Arithmetic and Logic Instructions
EXAMPLE: ADD / ADC (ver. 1)

Write a program to calculate the total sum of 5 bytes of data. Each byte represents the daily wages of a worker. This person does not make more than $255 (FFH) a day. The decimal data is as follows: 125, 235, 197, 91, and 48

COUNT EQU 05
DATA DB 125, 235, 197, 91, 48
SUM DB ?

MOV CX, COUNT
MOV SI, OFFSET DATA
MOV AL, 00
BACK: ADD AL, [SI]
INC SI
DEC CX
JNZ BACK
MOV SUM, AL
MOV AH, 4CH
INT 21H
EXAMPLE: ADD / ADC (ver. 2)

COUNT    EQU  05
DATA     DB  125, 235, 197, 91, 48
SUM      DW  ?

MOV  CX, COUNT
MOV  SI, OFFSET DATA
MOV  AX, 00
BACK:  ADD  AL, [SI]
       JNC  OVER
       INC  AH
OVER:  INC  SI
       DEC  CX
       JNZ  BACK
       MOV  SUM, AX
       MOV  AH, 4CH
       INT  21H
EXAMPLE: ADD / ADC (ver. 3)

COUNT EQU 05
DATA DB 125, 235, 197, 91, 48
SUM DW ?

... ... ... ... ...
MOV CX, COUNT
MOV SI, OFFSET DATA
MOV AX, 00
CLC
BACK: ADD AL, [SI]
ADC AH, 00
INC SI
DEC CX
JNZ BACK
MOV SUM, AX
MOV AH, 4CH
INT 21H

Why?
EXAMPLE: ADD / ADC (ver. 4)

COUNT EQU 05
DATA DB 125, 235, 197, 91, 48
SUM DW ?

... ... ... ... ...
MOV CX, COUNT
MOV SI, OFFSET DATA
MOV AX, 00
MOV DX, 00
BACK: MOV DL, [SI]
ADD AX, DX
INC SI
LOOP BACK
MOV SUM, AX
MOV AH, 4CH
INT 21H
EXAMPLE: Addition of words

Write a program to calculate the total sum of five words of data. Each data value represents the yearly wages of a worker. This person does not make more than $65,555 (FFFFH) a year. The decimal data: 27345, 28521, 29533, 30105, and 32375.

COUNT EQU 05
DATA DW 27345, 28521, 29533, 30105, 32375
SUM DW 2 DUP(?)

MOV CX, COUNT
MOV SI, OFFSET DATA
MOV AX,00
MOV BX,AX

BACK: ADD AX, [SI]
ADC BX,00
INC SI
INC SI
DEC CX
JNZ BACK
MOV SUM,AX
MOV SUM+2,BX
MOV AH,4CH
INT 21H
EXAMPLE: Addition of multiword Numbers

Write a program that adds the following two multiword numbers and saves the result: DATA1 = 54FB963CE7H and DATA2 = 3FCD4FA23B8DH.

DATA1   DQ 54FB963CE7H
DATA1   DQ 3FCD4FA23B8D
SUM      DQ ?

... ... ... ... ...
MOV SI, OFFSET DATA1
MOV DI, OFFSET DATA2
MOV BX, OFFSET SUM
MOV CX, 4
CLC

BACK:   MOV AX, [SI]
        ADC AX, [DI]
        MOV [BX], AX
        INC SI
        INC SI
        INC DI
        INC SI
        INC DI
        INC BX
        INC BX
        LOOP BACK
        MOV AH, 4CH
        INT 21H
Subtraction of Unsigned Numbers

<table>
<thead>
<tr>
<th>SUB destination, source ; destination = destination - source</th>
</tr>
</thead>
<tbody>
<tr>
<td>2's complement is used in subtraction.</td>
</tr>
<tr>
<td>The 80x86 uses internal adder circuitry to perform the subtraction command.</td>
</tr>
<tr>
<td>The steps of the hardware in executing the SUB instruction for unsigned numbers, as follows.</td>
</tr>
<tr>
<td>1. Take the 2's complement of the source operand.</td>
</tr>
<tr>
<td>2. Add it to the destination operand.</td>
</tr>
<tr>
<td>3. Invert the carry.</td>
</tr>
</tbody>
</table>

Example: Show the steps involved in the following:

| MOV AL, 3FH ; AL = 3FH = 0011 1111 |
| MOV BH, 23H ; BH = 23H = 00100011 |
| SUB AL, BH ; AL = 0011 1111 - 00100011 = 0011 1111 + |
|                                       | 1101 1101 |
|                                       | 1 0001 1100 |

CF = 0, ZF = 0, AF = 0, PF = 0, and SF = 0.

The programmer must look at the carry flag (not the sign flag) to determine if the result is positive (CF = 0) or negative (CF = 1).
Example: Analyze the following program

- From the data segment:
  
  DATA1 DB 4CH
  DATA2 DB 6EH
  DATA3 DB ?

