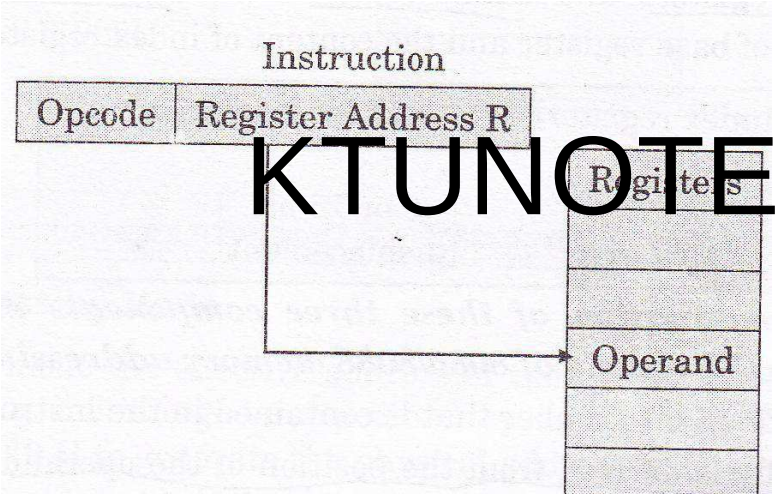
8086 Addressing Modes

1. Register Addressing Mode
2. Immediate Addressing Mode
3. Direct addressing mode
4. Register Indirect addressing mode
5. Based Indexed addressing mode
6. Register Relative addressing mode
7. Relative Based Indexed addressing mode
8. Implied addressing mode

1.Register Addressing Mode

7 Data is in register and Instruction Specifies the perticular register

 E.g MOV AX,BX

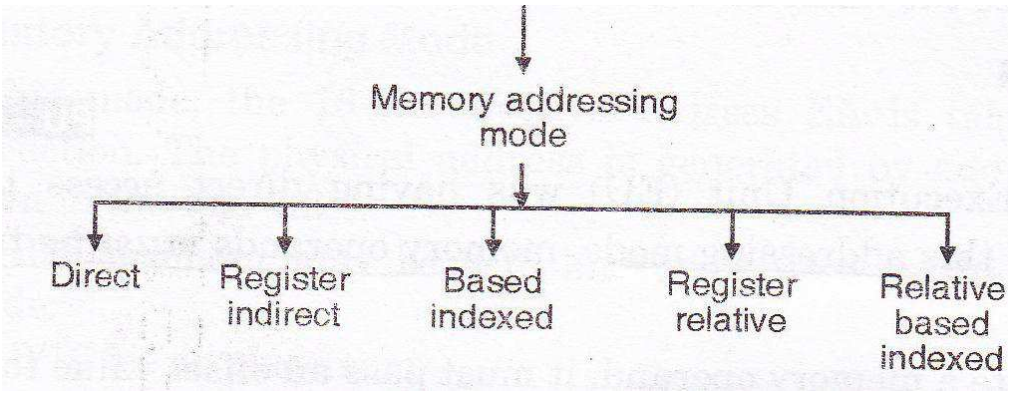


2.Immediate Addressing Mode

- Immediate operand is Constant data contained in an Instruction. The source operand is a part of instruction instead of register/memory

E.g MOV CL,02H

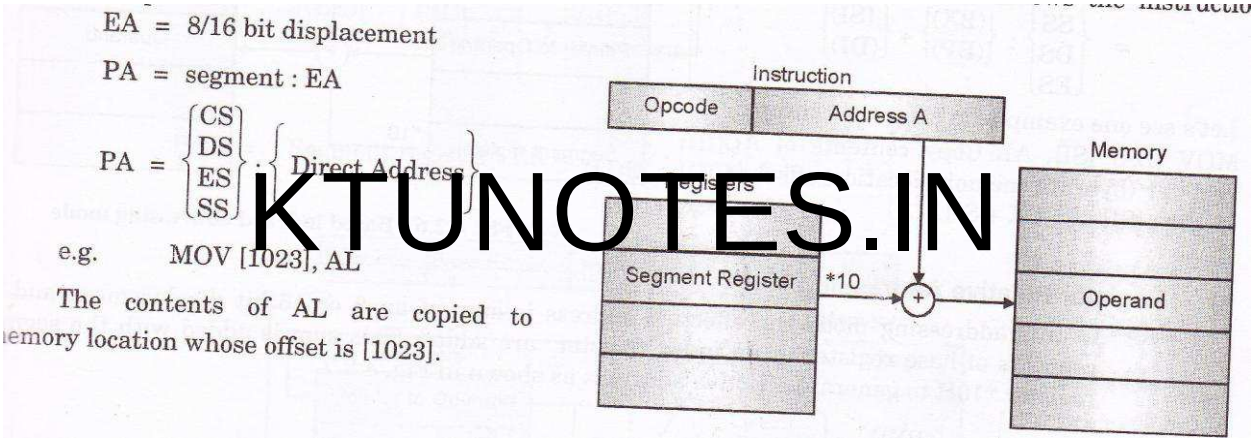
Memory Addressing Mode



3. **Direct addressing mode**

- EA is taken from the displacement field of instruction.
- PA=This addr. Is added with Seg.Regi*10 H

MOV [1023],AL



4. Register indirect addressing mode

- EA of may be taken directly from one of the base register or index register.
- PA=This addr. Is added with Seg.Regi*10 H

$$EA = \begin{Bmatrix} (BX) \\ (BP) \\ (SI) \\ (DI) \end{Bmatrix} \quad PA = Segment : EA$$

