UNIT 2

The 8086 Microprocessor: Architecture, Register organization, 8086 signal description, Physical memory organization, Minimum and Maximum mode system and timing diagrams, Addressing modes, 8086 Instruction Set and Assembler Directives, Assembly Language example programs, Stack structure of 8086, Interrupt structure of 8086, Interrupt vector table, Procedures and macros.

Arithmetic Instructions
ADD: Addition

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

- **Example:**

```
ADD AL, 7AH    ; adds 7AH to AL register
ADD DX, AX     ; adds AX to DX register
ADD AX, [BX]   ; adds [BX] to AX register
```

Arithmetic Instructions
ADC: Addition-with-Carry

- **ADC Des, Src**
- **It adds the two operands with CF.**
- **It effects AF, CF, OF, PF, SF, ZF flags.**

- **Example:**

```
ADC AL, 7AH    ; adds with carry, 7AH
                 to  AL register
ADC DX, AX     ; adds with carry, AX
                 to  DX register
ADC AX, [BX]   ; adds with carry, [BX]
                 to
                 AX register
```

Arithmetic Instructions
SUB: Subtraction

- **SUB Des, Src**
- **It subtracts a byte to byte or a word to word.**
- **It effects AF, CF, OF, PF, SF, ZF flags.**
- **For subtraction, CF acts as borrow flag.**

Example:

```
SUB AL, 74H    ; sub 74H from AL register
SUB DX, AX     ; sub AX from DX register
SUB AX, [BX]   ; sub [BX] from AX register
```

Arithmetic Instructions
SBB: Subtraction-with-Borrow

- **SBB Des, Src**
- **It subtracts the two operands and also the borrow from the result.**
- **It effects AF, CF, OF, PF, SF, ZF flags.**
- **Example:**

```
SBB AL, 74H      ; sub with borrow, 74H
                   from AL register
SBB DX, AX       ; sub with borrow, AX
                   from DX register
SBB AX, [BX]     ; sub with borrow, [BX]
                   from AX register
```

Arithmetic Instructions
INC: Increment

- **INC Src**
- It increments the byte or word by one.
- The INC instruction adds 1 to any register or memory location, except a segment register.
- The operand can be a register or memory location.
- It effects AF, OF, PF, SF, ZF flags.
- CF is not effected.

Example:

```
INC AX      ; adds 1 to AX register
INC DX      ; adds 1 to DX register
```

Arithmetic Instructions
DEC: Increment

- **DEC Src**
- **It decrements the byte or word by one.**
- **The DEC instruction subtract 1 from any register or memory location, except a segment register.**
- **The operand can be a register or memory location.**
- **It effects AF, OF, PF, SF, ZF flags.**
- **CF is not effected.**
- **Example:**

```
DEC AX      ; sub 1 from AX register
```

Arithmetic Instructions
NEG: Negation

- **NEG Src**
- **It creates two's complement of a given number.**
- **That means, it changes the sign of a number.**
- **The arithmetic sign of a signed number changes from positive to negative or negative to positive.**
- **The CF flag cleared to 0 if the source operand is 0; otherwise it is set to 1. Other flags are set according to the result.**
- **NEG can use any addressing mode.**
- **NEG function is considered an arithmetic operation.**

Arithmetic Instructions
CMP: Compare

- **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.**

Arithmetic Instructions
CMP: Compare

- **CMP Des, Src**
- **The comparison instruction (CMP) is a subtraction that changes only the flag bits.**
- **Useful for checking the contents of a register or a memory location against another value.**
- **A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.**

Example:

```
CMP AL, 10H
JAE NEXT        ; jump if above or equal
```

Arithmetic Instructions

Sign Extension Instructions (CBW, CWD)

- **CBW**
- The CBW instruction (convert byte to word) extends the sign bit of AL into AH, preserving the number's sign.
- In the next example, 9BH (in AL) and FF9BH (in AX) both equal −101 decimal:
- Example

```
MOV AL, 9BH      ; AL = 9BH
CBW              ; AX = FF9BH
```

Arithmetic Instructions

Sign Extension Instructions (CBW, CWD)

- **CWD**
- The CWD instruction (convert word to double word) extends the sign bit of AX into DX, preserving the number's sign.
- In the next example, FF9BH (in AX) and FFFFH (in DX) :

- **Example**

```
MOV AX, FF9BH     ; AX = FF9BH
CWD               ; DX:AX = FFFFFF9BH
```

Arithmetic Instructions
MUL: Unsigned Multiplication

- **MUL Src**
- **It is an unsigned multiplication instruction.**
- **It multiplies two bytes to produce a word or two words to produce a double word.**
- **Affected flags are C and O.**
- **Set: if higher byte of result not zero**
- **Reset: the result fit exactly the lower half.**
- **Product after a multiplication always a double-width product.**

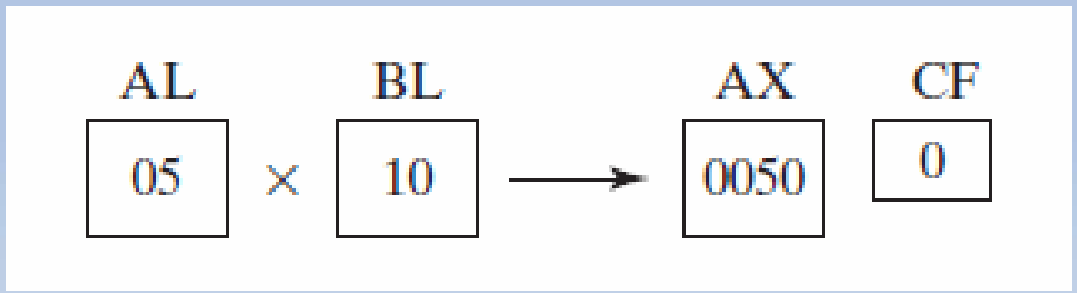
Arithmetic Instructions
MUL: Unsigned Multiplication

- **MUL Src**
- **Two 8-bit numbers multiplied generate a 16-bit product.**
- **Two 16-bit numbers multiplied generate a 32-bit product.**
- **Two 32-bit numbers multiplied generate a 64-bit product.**
- **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.**

Arithmetic Instructions
MUL: Unsigned Multiplication

- **8-Bit Multiplication**
- **The following statements multiply AL by BL, storing the product in AX. The Carry flag is clear (CF = 0) because AH (the upper half of the product) equals zero:**

```
MOV AL, 5H        ; a byte is moved to AL
MOV BL, 10H       ; immediate data must be in
                    BL register
MUL BL            ; AX = 0050h, CF = 0
```



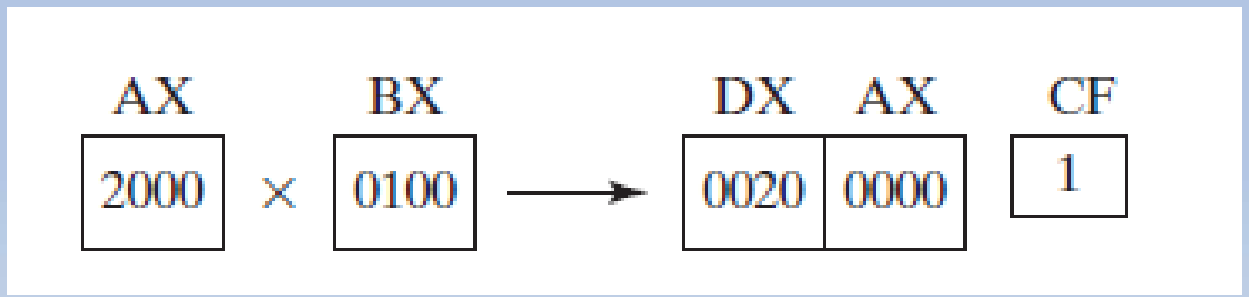
Arithmetic Instructions
MUL: Unsigned Multiplication

- **16-Bit Multiplication**
- **Word (16-bit) multiplication is very similar to byte (8-bit) multiplication.**
- **AX contains the multiplicand instead of AL.**
- **32-bit product appears in DX–AX instead of AX.**
- **The DX register always contains the most significant 16-bits (higher word) of the product; AX contains the least significant 16-bits (lower word).**
- **As with 8-bit multiplication, the choice of the multiplier is up to the programmer.**

Arithmetic Instructions
MUL: Unsigned Multiplication

- **16-Bit Multiplication**
- **The following statements multiply the 16-bit value 2000H by 0100H. The Carry flag is set because the upper part of the product (located in DX) is not equal to zero:**

```
MOV AX, 2000H      ; a word is moved to AX
MOV BX, 0100H      ; immediate data must be in
                     BX register
MUL BX             ; DX:AX = 00200000H, CF = 1
```



Arithmetic Instructions
MUL: Unsigned Multiplication

Summary of Multiplication of Unsigned Numbers

Multiplication	Operand 1	Operand 2	Result
byte x byte	AL	register or memory	AX
word x word	AX	register or memory	DX-AX
word x byte	AL=byte, AH=0	register or memory	DX-AX

Arithmetic Instructions
IMUL: Signed Multiplication

- **IMUL Src**
- **The IMUL (signed multiply) instruction performs signed integer multiplication.**
- **Unlike the MUL instruction, IMULpreserves the sign of the product.**
- **It does this by sign extending the highest bit of the lower half of the product into the upper bits of the product.**
- **In the one-operand format, the multiplier and multiplicand are the same size and the product is twice their size.**

Arithmetic Instructions
IMUL: Signed Multiplication

- **IMUL Src**
- **The following instructions multiply -4 by 4, producing (-16) in AX. AH is a sign extension of AL so the Overflow flag is clear:**

```
MOV AL, -4          ; a byte is moved to AL
MOV BL, 4           ; immediate data must
                      be in BL register
IMUL BL             ; AX = FFF0H, OF = 0
```

Arithmetic Instructions
DIV/IDIV: Unsigned/Signed Division

- **Division**
- It is an unsigned/signed division instruction.
- Occurs on 8- or 16-bit and 32-bit numbers depending on microprocessor.
- Signed (IDIV) or unsigned (DIV) integers.
- It divides word by byte or double word by word.
- There is no immediate division instruction available to any microprocessor.

Arithmetic Instructions
DIV/IDIV: Unsigned/Signed Division

- A division can result in two types of errors:
  - Attempt to divide by zero
  - Other is a divide overflow, which occurs when a small number divides into a large number
- In either case, the microprocessor generates an interrupt if a divide error occurs.
- In most systems, a divide error interrupt displays an error message on the video screen.

Arithmetic Instructions
DIV: Unsigned Division

- **Division**
- **The following table shows the relationship between the dividend, divisor, quotient, and remainder.**

Dividend	Divisor	Quotient	Remainder
AX	register or memory8	AL	AH
DX:AX	register or memory16	AX	DX

Arithmetic Instructions
DIV: Unsigned Division

- **8-Bit Division**
- **DIV Src**
- **It is an unsigned division instruction.**
- **It divides word by byte or double word by word.**
- **Uses AX to store the dividend, divided by the contents of any 8-bit register or memory location.**
- **The dividend is stored in AX, divisor is Src and the result is stored as AH = remainder, AL = quotient.**
- **Quotient is positive or negative; remainder always assumes sign of the dividend; always an integer.**

Arithmetic Instructions
DIV: Unsigned Division

- 8-Bit Division
- DIV Src
- Numbers usually 8-bits wide in 8-bit division.
- The dividend must be converted to a 16-bit wide number in AX ; accomplished differently for signed and unsigned numbers.
- For singed numbers, the least significant 8-bits are sign-extended into the most 8-bits. In the microprocessor, a special instruction sign-extends AL to AH, or convert an 8-bit singed number in AL into a 16-bit singed number in AX. CBW instruction performs this conversion.

Arithmetic Instructions
DIV: Unsigned Division

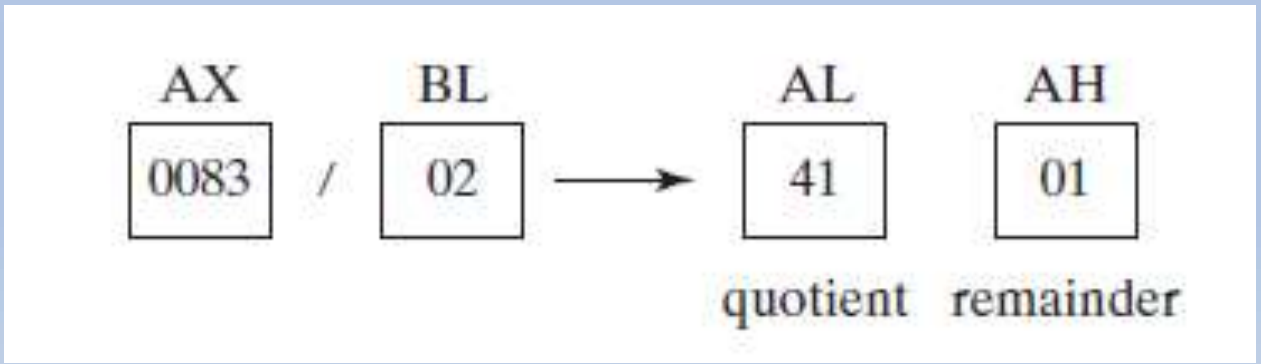
- 8-Bit Division
- DIV Src
- The DIV (unsigned divide) instruction performs 8-bit and 16-bit unsigned integer division.
- The single register or memory operand is the divisor. The formats are:

DIV reg/mem8
DIV reg/mem16

Arithmetic Instructions
DIV: Unsigned Division

- **DIV Src**
- **The following instructions perform 8-bit unsigned division (83H/2), producing a quotient of 41H and a remainder of 1:**

```
MOV AX, 0083H        ; dividend
MOV BL, 02H          ; divisor
DIV BL               ; AL=41H,AH=01H
```



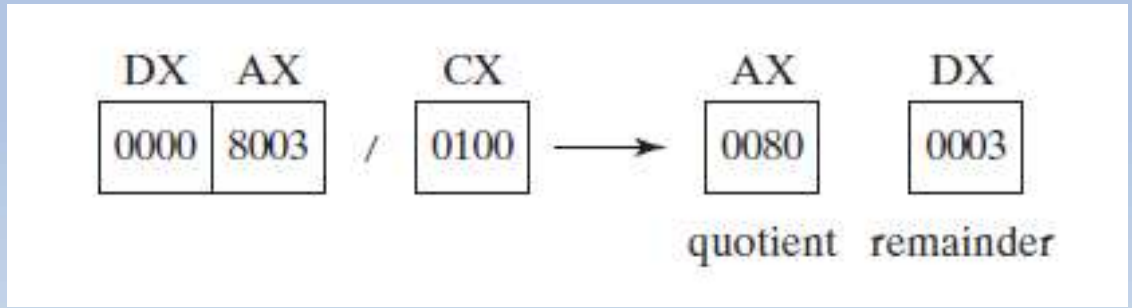
Arithmetic Instructions
DIV: Unsigned Division

- **16-Bit Division**
- **16-bit division is similar to 8-bit division.**
- **instead of dividing into AX, the 16-bit number is divided into DX:AX, a 32-bit dividend.**
- **As with 8-bit division, numbers must often be converted to the proper form for the dividend.**
- **If a 16-bit unsigned number is placed in AX, DX must be cleared to zero.**
- **If AX is a 16-bit singed number, the CWD (convert word to double word) instruction sign-extends it into a singed 32-bit number.**
- **The operand is stored in AX, divisor is Src and the result is stored as DX = remainder, AX = quotient.**

Arithmetic Instructions
DIV: Unsigned Division

- **DIV Src**
- **The following instructions perform 16-bit unsigned division (8003H/100H), producing a quotient of 80H and a remainder of 3. DX contains the high part of the dividend, so it must be cleared before the DIV instruction executes:**

```
MOV DX, 00          ; clear dividend, high
MOV AX, 8003H       ; dividend
MOV CX, 100H        ; divisor
DIV BL              ; AX=0080H, DX=0003H
```



Arithmetic Instructions
IDIV: Signed Division

- **IDIV Src**
- The IDIV (signed divide) instruction performs signed integer division, using the same operands as DIV.
- Signed integers must be sign-extended before division takes place.
- Before executing 8-bit division, the dividend (AX) must be completely sign-extended. The remainder always has the same sign as the dividend.
- Fill high byte/word/doubleword with a copy of the low byte/word/doubleword's sign bit.