- From the code segment:

  MOV DH, DATA1
  SUB DH, DATA2
  JNC NEXT
  NOT DH
  INC DH
  
  NEXT: MOV DATA3, DH

Solution:

    DH = 4C \rightarrow 01001100 \rightarrow 0100 1100
    -6E \rightarrow -01101110 \rightarrow +1001 0010
    DH = -22 \rightarrow 1101 1110 \rightarrow 00100010 \rightarrow -22

CF=1 the result is negative
SBB (Subtract with Borrow)

SBB destination, source ; destination = destination - source - CF

- Analyze the following program:
  ; From the data segment:
    DATA1 DD 62562FAH
    DATA2 DD 412963BH
    RESULT DD ?
  ; From the code segment:
    MOV AX, WORD PTR DATA1
    SUB AX, WORD PTR DATA2
    MOV WORD PTR RESULT, AX
    MOV AX, WORD PTR DATA1+2
    SBB AX, WORD PTR DATA2+2
    MOV WORD PTR RESULT+2, AX

- Solution:
  DATA1 = 62562FAH = 0110001001010110001011111010
  DATA2 = 412963BH = 0100000100101001011000111011
  - After the SUB \( \text{AX} = 62FA - 963B = CCBF \) \( \text{CF} = 1 \).
  - Since \( \text{CF} = 1 \), when SBB is executed, \( \text{AX} = 625 - 412 - 1 = 212 \).
  - Therefore, the value stored in RESULT is 0212CCBF.
Multiplication of unsigned numbers

**Byte  Byte**

- One of the operands must be in the **AL** register
- The second operand can be either in a **register** or in **memory**
- After the multiplication, the result is in **AX**.

**Example:**

Multiply 25H by 65H and store the result in memory

**Solution:**

; From the data segment:

```
RESULT DW ?
```

; From the code segment:

```
MOV AL, 25H
MOV BL, 65H
MUL BL
MOV RESULT, AX
```
Example: Byte × Byte Multiplication

; From the data segment
DATA1  DB  25H
DATA2  DB  65H
RESULT DW ?

; From the code segment
MOV AL,DATA1
MOV BL,DATA2
MUL BL ; register addressing mode
MOV RESULT,AX

OR
MOV AL,DATA1
MUL DATA2 ; direct memory addressing mode
RESULT,AX

OR
MOV AL,DATA1
MOV SI,OFFSET DATA2
MUL BYTE PTR [SI] ; register indirect addressing mode
MOV RESULT,AX
Multiplication of unsigned numbers
word x word

- One operand must be in AX
- The second operand can be in a register or memory
- After the multiplication, registers DX and AX will contain the result.
  - AX will hold the lower word and
  - DX the higher word.

**Example:**
Multiply 2378H by 2F79H and store the result in memory

; From the data segment:
DATA3 DW 2378H
DATA4 DW 2F79H
RESULT DW 2 DUP(?)

; From the code segment:
MOV AX, DATA3
MUL DATA4
MOV RESULT, AX
MOV RESULT+2, DX
Multiplication of unsigned numbers
word x byte

- **AL** contain the byte
- **AH** must be zero
- The second operand can be in a register or memory.
- After the multiplication, registers **DX** and **AX** will contain the result.
  - **AX** will hold the lower word and **DX** the higher word.

**Example:**

Multiply 2378H by 79H and store the result in memory

; From the data segment:
DATA3  DW   2378H
DATA4  DB   79H
RESULT DW   2 DUP(?)

; From the code segment:

MOV   AL, DATA3
MOV   AH, 0
MUL   DATA4
MOV   BX, OFFSET RESULT
MOV   [BX], AX
MOV   [BX]+2, DX
## Unsigned Multiplication Summary

<table>
<thead>
<tr>
<th>Multiplication</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte × byte</td>
<td>AL</td>
<td>register or memory</td>
<td>AX</td>
</tr>
<tr>
<td>word × word</td>
<td>AX</td>
<td>register or memory</td>
<td>DX AX</td>
</tr>
<tr>
<td>word × byte</td>
<td>AL = byte, AH = 0</td>
<td>register or memory</td>
<td>DX AX</td>
</tr>
</tbody>
</table>

**Flags** ➔ **Affected OF, CF**

**Unpredictable SF, ZF, AF, PF**
Division of unsigned numbers
Byte over byte

- The numerator must be in the AL register \((AL \leftarrow 32)\)
- AH must be set to zero \((AH \leftarrow 0)\)
- The denominator cannot be immediate but can be in a register or memory.
  - \((DL \leftarrow 5)\)
- After the DIV instruction is performed (DIV DL),
  - the quotient is in AL \((AL \leftarrow 6)\)
  - the remainder is in AH \((AH \leftarrow 2)\)
DIV & Addressing Modes

DATA1  DB  95
DATA2  DB  10
QUOT   DB  ?
REM    DB  ?

; using immediate addressing mode will give an error
MOV AL,DATA1  ;move data into AL
SUB AH,AH     ;clear AH
DIV 10        ;immediate mode not allowed!!

; using direct mode
MOV AL,DATA1   ;AL holds numerator
SUB AH,AH      ;AH must be cleared
DIV DATA2      ;divide AX by DATA8
MOV QUOT,AL    ;quotient = AL = 09
MOV REM,AH     ;remainder = AH = 05
DIV & Addressing Modes (Cont.)

