Instruction Types

Data transfer instructions

General-purpose data transfer

MOV dst,src (dst)←(src)

Copies the second operand to the first operand.

XCHG dst,src (dst)↔(src)

Exchange bytes or exchange words.

Data transfer with stack

PUSH src Copy specified word to top of stack.

POP dst Copy word from top of stack to specific location.

Flag transfer

PUSHF Copy flag register to top of stack.

POPF Copy word at top of stack to flag register

LAHF Load AH with the low byte of the flag register. No operands

SAHF Store AH register into low 8 bits of Flags register. No operands

Address transfer

LEA reg,src Load effective address of operand in specified register.

Lea SI, X

LDS reg, src Load DS register and other specified register from memory.

LDS SI, Y ,

where Y is dd- double word

LES reg,src Load ES register and other specified register from memory.

I/O port transfer

IN ac, port ; Copy a byte or word from specified port to accumulator (AX or AL).

IN ac, DX

OUT port, ac; Copy a byte or word from accumulator to specified port.

OUT DX, ac

Arithmetic instructions

Arithmetic operations are executed on integer numbers in 4 formats:

- unsigned binary (byte or word) 5h - 0000 0101
- signed binary (byte or word), -5h or 0FAh 1111 1011
- packed decimal (the string of decimal digits are stored in consecutive 4-bit groups : 3251- 0011 0010 0101 0001)
- unpacked decimal (each digit is stored in low 4-bit part of the byte: 3251 - ****0011 ****0010 ****0101 ****0001)
- All arithmetic instructions influence flags that can be checked with conditional transfer instructions.
- Arithmetic operations can use all addressing modes but one operand should be a register.

ADD dst, src, dst←(dst)+(scr). Src can be also immediate value of 8 or 16 bits

ADC dst,src, dst←(dst) + (src)+CF.

SUB dst, src dst←(dst)-(src). Subtract byte from byte or word from word.

SBB dst, src dst← (dst)-(src)-CF It is used in multiple precision operations

INC opr opr←(opr)+1 do not change CF.

DEC opr opr←(opr)-1

NEG opr opr←-(opr). Negate – invert each bit of a specified byte or word and add 1 (form 2's complement).

Ex:

Mov ax, 10H AX= 0010

Neg ax AX=FFF0

CMP opr1, opr2 opr1-opr2. Compare two specified bytes or two specified words and do not keep the result, just for flags (OF, SF, ZF, AF, PF, CF according to result). It is used with conditional jump instructions.

Ex:

MOV AL, 5

MOV BL, 5

CMP AL, BL ; AL = 5, ZF = 1 (so equal!)

JE L1 (JNE L1)

CBW (no opr) (for signed binary) converts byte to word.

If the high digit in AL is 0 then all AH bits are 0, if high bit in AL is 1 then all AH bits are 1.

CWD (no opr) convert word to double word. Works with AX and DX (high word).

Multiplication Instructions

MUL Multiply unsigned byte by byte or unsigned word by word. The product is a word or double word. Cannot use immediate operands.

MUL src . (AX) ←(AL)*(src) for bytes CF and OF =1 if the high byte is not 0.

MUL src (DX:AX) ←(AX)*(src) for words.

IMUL src Multiply signed byte by byte or signed word by word CF and OF =1 if the high byte is not the extension of sign

EX. (AL)=B4 1011 0100cc (11001100)cd -76 (signed) or 180(unsigned) (BL)=11h(17 decimal)

IMUL will form FAF4=-129210 CF=OF=1

MUL will form 0BF4=306010 CF=OF=1

Division Instructions

Division Instructions cannot use immediate operands. After division the quotient and the remainder are obtained. In case of overflow the division is interrupted. CF does not show this.

