ORG ; FIVE

OTHER COMMANDS
Conditions
Strings
etc.

The x86 PC
assembly language,
design, and interfacing
fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY
**XLAT**

- Adds the contents of AL to BX and uses the resulting offset to point to an entry in an 8 bit translate table.
- This table contains values that are substituted for the original value in AL.
- The byte in the table entry pointed to by BX+AL is moved to AL.

- XLAT [tablename] ; optional because table is assumed at BX

- Table db ‘0123456789ABCDEF’

```assembly
Mov AL,0A; index value
Mov bx,offset table
Xlat; AL=41h, or ‘A’
```
## Data Transfer Instructions - XCHG

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Format</th>
<th>Operation</th>
<th>Flags Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCHG</td>
<td>Exchange</td>
<td>XCHG D,S</td>
<td>(Dest) ↔ (Source)</td>
<td>None</td>
</tr>
</tbody>
</table>

### Destination

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg16</td>
<td>Reg16</td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
</tr>
</tbody>
</table>

Example: XCHG [1234h], BX
## Data Transfer Instructions – LEA, LDS, LES

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Format</th>
<th>Operation</th>
<th>Flags Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>Load Effective Address</td>
<td>LEA Reg16,EA</td>
<td>EA → (Reg16)</td>
<td>None</td>
</tr>
<tr>
<td>LDS</td>
<td>Load Register and DS</td>
<td>LDS Reg16, MEM32</td>
<td>(Mem32) → (Reg16)</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Mem32 + 2) → (DS)</td>
<td></td>
</tr>
<tr>
<td>LES</td>
<td>Load Register and ES</td>
<td>LES Reg16, MEM32</td>
<td>(Mem32) → (Reg16)</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Mem32 + 2) → (ES)</td>
<td></td>
</tr>
</tbody>
</table>
Examples for LEA, LDS, LES

LDS BX, [DI];

DATAX DW 1000H
DATAY DW 5000H
.CODE
LEA SI, DATAX
MOV DI, OFFSET DATAY; THIS IS MORE EFFICIENT
LEA BX,[DI]; IS THE SAME AS…
MOV BX,Di; THIS JUST TAKES LESS CYCLES.
LEA BX,Di; INVALID!

LDS BX, [DI];
Flag Control Instructions

- **LAHF**  Load AH from flags (AH) ← (Flags)
- **SAHF**  Store AH into flags (Flags) ← (AH)
  - Flags affected: SF, ZF, AF, PF, CF
- **CLC**  Clear Carry Flag (CF) ← 0
- **STC**  Set Carry Flag (CF) ← 1
- **CLI**  Clear Interrupt Flag (IF) ← 0
- **STI**  Set interrupt flag (IF) ← 1
- **Example**  (try with debug)
  
  ```asm
  LAHF
  MOV AX,0000
  ADD AX,00
  SAHF
  ```
  - Check the flag changes!

Bulk manipulation of the flags

Individual manipulation of the flags
Jump Instructions

- Unconditional vs conditional jump

(a)

(b)
# Conditional Jump

These flags are based on general comparison

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags/Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ</td>
<td>Jump if ZERO</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if EQUAL</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if NOT ZERO</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if NOT EQUAL</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if CARRY</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if NO CARRY</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump if CX = 0</td>
<td>CX = 0</td>
</tr>
<tr>
<td>JECXZ</td>
<td>Jump if ECX = 0</td>
<td>ECX = 0</td>
</tr>
</tbody>
</table>
## Conditional Jump based on flags

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags/Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JS</td>
<td>JUMP IF SIGN (NEGATIVE)</td>
<td>SF = 1</td>
</tr>
<tr>
<td>JNS</td>
<td>JUMP IF NOT SIGN (POSITIVE)</td>
<td>SF = 0</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if PARITY EVEN</td>
<td>PF = 1</td>
</tr>
<tr>
<td>JNP</td>
<td>Jump if PARITY ODD</td>
<td>PF = 0</td>
</tr>
<tr>
<td>JO</td>
<td>JUMP IF OVERFLOW</td>
<td>OF = 1</td>
</tr>
<tr>
<td>JNO</td>
<td>JUMP IF NO OVERFLOW</td>
<td>OF = 0</td>
</tr>
</tbody>
</table>
## Jump Based on Unsigned Comparison

