MUL CX; AX is multiplied by CX, the unsigned product is in DX-AX

IMUL DI; AX is multiplied by DI, the signed product is in DX-AX

MUL WORD PTR[SI]; AX is multiplied by the word contents of the
data segment memory location addressed by SI, the unsigned
product is stored in DX-AX
```

- MUL Sample

- Division: This operation can be performed using 8 bit or 16 bit numbers.
- None of the flag bits change predictably for a division.
- A division can result in different types of errors: divide by zero and divide overflow.
  - For example, assume AX = 3000 and this is divided by 2. Since the quotient for an 8 bit division appears in AL, the result of 1500 causes a divide overflow. Because it does not fit in AL.

❖ In most systems, a divide error interrupt display an error message.

- 8 bit division divides a 16 bit number by an 8 bit number.
- 16 bit division divides a 32 bit number by a 16 bit number.

- 8 bit division: Uses AX for the dividend that is divided by the contents of any 8 bit register or memory location. The quotient moves into AL with AH containing the whole number remainder.
  - If AX=0010H (+16) and BL = 0FDH (-3) and IDIV BL executes, then AX=01FBH. This represents the quotient of -5 (AL) and remainder of 1 (AH).

```
DIV CL; AX is divided by CL, the unsigned quotient is in
AL and the unsigned remainder is in AH.
```

❖With 8 bit division, the numbers are usually 8 bits wide. This means the dividend must be converted to a 16 bit number in AX. CBW (convert byte to word) performs this conversion for signed numbers. For unsigned, we can zero extend the number by clearing AH.

```
MOV AL, NUMB; get NUMB
MOV AH, 0; zero extend
DIV NUMB1; divide by NUMB1
MOV ANSQ, AL; save quotient
MOV ANSR, AH; save remainder
```

CBW Operation

```
if high bit of AL = 1 then
  AH = 255 (0FFh)
else
  AH = 0
```

- 16 bit division: Similar to 8 bit division, except that instead of dividing into AX, the 16 bit number is divided into DX-AX, a 32 bit dividend. The quotient appears in AX and the remainder in DX:

```
DIV CX; DX-AX is divided by CX, the unsigned quotient
is in AX and the unsigned remainder is in DX.
```

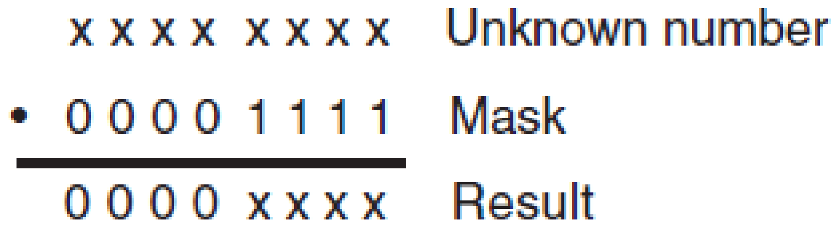
❖If AX is a 16 bit signed number, CWD (convert word to double) sign extends it into a 32 bit number. For unsigned numbers, zero extension is used.

```
MOV AX, -100; load -100
MOV CX, 9; load 9
CWD; sign extend, AX is extended to DX-AX
IDIV CX
```

```
CWD Operation
if high bit of AX = 1
then
  DX = 65535 (0FFFFh)
else
  DX = 0
```

Basic Logic Operations

- AND: Performs logical multiplication.
- It can be used to clear bits of a binary number.
    - This process is called masking.



```
AND AL, BL; AL = AL AND BL
AND CX, DX; CX = CX AND DX
```

- An ASCII-coded number can be converted to BCD by using the AND instruction to mask off the leftmost four binary bit positions. This converts the ASCII 30H to 39H to 0–9. Program below converts the ASCII contents of BX into BCD. The AND instruction in this example converts two digits from ASCII to BCD simultaneously.

```
MOV BX, 3135H; load ASCII
AND BX, 0F0FH; mask BX
```

- AND Sample

- OR: Performs logical addition.
- It can be used to set bits of a binary number.