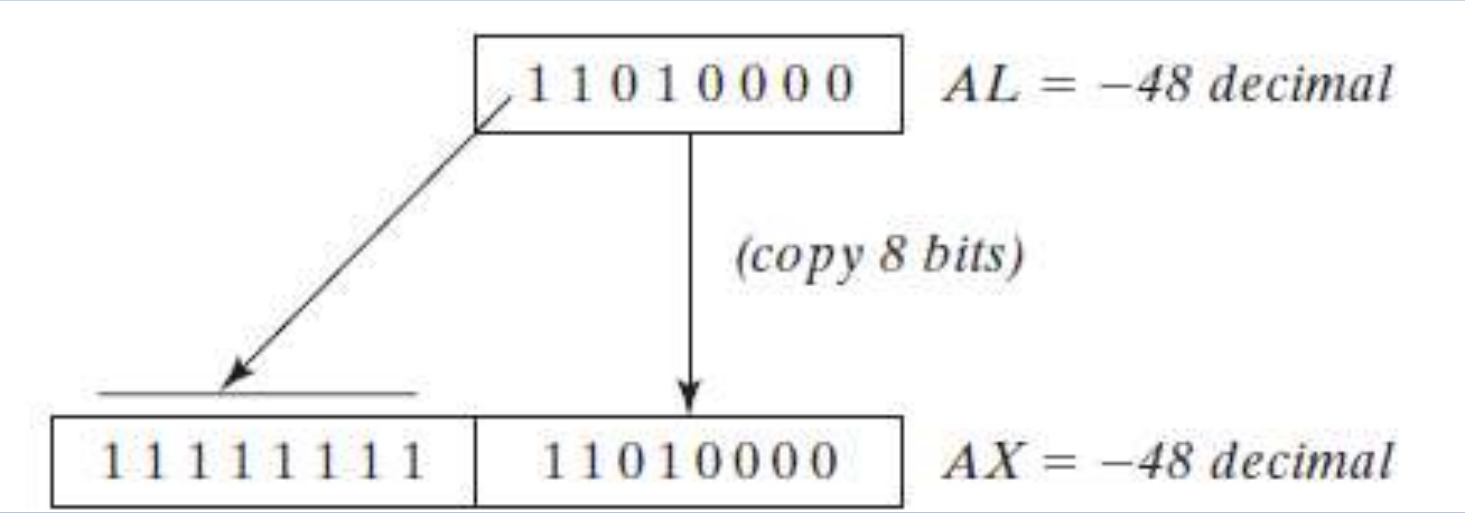
Arithmetic Instructions
IDIV: Signed Division

- **IDIV Src**
- **The following instructions divide -48 by 5. After IDIV executes, the quotient in AL is -9 and the remainder in AH is -3:**

```
MOV AL,-48          ; lower half of dividend
                      in AL register
CBW                 ; extend AL into AH
MOV BL, +5          ; divisor
IDIV BL             ; AL=-9, AH=-3
```

Arithmetic Instructions
IDIV: Signed Division

- **IDIV Src**
- **The following illustration shows how AL is sign-extended into AX by the CBW instruction:**



Arithmetic Instructions
IDIV: Signed Division

- **IDIV Src**
- To understand why sign extension of the dividend is necessary, let's repeat the previous example without using sign extension. The following code initializes AH to zero so it has a known value, and then divides without using CBW to prepare the dividend:

```
MOV AH,00          ; upper half of dividend
MOVAL,-48          ; lower half of dividend
                     in AL register

MOV BL, +5         ; divisor
IDIV BL            ; AL=41,AH=3
```

Arithmetic Instructions
IDIV: Signed Division

- **IDIV Src**

- **Before the division, AX = 00D0H (208 decimal). IDIV divides this by 5, producing a quotient of 41 decimal, and a remainder of 3. That is certainly not the correct answer.**

Arithmetic Instructions
IDIV: Signed Division

- **IDIV Src**
- **16-bit division requires AX to be sign-extended into DX. The following instructions divide -5000 by 256. After IDIV executes, the quotient in AX is -19 and the remainder in DX is -136:**

```
MOV AX,-5000        ; lower half of dividend
                      in AX register
CWD                 ; extend AX into DX
MOV BX, +256        ; divisor
IDIV BX             ; quotient AX= -19,
                      remainder DX= -136
```

Arithmetic Instructions
DIV: Divide Overflow

- If a division operand produces a quotient that will not fit into the destination operand, a divide overflow condition results. This causes a CPU interrupt, and the current program halts. The following instructions, for example, generate a divide overflow because the quotient (100H) will not fit into the AL register:

```
MOV AX, 1000H      ; dividend in AX register
MOV BL, 10H        ; divisor
DIV BL             ; quotient AL cannot hold
                     100H
```

Arithmetic Instructions
BCD and ASCII Arithmetic

- The microprocessor allows arithmetic manipulation of both BCD (binary-coded decimal) and ASCII (American Standard Code for Information Interchange) data.
- BCD operations occur in systems such as point-of-sales terminals (e.g., cash registers) and others that seldom require complex arithmetic.

Arithmetic Instructions
BCD Number System

- In computer literature one encounters two terms for BCD numbers.
- Unpacked BCD: the lower 4 bits of the number represent the BCD number and the rest of the bits are 0.
- Example: 0000 1001 and 0000 0101 are unpacked BCD for 9 and 5, respectively.
- Packed BCD: a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits.
- Example: 0101 1001 is packed BCD for 59. It takes only a byte of memory to store the packed BCD operands.

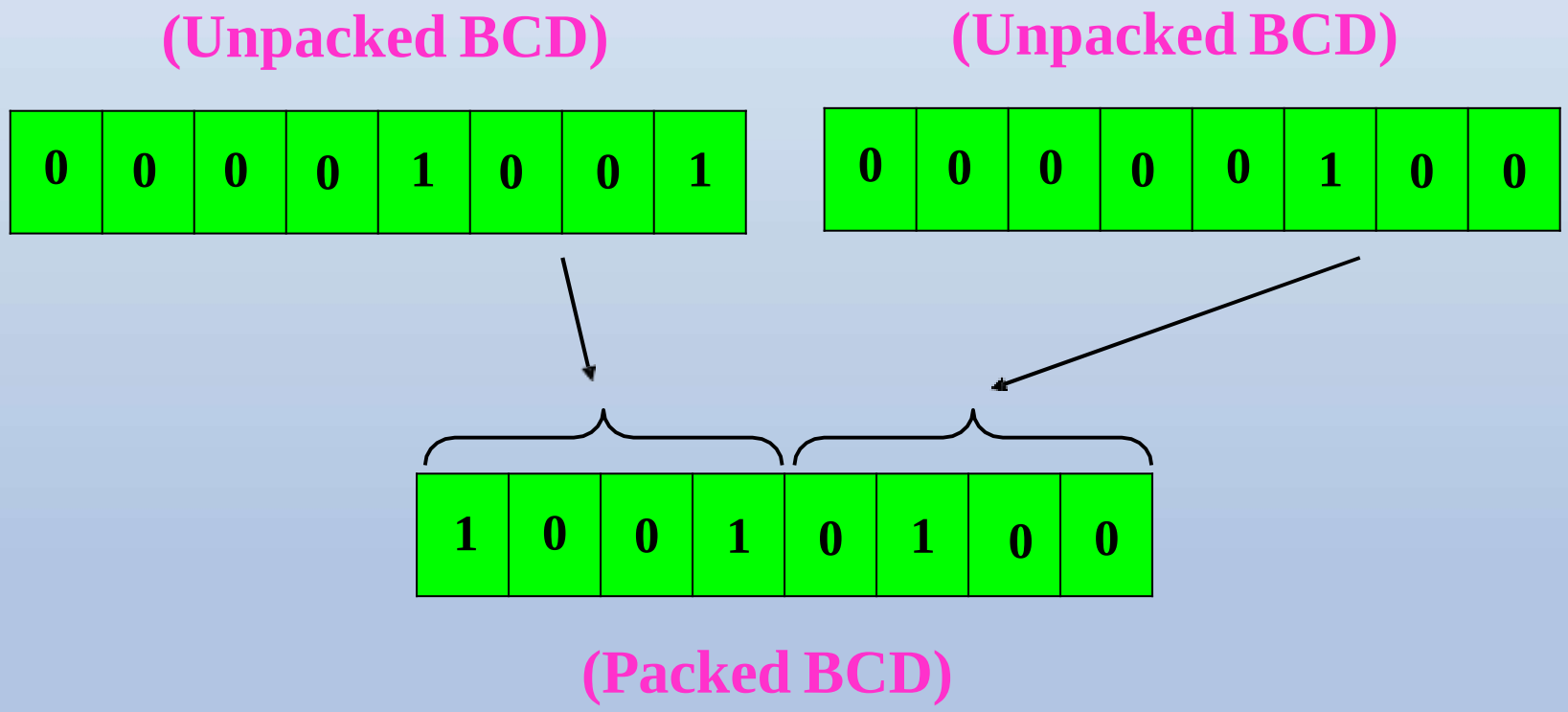
Arithmetic Instructions
BCD Number System

Digit	BCD
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
4	0000 0100
5	0000 0101
6	0000 0110
7	0000 0111
8	0000 1000
9	0000 1001

- A BCD number can only have digits from 0000 to 1001 (0 to 9).
- To represent 10 digits only 4 bits are needed.
- Using 8 bits is not optimal → 4 bits are lost.
- 8 bits can be used to store 1 BCD number (Unpacked BCD)
- 8 bits can be used to store 2 BCD numbers (Packed BCD)

Arithmetic Instructions
BCD Number System

- Use 8 bits to store 2 BCD digits



Arithmetic Instructions
ASCII Number System

Digit	ASCII
0	(30H) 011 0000
1	(31H) 011 0001
2	(32H) 011 0010
3	(33H) 011 0011
4	(34H) 011 0100
5	(35H) 011 0101
6	(36H) 011 0110
7	(37H) 011 0111
8	(38H) 011 1000
9	(39H) 011 1001

- 7-bit representation.
- Keyboards, printers, and monitors are all in ASCII.
- To process data in BCD, ASCII data should be converted first to BCD.
- Remember that ASCII code representation of a number is : number + 30
- ASCII Code for (3) is : 3 +30 = $(33)_{ASCII}$

Arithmetic Instructions
BCD versus ASCII

Digit	BCD	ASCII
0	0000 0000	011 0000
1	0000 0001	011 0001
2	0000 0010	011 0010
3	0000 0011	011 0011
4	0000 0100	011 0100
5	0000 0101	011 0101
6	0000 0110	011 0110
7	0000 0111	011 0111
8	0000 1000	011 1000
9	0000 1001	011 1001

Arithmetic Instructions
BCD Arithmetic

- Two arithmetic techniques operate with BCD data: addition and subtraction.
- DAA (decimal adjust after addition) instruction follows BCD addition.
- It is used to make sure that the result of adding two BCD numbers is adjusted to be a correct BCD number.
- It only works on AL register.

Arithmetic Instructions
BCD Arithmetic

- DAS (decimal adjust after subtraction) follows BCD subtraction.
- It is used to make sure that the result of subtracting two BCD numbers is adjusted to be a correct BCD number.
- It only works on AL register.
- Both DAA and DAS correct the result of addition or subtraction so it is a BCD number

Arithmetic Instructions
DAA Instruction

- The DAA instruction works as follows :

```
MOV AL, 71H    ; load 71 into AL
ADD AL, 43H    ; AL 71H+43H =B4H
DAA            ; AL = 14 H and CF= 1
```

- If the least significant four bits in AL are >9 or if AF=1, it adds 6 to AL and sets AF.
- If the most significant four bits in AL are >9 or if CF=1, it adds 60 to AL and sets the CF.

Arithmetic Instructions
DAS Instruction

- The DAS instruction works as follows :

```
MOV AL, 71H      ; load 71 into AL
SUB AL, 43H      ; AL 71H-43H=2EH
DAS              ; AL = 28 H
```

- 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 60 from AL and sets the CF.

Arithmetic Instructions
ASCII Arithmetic

- ASCII arithmetic instructions function with coded numbers, value 30H to 39H for 0–9.
- Four instructions in ASCII arithmetic operations:
  - AAA (ASCII adjust after addition)
  - AAD (ASCII adjust before division)
  - AAM (ASCII adjust after multiplication)
  - AAS (ASCII adjust after subtraction)

Arithmetic Instructions
AAA Arithmetic

- AAA (ASCII adjust after addition):
  - The data entered from the terminal is in ASCII format.
  - In ASCII, 0 – 9 are represented by 30H – 39H.
  - This instruction allows us to add the ASCII codes.
  - This instruction does not have any operand.

Arithmetic Instructions
AAA Arithmetic

- AAA (ASCII adjust after addition):

- Example:

```
MOV   AL, '5'     ;  AL = 35
ADD   AL, '2'     ;  add to AL 32 the
                     ASCII of 2, (AL) = 35 + 32
AAA               ; changes 67H to 07H,
                     AL = 7
OR    AL, 30      ; OR AL with 30 to get ASCII
```

Arithmetic Instructions
AAA Arithmetic

- AAA (ASCII adjust after addition):

- Example:

```
MOV AX, 31H          ; AX = 0031H
ADD AL, 39H          ; AX = 006AH
AAA                  ; AX = 0100H
ADD AX, 3030H        ; AX = 3130 which is
                       the ASCII for 10H
```

Arithmetic Instructions
AAA Arithmetic

- AAA (ASCII adjust after addition):

- Example:

```
SUB AH, AH          ; AH = 00H
MOV AL,'7'          ; AL = 37H
MOV BL, '5'         ; BL = 35H
ADD AL, BL          ; AL= 37H+35H=6CH
AAA                 ; changes 6CH to 02 in
                      AL and AH=CF=1
OR AX, 3030H        ; AX = 3132 which is
                      the ASCII for 12H
```

Arithmetic Instructions
AAS Arithmetic

- AAS (ASCII adjust after subtraction):
- Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result.
- The AL register is the implied source and destination operand for this instruction.
- only useful when it follows an SUB instruction.

Arithmetic Instructions
AAS Arithmetic

- **AAS (ASCII adjust after subtraction):**

- **Example: Positive Result**

```
SUB AH,AH          ; AH = 00H
MOV AL,'9'         ; AL = 39H
SUB AL, '3'        ; AL =39H-33H = 06H
AAS                ; AX=0006H
OR AL, 30H         ; AL = 36H
```

Arithmetic Instructions
AAS Arithmetic

- **AAS (ASCII adjust after subtraction):**

- **Example: Negative Result**

```
SUB AH,AH          ; AH = 00H
MOV AL,'3'         ; AL = 33H
SUB AL, '9'        ; AL =33H-39H = FAH
AAS                ; AX=FF04H
OR AL, 30H         ; AL = 34H
```

Arithmetic Instructions
AAM Instruction

- AAM (ASCII adjust after multiplication):
- Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked BCD values.
- The AX register is the implied source and destination.
- The AAM instruction is only useful when it follows an MUL instruction.

Arithmetic Instructions
AAM Instruction

- **AAM (ASCII adjust after multiplication):**

- **Example:**

```
MOV  BL,5          ; BL = 5H
MOV AL, 6          ; AL = 6H
MUL DL             ; AX=ALxDL=05x06
                         =001EH
AAM                ; AX=0300
```

Arithmetic Instructions
AAD Instruction

- AAD (ASCII adjust before division):
- Appears before a division.
- The AAD instruction requires the AX register contain a two-digit unpacked BCD number (not ASCII) before executing.
- Before dividing the unpacked BCD by another unpacked BCD, AAD is used to convert it to HEX. By doing that the quotient and reminder are both in unpacked BCD.

