Instruction Set of 8086

Dr. Manju Khurana
Assistant Professor, CSED
TIET, Patiala
manju.khurana@thapar.edu

Instruction Set of 8086

- An instruction is a binary pattern designed inside a microprocessor to perform a specific function.
- The entire group of instructions that a microprocessor supports is called **Instruction Set**.
- 8086 has more than **20,000** instructions.

Classification of Instruction Set

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- Program Execution Transfer Instructions
- String Instructions
- Processor Control Instructions

Data Transfer Instructions

- These instructions are used to transfer data from source to destination.
- The operand can be a constant, memory location, register or I/O port address.

Data Transfer Instructions

- **MOV Des, Src:**

MOV Destination, Source
Moves a byte/word **from** the **source to** the **destination** specified in the instruction.
Source: *Register, Memory Location, Immediate Number*
Destination: Register, Memory Location
Both, source and destination cannot be memory locations.

```
Eg: MOV CX, 0037H      ; CX ← 0037H
    MOV BL, [4000H]    ; BL ← DS:[4000H]
    MOV AX, BX         ; AX ← BX
    MOV DL, [BX]       ; DL ← DS:[BX]
    MOV DS, BX         ; DS ← BX
```

Data Transfer Instructions

- **PUSH Operand:**
  - It pushes the operand into top of stack.
  - E.g.: PUSH BX
- **POP Des:**
  - It pops the operand from top of stack to Des.
  - **Des can be a general purpose register, segment register (except CS) or memory location.**
  - E.g.: POP AX

Data Transfer Instructions

PUSH Source

Push the **source** (word) **into** the **stack** and decrement the stack pointer by two.

The source MUST be a WORD (**16 bits**).

Source: *Register, Memory Location*

Eg: **PUSH CX** ; *SS:[SP-1]* ← *CH, SS:[SP-2]* ← *CL*
; *SP* ← *SP – 2*

PUSH DS ; *SS:[SP-1, SP-2]* ← *DS*
; *SP* ← *SP – 2*

POP a word from the stack into the given destination and increment the Stack Pointer by 2.

The destination MUST be a WORD (16 bits).

Destination: *Register [EXCEPT CS], Memory Location*

Eg: **POP CX** ; *CL* ← *SS:[SP], CH* ← *SS:[SP+1]*
; *SP* ← *SP + 2*

POP DS ; *DS* ← *SS:[SP, SP+1]*
; *SP* ← *SP + 2*

Data Transfer Instructions

Note :- MOV, PUSH, POP are the only instructions that use the Segment Registers as operands except CS.

Data Transfer Instructions

- **XCHG Des, Src:**
  - This instruction exchanges Src with Des.
  - It cannot exchange two memory locations directly.
  - E.g.: XCHG DX, AX

    XCHG BL,CH

XCHG Destination, Source
Exchanges a byte/word between the **source and** the **destination** specified in the instruction.
Source: *Register, Memory Location*
Destination: Register, Memory Location
Even here, both operands cannot be memory locations.
Eg: **XCHG CX, BX** ; *CX ↔ BX*
XCHG BL, CH ; *BL ↔ CH*

Data Transfer Instructions

- **IN Accumulator, Port Address:**

IN destination register, source port
Loads the **destination register with** the contents of the **I/O port** specified by the source.
Source: 8-bit address of I/O Port **[Direct addressing]** Or: DX register **[Indirect addressing]**
Destination: AL/ AH for 8-bit Port (NOT necessarily 8-bit port address),
AX for a 16-bit Port

```
Eg: IN AL, 80H      ; AL ← [80H]I/O as I/O Port 80H is an 8-bit Port
    IN AX, 80H      ; AX ← [80H]I/O as I/O Port 80H is an 16-bit Port
    IN AL, DX       ; AL ← [DX]I/O as I/O Port pointed by DX is an 8-bit Port
    IN AX, DX       ; AX ← [DX]I/O as I/O Port pointed by DX is a 16-bit Port
```

- **OUT Port Address, Accumulator:**

OUT destination port, source register
Loads the destination I/O port with the contents of the source register.

```
Eg: OUT 80H, AL     ; [80H]I/O ← AL as I/O Port 80H is an 8-bit Port
    OUT 80H, AX     ; [80H]I/O ← AX as I/O Port 80H is an 16-bit Port
    OUT DX, AL      ; [DX]I/O  ← AL as I/O Port pointed by DX is an 8-bit Port
    OUT DX, AX      ; [DX]I/O  ← AX as I/O Port pointed by DX is a 16-bit Port
```

Data Transfer Instructions

- **LEA Register, Src:**

LEA register, source

Loads Effective Address (offset address) **of** the **source into** the **given register.**

Eg: LEA BX, Total ; *BX ← Effective Address (offset address) of Total in Data Segment.*

Data Transfer Instructions

- **LDS Des, Src:**

LDS destination register, source

Loads the **destination register and DS** register **with** the **offset** address **and** the **segment address** indirectly specified by the **source.**

Eg: **LDS BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*
; *DS ← {DS: [Total + 2], DS:[Total + 3]}*

Data Transfer Instructions

- **LDS Des, Src:**
- This instruction loads new values into the specified register and into the DS register from four successive memory locations.
- The word from two memory locations is copied into the specified register and the word from the next two memory locations is copied into the DS registers.
- LDS does not affect any flag.

Data Transfer Instructions

- **LES Des, Src:**
  - It loads 32-bit pointer from memory source to destination register and ES.
  - The offset is placed in the destination register and the segment is placed in ES.
  - This instruction is very similar to LDS except that it initializes ES instead of DS.
  - E.g.: same as LDS (except segment is placed in ES.)

LES destination register, source

Loads the **destination register and ES** register with the **offset address and the segment address** indirectly specified by the **source.**

Eg: **LES BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*
; *ES ← {DS: [Total + 2], DS:[Total + 3]}*

Data Transfer Instructions

- **LAHF:**
  - It copies the lower byte of flag register to AH.
- **SAHF:**
  - It copies the contents of AH to lower byte of flag register.
- **PUSHF:**
  - Pushes flag register to top of stack.
- **POPF:**
  - Pops the stack top to flag register.

The XLAT Instruction

Mnemonic	Meaning	Format	Operation	Flags
XLAT	Translate	XLAT	((AL)+(BX)+(DS)0) → (AL)	None

Example:

Assume (DS) = 0300H, (BX)=0100H, and (AL)=0DH

XLAT replaces contents of AL by contents of memory location with

PA=(DS)0 +(BX) +(AL)

= 03000H + 0100H + 0DH = 0310DH

Thus

(0310DH) → (AL)

XLAT

- **Moves into AL, Content of mem. Loc. In DS.**
- **whose EA is formed by sum of BX and AL.**

AL ← DS:[BX + AL]
i.e. if DS = 1000H; BX = 0200H; AL = 03H

```
∴ 10000    ... DS × 16
+  0200    ... BX
+    03    ... AL
=10203  ∴ AL ← [10203H]
```


Thank you!

Instruction Set of 8086

Dr. Manju Khurana
Assistant Professor, CSED
TIET, Patiala
manju.khurana@thapar.edu

Arithmetic Instructions

- **ADD Des, Src:**
  - It adds a byte to byte or a word to word.
  - It effects AF, CF, OF, PF, SF, ZF flags.

ADD/ADC destination, source
Adds the source to the destination and stores the **result** back **in** the **destination.**
Source: Register, Memory Location, Immediate Number
Destination: Register
Both, source and destination have to be of the same size.
ADC also adds the carry into the result.

```
Eg: ADD AL, 25H      ; AL ← AL + 25H
    ADD BL, CL       ; BL ← BL + CL
    ADD BX, CX       ; BX ← BX + CX
    ADC BX, CX       ; BX ← BX + CX + Carry Flag
```

Arithmetic Instructions

- **ADC Des, Src:**
  - It adds the two operands with CF.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - E.g.:
    - ADC AL, 74H
    - ADC DX, AX
    - ADC AX, [BX]

Arithmetic Instructions

- **SUB Des, Src:**
  - It subtracts a byte from byte or a word from word.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - For subtraction, CF acts as borrow flag.
  - E.g.:
    - SUB AL, 74H
    - SUB DX, AX
    - SUB AX, [BX]

Arithmetic Instructions

- **SBB Des, Src:**
  - It subtracts the two operands and also the borrow from the result.
  - It effects AF, CF, OF, PF, SF, ZF flags.
  - E.g.:
    - SBB AL, 74H
    - SBB DX, AX
    - SBB AX, [BX]

Arithmetic Instructions

- **INC Src:**
  - It increments the byte or word by one.
  - The operand can be a register or memory location.
  - It effects AF, OF, PF, SF, ZF flags.
  - CF is not effected.