```
  x x x x  x x x x   Unknown number
+ 0 0 0 0  1 1 1 1   Mask
-------------------
  x x x x  1 1 1 1   Result
```

```
OR AL, BL; AL = AL OR BL
OR DX, [BX]; DX is Ored with the word contents
of data segment memory location addressed by BX.
```

- XOR: Performs exclusive-OR operation.
- It is used to invert some of the bits in a binary number.
  - XORing with 1, inverts
  - XORing with 0, keeps

$$\begin{array}{rl} \text{x x x x x x x x} & \text{Unknown number} \\ \oplus\, 0\,0\,0\,0\;1\,1\,1\,1 & \text{Mask} \\ \hline \text{x x x x}\;\overline{\text{x}}\,\overline{\text{x}}\,\overline{\text{x}}\,\overline{\text{x}} & \text{Result} \end{array}$$

```
XOR CH, DL; CH = CH XOR DL
OR CX, 0600H; set bits 9 and 10
AND CX, 0FFFCH; clear bits 0 and 1
XOR CX, 1000H; invert bit 12
```

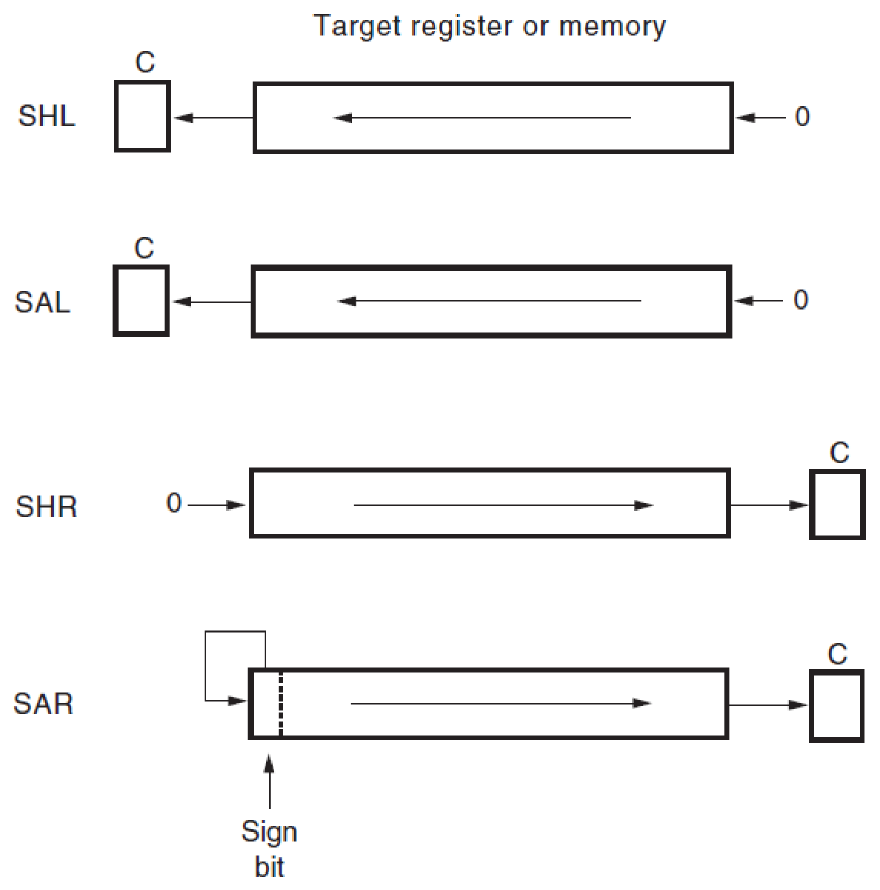
- TEST: This instruction performs the AND operation, but it does not change the destination operand. It affects the condition of the flag register.
  - It is very similar to CMP but it checks a single bit where CMP is used to check a byte, word or doubleword.
- Zero flag (Z) is logic 1 if the bit under test is a zero and Z=0 if the bit is not zero.
- It is followed by a JZ (jump if zero) or JNZ (jump if not zero) instruction.