Arithmetic Instructions
AAD Instruction

- **AAD (ASCII adjust after division):**

- **Example:**

```
MOV AX, 0307H        ; AX = 0307H
AAD                  ; AX=0025H
MOV BL, 5H           ; divide by 05
DIV BL               ; AX=0207
```

Logic (Bit Manipulation) Instructions
AND, OR, XOR, NOT

- Include AND, OR, Exclusive-OR, and NOT.
- 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
- Logic operations provide binary bit control in low-level software. Allow bits to be set, cleared, or complemented.
- All logic instructions affect the flag bits.

Logic (Bit Manipulation) Instructions
AND: Instruction

- Performs logical multiplication, illustrated by a truth table.
- AND can replace discrete AND gates if the speed required is not too great. Normally reserved for embedded control applications.
- In 8086, the AND instruction often executes in about a microsecond.
- With newer versions, the execution speed is greatly increased.

Logic (Bit Manipulation) Instructions
AND: Instruction

- AND clears bits of a binary number called masking.
- AND uses any mode except memory-to-memory and segment register addressing.
- An ASCII number can be converted to BCD by using AND to mask off the leftmost four binary bit positions.

```
MOV BX, 3135H
AND BX, 0F0FH
```

	x x x x x x x x	Unknown number
•	0 0 0 0 1 1 1 1	Mask
	0 0 0 0 x x x x	Result

Logic (Bit Manipulation) Instructions
AND: Instruction

- **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.**

Logic (Bit Manipulation) Instructions
AND: Instruction

- **AND Des, Src:**
- **The AND instruction performs a boolean (bitwise) AND operation between each pair of matching bits in two operands and places the result in the destination operand.**
- **The following operand combinations are permitted:**

AND reg, reg
AND reg, mem
AND reg, imm
AND mem, reg
AND mem, imm

Logic (Bit Manipulation) Instructions
OR: Instruction

- Performs logical addition, illustrated by a truth table. Often called the Inclusive-OR function.
- The OR function generates a logic 1 output if any inputs are 1. A 0 appears at output only when all inputs are 0.

Logic (Bit Manipulation) Instructions
OR: Instruction

- OR sets bits of a binary number.
- OR uses any mode except memory-to-memory and segment register addressing.
- The operation of the OR function showing how bits of a number are set to one.

```
  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
```

Logic (Bit Manipulation) Instructions
OR: Instruction

- **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.

Logic (Bit Manipulation) Instructions
OR: Instruction

- **OR Des, Src:**
- **The OR instruction performs a boolean OR operation between each pair of matching bits in two operands and places the result in the destination operand.**
- **The following operand combinations are permitted:**

OR reg, reg

OR reg, mem

OR reg, imm

OR mem, reg

OR mem, imm

Logic (Bit Manipulation) Instructions
Exclusive-OR (XOR): Instruction

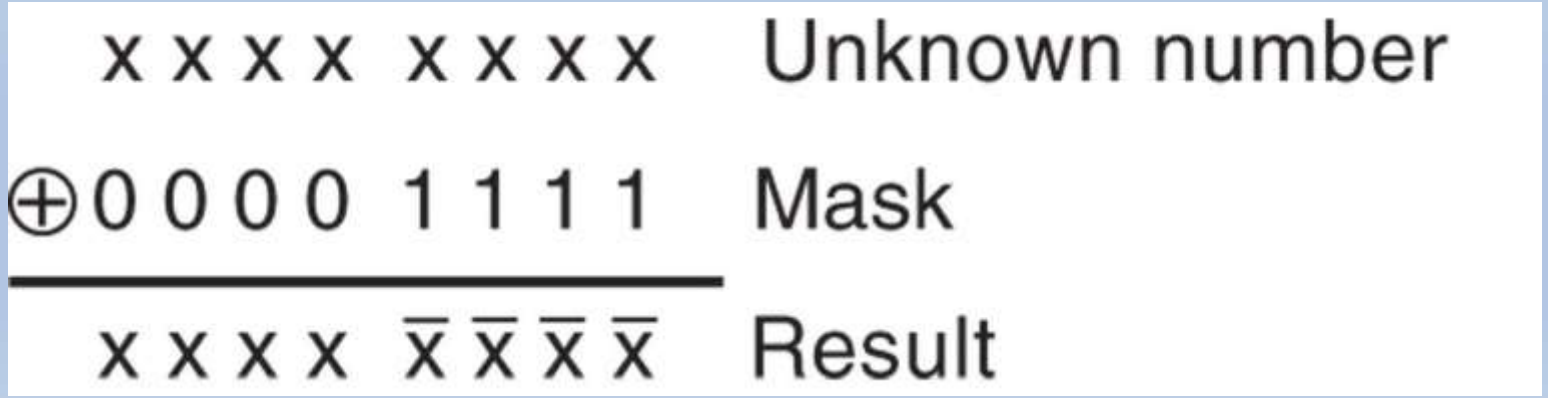
- Differs from Inclusive-OR (OR) in that the 1,1 condition of Exclusive-OR produces a 0. A 1,1 condition of the OR function produces a 1.
- The Exclusive-OR (XOR) operation excludes this condition; the Inclusive-OR includes it.
- If inputs of the Exclusive-OR function are both 0 or both 1, the output is 0; if the inputs are different, the output is 1.
- Exclusive-OR is sometimes called a comparator.

Logic (Bit Manipulation) Instructions
Exclusive-OR (XOR): Instruction

- XOR uses any addressing mode except segment register addressing.
- The XOR is useful if some bits of a register or memory location must be inverted.
- XOR operation shows that how just part of an unknown quantity can be inverted by XOR.
  When a 1 Exclusive-ORs with X, the result is X.
  If a 0 Exclusive-ORs with X, the result is X.
- A common use for the Exclusive-OR instruction is to clear a register to zero.

Logic (Bit Manipulation) Instructions
Exclusive-OR (XOR): Instruction

- The XOR instruction uses the same operand combinations and sizes as the AND and OR instructions.
- For each matching bit in the two operands, the following applies: If both bits are the same (both 0 or both 1), the result is 0; otherwise, the result is 1.



Logic (Bit Manipulation) Instructions
Exclusive-OR (XOR): Instruction

- **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.**

Logic (Bit Manipulation) Instructions
Exclusive-OR (XOR): Instruction

- **XOR Des, Src:**
- **Example:**

```
MOV  AL, 54H      ; AL=01010100 b
XOR  AL, 87H      ; result in AL=00101100 b
```

54H	0101	0100
78H	0111	1000
2CH	0010	1100

Logic (Bit Manipulation) Instructions
Exclusive-OR (XOR): Instruction

- **XOR Des, Src:**
- **Example: Clearing the contents of register.**

```
MOV  AL, 54H      ; AL=01010100 b
XOR  AL, AL       ; result in AL=00000000 b
```

- **Flags: SF = CF = OF = 0; ZF = PF = 1**

54H	0101	0100
54H	0101	0100
00H	0000	0000

Logic (Bit Manipulation) Instructions
Exclusive-OR (XOR): Instruction

- **XOR Des, Src:**
- **Example: Bit toggle.**

```
MOV  AL, 54H      ; AL=01010100 b
XOR  AL, 02H      ; Toggle bit No. 2
```

```
54H      0101 0100
02H      0000 0010
56H      0101 0110
```

TEST: Logical Compare Instruction The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to 1, if the corresponding bits of both operands are 1, else the result bit is reset to 0. The result of this ANDing operation is not available for further use, but flags are affected. The affected flags are OF, CF, SF, ZF and PF. The operands may be registers, memory or immediate data. The examples of this instruction are as follows:

Example 2.41

```
1.TEST    AX, BX
2.TEST    [0500], 06H
3.TEST    [BX] [DI], CX
```

Logic (Bit Manipulation) Instructions
NOT: Instruction

- **NOT Src:**
- **It complements each bit of Src to produce one's complement of the specified operand.**
- **The operand can be a register or memory location.**
- **NOT can use any addressing mode except segment register addressing.**
- **The NOT function is considered logical, NEG function is considered an arithmetic operation.**
- **None of flags are affected by NOT instruction.**

Logic (Bit Manipulation) Instructions
NOT: Instruction

- **NOT Src:**
- **The NOT instruction toggles (inverts) all bits in an operand.**
- **The following operand combinations are permitted:**

```
NOT  reg
NOT  mem
```

- **Example: The one's complement of F0h is 0Fh:**

```
MOV  AL, F0H          ; AL=11110000b
NOT  AL               ; AL=00001111b
```

Logic (Bit Manipulation) Instructions
AND, OR, XOR, NOT

- **Example:**

```
MOV  AL, 55H     ; AL=01010101b
AND  AL, 1FH     ; AL=15H=00010101b,
                   clear bit 7
OR   AL, C0H     ; AL=D5H=11010101b,
                   set bits 1, 3, 5, 7, 8
XOR  AL, 0FH     ; AL=DAH=11011010b,
                   invert bits 1, 2, 3, 4
NOT  AL          ; AL=25H=00100101b,
                   toggles (invert) all bits
```

Logic (Bit Manipulation) Instructions
Shift and Rotate

- **Shift and rotate instructions manipulate binary numbers at the binary bit level as did AND, OR, Exclusive-OR, and NOT.**
- **Common applications in low-level software used to control I/O devices.**
- **The microprocessor contains a complete complement of shift and rotate instructions that are used to shift or rotate any memory data or register.**

Logic (Bit Manipulation) Instructions
Shift

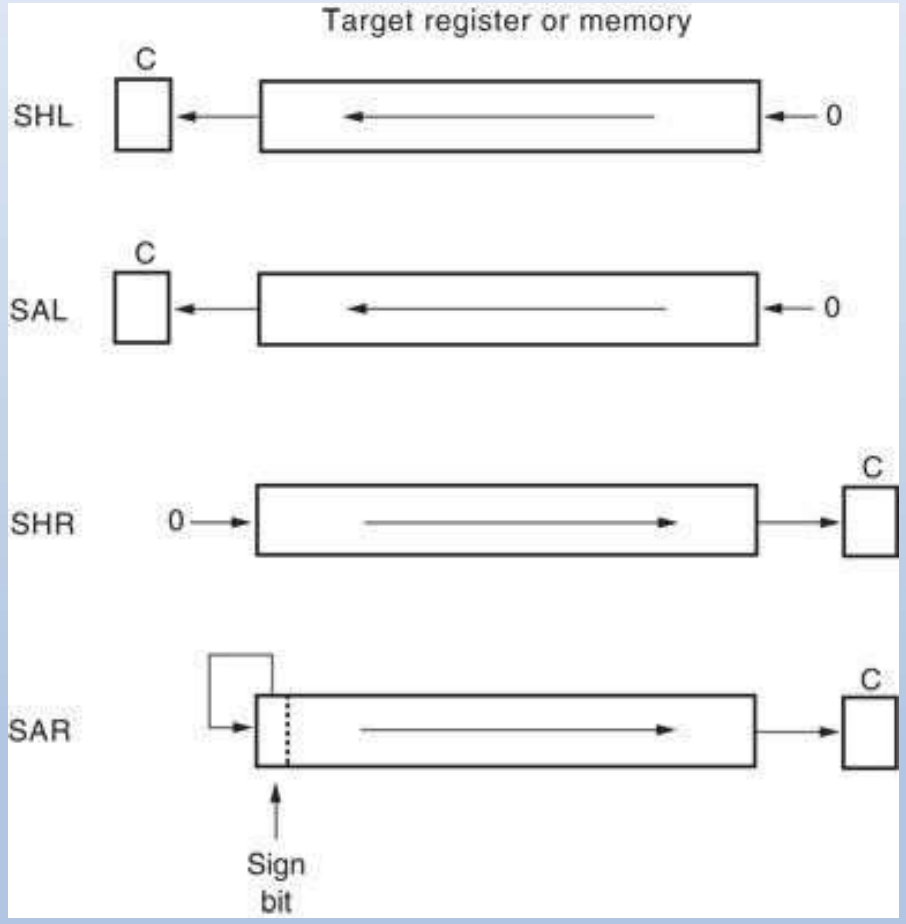
- Shift: position or move numbers to the left or right within a register or memory location.
- The microprocessor's instruction set contains four different shift instructions:
  - Two Logical shift
  - Two Arithmetic shift
- Logical shift function with unsigned numbers.
- Arithmetic shift function with signed numbers.

Logic (Bit Manipulation) Instructions
Shift

- Logical shifts move 0 in the rightmost bit for a logical left shift.
- The arithmetic shift left is identical to the logical shift left.
- 0 to the leftmost bit position for a logical right shift.
- The arithmetic right shift copies the sign-bit through the number.

Logic (Bit Manipulation) Instructions

Shift



The shift instructions showing the operation and direction of the shift.

Logic (Bit Manipulation) Instructions
Shift

- Logical shifts multiply or divide unsigned data; arithmetic shifts multiply or divide signed data.
  - A shift left always multiplies by 2 for each bit position shifted.
  - A shift right always divides by 2 for each position.
  - Shifting a two places, multiplies or divides by 4.
- Segment shift not allowed.

Logic (Bit Manipulation) Instructions
Shift Instructions

- Shifting means to move bits right and left inside an operand.
- All four shifting instruction affecting the Overflow and Carry flags.

Mnemonic	Meaning
SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right

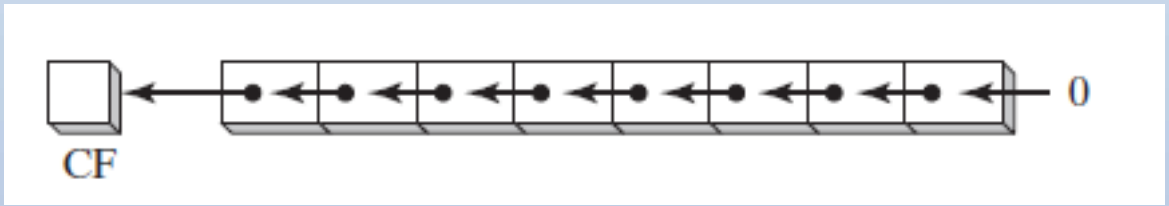
Logic (Bit Manipulation) Instructions
SHL: Shift Left

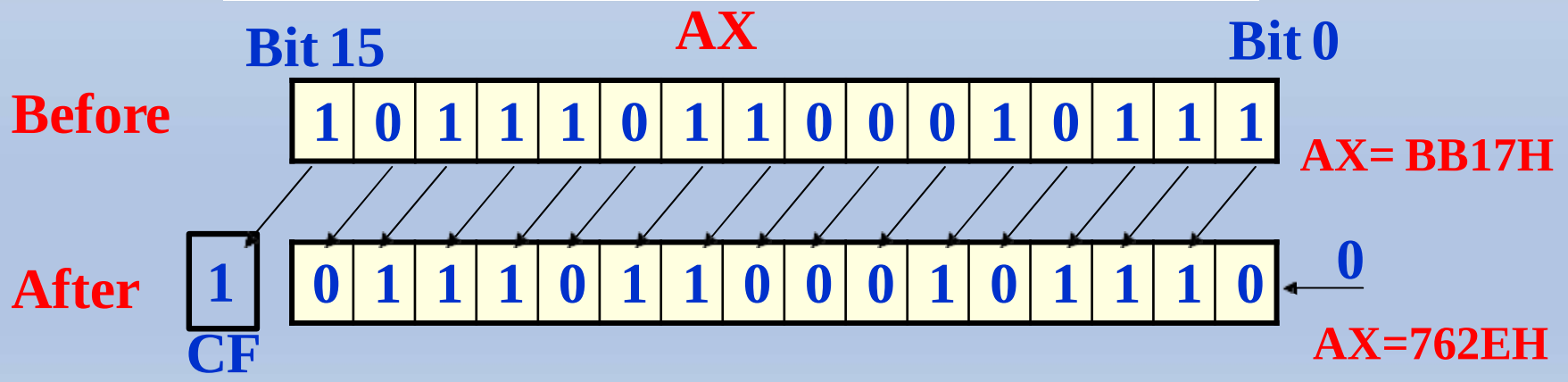
- **SHL Des, Count:**
- **It shift bits of byte or word left, by count.**
- **It puts zero(s) in least significant bits (LSBs).**
- **Most significant bit (MSB) is shifted into carry flag.**
- **If the number of bits desired to be shifted left is 1, then the immediate number 1 can be written in Count.**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