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \\ SI \\ DI \end{Bmatrix}$$

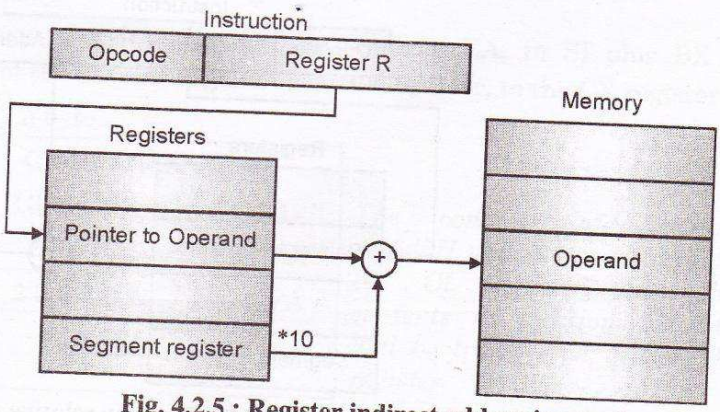


Fig. 4.2.5 : Register indirect addressing modes

Eg: MOV[SI],AL

5. Based indexed addressing mode

- EA is sum of Base register and Index register.
- Both of which are specified by the instruction
- PA=This addr. Is added with Seg.Regi*10 H

Eg: MOV [BX+SI], AL

$$\therefore EA = \{\text{Base register}\} + \{\text{Index register}\}$$

$$= \begin{Bmatrix} (BX) \\ (BP) \end{Bmatrix} + \begin{Bmatrix} (SI) \\ (DI) \end{Bmatrix}$$

$$PA = \text{Segment register} : EA$$

$$= \begin{Bmatrix} CS \\ SS \\ DS \\ ES \end{Bmatrix} : \begin{Bmatrix} (BX) \\ (BP) \end{Bmatrix} + \begin{Bmatrix} (SI) \\ (DI) \end{Bmatrix}$$

Let's see one example, to clear the concept. MOV [BX] [SI], AL Copy contents of AL register (byte) to memory location offset is in [BX] [SI] i.e. [BX + SI].

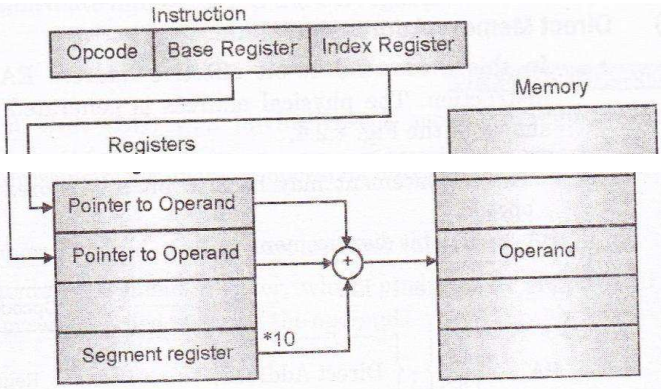


Fig. 4.2.6 : Based indexed addressing mode

6. Register relative addressing mode

- EA is Sum of 8 or 16 bit displacement and contents of base register or an index register
- PA=This addr. Is added with Seg.Regi*10 H

MOV[BX+1100],AL

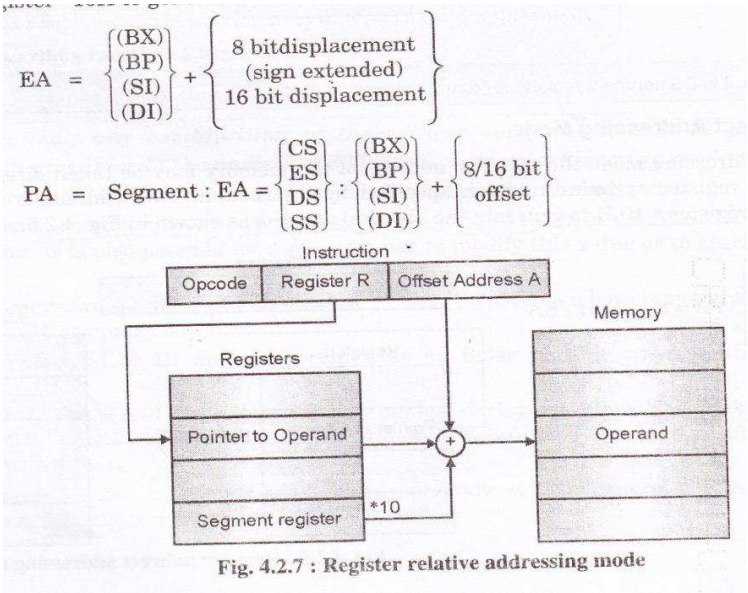


Fig. 4.2.7 : Register relative addressing mode

7. Relative based indexed mode

- EA is Sum of a Base register ,an Index Register and Displacement .
- PA=This addr. Is added with Seg.Regi*10 H

Eg: MOV CX,[BX+SI+0400]