INC destination
Adds 1 to the specified **destination.**
Destination: Register, Memory Location
Note: Carry Flag is NOT affected.

```
Eg: INC AX              ; AX ← AX + 1
    INC BL              ; BL ← BL + 1
    INC BYTE PTR [BX]; Increment the byte pointed by BX in the Data Segment
                        ; i.e. DS:[BX] ← DS;[BX] + 1
    INC WORD PTR [BX]; Increment the word pointed by BX in the Data Segment
                        ; i.e. {DS:[BX], DS:[BX+1]} ← {DS:[BX], DS:[BX+1]} + 1
```

Arithmetic Instructions

- **DEC Src:**
  - It decrements the byte or word by one.
  - The operand can be a register or memory location.
  - It effects AF, OF, PF, SF, ZF flags.
  - CF is not effected.
  - E.g.: DEC AX

Arithmetic Instructions

- **MUL Src:**
  - It is an unsigned multiplication instruction.
  - It multiplies two bytes to produce a word or two words to produce a double word.
  - AX = AL * Src
  - DX : AX = AX * Src
  - This instruction assumes one of the operand in AL or AX.
  - Src can be a register or memory location.
- **IMUL Src:**
  - It is a signed multiplication instruction.

Arithmetic Instructions

- **MUL – MUL Source**
- This instruction multiplies an *unsigned byte in some source with an unsigned byte in AL register or an unsigned word in some source with an unsigned word in AX register.*
- *The source can be a register or a memory location. When a byte is* ***multiplied by the content of AL, the result (product) is put in AX.*** *When a* ***word is multiplied by the content of AX, the result is put in DX and AX registers****.*
- *If the* ***most significant byte of a 16-bit result or the most significant word of a 32-bit result is 0****, CF and OF will both be 0's. AF, PF, SF and ZF are undefined after a MUL instruction.*

Arithmetic Instructions

- **MUL – MUL Source**
- If you want to **multiply a byte with a word**, you must first move the byte to a word location such as an extended register and **fill the upper byte of the word with all o's**.
- You **cannot use the CBW instruction** for this, because the CBW instruction fills the upper byte with copies of the most significant bit of the lower byte.

MUL source(**unsigned** 8/16-bit register)
If the **source** is **8-bit**, it is **multiplied with AL** and the **result** is stored in **AX** (AH–higher byte, AL–lower byte)
If the **source** is **16-bit**, it is **multiplied** with **AX** and the **result** is stored in **DX-AX** (DX–higher byte, AX–lower byte)
Source: Register, Memory Location
MUL affects AF, PF, SF and ZF.

```
Eg:MUL BL              ; AX ← AL x BL
   MUL BX              ; DX-AX ← AX x BX
   MUL BYTE PTR [BX]; AX ← AL x DS:[BX]
```

Arithmetic Instructions

- **IMUL – IMUL Source**
- multiplies a *signed byte from source with a signed byte in AL or a signed word from some source with a signed word in AX.*
- *If the magnitude of the product does not require all the bits of the destination, the* ***unused byte / word will be filled with copies of the sign bit****.*

- ***CF,OF --???:-*** *If the upper byte of a 16-bit result or the upper word of a 32-bit result contains only copies of the sign bit (all 0's or all 1's), then CF and the OF will both be 0;*
- *If it contains a part of the product, CF and OF will both be 1.*
- *AF, PF, SF and ZF are undefined after IMUL.*

Arithmetic Instructions

- **IMUL – IMUL Source**
- If you want to multiply a signed byte with a signed word, you must first move the byte into a word location and fill the upper byte of the word with copies of the sign bit.

- If you move the byte into AL, you can use the **CBW** instruction to do this.

Arithmetic Instructions

- **DIV Src:**
  - It is an unsigned division instruction.
  - To divide an *unsigned word by a byte or to divide an unsigned double word (32 bits) by a word.* The operand is stored in AX, divisor is Src and the result is stored as:
    - AH = remainder AL = quotient

DIV source(unsigned 8/16-bit register – divisor)
This instruction is used for **UNSIGNED** division.
Divides a **WORD by a BYTE**, OR a **DOUBLE WORD by a WORD.**
If the **divisor is 8-bit** then the **dividend is in AX** register.
After division, the **quotient is in AL** and the **Remainder in AH.**
If the **divisor is 16-bit** then the **dividend is in DX-AX** registers.
After division, the **quotient is in AX** and the **Remainder in DX.**
Source: Register, Memory Location
ALL flags are **undefined** after DIV instruction.

Eg: **DIV BL** ; *AX ÷ BL :- AL ← Quotient; AH ← Remainder*
DIV BX ; *{DX,AX} ÷ BX :- AX ← Quotient; DX ← Remainder*

Please Note: If the divisor is 0 or the result is too large to fit in AL (or AX for 16-bit divisor), them 8086 does a Type 0 interrupt (Zero Divide Error).

Arithmetic Instructions

- **DIV – DIV Source**
- *When a word is divided by a byte, the word must be in the AX register. The divisor can be in a register or a memory location.*
- *When a double word is divided by a word, the most significant word of the double word must be in DX, and the least significant word of the double word must be in AX. After the division, AX will contain the 16-bit quotient and DX will contain the 16-bit remainder.*

Arithmetic Instructions

- **DIV – DIV Source**
- *If an attempt is made to divide by 0 or if the quotient is too large to fit in the destination (greater than FFH / FFFFH), the 8086 will generate a type 0 interrupt. All flags are undefined after a DIV instruction.*

Arithmetic Instructions

- **DIV – DIV Source**
- If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all o's.
- Likewise, if you want to divide a word by another word, then put the dividend word in AX and fill DX with all o's.

Arithmetic Instructions

- **IDIV – IDIV Source**
- This instruction is used to divide a ***signed word by a signed byte***, *or to divide a signed double word by a signed word.*
- When dividing a signed word by a signed byte, the **word must be in the AX register**.

Arithmetic Instructions

- **IDIV – IDIV Source**
- When dividing a **signed word by a signed byte**, the word must be in the AX register. The divisor can be in an 8-bit register or a memory location.
- After the division, **AL will contain the signed quotient, and AH will contain the signed remainder.**
- The **sign of the remainder** will be the same as the sign of the dividend. If an attempt is made to divide by 0, the quotient is greater than 127 (7FH) or less than –127 (81H), the 8086 will automatically generate a **type 0 interrupt.**

Arithmetic Instructions

- **IDIV – IDIV Source**
- If you want ***to divide a signed byte by a signed byte***, you must first put the dividend byte in AL and **sign-extend AL into AH**.
- The **CBW instruction** can be used for this purpose.
- Likewise, if you want to divide a signed word by a signed word, you must put the dividend word in AX and **extend the sign of AX to all the bits of DX.**
- The **CWD instruction** can be used for this purpose.

Arithmetic Instructions

- **IDIV – IDIV Source**
- When dividing **a signed double word by a signed word**, the **most significant word** of the dividend (numerator) must be in the **DX register**, and the **least significant word** of the dividend must be in the **AX register**.

Arithmetic Instructions

- **IDIV – IDIV Source**
- The divisor can be in any other 16-bit register or memory location.
- After the division, **AX will contain a signed 16-bit quotient**, and **DX will contain a signed 16-bit remainder**. The sign of the remainder will be the same as the sign of the dividend.
- Again, if an attempt is made to divide by 0, the quotient is greater than +32,767 (7FFFH) or less than –32,767 (8001H), the 8086 will automatically generate a **type 0 interrupt**.
- All flags are undefined after an IDIV.

Arithmetic Instructions

- **NEG Src:**
  - It creates 2's complement of a given number.
  - That means, it changes the sign of a number.

NEG destination
This instruction forms the **2's complement** of the destination, and stores it back in the destination.
Destination: Register, Memory Location
ALL condition flags are updated.
Eg: **Assume** AL= 0011 0101 = 35 H then
NEG AL ; *AL ← 1100 1011 = CBH. i.e. AL ← 2's Complement (AL)*

Arithmetic Instructions

- **CMP Des, Src:**
  - It compares two specified bytes or words.
  - The Src and Des can be a constant, register or memory location.
  - **Both operands cannot be a memory location at the same time.**
  - The comparison is done simply by internally subtracting the source from destination.
  - The value of source and destination does not change, but the **flags are modified to indicate the result**.

CMP destination, source
This instruction **compares the source with the destination.**
The source and the destination must be of the same size.
Comparison is **done by internally SUBTRACTING** the **SOURCE form DESTINATION.**
The result of this subtraction is NOT stored anywhere, instead the Flag bits are affected.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
ALL condition flags are **updated.**

```
Eg: CMP BL, 55H        ; BL compared with 55H i.e. BL – 55H.
    CMP CX, BX         ; CX compared with BX i.e. CX – BX.
    CMP AL, Total      ; AL compared with DS:[Total] i.e. AL – DS:[Total].
```

Arithmetic Instructions

- **CBW (Convert Byte to Word):**
  - This instruction converts byte in AL to word in AX.
  - The conversion is done by extending the sign bit of AL throughout AH.
- **CWD (Convert Word to Double Word):**
  - This instruction converts word in AX to double word in DX : AX.
  - The conversion is done by extending the sign bit of AX throughout DX.