These flags are based on unsigned comparison

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags/Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if above $op1 &gt; op2$</td>
<td>$CF = 0$ and $ZF = 0$</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not below or equal $op1 \not\leq op2$</td>
<td>$CF = 0$ and $ZF = 0$</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if above or equal $op1 \geq op2$</td>
<td>$CF = 0$</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not below $op1 \not&lt; op2$</td>
<td>$CF = 0$</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below $op1 &lt; op2$</td>
<td>$CF = 1$</td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump if not above nor equal $op1 &lt; op2$</td>
<td>$CF = 1$</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if below or equal $op1 \leq op2$</td>
<td>$CF = 1$ or $ZF = 1$</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not above $op1 \not\leq op2$</td>
<td>$CF = 1$ or $ZF = 1$</td>
</tr>
</tbody>
</table>
# Jump Based on Signed Comparison

These flags are based on signed comparison

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Flags/Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JG</td>
<td>Jump if GREATER op1&gt;op2</td>
<td>SF = OF AND ZF = 0</td>
</tr>
<tr>
<td>JNLE</td>
<td>Jump if not LESS THAN or equal op1&gt;op2</td>
<td>SF = OF AND ZF = 0</td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if GREATER THAN or equal op1&gt;=op2</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not LESS THAN op1&gt;=op2</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if LESS THAN op1&lt;op2</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump if not GREATER THAN nor equal op1&lt;op2</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if LESS THAN or equal op1 &lt;= op2</td>
<td>ZF = 1 OR SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JNG</td>
<td>Jump if NOT GREATER THAN op1 &lt;= op2</td>
<td>ZF = 1 OR SF &lt;&gt; OF</td>
</tr>
</tbody>
</table>
Control Transfer Instructions (conditional)

- It is often necessary to transfer the program execution.
  - **Short**
    - A special form of the direct jump: “short jump”
    - **All conditional jumps are short jumps**
    - Used whenever target address is in range +127 or −128 (single byte)
    - Instead of specifying the address a relative offset is used.
Short Jumps

• Conditional Jump is a two byte instruction.
• In a jump backward the second byte is the 2’s complement of the displacement value.
• To calculate the target the second byte is added to the IP of the instruction after the jump.

Ex:

```
000D ADD AL,[BX]
000F INC BX
0010 DEC CX
0011 JNZ FA
0013
```

Short Jump 0013 + FA (-6) = 0D
.model small
.stack 100h
.data
org 0010
message1 db "You now have a small letter entered!",0dh,0ah,'$'
org 50
message2 db "You have NON small letters ",0dh,0ah,'$'
.code
main proc
    mov ax,@data
    mov ds,ax
    mov ah,00h
    int 16h
    cmp al,61h
    jb next
    cmp al,7Ah
    ja next
    mov ah,09h
    mov dx,offset message1
    mov ah,09h
    int 21h
    int 20h
next: mov dx,offset message2
    mov ah,09h
    int 21h
    mov ax,4c00h
    int 21h
main endp
end main
A Simple Example Program finds the sum

- Write a program that adds 5 bytes of data and saves the result. The data should be the following numbers: 25, 12, 15, 10, 11

```assembly
.model small
.stack 100h
.data
  Data_in DB 25,12,15,10,11
  Sum DB ?
.code
main proc far
  mov ax, @Data
  mov ds,ax
  mov cx,05h
  mov bx,offset data_in
  mov al,0
Again:
  add al,[bx]
  inc bx
  dec cx
  jnz Again
  mov sum,al
  mov ah,4Ch
  INT 21H
Main endp
end main
```
Example Output
Unconditional Jump

- **Short Jump**: `jmp short L1` (8 bit)
- **Near Jump**: `jmp near ptr Label`
  
  If the control is transferred to a memory location within the current code segment (intrasegment), it is NEAR. IP is updated and CS remains the same.
  
  The displacement (16 bit) is added to the IP of the instruction following jump instruction. The displacement can be in the range of −32,768 to 32,768.
  
  The target address can be register indirect, or assigned by the label.

- **Register indirect JMP**: the target address is the contents of two memory locations pointed at by the register.
  