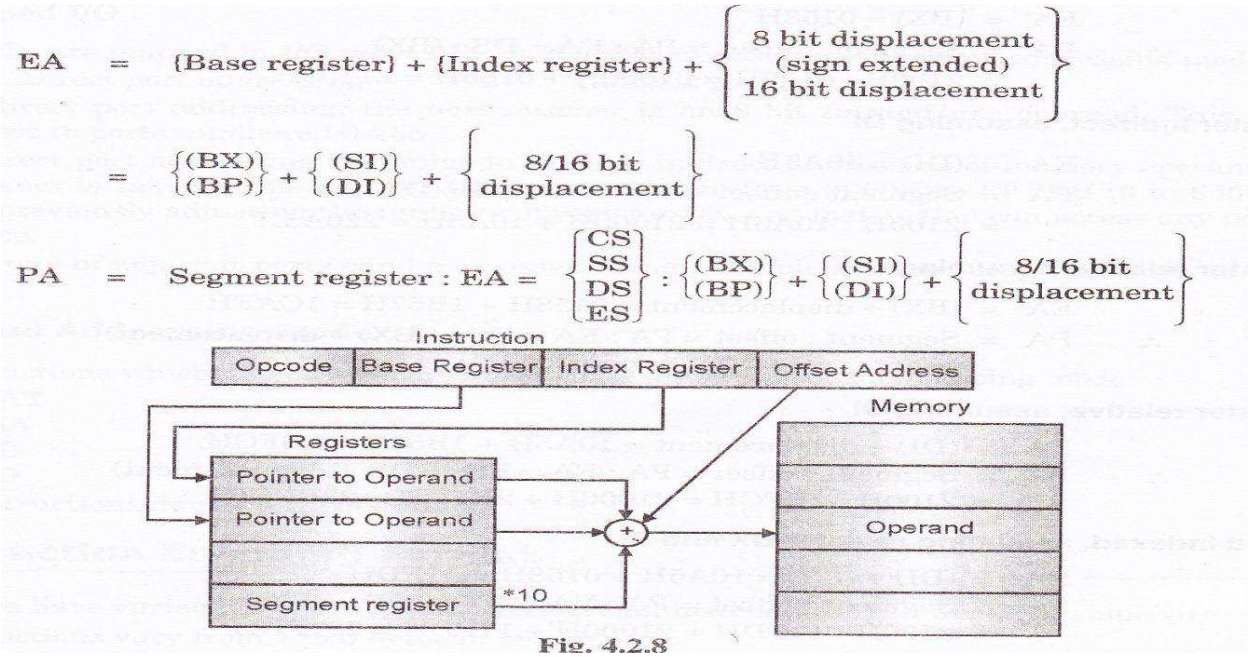
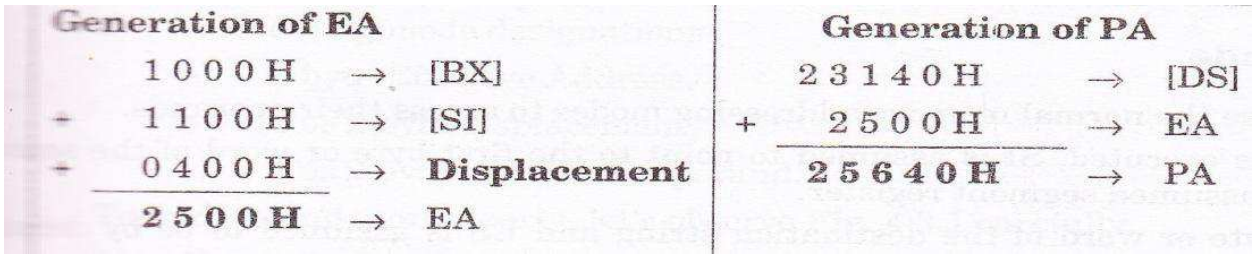


Fig. 4.2.8



8. Implied addressing mode

In Implicit or Implied addressing modes no operands are used to execute the instruction.

Eg: NOP No operation

CLC Clear carry flag to 0

8086 INSTRUCTION SET

DATA TRANSFER INSTRUCTIONS

MOV – MOV Destination, Source

The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location or an immediate number. The source and destination cannot both be memory locations. They must both be of the same type (bytes or words). MOV instruction does not affect any flag.

- MOV CX, 037AH Put immediate number 037AH to CX
- MOV BL, [437AH] Copy byte in DS at offset 437AH to BL
- MOV AX, BX Copy content of register BX to AX
- MOV DL, [BX] Copy byte from memory at [BX] to DL
- MOV DS, BX Copy word from BX to DS register

XCHG – XCHG Destination, Source

The XCHG instruction exchanges the content of a register with the content of another register or with the content of memory location(s). It cannot directly exchange the content of two memory

locations. The source and destination must both be of the same type (bytes or words). This instruction does not affect any flag.

- XCHG AX, DX Exchange word in AX with word in DX
- XCHG BL, CH Exchange byte in BL with byte in CH
- XCHG AL, PRICES [BX] Exchange byte in AL with byte in memory at EA = PRICE [BX] in DS.

LEA – LEA Register, Source

This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register. LEA does not affect any flag.

- LEA BX, PRICES Load BX with offset of PRICE in DS
- LEA BP, SS: STACK_TOP Load BP with offset of STACK_TOP in SS
- LEA CX, [BX][DI] Load CX with EA = [BX] + [DI]

LDS – LDS Register, Memory address of the first word

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.

- LDS BX, [4326] Copy content of memory at displacement 4326H in DS to BL, content of 4327H to BH. Copy content at displacement of 4328H and 4329H in DS to DS register.

LES – LES Register, Memory address of the first word

This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory locations is copied into the specified register, and the word from the next two memory locations is copied into the ES register. LES does not affect any flag.

- LES BX, [789AH] Copy content of memory at displacement 789AH in DS to BL, content of 789BH to BH, content of memory at displacement 789CH

and 789DH in DS is copied to ES register.

STACK RELATED INSTRUCTIONS

PUSH – PUSH Source

The PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment to which the stack pointer points. The source of the word can be general purpose register, segment register, or memory. This instruction does not affect any flag.

- PUSH BX Decrement SP by 2, copy BX to stack.

POP – POP Destination

The POP instruction copies a word from the stack location pointed to by the stack pointer to a destination specified in the instruction. The destination can be a general-purpose register, a segment register or a memory location. The POP instruction does not affect any flag.

