# 5. 8086 Komut Kümesi

## 5.1. Veri Aktarma Komutları

### 5.1.1. LDS Load Pointer using DS

LDS destination,source

Logic: DS ← (source + 2)
destination ← (source)

LDS loads into two registers the 32-bit pointer variable found in memory at source. LDS stores the segment value (the higher order word of source) in DS and the offset value (the lower order word of source) in the destination register. The destination register may be any any 16-bit general register (that is, all registers except segment registers).

Note: LES, Load Pointer Using ES, is a comparable instruction that loads the ES register rather than the DS register.

### 5.1.2. LES Load Pointer using ES

LES dest-reg,source

Logic: ES ← (source)
dest-reg ← (source + 2)

LES loads into two registers the 32-bit pointer variable found in memory at source. LES stores the segment value (the higher order word of source) in ES and the offset value (the lower order word of source) in the destination register. The destination register may be any any 16-bit general register (that is, all registers except segment registers).

Note: LDS, Load Pointer Using DS, is a comparable instruction that loads the DS register rather than the ES register.

### 5.1.3. LEA Load Effective Address

LEA destination,source

Logic: destination ← Addr(source)

LEA transfers the offset of the source operand (rather than its value) to the destination operand. The source must be a memory reference, and the destination must be a 16-bit general register.

Notes: This instruction has an advantage over using the OFFSET operator with the MOV instruction, in that the source operand can be subscripted. For example, this is legal:

```
LEA  BX, TABLE[SI]              ;Legal
```

the source operand can be subscripted. For example, this is legal:

```
LEA  BX, TABLE[SI]              ;Legal
```

whereas

```
MOV  BX, OFFSET TABLE[SI]       ;Not legal
```

is illegal, since the OFFSET operator performs its calculation at assembly time and this address is not known until run time.

Example : The DOS print string routine, Function 09h, requires a pointer to the string to be printed in DS:DX. If the string you wished to print was at address "PRINT-ME" in the same data segment, you could load DS:DX with the required pointer using this instruction:

```
LEA     DX, PRINT-ME
```

### 5.1.4. XLAT Translate

XLAT translate-table

Logic: AL ← (BX + AL)

XLAT translates bytes via a table lookup. A pointer to a 256-byte translation table is loaded into BX. The byte to be translated is loaded into AL; it serves as an index (ie, offset) into the translation table. After the XLAT instruction is executed, the byte in AL is replaced by the byte located AL bytes from the beginning of the translate-table.

Notes : Translate-table can be less than 256 bytes.
The operand, translate-table, is optional since a pointer to the table must be loaded into BX before the instruction is executed.

The following example translates a decimal value (0 to 15) to the corresponding single hexadecimal digit.

```
        lea     bx, hex_table           ;pointer to table into BX
        mov     al, decimal_digit       ;digit to be translated to AL
        xlat    hex_table               ;translate the value in AL
          .                             ;AL now contains ASCII hex digit
hex_table       db      '0123456789ABCDEF'
```

## 5.2. Aritmetik İşlem Komutları

### 5.2.1. AAA ASCII Adjust after Addition

AAA

```
Logic:  If (AL & 0Fh) > 9 or (AF = 1) then
           AL ← AL + 6
           AH ← AH + 1
           AF ← 1;  CF ← 1
        else
           AF ← 0;  CF ← 0
           AL ← AL & 0Fh
```

Converts the number in the lower 4 bits (nibble) of AL to an unpacked BCD number (high-order nibble of AL is zeroed).
If the lower 4 bits of the number in AL is greater than 9, or the auxiliary carry flag is set, this instruction converts AL to its unpacked BCD form by adding 6 (subtracting 10) to AL; adding 1 to AH; and setting the auxiliary flag and carry flags. This instruction will always leave 0 in the upper nibble of AL.

Note: Unpacked BCD stores one digit per byte; AH contains the most-significant digit and AL the least-significant digit.

### 5.2.2. AAD ASCII Adjust before Division

AAD

```
Logic:  AL ← AH * 10 + AL
        AH ← 0
```

AAD converts the unpacked two-digit BCD number in AX into binary in preparation for a division using DIV or IDIV, which require binary rather than BCD numbers.
AAD modifies the numerator in AL so that the result produced by a division will be a valid unpacked BCD number. For the subsequent DIV to produce the correct result, AH must be 0. After the division, the quotient is returned in AL, and the remainder in AH. Both high-order nibbles are zeroed.

Note: Unpacked BCD stores one digit per byte; AH contains the most-significant digit and AL the least- significant digit.

### 5.2.3. AAM ASCII Adjust after Multiply

AAM

```
Logic:  AH ← AL / 10
        AL ← AL MOD 10
```

This instruction corrects the result of a previous multiplication of two valid unpacked BCD operands. A valid 2-digit unpacked BCD number is taken from AX, the adjustment is performed, and the result is returned to AX. The high-order nibbles of the operands that were multiplied must have been 0 for this instruction to produce a correct result.

Note: Unpacked BCD stores one digit per byte; AH contains the most-significant digit and AL the least-significant digit.