  Ex: `JMP [SI]` will replace the IP with the contents of the memory locations pointed by DS:DI and DS:DI+1 or `JMP [BP + SI + 1000]` in SS

- **Far Jump**: If the control is transferred to a memory location outside the current segment. Control is passing outside the current segment both CS and IP have to be updated to the new values. ex: `JMP FAR PTR label = EA 00 10 00 20`
  `jmp far ptr Label` ; this is a jump out of the current segment.
Near Jump

Jumps to the specified IP with +/- 32K distance from the next instruction following the jmp instruction
Far Jump

Jumps to the specified CS:IP

0B20:1000  jmp  3000:1200
0B20:1005  -u  1000
0B20:1000  EA00120030  JMP  3000:1200
0B20:1005  FF750B  PUSH  [DI+0B]
Nested Loops

**Single Loop**

```
MOV CX,A
BACK: ...
...
...
LOOP BACK
```

**Nested Loops**

```
OUTER:  MOV CX,A
PUSH CX  
MOV CX, 99
INNER:  NOP
...
...
...
LOOP INNER
POP CX
LOOP OUTER
```

```
MOV CX,0
DLOOP: JCXZ SKIP ; guarding
BACK: MUL AX,2H
ADD AX,05H
LOOP BACK
SKIP: INC AX; if CX=0
```

How many times will the loop execute, if JCXZ wasn’t there?
## Loop and Loop Handling Instructions

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

**Figure 6-28 Loop instructions.**
The x86 PC
Assembly Language, Design, and Interfacing
By Muhammad Ali Mazidi, Janice Gillespie Mazidi and Danny Causey

Loop

NEXT:

MOV CX,COUNT
...
...
...
Body of routine that is repeated
...
...
...

LOAD NEXT

Loop back to label NEXT if count not zero

(a)

MOV AX,DATASEGADDR
MOV DS,AX
MOV SI,BLK1ADDR
MOV DI,BLK2ADDR
MOV CX,N

NXTPT:

MOV AH,[SI]
MOV [DI],AH
INC SI
INC DI
LOOP NXTPT
HLT

(b)
3.4: BCD AND ASCII CONVERSION

BCD Number System

- BCD stands for binary coded decimal.
  - Needed because we use the digits 0 to 9 for numbers in everyday life.
  - Computer literature features two terms for BCD numbers:
    - Unpacked BCD.
    - Packed BCD.

<table>
<thead>
<tr>
<th>Digit</th>
<th>BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
</tbody>
</table>
3.4: BCD AND ASCII CONVERSION

BCD unpacked vs. packed

• In unpacked BCD, the lower 4 bits of the number represent the BCD number.
  – The rest of the bits are 0.
    • "0000 1001" and "0000 0101" are unpacked BCD for 9 & 5.
  – Unpacked BCD it takes 1 byte of memory location.
    • Or a register of 8 bits to contain the number.

• In packed BCD, a single byte has two BCD numbers.
  – One in the lower 4 bits; One in the upper 4 bits.
    • "0101 1001" is packed BCD for 59.
  – As it takes only 1 byte of memory to store the packed BCD operands, it is twice as efficient in storing data.
3.4: BCD AND ASCII CONVERSION

ASCII numbers

- In ASCII keyboards, when key "0" is activated "011 0000" (30H) is provided to the computer.
  - 31H (011 0001) is provided for key "1", etc.

<table>
<thead>
<tr>
<th>Key</th>
<th>ASCII (hex)</th>
<th>Binary</th>
<th>BCD (unpacked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
<td>011 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>011 0001</td>
<td>0000 0001</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>011 0010</td>
<td>0000 0010</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>011 0011</td>
<td>0000 0011</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>011 0100</td>
<td>0000 0100</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>011 0101</td>
<td>0000 0101</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>011 0110</td>
<td>0000 0110</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td>011 0111</td>
<td>0000 0111</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>011 1000</td>
<td>0000 1000</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td>011 1001</td>
<td>0000 1001</td>
</tr>
</tbody>
</table>

- To convert ASCII data to BCD, removed the tagged "011" in the higher 4 bits of the ASCII.
  - Each ASCII number is ANDed with "0000 1111". (0FH)
Programs 3-5a, 3-5b, and 3-5c show three methods for converting the 10 ASCII digits to unpacked BCD.