- POP DX Copy a word from top of stack to DX; increment SP by 2

PUSHF (PUSH FLAG REGISTER TO STACK)

The PUSHF instruction decrements the stack pointer by 2 and copies a word in the flag register to two memory locations in stack pointed to by the stack pointer. The stack segment register is not affected. This instruction does to affect any flag.

POPF (POP WORD FROM TOP OF STACK TO FLAG REGISTER)

The POPF instruction copies a word from two memory locations at the top of the stack to the flag register and increments the stack pointer by 2. The stack segment register and word on the stack are not affected. This instruction does to affect any flag

INPUT-OUTPUT INSTRUCTIONS

IN – IN Accumulator, Port

The IN instruction copies data from a port to the AL or AX register. If an 8-bit port is read, the data will go to AL. If a 16-bit port is read, the data will go to AX. The IN instruction has two possible formats, fixed port and variable port. For fixed port type, the 8-bit address of a port is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

- IN AL, OC8H Input a byte from port OC8H to AL
- IN AX, 34H Input a word from port 34H to AX

OUT – OUT Port, Accumulator

The OUT instruction copies a byte from AL or a word from AX to the specified port. The OUT instruction has two possible forms, fixed port and variable port. For the fixed port form, the 8-bit port address is specified directly in the instruction.

- OUT 3BH, AL Copy the content of AL to port 3BH
- OUT 2CH, AX Copy the content of AX to port 2CH

ARITHMETIC INSTRUCTIONS

ADD – ADD Destination, Source

ADC – ADC Destination, Source

These instructions add a number from some *source* to a number in some *destination* and put the result in the specified destination. The ADC also adds the status of the carry flag to the result. The source may be an immediate number, a register, or a memory location. The destination may be a register or a memory location. The source and the destination in an instruction cannot both be memory locations. The source and the destination must be of the same type (bytes or words). Flags affected: AF, CF, OF, SF, ZF.

- ADD AL, 74H Add immediate number 74H to content of AL. Result in AL
- ADC CL, BL Add content of BL plus carry status to content of CL
- ADD DX, BX Add content of BX to content of DX

SUB – SUB Destination, Source

SBB – SBB Destination, Source

These instructions subtract the number in some *source* from the number in some *destination* and put the result in the destination. The SBB instruction also subtracts the content of carry flag from the destination. The source may be an immediate number, a register or memory location. The destination can also be a register or a memory location. However, the source and the destination cannot both be memory location. Flags affected: AF, CF, OF, PF, SF, ZF.

- SUB CX, BX CX – BX; Result in CX
- SBB CH, AL Subtract content of AL and content of CF from content of CH. Result in CH
- SUB AX, 3427H Subtract immediate number 3427H from AX

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.

- MUL BH Multiply AL with BH; result in AX
- MUL CX Multiply AX with CX; result high word in DX, low word in AX
- MUL BYTE PTR [BX] Multiply AL with byte in DS pointed to by [BX]

IMUL – IMUL Source

This instruction 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. The source can be a register or a memory location. When a byte from source is multiplied with content of AL, the signed result (product) will be put in AX. When a word from source is multiplied by AX, the result is put in DX and 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. 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.

- IMUL BH Multiply signed byte in AL with signed byte in BH result in AX.
- IMUL AX Multiply AX times AX; result in DX and AX

DIV – DIV Source

This instruction is used to divide an *unsigned* word by a byte or to divide an *unsigned* double word (32 bits) by a word. 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. After the division, AL will contain the 8-bit quotient, and AH will contain the 8-bit remainder. 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.

- DIV BL Divide word in AX by byte in BL; Quotient in AL, remainder in AH

- DIV CX Divide down word in DX and AX by word in CX; Quotient in AX, and remainder in DX.

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. 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. 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. 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

- IDIV BL Signed word in AX/signed byte in BL
- IDIV BP Signed double word in DX and AX/signed word in BP

INC – INC Destination

The INC instruction adds 1 to a specified register or to a memory location. AF, OF, PF, SF, and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result will be all 0's with no carry.

- INC BL Add 1 to contains of BL register
- INC CX Add 1 to contains of CX register

DEC – DEC Destination

This instruction subtracts 1 from the destination word or byte. The destination can be a register or a memory location. AF, OF, SF, PF, and ZF are updated, but CF is not affected. This means

that if an 8-bit destination containing 00H or a 16-bit destination containing 0000H is decremented, the result will be FFH or FFFFH with no carry (borrow).

- DEC CL Subtract 1 from content of CL register
- DEC BP Subtract 1 from content of BP register

DAA (DECIMAL ADJUST AFTER BCD ADDITION)

This instruction is used to make sure the result of adding two packed BCD numbers is adjusted to be a legal BCD number. The result of the addition must be in AL for DAA to work correctly. If the lower nibble in AL after an addition is greater than 9 or AF was set by the addition, then the DAA instruction will add 6 to the lower nibble in AL. If the result in the upper nibble of AL in now greater than 9 or if the carry flag was set by the addition or correction, then the DAA instruction will add 60H to AL

DAS (DECIMAL ADJUST AFTER BCD SUBTRACTION)

This instruction is used after subtracting one packed BCD number from another packed BCD number, to make sure the result is correct packed BCD. The result of the subtraction must be in AL for DAS to work correctly. If the lower nibble in AL after a subtraction is greater than 9 or the AF was set by the subtraction, then the DAS instruction will subtract 6 from the lower nibble AL. If the result in the upper nibble is now greater than 9 or if the carry flag was set, the DAS instruction will subtract 60 from AL.