;using register addressing mode
MOV AL, DATA1
SUB AH, AH
MOV BH, DATA2
DIV BH
MOV QUOT, AL
MOVREM, AH

;using register indirect addressing mode
MOV AL, DATA1
SUB AH, AH
MOV BX, OFFSET DATA2
DIV BYTE PTR [BX]
MOV QUOT, AL
MOV REM, AH
The numerator is in AX
DX must be cleared
The denominator can be in a register or memory.
After the DIV, AX will have the quotient and the remainder will be in DX.

```assembly
MOV AX,10050 ;AX holds numerator
SUB DX,DX ;DX must be cleared
MOV BX,100 ;BX used for denominator
DIV BX
MOV QUOT,AX ;quotient = AX = 64H = 100
MOV REM,DX ;remainder = DX = 32H = 50
```
Word/Byte

- The numerator is in AX
- The denominator can be in a register or memory.
- After the DIV instruction, AL will contain the quotient, and AH will contain the remainder.
- The following program divides AX = 2055 by CL = 100.
- Then AL = 14H (20 decimal) is the quotient and AH = 37H (55 decimal) is the remainder.

```
MOV AX,2055 ;AX holds numerator
MOV CL,100 ;CL used for denominator
DIV CL
MOV QUOT,AL ;AL holds quotient
MOV REM,AH ;AH holds remainder
```
Doubleword / Word

- The numerator is in DX and AX
  - The most significant word in DX and the least significant word in AX.
- The denominator can be in a register or in memory.
- After the DIV instruction, the quotient will be in AX, the remainder in DX.

; from the data segment:
DATA1 DD 105432
DATA2 DW 10000
QUOT DW ?
REMAIN DW ?

; from the code segment:
MOV AX, WORD PTR DATA1 ; AX holds lower word
MOV DX, WORD PTR DATA1+2 ; DX higher word of numerator
DIV DATA2
MOV QUOT, AX ; AX holds quotient
MOV REMAIN, DX ; DX holds remainder
## Unsigned Division Summary

<table>
<thead>
<tr>
<th>Division</th>
<th>Numerator</th>
<th>Denominator</th>
<th>Quotient</th>
<th>Rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte/byte</td>
<td>AL = byte, AH = 0</td>
<td>register or memory</td>
<td>AL</td>
<td>AH</td>
</tr>
<tr>
<td>word/word</td>
<td>AX = word, DX = 0</td>
<td>register or memory</td>
<td>AX</td>
<td>DX</td>
</tr>
<tr>
<td>word/byte</td>
<td>AX = word</td>
<td>register or memory</td>
<td>AL</td>
<td>AH</td>
</tr>
<tr>
<td>doubleword/word</td>
<td>DX AX = doubleword</td>
<td>register or memory</td>
<td>AX</td>
<td>DX</td>
</tr>
</tbody>
</table>

Flags ➔ Unpredictable OF, SF, ZF, AF, PF, CF
LOGIC INSTRUCTIONS
AND

AND destination, source

- This instruction will perform a bitwise logical AND on the operands and place the result in the destination.
- The destination operand can be a register or in memory.
- The source operand can be a register, in memory, or immediate.
- **AND will automatically change the CF and OF to zero**
- Show the results of the following:
  
  - MOV BL, 35H
  - AND BL, 0FH

- **Solution:**
  
  - 35H → 0011 0101
  - 0FH → 0000 1111
  - 05H ← 0000 0101

  **Flag settings will be:** SF = 0, ZF = 0, PF = 1, CF = 0, OF = 0
The Usage of AND

- AND can be used to mask certain bits of the operand.
- It can also be used to test for a zero operand:
  
  ```
  AND DH, 0FFH
  AND DH, DH
  JZ XXX
  ...
  XXX: ...
  ```

- After executing `AND DH, DH`, the content of DH does not change, but, the ZF will be affected.

- AND instruction can be used to reset certain bits
  
  ```
  AND DH, 11111100B
  ```
The destination and source operands are ORed and the result is placed in the destination.

- The destination operand can be a register or in memory.
- The source operand can be a register, in memory, or immediate.
- OR can be used to set certain bits of an operand to 1.

```
OR DH,00000011B
```

- CF and OF will be reset to zero
- SF, ZF, and PF will be set according to the result.
- All other flags are not affected.
- The OR instruction can also be used to test for a zero operand.

```
OR DH,0H
OR DH,DH
JZ XXX
```

XXX: ...
LOGIC INSTRUCTIONS

XOR

XOR destination, source

- The XOR instruction will eXclusive-OR the operands and place the result in the destination.
- XOR sets the result bits to 1 if they are not equal; otherwise, they are reset to 0.
- CF = 0 and OF = 0 are set internally
- The rest are changed according to the result of the operation.
- Show the results of the following:
  
  MOV DH, 54H
  XOR DH, 78H

- 54H $\rightarrow$ 01010100
- 78H $\rightarrow$ 01110000
- 2CH $\leftarrow$ 00101100
- Flag settings will be: SF = 0, ZF = 0, PF = 0, CF = OF = 0.
There are two kinds of shift

- Logical shift: for unsigned operands
- Arithmetic shift: for signed operands.