- Using this data segment:

```
  ASC DB '9562481273'
  ORG 0010H
  UNPACK DB 10 DUP(?)
```

The data is defined as DB, a byte definition directive, and is accessed in word-sized chunks.

```
MOV CX, 5
MOV BX, OFFSET ASC ;BX points to ASCII data
MOV DI, OFFSET UNPACK ;DI points to unpacked BCD data
AGAIN:
  MOV AX, [BX] ;move next 2 ASCII numbers to AX
  AND AX, 0FOFH ;remove ASCII 3s
  MOV [DI], AX ;store unpacked BCD
  ADD DI, 2 ;point to next unpacked BCD data
  ADD BX, 2 ;point to next ASCII data
  LOOP AGAIN
```

Program 3-5a
3.4: BCD AND ASCII CONVERSION
ASCII to unpacked BCD conversion

• Programs 3-5a, 3-5b, and 3-5c show three methods for converting the 10 ASCII digits to unpacked BCD.
  – Using this data segment:

<table>
<thead>
<tr>
<th>ASC</th>
<th>DB</th>
<th>'9562481273'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORG</td>
<td>0010H</td>
<td></td>
</tr>
<tr>
<td>UNPACK</td>
<td>DB</td>
<td>10 DUP(?)</td>
</tr>
</tbody>
</table>

Using the PTR directive as shown, makes the code more readable for programmers.

Program 3-5b

```
AGAIN:
    MOV    AX, WORD PTR [BX]  ; move next 2 ASCII numbers to AX
    AND    AX, OF0FH          ; remove ASCII 3s
    MOV    WORD PTR [DI], AX  ; store unpacked BCD
    ADD    DI, 2              ; point to next unpacked BCD data
    ADD    BX, 2              ; point to next ASCII data
    LOOP   AGAIN
MOV    CX, 5                ; CX is loop counter
MOV    BX, OFFSET ASC       ; BX points to ASCII data
MOV    DI, OFFSET UNPACK    ; DI points to unpacked BCD data
```
3.4: BCD AND ASCII CONVERSION
ASCII to unpacked BCD conversion

- Programs 3-5a, 3-5b, and 3-5c show three methods for converting the 10 ASCII digits to unpacked BCD.

  - Using this data segment:

    | ASC  | DB | '9562481273' |
    |------|----|--------------|
    | ORG  | 0010H |
    | UNPACK | DB | 10 DUP (?) |

    3-5c uses based addressing mode since BX+ASC is used as a pointer.

    ```asm
    MOV CX, 10 ; load the counter
    SUB BX, BX ; clear BX
    AGAIN:
    MOV AL, ASC[ BX ] ; move to AL content of mem [ BX+ASC ]
    AND AL, 0FH ; mask the upper nibble
    MOV UNPACK[ BX ], AL ; move to mem [ BX+UNPACK ] the AL
    INC BX ; point to next byte
    LOOP AGAIN ; loop until it is finished
    ```

    Program 3-5c
3.4: BCD AND ASCII CONVERSION
ASCII/BCD conversions

- To convert ASCII to packed BCD, it is converted to unpacked BCD (eliminating the 3), then combined to make packed BCD.
- To convert packed BCD to ASCII, it must first be converted to unpacked.
  - The unpacked BCD is tagged with 011 0000 (30H).
3.4: BCD AND ASCII CONVERSION
ASCII to packed BCD conversion

• For 4 & 7, the keyboard gives 34 & 37, respectively.
  – The goal is to produce packed BCD 47H or “0100 0111“.

<table>
<thead>
<tr>
<th>Key</th>
<th>ASCII</th>
<th>Unpacked BCD</th>
<th>Packed BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>34</td>
<td>00000100</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td>00000111</td>
<td>01000111 or 47H</td>
</tr>
</tbody>
</table>

ORG 0010H
VAL_ASC DB '47'
VAL_BCD DB ?
; reminder: DB will put 34 in 0010H location and 37 in 0011H
MOV AX, WORD PTR VAL_ASC ;AH=37, AL=34
AND AX, 0F0Fh ;mask 3 to get unpacked BCD
XCHG AH, AL ;swap AH and AL.
MOV CL, 4 ;CL=04 to shift 4 times
SHL AH, CL ;shift left AH to get AH=40H
OR AL, AH ;OR them to get packed BCD
MOV VAL_BCD, AL ;save the result
3.4: BCD AND ASCII CONVERSION

packed BCD to ASCII conversion

- Converting from packed BCD to ASCII.

