EXPERIMENT NO. 6 - 7

6 ARITHMETIC OF SIGNED INTEGERS, DOUBLE PRECISION, BCD AND FLOATING POINT NUMBERS IN 8086

6.1 Objectives

The objectives of this experiment are

- Revisiting **ADD**, **SUB**, **MUL**, **DIV** instructions.
- To learn how to handle carry and borrow propagation using **ADC** and **SBB** instructions.
- To get familiarized with arithmetic of signed integers using **NEG**, **IMUL** and **IDIV** instructions.
- To learn arithmetic of double precision, BCD and floating point numbers.

6.2 Learning Outcome

At the end of the experiment the students will be able to

- Have an in-depth understanding of effects of different arithmetic instructions on flags.
- Handle negative numbers for arithmetic operations.
- Perform arithmetic operations on numbers having more than 16bits.
- Perform arithmetic operations on BCD numbers.

6.3 New Instructions in this experiment

Part A		Meaning
1	ADC	Add with carry
2	SBB	Subtract with borrow
Part B		
3	NEG	Negate
4	IMUL	Integer multiplication
5	IDIV	Integer division
6	CWD	Convert word to double word
7	CBW	Convert byte to word
Part C		
8	AAA	ASCII adjust for addition
9	AAS	ASCII adjust for subtraction
10	AAM	ASCII adjust for multiplication
11	AAD	ASCII adjust for division

PART A: REVISITING ADD, SUB, MUL, DIV INSTRUCTIONS

6.4 Revisiting ADD, SUB, MUL, DIV instructions

You have already been familiarized with the instructions ADD, SUB, DIV, MUL. In case of ADD and SUB instructions, the operands (source and destination) can be of either byte form (8bit) or word form (16bit). For MUL instruction, the multiplier and multiplicand can be both of either byte form or word form and the product is of either word form or double word form (32bit). For DIV instruction, the dividend and divisor are of byte and word forms respectively or double word and word form respectively. Table 1 briefly revisits these four instructions with their effects on the carry flags.

Table 1(a): Revisiting ADD instruction

From	Syntax	Flag effected	Sample Code
Byte	`ADD BYTE_DEST, BYTE_SOURCE`	C (carry)	`ADD BL, 5H`
Word	`ADD WORD_DEST, WORD_SOURCE`	C (carry)	`ADD BX, DATA[5]`

* WORD may be a AL, BL etc, or 8bit memory location.

* BYTE may be a AX, BX etc, or 16bit memory location.

Table 1(b): Revisiting SUB instruction

From	Syntax	Flag effected	Sample Code
Byte	`SUB BYTE_DEST, BYTE_SOURCE`	C (borrow)	SUB AL, BL
Word	`SUB WORD_DEST, WORD_SOURCE`	C (borrow)	SUB DATA, 5H

Table 1(c): Revisiting MUL instruction

From	Syntax	Multiplier	Multiplicand	Product	Sample Code	Flags effected
Byte	`MUL BYTE_MULP`	`BYTE_MULP`	AL	AX	`MUL BL`	CF/OF =0, if upper half of the product is zero =1, otherwise
Word	`MUL WORD_MULP`	`WORD_MULP`	AX	DX:AX	`MUL DAT_BYT`	

* CF/OF=1 means that the product is too big to fit in the lower half of the destination (AL for byte multiplication and AX for word multiplication)

* The effect of MUL on the SF, ZF are undefined.

Table 1(d): Revisiting DIV instruction

From	Syntax	Divisor	Dividend	Quotient	Remainder	Sample Code
Byte	`MUL BYTE_DIVR`	`BYTE_DIVR`	AX	AL	AH	`DIV DATA_8BIT`
Word	`MUL WORD_DIVR`	`BYTE_DIVR`	DX:AX	AX	DX	`DIV BX`
Divide overflow: It is possible that the quotient will be too big to fit in the specified destination (AL or AX). This can happen if the divisor is much smaller than the dividend. When this happened the program terminates and the system displays the message "Divide Overflow".						

* The effect of DIV on the flags is that all status flags are undefined.