DIV Divide unsigned word by byte or unsigned double word by word

DIV src *divisor is a byte*

- (AL)) ← quotient (AX)/(src)
- (AH)) ←remainder (AX)/(src)
- *divisor is a word*
- (AX)) ← quotient (DX:AX)/(src)
- (DX)) ←remainder (DX:AX)/(src)

IDIV src Divide signed word by byte or signed double word by word It's the same with DIV

- (AX)=0400 1024_{10}
- (BL)=B4 (-76 or 180)
- DIV BL quotient (AL)=05=5_{10}
  remainder (AH)=7C=124_{10}
- IDIV BL quotient (AL)=F3=-13_{10}
  remainder (AH)=24=36_{10}

Example: Perform double precision addition

```
.model small
.stack 100h
.data
x dd 1111FFFFh
y dd 11115555h
z dw ?
.code
 start: mov ax,@data   ;DS initialisation
        mov ds,ax
       mov ax,X        ; move in AX low word of X
         add ax,Y      ; add ax with low word of Y
         mov Z,ax      ; store the low word of the result
         mov ax, [X+2] ; move in ax high word of X
         adc ax,[Y+2]  ; add with high word of Y and carry
         mov [Z+2],ax ;store the high word of the result z=22235554
end start
```

Packed BCD arithmetic

DAA Decimal adjust After Addition.

DAS Decimal adjust After Subtraction

DAA

Corrects the result of addition of two packed BCD values.

Algorithm:

if low nibble of AL > 9 or AF = 1 then:

AL = AL + 6

AF = 1

if AL > 9Fh or CF = 1 then:

AL = AL + 60h

CF = 1

Example:

```
MOV AL, 0Fh  ; AL = 0Fh (15)
DAA          ; AL = 15h
RET
```

DAS

Corrects the result after subtraction of two packed BCD values.

Algorithm:
if low nibble of AL > 9 or AF = 1 then:

AL = AL - 6

AF = 1

if AL > 9Fh or CF = 1 then:

AL = AL - 60h

CF = 1

Example:

```
MOV AL, 0FFh  ; AL = 0FFh (-1)
DAS           ; AL = 99h, CF = 1
RET
```

Unpacked BCD arithmetic

AAA **-** ASCII (Unpacked) BCD correction after addition

AAS - ASCII (Unpacked) BCD correction after subtraction

AAM - ASCII adjust after multiplication

AAD - ASCII adjust before division

AAA - ASCII (Unpacked) BCD correction after addition

Corrects result in AH and AL after addition when working with CBD values.

if low nibble of AL > 9 or AF = 1 then:

AL = AL + 6

AH = AH + 1

AF = 1

CF = 1

else

AF = 0

CF = 0

in both cases:
clear the high nibble of AL.

Example:

```
MOV AX, 12   ; AH = 00, AL = 0Ch
AAA          ; AH = 01, AL = 02
RET
```

AAS - ASCII (Unpacked) BCD correction after subtraction.

Corrects result in AH and AL after subtraction when working with BCD values.

Algorithm: if low nibble of AL > 9 or AF = 1 then:

AL = AL - 6

AH = AH - 1

AF = 1

CF = 1

else

AF = 0

CF = 0

in both cases:

clear the high nibble of AL.

Example:

```
MOV AX, 02FFh  ; AH = 02, AL = 0FFh
AAS            ; AH = 01, AL = 09
RET
```

AAM - ASCII adjust after multiplication

Corrects the result of multiplication of two BCD values.
Algorithm:

AH = AL / 10

AL = remainder

Example:

```
MOV AL, 15   ; AL = 0Fh
AAM          ; AH = 01, AL = 05
RET
```

AAD - ASCII adjust before division;

Prepares two BCD values for division.
Algorithm:

AL = (AH * 10) + AL

AH = 0

Example:

```
MOV AX, 0105h   ; AH = 01, AL = 05
AAD             ; AH = 00, AL = 0Fh (15)
RET
```

Perform addition of packed BCD numbers (4 decimal digits)

```
.model small
.stack 10h
.data
bcd1 db 56h, 32h
bcd2 db 67h, 49h
bcd3 db ?,?
.code
Start: mov ax,data
        mov ds,ax
        mov al,bcd1
        add al,bcd2
        daa
        mov bcd3,al
        mov al,[bcd1+1]
        adc al,[bcd2+1]
        daa
        mov [bcd3+1],al
end start
```

Perform x/y, where x is a two digit number and x is one digit number represented as unpacked BCD

```
.MODEL  SMALL
 .STACK 10h
 .DATA
  x DB 05h
  y DB 03h,06h;63in unpacked bcd
  q       DB      2 DUP(?)
  r      DB      ?
 .CODE
start: mov  ax,DATA
    mov     ds,ax
    mov     ah,0
    mov     al,y+1
    aad
div     x
    mov     q+1,al
    mov     al,y
    aad
    div     x
    mov     q,al
    mov     r,ah
   mov ax,4c00h
   int 21h
  END     start
```

Program execution transfer instructions

These instructions are used to tell the 8086 to start fetching instructions from some new address, rather than continuing in sequence.

