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 be both memory locations. They must be of the same type (byte or words) MOV instruction does not affect any flag.

→ MOV CX, 037AH PUT immediate number 037AH into 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 AX.

→ MOV, DS, BX copy word from BX to DS register.

→ MOV RESULT [BP], AX COPY AX to two memory locations, AL to first location, AL to second location.

EA of the memory location is sum of the displacement represented by RESULTS and content of BP.
Physical address = EA + SS.

→ MOV, ES: RESULT [BP], AX. Same as the before but physical address is EA + ES, because the segment override prefix ES.

LEA – LEA Register, Source

This instruction determines the offset of the varriable or memory location named as the source and put this offset in the 16-bits registers. LEA doesnot 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]

ARITHMETIC INSTRUCTIONS

ADD - ADD Destination, Source

ADC - ADC Destination, Source

These instruction add a number from Source to a number in some destination and put the result in specified destination. ADC also add the status of of the carry flag to the result. The source may be an immediate number, a register, or a memory location. The ~~register~~ destination may be a register or a memory location. The source and destination in an instruction cannot be both memory locations. The source and destination must be of same type. (byte or word). If you want to add a byte to a word you must copy a byte to a word location and fill the upper byte of the word with 0's before adding. Flag were affected: AF, CF, OF, SF, ZF.

ADD AL, 74H ADD immediate numbe 74H to content of AL and Result in AL.

ADD CL, BL Add content of BL to content of CL and result in CL.

ADD DX, CX Add content of CX to DX.

ADD DX, [SI] Add word from memory at offset [SI] in DS to content of DX.

ADC AL, PRICE[BX] Add byte from EA PRICE [BX] plus carry status to content of AL

ADD AL, PRICES[BX] Add content of memory at effective address PRICE[BX] to AL

SUB - SUB. DESTINATION, SOURCE

SBB - SBB DESTINATION, SOURCE

These instructions subtract the number in source from the number in some destination and put the result in Destination. The SBB instruction also subtracts the content of carry flag from the destination.

The source may be an destination, registers, immediate number or memory locations. The destination may be and memory locations or an registers. However the source and destination both can not be ~~are~~ memory locations. The destination ~~can also be a reg~~ and source must be of same type (word or byte). If we want to subtract a byte from a word we have to move the byte in to a word location and fill the upper 8 bits with 0's. ~~And~~ Flags affected: CF, OF, ZF, SF, AF.

SUB BX, BX — CX − BX, Result in CX

SBB CH, AL — Subtract content of AL ~~from~~ ~~And~~ content of CH ~~from~~ Result in CH. also subtra carray.

SUB AX, 3274H — Subtratet immediate number 3274H from AX.

SBB BX, [342ZH] — Subtract word at displace -ment 342ZH in DS from BX declared as word byte as PT

SUB PRICES[BX], 04H Subtract byte from offset PRICES[BX] to 04H imimidiate value.

SBB CX, TABLE[BX], Subtract word from effective address TABLE[BX] of Ds and Status of CF from CX.

SBB TABLE[BX], CX Subtract content of CX and status of CF from value located at offset TABLE[BX] of DS.

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 canbe a register or a memory location. When a byte is multiplied by the content of AL. the result 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 bit of the a 16 bit result or 32 bit result is 0 CF and OF both will be zero. AF, PF, SF, ZF are undefined after a MUL instructions.

If you want to multiply a byte with a word you must put the byte in word location and fill the upper byte with all 0's. we cannot use CBW for fill the upper byte with copies of most significant bit of the lower byte.

MUL BH — Multiply AL with BH — result in AX.

MUL CX — Multiply AX with CX result high word in DX and low word in AX.

MUL BYTE PTR [BX] — multiply AX with byte pointed by [BX]

MUL FACTOR [BX] — Multiply AL with byte at EA factor [BX]. if it is declared with type byte as DB Multiply Factor [BX] if it is declared as word byte as DW.