CBW

CBW [Convert signed BYTE to signed WORD]
This instruction **copies sign of** the byte in **AL into** all the bits of **AH.**
AH is then called *sign extension of AL.*
No Flags affected.
Eg: Assume
AX = XXXX XXXX 1001 0001
Then **CBW** gives
AX = **1111 1111** 1001 0001

CWD

CWD [Convert signed WORD to signed DOUBLE WORD]
This instruction **copies sign of** the WORD in **AX into** all the bits of **DX.**
DX is then called *sign extension of AX.*
No Flags affected.
Eg: Assume
AX = 1000 0000 1001 0001
DX = XXXX XXXX XXXX XXXX
Then **CWD** gives
AX = 1000 0000 1001 0001
DX = **1111 1111 1111 1111**


Thank you!

Instruction Set of 8086

Dr. Manju Khurana
Assistant Professor, CSED
TIET, Patiala
manju.khurana@thapar.edu

DAA (Decimal Adjust for Addition)

- The **`daa`** instruction works as follows:
  * If the least significant four bits in AL are > 9 or if AF =1, it adds 06H to AL and sets AF.
  * If the most significant four bits in AL are > 9 or if CF =1, it adds 60H to AL and sets CF.

Example:

```
mov     AL,71H
add     AL,43H          ; AL := B4H
daa                     ; AL := 14H and CF := 1
```

* The result including the carry (i.e., 114H) is the correct answer

DAS (Decimal Adjust for Subtraction)

- Packed BCD subtraction
  - The `das` instruction works as follows:
    - If the least significant four bits in AL are > 9 or if AF =1, it subtracts 6 from AL and sets AF.
    - If the most significant four bits in AL are > 9 or if CF =1, it subtracts 60H from AL and sets CF.
- Example:

```
mov     AL,71H
sub     AL,43H      ; AL := 2EH
das                 ; AL := 28H
```

Representation of Numbers

- ASCII representation
  - * Numbers are stored as a string of ASCII characters
    - » Example: 1234 is stored as 31 32 33 34H
      - > ASCII for 1 is 31H, for 2 is 32H, etc.
- BCD representation
  - * Unpacked BCD
    - » Example: 1234 is stored as 01 02 03 04H
      - – Additional byte is used for sign
        - > Sign byte: **00H for +** and **80H for −**
  - * Packed BCD
    - » Saves space by packing two digits into a byte
      - – Example: 1234 is stored as 12 34H

AAA (ASCII Adjust After Addition)

- If lower nibble of AL > 9 or AF = 1,then AL<- AL+6.
- AH <- AH+1 Set AF - 1, CY- 1.
- else
- AF and CY <- Reset i.e. 0.

- In both cases, Clear the higher nibble of AL.
- eg.
- MOV AX,0F ;
- AAA;
- HLT

AAA (ASCII Adjust After Addition)contd.

- If lower nibble of AL > 9 or AF = 1,then AL<- AL+6.
- AH <- AH+1 Set AF - 1, CY- 1.
- else
- AF and CY <- Reset i.e. 0.

- In both cases, Clear the higher nibble of AL.
- eg.
- MOV AX,0F ; AX<- 000F, AH<- 00, AL <- 0F
- AAA; AL<- F(15) + 6 = 05(21) , AH <- AH+1, ->01
  AF and CY <- 1. 0105H
- HLT

AAA (ASCII Adjust After Addition) contd.

- The **aaa** instruction works as follows:
  - * If the least significant four bits in AL are > 9 or if AF =1, it adds 6 to AL and 1 to AH.
    - – Both CF and AF are set
  - * In all cases, the most significant four bits in AL are cleared
  - * Example:

```
sub    AH,AH     ;
mov    AL,'6'    ; AL :=
add    AL,'7'    ; AL :=
aaa              ; AX :=
or     AL,30H    ; AL :=
```

AAA (ASCII Adjust After Addition) contd.

- The **aaa** instruction works as follows:
  - * If the least significant four bits in AL are > 9 or if AF =1, it adds 6 to AL and 1 to AH.
    - – Both CF and AF are set
  - * In all cases, the most significant four bits in AL are cleared
  - * Example:

```
sub   AH,AH     ; clear AH
mov   AL,'6'    ; AL := 36H
add   AL,'7'    ; AL := 36H+37H = 6DH
                        (D+6=3,AH=0(CLEARED IN
                        STEP 1)
aaa             ; AX := 0103H
or    AL,30H    ; AL := 33H
```

AAS (ASCII Adjust After Subtraction)

ASCII subtraction

- The **`aas`** instruction works as follows:
  - * If the least significant four bits in AL are > 9 or if AF =1, it subtracts 6 from AL and 1 from AH.
    - – Both CF and AF are set
  - * In all cases, the most significant four bits in AL are cleared
- This adjustment is needed only if the result is negative

AAS (ASCII Adjust After Subtraction)contd.

ASCII subtraction

- If lower nibble of AL > 9 or AF = 1,then AL<- AL-6.
- AH <- AH-1 Set AF - 1, CY- 1.
- else
- AF and CY <- Reset i.e. 0.
- **In both cases, Clear the higher nibble of AL**.
- eg.
- MOV AX,020FH ;
- AAS;
- HLT

AAS (ASCII Adjust After Subtraction)contd.

ASCII subtraction

- If lower nibble of AL > 9 or AF = 1,then AL<- AL-6.
- AH <- AH-1 Set AF - 1, CY- 1.
- else
- AF and CY <- Reset i.e. 0.
- **In both cases, Clear the higher nibble of AL**.
- eg.
- MOV AX,020FH ; AX<- 020F, AH<- 02, AL <- 0F
- AAS; AL<- F(15) - 6 = 09 , AH <- AH-1, ->01
  AF and CY <- 1. 0109 ANS
- HLT

AAS (ASCII Adjust After Subtraction)contd.

- Example 1: Positive result

```
sub     AH,AH     ;
mov     AL,'9'    ;
sub     AL,'3'    ;
aas               ;
or      AL,30H    ;
```

- Example 2: Negative result

```
sub AH,AH  ;
mov     AL,'3'    ;
sub     AL,'9'    ;
aas               ;
or      AL,30H    ;
```

AAS (ASCII Adjust After Subtraction) contd.

- Example 1: Positive result

```
sub     AH,AH     ; clear AH
mov     AL,'9'    ; AL := 39H
sub     AL,'3'    ; AL := 39H-33H = 6H
aas               ; AX := 0006H
or      AL,30H    ; AL := 36H
```

- Example 2: Negative result

```
sub AH,AH  ; clear AH
mov     AL,'3'    ; AL := 33H
sub     AL,'9'    ; AL := 33H-39H = FAH(2' OF
                          -6)
aas               ; AX := FF04H
or      AL,30H    ; AL := 34H
```

AAM (ASCII Adjust After Multiplication)

ASCII multiplication

- The **`aam`** instruction adjusts the result of a **`mul`** instruction
  - * Multiplication should not be performed on ASCII
    - » Can be done on unpacked BCD
- The **`aam`** instruction works as follows
  - * AL is divided by 10
  - * Quotient is stored in AH
  - * Remainder in AL
- **`aam`** does not work with **`imul`** instruction

AAM (ASCII Adjust After Multiplication)contd.

ASCII multiplication

- eg.
- MOV AL,5H
- MOV BL,7H
- MUL BL
- AAM
- HLT

AAM (ASCII Adjust After Multiplication)contd.

ASCII multiplication

- eg.
- MOV AL,5H
- MOV BL,7H
- MUL BL 5*7 = **23H** (35)-----(35/16 = 2(Q), 3(R))
  Result will go in AX(16 bit)
- AAM 35/10 = 3(Q), 5(R) , AH<- 03 AL <- 05
- HLT

AAM (ASCII Adjust After Multiplication) contd.

- Example 1

```
movAL,3                 ;
movBL,9                 ;
mulBL                   ;
aam                     ;
or AX,3030H             ;
```

- Example 2

```
 mov     AL,'3'      ;
 mov     BL,'9'      ;
 and     AL,0FH      ;
 and     BL,0FH      ;
mul      BL          ;
aam                  ;
or       AL,30H      ;
```

AAM (ASCII Adjust After Multiplication) contd.

- Example 1

```
movAL,3              ; multiplier in unpacked BCD form
movBL,9              ; multiplicand in unpacked BCD form
mulBL                ; result 001BH is inAX
aam                  ; AX := 0207H
or AX,3030H          ; AX := 3237H
```

- Example 2

```
 mov     AL,'3'      ; multiplier in ASCII
 mov     BL,'9'      ; multiplicand in ASCII
 and     AL,0FH      ; multiplier in unpacked BCD form
 and     BL,0FH      ; multiplicand in unpacked BCD form
mul      BL          ; result 001BH is inAX
aam                  ; AX := 0207H
or       AL,30H      ; AL := 37H
```

AAD (ASCII Adjust Before Division)

- The **aad** instruction works as follows
  - * Multiplies AH by 10 and adds it to AL and sets AH to 0
  - * Example:
    - » If AX is 0207H before **AAD**
    - » AX is changed to 001BH after **AAD**
- **AAD** instruction reverses the changes done by **AAM**

AAD (ASCII Adjust Before Division)contd.