Logic (Bit Manipulation) Instructions
SHL: Shift Left

- SHL Des, Count:
- Example:

```
MOV  AX, BB17H     ; AX=1011101100010111b
SHL  AX, 1         ; AX=0111011000101110b, CF=1
```





Logic (Bit Manipulation) Instructions
SHL: Shift Left

- **SHL Des, Count:**
- **Example: In the following instructions, BL is shifted once to the left. The highest bit is copied into the Carry flag and the lowest bit position is assigned zero:**

```
MOV  BL, 8FH      ; BL=10001111b
SHL  BL, 1        ; BL=00011110b, CF=1
```

Logic (Bit Manipulation) Instructions
SHL: Shift Left

- **SHL Des, Count:**
- **Bitwise Multiplication : SHL can perform multiplication by powers of 2. Shifting any operand left by n bits multiplies the operand by 2^n.**
- **Example: In the following instructions, shifting the integer 5 left by 1 bit yields the product of $5x2^1= 10$:**

```
MOV  DL, 5        ; DL=00000101b
SHL  DL, 1        ; DL=00001010b, CF=0
```

Logic (Bit Manipulation) Instructions
SHL: Shift Left

- **SHL Des, Count:**
- **The following lists the types of operands permitted by this instruction:**

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

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

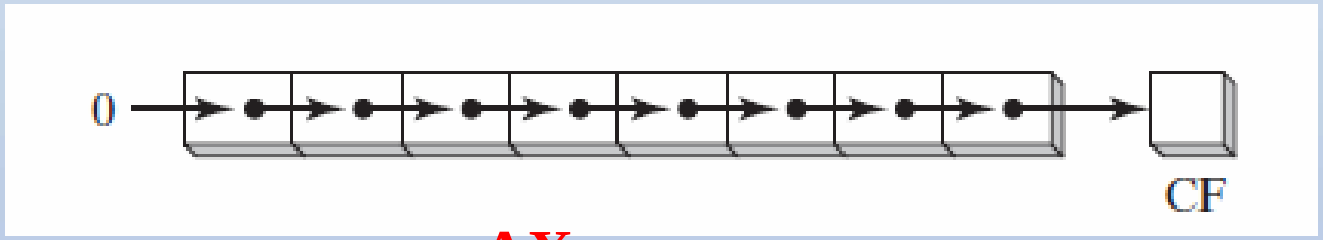
Logic (Bit Manipulation) Instructions
SHR: Shift Right

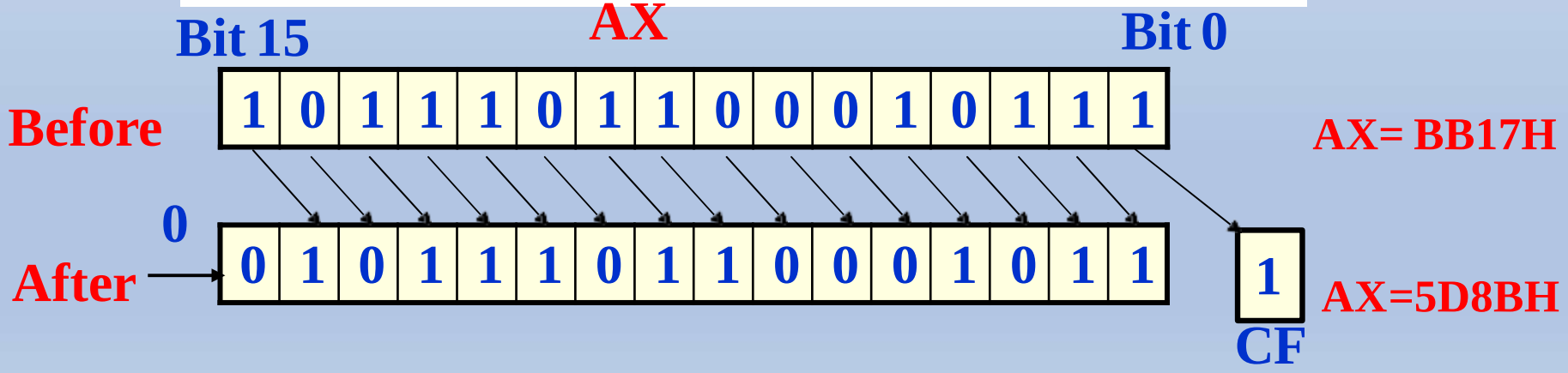
- **SHR Des, Count:**
- It shift bits of byte or word right, by count.
- It puts zero(s) in most significant bits (MSBs).
- Least significant bit (LSB) is shifted into carry flag.
- If the number of bits desired to be shifted right is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted right is more than 1, then the count is put in CL register

Logic (Bit Manipulation) Instructions
SHR: Shift Right

- SHR Des, Count:
- Example:

```
MOV  AX, BB17H     ; AX=1011101100010111b
SHR  AX, 1         ; AX=0101110110001011b, CF=1
```





Logic (Bit Manipulation) Instructions
SHR: Shift Right

- **SHR Des, Count:**
- **Example: In the following instructions, AL is shifted once to the right. The lowest bit is copied into the Carry flag and the highest bit position is filled zero:**

```
MOV  AL, D0H      ; AL=11010000b
SHR  AL, 1        ; AL=01101000b, CF=0
```

Logic (Bit Manipulation) Instructions
SHR: Shift Right

- **SHR Des, Count:**
- **Bitwise Division : SHR Logically shifting an unsigned integer right by n bits divides the operand by 2^n. Shifting any operand right by n bits divides the operand by 2^n.**
- **Example: In the following instructions, shifting the integer 32 decimal (20H) right by 2 bit yields the division of $32/2^1$= 16 decimal (10H):**

```
MOV  DL, 20H    ; DL=00100000b
SHR  DL, 1      ; DL=00010000b, CF=0
```

Logic (Bit Manipulation) Instructions
SHR: Shift Right

- **SHR Des, Count:**
- **The following lists the types of operands permitted by this instruction:**

SHR reg, imm8
SHR mem, imm8
SHR reg, CL
SHR mem, CL

- **The first operand in SHR is the destination and the second is the shift count.**

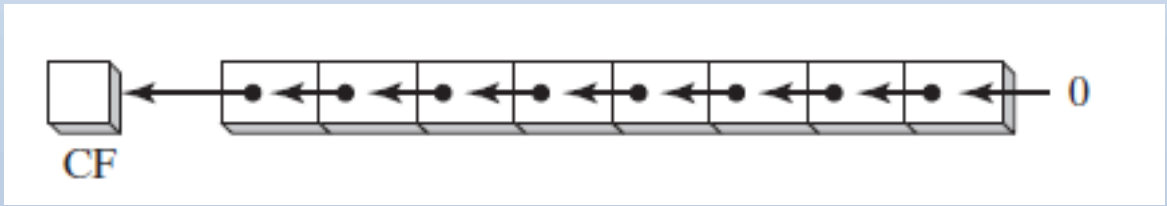
Logic (Bit Manipulation) Instructions
SAL: Shift Arithmetic Left

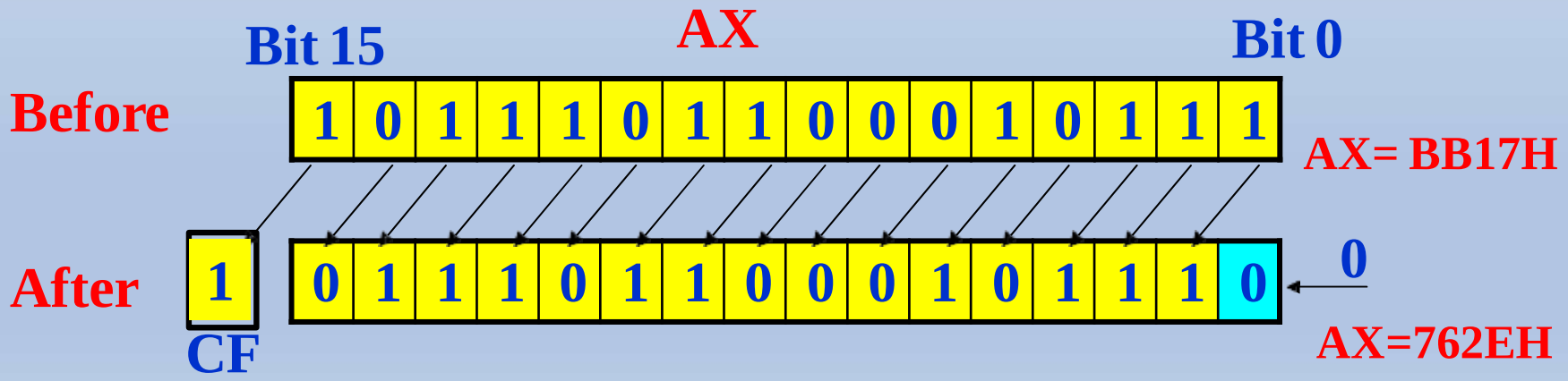
- **SAL Des, Count:**
- **The SAL (shift arithmetic left) instruction works the same as the SHL instruction but for signed numbers.**
- **It shift bits of byte or word left, by count.**
- **It puts zero(s) in least significant bits (LSBs).**
- **Most significant bit (MSB) is shifted into carry flag and the bit that was in the Carry flag is discarded.**
- **If the number of bits desired to be shifted left is 1, then the immediate number 1 can be written in Count.**
- **However, if the number of bits to be shifted left is more than 1, then the count is put in CL register.**

Logic (Bit Manipulation) Instructions
SAL: Shift Arithmetic Left

- SAL Des, Count:
- Example:

```
MOV  AX, BB17H     ; AX=1011101100010111b
SAL  AX, 1         ; AX=0111011000101110b, CF=1
```





Logic (Bit Manipulation) Instructions
SAL: Shift Arithmetic Left

- **SAL Des, Count:**
- **Example:** In the following instructions, BL is shifted once to the left. The highest bit is copied into the Carry flag and the lowest bit position is assigned zero:

```
MOV  BL, F0H      ; BL=11110000b
SAL  BL, 1        ; BL=11100000b, CF=1
```

Before: BL=11110000b (-16 decimal, F0H)
After: BL=11100000b (-32 decimal, E0H) , CF=1

Logic (Bit Manipulation) Instructions
SAL: Shift Arithmetic Left

- **SAL Des, Count:**
- **The following lists the types of operands permitted by this instruction:**

```
SAL  reg, imm8
SAL  mem, imm8
SAL  reg, CL
SAL  mem, CL
```

- **The first operand in SAL is the destination and the second is the shift count.**

Logic (Bit Manipulation) Instructions
SAR: Shift Arithmetic Right

- **SAR Des, Count:**
- **The SAR (shift arithmetic right) instruction works the same as the SHR instruction but for signed numbers.**
- **The vacated bits at the left are filled with the value of the original MSB of the operand.**
- **Thus, the original sign of the number is maintained.**
- **It shifts bits of byte or word right, by count.**

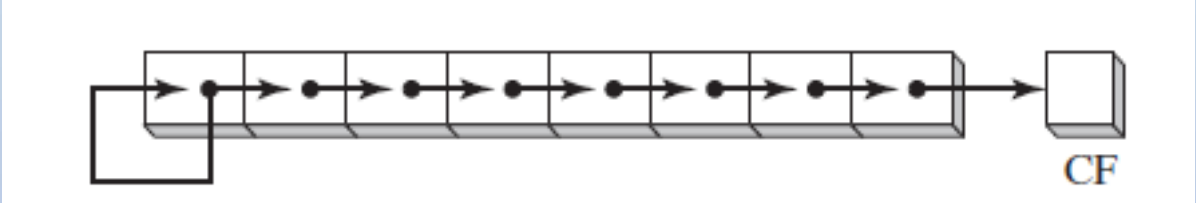
Logic (Bit Manipulation) Instructions
SAR: Shift Arithmetic Right

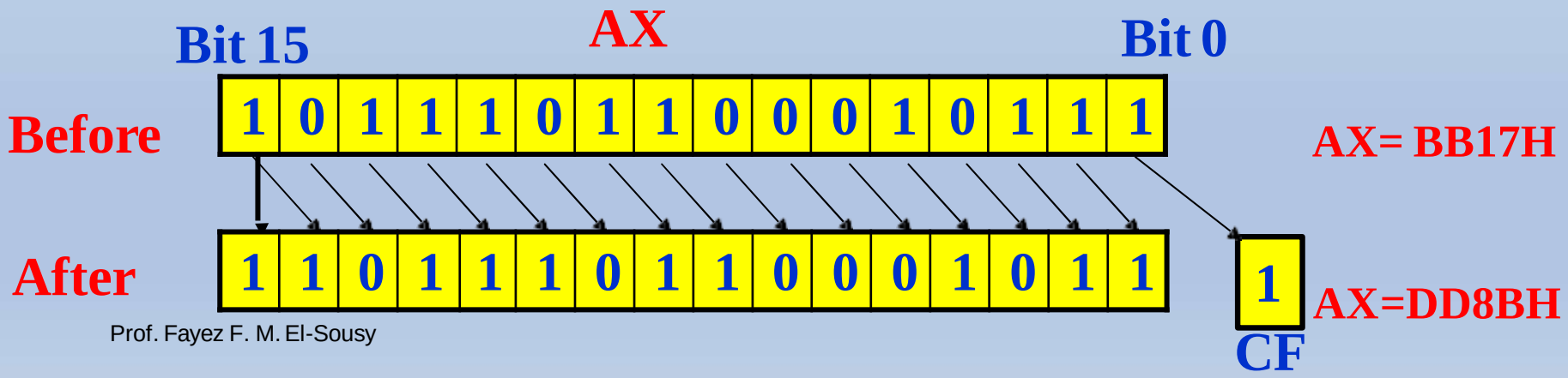
- **SAR Des, Count:**
- **Least significant bit (LSB) is shifted into carry flag.**
- **If the number of bits desired to be shifted right is 1, then the immediate number 1 can be written in Count.**
- **However, if the number of bits to be shifted right is more than 1, then the count is put in CL register.**

Logic (Bit Manipulation) Instructions
SAR: Shift Arithmetic Right

- **SAR Des, Count:**
- **Example:**

```
MOV  AX, BB17H     ; AX=1011101100010111b
SHR  AX, 1         ; AX=1101110110001011b, CF=1
```





Logic (Bit Manipulation) Instructions
SAR: Shift Arithmetic Right

- **SAR Des, Count:**
- **Example: In the following instructions, AL is shifted once to the right. The lowest bit is copied into the Carry flag and the highest bit position is filled with the value of the original MSB :**

```
MOV  AL, F0H      ; AL=11110000b, (-16d)
SAR  AL, 1        ; AL=11111000b, (-8d) CF=0
```

Before: AL=11110000b (-16 decimal, F0H)
After: AL=11111000b (-8 decimal, F8H) , CF=0

Logic (Bit Manipulation) Instructions
SAR: Shift Arithmetic Right

- **SAR Des, Count:**
- **Multiple Shifts:** When a value is shifted arithmetic rightward multiple times, the Carry flag contains the last bit to be shifted out of the least significant bit (LSB).
- **Example:** In the following instructions, AL is shifted arithmetic triple to the right. -128 is divided by 2^3. After SAR, AL=F0H (-16d).

```
MOV  AL, 80H    ; AL=10000000b (-128d)
SAR  AL, 3      ; AL=11110000b (-16d), CF=0
```

Logic (Bit Manipulation) Instructions
SAR: Shift Arithmetic Right

- **SAR Des, Count:**
- **The following lists the types of operands permitted by this instruction:**