Unconditional transfer instructions

JMP operand , where operand can be a short, near, or far address

- A jump operation reaches a short address by a one-byte offset, limited to a distance of -128 to 127 bytes (the same segment).
- A jump operation reaches near address by a one-word offset, limited to a distance of

  -32,768 to 32767 bytes within the same segment (the same segment).
- A far address may be another segment and is reached by a segment address and offset;

- Address specification:
- a) implicit
- b) using PTR directive:
- JMP SHORT PTR operand
- JMP NEAR PTR operand
- JMP FAR PTR operand

Conditional transfer instructions

- All instructions have the following format: opcode data8
- The first byte is the operation code and the second byte is the 8- bit displacement to the next instruction in 2-s complement system. The negative displacement means go back and positive disp. means go forward. 8-bit displacement constraint the distance of jumping in range of -128…127

Dist in dec	D8 in hex	Address of jumping
-128	80	(IP)-128
0	0	(IP)
127	7F	(IP)+127

These instructions are often used after a compare instruction. The terms B (below) and A (above) refer to unsigned binary numbers. Above means larger in magnitude. The terms G (greater than) or L (less than) refer to signed binary numbers. Greater than means more positive.

instruction	Jump condition	function
JE, JZ	ZF=1	Jump if equal/Jump if zero
JNE, JNZ	ZF=0	Not Zero, Not Equal
JS	SF=1	Sign
JNS	SF=0	Not Sign
JO	OF=1	Overflow
JNO	OF=0	Not Overflow
JP, JPE	PF=1	Parity, Parity Even
JNP, JPO	PF=0	Not Parity, Parity Odd
JB, JNAE, JC	CF=1	Below, Not Above or Equal, Carry
JNB, JAE, JNC	CF = 0	Not Below, Above or Equal, Not Carry
JL, JNGE	SF≠OF	Less, Not Greater or Equal
JLE,JNG	SF≠OF sau ZF=1	Less or Equal, Not Greater
JBE, JNA	CF=1 sau ZF=1	Below or Equal, Not Above
JNL, JGE	SF=OF	Not Less, Greater or Equal
JNLE, JG	SF=OF şi ZF=0	Not Less or Equal, Greater
JNBE, JA	CF=0 şi ZF=0	Not Below or Equal, Above
JNP, JPO	PF=0	Not Parity, Parity Odd

- ; IF ((X > Y) AND (Z < T)) OR (A <> B) THEN C := D;
- ; Test the boolean expression:
- mov ax, A
- cmp ax, B
- jne DoIF
- mov ax, X
- cmp ax, Y
- jng EndOfIf
- mov ax, Z
- cmp ax, T
- jnl EndOfIf
- DoIf: mov ax, D
- mov C, ax
- ; End of IF statement
- EndOfIF:

```
mov    al, 25     ; set al to 25.
mov    bl, 10     ; set bl to 10.

cmp    al, bl     ; compare al - bl.

je     equal      ; jump if al = bl (zf = 1).

mov ah,6
mov dl, 'n'
int 21h
jmp    stop       ; so print 'n', and jump to stop.

equal:            ; if gets here,
mov ah,6
mov dl, 'y'
int 21h

stop:

ret               ; gets here no matter what.
```

Iteration control instructions

These instructions can be used to execute a series of instructions some number of times.

- LOOP opr ;Loop through a sequence of instructions until CX= 0
- LOOPE/LOOPZ opr ; Loop through a sequence instructions while ZF= l and CX $\neq$ 0
- LOOPNE/LOOPNZ opr ;Loop through a sequence instructions while ZF=0 and CX $\neq$ 0
- JCXZ ; Jump to specified address if CX=0

String instructions

A string is a series of bytes or a series of words in sequential memory locations. A string often consists of ASCII character codes. A ”B” in a mnemonic is used to specifically indicate that a string of bytes is to be acted upon. A ”W” in the mnemonic is used to indicate that a string of words is to be acted upon.