### 5.2.4. AAS ASCII Adjust after Subtraction

AAS

```
Logic:  If (AL & 0Fh) > 9 or AF = 1 then
          AL ← AL - 6
          AH ← AH - 1
          AF ← 1;  CF ← 1
        else
          AF ← 0;  CF ← 0
          AL ← AL & 0Fh
```

AAS corrects the result of a previous subtraction of two valid unpacked BCD operands, changing the content of AL to a valid BCD number. The destination operand of the subtraction must have been specified as AL. The high-order nibble of AL is always set to 0.

Note: Unpacked BCD stores one digit per byte; AH contains the most-significant digit and AL the least- significant digit.

### 5.2.5. CBW Convert Byte to Word

CBW

```
Logic:  if (AL < 80h) then
          AH ← 0
        else
          AH ← FFh
```

CBW extends the sign bit of the AL register into the AH register. This instruction extends a signed byte value into the equivalent signed word value.

Note: This instruction will set AH to 0FFh if the sign bit (bit 7) of AL is set; if bit 7 of AL is not set, AH will be set to 0. The instruction is useful for generating a word from a byte prior to performing byte division.

### 5.2.6. CWD Convert Word to Doubleword

CWD

```
Logic:   if (AX < 8000h) then
           DX ← 0
         else
           DX ← FFFFh
```

CWD extends the sign bit of the AX register into the DX register. This instruction generates the double-word equivalent of the signed number in the AX register.

Note: This instruction will set DX to 0FFFFh if the sign bit (bit 15) of AX is set; if bit 15 of AX is not set, DX will be set to 0.

### 5.2.7. DAA Decimal Adjust after Addition

DAA

```
Logic:   If (AL & 0Fh) > 9 or (AF = 1) then
           AL ← AL + 6
           AF ← 1
         else AF ← 0
         If (AL > 9Fh) or (CF = 1) then
           AL ← AL + 60h
           CF ← 1
         else CF ← 0
```

DAA corrects the result of a previous addition of two valid packed decimal operands (note that this result must be in AL). This instruction changes the content of AL so that it will contain a pair of valid packed decimal digits.

Note: Packed BCD stores one digit per nibble (4 bits); the least significant digit is in the lower nibble. It is not possible to apply an adjustment after division or multiplication of packed BCD numbers. If you need to use multiplication or division, it is better to use unpacked BCD numbers. See, for example, the description of AAM (ASCII Adjust after Multiply).

### 5.2.8. DAS Decimal Adjust after Subtraction

DAS

```
Logic:  If  (AL & 0Fh) > 9 or (AF = 1) then
           AL ← AL - 6
           AF ← 1
        else AF ← 0
        If (AL > 9Fh) or (CF = 1) then
           AL ← AL - 60h
           CF ← 1
        else CF ← 0
```

DAS corrects the result of a previous subtraction of two valid packed decimal operands (note that this result must be in AL). This instruction changes the content of AL so that it will contain a pair of valid packed decimal digits.

Note: Packed BCD stores one digit per nibble (4 bits); the least significant digit is in the lower nibble. It is not possible to apply an adjustment after division or multiplication of packed BCD numbers. If you need to use multiplication and division, it is better to use unpacked BCD numbers. See, for example, the description of AAM (ASCII Adjust after Multiply).

### 5.2.9. DIV Divide, Unsigned

DIV source

```
Logic:  AL ← AX / source          ; Source is byte
        AH ← remainder
   or
        AX ← DX:AX / source       ; Source is word
        DX ← remainder
```

This instruction performs unsigned division. If the source is a byte, DIV divides the word value in AX by source, returning the quotient in AL and the remainder in AH. If the source is a word, DIV divides the double-word value in DX:AX by the source, returning the quotient in AX and the remainder in DX.

Notes: If the result is too large to fit in the destination AL or AX), an INT 0 (Divide by Zero) is generated, and the quotient and remainder are undefined.
When an Interrupt 0 (Divide by Zero) is generated, the saved CS:IP value on the 80286 and 80386 points to the instruction that failed (the DIV instruction). On the 8088/8086, however, CS:IP points to the instruction following the failed DIV instruction.

### 5.2.10. IDIV Integer Divide, Signed

IDIV source

```
Logic:   AL ← AX / source          ; Byte source
         AH ← remainder
   or
         AX ← DX:AX / source       ; Word source
         DX ← remainder
```

IDIV performs signed division. If source is a byte, IDIV divides the word value in AX by source, returning the quotient in AL and the remainder in AH. If source is a word, IDIV divides the double-word value in DX:AX by source, returning the quotient in AX and the remainder in DX.

Notes: If the result is too large to fit in the destination (AL or AX), an INT 0 (Divide by Zero) is generated, and the quotient and remainder are undefined.
The 80286 and 80386 microprocessors are able to the largest negative number (80h or 8000h) as a quotient for this instruction, but the 8088/8086 will generate an Interrupt 0 (Divide by Zero) if this situation occurs.
When an Interrupt 0 (Divide by Zero) is generated, the saved CS:IP value on the 80286 and 80386 points to the instruction that failed (the IDIV instruction). On the 8088/8086, however, CS:IP points to the instruction following the failed IDIV instruction.