ASCII division

- MOV AX,0205H ; AX<- 0205, AH<- 02, AL <- 05
- AAD AL<- (AH*10)+AL= (02*10)+05
  == 19(25), AH <- 00
- HLT

AAD (ASCII Adjust Before Division) contd.

- Example: Divide 27 by 5

```
mov     AX,0207H ; dividend in unpacked BCD form
mov     BL,05H   ; divisor in unpacked BCD form
aad              ; AX := 001BH
div     BL       ; AX := 0205H
```

- **AL <- 05 AH <- 02**

- **(vice versa of AAM)**

- **aad** converts the unpacked BCD number in AX to hexadecimal form so that **div** can be used.

WAP to perform the division 15/6 using the ASCII codes. Store the ASCII codes of result in Register DX.

- MOV AX, ’15’ ; AX=31 35
- MOV BH, ’6’ ; BX=36 00
- SUB AX,3030h ; AX=01 05
- SUB BH,30H ; BH=06 00
- AAD ; AX=00 0F
- DIV BH ; AX=03 02
- ADD AX,3030H ; AX=33 32
- MOV DX,AX ; DX=33 32
- HLT


Thank you!

Instruction Set of 8086

Dr. Manju Khurana
Assistant Professor, CSED
TIET, Patiala
manju.khurana@thapar.edu

WAP to find the largest of 8 bit numbers.

- MOV CL, 0AH
- LEA SI,[1000H]
- MOV AL,[SI]
- L1:INC SI
- MOV BL,[SI]
- CMP AL,BL
- JC L2
- JMP L3
- L2: MOV AL,BL
- L3: DEC CL
- JNZ L1
- MOV [100AH],AL
- HLT

1000-23
1001-40
1002-12
1003-14
1004-20
1005-35
1006-67
1007-70
1008-90
1009-80
100A- OUTPUT

WAP to find the largest of 16 bit numbers.

- MOV BX, 1000H ; BX=10 00
- MOV CL, [BX] ; CL=00 05
- INC BX ; BX=10 01
- MOV AX, [BX] ; AX=30 20
- DEC CL ; CL=00 04
- BACK: INC BX ; BX=10 02
- INC BX ; BX=10 03
- CMP AX,[BX]
- JNC Next
- MOV AX, [BX] ; AX= 50 40 70 60 90 80
- Next: DEC CL ; CL=00 03 00 02 00 01
- JNZ Back
- MOV [1020H],AX ; AX= 90 80
- HLT

1000-05
1001-20
1002-30
1003-40
1004-50
1005-60
1006-70
1007-80
1008-90
1009-10
100A-20
1020-OUTPUT

WAP to find out the count of positive numbers and negative numbers from a series of signed numbers in 8086.

- MOV CL, 0AH
- MOV BL, 00H
- MOV DL,00H
- LEA SI,[1000H]
- L1:MOV AL,[SI]
- SHL AL,01
- JNC L2
- INC DL
- JMP L3
- L2: INC BL
- L3: INC SI

1000-01
1001-02
1002-03
1003-04
1004-80
1005-81
1006-82
1007-83
1008-84
1009-85
100A-04 O/P
1020-06 O/P

WAP to find out the count of positive numbers and negative numbers from a series of signed numbers in 8086.

- DEC CL
- JNZ L1
- MOV [100AH],BL
- MOV [100BH],DL
- HLT

1000-01
1001-02
1002-03
1003-04
1004-80
1005-81
1006-82
1007-83
1008-84
1009-85
100A-04 O/P
1020-06 O/P

Fibonacci Series:-

```
MOV SI,3000H  ; SI=3000
MOV CX,0AH   ; CX=00 0A
XOR AL,AL        ; AL=00
MOV [SI],0AH      ; [3000]=0A
INC SI          ;3001
MOV [SI],00H     ;[3001]=00
ADD AL,01H     ; AL=01
INC SI             ; 3002
MOV [SI],AL    ; 3002= 01
INC SI             ; 3003
MOV [SI],AL    ;3003= 01
Back: ADD AL,[SI] ;  01,01 =02
INC SI                 ; 3004
MOV [SI],AL        ; 3004=02
DEC SI                 ; 3003
MOV AL,[SI]        ; AL=01
INC SI                 ; 3004
LOOP Back
HLT          ; halt!
```


Thank you!

Bit Manipulation Instructions
or
Logical Instructions

Dr. Manju Khurana
Assistant Professor, CSED
TIET, Patiala
manju.khurana@thapar.edu

Bit Manipulation Instructions

- These instructions are used at the bit level.
- These instructions can be used for:
  - Testing a zero bit
  - Set or reset a bit
  - Shift bits across registers

Bit Manipulation Instructions

- **NOT Src:**
- The NOT instruction inverts each bit (forms the 1's complement) of a byte or word in the specified destination. The **destination can be a register or a memory location**. This instruction does not affect any flag.
  - NOT BX Complement content or BX register
  - NOT BYTE PTR [BX] Complement memory byte at offset [BX] in data segment.

NOT destination
This instruction forms the **1's complement** of the destination, and stores it back in the destination.
Destination: Register, Memory Location
No Flags affected.
Eg: **Assume** AL= 0011 0101
NOT AL ; *AL ← 1100 1010 ... i.e. AL = 1's Complement (AL)*

Bit Manipulation Instructions

- **AND Des, Src:**
  - It performs AND operation of Des and Src.
  - **Src can** be immediate number, register or memory location.
  - **Des can** be register or memory location.
  - Both operands cannot be memory locations at the same time.
  - CF and OF become zero after the operation.
  - PF, SF and ZF are updated.

AND destination, source
This instruction **logically ANDs** the source with the destination and stores the **result in the destination.**
Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; CF, OF ← 0; AF becomes **undefined.**
Eg: **AND BL, CL** ; *BL ← BL AND CL*

Bit Manipulation Instructions

- **OR Des, Src:**
  - It performs OR operation of Des and Src.
  - Src can be immediate number, register or memory location.
  - Des can be register or memory location.
  - Both operands cannot be memory locations at the same time.
  - CF and OF become zero after the operation.
  - PF, SF and ZF are updated.

OR destination, source
This instruction **logically Ors** the source with the destination and stores the **result in the destination.**
Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; CF, OF ← 0; AF becomes **undefined.**
Eg: **OR BL, CL** ; *BL ← BL OR CL*

Bit Manipulation Instructions

- **XOR Des, Src:**
  - It performs XOR operation of Des and Src.
  - Src can be immediate number, register or memory location.
  - Des can be register or memory location.
  - Both operands cannot be memory locations at the same time.
  - CF and OF become zero after the operation.
  - PF, SF and ZF are updated.

XOR destination, source
This instruction **logically X-Ors** the source with the destination and stores the **result in the destination.**
Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; CF, OF ← 0; AF becomes **undefined.**
Eg: **XOR BL, CL** ; *BL ← BL XOR CL*

Bit Manipulation Instructions

- **NEG Destination:**
- This instruction replaces the number in a destination with its 2's complement. The destination can be a register or a memory location. It gives the same result as the *invert each bit and add one algorithm. The NEG instruction updates AF, AF, PF, ZF, and OF.*
- NEG AL Replace number in AL with its 2's complement
- NEG BX Replace number in BX with its 2's complement
- NEG BYTE PTR [BX] Replace byte at offset BX in DX with its 2's complement
- NEG WORD PTR [BP] Replace word at offset BP in SS with its 2's complement

Bit Manipulation Instructions

- **TEST des,src:**
- This instruction ANDs the byte / word in the specified source with the byte / word in the specified destination.
- Result is not stored anywhere.
- Flags are updated, but neither operand is changed.
- The test instruction is **often used to** set flags before a Conditional jump instruction.
- The **source** can be an immediate number, the content of a register, or the content of a memory location. The **destination** can be a register or a memory location. The source and the destination cannot both be memory locations.
- CF and OF are both **o's** after TEST. PF, SF and ZF will be updated to show the results of the destination.
- AF is be undefined.