CBW (CONVERT SIGNED BYTE TO SIGNED WORD)

This instruction copies the sign bit of the byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. CBW does not affect any flag.

CWD (CONVERT SIGNED WORD TO SIGNED DOUBLE WORD)

This instruction copies the sign bit of a word in AX to all the bits of the DX register. In other words, it extends the sign of AX into all of DX. CWD affects no flags.

LOGICAL INSTRUCTIONS

AND – AND Destination, Source

This instruction ANDs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the

specified source is not changed. 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 0 after AND. PF, SF, and ZF are updated by the AND instruction.

- AND CX, [SI] AND word in DS at offset [SI] with word in CX register;Result in CX register
- AND BH, CL AND byte in CL with byte in BH; Result in BH
- AND BX, 00FFH 00FFH Masks upper byte, leaves lower byte unchanged.

OR – OR Destination, Source

This instruction ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

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 destination cannot both be memory locations. CF and OF are both 0 after OR. PF, SF, and ZF are updated by the OR instruction.

- OR AH, CL CL ORed with AH, result in AH; CL not changed
- OR BL, 80H BL ORed with immediate number 80H; sets MSB of BL to 1

XOR – XOR Destination, Source

This instruction Exclusive-ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed. 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 destination cannot both be memory locations. CF and OF are both 0 after XOR. PF, SF, and ZF are updated.

- XOR CL, BH Byte in BH exclusive-ORed with byte in CL. Result in CL. BH not changed.

NOT – NOT Destination

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.

NEG – 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. 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

CMP – CMP Destination, Source

This instruction compares a byte / word in the specified source with a byte / word in the specified destination. The source can be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, the source and the destination cannot both be memory locations. The comparison is actually done by subtracting the source byte or word from the destination byte or word. The source and the destination are not changed, but the flags are set to indicate the results of the comparison. AF, OF, SF, ZF, PF, and CF are updated by the CMP instruction. For the instruction CMP CX, BX, the values of CF, ZF, and SF will be as follows:

	CF	ZF	SF	
CX = BX	0	1	0	Result of subtraction is 0
CX > BX	0	0	0	No borrow required, so CF = 0
CX < BX	1	0	1	Subtraction requires borrow, so CF = 1

CMP AL, 01H Compare immediate number 01H with byte in AL

CMP BH, CL Compare byte in CL with byte in BH

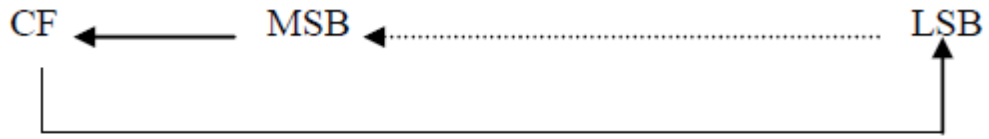
TEST – TEST Destination, Source This instruction ANDs the byte / word in the specified source with the byte / word in the specified destination. Flags are updated, but neither operand is changed. The test instruction is often used to set flags before a Conditional jump instruction. The source can be 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 0's after TEST. PF, SF and ZF will be updated to show the results of the destination.

TEST AL, BH AND BH with AL. No result stored; Update PF, SF, ZF.

ROTATE AND SHIFT INSTRUCTIONS

RCL – RCL Destination, Count 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.

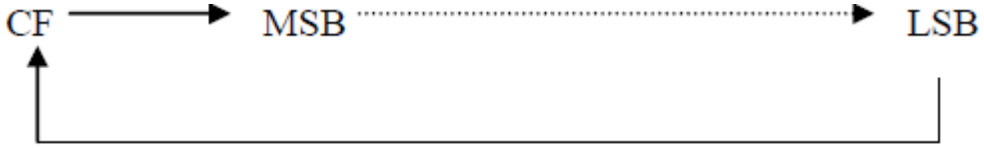


For multi-bit rotates, CF will contain the bit most recently rotated out of the MSB. The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate by more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

- RCL DX, 1 Word in DX 1 bit left, MSB to CF, CF to LSB

RCR – RCR Destination, Count

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.



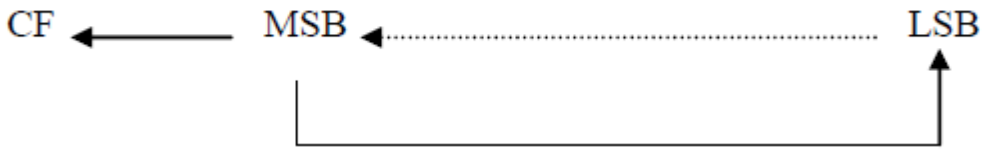
For multi-bit rotate, CF will contain the bit most recently rotated out of the LSB. The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate more than one bit

position, load the desired number into the CL register and put "CL" in the count position of the instruction.

- RCR BX, 1 Word in BX right 1 bit, CF to MSB, LSB to CF.

ROL – ROL Destination, Count

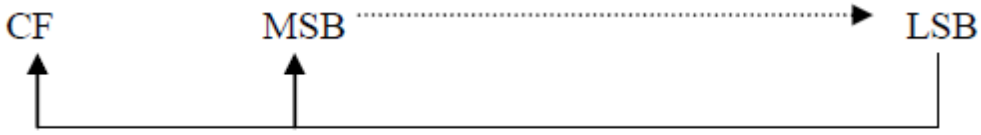
This instruction rotates all the bits in a specified word or byte to the left some number of bit positions. The data bit rotated out of MSB is circled back into the LSB. It is also copied into CF. In the case of multiple-bit rotate, CF will contain a copy of the bit most recently moved out of the MSB.