MOV AX, MCAND_16	LOAd 16 multiplicand into AX
MOV CL, MPLIER_8	load 8-bit multiplier into CL
MOV CH, 00H	set upper bit CX '0'
MUL CX	AX times CX, 32-bit result in DX and AX.

DIV - DIV source

This instruction is used for divide an unsigned word by a byte or to devide an unsigned double word ~~with~~ by an word: When a word is divide by a byte the word must be in AX register. After the division AL will contain 8-bit quotient and AH will contain 8-bit remainder. when a double word is divide by a word the most significant word of the double word must be in DX, and the least signifieant word of the double word must be in AX. After the division AX will contain ~~upper~~ 16 bit of the quotient and DX will contain 16 bit remainder. If an attempt is made to divide by 0 or if the quotient is so large to fit in the

destination (greater than FFH / FFFFH), the 8086 will generate a type 0 interrupt. All flags are undefined after the DIV instructions.

If you want to divide a byte by byte you must put the divident byte in AL and fill AH with all 0's. Likewise, if you want to divide a word by another word, then put the dividend word in AX and fill DX with all 0's.

DIV BL — Divide AX by BL. Quotient in AL and remainder in AH.

DIV CX — Divide DX and AX by CX. Quotient in AX remainder in DX.

DIV SCALE [BX] — if AX / (byte at effective address scale [BX]) if SCALE[BX] is of type byte or (DX an AX) / (scale [BX] if type word].

INC - INC. Destination

The INC instruction adds 1 to ~~the~~ a specified register or to a memory location.

AF, OF, PF, SF and ZF are updated. but CF is not affected. This means if a 8 bit register contain FFH or a 16 bit contain FFFFH is incremented the result will be 0 with no carry.

INC BL ~~inc~~ Add 1 to contains of BL.

INC CX add 1 to contains of CX

INC BYTE PTR[BX] add 1 to ~~data~~ segment at offset contaiint in BX.

INC WORD PTR[BX] add 1 to data segment at offset contain in [BX] and [BX+1]

INC TEMP Increment TEMP in the data segment

INC PRICES[BX] Increment elemented pointed by (BX) in array prices. increment word if prices declared with DW or increment byte if PRICES declared with DB.

DEC- DEC destination

This instruction subtracts 1 from the destination word or byte. The destination canbe register or memory locations.

AF, OF, SF, PF, ZF are updated but CF not affected. That means if an 8 bit register containing 00H or a 16-bit register 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 the content of BP

→ DEC BYTE PTR [BX] Subtract 1 from byte at offset [BX] in DS.

→ DEC WORD PTR [BP] subtract 1 from word at offset [BP] in SS

→ DEC COUNT Subtract 1 from byte if count is declared with DB otherwise if COUNT declared with DW then subtract 1 from word.

DAA (DECimal adjust after bcd addition):

This instruction is used to make sure that the results of adding two packed BCD number is adjusted to be a legal BCD number. The result of the addition must be in AL for DAA work correctly. If the lower nibble in AL is after addition is greated than 9 or AF was set by

the addition then DAA instruction will add 6H to the ~~low AL~~ lower nibble in AL. If the result in the upper nibble of AL is now greater than 9 or if the carry flag is set by the addition or correction, then DAA instruction will add 60H to AL.

Let AL = 59 BCD, BL = 35 BCD

ADD AL, BL	AL = 8EH	lower nibble > 9 add 06H to AL.
DAA	AL = 94 BCD, CF = 0.	

Let AL = 88 BCD and BL = 49 BCD

ADD AL, BL	AL = D1H;	AF = 1 add 06 to AL
DAA	AL = D7H	Upper nibble > 9 add 60H to AL
	AL = 37 BCD	CF = 1.

The DAA instruction updates AF, CF, SF, PF and ZF. but OF is undefined.

AAA. (ASCII Adjust for addition).

Numerical data coming into a computer from ~~te~~ a terminal is usually in ASCII code. In this code the number 0 to 9 represented by the ASCII codes 03H to 39H. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each. After the addition, the AAA instruction is

is used to make sure that the result is correct unpacked ~~(BCD)~~ BCD.