Chain instructions MOVS/ MOVSB/ MOVSW

- Copy byte /word from DS:[SI] to ES:[DI]. Update SI and DI.

  ES:[DI] = DS:[SI]

- if DF = 0 then

  SI = SI + 1 (2)

  DI = DI + 1 (2)

- else

  SI = SI - 1 (2)

  DI = DI - 1 (2)

COMPS/ COMPSB/ COMPSW

- Compare bytes/words: ES:[DI] and DS:[SI].

  DS:[SI] - ES:[DI]
- set flags according to result:
  OF, SF, ZF, AF, PF, CF
- if DF = 0 then

  SI = SI + 1 (2)

  DI = DI + 1 (2)
- else

  SI = SI - 1 (2)

  DI = DI - 1 (2)

SCAS/ SCASB/ SCASW

- Compare bytes/words: AL/AX and ES:[DI].
  ES:[DI] – AL/AX
- set flags according to result:
  OF, SF, ZF, AF, PF, CF
- if DF = 0 then

  DI = DI + 1 (2)
- else

  DI = DI - 1 (2)

LODS/ LODSB/ LODSW

- Load byte from DS:[SI] into AL or string word into AX. Update SI.
  AL/AX = DS:[SI]
- if DF = 0 then

  SI = SI + 1 (2)
- else

  SI = SI - 1 (2)

STOS/ STOSB/ STOSW

-
- Store byte from or word from AL/ AX into ES:[DI]. Update DI.

  ES:[DI] = AL/AX
- if DF = 0 then

  DI = DI + 1(2)
- else

  DI = DI - 1 (2)

REP chain instruction

- Repeat following chain instructions: MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times.
  Algorithm:
  check cx: if CX <> 0 then
- do following <u>chain instruction</u>
- CX = CX - 1
- go back to check_cx
- else
- exit from REP cycle
-

REPE/REPZ

- Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal/Zero), maximum CX times.
  Algorithm:
  check_cx: if CX <> 0 then
- do following <u>chain instruction</u>
- CX = CX - 1
- if ZF = 1 then:
  - go back to check_cx
- else
  - exit from REPE/REPZ cycle
- else
- exit from REPE/REPZ cycle

REPNE/REPNZ

- Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Equal/Not Zero), maximum CX times.
  Algorithm:
  check_cx: if CX <> 0 then
- do following chain instruction
- CX = CX - 1
- if ZF = 0 then:
  - go back to check_cx
- else
  - exit from REPNE/REPNZ cycle
- else
- exit from REPNE/REPNZ cycle

XLATB

- Translate byte from table.
  Copy value of memory byte at DS:[BX + unsigned AL] to AL register.
  Algorithm:
  AL = DS:[BX + unsigned AL]
  Example:
- ORG 100h
- x DB 11h, 22h, 33h, 44h, 55h
- LEA BX, x
- MOV AL, 2
- XLATB ; AL = 33h
- RET

Example: Strings

```
DATA      SEGMENT
          a:
      x      DB   0,1,2,3,4,5,6,7,8,9
      y      DB   10 DUP(?)
      z      DB   0,1,2,3,4,0,1,2,3,4
      size   equ ($-a)/3
      DATA      ENDS
      CODE      SEGMENT
            ASSUME    cs:CODE,ds:DATA,es:DATA
      start:    mov  ax,DATA
            mov  ds,ax
            mov  es,ax
            lea si,x      ; offset of x in si
            lea  di,y     ; offset of y in di
```

```
        mov  cx,size
        cld          ;DF=0
        rep movsb     ;move x to y (10 times)
        mov cx,size
l:      lodsb         ; load x into AL while size=0
        loop l
        lea si,x      ; offset of x in si
        lea di,z      ; offset of z in di
        mov cx,size
        repe cmpsw    ; compare x and z while ZF=1
        jnz n         ; jump to interrupt that displays a character 'n'
                      ; on the screen
```

- mov al,'y'   ; else display 'y'
- mov ah,0Eh ; teletype output, in AL the
  ; character to write
- int 10h
- n: mov al,'n'
- mov ah,0Eh ; teletype output, in AL the
  ;character to write
- int 10h
- mov ah,0   ; wait for any key
- int 16h
- CODE   ENDS
- END start