Using shift instructions shifts the contents of a register or memory location right or left.

The number of times (or bits) that the operand is shifted can be specified directly if it is once only, or through the CL register if it is more than once.

**Flags** ➞ Affected: OF, SF, ZF, PF, CF

**Unpredictable:** AF
Logical Shift Right

- The operand is shifted right bit by bit, and for every shift
  - the LSB will go to the carry flag (CF) and
  - the MSB is filled with 0

```assembly
MOV AL, 9AH
MOV CL, 3
SHR AL, CL
```

**Solution:**

9AH \(\rightarrow\) 10011010

<table>
<thead>
<tr>
<th>Shift</th>
<th>Binary</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01001101</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00100110</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>00010011</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>000010011</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical Shift Right (Cont.)

- Although SHR does affect the OF, SF, PF, and ZF flags, they are not important in this case.
- The operand to be shifted can be in a register or in memory, but immediate addressing mode is not allowed for shift instructions.
  - SHR 25,CL ; this instruction will cause an error.
- Show the results of SHR in the following:

  ; from the data segment:
  DATA1 DW 7777H

  ; from the code segment:
  TIMES EQU 4
  MOV CL, TIMES
  SHR DATA1, CL

- Solution

  0111011101110111 ➔ 0011101110111011 ➔ 0001110111011101
  ➔ 0000111011101110 ➔ 0000011101110111
Logical Shift Lift

- After every shift, the LSB is filled with 0 and the MSB goes to CF.
- Show the effects of SHL in the following:
  - MOV DH,6
  - MOV CL,4
  - SHL DH,CL
- Solution:
  - Initially  
    
    | CF | 00000110 |
    | CF=0 | 00011000 |
    | CF=0 | 00110000 |
    | CF=0 | 01100000 |
    | CF=0 | 11000000 |

  - After the four shifts left, the DH register has 60H and CF = 0.
COMPARE of unsigned numbers

CMP destination, source

- The CMP instruction compares two operands and changes the flags according to the result of the comparison.
- The operands themselves remain unchanged.
- The destination operand can be in a register or in memory.
- The source operand can be in a register, in memory, or immediate.
- Although all the CF, AF, SF, PF, ZF, and OF flags reflect the result of the comparison, only the CF and ZF are used.

<table>
<thead>
<tr>
<th>Compare operands</th>
<th>CF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination &gt; source</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>destination = source</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>destination &lt; source</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Unsigned J instructions

- JA ; Jump Above  
  Jump if CF=0 and ZF=0

- JAE ; Jump Above or Equal  
  Jump if CF=0

- JB ; Jump Below  
  Jump if CF=1

- JBE ; Jump Below or Equal  
  Jump if CF=1 and ZF=1

- JE ; Jump Equal  
  Jump if ZF=1

- JNE ; Jump Not Equal  
  Jump if ZF=0
The rotation instructions **ROR**, **ROL** and **RCR**, **RCL** are designed specifically to perform a bitwise rotation of an operand.

They allow a program to rotate an operand right or left.

The operand can be in a register or memory.

If the number of times an operand is to be rotated is more than 1, this is indicated by CL. (This is similar to the shift instructions.)

**Flags ➔ Affected: OF, CF**
In rotate right, as bits are shifted from left to right they exit from the right end (LSB) and enter the left end (MSB).

In ROR the LSB is moved to the MSB and is also copied to CF

If the operand is to be rotated once, the 1 is coded

If it is to be rotated more than once, register CL is used to hold the number of times it is to be rotated.

\[
\begin{align*}
\text{MOV AL, 36H} & ; \text{ AL = 00110110} \\
\text{ROR AL, 1} & ; \text{ AL = 00011011 CF=0} \\
\text{ROR AL, 1} & ; \text{ AL = 10001101 CF=1} \\
\text{ROR AL, 1} & ; \text{ AL = 11000110 CF=1} \\
\text{MOV CL, 3} & \\
\text{ROR AL, CL} & 
\end{align*}
\]
ROL rotate left

- In rotate left, bits are shifted from right to left they exit the left end (MSB) and enter the right end (LSB).
- In ROL the MSB is moved to the LSB and is also copied to CF
- If the operand is to be rotated once, the 1 is coded.
- Otherwise, the number of times it is to be rotated is in CL.

```
MOV BH, 72H ;BH = 0111 0010
ROL BH, 1 ;BH = 11100100 CF=0
ROL BH, 1 ;BH = 11001001 CF=1
ROL BH, 1 ;BH = 1001 0011 CF=1
ROL BH, 1 ;BH = 00100111 CF=1

MOV BH, 72H ;BH=0111 0010
MOV CL, 4 ;CL=4 number of times to rotate
ROL BH, CL ;BH=00100111 CF=1
```
BCD, ASCII, and Control Instructions
BCD AND ASCII OPERANDS AND INSTRUCTIONS

- BCD (binary coded decimal) is needed because human beings use the digits 0 to 9 for numbers.
- Binary representation of 0 to 9 is called BCD
  - Unpacked BCD: one BCD/byte
  - Packed BCD: two BCD/byte
- Examples
  - \((94)_{\text{unpacked BCD}} \Rightarrow 00001001 \ 00000100\)
  - \((94)_{\text{packed BCD}} \Rightarrow 10010100\)

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

; from the data segment:
ASC DB '357894162'
UNPACK DB 10 DUP (?)