Let. AL = 00110101 (ASCII 5) ad BL = 00111001 (ASCII 9)

ADD AL, BL AL = 01101110 (6EH, which is incorrect BCD)

AAA AL = 00000100 (unpacked BCD 4)

CF = 1 indicates answer is 14 Decimal.

The AAA instruction works only on the AL register. The AAA instruction updates AF and CF but OF, ZF, PF, SF are left undefined.

LOGICAL INSTRUCTIONS

AND- AND DESTINATION, SOURCE.

This instructions AND's each bit in a source byte or word with the same numbred byte or word in destination. The result will put in specified destination. The content of specified source is not changed. The source can be an immidiate number, registers or memory location. The destination canbe register or memory location. The source and destination cannot be both memory location. CF and OF both are 0 after AND. PF, SF and ZF are updated by AND instructions.

AF is undefined. PF has meaning only for an 8-bit operand.

AND CX, [SI] — AND word in DS at offset [SI] with word in CX. Result in CX.

AND BH, CL — AND byte CL with byte BH result in BH.

AND BX, 00FFH — AND immediate value 00FFH with value in BX result in BX. 00FFH masks upper byte lower byte will be 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 canbe immediate number, register or memory location. The destination can be register or memory location, Both source and destination can not be memory location. CF and OF both zero after OR. PF, SF and ZF are updated by the OR instructions. AF is undefined PF has meaning only for an 8-bit operand.

OR AH, CL — CL ORed with AH. result in AH
CL is not changed.

OR BP, SI — SI ORed with BP. result in BP
SI not changed.

OR BL, 80H — BL ORed with immediate value 80H
Set MSB of BL to 1.

OR CX, TABLE[SI]. — CX ORed wit word from
EA TABLE[SI]. content of
memory is not changed.

XOR- XOR Source, Destination.

This instruction exclusive-ORs each bit state in source byte or word withe the same number of bits in destination, The source can be immediate number, register or memory location. Destination can be register or memory location. Both source and destination can not be memory location. CF and OF are both 0 after XOR. PF, SF and ZF are updated. PF has meaning only for an 8-bit operant. AF is undefined.

XOR CL, BH — Byte in BH exclusive-OR with byte in CL. result in CL and BH unchanged.

XOR BP, DI — Word in DI exclusive-OR with word in BP. result in BP. DI unchanged.

XOR WORD PTR[BX], 00FFH Exclusive-OR immediate number 00FFH with word at offset [BX] in the data segment. Result store in memory location [BX].

CMP – CMP Destination, Source

This instruction compare a byte/word in a specified source with a byte/word in the specified destination. The source can be immediate number, a register or a memory location. The destination can be memory location or register. But the source and destination both can't be memory location. The comparision actually done by subtracting source from destination. byte/word. The source and destination wont change. but the flags are set to indicate the results of the comparison. AF, OF, SF, ZF, PF and CF are updated by CMP instruction. for the instruction CMP CX, BX. the value of CF, ZF and SF will be as follows:

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

CMP AL, 01H — Compare immediate value 01H with byte in AL.

CMP CX, TEMP — Compare word in DS at displacement temp with word at CX.

CMP PRICES [BX], 49H — Compare immidiate values 49H with byte at offset (BX) in array PRICES.

TEST - TEST Destination, Source.

This instruction ANDs the byte/word in the specified source with the destination. Flags are updated, but neither operand is changed. The test instruction is often used to set the flag before conditional jump instruction. The source can be immediate number, register or memory location. and the destination can be memory location or register. The source and destination can not be both memory location. CF and OF flags both will be 0 after TEST instructions. PF, SF, ZF will be updated to show the results of the destination. AF is be undefined.