### 5.2.11. MUL Multiply, Unsigned

MUL source

```
Logic:   AX ← source * AL          ; if source is a byte
   or
         DX:AX = source * AX       ; if source is a word
```

MUL performs unsigned multiplication. If source is a byte, MUL multiplies source by AL, returning the product in AX. If source is a word, MUL multiplies source by AX, returning the product in DX:AX. The Carry and Overflow flags are set if the upper half of the result (AH for a byte source, DX for a word source) contains any significant digits of the product, otherwise they are cleared.

### 5.2.12. IMUL Integer Multiply, Signed

IMUL source

Logic: AX ← AL * source ; if source is a byte
or
DX:AX ← AX * source ; if source is a word

IMUL performs signed multiplication. If source is a byte, IMUL multiplies source by AL, returning the product in AX. If source is a word, IMUL multiplies source by AX, returning the product in DX:AX. The Carry Flag and Overflow Flag are set if the upper half of the result (AH for a byte source, DX for a word source) contains any significant digits of the product, otherwise they are cleared.

### 5.2.13. CMP Compare

CMP destination,source

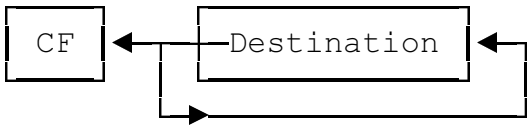
Logic: Flags set according to result of
(destination - source)

CMP compares two numbers by subtracting the source from the destination and updates the flags. CMP does not change the source or destination. The operands may be bytes or words.

## 5.3. Lojik İşlem Komutları

### 5.3.1. ROL Rotate Left

ROL destination,count




### 5.3.2. ROR Rotate Right

ROR destination,count

### 5.3.3. RCL Rotate through Carry Left

RCL destination,count


### 5.3.4. RCR Rotate through Carry Right

RCR destination,count


### 5.3.5. SAL/SHL Shift Arithmetic Left/Shift Logical Left

SAL/SHL destination,count


### 5.3.6. SHR Shift Logical Right

SHR destination,count


### 5.3.7. SAR Shift Arithmetic Right

SAR destination,count

## 5.4. Dizi İşlem Komutları

### 5.4.1. REP Repeat

REP string-instruction

```
Logic:  while CX <> 0                  ;for MOVS, LODS or STOS
          execute string instruction
          CX ← CX - 1
```

---

```
while CX <> 0
   execute string instruction     ;for CMPS or SCAS
   CX ← CX - 1
   if ZF = 0 terminate loop
```

REP is a prefix that may be specified before any string instruction (CMPS, LODS, MOVS, SCAS, and STOS). REP causes the string instruction following it to be repeated, as long as CX does not equal 0; CX is decremented after each execution of the string instruction. (For CMPS and SCAS, REP will also terminate the loop if the Zero Flag is clear after the string instruction executes.)

Notes: If CX is initially 0, the REPeated instruction is skipped entirely.

The test for CX equal to 0 is performed before the instruction is executed. The test for the Zero Flag clear—done only for CMPS and SCAS—is performed after the instruction is executed.

REP, REPE (Repeat While Equal), and REPZ (Repeat While Zero) are all synonyms for the same instruction.

REPNZ (Repeat Not Zero) is similar to REP, but when used with CMPS and SCAS, will terminate with the Zero Flag set, rather than cleared.

REP is generally used with the MOVS (Move String) and STOS (Store String) instructions; it can be thought of as "repeat while not end of string."

Example: The following example moves 100 bytes from BUFFER1 to BUFFER2:

```
CLD                ;Move in the forward direction
LEA   SI, BUFFER1  ;Source pointer to SI
LEA   DI, BUFFER2  ;  ...and destination to DI
MOV   CX,100       ;REP uses CX as the counter
REP   MOVSB        ;  ...and do it
```

### 5.4.2. REPNE Repeat While not Equal

REPNE string-instruction

```
Logic:  while CX <> 0                ;for MOVS, LODS or STOS
          execute string instruction
          CX ← CX - 1
```

---

```
while CX <> 0                ;for CMPS or SCAS
   execute string instruction
   CX ← CX - 1
   if ZF <> 0 terminate loop    ;This is only difference
                                ;between REP and REPNE
```

REPNE is a prefix that may be specified before any string instruction (CMPS, LODS, MOVS, SCAS, and STOS). REPNE causes the string instruction following it to be repeated, as long as CX does not equal 0; CX is decremented after each execution of the string instruction. (For CMPS and SCAS, REP will also terminate the loop if the Zero Flag is set after the string instruction executes. Compare this to REP, which will terminate if the Zero Flag is clear.)

Notes: If CX is initially 0, the REPeated instruction is skipped entirely.

The test for CX equal to 0 is performed before the instruction is executed. The test for the Zero Flag set—done only for CMPS and SCAS—is performed after the instruction is executed.

REPNE and REPNZ are synonyms for the same instruction.

You do not need to initialize ZF before using repeated string instructions.