Bit Manipulation Instructions

- **TEST des,src:**
- TEST AL, BH ;AND BH with AL. No result stored; Update PF, SF, ZF.
- TEST CX, oooıH ;AND CX with immediate number oooıH; No result stored; Update PF, SF, ZF
- TEST BP, [BX][DI] ;AND word are offset [BX][DI] in DS with word in BP. No result stored. Update PF, SF, and ZF

TEST destination, source
This instruction **logically ANDs** the source with the destination **BUT** the **RESULT** is **NOT STORED ANYWHERE.**
ONLY the **FLAG** bits are **AFFECTED.**
Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; CF, OF ← 0; AF becomes **undefined.**
Eg: **TEST BL, CL** *; BL AND CL; result not stored; Flags affected.*
Note: Don't forget this instruction because it will be used later in ***multiprocessor*** *systems!*

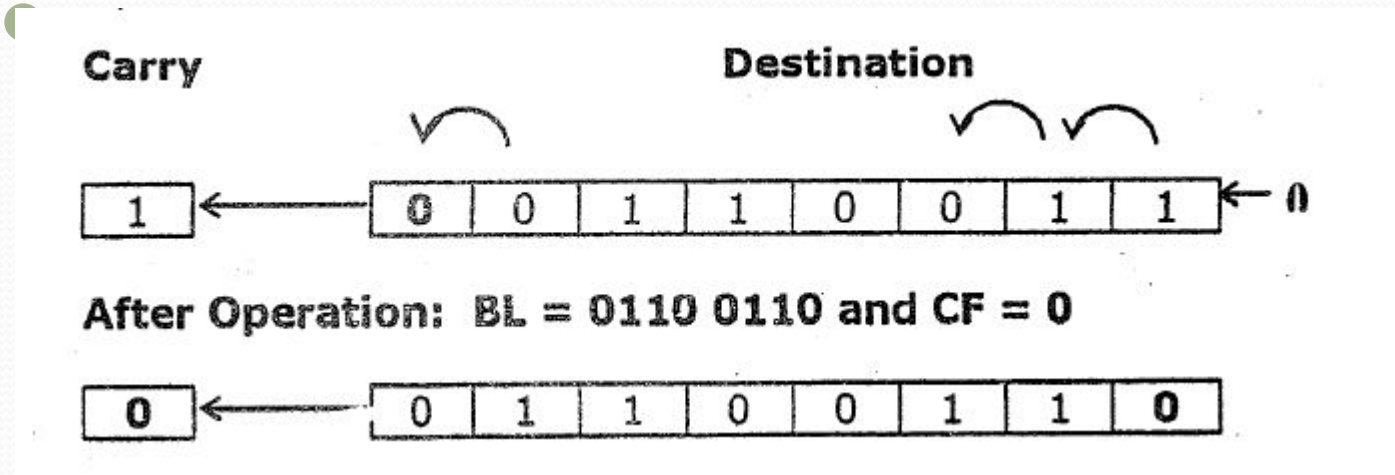
Bit Manipulation Instructions

- **SHL/SAL Des, Count: Shift Logic/Arithmatic left**
  - It shift bits of byte or word left, by count.
  - It puts zero(s) in LSBs.
  - MSB is shifted into carry flag.
  - If the number of bits desired to be shifted is **1**, then the **immediate number 1** can be written in Count.
  - However, if the number of bits to be shifted is **more than 1,** then the count is put in CL register.

```
MOV CL, 05H      ; Load number of shifts in CL register.
SAL BL, CL       ; Left-Shift BL bits CL (5) number of times.
```

Bit Manipulation Instructions

- **SHL/SAL Des, Count: Shift Logic/Arithmatic left**

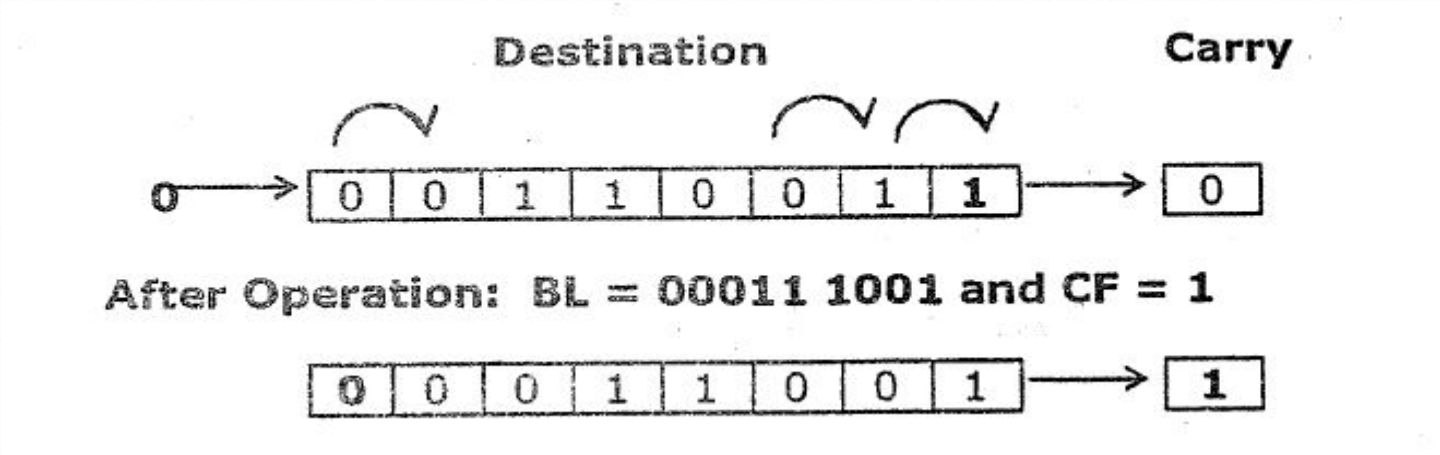


Bit Manipulation Instructions

- **SHR Des, Count:**
  - This instruction shifts each bit in the specified destination to the right.
  - As a bit is shifted out of the MSB position, a 0 is put in its place. The bit shifted out of the LSB position goes to CF.
  - In the case of multi-bit shifts, CF will contain the bit most recently shifted out from the LSB.
  - Bits shifted into CF previously will be lost.

Bit Manipulation Instructions

- **SHR Des, Count:**



Bit Manipulation Instructions

- **SHR Des, Count:**
  - If you want to shift the operand by **one bit position**, you can specify this by putting a 1 in the count position of the instruction.
  - For shifts of **more than 1 bit position**, load the desired number of shifts into the CL register, and put “CL” in the count position of the instruction.

Bit Manipulation Instructions

- **SAR Des, Count:**
  - This instruction shifts each bit in the specified destination to the right.
  - As a bit is shifted out of the MSB position**, a copy of the old MSB is put in the MSB position**. In other words, the sign bit is copied into the MSB.
  - The LSB will be shifted into CF. In the case of multiple-bit shift, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.
  - MSB -□MSB-□ LSB-□ CF

Bit Manipulation Instructions

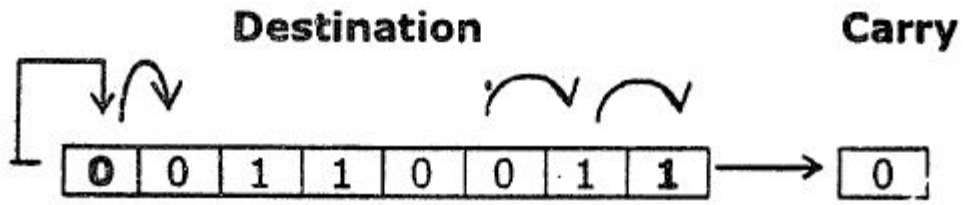
- **SAR Des, Count:**

- **SAR DX, 1 ;**Shift word in DI one bit position right, new MSB = old MSB
- ***MOV CL, 02H*** ;Load desired number of shifts in CL
- ***SAR WORD PTR [BP], CL*** ;Shift word at offset [BP] in stack segment right by two bit positions, the two MSBs are now copies of original MSB

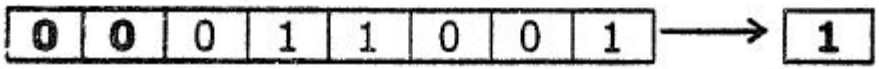
- Eg:

```
SAR BL, 1          ; Right-Shift BL bits, once.
```