- ROL AX, 1 Rotate the word in AX 1 bit position left, MSB to LSB and CF

ROR – ROR Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to right. The operation is desired as a rotate rather than shift, because the bit moved out of the LSB is rotated around into the MSB. The data bit moved out of the LSB is also copied into CF. In the case of multiple bit rotates, CF will contain a copy of the bit most recently moved out of the LSB.



- ROR BL, 1 Rotate all bits in BL right 1 bit position LSB to MSB and to CF

SAL – SAL Destination, Count

SHL – SHL Destination, Count

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination some number of bit positions to the left. As a bit is shifted out of the LSB operation, a 0 is put in the LSB position. The MSB will be shifted into CF. In the case of multi-bit shift, CF will contain the bit most recently shifted out from the MSB. Bits shifted into CF previously will be lost.

The destination operand can be a byte or a word. It can be in a register or in a memory location. 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.

- SAL BX, 1 Shift word in BX 1 bit position left, 0 in LSB

SAR – SAR Destination, Count

This instruction shifts each bit in the specified destination some number of bit positions 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.

The destination operand can be a byte or a word. It can be in a register or in a memory location. 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.

- SAR DX, 1 Shift word in DI one bit position right, new MSB = old MSB

SHR – SHR Destination, Count

This instruction shifts each bit in the specified destination some number of bit positions 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.

The destination operand can be a byte or a word in a register or in a memory location. 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.

- SHR BP, 1 Shift word in BP one bit position right, 0 in MSB

TRANSFER-OF-CONTROL INSTRUCTIONS

JMP (UNCONDITIONAL JUMP TO SPECIFIED DESTINATION)

This instruction will fetch the next instruction from the location specified in the instruction rather than from the next location after the JMP instruction. If the destination is in the same code segment as the JMP instruction, then only the instruction pointer will be changed to get the destination location. This is referred to as a near jump. If the destination for the jump instruction is in a segment with a name different from that of the segment containing the JMP instruction, then both the instruction pointer and the code segment register content will be changed to get the destination location. This referred to as a far jump. The JMP instruction does not affect any flag.

- **JMP CONTINUE** This instruction fetches the next instruction from address at label CONTINUE. If the label is in the same segment, an offset coded as part of the instruction will be added to the instruction pointer to produce the new fetch address. If the label is another segment, then IP and CS will be replaced with value coded in part of the instruction. This type of jump is referred to as direct because the displacement of the destination or the destination itself is specified directly in the instruction

JA / JNBE (JUMP IF ABOVE / JUMP IF NOT BELOW OR EQUAL)

If, after a compare or some other instructions which affect flags, the zero flag and the carry flag both are 0, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are not both 0, the instruction will have no effect on program execution

JAE / JNB / JNC (JUMP IF ABOVE OR EQUAL / JUMP IF NOT BELOW / JUMP IF NO CARRY)

JB / JC / JNAE (JUMP IF BELOW / JUMP IF CARRY / JUMP IF NOT ABOVE OR EQUAL

JBE / JNA (JUMP IF BELOW OR EQUAL / JUMP IF NOT ABOVE)

JG / JNLE (JUMP IF GREATER / JUMP IF NOT LESS THAN OR EQUAL)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction, if the zero flag is 0 and the carry flag is the same as the overflow flag.

JGE / JNL (JUMP IF GREATER THAN OR EQUAL / JUMP IF NOT LESS THAN)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction, if the sign flag is equal to the overflow flag.

JL / JNGE (JUMP IF LESS THAN / JUMP IF NOT GREATER THAN OR EQUAL)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the sign flag is not equal to the overflow flag.

JLE / JNG (JUMP IF LESS THAN OR EQUAL / JUMP IF NOT GREATER)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the zero flag is set, or if the sign flag not equal to the overflow flag.

JE / JZ (JUMP IF EQUAL / JUMP IF ZERO)

This instruction is usually used after a Compare instruction. If the zero flag is set, then this instruction will cause a jump to the label given in the instruction.

JNE / JNZ (JUMP NOT EQUAL / JUMP IF NOT ZERO)

This instruction is usually used after a Compare instruction. If the zero flag is 0, then this instruction will cause a jump to the label given in the instruction.

JS (JUMP IF SIGNED / JUMP IF NEGATIVE)

This instruction will cause a jump to the specified destination address if the sign flag is set.

JNS (JUMP IF NOT SIGNED / JUMP IF POSITIVE)

This instruction will cause a jump to the specified destination address if the sign flag is 0.

JP / JPE (JUMP IF PARITY / JUMP IF PARITY EVEN)

If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is even, then the parity flag will be set.

JNP / JPO (JUMP IF NO PARITY / JUMP IF PARITY ODD) If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is odd, then the parity flag is 0.

JNO (JUMP IF NO OVERFLOW)

The overflow flag will be set if some signed arithmetic operation is too large to fit in the destination register or memory location. The JNO instruction will cause a jump to the destination given in the instruction, if the overflow flag is not set.

JCXZ (JUMP IF THE CX REGISTER IS ZERO)

This instruction will cause a jump to the label to a given in the instruction, if the CX register contains all 0's. The instruction does not look at the zero flag when it decides whether to jump or not

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.

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.

LOOPE NEXT LOOPNE / LOOPNZ (LOOP WHILE CX ≠ 0 AND ZF = 0)

This instruction is used to repeat a group of instructions some number of times, or until the zero flag becomes a 1. The number of times the instruction sequence is to be repeated is loaded into the count register CX. Each time the LOOPNE / LOOPNZ instruction executes, CX is automatically decremented by 1. If CX ≠ 0 and ZF = 0, execution will jump to a destination specified by a label in the instruction. If CX = 0, after the auto decrement or if ZF = 1, execution simply go on the next instruction after LOOPNE / LOOPNZ. In other words, the two ways to exit the loop are CX = 0 or ZF = 1.