Example :The following example will find the first byte equal to 'A' in the 100-byte buffer at STRING.

```
      CLD                ;Scan string in forward direction
      MOV    AL,'A'      ;Scan for 'A'
      LEA    DI, STRING  ;Address to start scanning at
      MOV    CX,100      ;Scanning 100 bytes
REPNE SCASB               ;  ...and scan it
      DEC    DI          ;Back up DI to point to the 'A'
```

Upon termination of the repeated SCASB instruction, CX will be equal to zero if a byte value of 'A' was not found in STRING, and non-zero if it was.

### 5.4.3. MOVSB Move String Byte

MOVSB

```
Logic:   (ES:DI) ← (DS:SI)
         if DF = 0
            SI ← SI + 1
            DI ← DI + 1
         else
            SI ← SI - 1
            DI ← DI - 1
```

This instruction copies the byte pointed to by DS:SI into the location pointed to by ES:DI. After the move, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), to point to the next byte.

Example : Assuming BUFFER1 as been defined somewhere as:

```
BUFFFER1     DB     100 DUP (?)
```

the following example moves 100 bytes from BUFFER1 to BUFFER2:

```
     CLD                ;Move in the forward direction
     LEA    SI, BUFFER1 ;Source address to SI
     LEA    DI, BUFFER2 ;Destination address to DI
     MOV    CX,100      ;CX is used by the REP prefix
REP  MOVSB              ;   ...and move it.
```

### 5.4.4. MOVSW Move String Word

MOVSW

```
Logic:   (ES:DI) ← (DS:SI)
         if DF = 0
            SI ← SI + 2
            DI ← DI + 2
         else
            SI ← SI - 2
            DI ← DI - 2
```

This instruction copies the word pointed to by DS:SI to the location pointed to by ES:DI. After the move, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), to point to the next word.

Example : Assuming BUFFER1 as been defined somewhere as:

```
BUFFFER1    DB    100 DUP (?)
```

the following example moves 50 words (100 bytes) from BUFFER1 to BUFFER2:

```
    CLD                 ;Move in the forward direction
    LEA   SI, BUFFER1   ;Source address to SI
    LEA   DI, BUFFER2   ;Destination address to DI
    MOV   CX,50         ;Used by REP; moving 50 words
REP MOVSW               ;   ...and move it.
```

### 5.4.5. CMPS Compare String (Byte or Word)

CMPS destination-string,source-string

```
Logic:  CMP (DS:SI), (ES:DI)      ; Sets flags only
        if DF = 0
           SI ← SI + n            ; n = 1 for byte, 2 for word
           DI ← DI + n
        else
           SI ← SI - n
           DI ← DI - n
```

This instruction compares two values by subtracting the byte or word pointed to by ES:DI, from the byte or word pointed to by DS:SI, and sets the flags according to the results of the comparison. The operands themselves are not altered. After the comparison, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next element of the string.

Note: This instruction is always translated by the assembler into either CMPSB, Compare String Byte, or CMPSW, Compare String Word, depending upon whether source refers to a string of bytes or words. In either case, you must explicitly load the SI and DI registers with the offset of the source and destination strings.

Example:

Assuming the definition:

```
buffer1 db    100 dup (?)
buffer2 db    100 dup (?)
```

the following example compares BUFFER1 against BUFFER2 for the first mismatch.

```
    cld                 ;Scan in the forward direction
    mov   cx, 100       ;Scanning 100 bytes (CX is used by REPE)
    lea   si, buffer1   ;Starting address of first buffer
    lea   di, buffer2   ;Starting address of second buffer
repe  cmps  buffer1,buffer2 ;...and compare it.
    jne   mismatch      ;The Zero Flag will be cleared if there
                        ;is a mismatch
match:      .           ;If we get here, buffers match
            .
mismatch:
    dec   si            ;If we get here, we found a mismatch.
    dec   di            ;Back up SI and DI so they point to the first mismatch
```

Upon exit from the REPE CMPS loop, the Zero Flag will be cleared if a mismatch was found, and set otherwise. If a mismatch was found, DI and SI will be pointing one byte past the byte that didn't match; the DEC DI and DEC SI backup these registers so they point to the mismatched characters.

### 5.4.6. CMPSB Compare String Byte

CMPSB

```
Logic:   CMP (DS:SI), (ES:DI)          ; Sets flags only
         if DF = 0
            SI ← SI + 1
            DI ← DI + 1
         else
            SI ← SI - 1
            DI ← DI - 1
```

This instruction compares two values by subtracting the byte pointed to by ES:DI, from the byte pointed to by DS:SI, and sets the flags according to the results of the comparison. The operands themselves are not altered. After the comparison, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next element of the string.

Example: The following example compares BUFFER1 against BUFFER2 for the first mismatch.

```
        cld                 ;Scan in the forward direction
        mov     cx, 100         ;Scanning 100 bytes (CX is used by REPE)
        lea     si, buffer1     ;Starting address of first buffer
        lea     di, buffer2     ;Starting address of second buffer
        repe    cmpsb           ;   ...and compare it.
        jne     mismatch        ;The Zero Flag will be cleared if there
                                ;   is a mismatch
match:          .               ;If we get here, buffers match
                .
mismatch:
        dec     si              ;If we get here, we found a mismatch.
        dec     di              ;Back up SI and DI so they point to the
                .               ;   first mismatch
```

Upon exit from the REPE CMPSB loop, the Zero Flag will be cleared if a mismatch was found, and set otherwise. If a mismatch was found, DI and SI will be pointing one byte past the byte that didn't match; the DEC DI and DEC SI instructions backup these registers so they point to the mismatched characters.