Assume:
Before Operation: BL = 0011 0011 and CF = 0



After Operation: BL = 0001 1001 and CF = 1

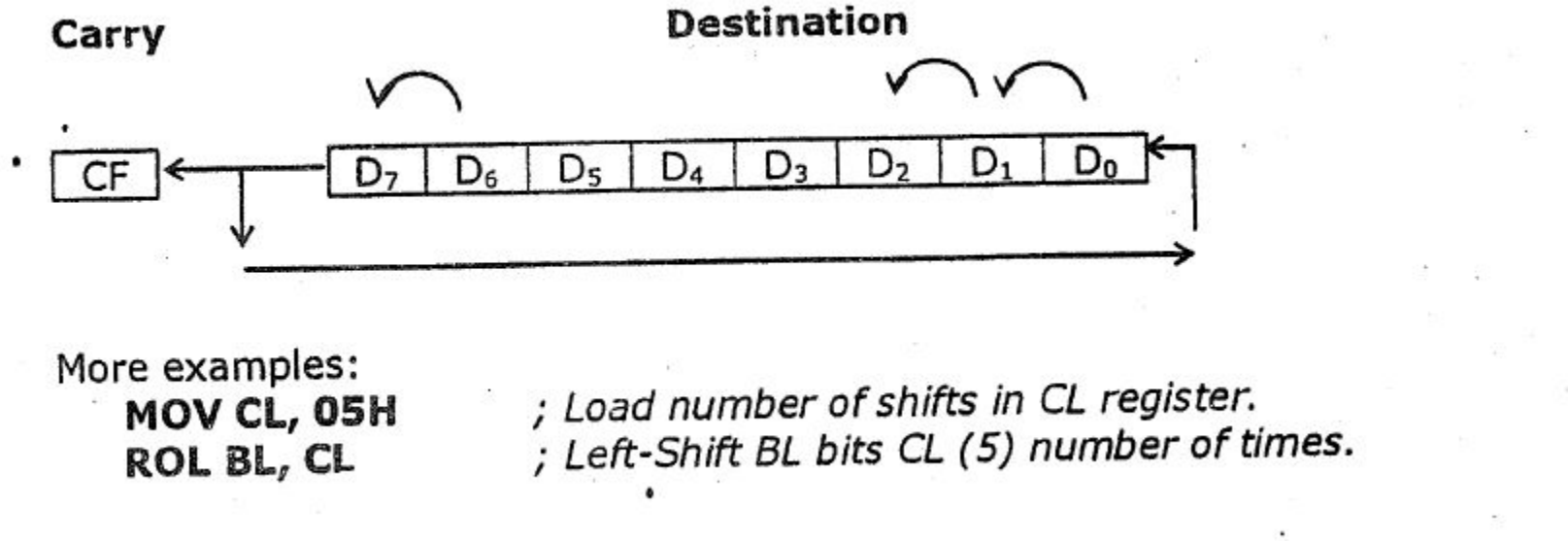


Bit Manipulation Instructions

- **ROL Des, Count:rotate left without carry**
  - It rotates bits of byte or word left, by count.
  - MSB is transferred to LSB and also to CF.
  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Bit Manipulation Instructions

- **ROL Des, Count:**



More examples:

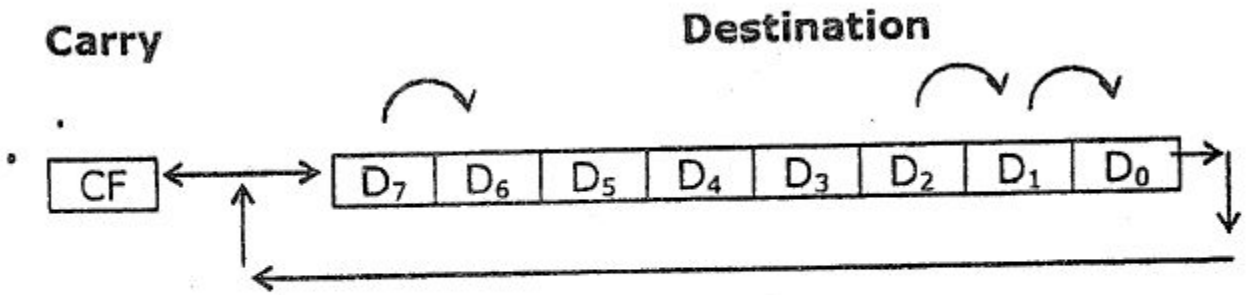
```
MOV CL, 05H     ; Load number of shifts in CL register.
ROL BL, CL      ; Left-Shift BL bits CL (5) number of times.
```

Bit Manipulation Instructions

- **ROR Des, Count: rotate right without carry**
  - It rotates bits of byte or word right, by count.
  - LSB is transferred to MSB and also to CF.
  - If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
  - However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Bit Manipulation Instructions

- **ROR Des, Count:**

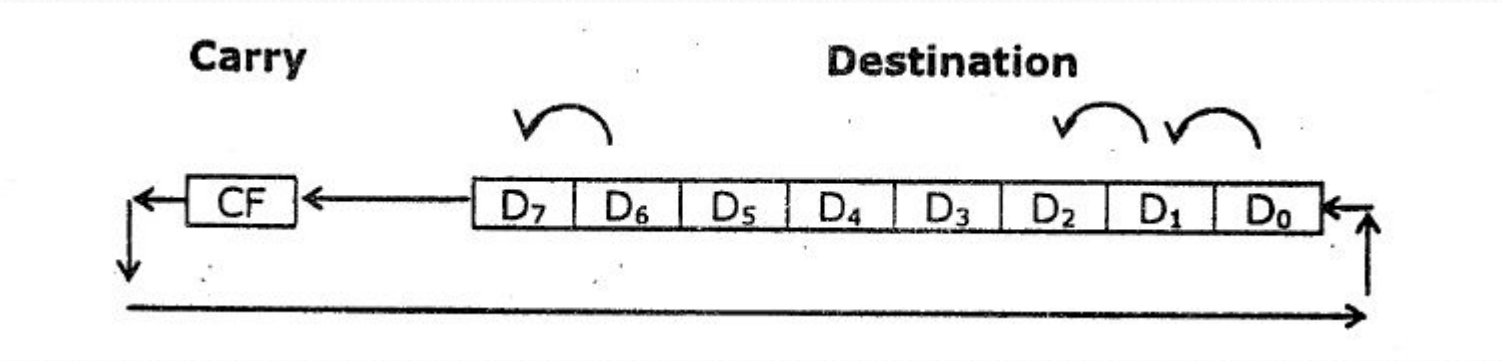


More examples:

```
MOV CL, 05H     ; Load number of shifts in CL register.
ROR BL, CL      ; Right-Shift BL bits CL (5) number of times.
```

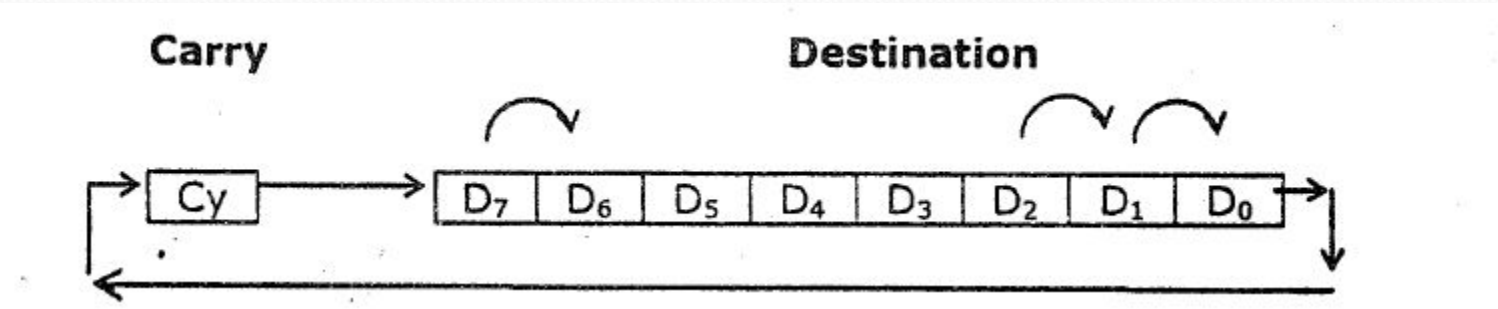
Bit Manipulation Instructions

- **RCL Des, Count: rotate left through carry**
  - This instruction rotates all the bits in a specified word or byte some number of bit positions to the left.
  - The operation circular because the **MSB of the operand is rotated into the carry flag** and the bit in the carry flag is rotated around into LSB of the operand.



Bit Manipulation Instructions

- **RCR Des, Count: rotate right through carry**
  - This instruction rotates all the bits in a specified word or byte some number of bit positions to the right.
  - The operation circular because the LSB of the operand is rotated into the carry flag and the bit in the carry flag is rotate around into MSB of the operand.




Thank you!

Program Execution Transfer Instructions

Dr. Manju Khurana
Assistant Professor, CSED
TIET, Patiala
manju.khurana@thapar.edu

Program Execution Transfer Instructions

- These instructions cause change in the sequence of the execution of instruction.
- This change can be through a condition or sometimes unconditional.
- The conditions are represented by flags.

Program Execution Transfer Instructions

- 2 main type of branching:
  - Near :- Intra-segment(branch to new location within current segment only)
    - **Only IP needs to be changed.**
    - Range of -128 to 127 □ short branch.
  - Far:- Inter-segment(branch to new location in a different segment.)
  - Value of CS and IP need to be changed.

Program Execution Transfer Instructions

- **JMP Des:**
  - Intrasegment direct jump:- new branch location is specified directly in instruction.
    - New address calculated by 8 bit(16 bit) displacement to the IP.
    - CS not change.
    - +ve disp. forward jump
    - -ve disp. backward jump
    - Also called relative jump
    - JMP Prev(or Next), IP offset addr. Of “Prev” or “Next”.