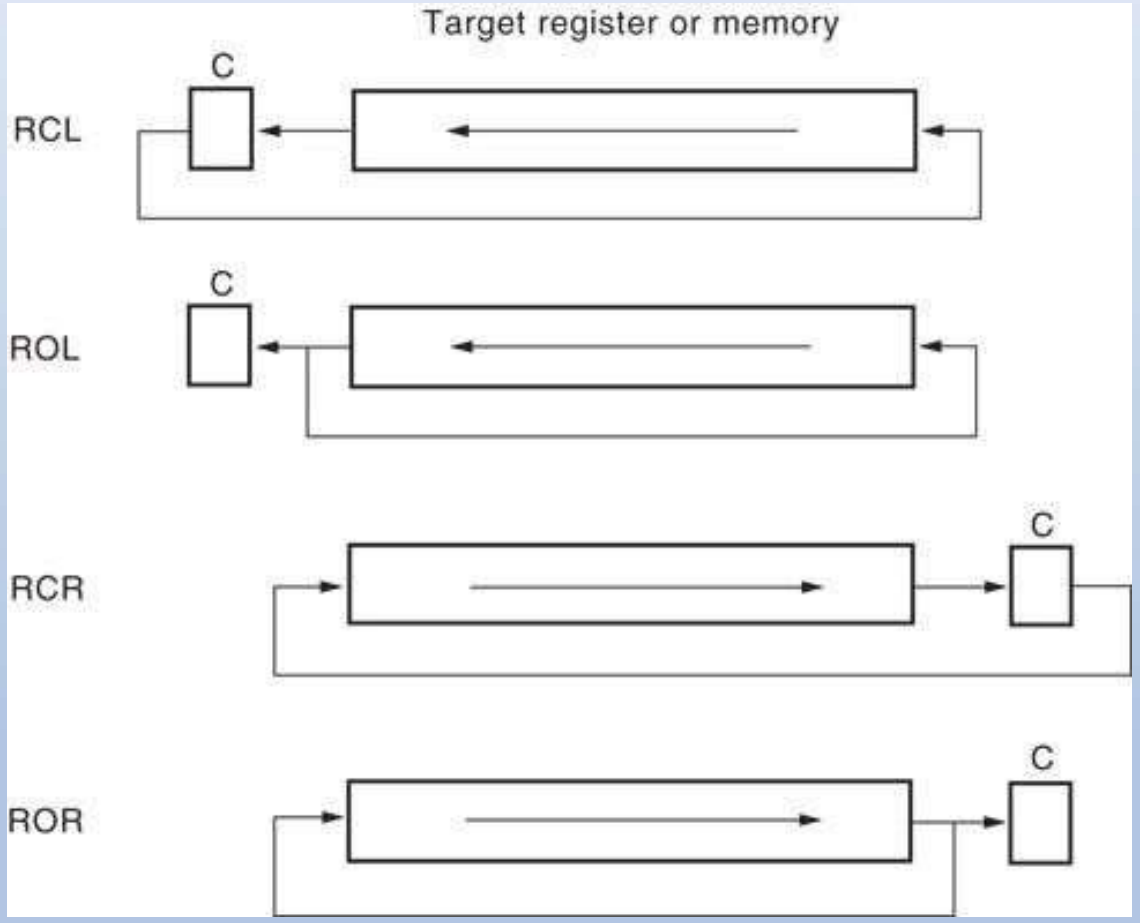
```
SAR  reg, imm8
SAR  mem, imm8
SAR  reg, CL
SAR  mem, CL
```

- **The first operand in SAR is the destination and the second is the shift count.**

Logic (Bit Manipulation) Instructions
Rotate

- **Rotate: Positions binary data by rotating information in a register or memory location, either from one end to another or through the carry flag.**
- **With either type of instruction, the programmer can select either a left or a right rotate.**
- **Addressing modes used with rotate are the same as those used with shifts.**
- **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.**

Logic (Bit Manipulation) Instructions
Rotate



The rotate instructions showing the operation and direction of each rotate.

Logic (Bit Manipulation) Instructions
Rotate Instructions

- Rotating means each bit shifted from rightmost/leftmost entered from leftmost/rightmost side inside an operand.
- All four rotating instructions affecting the Carry flag and Overflow flag undefined if count not = 1.

Mnemonic	Meaning
ROL	Rotate left
ROR	Rotate right
RCL	Rotate left through carry
RCR	Rotate right through carry

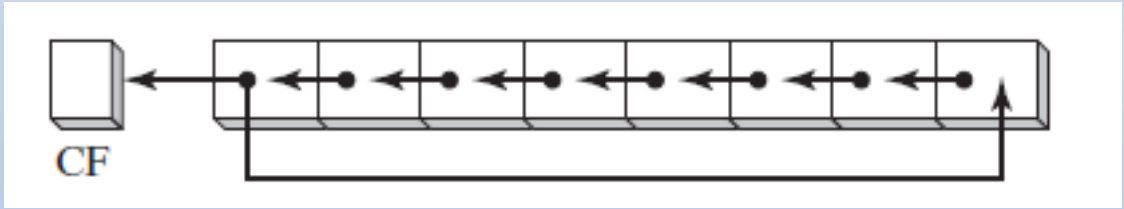
Logic (Bit Manipulation) Instructions
ROL: Rotate Left

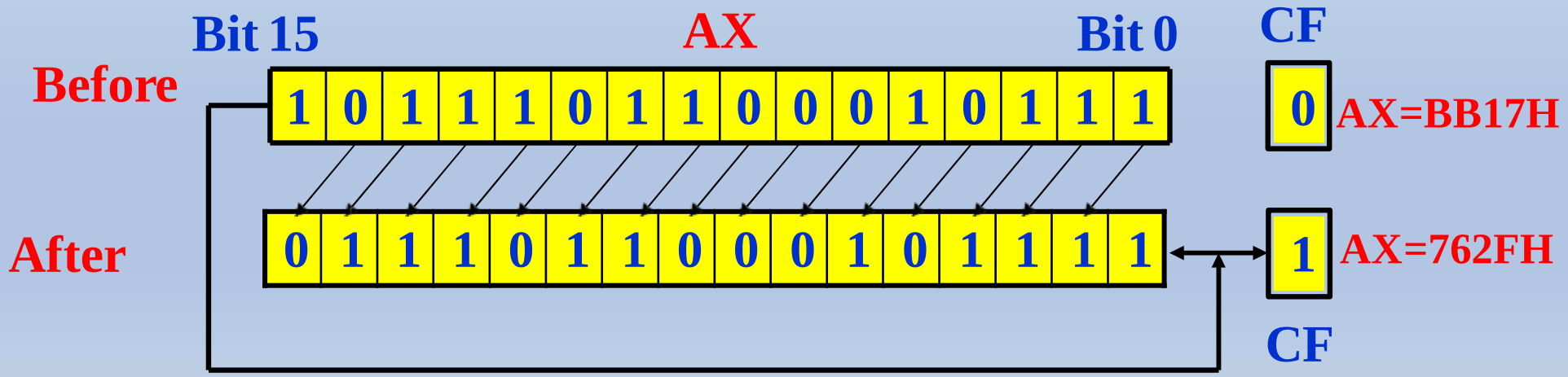
- **ROL Des, Count:**
- It rotates bits of byte or word left, by count.
- MSB is transferred to LSB and also to Carry Flag.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Logic (Bit Manipulation) Instructions
ROL: Rotate Left

- ROL Des, Count:
- Example:

```
MOV  AX, BB17H     ; AX=1011101100010111b
ROL  AX, 1         ; AX=0111011000101111b, CF=1
```





Logic (Bit Manipulation) Instructions
ROL: Rotate Left

- **ROL Des, Count:**
- **Example: In the following instructions, AL is rotated once to the left. MSB is transferred to LSB and also to Carry Flag. :**

```
MOV  AL, 40H     ; AL=01000000b, (64d)
ROL  AL, 1       ; AL=10000000b, (128d), CF=0
```

Before: AL=01000000b (64 decimal, 40H)
After: AL=10000000b (128 decimal, 80H) , CF=0

Logic (Bit Manipulation) Instructions
ROL: Rotate Left

- **ROL Des, Count:**
- **Bit rotation does not lose bits. A bit rotated off one end of a number appears again at the other end.**
- **Example: In the following instructions, how the high bit is copied into both the Carry flag and bit position 0:**

```
MOV  AL, 40H     ; AL=01000000b, (64d)
ROL  AL, 1       ; AL=10000000b, (128d), CF=0
ROL  AL, 1       ; AL=00000001b, (1d),   CF=1
ROL  AL, 1       ; AL=00000010b, (2d),   CF=0
```

Logic (Bit Manipulation) Instructions
ROL: Rotate Left

- **ROL Des, Count:**
- **Multiple Rotations: When using a rotation count greater than 1, the Carry flag contains the last bit rotated out of the MSB position:**
- **Example: In the following instructions, AL is rotated 3 times to the left.**

```
MOV  AL, 20H     ; AL=00100000b (32d)
ROL  AL, 3       ; AL=00000001b (1d), CF=1
```

Before: AL=00100000b (32 decimal, 20H)
After: AL=00000001b (1 decimal, 1H) , CF=1

Logic (Bit Manipulation) Instructions
ROL: Rotate Left

- **ROL Des, Count:**
- **Exchanging Groups of Bits:** You can use ROL to exchange the upper (bits 4–7) and lower (bits 0–3) halves of a byte.
- **Example:** In the following instructions, AL is rotated 4 times to the left. For example, 26H rotated four its in either direction becomes 62H:

```
MOV  AL, 26H     ; AL=00100110b
ROL  AL, 4       ; AL=01100010b, CF=0
```

Before: AL=00100110b (26H)
After: AL=01100010b (62H) , CF=0

Logic (Bit Manipulation) Instructions
ROL: Rotate Left

- **ROL Des, Count:**
- **Multiple Rotations: When rotating a multibyte integer by 4 bits, the effect is to rotate each hexadecimal digit one position to the right or left.**
- **Example: In the following instructions, we repeatedly rotate 6A4BH left 4 bits, eventually ending up with the original value:**

```
MOV  AX, 6A4BH    ; AX=0110101001001011b
ROL  AX, 4        ; AX=A4B6H, CF=0   (6H=0110b)
ROL  AX, 4        ; AX=4B6AH, CF=0   (AH=1010b)
ROL  AX, 4        ; AX=B6A4H, CF=0   (4H=0100b)
ROL  AX, 4        ; AX=6A4BH, CF=1   (BH=1011b)
```

Logic (Bit Manipulation) Instructions
ROL: Rotate Left

- **ROL Des, Count:**
- **The following lists the types of operands permitted by this instruction:**

```
ROL  reg, imm8
ROL  mem, imm8
ROL  reg, CL
ROL  mem, CL
```

- **The first operand in ROL is the destination and the second is the shift count.**

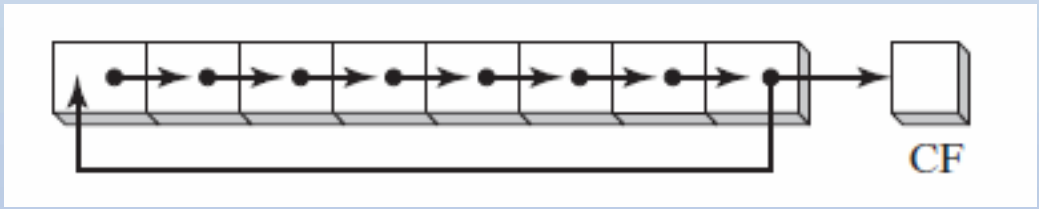
Logic (Bit Manipulation) Instructions
ROR: Rotate Right

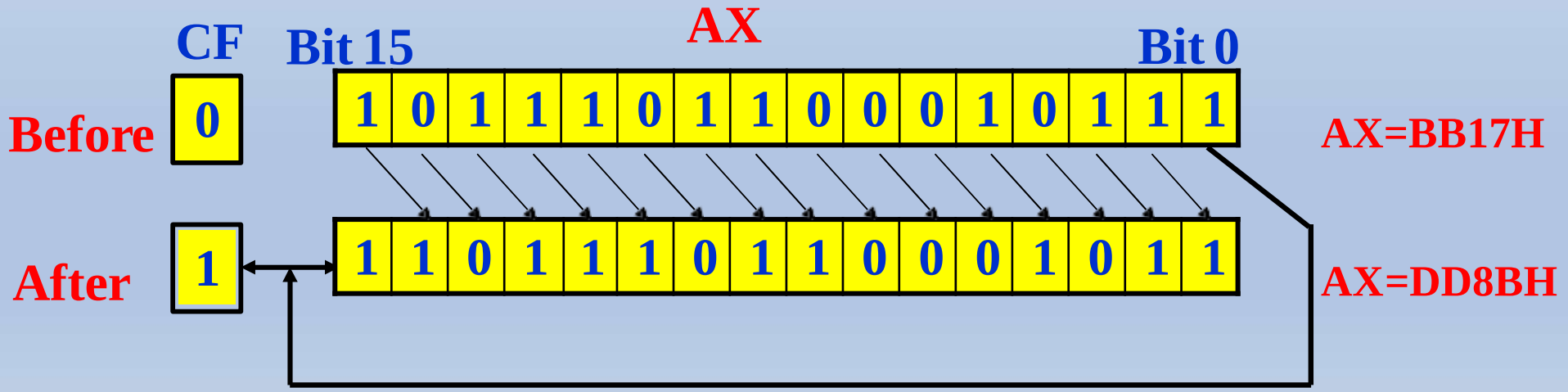
- **ROR Des, Count:**
- It rotates bits of byte or word right, by count.
- LSB is transferred to MSB and also to Carry Flag.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Logic (Bit Manipulation) Instructions
ROR: Rotate Right

- ROR Des, Count:
- Example:

```
MOV  AX, BB17H     ; AX=1011101100010111b
ROR  AX, 1         ; AX=1101110110001011b, CF=1
```





Logic (Bit Manipulation) Instructions
ROR: Rotate Right

- **ROR Des, Count:**
- **Bit rotation does not lose bits. A bit rotated off one end of a number appears again at the other end.**
- **Example: In the following instructions, how the lowest bit is copied into both the Carry flag and the highest bit position of the result:**

```
MOV  AL, 01H    ; AL=00000001b, (1d)
ROR  AL, 1      ; AL=10000000b, (128d), CF=1
ROR  AL, 1      ; AL=01000000b, (64d),  CF=0
```

Logic (Bit Manipulation) Instructions
ROR: Rotate Right

- **ROR Des, Count:**
- **Multiple Rotations: When using a rotation count greater than 1, the Carry flag contains the last bit rotated out of the LSB position:**
- **Example: In the following instructions, AL is rotated 3 times to the right.**

```
MOV  AL, 4H     ; AL=00000100b (4d)
ROR  AL, 3      ; AL=10000000b (128d), CF=1
```

Before: AL=00000100b (4 decimal, 4H)
After: AL=10000000b (128 decimal, 80H) , CF=1

Logic (Bit Manipulation) Instructions
ROR: Rotate Right

- **ROR Des, Count:**
- **The following lists the types of operands permitted by this instruction:**

```
ROR  reg, imm8
ROR  mem, imm8
ROR  reg, CL
ROR  mem, CL
```

- **The first operand in ROR is the destination and the second is the shift count.**

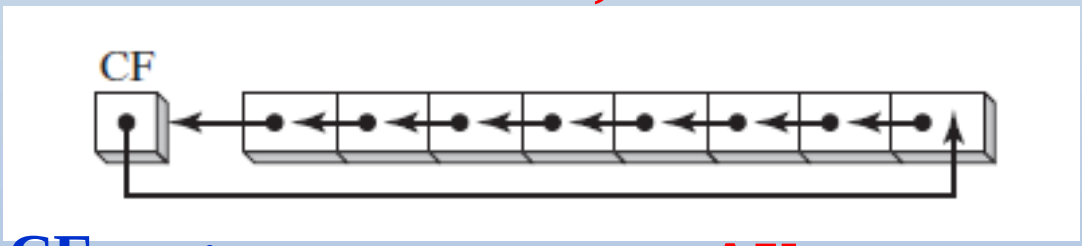
Logic (Bit Manipulation) Instructions
RCL: Rotate Left Through Carry