; from the code segment:
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, 0F0FH ; 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
ASCII to Unpacked BCD (Cont.)

; from the data segment:
ASC DB '357894162'
UNPACK DB 10 DUP (?)

; from the code segment:
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, WORD PTR [BX] ; move next 2 ASCII numbers to AX
AND AX, 0F0FH ; 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

- ; from the data segment:
- ASC DB '357894162'
- UNPACK DB 10 DUP (?)

- ; from the code segment:
- 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, WORD PTR [BX] ; move next 2 ASCII numbers to AX
- AND AX, 0F0FH ; 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
ASCII to Unpacked BCD (Cont.)

; from the data segment:
ASC DB '357894162'
UNPACK DB 10 DUP(?)
; from the code segment:

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 ; make the pointer to point at next byte
LOOP AGAIN ; loop until it is finished
ASCII to Packed BCD

; from the data segment:
VAL_ASC DB '47'
VAL_BCD DB ?

; from the code segment:
MOV AX, WORD PTR VAL_ASC
MOV BX, OFFSET ASC
; AH=37, AL=34
AND AX, 0F0FH
; AH=7, AL=4
XCHG AH, AL
; swap AH and AL: AH=4, AL=7
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: AL=47
MOV VAL_BCD, AL
; save the result
**Packed BCD to ASCII conversion**

- To convert packed BCD to ASCII
  - it must first be converted to unpacked and then
  - the unpacked BCD is tagged with 011 0000 (30H).

<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>00000010 &amp; 0000 1001</td>
<td>011 0010&amp;011 1001</td>
</tr>
</tbody>
</table>

MOV AL, VAL_BCD ; AL = 29H
MOV AH, AL ; copy AL to AH, now AH = 29, AL = 29H
AND AX, 0F00FH ; AH = 20, AL = 09H
MOV CL, 4
SHR AH, CL ; AH = 02, AL = 09H
OR AX, 3030H ; AH = 32, AL = 39H
XCHG AH, AL ; AH = 39, AL = 32H
MOV VAL_ASC, AX
MOV AL, 17H
ADD AL, 28H

- Adding 17H and 28H gives 0011 1111B (3FH), which is not BCD!
- A BCD number can only have digits from 0000 to 1001 (or 0 to 9).
- The result above should have been 17 + 28 = 45 (0100 0101).
- To correct this problem, the programmer must add 6 (0110) to the low digit: 3F + 06 = 45H.
DAA (Decimal Adjust for Addition) Instruction

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 (AL=72H)
MOV DATA3,AL ; store result in correct BCD form

- **DAA works only on AL**

- If after an ADD or ADC instruction 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.

- In reality there is no other use for the AF except for BCD addition and correction.
BCD subtraction and correction

- When subtracting packed BCD (single-byte or multi-byte) operands, the DAS instruction is put after the SUB or SBB instruction.
- AL must be used as the destination register to make DAS work.
- If after a SUB or SBB instruction 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.
Example

Assume that the following operands represent the budget, the expenses, and the balance, which is the budget minus the expenses.

BUDGET DT 87965141012
EXPENSES DT 31610640392
BALANCE DT ?

MOV cx,10
MOV BX,00
CLC
BACK: MOV AL, BYTE PTR BUDGET[BX]
SBB AL, BYTE PTR EXPENSES[BX]
DAS
MOV BYTE PTR BALANCE[BX], AL
INC BX
LOOP BACK
ASCII addition

MOV AL, '5' ; AL = 35
ADD AL, '2' ; add to AL 32 the ASCII for ‘2’
AAA ; changes 67H to 07H
OR AL, 30H ; OR AL with 30H to get ASCII

If the addition results in a value of more than 9, AAA will correct it and pass the extra bit to carry and add 1 to AH.

SUB AH, AH ; AH = 00
MOV AL, '7' ; AL = 37H
MOV BL, '5' ; BL = 35H
ADD AL, BL ; 37H + 35H = 6CH therefore AL = 6C.
AAA ; changes 6CH to 02 in AL and AH = CF = 1
OR AX, 3030H ; AX = 3132 which is the ASCII for 12H

AAA only on the AL register
The data added can be unpacked BCD rather than ASCII, and AAA and AAS will work fine.