### 5.4.7. CMPSW Compare String Word

CMPSW

```
Logic:   CMP (DS:SI), (ES:DI)        ; Sets flags only
         if DF = 0
            SI ← SI + 2
            DI ← DI + 2
         else
            SI ← SI - 2
            DI ← DI - 2
```

This instruction compares two numbers by subtracting the word pointed to by ES:DI, from the word pointed to by DS:SI, and sets the flags according to the results of the comparison. The operands themselves are not altered. After the comparison, SI and DI are incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next element of the string.

Example :The following example compares BUFFER1 against BUFFER2 for the first mismatch.

```
        cld                 ;Scan in the forward direction
        mov     cx, 50      ;Scanning 50 words (100 bytes)
        lea     si, buffer1 ;Starting address of first buffer
        lea     di, buffer2 ;Starting address of second buffer
repe    cmpsw               ;   ...and compare it.
        jne     mismatch    ;The Zero Flag will be cleared if there
                            ;   is a mismatch
match:      .               ;If we get here, buffers match
            .
mismatch:
        dec     si          ;If we get here, we found a mismatch.
        dec     si          ;Back up DI and SI so they point to the
        dec     di          ;   first mismatch
        dec     di
```

Upon exit from the REPE CMPSW loop, the Zero Flag will be cleared if a mismatch was found, and set otherwise. If a mismatch was found, DI and SI will be pointing one word (two bytes) past the word that didn't match; the DEC DI and DEC SI pairs backup these registers so they point to the mismatched characters.

### 5.4.8. SCAS Scan String (Byte or Word)

SCAS destination-string

```
Logic:   CMP Accumulator, (ES:DI)     ;Sets flags only
         if DF = 0
            DI ← DI + n              ;n = 1 for byte, 2 for word
         else
            DI ← DI - n
```

This instruction compares the accumulator (AL or AX) to the byte or word pointed to by ES:DI. SCAS sets the flags according to the results of the comparison; the operands themselves are not altered. After the comparison, DI is incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for comparing the next element of the string.

Notes: This instruction is always translated by the assembler into either SCASB, Scan String Byte, or SCASW, Scan String Word, depending upon whether destination-string refers to a string of bytes or words. In either case, however, you must explicitly load the DI register with the offset of the string.

SCAS is useful for searching a block for a given byte or word value. Use CMPS, Compare String, if you wish to compare two strings (or blocks) in memory, element by element.

Example : Assuming the definition:

```
LOST_A   DB   100  DUP(?)
```

the following example searches the 100-byte block starting at LOST_A for the character 'A' (65 decimal).

```
        MOV     AX, DS
        MOV     ES, AX          ;SCAS uses ES:DI, so copy DS to ES
        CLD                     ;Scan in the forward direction
        MOV     AL, 'A'         ;Searching for the lost 'A'
        MOV     CX,100          ;Scanning 100 bytes (CX is used by REPNE)
        LEA     DI, LOST_A      ;Starting address to DI
REPNE   SCAS    LOST_A          ;   ...and scan it.
        JE      FOUND           ;The Zero Flag will be set if we found
                                ;   a match.
NOTFOUND:       .               ;If we get here, no match was found
                .
                .
FOUND:  DEC     DI              ;Back up DI so it points to the first matching 'A'
```

.

Upon exit from the REPNE SCAS loop, the Zero Flag will be set if a match was found, and cleared otherwise. If a match was found, DI will be pointing one byte past the match location; the DEC DI at FOUND takes care of this problem.

### 5.4.9. LODSB Load String Byte

LODSB

Logic: AL ← (DS:SI)
if DF = 0
SI ← SI + 1
else
SI ← SI - 1

LODSB transfers the byte pointed to by DS:SI into AL and increments or decrements SI (depending on the state of the Direction Flag) to point to the next byte of the string.

Note: Although it is legal to repeat this instruction, it is almost never done since doing so would continually overwrite the value in AL.

Example : The following example sends the eight bytes at INIT_PORT to port 250. (Don't try this on your machine, unless you know what's hanging off port 250.)

```
INIT_PORT:
      DB     '$CMD0000'     ;The string we want to send
      .
      .
      CLD                   ;Move forward through string at INIT_PORT
      LEA    SI, INIT_PORT  ;SI gets starting address of string
      MOV    CX, 8          ;CX is counter for LOOP instruction
AGAIN:  LODSB                ;Load a byte into AL...
      OUT    250,AL         ;   ...and output it to the port.
      LOOP   AGAIN
```

### 5.4.10. LODSW Load String Word

LODSW

```
Logic:   AX ← (DS:SI)
         if DF = 0
            SI ← SI + 2
         else
            SI ← SI - 2
```

LODSW transfers the word pointed to by DS:SI into AX and increments or decrements SI (depending on the state of the Direction Flag) to point to the next word of the string.

Note: Although it is legal to repeat this instruction, it is almost never done since doing so would continually overwrite the value in AL.