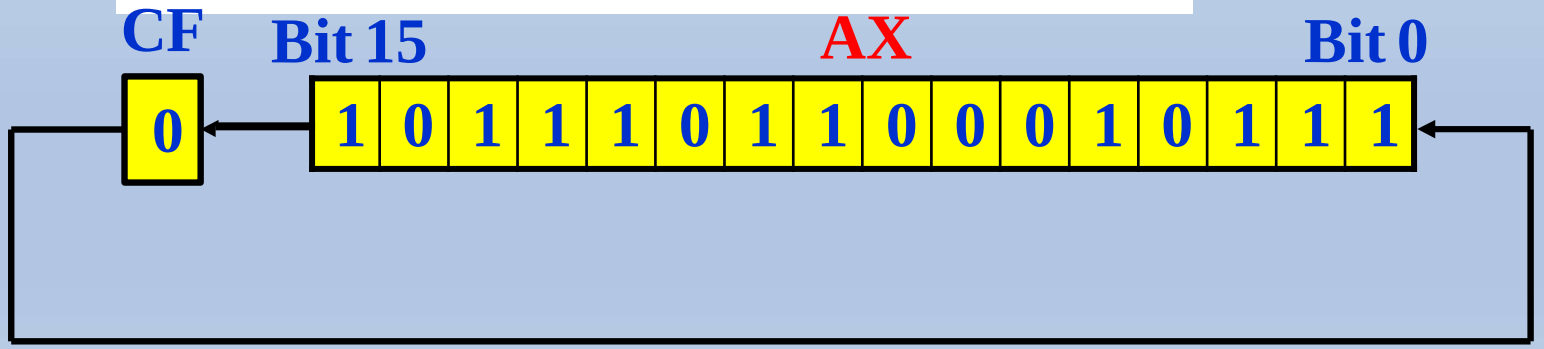
- **RCL Des, Count:**
- The RCL instruction shifts each bit to the left, copies the Carry flag to the LSB, and copies the MSB into the Carry flag.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Logic (Bit Manipulation) Instructions
RCL: Rotate Left Through Carry

- **RCL Des, Count:**
- **Example:**

```
CLC                 ; CF=0
MOV  AX, BB17H      ; AX=1011101100010111b, CF=0
RCL  AX, 1          ; AX=0111011000101110b, CF=1
```





Before: AX=BB17H **After: AX=762EH**

Logic (Bit Manipulation) Instructions
RCL: Rotate Left Through Carry

- **RCL Des, Count:**
- **The following lists the types of operands permitted by this instruction:**

```
RCL  reg, imm8
RCL  mem, imm8
RCL  reg, CL
RCL  mem, CL
```

- **The first operand in RCL is the destination and the second is the shift count.**

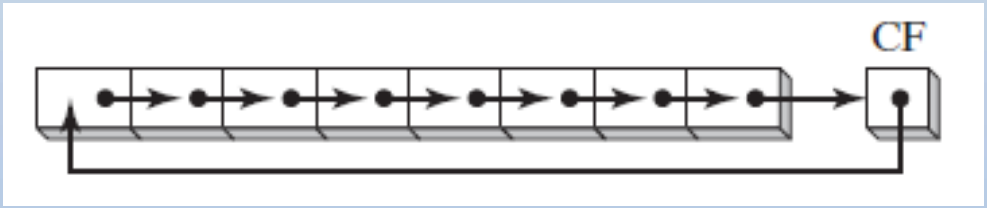
Logic (Bit Manipulation) Instructions
RCR: Rotate Right Through Carry

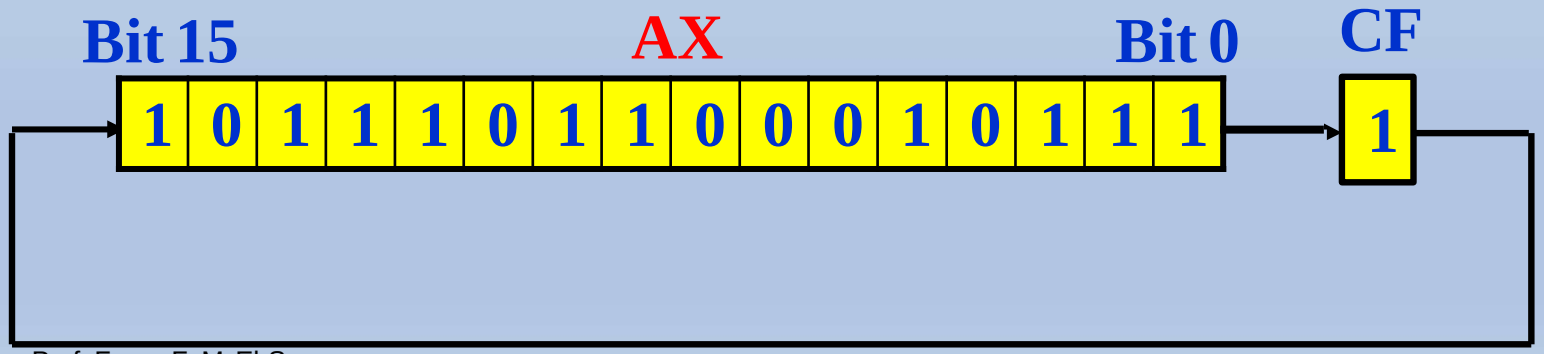
- **RCR Des, Count:**
- **The RCR instruction shifts each bit to the right, copies the Carry flag to the MSB, and copies the LSB into the Carry flag.**
- **If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.**
- **However, if the number of bits to be shifted is more than 1, then the count is put in CL register.**

Logic (Bit Manipulation) Instructions
RCR: Rotate Right Through Carry

- **RCR Des, Count:**
- **Example:**

```
STC                 ; CF=1
MOV  AX, BB17H      ; AX=1011101100010111b, CF=1
RCR  AX, 1          ; AX=1101110110001011b, CF=1
```





Before: AX=BB17H

After: AX=DD8BH

Logic (Bit Manipulation) Instructions
RCR: Rotate Right Through Carry

- **RCR Des, Count:**
- **The following lists the types of operands permitted by this instruction:**

```
RCR  reg, imm8
RCR  mem, imm8
RCR  reg, CL
RCR  mem, CL
```

- **The first operand in RCR is the destination and the second is the shift count.**

Flag-Control Instructions CLC, STC, CMC, CLD, STD, CLI, STI

- The microprocessor has a set of flags that either monitors the state of executing instructions or controls options available in its operation.
- CLC: Clear Carry Flag - It clears the carry flag to 0.
- STC: Set Carry Flag - It sets the carry flag to 1.
- CMC: Complement Carry Flag - It complements the carry flag.
- CLD: Clear Direction Flag - It clears the direction flag to 0. If it is reset or cleared, the string bytes are accessed from lower memory address to higher memory address.

Flag-Control Instructions CLC, STC, CMC, CLD, STD, CLI, STI

- **STD: Set Direction Flag - It sets the direction flag to 1. If it is set, string bytes are accessed from higher memory address to lower memory address.**
- **CLI: Clear Interrupt Flag – It clears the interrupt flag to 0.**
- **STI: Set Interrupt Flag – It sets the interrupt flag to 1.**
- **These instructions control the processor itself.**

Flag-Control Instructions
CLC, STC, CMC, CLD, STD, CLI, STI

Mnemonic	Meaning	Operation	Flags affected
STC	Set carry flag	(CF) ←1	CF
CLC	Clear carry flag	(CF) ←0	CF
CMC	Complement carry flag	(CF) ← (CF)	CF
CLD	Clear direction flag	(DF) ←0	DF
STD	Set direction flag	(DF) ←1	DF
CLI	Clear interrupt flag	(IF) ←0	IF
STI	Set interrupt flag	(IF) ←1	IF

Flag-Control Instructions

Table 2.6 *Machine Control Instructions*

WAIT	-	Wait for Test input pin to go low
HLT	-	Halt the processor
NOP	-	No operation
ESC	-	Escape to external device like NDP (numeric co-processor)
LOCK	-	Bus lock instruction prefix.

Control Transfer or Branching Instructions

- Introduction
- 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.

The Jump Group Instructions

- Allows programmer to skip program sections and branch to any part of memory for the next instruction.
- A conditional jump instruction allows decisions based upon numerical tests.
  - results are held in the flag bits, then tested by conditional jump instructions
- LOOP and conditional LOOP are also forms of the jump instruction.

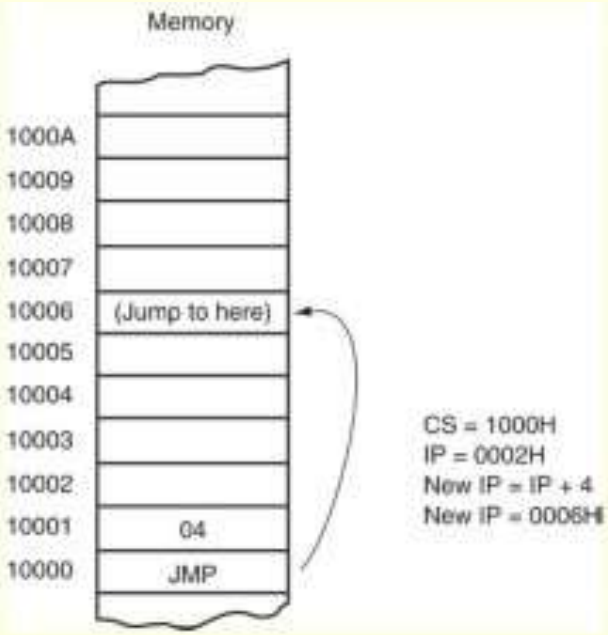
The Jump Group Instructions Unconditional Jump, JMP Des

- **JMP Des:**
- **This instruction is used for unconditional jump from one place to another.**
- **Two types: short jump and near jump .**
- **Short jump is a 2-byte instruction that allows jumps or branches to memory locations within +127 and – 128 bytes from the address following the jump**

- **Near jump is a 3-byte instruction that allows a branch or jump within ±32K bytes from the instruction in the current code segment of the jump instruction.**

The Jump Group Instructions Unconditional Jump, JMP Des

- **Short Jump:**
- **When the microprocessor executes a short jump, the displacement is sign- extended and added to the instruction pointer (IP) to generate the jump address within the current code segment.**
- **The instruction branches to this new address for the next instruction in the program**



A short jump to four memory locations beyond the address of the next instruction.

The Jump Group Instructions Conditional Jump, Jxx Des

- **Jxx Des (Conditional jump):**
- **All the conditional jumps follow some conditional statements or any instruction that affects the flag.**
- **In the conditional jump, control is transferred to a new location if a certain condition is met.**
- **Always short jumps in 8086.**
  - **limits range to within +127 and –128 bytes from the location following the conditional jump**
- **Allows a conditional jump to any location within the current code segment.**

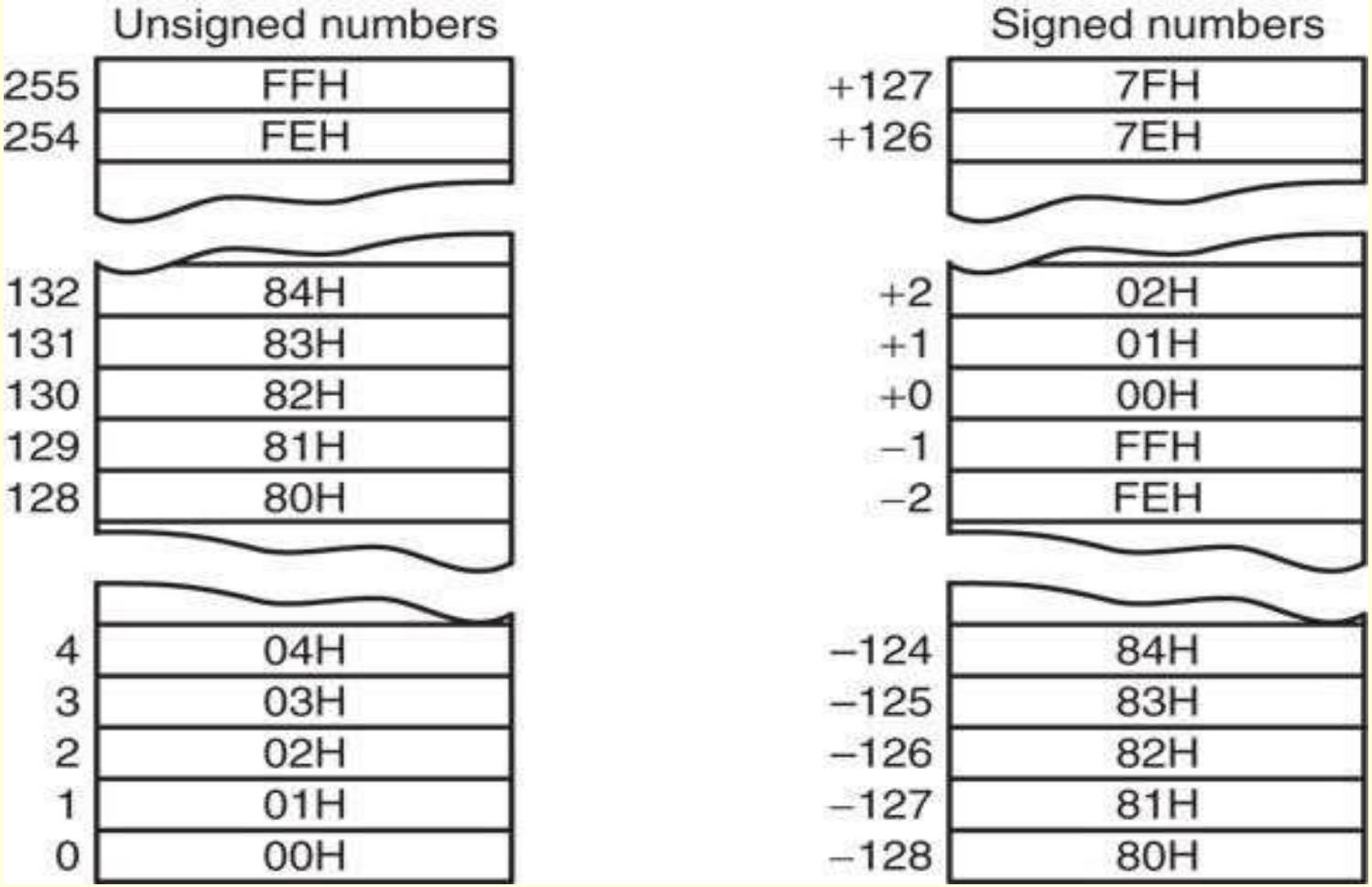
The Jump Group Instructions
Conditional Jump, Jxx Des

- **Jxx Des (Conditional jump):**
- **Conditional jump instructions test flag bits:**
  - **sign (S), zero (Z), carry (C)**
  - **parity (P), overflow (O)**
- **If the condition under test is true, a branch to the label associated with the jump instruction occurs.**
  - **if false, next sequential step in program executes**
  - **for example, a JC will jump if the carry bit is set**
- **Most conditional jump instructions are straightforward as they often test one flag bit**

The Jump Group Instructions
Conditional Jump, Jxx Des

- **Jxx Des (Conditional jump):**
- **Because both signed and unsigned numbers are used in programming.**
- **Because the order of these numbers is different, there are two sets of conditional jump instructions for magnitude comparisons.**
- **16-bit numbers follow the same order as 8-bit numbers, except that they are larger.**
  - **The following Figure shows the order of both signed and unsigned 8-bit numbers.**

The Jump Group Instructions Conditional Jump, Jxx Des



Signed and unsigned numbers follow different orders

The Jump Group Instructions
Conditional Jump, Jxx Des

- **Jxx Des (Conditional jump):**
- **When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions.**

- **terms greater than and less than refer to signed numbers**

- **When unsigned numbers are compared, use the JA, JB, JAB, JBE, JE, and JNE instructions.**

- **terms above and below refer to unsigned numbers**

The Jump Group Instructions
Conditional Jump, Jxx Des

- **Jxx Des (Conditional jump):**

- **Remaining conditional jumps test individual flag bits, such as overflow and parity.**

  - **notice that JE has an alternative opcode JZ**

- **All instructions have alternates, but many aren't used in programming because they don't usually fit the condition under test.**

The Jump Group Instructions Conditional Jump, Jxx Des

- Conditional jumps, have mnemonics such as JNZ (jump not zero) and JC (jump if carry).
- In the conditional jump, control is transferred to a new location if a certain condition is met.
- The flag register is the one that indicates the current condition.
- For example, with "JNZ label",
  - the processor looks at the zero flag to see if it is raised
  - If not, the CPU starts to fetch and execute instructions from the address of the label.
  - If ZF = 1, it will not jump but will execute the next instruction below the JNZ.