MOV AX, 105H ; AX=0105H unpacked BCD for 15
MOV CL, 06 ; CL=06H
SUB AL, CL ; 5-6 = -1 (FFH)
AAS ; FFH in AL is adjusted to 09, and
; AH is decremented, leaving AX = 0009
Unpacked BCD multiplication

AAM: ASCII adjust multiplication

- The Intel manual says that this mnemonic stands for "ASCII adjust multiplication," but it really is unpacked multiplication correction.
- If two unpacked BCD numbers are multiplied, the result can be converted back to BCD by **AAM**

```
MOV AL, '7' ; AL=37H
AND AL, 0FH ; AL=07 unpacked BCD
MOV DL, '6' ; DL=36H
AND DL, 0FH ; DL=06 unpacked BCD
MUL DL ; AX=ALxDL =07x06=002AH=42
AAM ; AX=0402 (7x6=42 unpacked BCD)
OR AX, 3030H ; AX=3432 result in ASCII
```

**AAM works only on AX.**
Unpacked BCD division
AAD: ASCII adjust for division

- Before dividing the unpacked BCD by another unpacked BCD, AAD is used to convert it to HEX.
- By doing that the quotient and remainder are both in unpacked BCD.

```
MOV AX, 3539H ; AX=3539 ASCII for 59
AND AX, 0F0FH ; AH=05, AL=09 unpacked BCD data
AAD ; AX=003BH hex equivalent of 59
MOV BH, 08H ; divide by 08
DIV BH ; 3B/08 gives AL=07, AH=03
OR AX, 3030H ; AL=37H (quotient) AH=33H (rem)
```

AAD works only on AX.
Note that

<table>
<thead>
<tr>
<th>DAA</th>
<th>Add packed BCD numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAS</td>
<td>Works only on AL</td>
</tr>
<tr>
<td>AAA</td>
<td>Operands of addition are ASCII</td>
</tr>
<tr>
<td>AAS</td>
<td>After AAA, AL has unpacked BCD, AH has 0 or 1</td>
</tr>
<tr>
<td>AAA</td>
<td>Operands of addition are unpacked BCD</td>
</tr>
<tr>
<td>AAS</td>
<td>After AAA, AL has unpacked BCD, AH has 0 or 1</td>
</tr>
<tr>
<td>AAM</td>
<td>The operand of MUL is unpacked BCD</td>
</tr>
<tr>
<td></td>
<td>AL has unpacked BCD</td>
</tr>
<tr>
<td></td>
<td>After AAM, the result in AH and AL is in BCD</td>
</tr>
<tr>
<td>AAD</td>
<td>AX has unpacked BCD</td>
</tr>
<tr>
<td></td>
<td>After AAM, AX has binary value</td>
</tr>
<tr>
<td></td>
<td>Perform normal division</td>
</tr>
</tbody>
</table>
Control Transfer Instructions
NEAR and FAR

- **FAR**: The control is transferred outside the current code segment (*intersegment*: between segments).
  - CS and IP have to be updated.

- **NEAR**: The control is transferred to a memory location within the current code segment (*intrasegment*: within segment).
  - The next instruction to be executed is addressed by CS:IP registers.
    - IP is updated
    - CS remains the same

- **SHORT**: the address of the target must be within -128 to 127 bytes of the IP
  - The short jump is two byte instruction.
# Conditional Jumps

All conditional jumps are short jumps

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition Tested</th>
<th>&quot;Jump IF ...&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA / JNBE</td>
<td>(CF = 0) and (ZF = 0)</td>
<td>Above / not below nor equal</td>
</tr>
<tr>
<td>JAE / JNB</td>
<td>CF = 0</td>
<td>Above or equal/not below</td>
</tr>
<tr>
<td>JB / JNAE</td>
<td>CF = 1</td>
<td>Below / not above nor equal</td>
</tr>
<tr>
<td>JBE / JNA</td>
<td>(CF or ZF) = 1</td>
<td>Below or equal / not above</td>
</tr>
<tr>
<td>JC</td>
<td>CF = 1</td>
<td>Carry</td>
</tr>
<tr>
<td>JE / JZ</td>
<td>ZF = 1</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>JG / JNLE</td>
<td>((SF xor OF) or ZF) = 0</td>
<td>Greater / not less nor equal</td>
</tr>
<tr>
<td>JGE / JNL</td>
<td>(SF xor OF) = 0</td>
<td>Greater or equal / not less</td>
</tr>
<tr>
<td>JL / JNGE</td>
<td>(SF xor OF) = 1</td>
<td>Less / not greater nor equal</td>
</tr>
<tr>
<td>JLE / JNG</td>
<td>((SF xor OF) or ZF) = 1</td>
<td>Less or equal / not greater</td>
</tr>
<tr>
<td>JNC</td>
<td>CF = 0</td>
<td>Not carry</td>
</tr>
<tr>
<td>JNE / JNZ</td>
<td>ZF = 0</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>JNO</td>
<td>OF = 0</td>
<td>Not overflow</td>
</tr>
<tr>
<td>JNP / JPO</td>
<td>PF = 0</td>
<td>Not parity / parity odd</td>
</tr>
<tr>
<td>JNS</td>
<td>SF = 0</td>
<td>Not sign</td>
</tr>
<tr>
<td>JO</td>
<td>OF = 1</td>
<td>Overflow</td>
</tr>
<tr>
<td>JP / JPE</td>
<td>PF = 1</td>
<td>Parity / parity equal</td>
</tr>
<tr>
<td>JS</td>
<td>SF = 1</td>
<td>Sign</td>
</tr>
</tbody>
</table>

TRUE: The control is transfer to a new location
FALSE: The instruction follows the jump will be fetched
Short Jumps

- **All Conditional Jump are short jump**
  - The address of the target must be within -128 to +127 bytes of the IP.