Example : The following example sends the eight bytes at INIT_PORT to port 250. (Don't try this on your machine, unless you know what's hanging off port 250.)

```
INIT_PORT:
      DB     '$CMD0000'     ;The string we want to send
      .
      .
      CLD                   ;Move forward through string at INIT_PORT
      LEA    SI, INIT_PORT  ;SI gets starting address of string
      MOV    CX, 4          ;Moving 4 words (8 bytes)
AGAIN:  LODSW                ;Load a word into AX...
      OUT    250,AX         ;   ...and output it to the port.
      LOOP   AGAIN
```

### 5.4.11. STOSB Store String Byte

STOSB

```
Logic:   (ES:DI) ← AL
         if DF = 0
            DI ← DI + 1
         else
            DI ← DI - 1
```

STOSB copies the value in AL into the location pointed to by ES:DI. DI is then incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for storing AL in the next location.

Notes: This instruction is always translated by the assembler into either STOSB, Store String Byte, or STOSW, Store String Word, depending upon whether destination-string refers to a string of bytes or words. In either case, however, you must explicitly load the DI register with the offset of the string.

Example : When used in conjunction with the REP prefixes, the Store String instructions are useful for initializing a block of memory. For example, the following code would initialize the 100-byte memory block at BUFFER to 0:

```
MOV    AL,0          ;The value to initialize BUFFER to
LEA    DI,BUFFER     ;Starting location of BUFFER
MOV    CX,100        ;Size of BUFFER
CLD                  ;Let's move in forward direction
REP    STOS BUFFER   ;Compare this line to example for STOSB
```

### 5.4.12. STOSW Store String Word

```
Logic:  (ES:DI) ← AX
        if DF = 0
           DI ← DI + 2
        else
           DI ← DI - 2
```

STOSW copies the value in AX into the location pointed to by ES:DI. DI is then incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for storing AX in the next location.

When used in conjunction with the REP prefixes, the Store String instructions are useful for initializing a block of memory. For

When used in conjunction with the REP prefixes, the Store String instructions are useful for initializing a block of memory. For example, the following code would initialize the 100-byte memory block at BUFFER to 0:

```
    MOV    AX,0          ;The value to initialize BUFFER to
    LEA    DI,BUFFER     ;Starting location of BUFFER
    MOV    CX,50         ;Size of BUFFER, in words
    CLD                  ;Let's move in forward direction
REP STOSW                ;Compare this line to example for STOS
```

## 5.5. Denetim Aktarma Komutları

### 5.5.1. JMP Jump Unconditionally

JMP target

Jump Condition: Jump always

JMP always transfer control to the target location. Unlike CALL, JMP does not save IP, because no RETurn is expected. An intrasegment JMP may be made either through memory or through a 16-bit register; an intersegment JMP can be made only through memory.

Notes: If the assembler can determine that the target of an intrasegment jump is within 127 bytes of the current location, the assembler will automatically generate a short-jump (two-byte) instruction; otherwise, a 3-byte NEAR JMP is generated.

You can force the generation of a short jump by explicit use of the operator "short," as in:

```
JMP  short near_by
```

### 5.5.2. CALL Call Procedure

CALL procedure_name

Logic:
```
if FAR CALL (inter-segment)
   PUSH CS
   CS ← dest_seg
PUSH IP
IP ← dest_offset
```

CALL transfers control to a procedure that can either be within the current segment (a NEAR procedure) or outside it (a FAR procedure). The two types of CALLs result in different machine instructions, and the RET instruction that exits from the procedure must match the type of the CALL instruction (the potential for mismatch exists if the procedure and the CALL are assembled separately).

### 5.5.3. RET Return from Procedure

RET optional-pop-value

Logic: POP IP
If FAR RETURN (inter-segment)
POP CS
SP ← SP + optional-pop-value (if specified)

RET transfers control from a called procedure back to the instruction following the CALL, by:

- Popping the word at the top of the stack into IP
- If the return is an intersegment return:
- Popping the word now at the top of the stack into CS
- Adding the optional-pop-value, if specified, to SP

The assembler will generate an intrasegment RET if the procedure containing the RET was designated by the programmer as NEAR, and an intersegment RET if it was designated FAR. The optional-pop-value specifies a value to be added to SP, which has the effect of popping

### 5.5.4. JA Jump If Above

JA short-label

Jump Condition: Jump if CF = 0 and ZF = 0

Used after a CMP or SUB instruction, JA transfers control to short-label if the first operand (which should be unsigned) was greater than the second operand (also unsigned). The target of the jump must be within -128 to +127 bytes of the next instruction.

Notes: JNBE, Jump Not Below or Equal, is the same instruction as JA.
JA, Jump on Above, should be used when comparing unsigned numbers.
JG, Jump on Greater, should be used when comparing signed numbers.

### 5.5.5. JAE Jump If Above or Equal

JAE short-label

Jump Condition: Jump if CF = 0

Used after a CMP or SUB instruction, JAE transfers control to short-label if the first operand (which should be unsigned) was greater than or equal to the second operand (also unsigned). The target of the jump must be within -128 to +127 bytes of the next instruction.