The Jump Group Instructions
Conditional Jump Instructions

- **JAE/JNB/JNC (jump if above or equal / jump if not below/ jump if no carry)**
  - **If, after a compare or some other instructions which affect flags, the carry flag is 0, this instruction will cause execution to jump to a label given in the instruction. If CF is 1, the instruction will have no effect on program execution.**
- **Example:**

```
CMP AX,4371H        ; Compare (AX– 4371H)
JAE NEXT            ; Jump to label NEXT if AX above or
                      equal 4371H

CMP AX,4371H        ; Compare (AX – 4371H)
JNB NEXT            ; Jump to label NEXT if AX not below 4371H
ADD AL, BL          ; Add two bytes
JNC NEXT            ; If the result with in acceptable range, continue
```

The Jump Group Instructions
Conditional Jump Instructions

Mnemonic	Condition Tested	Jump if:
JA / JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF=0	above or equal/not below
JB/JNAE	CF=1	below/not above nor equal
JBE/JNA	(CForZF)=1	below or equal/not above
JC	CF=1	Carry
JE/JZ	ZF=1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF=0	not carry

The Jump Group Instructions
Conditional Jump Instructions

Mnemonic	Condition Tested	Jump if:
JNE/JNZ	ZF=0	not equal/not zero
JNO	OF=0	not overflow
JNP/JPO	PF=0	not parity/parity odd
JNS	SF=0	not sign
JO	OF=1	Overflow
JP/TPE	PF=1	parity/parity equal
JS	SF=1	sign

Notes:

1. Above and Below are used with unsigned values
2. Greater and Less are used with signed values

The Jump Group Instructions
Conditional Jump Instructions

Conditional Jumps Based on Specific Flag Values

Mnemonic	Description	Flags / Registers
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

The Jump Group Instructions
Conditional Jump Instructions

Conditional Jumps Based on Unsigned Comparisons

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* ≥ *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* ≤ *rightOp*)
JNA	Jump if not above (same as JBE)

The Jump Group Instructions
Conditional Jump Instructions

Conditional Jumps Based on Signed Comparisons

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* ≥ *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* ≤ *rightOp*)
JNG	Jump if not greater (same as JLE)

The Loop Instructions
LOOP Instructions

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label	(CX)←(CX) – 1 Jump is initiated to location defined by short-label if (CX) ≠0; otherwise, execute next sequential instruction.
LOOPE/LOOPZ	Loop while equal/loop while zero	LOOPE/LOOPZ Short-label	(CX)←(CX) – 1 Jump to the location by short-label if (CX)≠0 and (ZF) = 1; otherwise execute next sequential instruction.
LOOPNE/ LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ Short-label	(CX)←(CX) – 1 Jump to the location defined by short label if (CX)≠0 and (ZF) = 0; otherwise execute next sequential instruction

The Loop Instructions
Unconditional LOOP Instruction

- **Loop Des:**
- **LOOP (jump to specified label if CX≠ 0, after auto decrement)**
- **This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1.**
- **If CX is not 0, execution will jump to a destination specified by a label in the instruction.**
- **If CX = 0 after the auto decrement, execution will simply go on to the next instruction after LOOP.**
- **The destination address for the jump must be in the range of – 128 bytes to +127 bytes from the address of the instruction after the LOOP instruction. This instruction does not affect any flag.**

The Loop Instructions
Conditional LOOP Instruction

- LOOPE/LOOPZ (loop while CX ≠ 0 AND ZF=1)
  - This instruction is used to repeat a group of instructions some number of times, or until the zero flag becomes 0. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1.

  - If CX ≠ 0 and ZF = 1, execution will jump to a destination specified by a label in the instruction. If CX = 0, execution simply go on the next instruction after LOOPE/LOOPZ. In other words, the two ways to exit the loop are CX = 0 or ZF = 0. The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOPE/LOOPZ instruction. This instruction does not affect any flag.

The Loop Instructions
Conditional LOOP Instruction

- LOOPE/LOOPZ (loop while CX ≠ 0 AND ZF=1)
- Example:

```
        MOV BX, OFFSET ARRAY ; Point BX to address of ARRAY
                               before start of array
        DEC BX               ; Decrement BX
        MOV CX, 100          ; Load CX with number of elements in
                               array
NEXT:   INC BX               ; Point to next element in array
        CMP [BX], 0FFH       ; Compare array element with FFH
        LOOPE   NEXT         ; Repeat until all elements adjusted
```

The Procedures CALL/RET Instructions

- A procedure is a group of instructions that usually performs one task.
  - subroutine, method, or function is an important part of any system's architecture
- A procedure is a reusable section of the software stored in memory once, used as often as necessary.
  - saves memory space and makes it easier to develop software
- Disadvantage of procedure is time it takes the computer to link to, and return from it.
  - CALL links to the procedure; the RET (return)y

The Procedures
CALL/RET Instructions

- CALL pushes the address of the instruction following the CALL (return address) on the stack.
  - the stack stores the return address when a procedure is called during a program
- RET instruction removes an address from the stack so the program returns to the instruction following the CALL.

The Procedures
CALL/RET Instructions

- **CALL Des:**
- **This instruction is used to call a subroutine or function or procedure.**
- **The address of next instruction after CALL is saved onto stack.**
- **RET:**
- **It returns the control from procedure to calling program**
  - **Every CALL instruction should have a RET.**
  - **The address of the instruction following the CALL (return address) on the stack is retrieved.**

The Procedures
CALL/RET Instructions

- **CALL Des:**
- **Transfers the flow of the program to the procedure.**
- **CALL instruction differs from the jump instruction because a CALL saves a return address on the stack.**
- **The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.**
- **CALLs to procedures are used to perform tasks that need to be performed frequently**

The Procedures
CALL/RET Instructions

- **CALL Des:**
- **This makes a program more structured**
- **The target address could be:**
  - **in the current segment, in which case it will be a NEAR CALL**
  - **or outside the current CS segment, which is a FAR CALL**
- **in the NEAR call only the IP is saved on the stack,**
- **in a FAR call both CS and IP are saved.**

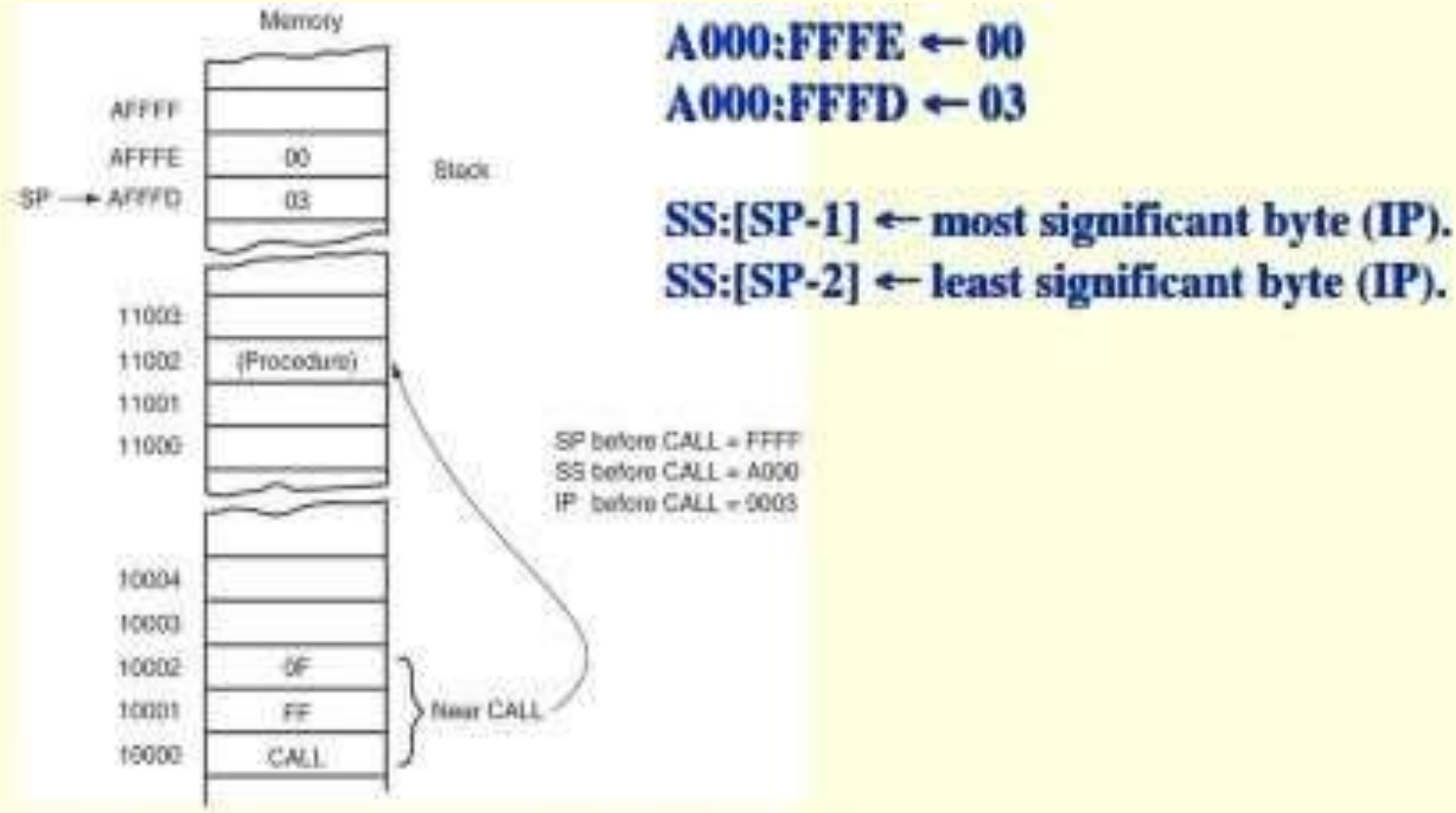
The Procedures
CALL/RET Instructions

- Near CALL
- When the near CALL executes, it first pushes the offset address of the next instruction onto the stack.

offset address of the next instruction appears in the instruction pointer (IP)

The Procedures
CALL/RET Instructions

- **Near CALL**



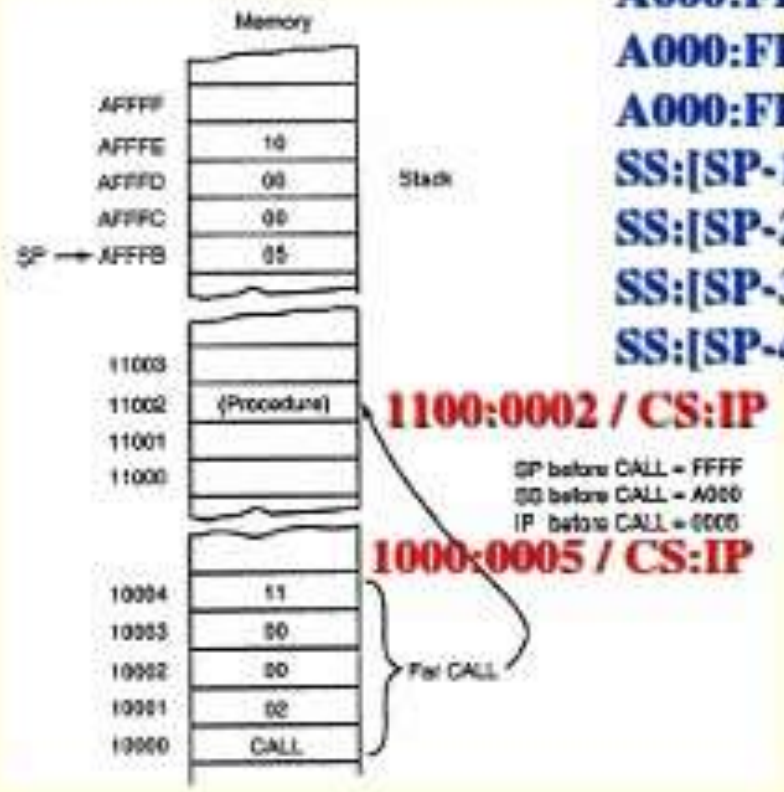
The effect of a near CALL on the stack and the instruction pointer IP.

The Procedures
CALL/RET Instructions

- **Far CALL**
- **Far CALL places the contents of both IP and CS on the stack before jumping to the address.**
- **far CALL allows to call a procedure located anywhere in the memory and return from that procedure.**

The Procedures
CALL/RET Instructions

Far CALL



A000:FFFE ← 10
A000:FFFD ← 00
A000:FFFC ← 00
A000:FFFB ← 05
SS:[SP-1] ← most significant byte (CS).
SS:[SP-2] ← least significant byte (CS).
SS:[SP-3] ← most significant byte (IP).
SS:[SP-4] ← least significant byte (IP).

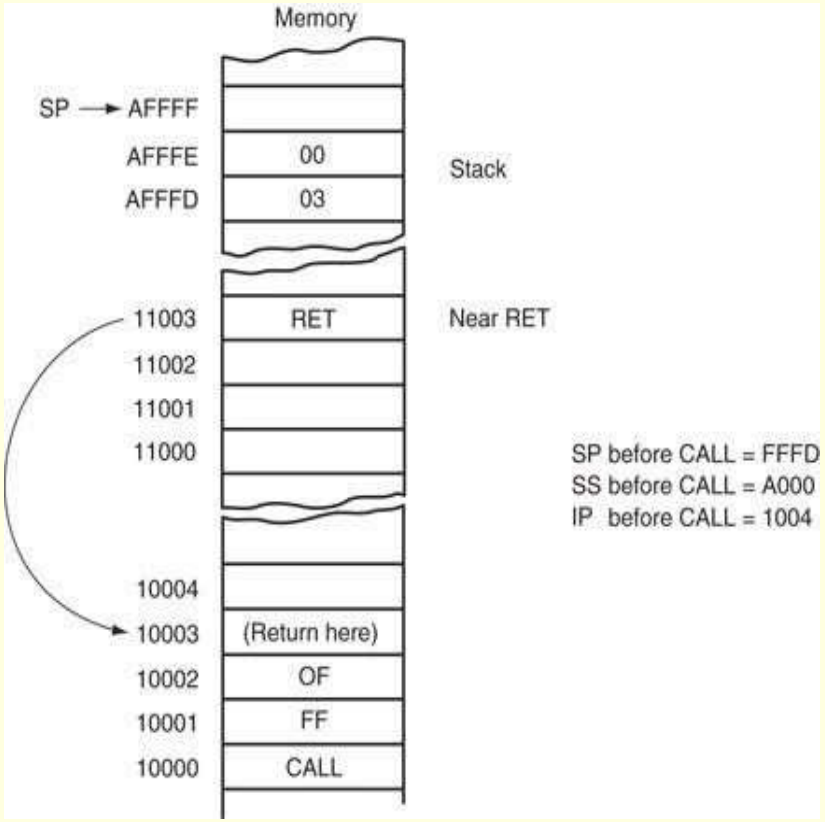
The effect of far CALL on the stack and the instruction pointer IP.