<table>
<thead>
<tr>
<th>Packed BCD</th>
<th>Unpacked BCD</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>29H</td>
<td>02H &amp; 09H</td>
<td>32H &amp; 39H</td>
</tr>
<tr>
<td>0010 1001</td>
<td>0000 0010 &amp; 0000 1001</td>
<td>011 0010 &amp; 011 1001</td>
</tr>
</tbody>
</table>

VAL1_BCD    DB 29H
VAL3_ASC    DW ?

... MOV AL,VAL1_BCD
MOV AH,AL ; copy AL to AH. now AH=29, AL=29H
AND AX,0F00FH ; mask 9 from AH and 2 from AL
MOV CL,4 ; CL=04 for shift
SHR AH,CL ; shift right AH to get unpacked BCD
OR AX,3030H ; combine with 30 to get ASCII
XCHG AH,AL ; swap for ASCII storage convention
MOV VAL3_ASC,AX ; store the ASCII
Ex. ASCII CODE 0-9 = 30h → 39h
MOV AX, 38H ;(ASCII code for number 8)
ADD AL, 39H ;(ASCII code for number 9)
AAA; used for addition AX has → 0107
ADD AX, 3030H; change answer to ASCII if you needed
3.4: BCD AND ASCII CONVERSION

BCD addition and subtraction

• After adding packed BCD numbers, the result is no longer BCD.

```assembly
MOV AL, 17H
ADD AL, 28H
```

Adding them gives 0011 1111B (3FH). (not BCD)

– The result should have been 17 + 28 = 45 (0100 0101).
  • To correct, add 6 (0110) to the low digit: 3F + 06 = 45H.
  – The same could have happened in the upper digit.
    • This problem is so pervasive that the vast majority of microprocessors have an instruction to deal with it.
3.4: BCD AND ASCII CONVERSION

DAA

- DAA (decimal adjust for addition) is provided in the x86 for correcting the BCD addition problem.
  - DAA will add 6 to the lower, or higher nibble if needed
  - Otherwise, it will leave the result alone.

```
DATA1 DB 47H
DATA2 DB 25H
DATA3 DB ?

MOV AL,DATA1 ;AL holds first BCD operand
MOV BL,DATA2 ;BL holds second BCD operand
ADD AL,BL ;BCD addition
DAA ;adjust for BCD addition
MOV DATA3,AL ;store result in correct BCD form
```

After execution, DATA3 will contain 72H.
3.4: BCD AND ASCII CONVERSION

DAA general rules & summary

• General rules for DAA:
  – The source can be an operand of any addressing mode.
    • The destination must be AL in order for DAA to work.
  – DAA must be used after the addition of BCD operands.
    • BCD operands can never have any digit greater than 9.
  – DAA works only after an ADD instruction.
    • It will not work after the INC instruction.

• After an ADD or ADC instruction:
  – If the lower nibble (4 bits) is greater than 9, or if AF = 1.
    • Add 0110 to the lower 4 bits.
  – If the upper nibble is greater than 9, or if CF = 1.
    • Add 0110 to the upper nibble.
3.4: BCD AND ASCII CONVERSION

DAA summary of action

Use of DAA after adding multibyte packed BCD numbers.

Two sets of ASCII data have come in from the keyboard. Write and run a program to:
1. Convert from ASCII to packed BCD.
2. Add the multibyte packed BCD and save it.
3. Convert the packed BCD result to ASCII.

```
TITLE PROG3-6 (EXE) ASCII TO BCD CONVERSION AND ADDITION
PAGE 60,132
.MODE SMALL
.STACK 64
;-----------------------
    .DATA
DATA1_ASC DB '0649147816'
    ORG 0010H
DATA2_ASC DB '0072687188'
    ORG 0020H
DATA3_BCD DB 5 DUP (?)
    ORG 0028H
DATA4_BCD DB 5 DUP (?)
    ORG 0030H
```

See the entire program listing on pages 116-117 of your textbook.
DAA Example

Ex. 4 AL contains 25 (packed BCD)
BL contains 56 (packed BCD)

ADD AL, BL
DAA

25
56
+ -------
7B → 81
Example

- Write an 8086 program that adds two packed BCD numbers input from the keyboard and computes and displays the result on the system video monitor
- Data should be in the form 64+89= The answer 153 should appear in the next line.
Example Continued

Mov dx, offset bufferaddress
Mov ah,0a
Mov si,dx
Mov byte ptr [si], 6
Int 21
Mov ah,0eh
Mov al,0ah
Int 10
; BIOS service 0e line feed position cursor