Program Execution Transfer Instructions

- **JMP Des:**
  - **Intrasegment indirect jump**:- new branch location is specified indirectly through a reg or mem. Loc (in DS only).
    - Value of IP is replaced with new value.
    - CS not change.
    - JMP WORD PTR[BX];

      IP□ {DS:[BX], DS:[BX+1]}

Program Execution Transfer Instructions

- **JMP Des:**
  - Inter-segment(FAR) jump:- Jump add. is specified in 2 ways:-
    - Inter-segment **Direct** jump
    - New branch location is specified directly in the instruction.
    - CS & IP get new value.
    - Eg. JMP NextSeg; CS & IP get new value from label NextSeg.

Program Execution Transfer Instructions

- **JMP Des:**
  - Inter-segment(FAR) jump:- Jump add. is specified in 2 ways:-
    - Inter-segment **indirect** jump
    - New branch location specified indirectly through a reg. or mem. Loc.
    - CS & IP get new value.
    - Eg. JMP DWORD PTR[BX]; IP□ {DS:[BX],DS:[BX+1]}, CS□ {DS:[BX+2],DS:[BX+3]}

JCondition (Conditional Jump)
This is a conditional branch instruction.
If condition is **TRUE**, then it is **similar to** an **INTRA-Segment Direct Jump.**
If condition is FALSE, then branch does not take place and the next sequential instruction is executed.
The destination must be in the range of -128 to 127 from the address of the instruction (i.e. **ONLY SHORT Jump**).
Eg: **JNC Next** *; Short Jump to Next ONLY if Carry Flag is not set (CF = 0).*

The various conditional jump instructions are as follows:

Mnemonic	Description	Jump Condition
Common Operations		
JC	Carry	**CF = 1**
JNC	Not Carry	CF = 0
JE/JZ	Equal or Zero	**ZF = 1**
JNE/JNZ	Not Equal or Not Zero	ZF = 0
JP/JPE	Parity or Parity Even	**PF = 1**
JNP/JPO	Not Parity or Parity Odd	PF = 0
Signed Operations		
JO	Overflow	**OF = 1**
JNO	Not Overflow	OF = 0
JS	Sign	**SF = 1**
JNS	Not Sign	SF = 0
JL/JNGE	Less	**(SF Ex-Or OF) = 1**
JGE/JNL	Greater or Equal	(SF Ex-Or OF) = 0
JLE/JNG	Less or Equal	**((SF Ex-Or OF) + ZF) = 1**
JG/JNLE	Greater	((SF Ex-Or OF) + ZF) = 0
Unsigned Operations		
JB/JNAE	Below	**CF = 1**
JAE/JNB	Above or Equal	CF = 0
JBE/JNA	Below or Equal	**(CF Ex-Or ZF) = 1**
JA/JNBE	Above	(CF Ex-Or ZF) = 0

Program Execution Transfer Instructions

CALL (Unconditional CALL)

CALL is an instruction that transfers the program control to a sub-routine, with the intention of coming back to the main program.

Thus, in CALL 8086 **saves the address of the next instruction into the stack** before branching to the sub-routine.

At the end of the subroutine, control transfers back to the main program using the return address from the stack.

There are two types of CALL: Near CALL and Far CALL.

Program Execution Transfer Instructions

- **CALL(Unconditional): 2 types of call:- Near & Far**

INTRA-Segment (NEAR) CALL

The **new subroutine** called must be **in the same segment** (hence intra-segment).

The CALL **address** can be **specified directly** in the instruction **OR indirectly** through Registers or Memory Locations.

The following sequence is executed for a NEAR CALL:

i. 8086 will **PUSH Current IP** into the Stack.
ii. **Decrement SP by 2.**
iii. **New value** loaded **into IP.**
iv. **Control transferred** to a subroutine within the same segment.

Eg: **CALL subAdd** ; *{SS:[SP-1], SS:[SP-2]} ← IP,*
; *SP ← SP – 2,*
; *IP ← New Offset Address of subAdd.*

Program Execution Transfer Instructions

- **CALL(Unconditional): 2 types of call:- Near & Far**

INTER-Segment (FAR) CALL

The **new subroutine** called is in **another segment** (hence inter-segment).

Here CS and IP both get new values.

The CALL address can be specified directly OR through Registers or Memory Locations.

The following sequence is executed for a Far CALL:

i. **PUSH CS** into the Stack.
ii. **Decrement SP** by 2.
iii. **PUSH IP** into the Stack.
iv. **Decrement SP** by 2.
v. **Load CS** with new segment address.
vi. **Load IP** with new offset address.
vii. **Control transferred** to a subroutine in the new segment.

Eg: **CALL subAdd** ; *{SS:[SP-1], SS:[SP-2]} ← CS,*
; *SP ← SP – 2,*
; *{SS:[SP-1], SS:[SP-2]} ← IP*
; *SP ← SP – 2,*
; *CS ← New Segment Address of subAdd,*
; *IP ← New Offset Address of subAdd.*

There is **NO PROVISION for Conditional CALL.**

Program Execution Transfer Instructions

RET --- Return instruction

RET instruction causes the control to return to the main program from the subroutine.

Intrasegment-RET

Eg: RET

```
; IP ← SS:[SP], SS:[SP+1]
; SP ← SP + 2
```

RET n

```
; IP ← SS:[SP], SS:[SP+1]
; SP ← SP + 2 + n
```

Intersegment-RET

Eg: RET

```
; IP ← SS:[SP], SS:[SP+1]
; CS ← SS:[SP+2], SS:[SP+3]
; SP ← SP + 4
```

RET n

```
; IP ← SS:[SP], SS:[SP+1]
; CS ← SS:[SP+2], SS:[SP+3]
; SP ← SP + 4 + n
```

Program Execution Transfer Instructions (Iteration Control Instructions)

These instructions **cause** a series of **instructions to be executed repeatedly.**
The **number of iterations** is loaded **in CX** register.
CX is **decremented by 1**, after every iteration.
Iterations occur **until CX = 0.**
The **maximum difference between** the **address** of the instruction and the address of the Jump **can be 127.**

1) LOOP Label

Jump to specified label if CX not equal to 0; and decrement CX.

Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
          .
          .
          .
        MOV BL, AL
        LOOP BACK   ; Repeat from the instruction having BACK label if CX not equal to 0;
                    ; CX ← CX – 1.
```

Program Execution Transfer Instructions (Iteration Control Instructions)

2) LOOPE/LOOPZ Label (Loop on Equal / Loop on Zero)
Same as above except that looping occurs ONLY if Zero Flag is set (i.e. ZF = 1)
Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
          .
          .
          .
        MOV BL, AL
        LOOPZ BACK; Repeat from BACK label if CX is equal to 0 AND ZF =1;
                  ; CX ← CX – 1.
```

3) LOOPNE/LOOPNZ Label (Loop on NOT Equal / Loop on NO Zero)
Same as above except that looping occurs ONLY if Zero Flag is reset (i.e. ZF = 0)
Eg:

```
        MOV CX, 40H
BACK:   MOV AL, BL
        ADD AL, BL
          .
          .
          .
        MOV BL, AL
LOOPNZ  BACK; Repeat from BACK label if CX not equal to 0 AND ZF = 0;
            ; CX ← CX – 1.
```


Thank you!

Processor Control/Machine Control Instructions

Dr. Manju Khurana
Assistant Professor, CSED
TIET, Patiala
manju.khurana@thapar.edu

Processor Control Instructions

- These instructions control the processor itself.
- 8086 allows to control certain control flags that:
  - causes the **processing in a certain direction**
  - **processor synchronization** if more than one microprocessor attached.

Processor Control Instructions

For Carry Flag

1) **STC**
   This instruction **sets** the **Carry Flag.**
   No Other Flags are affected.

2) **CLC**
   This instruction **clears** the **Carry Flag.**
   No Other Flags are affected.

3) **CMC**
   This instruction **complements** the **Carry Flag.**
   No Other Flags are affected.

For Direction Flag

4) **STD**
   This instruction **sets** the **Direction Flag.**
   No Other Flags are affected.

5) **CLD**
   This instruction **clears** the **Direction Flag.**
   No Other Flags are affected.

Processor Control Instructions

Please Note

There is no direct way to alter the TF.It can be altered through program as follows:

To set TF:

```
PUSHF          ; push contents of Flag register into the stack
POP BX         ; pop contents of flag reg from the stack-top into BX
OR BH, 01H     ; set the bit corresponding to TF, in the BH register
PUSH BX        ; push the modified BX register into the stack
POPF           ; pop the modified contents into flag register.
```

To reset TF:

```
PUSHF          ; push contents of Flag register into the stack
POP BX         ; pop contents of flag reg from the stack-top into BX
AND BH, FEH    ; reset the bit corresponding to TF, in the BH register
PUSH BX        ; push the modified BX register into the stack
POPF           ; pop the modified contents into flag register.
```

For Interrupt Enable Flag

6) **STI**
This instruction **sets** the **Interrupt Enable Flag.**
No Other Flags are affected.