```
TEST AL, 1; test right bit
JNZ RIGHT; if set
TEST AL, 128; test left bit
JNZ LEFT; if set
```

- NOT and NEG
  - NOT is logical inversion or one’s complement
  - NEG is arithmetic sign inversion or two’s complement

  - NOT CH; CH is one’s complemented
  - NEG AX; AX is two’s complemented

- Shift: Move bit to left or right within a register or memory location.



- Left shift->multiplication by powers of 2^{+n}
- Right shift->multiplication by powers of 2^{-n}

```
SHL AX, 1; AX is shifted left 1 place
SHR BX, 12; BX is logically shifted right 12 places
```

❖Logical shift is for unsigned data.

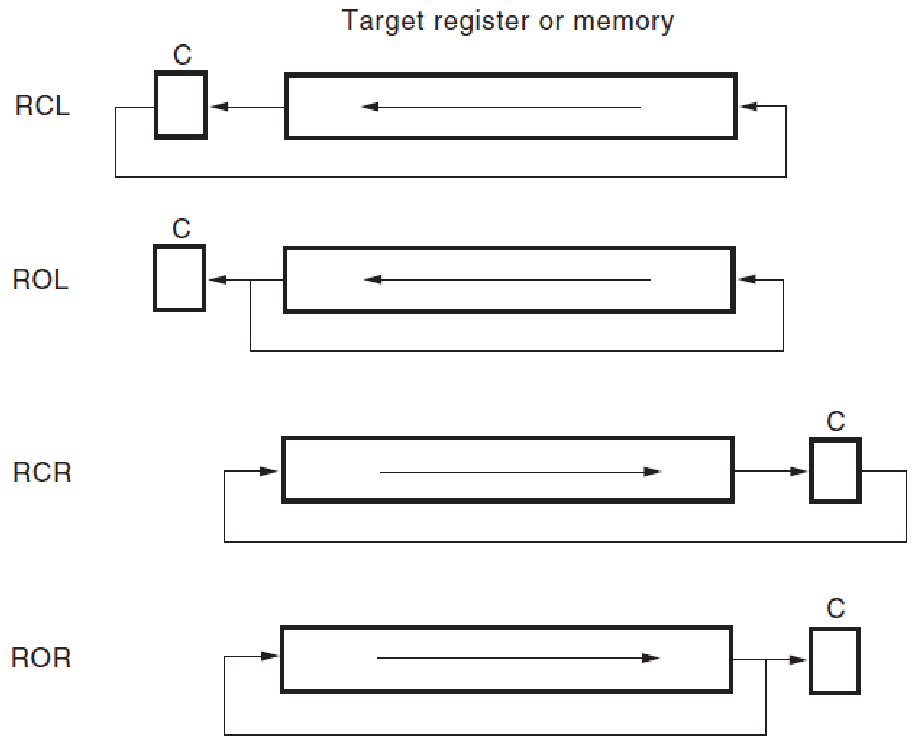
❖Arithmetic shift is for signed data.

- Rotate: Rotates the information in a register or memory location either from one end to another or through carry flag.

```
ROL SI, 14; SI rotates left 14 places
RCL BL, 6; BL rotates left through
carry 6 places.
```

- These instructions often used to shift wide numbers to the left or right.
- The program below shifts the 48-bit number in registers DX, BX, and AX left one binary place. Notice that the least significant 16 bits (AX) shift left first. This moves the leftmost bit of AX into the carry flag bit. Next, the rotate BX instruction rotates carry into BX, and its leftmost bit moves into carry. The last instruction rotates carry into DX, and the shift is complete.

```
SHL AX, 1
RCL BX, 1
RCL DX, 1
```


Target register or memory
C
RCL
C
ROL
C
RCR
C
ROR