TEST AL, BH — AND BH with AL. No result stored. Updated PF, SF. and ZF ~~and CX~~

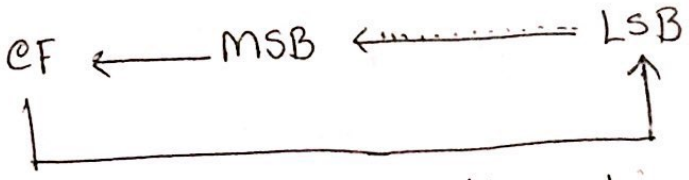
TEST CX, 0001H — AND CX with immediate number 0001H. No result stored, Updated PF, SF and ZF

TEST BP, [BX][DI] — AND WORD are offset [BX][DI] in DS with word BP. No result stored. Updated ZF, SF, PF.

ROTATE AND SHIFT INSTRUCTIONS

RCL - RCL destination, count

This instruction rotate all the bits in a specifie word or byte some number of bits positions to the left. the operation circular because MSB of the operand is rotated into the carry flag and the bit in the carry flag rotate arround into the LSB of the operand.



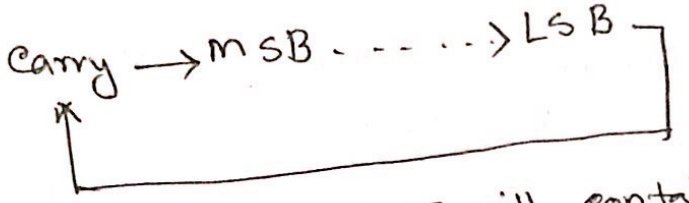
For multi-bits rotate, CF will conbain the bit most recently rotated out of the MSB.
The destination can be a register or a memory location. If you want to rotate the operand by one bit, you can specify by putting 1 in the count position of the instruction. To rotate by more than 1 bit position, load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

RCL affects only CF and OF. OF will be a 1 after a single bit RCL if the MSB is changed by the rotate. OF is undefined after the multibit rotate.

```
RCL DX, 1          Word in DX 1 bit left, MSB to CF, CF to LSB
mov CL, 4          LOAD the number of bit position to rotate
RCL SUM[BX], CL    Rotate byte or word at EA SUM [BX] 4 bits left Original bit 4 now in CF. CF now in bit 3.
```

RCR- RCR Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit position to right. The operation circular because the LSB of the operand is rotated into the carry flag and the bit in carry flag is rotate around to msb of the operand.



for multi bit rotate CF will contain the bit most recently rotated out of the LSB.
The destination can be register or memory locations. If you want to rotate the operand by one bit position than this can be specified by putting 1 in the position of count. To rotate more than 1 number of rotation load in to CR and put "CL" in count position of the instruction.
RCL affects only CF and OF. OF will be a 1 after single bit RCR if the MSB was changed by the rotation. OF is undefined after multi rotate.

RCR BX, 1 — Word in BX rigth 1 bit, CF to MSB and LSB to CF

MOV CL, 4 — Load CF for rotating 4 bit position

RCR BYTE PTR [BX], CL — Rotate the byte at offset [BX] in Ds 4 bit position right CF = original bit 3 and Bit 4 – original CF

SAL - SAL Destination, Count

SHL - SHL Destination, Count

SAL and SHL are two mnemonics of the same ~~function~~. instruction. this instruction shift each bit in the specified destination some number of bits position to the left. As a bit is shift out of the LSB ~~operation~~ position, a 0 is put in the LSB position. the MSB will be shifted in the carry flag. In the case of multibit shift, CF will contain the bit most recently shifted out from the MSB. Bit shifted into CF previously will be lost.

CF ← MSB ← ..... LSB ← 0 .

The destination operand can be either bit or byte word It can be register or memory location. If you want to shift the operand by 1 position. you can specify this by putting a 1 in the count position of the instructions. for shift of more than 1 bit position, load the desired number of shifts in to the CL register, and put "CL" in the count position of the instruction. The flags are affected as follow: CF contains the bit ~~most~~ recently shifted out from MSB. For a count of one, OF will be 1 if and the current MSB are not the same MSB are not the same. For mutiple-bit shifts, OF is ~~the~~ unsigned. SF and ZF will be updated to reflect the condition of the destination. PF will be have meaning only for an operand in AL. AF is undefined.

→ SAL BX, 1 Shift word in BX 1 bit position left.