Notes: JNB (Jump Not Below) is the same instruction as JAE.
JAE, Jump on Above or Equal, should be used when comparing unsigned numbers.
JGE, Jump on Greater or Equal, should be used when comparing signed numbers.

### 5.5.6. JC Jump If Carry

JC short-label

Jump Condition: Jump if CF = 1

JC transfers control to short-label if the Carry Flag is set. The target of the jump must be within -128 to +127 bytes of the next instruction.

Note: JB (Jump if Below), JC, and JNAE (Jump if Not Above or Equal) are all synonyms for the same instruction.
Use JNC, Jump if No Carry, to jump if the carry flag is clear.

### 5.5.7. LOOP Loop on Count

LOOP short-label

```
Logic:   CX ← CX - 1
         If (CX <> 0)
            JMP short-label
```

LOOP decrements CX by 1, then transfers control to short-label if CX is not 0. Short-label must be within -128 to +127 bytes of the next instruction.

### 5.5.8. LOOPE Loop While Equal

LOOPE short-label

```
Logic:   CX ← CX - 1
         If CX <> 0 and ZF = 1
            JMP short-label
```

Used after a CMP or SUB, LOOPE decrements CX by 1, then transfers control to short-label if the first operand of the CMP or SUB is equal to the second operand. Short-label must be within -128 to +127 bytes of the next instruction.

Note: LOOPZ, Loop if Zero, is the same instruction.

### 5.5.9. LOOPNE Loop While not Equal

LOOPNE short-label

```
Logic:   CX ← CX - 1
         If CX <> 0 and ZF = 0
            JMP short-label
```

Used after a CMP or SUB, LOOPNE decrements CX by 1, then transfers control to short-label if the first operand of the CMP or SUB is not equal to the second operand. Short-label must be within -128 to +127 bytes of the next instruction.

Note : LOOPNZ, Loop While Not Zero, is the same instruction.

### 5.5.10. JCXZ Jump If CX Register Zero

JCXZ short-label

Jump Condition: Jump if CX = 0

JCXZ transfers control to short-label if the CX register is 0. The target of the jump must be within -128 to +127 bytes of the next instruction.

Note: This instruction is commonly used at the beginning of a loop to bypass the loop if the counter variable (CX) is at 0.

### 5.5.11. INT Interrupt

INT interrupt-num

```
Logic:   PUSHF              ; Push flags onto stack
         TF ← 0             ; Clear Trap Flag
         IF ← 0             ; Disable Interrupts
         CALL FAR (INT*4)   ; Call the interrupt handler
```

INT pushes the flags register, clears the Trap and Interrupt-enable Flags, pushes CS and IP, then transfers control to the interrupt handler specified by the interrupt-num. If the interrupt handler returns using an IRET instruction, the original flags are restored.

Notes: The flags are stored in the same format as that used by the PUSHF instruction.The address of the interrupt vector is determined by multiplying the interrupt-num by 4. The first word at the resulting address is loaded into IP, and the second word into CS.
All interrupt-nums except type 3 generate a two-byte opcode; type 3 generates a one-byte instruction called the Breakpoint interrupt.

### 5.5.12. IRET Interrupt Return

IRET

```
Logic:  POP IP
        POP CS
        POPF            ; Pop flags from stack
```

IRET returns control from an interrupt routine to the point where the interrupt occurred, by popping IP, CS, and the Flag registers.

### 5.5.13. HARDWARE INTERRUPTS

| | |
|---|---|
| INT 00h (0) | Divide by 0 |
| INT 01h (1) | Single Step |
| INT 02h (2) | Non-Maskable Interrupt (NMI) |
| INT 03h (3) | Breakpoint |
| INT 04h (4) | Overflow |
| INT 05h (5) | Print Screen |
| INT 08h (8) | System Timer |
| INT 09h (9) | Keyboard |
| INT 10h (16) | Video and Screen Services |
| INT 11h (17) | Read Equipment-List |
| INT 12h (18) | Report Memory Size |
| INT 13h (19) | Disk I/O Services, Floppy and Hard Disks |
| INT 14h (20) | Serial I/O Services (Communications Ports) |
| INT 15h (21) | Cassette and Extended Services |
| INT 16h (22) | Keyboard I/O Services |
| INT 17h (23) | Printer I/O Services |
| INT 18h (24) | BASIC Loader Service |
| INT 19h (25) | Bootstrap Loader Service |
| INT 1Ah (26) | System Timer and Clock Services |
| INT 1Bh (27) | Keyboard Break |
| INT 1Ch (28) | User Timer Tick |
| INT 4Ah (74) | User Alarm |
| INT 70h (112) | Real-Time Clock |

Example: INT 05h (5) Print Screen

Prints the current screen contents to the first parallel printer (LPT1).