The following example illustrates the effects of **MUL** and **DIV** instruction on flags.

```
;EXAMPLE 1
;RUN THE CODE IN SINGLE STEP MODE

CODE  SEGMENT
      ASSUME CS:CODE

;MULTIPLICATION IN WORD FORM
      MOV AX, 23h
      MOV BX, 25h
      XOR DX, DX
      MUL BX      ;CHECK THE CARRY FLAG, OVERFLOW FLAG, ZERO FLAG

      MOV AX, 0FFFEH
      MOV BX, 0FF06H
      MOV DX, 0
      MUL BX      ;CHECK THE CARRY FLAG, OVERFLOW FLAG, ZERO FLAG

;MULTIPLICATION IN BYTE FORM
      MOV AL, 9h
      MOV BL, 5h
      XOR AH, AH
      MUL BL      ;CHECK THE CARRY FLAG, OVERFLOW FLAG, ZERO FLAG

      MOV AL, 0FFH
      MOV BL, 0A6H
      MOV AH, 0
      MUL BL     ;CHECK THE CARRY FLAG, OVERFLOW FLAG, ZERO FLAG

;DIVISION IN WORD FORM
      MOV DX, 0FFF4H
      MOV AX, 0FFA4H
      MOV CX, 0FFH
      DIV CX     ;CHECK THE REMAINDER AND QUOTIENT

;DIVISION IN BYTE FORM
      MOV AX, 0FAH
      DIV A     ;CHECK THE REMAINDER AND QUOTIENT
```

```
;CHECK THE CARRY FLAG, OVERFLOW FLAG, ZERO FLAG, NOTE EFFECTS OF DIV ON
;FLAGS ARE UNDEFINED


     HLT


ORG 50H
A DB 0FH


CODE  ENDS
      END
```

Problem Set 6.4

1 Can you perform the division FFFF FFFFh ÷ 5h using DIV instruction? Explain why divide overflow occurs in this case?

2 If the result of a multiplication in word form is zero, how can you check it? Verify your suggested method by writing an assembly code.

6.5 Handling carry and borrow in addition and subtraction

The instruction **ADC** (add with carry) adds the source operand and CF to destination, and the instruction **SBB** (subtract with borrow) subtracts the source operand and CF from the destination.

Table 2: Syntaxes and operations of ADC and SBB instructions

Syntax	Operation
`ADC destination, source`	destination←source+destination+carry(CF)
`SBB destination, source`	destination←destination-source-borrow(CF)

6.6 Report (Part A)

Submit the solution of the problem set 6.4. Check the integrity of your codes by an 8086 emulator.

PART B: ARITHMETIC WITH SIGNED NUMBERS

6.7 Unsigned and Signed Integers

An unsigned integer is an integer that represents a magnitude, so it is never negative. So far in this laboratory we have dealt with unsigned integers. A signed integer can be positive or negative. In a signed integer the most significant bit is reserved for the sign: 1 means negative and 0 means.

For example,

If FFFFh is handled as an unsigned integer it will represent +65536 in decimal. On the other hand, If FFFFh is handled as a signed integer it will represent -1 in decimal.

6.8 Obtaining Two's Complement

The following syntax transfers the 2's complement of a constant into the source.

```
MOV source, -constant
```

For example,

```
MOV AX, -4H
```

transfers the 2's complement of 4 = FFFC to AX.

To negate (to obtain the 2's complement of) the contents of register or a memory location the **NEG** instruction can be used. The syntax is

```
NEG destination
```

The following example illustrates conversion on numbers to their two's complements. For each number, calculate the two's complement in a scratch-paper and verify them with the two's complement calculated by 8086.

```
;EXAMPLE 2
CODE   SEGMENT
```

```
      ASSUME CS:CODE

      MOV CX, 5
      MOV DI, 0
NEG_: MOV AX, [A+DI]
      NEG AX
      MOV [B+DI], AX
      ADD DI, 2
      LOOP NEG_
      HLT

ORG 050H
A DW 1H , 0FFFFH, 0F056H, 1056H, 4059H
ORG 060H
B DW ?, ?, ?, ?, ?

CODE  ENDS
      END
```

Problem Set 6.7:

1 Write a code to perform subtraction without using **SUB** instruction. One operand is a register and another is a memory location.