→ MOV CL, 02H Load desired number of shifts in CL

SAL BP, CL shifts word in BP left CL bit positions. 0 in LSBs

→ SAL BYTE PTR[BX], 1 shift bytes in DX · at offset [BX] 1 bit position left, 0 in MSB.

SAR - SAR Destination, Count

This instruction shift each bit in the specified destination some number of bit position to the right. As a bit is shifted out of the msb position, a copy of previous MSB is put in the MSB position. In other word the sign bits 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 from the LSB. Bit shifted into CF previously will be lost

MSB ⟶ MSB ..... ⟶ LSB ⟶ CF

The destination ~~can be~~ operand can be a byte or word. It can be in a register or in a memory locations. If you want to shift the operation by 1 bit position, you can specify this by putting 1 in the count position of the instruction. For shift of more than 1 bit position, load the desire number of shift into the CL register, and put 'CL' in the position of count in the instruction.

The flag are affected by ~~this~~ follow: CF contains the bit most recently shifted in the LSB. After a multi-bit SAR, OF will be 0. SF and ZF will be updated to show the condition of the destination. PF will have meaning only for an 8-bit destination. AF will be undefined after SAR.

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

MOV CL, 02H $load desired number of shift in CL.

SAR WORD PTR [BP], CL Shift word offset [BP] is stack segment right by two bit position, the two MSBs are now copies of

SHR- SHR, Destination, Count

This instruction shift each bits in the specified destination some number of bit position 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. ~~and the CF will be cleared.~~ In the case of multi-bit shifts, CF will contain the most recent bit shifted from LSB. Bits shifted into CF previously will be lost.

0 ⟶ MSB .....> LSB ⟶ CF

The destination can be a byte or word. The destination can be register or memory location. If you want to shift one bit position then you can specified that in the position of count. for multiple bit position shift you have to load the numeb of bit position in to CR register and put "CR" in the position of Count in instruction.

The flag will be changed as follow: CF contain the most recent ~~LSB~~ bit shifted out from LSB. OF will be 1 if two msb both are not 0. for multiple bit shift OF will be meaningless. SF and ZF will be updated to show the condition of the destim. PF will have meaning if operand is a 8-bit destination. AF will be undefined.

→SHR, BP, 1 shift word in BP one bit position right. 0 in MSB.

→mov CL, 03H load desired number of shifts into CL

SHR BYTEPTR [BX] shift byte in DS at offset [BX] 3 bits right, 0s in ~~the~~ 3MSBs.

TRANSFER OF CONTROL INSTRUCTION

JMP (UNCONDITIONAL JUMP TO SPECIFIED DESTINATION)

This instruction will fetch the next instruction from the location specified in the instruction rather from the location after the JMP instruction. If the destination is the same code segment as the JMP instruction, then only the instruction pointer will be changed to the destination location. This is reffered to as near Jump. If the destination for the jump instruction is in a segment other than from the segment where JMP located then both the segment register and the instruction pointer will be changed to get the destination location. This reffered to as a far jump. The JMP instruction doesnot change any flag.

JBE/ JNA (Jump if below or equal or Jump if not Above)

If after a compare or some other instruction which affect the flags, either zero flag or carry flag is 1, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF both 0. the instruction will have no effect on program execution.

```
→CMP  AX, 4321H   (Compare AX - 4321H)
  JBE  NEXT       Jump to label NEXT if AX is below
                  or equal to 4321H

→CMP  AX, 4321H   Compare (AX - 4321H)
  JNA  NEXT       Jump to label next if AX not
                  above 4321H
```

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

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

```
→ CMP BL, 39H        Compare by subtractig 39H from BL
  JG NEXT            Jump label NEXT if BL more positive than 39H
→ CMP BL, 39H        Compare by subtracting 39H from BL
  JNLE NEXT          Jump to label NEXT if BL is not loss than or equal to 39H.
```

JNG JNGE (Jump is less than/ Jump if not greater or equal)