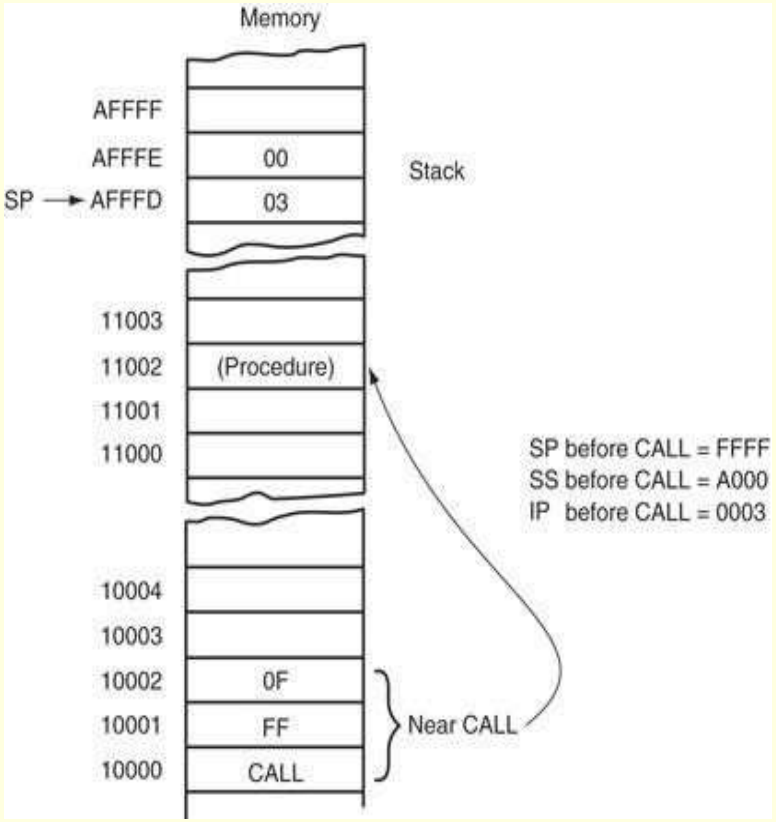
The Procedures
CALL/RET Instructions

- **RET:**
- **The last instruction of the called subroutine must be a RET instruction.**
- **which directs the CPU to POP the top 2 bytes of the stack into the IP.**
- **Note: the number of PUSH and POP instructions (which alter the SP) must match.**
- **In other words, for every PUSH there must be a POP.**
- **Near RET: Removes a 16-bit number (near return) from the stack placing it in IP.**
- **Far RET: Removes a 32-bit number (far return) from the stack and places it in IP & CS.**

The Procedures
CALL/RET Instructions

- **RET:**
- **The following Figure shows how the CALL instruction links to a procedure and how RET returns.**





INT N: Interrupt Type N In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from 00H to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (N′4) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4 (offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine. For the execution of this instruction, the IF must be enabled.

Example 2.45

Thus the instruction INT 20H will find out the address of the interrupt service routine as follows:

```
INT      20H
Type* 4 = 20 * 4 = 80H
```

Pointer to IP and CS of the ISR is 0000 : 0080 H

Figure 2.14 shows the arrangement of CS and IP addresses of the ISR in the interrupt vector table.

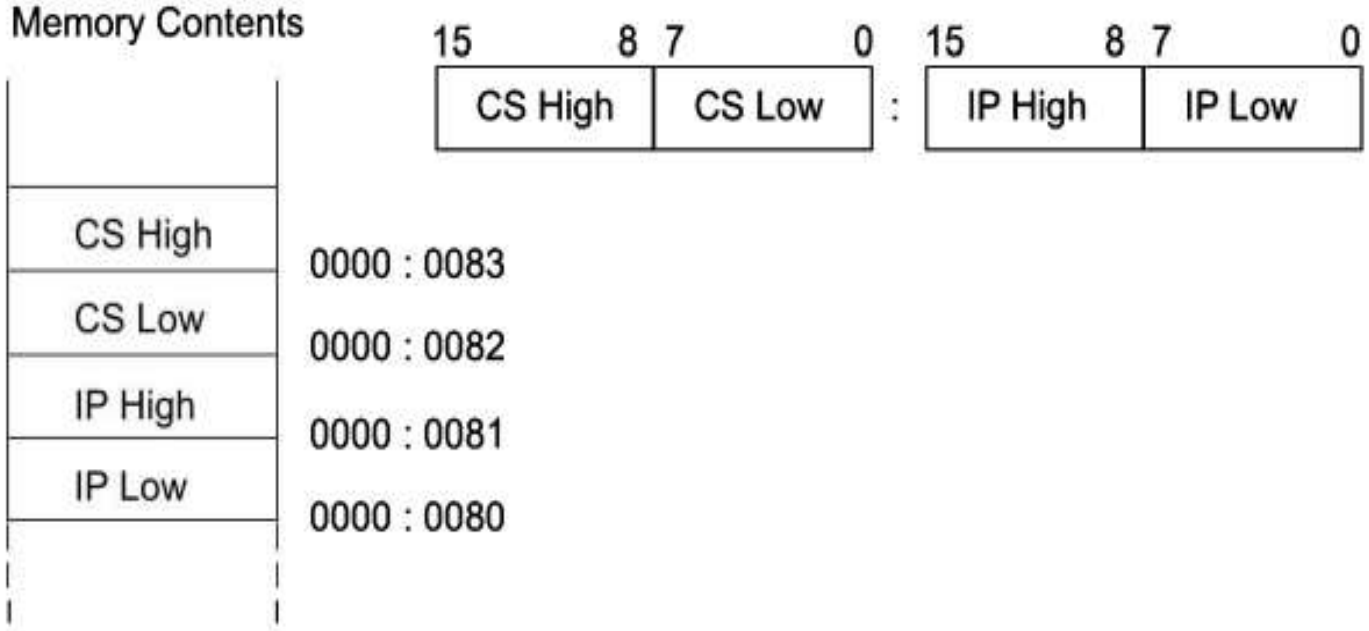


Fig. 2.14 *Contents of IVT*

INTO: Interrupt on Overflow This command is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0010 as explained in INT type instruction. This is equivalent to a Type 4 interrupt instruction.

IRET: Return from ISR When an interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored on to the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the values of IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

String Instructions MOVS/CMPS/SCAS/LODS/STOS

- String in assembly language is just a sequentially stored bytes or words.
- By using these string instructions, the size of the program is considerably reduced.
- There are five basic string instructions in the instruction set of the 8086.
- Move byte or word string (MOVSB/MOVSW).
- Compare byte or word string (CMPSB/CMPSW).
- Scan byte or word string (SCASB/SCASW).
- Load byte or word string (LODSB/LODSW).
- Store byte or word string (STOSB/STOSW).

String Instructions
MOVS/CMPS/SCAS/LODS/STOS

- Before the string instructions are presented, the operation of the D flag-bit (direction), DI, and SI must be understood as they apply to the string instructions.
- The direction flag (D, located in the flag register) selects the auto-increment or the auto-decrement operation for the DI and SI registers during string operations.
  - used only with the string instructions
- The CLD instruction clears the D flag and the STD instruction sets it.
  - CLD instruction selects the auto-increment mode STD selects the auto-decrement mode.

String Instructions
Move String–MOVSB/MOV

- MOVS
  - MOVS Des String Name, Src String Name
- MOVSB
  - MOVSB Des String Name, Src String Name
- MOVSW
  - MOVSW Des String Name, Src String Name
- Transfers a byte, word, or doubleword from data segment addressed by SI to extra segment location addressed by DI.
  - pointers are incremented or decremented, as dictated by the direction flag.

String Instructions
Move String–MOVSB/MOVSW

- MOVSB transfers byte from data segment to extra segment.
- MOVSW transfers word from data segment to extra segment.
- Only the source operand (SI), located in the data segment may be overridden so another segment may be used.
- The destination operand (DI) must always be located in the extra segment.

String Instructions
Move String–MOVSB/MOVSW

- This instruction copies a byte or a word from location in the data segment to a location in the extra segment.
- The offset of the source in the data segment must be in the SI register.
- The offset of the destination in the extra segment must be in the DI register.
- For multiple-byte or multiple-word moves, the number of elements to be moved is put in the CX register so that it can function as counter.
- After the byte or a word is moved, SI and DI are automatically adjusted to point to the next source element and the next destination element.

String Instructions
Move String–MOVSB/MOVSW

- If DF is 0, then SI and DI will be incremented by 1 after a byte move and by 2 after a word move.
- If DF is 1, then SI and DI will be decremented by 1 after a byte move and by 2 after a word move.
- MOVS does not affect any flag.
- When using the MOVS instruction, you must in some way tell the assembler whether you want to move a string as bytes or as word.
- There are two ways to do this.
- The first way is to indicate the name of the source and destination strings in the instruction, as, for example.

  MOVS DEST, SRC.

String Instructions
Move String–MOVSB/MOVSW

- The assembler will code the instruction for a byte / word move if they were declared with a DB / DW.
- The second way is to add a "B" or a "W" to the MOVS mnemonic.
- MOVSB says move a string as bytes.
- MOVSW says move a string as words.

String Instructions
Move String–MOVSB/MOVSW

❒**Example:**

```
        MOV SI, OFFSET SRC          ; Load offset of start of source
                                    string in DS into SI
        MOV DI, OFFSET DES          ; Load offset of start of dest.
                                    String in ES into DI
        CLD                         ; Clear DF to auto increment SI
                                     and DI after move
        MOV CX, 04H                 ; Load length of string into CX as counter
NEXT:   MOVSB
            LOOP   NEXT             ; Move string byte until CX = 0
```

String Instructions
Loads String–LODSB/LODSW

- LODS / LODSB / LODSW (LOAD STRING BYTE INTO AL OR STRING WORD INTO AX)
- This instruction copies a byte from a string location pointed to by SI to AL, or a word from a string location pointed to by SI to AX.
- If DF is 0, SI will be automatically incremented (by 1 for a byte string, and 2 for a word string) to point to the next element of the string.
- If DF is 1, SI will be automatically decremented (by 1 for a byte string, and 2 for a word string) to point to the previous element of the string.
- LODS does not affect any flag.

String Instructions
Loads String–LODSB/LODSW

- Loads AL or AX with data at segment offset address indexed by the SI register.
- a 1 is added to or subtracted from SI for a byte-sized LODS
- a 2 is added or subtracted for a word-sized LODS.

String Instructions
Loads String–LODSB/LODSW

```
    CLD
; Clear direction flag so that SI is auto-incremented
        MOV SI, OFFSET SOURCE
; Point SI to start of string
LODS  SOURCE
; Copy a byte or a word from string to AL or AX
```

String Instructions
Stores String–STOSB/STOSW

- STOS / STOSB / STOSW (STORE STRING BYTE OR STRING WORD)
- This instruction copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI.
- In effect, it replaces a string element with a byte from AL or a word from AX. After the copy, DI is automatically incremented or decremented to point to next or previous element of the string.
- If DF is cleared, then DI will automatically incremented by 1 for a byte string and by 2 for a word string.
- If DI is set, DI will be automatically decremented by 1 for a byte string and by 2 for a word string.
- STOS does not affect any flag.

String Instructions
Stores String–STOSB/STOSW

MOV DI,OFFSET TARGET

STOSB

String Instructions
Compare String–CMPSB/CMPSW

- CMPS / CMPSB / CMPSW (COMPARE STRING BYTE OR STRING WORD)
- This instruction can be used to compare a byte / word in one string with a byte / word in another string. SI is used to hold the offset of the byte or word in the source string, and DI is used to hold the offset of the byte or word in the destination string.
- The AF, CF, OF, PF, SF, and ZF flags are affected by the comparison, but the two operands are not affected. After the comparison, SI and DI will automatically be incremented or decremented to point to the next or previous element in the two strings.

String Instructions
Compare String–CMPSB/CMPSW

- CMPS / CMPSB / CMPSW (COMPARE STRING BYTE OR STRING WORD)
- If DF is set, then SI and DI will automatically be decremented by 1 for a byte string and by 2 for a word string.
- If DF is reset, then SI and DI will automatically be incremented by 1 for byte strings and by 2 for word strings.
- The string pointed to by SI must be in the data segment.
- The string pointed to by DI must be in the extra segment.

String Instructions
Compare String–CMPSB/CMPSW

- The CMPS instruction can be used with a REPE or REPNE prefix to compare all the elements of a string.

```
MOV SI, OFFSET FIRST
        ; Point SI to source string
MOV DI, OFFSET SECOND
        ; Point DI to destination string
CLD DF
        ; cleared, SI and DI will auto-increment after compare
MOV CX, 100
        ; Put number of string elements in CX
REPE CMPSB
        ; Repeat the comparison of string bytes until end of
          string or until compared bytes are not equal
```

String Instructions
Compare String–CMPSB/CMPSW

- CX functions as a counter, which the REPE prefix will cause CX to be decremented after each compare.
- The B attached to CMPS tells the assembler that the strings are of type byte. If you want to tell the assembler that strings are of type word, write the instruction as CMPSW.
- The REPE CMPSW instruction will cause the pointers in SI and DI to be incremented by 2 after each compare, if the direction flag is set.

String Instructions
Scan String–SCASB/SCASW

- SCAS / SCASB / SCASW (SCAN A STRING BYTE OR A STRING WORD)
- SCAS compares a byte in AL or a word in AX with a byte or a word in ES pointed to by DI. Therefore, the string to be scanned must be in the extra segment, and DI must contain the offset of the byte or the word to be compared.
- If DF is cleared, then DI will be incremented by 1 for byte strings and by 2 for word strings.
- If DF is set, then DI will be decremented by 1 for byte strings and by 2 for word strings. SCAS affects AF, CF, OF, PF, SF, and ZF, but it does not change either the operand in AL (AX) or the operand in the string.

String Instructions
Scan String–SCASB/SCASW

- SCAS / SCASB / SCASW (SCAN A STRING BYTE OR A STRING WORD)
- The following program segment scans a text string of 80 characters for a carriage return, 0DH, and puts the offset of string into DI:

```
MOV DI, OFFSET STRING
MOV AL, 0DH          ; Byte to be scanned for into AL
MOV CX, 80           ; CX used as element counter
CLD                  ; Clear DF, so that DI auto increments
REPNE SCASB          ; Compare byte in string with byte in AL
```

String Instructions
Repeat String--REPE/REPZ/REPNE/REPNZ

- REP / REPE / REPZ / REPNE / REPNZ (PREFIX) (REPEAT STRING INSTRUCTION UNTIL SPECIFIED CONDITIONS EXIST)
- REP is a prefix, which is written before one of the string instructions. It will cause the CX register to be decremented and the string instruction to be repeated until CX = 0.
- The instruction REP MOVSB, for example, will continue to copy string bytes until the number of bytes loaded into CX has been copied.
- REPE and REPZ are two mnemonics for the same prefix.

String Instructions
Repeat String–REPE/REPZ/REPNE/REPNZ

- REP / REPE / REPZ / REPNE / REPNZ (PREFIX) (REPEAT STRING INSTRUCTION UNTIL SPECIFIED CONDITIONS EXIST)
- They stand for repeat if equal and repeat if zero, respectively.
- They are often used with the Compare String instruction or with the Scan String instruction.
- They will cause the string instruction to be repeated as long as the compared bytes or words are equal (ZF = 1) and CX is not yet counted down to zero. In other words, there are two conditions that will stop the repetition: CX = 0 or string bytes or words not equal.

String Instructions
Repeat String–REPE/REPZ/REPNE/REPNZ

- REP / REPE / REPZ / REPNE / REPNZ (PREFIX) (REPEAT STRING INSTRUCTION UNTIL SPECIFIED CONDITIONS EXIST)
- REPNE and REPNZ are also two mnemonics for the same prefix.
- They stand for repeat if not equal and repeat if not zero, respectively.
- They are often used with the Compare String instruction or with the Scan String instruction.
- They will cause the string instruction to be repeated as long as the compared bytes or words are not equal (ZF = 0) and CX is not yet counted down to zero.

String Instructions
Repeat String–REPE/REPZ/REPNE/REPNZ

- **REP / REPE / REPZ / REPNE / REPNZ (PREFIX) (REPEAT STRING INSTRUCTION UNTIL SPECIFIED CONDITIONS EXIST)**

REPE CMPSB ; **Compare string bytes until end of string or until string bytes not equal.**

REPNE SCASW ; **Scan a string of word until a word in the string matches the word in AX or until all of the string has been scanned.**