- The conditional jump is **two-byte** instruction
  - One-byte is the opcode of the jump
  - The second byte is the value between 00 to FF (256 addresses from 0 → 127 → -128 → -1)

- **Forward jump**: the second byte is +ve number

- **Backward jump**: the second byte is -ve number
Conditional Jumps and Conditional Sets

- Always short jumps in 8086 - 80286.
  - limits range to within +127 and −128 bytes from the location following the conditional jump
- In 80386 and above, conditional jumps are either short or near jumps (±32K).
  - in 64-bit mode of the Pentium 4, the near jump distance is ±2G for the conditional jumps
- Allows a conditional jump to any location within the current code segment.
Figure 6–1 The three main forms of the JMP instruction. Note that Disp is either an 8- or 16-bit signed displacement or distance.
Example

0000  DTSEG  SEGMENT
0000  2512 15 1F 2B  DATA_IN  DB  25H,12H,15H,1FH,2BH
0005  00  SUM  DB  ?
0006  DTSEG  ENDS

1067:0000  B86610  MOV  AX, 1066
1067:0003  8ED8  MOV  DS,AX
1067:0005  B90500  MOV  CX,0005
1067:0008  BB0000  MOV  BX,0000
1067:000D  0207  ADD  AL,[BX] ; AGAIN
1067:000F  43  INC  BX
1067:0010  49  DEC  CX
1067:0011  75FA  JNZ  000D ; JNZ AGAIN
1067:0013  A20500  MOV  [0005],AL
1067:0016  B44C  MOV  AH,4C
1067:0018  CD21  INT  21

0DH - 13H
Unconditional Jump

JMP label ; the control is transferred unconditionally to the target location label.

- **JMP SHORT label** ; 2-byte instruction
  - The address of the target is within -128 to +127 bytes of the current IP.

- **JMP label** ; NEAR jump (3-byte instruction): within the current code segment.
  - Direct: JMP label ; IP±= 32K
  - Register indirect: JMP BX ; IP= BX
  - Memory indirect: JMP word ptr [DI] ; IP= [DI]

- **JMP FAR PTR label** ; 5-byte instruction, both IP and CS are updated.
LOOP

- A combination of a decrement CX and the JNZ conditional jump.

- In 8086 - 80286 LOOP decrements CX.
  - if CX != 0, it jumps to the address indicated by the label
  - If CX becomes 0, the next sequential instruction executes

- In 80386 and above, LOOP decrements either CX or ECX, depending upon instruction mode.
In 16-bit instruction mode, LOOP uses CX; in the 32-bit mode, LOOP uses ECX.
  - default is changed by the LOOPW (using CX) and LOOPD (using ECX) instructions 80386 - Core2
In 64-bit mode, the loop counter is in RCX.
  - and is 64 bits wide
There is no direct move from segment register to segment register instruction.
Conditional LOOPS

- LOOP instruction also has conditional forms: LOOPE and LOOPNE

- LOOPE (loop while equal) instruction jumps if CX != 0 while an equal condition exists.
  - will exit loop if the condition is not equal or the CX register decrements to 0

- LOOPNE (loop while not equal) jumps if CX != 0 while a not-equal condition exists.
  - will exit loop if the condition is equal or the CX register decrements to 0
- In 80386 - Core2 processors, conditional LOOP can use CX or ECX as the counter.
  - LOOPEW/LOOPED or LOOPNEW/LOOPNED override the instruction mode if needed
- Under 64-bit operation, the loop counter uses RCX and is 64 bits in width
- Alternates exist for LOOPE and LOOPNE.
  - LOOPE same as LOOPZ
  - LOOPNE instruction is the same as LOOPNZ
- In most programs, only the LOOPE and LOOPNE apply.
Call Statements

- Calling a procedure is a control transfer statement.
- Procedures make the program more structured.
- NEAR: the target address within the current code segment
  - IP must be saved in the stack
- FAR: the target address outside the current code segment
  - CS and IP must be saved in the stack
- The opcode for the RET instruction in case of NEAR and FAR is different.
  - NEAR: IP is restored
  - FAR: CS and IP are restored
Examples

CALL sum
:
sum PROC NEAR

    RET
sum ENDP

CALL prod
:
prod PROC FAR

    RETF
prod ENDP
.CODE
MAIN PROC FAR
    MOV AX, @DATA
    MOV DS, AX
    CALL SUBR1
    CALL SUBR2
    CALL SUBR3
    MOV AH, 4CH
    INT 21H
MAIN ENDP

SUBR1 PROC ...
    RET
SUBR1 ENDP

SUBR2 PROC ...
    RET
SUBR2 ENDP

SUBR3 PROC ...
    RET
SUBR3 ENDP
END MAIN
INTRO TO INTERRUPTS

- An interrupt is a **hardware-generated CALL**
  - externally derived from a hardware signal
- Or a **software-generated CALL**
  - internally derived from the execution of an instruction or by some other internal event
  - at times an internal interrupt is called an *exception*
- Either type interrupts the program by calling an **interrupt service procedure (ISP)** or interrupt handler.
Interrupt Vectors