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

```
→ CMP BL, 39H        Compare by subtracting 39H from BL
  JL AGAIN           Jump to label AGAIN if BL is more negetive than 39H
→ CMP BL, 39H        Compare by subtracting 39H from BL
  JNGE AGAIN         Jump to label Again if BL is not greater than or equal to 39H.
```

JLE/JNG (JUMP If Less tem or Jbump if not greater)

This instruction is usually used after the compare instruction. The instruction will cause a Jump to the label if given in the instruction if the zero flag is set or if the sign flag not equal to the OF.

```
→ CMP BL, 39H        Compare by subtracting 39H from BL
  JLE NEXT           Jump to label if BL is more negetive than or equal to 39H.
```

→ CMP BL, 39H Compare by subtracting 39H from BL.
  JNG. NEXT Jump to next label if BL not more positive than 39H.

JE/JZ (Jump if equal or Jump if zero).
This instruction usually used after cmp instruction. If the zero flag is set then this instruction will cause a jump to the given label in the instruction.

→ CMP BX, DX Compare (BX-DX)
  JE DONE Jump to ~~next~~ DONE if BX=DX.

~~→ cmp B~~

→ IN AL, 30H Read data from Port 30H
  SUB AL, 30H subtract value with 30H
  JZ START Jump to label START if result is zero.

JNE/JNZ (Jump not equal / Jump if not zero).
This instruction is usually used after compare instruction. If the zero flag is 0, then the instruction will cause a jump to given label in the instruction.

→ IN AL, 0F8H Read data from port
  CMP AL, 72 compare (AL-72)
  JNE NEXT Jump to label NEXT if AL≠72

→ ADD AX, 0002H add count factor 0002H to AX
  DEC BX decrement BX.
  JMP NEXT Jump to label if BX≠0.

STACK RELATED INSTRUCTIONS

PUSH - PUSH SOURCE

the push instruction decrement the stack pointer by 2 and copies a word from a specic 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. the stack segment register and the SP must be initialized before use of PUSH. PUSH can be used to save data to the stack so that it will be not destroyed by the procedure. this instruction does not affect any flag.

→ PUSH BX Decrement SP by 2, copy BX to stack
→ PUSH DS Decrement SP by 2, copy DS to stack
→ PUSH BL Illegal, must be a word.
→ PUSH TABLE[BX] Decrement SP by 2 and copy word from memory segment DS at EA = TABLE + [BX] to stack.

POP - POP Destination.

the POP instruction copies a word from the stack location pointed to the stack pointer to a destination specified in the instruction. the instruction can be a general-purpose register, segment register or a memory locations. the data in the stack is not changed. after the word copied to the specified destination the stack pointer automatically incremented by 2 to point the next word on the stack. the pop instruction doesnot affect any flag.

POP DX Copy a word from top of stack to DX. SP incremented by 2

POP DS copy a word from top of stack to DS, SP incremented by 2

POP TABLE[BX] Copy a word from top of stack to memory in DS with EA = TABLE + [BX]. incremented SP by 2.

INPUT OUTPUT INSTRUCTIONS

IN - IN ACCUMULATOR, PORT

The IN instruction copies data from port to the AL or AX register. If an 8-bit port is read, the data will be in AL and if a 16-bit port is read data will be in AX.

The IN instruction has two possible formats, fixed port and variable port. for fixed port type, 8bit address of the port specified directly in the instruction. With this form any on 256 possible port can be addressed.

→IN AL, 0C8H INPUT a byte from port 0C8H to AL

→IN AL, 34H INPUT a word from port 34H to AX.

for the variable-port from the IN instruction, the the port address is landed into DX register before the IN Instructor. Since DX is a 16 bit register the port address can be between 0000H and FFFFH. upto 65536 port are available in this mode.

MOV DX, 00FFH initialize Dx to point to port

IN AL, DX Input a byte from 8-bit port 00FF(8bit) to AL.

IN AX, DX Input a word from 16 bit port 00FFH to AX.