7) **CLI**
This instruction **clears** the **Interrupt Enable Flag.**
No Other Flags are affected.

External H/w Synchronization Instructions

1) **ESC**

This is an 8086 **instruction-prefix** used to **indicate that the current instruction is for the 8087 NDP.**

We write a *homogeneous (combined) program* for the two processors 8086 and 8087.

Instructions are fetched by 8086 into its queue.

8087 duplicates the instruction queue of 8086 and monitors this queue.

When an instruction with **ESC prefix** (binary code 11011) is **encountered, 8087 is activated,** and hence **it executes the instruction.**

8086 treats the instruction as NOP.

ESC has to be written before each 8087 instruction.

External H/w Synchronization Instructions

2) WAIT:- It is used to synchronize 8086 with co-processor 8087 via TEST(bar) I/P pin of 8086.

- **When 8087 is busy, it puts "1" on its BUSY O/P line connected to TEST I/P of MP.**
- **Wait instruction Make MP to check TEST pin.**

3) LOCK

This is an 8086 **instruction prefix.**

It **prevents any external bus master from taking control of the system** bus during execution of the instruction, which has a LOCK prefix.

It causes 8086 to activate the LOCK signal so that no other bus master takes control of the system bus.

External H/w Synchronization Instructions

4) NOP

There is **no operation performed** while executing this instruction.
8086 requires 3 T-States for this instruction.
It is mainly used to insert time delays, and can also be used while debugging.

5) HLT

This instruction causes 8086 to **stop fetching any more instructions.**
8086 enters **Halt state.**
8086 can **come out** of this halt state only if there is a **valid hardware interrupt** (NMI or INTR) or **by reset.**

Interrupt Control Instuctions

1) **INT Type**

This instruction causes an interrupt of the given type.
The **'Type'** can be a number between **0 ... 255.**
The following action takes place:

i. **PUSH Flag** Register onto the Stack.
SP decremented by 2.

ii. **PUSH CS** onto the Stack.
SP decremented by 2.

iii. **PUSH IP** onto the Stack.
SP decremented by 2.
∴ In all **SP decremented by 6.**

iv. **New value of IP** taken from location **type** × **4.**
Eg: INT 1 ; IP ← {[00004] and [00005]} (as 1 × 4 = 00004H)

v **New value of CS** taken from location **(type** × **4) + 2.**
Eg: INT 1 ; CS ← {[00006] and [00007]}

vi **Execution of** the **ISR begins** from the address formed by the new values of CS and IP.

Interrupt Control Instuctions

2) INTO (Interrupt on Overflow)

This instruction causes an interrupt of **type 4, ONLY if Overflow Flag (OF) is set.** The above sequence is followed and the control is transferred to the location pointed by 00010H.

Eg: **INTO** *; If OF = 1 then execute INT 4.*

3) IRET (Return from ISR)

This instruction causes the 8086 to return to the main program from an ISR.

The following action takes place:

i. **POP IP** from the Stack.
   SP incremented by 2.
ii. **POP CS** from the Stack.
   SP incremented by 2.
iii. **POP Flag Register** from the Stack.
   SP incremented by 2.
   ∴ In all **SP incremented by 6.**

Execution of the Main Program continues from the address formed values of CS and IP restored from the stack.


Thank you!

String Instructions

Dr. Manju Khurana
Assistant Professor, CSED
TIET, Patiala
manju.khurana@thapar.edu

String Instructions

String Instructions of 8086

- A **String** is a **series of bytes** stored sequentially in the memory. String Instructions operate on such "Strings".
- The **Source Element** is taken from the **Data Segment using** the **SI** register.
- The **Destination Element** is in the **Extra Segment** pointed by the **DI** register.
- **SI and/or DI** are **incremented/decremented** after each operation depending upon the direction flag **"DF"** in the flag register.

1) MOVS: MOVSB/MOVSW (*Move String*)

It is used to **transfer** a word/byte **from data segment to extra segment.**
The offset of the source in data segment is in SI.
The offset of the destination in extra segment is in DI.
SI and DI are incremented / decremented after the transfer depending upon the direction flag.

Eg: **MOVSB**

; ES:[DI] ← DS:[SI] ... byte transfer
; SI ← SI ± 1 ... depending upon DF
; DI ← DI ± 1 ... depending upon DF

MOVSW

; {ES:[DI], ES:[DI + 1]} ← {DS:[SI], DS:[SI + 1]} ... word.
; SI ← SI ± 2
; DI ← DI ± 2

String Instructions

2) LODS: LODSB/LODSW (*Load String*)

It is used to **Load AL** (or AX) register with a byte (or word) **from data segment.**
The offset of the source in data segment is in SI.
SI is incremented / decremented after the transfer depending upon the direction flag (DF).

Eg: **LODSB** ; *AL ← DS:[SI] ... byte transfer*
; *SI ← SI ± 1 ... depending upon DF*

LODSW ; *AL ← DS:[SI]; AH ← DS:[SI + 1] ... word transfer*
; *SI ← SI ± 2*

3) STOS: STOSB/STOSW (*Store String*)

It is used to **Store AL** (or AX) **into** a byte (or word) in the **extra segment.**
The offset of the source in extra segment is in DI.
DI is incremented / decremented after the transfer depending upon the direction flag (DF).

Eg: **STOSB** ; *ES:[DI] ← AL ... byte transfer*
; *DI ← DI ± 1 ... depending upon DF*

STOSW ; *ES:[DI] ← AL; ES:[DI+1] ← AH ... word transfer*
; *DI ← DI ± 2 ... depending upon DF*

String Instructions

4) CMPS: CPMSB/CMPSW (*Compare String*)

It is used to **compare** a **byte** (or word) **in** the **data segment with** a **byte** (or word) **in** the **extra segment.**

The offset of the byte (or word) in data segment is in SI.

The offset of the byte (or word) in extra segment is in DI.

SI and DI are incremented / decremented after the transfer depending upon the direction flag.

Comparison is done by subtracting the byte (or word) from extra segment from the byte (or word) from Data segment. The Flag bits are affected, but the result is not stored anywhere.

Eg: **CMPSB**

; Compare DS:[SI] with ES:[DI] ... byte operation
; SI ← SI ± 1 ... depending upon DF
; DI ← DI ± 1 ... depending upon DF

CMPSW

; Compare {DS:[SI], DS:[SI+1]} with {ES:[DI], ES:[DI+1]}
; SI ← SI ± 2 ... depending upon DF
; DI ← DI ± 2 ... depending upon DF

String Instructions

5) **SCAS: SCASB/SCASW** (*String Compare Accumulator*)

It is used to **compare** the contents of **AL** (or AX) **with** a **byte** (or word) **in** the **extra segment.**

The offset of the byte (or word) in extra segment is in DI.

DI is incremented / decremented after the transfer depending upon the direction flag (DF).

Comparison is done by subtracting the byte (or word) from extra segment from AL (or AX).

The Flag bits are affected, but the result is not stored anywhere.

Eg: **SCASB** ; *Compare AL with ES:[DI] ... byte operation*
; *DI ← DI ± 1 ... depending upon DF*

SCASW ; *Compare {AX} with {ES:[DI], ES:[DI+1]} ... word operation*
; *DI ← DI ± 1 ... depending upon DF*

String Instructions

6) REP (*Repeat*)

This is an **instruction prefix**, which can be used in string instructions.
It can be **used with string instructions only**.
It **causes** the **instruction** to be **repeated CX number** of times.
After each execution, the **SI** and **DI** registers are **incremented/decremented** based on the **DF** (Direction Flag) in the Flag register **and CX is decremented.**
i.e. **DF = 1; SI, DI decrements.**

Thus, it is important that before we use the REP instruction prefix the following steps must be carried out:

- **CX must be initialized** to the Count value.
  If **auto decrementing** is required, **DF** must be **set using STD** instruction **else cleared** using **CLD** instruction.

EG:

```
        .
        .
        .
        MOV CX, 0023H
        CLD
REP     MOVSB
        .
        .
        .
```

The above section of a program will cause the following string operation

MOVSB ; *ES:[DI] ← DS:[SI], CX ← CX – 1, SI ← SI + 1, DI ← DI + 1*

to be executed 23H times (as CX = 23H) in auto incrementing mode (as DF is cleared).

String Instructions

7) **REPZ/REPE** (*Repeat on Zero/Equal*)
It is a conditional repeat instruction prefix.
It behaves the same as a REP instruction **provided** the Zero Flag is set (i.e. **ZF = 1**).
It is used with CMPS instruction.

8) **REPNZ/REPNE** (*Repeat on No Zero/Not Equal*)
It is a conditional repeat instruction prefix.
It behaves the same as a REP instruction **provided** the Zero Flag is reset (i.e. **ZF = 0**).
It is used with SCAS instruction.


Thank you!