Example: Determine the ASCII code of the hex digit using XLATB

```
DATA  SEGMENT
      asc_tbl DB '0123456789ABCDEF'
      DATA ENDS
      CODE SEGMENT
  ASSUME cs:CODE, ds:DATA
      start:   mov ax,DATA
               mov ds,ax
               mov cx,10h ; counter=16
               xor al,al   ; zeroes al
               mov bx,OFFSET asc_tbl ; bx= 0
      bucl:     mov dh,al  ; remember the address of the first element
               xlatb      ; move in al the content of memory byte
                          ; from [bx+al] -ASCII code of zero
```

- mov dl,al ; store the ASCII code for next interrupt
- mov ah,06h ; direct console input or output.
- ; parameters for output should be in DL = 0..254 (ascii code)
- int 21h ;
- mov al,dh ;restore the address of the previous element
- inc al ; go to next address
- loop bucl ; repeat 16 times
- mov ax,4c00h ;return control to the operating system (stop program).
- int 21h
- CODE ENDS
- END start

Convert a 16-bit binary number to 4 hexadecimal digits and print them to the screen.

- .model small
- .data
- n dw 9A3Ch
- hex db '0123456789ABCDEF' ;the table of hex digits
- .code
- start: mov ax, @data
-         mov ds,ax
-         lea bx,hex
-         mov ah, 02h ; in AH - the code of "show character"
-         mov cx,n
-         mov al,ch
-         and al,0F0h ; the high digit
-         shr al,4
-         xlatb ; translate the digit to a character
-         mov dl,al
-         int 21h   ; show the character
-

- mov al,ch
- and al,0Fh
- xlatb
- mov dl,al
- int 21h
- mov al,cl
- and al,0F0h
- shr al,4
- xlatb
- mov dl,al
- int 21h
- mov al,cl
- and al,0fh
- xlatb
- mov dl,al
- int 21h
- mov ax,4c00h
- int 21h
- end start

PROCEDURES

- Organizing a program into procedures provides the following benefits:

- Reduces the amount of code because a common procedure can be called from any number of places in the code segment.
- Encourages better program organization.
- Facilitates debug in of a program because defects can be more clearly isolated.
- Helps in the ongoing maintenance of programs because procedures are readily identified for modification.

- The basic mechanism for declaring a procedure is:
- *procname proc {NEAR or FAR}*
- *<statements>*
- *procname endp*

The following “procedure” zeros out the 256 bytes starting at the address in the bx register:

- ZeroBytes proc
- xor ax, ax
- mov cx, 128
- ZeroLoop: mov [bx], ax
- add bx, 2
- loop ZeroLoop
- ret
- ZeroBytes endp

CALL and RETn Operations

- The **CALL** instructions provides for the transfer of control to a called procedure. The **RET** returns control back to the calling procedure.
- CALL   procedure-name
- CALL NEAR PTR procedure-name
- CALL FAR PTR procedure-name
- RETN [n]
- RETF [n]
- RET [n]

Near Call and Return

When a near procedure is called:

1. The IP is pushed onto the stack.
2. The IP is loaded with the address of the called procedure.
3. Upon executing the return the IP is popped off the stack.

CALL

$(SP) \leftarrow (SP) - 2$

$SS: ((SP) + 1{:}(SP)) \leftarrow (IP)$

RET

$(IP) \leftarrow SS: ((SP) + 1{:}(SP))$

$(SP) \leftarrow (SP) + 2$

Far Call and Return

1. The CS and IP are pushed onto the stack.
2. The IP and CS of the procedure are placed in the IP and CS registers.
3. Upon executing the return the IP and CS are popped off the stack.

CALL

$(SP) \leftarrow (SP) - 2$

$SS: ((SP) + 1:(SP)) \leftarrow (CS)$

$(SP) \leftarrow (SP) - 2$

$SS: ((SP) + 1:(SP)) \leftarrow (IP)$

RET

$(IP) \leftarrow SS: ((SP) + 1:(SP))$

$(SP) \leftarrow (SP) + 2$

$(CS) \leftarrow SS: ((SP) + 1:(SP))$

$(SP) \leftarrow (SP) + 2$