The variable-port instruction has advantage that the port address can be computed or dynamically determined in the program. Suppose that an 8086 based computer needs to input data from 10 terminals, each having its own port address. Instead of having a seperate procedure to input data from each port, you can write one generalized input procedure and simply pass the address of the desire port to the procedure in DX.

The IN instruction does not change any flag.

OUT- OUT PORT, Accumulator.

The out ~~port~~ instructions copies a byte from AL or a word from AX to the specified port. The out instruction has two possible forms. fixed port and varriable port.

For the fixed port from the 8-bit address is specified directly in the instruction. With this form any of 256 possible ports can be addressed.

~~for variable~~

→ OUT 3BH, AL Copy the content of AL to port 3BH

→ OUT 2CH, AX copy the content of AX to port 2CH

for varriable port form of the OUT Instructions. The content of AL or AX will be copied to the port at an address contained in DX. therefore the DX register must be loaded with the desired port address before this form of the OUT instruction is used.

→ MOV DX, 0FFF8H Load desired port address in DX

→ OUT DX, AL ~~load~~ copy content of AL to port FFF8H

→ OUT DX, AX copy content of AX to port FFF8H.

The OUT instruction doesnot affect any flag.

8086 ASSEMBLER DIRECTIVES

ENDS (END SEGMENT).

this directives is used with the name of a segment to indicate the end of that logical segment.

→ CODE SEGMENT Start of a Logical segment
  CODE ENDS Containing CODE Instruction.
  End of segment name CODE.

END (END PROCEDURE)

the END directives is put after the last

last segment of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statement after the END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive.

DW (Define Word)

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage location of type word in memory. The statement MULTIPLIER DW 432AH for example declares a ~~location of~~ variable of type word named MULTIPLIER, and initialized with the value 432AH when the program is loaded into the memory to be run.

→ WORDS DW 1234H, 3456H Declare a array of 2 words and initialize them with the specified value.

→ STORAGE DW 100 DUP(0). Reserve a array of 100 words of memory and initialized all 100 words wit 000. the name of array as STORAGE.

→ STORAGE DW 100 DUP(?). Reserve 100 word of storage in memory and give it the name STORAGE but leave the words un-initialized

PROC (PROCEDURE)

The PROC directive is used to identify the start of procedure. The PROC directive follows a name you give the procidure. After the PROC directive the term near or the term far used to specify the type of the procedure. the statement DIVIDE PROC FAR, for example, identifies the start of a procedure named DIVIDE and tell the assembler that the procedure is far (in a segment that different name from the one that containg the instruction which call procedure The PROC directive is used with the ENDP directive. to 'bracket' a procedure.

ENDP (END PROCEDURE)

the directive is used along with the name of the procedure to indicate ~~that~~ end of a procedure to the assembler. the directive, together with the procedure directive, PROC is used to "bracket" a procedure.

```
→SQUARE_ROOT  PROC      start of a procedure
  SQUARE_ROOT  ENDP      End of a procedure
```

LABEL

As an assemblers assembles a section of a data declarations or instructions statements, it use a location counter to be keep track how many bytes it is from the start of a segment at any time. the Label directive is used to give a name

to the current value in the location counter. The label directive must be followed by a form that specifies that the type you want to associate with that name. If the label is going to be used as the destination for a jump or a call, then the label must be specified as a type near or type far. If the label is going to be used to reference data item, then the label must be specified as type byte, type word, or type double word. Here's how we use the LABEL directive for a jump address.

```
→ ENTRY_POINT LABEL FAR      can jump to here from another segment
  NEXT: MOV AL, BL           can not do a far jump directly to a label with colon.
```

the following example shows how we use the label directive to a data reference.

```
→ STACK_SEG SEGMENT STACK
    DW 100 DUP(0)            set aside 100 words for stack
    STACK_TOP LABEL WORD     Give name to the next location after last word in stack.
    STACK_SEG ENDS
```

To initialize stack pointer, use ~~MOV SP,~~

MOV SP, OFFSET STACK_TOP,

INCLUDE (INCLUDE SOURCE CODE FROM FILE)

this directive is used to tell the assembler to insert a block of source code from the named file into the current source module.