LOOPNE / LOOPNZ (LOOP WHILE CX ≠ 0 AND ZF = 0)

This instruction is used to repeat a group of instructions some number of times, or until the zero flag becomes a 1. The number of times the instruction sequence is to be repeated is loaded into

the count register CX. Each time the LOOPNE / LOOPNZ instruction executes, CX is automatically decremented by 1. If CX ≠ 0 and ZF = 0, execution will jump to a destination specified by a label in the instruction. If CX = 0, after the auto decrement or if ZF = 1, execution simply go on the next instruction after LOOPNE / LOOPNZ. In other words, the two ways to exit the loop are CX = 0 or ZF = 1.

CALL (CALL A PROCEDURE)

The CALL instruction is used to transfer execution to a subprogram or a procedure. There two basic type of calls near and far.

1. A near call is a call to a procedure, which is in the same code segment as the CALL instruction. When the 8086 executes a near CALL instruction, it decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL into the stack. This offset saved in the stack is referred to as the return address, because this is the address that execution will return to after the procedure is executed. A near CALL instruction will also load the instruction pointer with the offset of the first instruction in the procedure. A RET instruction at the end of the procedure will return execution to the offset saved on the stack which is copied back to IP.

2. A far call is a call to a procedure, which is in a different segment from the one that contains the **CALL** instruction. When the 8086 executes a far call, it decrements the stack pointer by 2 and copies the content of the CS register to the stack. It then decrements the stack pointer by 2 again and copies the offset of the instruction after the CALL instruction to the stack.

- CALL MULT This is a direct within segment (near or intra segment) call. MULT is the name of the procedure. The assembler determines the displacement of MULT from the instruction after the CALL and codes this displacement in as part of the instruction.

RET (RETURN EXECUTION FROM PROCEDURE TO CALLING PROGRAM)

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction which was used to call the procedure. If the procedure is near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the IP with a word from the top of the stack. The word from the top of the stack is the offset of the next instruction after the CALL. This offset was pushed into the stack as part of the operation of the CALL instruction. The stack pointer will be incremented by 2 after the return address is popped off the stack.

STRING MANIPULATION INSTRUCTIONS

MOVS – MOVS Destination String Name, Source String Name

MOVSB – MOVSB Destination String Name, Source String Name

MOVSW – MOVSW Destination String Name, Source String Name

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 a 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. If DF is 0, then SI and DI will 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.

- MOV SI, OFFSET SOURCE — Load offset of start of source string in DS into SI
  MOV DI, OFFSET DESTINATION — Load offset of start of destination 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 REP
  MOVSB — Move string byte until CX = 0

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.

- 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

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.

- MOV DI, OFFSET TARGET
  STOS TARGET

CMPS / CMPSB / CMPSW (COMPARE STRING BYTES OR STRING WORDS)

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. 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.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

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. 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.

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

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.

- 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

FLAG MANIPULATION INSTRUCTIONS

STC (SET CARRY FLAG)

This instruction sets the carry flag to 1. It does not affect any other flag.

CLC (CLEAR CARRY FLAG)

This instruction resets the carry flag to 0. It does not affect any other flag.

CMC (COMPLEMENT CARRY FLAG)

This instruction complements the carry flag. It does not affect any other flag.

STD (SET DIRECTION FLAG)

This instruction sets the direction flag to 1. It does not affect any other flag.

CLD (CLEAR DIRECTION FLAG)

This instruction resets the direction flag to 0. It does not affect any other flag.

STI (SET INTERRUPT FLAG)

Setting the interrupt flag to a 1 enables the INTR interrupt input of the 8086. The instruction will not take affect until the next instruction after STI. When the INTR input is enabled, an

interrupt signal on this input will then cause the 8086 to interrupt program execution, push the return address and flags on the stack, and execute an interrupt service procedure. An IRET instruction at the end of the interrupt service procedure will restore the return address and flags that were pushed onto the stack and return execution to the interrupted program. STI does not affect any other flag.

CLI (CLEAR INTERRUPT FLAG)

This instruction resets the interrupt flag to 0. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input. The CLI instructions, however, has no effect on the non-maskable interrupt input, NMI. It does not affect any other flag.

LAHF (COPY LOW BYTE OF FLAG REGISTER TO AH REGISTER)

The LAHF instruction copies the low-byte of the 8086 flag register to AH register. It can then be pushed onto the stack along with AL by a PUSH AX instruction. LAHF does not affect any flag. **SAHF** (COPY AH REGISTER TO LOW BYTE OF FLAG REGISTER)

The SAHF instruction replaces the low-byte of the 8086 flag register with a byte from the AH register. SAHF changes the flags in lower byte of the flag register.

INTERUPT RELATED INSTRUCTIONS

INT – INT TYPE

The term type in the instruction format refers to a number between 0 and 255, which identify the interrupt. When an 8086 executes an INT instruction, it will 1. Decrement the stack pointer by 2 and push the flags on to the stack. 2. Decrement the stack pointer by 2 and push the content of CS onto the stack. 3. Decrement the stack pointer by 2 and push the offset of the next instruction after the INT number instruction on the stack. 4. Get a new value for IP from an absolute memory address of 4 times the type specified in the instruction.