Mov cl,4
Rol byte ptr [si+3],cl
Rol byte ptr [si+6],cl
Ror word ptr [si+5], cl
Ror word ptr [si+2], cl

Mov al, [si+3]
Add al, [si+6]
DAA
Mov bh,al
Jnc display
Mov al,1
Call display
Mov al,bh
Call display
Int 20

sub byte ptr[si+2], 30h
sub byte ptr[si+3], 30h
sub byte ptr[si+5], 30h
sub byte ptr[si+6], 30h
3.4: BCD AND ASCII CONVERSION
BCD subtraction and correction

• DAS (decimal adjust for subtraction) is provided in the x86 for correcting the BCD subtraction problem.
  – When subtracting packed BCD (single-byte or multibyte) operands, the DAS instruction is used after SUB or SBB.
    • AL must be used as the destination register.

• After a SUB or SBB instruction:
  – If the lower nibble is greater than 9, or if AF = 1.
    • Subtract 0110 from the lower 4 bits.
  – If the upper nibble is greater than 9, or CF = 1.
    • Subtract 0110 from the upper nibble.
3.4: BCD AND ASCII CONVERSION

BCD subtraction and correction

- Due to the widespread use of BCD numbers, a specific data directive, DT, has been created.
  - To represent BCD numbers 0 to 1020 - 1. (twenty 9s)

<table>
<thead>
<tr>
<th>BUDGET</th>
<th>DT</th>
<th>87965141012</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPENSES</td>
<td>DT</td>
<td>31610640392</td>
</tr>
<tr>
<td>BALANCE</td>
<td>DT</td>
<td>?</td>
</tr>
</tbody>
</table>

; balance = budget - expenses

MOV CX,10 ;counter=10
MOV BX,00 ;pointer=0
CLC ;clear carry for the 1st iteration
BACK:
  MOVAL, BYTE PTR BUDGET[ BX ] ;get a byte of the BUDGET
  SBB AL, BYTE PTR EXPENSES[ BX ] ;subtract a byte from it
  DAS ;correct result for BCD
  MOV BYTE PTR BALANCE[ BX ], AL ;save it in BALANCE
  INC BX ;increment for the next byte
  LOOP BACK ;continue until CX=0
BCD and ASCII Numbers

- BCD (Binary Coded Decimal)
  - Unpacked BCD: One byte per digit
  - Packed BCD: 4 bits per digit (more efficient in storing data)
- ASCII to unpacked BCD conversion
  - Keyboards, printers, and monitors all use ASCII.
  - Digits 0 to 9 are represented by ASCII codes 30 – 39.

- **Example.** Write an 8086 program that displays the packed BCD number in register AL on the system video monitor
  - The first number to be displayed should be the MS Nibble
  - It is found by masking the LS Nibble and then rotating the MS Nibble into the LSD position
  - The result is then converted to ASCII by adding 30h
  - The BIOS video service is then called to display this result.
ASCII Numbers Example

MOV BL, AL; save
AND AL, F0H
MOV CL, 4
ROR AL, CL
ADD AL, 30H
MOV AH, 0EH
INT 10H ; display single character

MOV AL, BL; use again
AND AL, 0FH
ADD AL, 30H
INT 10H
INT 20H ; RETURN TO DOS
80x86 is equipped with special instructions to handle string operations

**String**: A series of data words (or bytes) that reside in consecutive memory locations

**Operations**: move, scan, compare

**String Instruction:**
- Byte transfer, SI or DI increment or decrement by 1
- Word transfer, SI or DI increment or decrement by 2
- DWord transfer, SI or DI increment or decrement by 4
String Instructions - D Flag

The Direction Flag: Selects the auto increment $D=0$ or the auto decrement $D=1$ operation for the DI and SI registers during string operations. $D$ is used only with strings.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Format</th>
<th>Operation</th>
<th>Flags Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLD</td>
<td>Clear DF</td>
<td>CLD</td>
<td>($DF) \leftarrow 0$</td>
<td>DF</td>
</tr>
<tr>
<td>STD</td>
<td>Set DF</td>
<td>STD</td>
<td>($DF) \leftarrow 1$</td>
<td>DF</td>
</tr>
</tbody>
</table>