This interrupt can be called from within a program as well as by means of the Shift-PrtSc key combination. The service returns no register values but sets a status code at memory location 0000:0500h. The possible values of this status code, and their meanings, are as follows:

| | |
|---|---|
| 00h | Print Screen not called, or operation completed |
| 01h | Print Screen currently in progress |
| FFh | Error encountered during most recent Print Screen |

## 5.5.14. DOS INTERRUPTS

| | | |
|---|---|---|
| INT 20h (32) | Terminate Program | |
| INT 21h (33) | DOS Service Calls | |
| INT 22h (34) | Terminate Address | |
| INT 23h (35) | Break Address | |
| INT 24h (36) | Critical-Error Handler Address | |
| INT 25h (37) | Absolute Disk Read | |
| INT 26h (38) | Absolute Disk Write | |
| INT 27h (39) | Terminate and Stay Resident | |
| INT 28h (40) | Reserved | |
| INT 29h (41) | Reserved | |
| INT 2Ah (42) | Reserved | |
| INT 2Bh (43) | Reserved | |
| INT 2Ch (44) | Reserved | |
| INT 2Dh (45) | Reserved | |
| INT 2Eh (46) | Reserved | |
| INT 2Fh... (47) | Multiplex Interrupt | Overview |
| INT 2Fh (47) | Multiplex Interrupt (PRINT) | □ DOS 3.1 |
| INT 2Fh (47) | Multiplex Interrupt (ASSIGN) | □ DOS 3.1 |
| INT 2Fh (47) | Multiplex Interrupt (SHARE) | □ DOS 3.1 |
| INT 2Fh (47) | Multiplex Interrupt (APPEND) | □ DOS 3.1 |
| INT 30h (48) | Reserved | |

Example : INT 25h (37) Absolute Disk Read

Reads one or more sectors on a specified logical disk.

| | | |
|---|---|---|
| On entry: | AL | Drive number (0=A, 1=B) |
| | CX | Number of sectors to read |
| | DX | Starting sector number |
| | DS:DX | Buffer to store sector read |
| Returns: | AX | Error code (if CF is set; see below) |
| | Flags | DOS leaves the flags on the stack |

## 5.5.15. DOS FUNCTIONS

| | |
|---|---|
| 00h (0) | Terminate Program |
| 01h (1) | Read Keyboard Character and Echo |
| 02h (2) | Character Output |
| 03h (3) | Auxiliary Character Input |
| 04h (4) | Auxiliary Character Output |
| 05h (5) | Printer Character Output |
| 06h (6) | Direct Console Character I/O |
| 07h (7) | Direct Console Character Input without Echo |
| 08h (8) | Console Character Input without Echo |
| 09h (9) | Print String |
| 0Ah (10) | Buffered Input |

| | |
|---|---|
| 0Bh (11) | Check Standard Input Status |
| 0Ch (12) | Clear Input Buffer, then Invoke Function |
| 0Dh (13) | Disk Reset |
| 0Eh (14) | Select Default Drive |
| 0Fh (15) | Open File, Using FCBs |
| 10h (16) | Close File, Using FCBs |
| 11h (17) | Search for First Matching File, Using FCBs |
| 12h (18) | Search for Next Matching File, Using FCBs |
| 13h (19) | Delete File, Using FCBs |
| 14h (20) | Sequential Read, Using FCBs |

Example : Function 01h (1) Read Keyboard Character and Echo

Reads a character from the standard input device (usually the keyboard), and echoes it to the standard output device (usually the screen).

On entry: AH 01h

Returns: AL Character read

Extended ASCII For the special keys, such as the cursor and function keys, this function returns a 0 in AL; call the function again to read the extended code of the special character. (See the Key codes for a listing of the extended codes.)

Ctrl-Break & Ctrl-C DOS generates an INT 23h.

Note: This function checks for Ctrl-Break and Ctrl-C. Use Function 07h if you don't wish to check Ctrl-Break and Ctrl-C.

## 5.6. İşlemci Denetim Komutları

### 5.6.1. ESC Escape

ESC coprocessor's-opcode,source

ESC is used to pass control from the microprocessor to a coprocessor, such as an 8087 or 80287. In response to ESC, the microprocessor accesses a memory operand—the instruction for the coprocessor—and places it on the bus. The coprocessor watches for ESC commands and executes the instruction placed on the bus, using the effective address of source.

Notes : In order to synchronize with the math coprocessor, the programmer must explicitly code the WAIT instruction preceding all ESC instructions. The instruction preceding all ESC instructions. The 80286 and 80386 have automatic instruction synchronization, hence WAITs are not needed.

### 5.6.2. HLT Halt the Processor

HLT

This instruction halts the CPU. The processor leaves the halted state in response to a non-maskable interrupt; a maskable interrupt with interrupts enabled; or activation of the reset line.

### 5.6.3. WAIT Wait

Logic: None

WAIT causes the processor to enter a wait state. The processor will remain inactive until the TEST input on the microprocessor is driven active.

Notes : This instruction is used to synchronize external hardware, such as a coprocessor.

### 5.6.4. LOCK Lock the Bus

LOCK

LOCK is a one-byte prefix that can precede any instruction. LOCK causes the processor to assert its bus lock signal while the instruction that follows is executed. If the system is configured such that the LOCK signal is used, it prevents any external device or event from accessing the bus, including interrupts and DMA transfers.

Note: This instruction was provided to support multiple processor systems with shared resources. In such a system, access to those resources is generally controlled via a software-hardware combination using semaphores.

This instruction should only be used to prevent other bus masters from interrupting a data movement operation. This prefix should only be used with XCHG, MOV, and MOVS.