INT 3 This is a special form, which has the single-byte code of CCH

INTO (INTERRUPT ON OVERFLOW)

If the overflow flag (OF) is set, this instruction causes the 8086 to do an indirect far call to a procedure you write to handle the overflow condition. Before doing the call, the 8086 will

1. Decrement the stack pointer by 2 and push the flags on to the stack.

2. Decrement the stack pointer by 2 and push CS on to the stack.

3. Decrement the stack pointer by 2 and push the offset of the next instruction after INTO instruction onto the stack.

4. Reset TF and IF. Other flags are not affected.

5. To do the call, the 8086 will read a new value for IP from address 00010H and a new value of CS from address 00012H.

IRET (INTERRUPT RETURN)

When the 8086 responds to an interrupt signal or to an interrupt instruction, it pushes the flags, the current value of CS, and the current value of IP onto the stack. It then loads CS and IP with the starting address of the procedure, which you write for the response to that interrupt. The IRET instruction is used at the end of the interrupt service procedure to return execution to the interrupted program. To do this return, the 8086 copies the saved value of IP from the stack to IP, the stored value of CS from the stack to CS, and the stored value of the flags back to the flag register.

Stacks

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.The process of storing the data in the stack is called 'pushing into' the stack and the reverse process of transferring the data back from the stack to the CPU register is known as 'popping off' the stack. The stack is essentially *Last-In-First-Out* (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first.

The stack pointer is a 16-bit register that contains the offset address of the memory location in the stack segment. The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the base address of the stack segment in the memory.

The Stack Segment register (SS) and Stack pointer register (SP) together address the stack-top as explained below:

SS ⟹ 5000H
SP ⟹ 2050H

If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data. The next 16 bit push operation will decrement the stack pointer by two, so that it will point to the new stack-top 5204EH and the decremented contents of SP will be 204EH. This location will now be occupied by the recently pushed data.Thus for a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations. If the SP starts with an initial value of FFFFH, it will be decremented by two whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stack overflow.

After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified address in the CALL instruction i.e. starting address of the procedure. Then the procedure is executed.

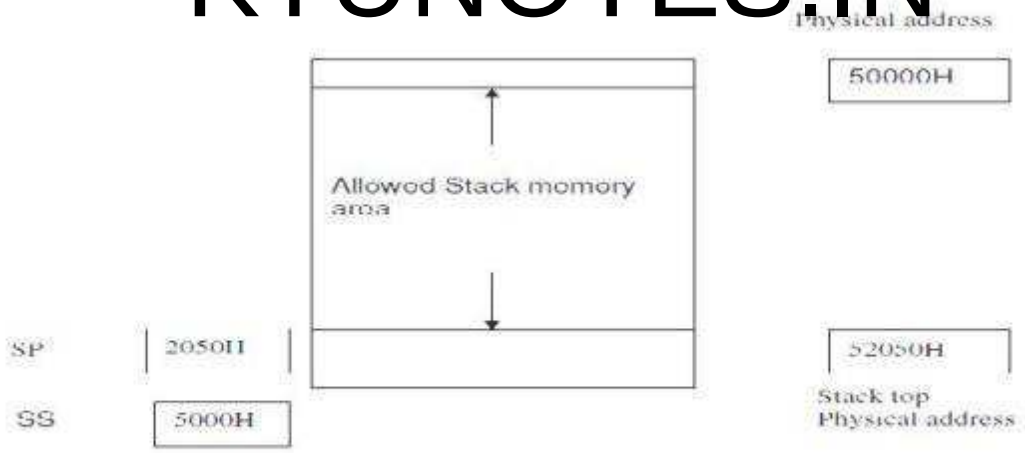


Fig. 1.11 Stack –top address calculation

Procedures

A procedure is a set of code that can be branched to and returned from in such a way that the code is as if it were inserted at the point from which it is branched to. The branch to procedure is

referred to as the *call,* and the corresponding branch back is known as the *return*. The return is always made to the instruction immediately following the call regardless of where the call is located.

1. Calls, Returns, and Procedure Definitions

The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. The RET instruction simply pops the return address from the stack. The registers used by the procedure need to be stored before their contents are changed, and then restored just before their contents are changed, and then restored just before the procedure is excited. A CALL may be direct or indirect and intra segment or intersegment. If the CALL is intersegment, the return must be intersegment. Intersegment call must push both (IP) and (CS) onto the stack. The return must correspondingly pop two words from the stack. In the case of intra segment call, only the contents of IP will be saved and retrieved when call and return instructions are used.

Procedures are used in the source code by placing a statement of the form at the beginning of the procedure. **Procedure name PROC** Attribute and by terminating the procedure with a statement **Procedure name ENDP**

Macros

Disadvantages of Procedure:

1. Linkage associated with them.

2. It sometimes requires more code to program the linkage than is needed to perform the task. If this is the case, a procedure may not save memory and execution time is considerably increased.

3. **Macros** is needed for providing the programming ease of a procedure while avoiding the linkage. Macro is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each reference.

A macro is unlike a procedure in that the machine instructions are repeated each time the macro is referenced. Therefore, no memory is saved, but programming time is conserved (no linkage is required) and some degree of modularity is achieved. The code that is to be repeated is called the

prototype code. The prototype code along with the statements for referencing and terminating is called the macro definition.

Once a macro is defined, it can be inserted at various points in the program by using macro calls. When a macro call is encountered by the assembler, the assembler replaces the call with the macro code. Insertion of the macro code by the assembler for a macro call is referred to as a macro expansion. In order to allow the prototype code to be used in a variety of situations, macro definition and the prototype code can use dummy parameters which can be replaced by the actual parameters when the macro is expanded. During a macro expansion, the first actual parameter replaces the first dummy parameter in the prototype code; the second actual parameter replaces the second dummy parameter, and so on. A macro call has the form **Macro name** (Actual parameter list) with the actual parameters being separated by commas.