CLD $\rightarrow$ Clears the D flag / STD $\rightarrow$ Sets the D flag
## String Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Format</th>
<th>Operation</th>
<th>Flags Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVNS</td>
<td>Move string</td>
<td>MOVS/MOVSW</td>
<td>((ES)0 + (DI)) ← ((DS)0 + (SI)) &lt;br&gt;(SI) ← (SI) ± 1 or 2 &lt;br&gt;(DI) ← (DI) ± 1 or 2</td>
<td>None</td>
</tr>
<tr>
<td>CMPS</td>
<td>Compare string</td>
<td>CMPSB/CPMSW</td>
<td>Set flags as per &lt;br&gt;((DS)0 + (SI)) − ((ES)0 + (DI)) &lt;br&gt;(SI) ← (SI) ± 1 or 2 &lt;br&gt;(DI) ← (DI) ± 1 or 2</td>
<td>CF, PF, AF, ZF, SF, OF</td>
</tr>
<tr>
<td>SCAS</td>
<td>Scan string</td>
<td>SCASB/SCASW</td>
<td>Set flags as per &lt;br&gt;(AL or AX) − ((ES)0 + (DI)) &lt;br&gt;(DI) ← (DI) ± 1 or 2</td>
<td>CF, PF, AF, ZF, SF, OF</td>
</tr>
<tr>
<td>LODS</td>
<td>Load string</td>
<td>LODSB/LODSW</td>
<td>(AL or AX) ← ((ES)0 + (SI)) &lt;br&gt;(SI) ← (SI) ± 1 or 2</td>
<td>None</td>
</tr>
<tr>
<td>STOS</td>
<td>Store string</td>
<td>STOSB/STOSW</td>
<td>((ES)0 + (DI)) ← (AL or AX) ± 1 or 2 &lt;br&gt;(DI) ← (DI) ± 1 or 2</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>AX,DATASEGADDR</td>
</tr>
<tr>
<td>MOV</td>
<td>DS,AX</td>
</tr>
<tr>
<td>MOV</td>
<td>ES,AX</td>
</tr>
<tr>
<td>MOV</td>
<td>SI,BLK1ADDR</td>
</tr>
<tr>
<td>MOV</td>
<td>DI,BLK2ADDR</td>
</tr>
<tr>
<td>MOV</td>
<td>CX, N</td>
</tr>
<tr>
<td>CLD</td>
<td></td>
</tr>
<tr>
<td>MOVSB</td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>NXTPT</td>
</tr>
<tr>
<td>HLT</td>
<td></td>
</tr>
</tbody>
</table>
Basic string operations must be repeated in order to process arrays of data; this is done by inserting a repeat prefix.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Used with:</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>MOVS, STOS</td>
<td>Repeat while not end of string CX ≠ 0</td>
</tr>
<tr>
<td>REPE/REPZ</td>
<td>CMPS, SCAS</td>
<td>Repeat while not end of string and strings are equal CX ≠ 0 and ZF = 1</td>
</tr>
<tr>
<td>REPNE/REPNZ</td>
<td>CMPS, SCAS</td>
<td>Repeat while not end of string and strings are not equal CX ≠ 0 and ZF = 0</td>
</tr>
</tbody>
</table>

Figure 6-36 Prefixes for use with the basic string operations.
Example. Find and replace

- Write a program that scans the name “Mr.Gohns” and replaces the “G” with the letter “J”.

```
Data1 db  'Mr.Gones','$'
.code
mov es,ds
cld ;set auto increment bit D=0
mov di, offset data1
mov cx,09; number of chars to be scanned
mov al,'G'; char to be compared against
repne SCASB; start scan AL =? ES[DI]
jne Over; if Z=0
dec di; Z=1
mov byte ptr[di], 'J'
Over:  mov ah,09
       mov dx,offset data1
       int 21h; display the resulting String
```
Strings into Video Buffer

Fill the Video Screen with a value

```
CLD
MOV AX, 0B800H
MOV ES, AX
MOV DI, 0
MOV CX, 2000H
MOV AL, 20H
REP STOSW
```
Example. Display the ROM BIOS Date

- Write an 8086 program that searches the BIOS ROM for its creation date and displays that date on the monitor.
- If a date cannot be found display the message “date not found”
- Typically the BIOS ROM date is stored in the form xx/xx/xx beginning at system address F000:FFF5
- Each character is in ASCII form and the entire string is terminated with the null character (00)
- Add a ‘$’ character to the end of the string and make it ready for DOS function 09, INT 21