- A 4-byte number stored in the first 1024 bytes of memory (00000H–003FFH) in real mode.
  - in protected mode, the vector table is replaced by an interrupt descriptor table that uses 8-byte descriptors to describe each of the interrupts
- 256 different interrupt vectors.
  - each vector contains the address of an interrupt service procedure
- Each vector contains a value for IP and CS that forms the address of the interrupt service procedure.
  - the first 2 bytes contain IP; the last 2 bytes CS
- Intel reserves the first 32 interrupt vectors for the present and future products.
  - interrupt vectors (32–255) are available to users
- Some reserved vectors are for errors that occur during the execution of software
  - such as the divide error interrupt
Some vectors are reserved for the coprocessor.
- others occur for normal events in the system

In a personal computer, reserved vectors are used for system functions.
Vectors 1–6, 7, 9, 16, and 17 function in the real mode and protected mode.
- the remaining vectors function only in the protected mode
Interrupt Instructions

- Three different interrupt instructions available:
  - INT, INTO, and INT 3

- In real mode, each fetches a vector from the vector table, and then calls the procedure stored at the location addressed by the vector.

- In protected mode, each fetches an interrupt descriptor from the interrupt descriptor table.

- Similar to a far CALL instruction because it places the return address (IP/EIP and CS) on the stack.
INTs

- 256 different software interrupt instructions (INTs) available to the programmer.
  - each INT instruction has a numeric operand whose range is 0 to 255 (00H–FFH)
- For example, INT 100 uses interrupt vector 100, which appears at memory address 190H–193H.
  - address of the interrupt vector is determined by multiplying the interrupt type number by 4
Address of the interrupt vector is determined by multiplying the interrupt type number by 4.

- INT 10H instruction calls the interrupt service procedure whose address is stored beginning at memory location 40H (10H \times 4) in the mode

In protected mode, the interrupt descriptor is located by multiplying the type number by 8

- because each descriptor is 8 bytes long

Each INT instruction is 2 bytes long.

- the first byte contains the opcode
- the second byte contains the vector type number
When a software interrupt executes, it:

- pushes the flags onto the stack
- clears the T and I flag bits
- pushes CS onto the stack
- fetches the new value for CS from the interrupt vector
- pushes IP/EIP onto the stack
- fetches the new value for IP/EIP from the vector
- jumps to the new location addressed by CS and IP/EIP
- INT performs as a far CALL
  - not only pushes CS & IP onto the stack, also pushes the flags onto the stack
- The INT instruction performs the operation of a PUSHF, followed by a far CALL instruction.
- Software interrupts are most commonly used to call system procedures because the address of the function need not be known.
- The interrupts often control printers, video displays, and disk drives.
- INT replaces a far CALL that would otherwise be used to call a system function.
  - INT instruction is 2 bytes long, whereas the far CALL is 5 bytes long
- Each time that the INT instruction replaces a far CALL, it saves 3 bytes of memory.
- This can amount to a sizable saving if INT often appears in a program, as it does for system calls.
IRET/IRETD

- Used only with software or hardware interrupt service procedures.
- IRET instruction will
  - pop stack data back into the IP
  - pop stack data back into CS
  - pop stack data back into the flag register
- Accomplishes the same tasks as the POPF followed by a far RET instruction.
- When IRET executes, it restores the contents of I and T from the stack.
  - preserves the state of these flag bits
- If interrupts were enabled before an interrupt service procedure, they are automatically re-enabled by the IRET instruction.
  - because it restores the flag register
- IRET is used in real mode and IRETD in the protected mode.
INT 3

- A special software interrupt designed to function as a breakpoint.
  - a 1-byte instruction, while others are 2-byte

- Common to insert an INT 3 in software to interrupt or break the flow of the software.
  - function is called a breakpoint
  - breakpoints help to debug faulty software

- A breakpoint occurs for any software interrupt, but because INT 3 is 1 byte long, it is easier to use for this function.
INTO

- Interrupt on overflow (INTO) is a conditional software interrupt that tests overflow flag (O).
  - If O = 0, INTO performs no operation
  - If O = 1 and an INTO executes, an interrupt occurs via vector type number 4
- The INTO instruction appears in software that adds or subtracts signed binary numbers.
  - Either these operations, it is possible to have an overflow
- JO or INTO instructions detect the overflow.
An Interrupt Service Procedure

- Interrupts are usually reserved for system events.
- Suppose a procedure is required to add the contents of DI, SI, BP, and BX and save the sum in AX.
  - as a common task, it may be worthwhile to develop the task as a software interrupt
- It is also important to save all registers are changed by the procedure using USES.
Interrupt Control

- Two instructions control the INTR pin.
- The **set interrupt flag** instruction (STI) places 1 in the I flag bit.
  - which enables the INTR pin
- The **clear interrupt flag** instruction (CLI) places a 0 into the I flag bit.
  - which disables the INTR pin
- The STI instruction enables INTR and the CLI instruction disables INTR.