6.9 Signed Multiplication and Division

The instructions **MUL** and **DIV** handle unsigned numbers. To handle signed numbers, two different instructions are used respectively. Their syntaxes are

```
IMUL multiplier
```

And

```
IDIV divisor
```

The following points for signed multiplication and division are to be noted:

1. For both the instructions all the operands the considered signed integers.
2. The product of signed multiplication is also a signed integer.
3. For signed division the remainder has the same sign as the dividend.

Both the instructions attributes are same for MUL and DIV instructions as depicted in table 1(c) and 1(d) except that the effect of IMUL on status flag is a bit different.

CF/OF	=0, if upper half of the product is the sign extension of the lower half (this means the bits of the upper half are the same as the sign bit og the lower half) =1, otherwise

The following examples illustrate the differences between MUL and IMUL.

Table 3: Examples for illustrating the differences of MUL and IMUL.

No.	AX	BX	Instruction	Decimal product	Hex product	DX	AX	CF/OF
1	1h	FFFFh	MUL BX	65535	0000 FFFF	0000	FFFF	0
			IMUL BX	-1	FFFF FFFF	FFFF	FFFF	0
2	FFFFh	FFFFh	MUL BX	4294836225	FFFE 0001	FFFE	0001	1
			IMUL BX	1	0000 0001	0000	0001	0
3	0FFFh	X	MUL AX	16769025	00FF E001	00FF	E001	1
			IMUL AX	16769025	00FF E001	00FF	E001	1
4	0100h	FFFFh	MUL BX	16776960	00FF FF00	00FF	FF00	1
			IMUL BX	-256	FFFF FF00	FFFF	FF00	0

The following examples illustrate the differences between DIV and IDIV.

Table 4: Examples for illustrating the differences of MUL and IMUL.

No.	DX	AX	BX	Instruction	Decimal quotient	Decimal Remainder	AX	DX
1	0000h	0005h	0002h	DIV BX	2	1	0002	0001
				IDIV BX	2	1	0002	0001
2	0000h	0005h	FFFEh	DIV BX	0	5	0000	0005
				IDIV BX	-2	1	FFFE	0001
3	FFFFh	FFFBh	0002h	DIV BX	Divide Overflow			
				IDIV BX	-2	-1	FFFE	FFFF

The following example illustrates the difference of **IMUL** and **MUL** instructions. Try to explain why the results in of **MUL** and **IMUL** instructions are different.

```
;EXAMPLE 3
CODE  SEGMENT
      ASSUME CS:CODE
      MOV AX,0FFFFH     ; AX=FFFFh (UNSIGNED), -1 (SIGNED)
      MOV BX, -1H       ; AX=FFFFh (UNSIGNED), -1 (SIGNED)

      PUSH AX

      ;MULTIPLICATION USING MUL
      XOR DX, DX
      MUL AX            ; DX:AX= FFFF FE01h
      MOV PROD_MUL, DX
      MOV PROD_MUL+2, AX

      ;MULTIPLICATION USING IMUL
      POP AX
      XOR DX, DX
      IMUL AX           ; DX:AX= 0000 0001h
      MOV PROD_IMUL, DX
      MOV PROD_IMUL+2, AX
      IMUL BX
      HLT

      ORG 050H
PROD_MUL DW ?, ?
      ORG 060H
PROD_IMUL DW ?, ?

CODE  ENDS
      END
```

Problem Set 6.8

1 Run the following assembly code to find the square of the number stored in AX in your microprocessor kit. Are the results obtained from IMUL and MUL equal? Explain the similarity or differences in the results.

```
;EXAMPLE 4
CODE        SEGMENT
```

```
            ASSUME CS:CODE

            MOV AX,0F056h
            PUSH AX

            ;MULTIPLICATION USING MUL
            XOR DX, DX
            MUL AX
            MOV SQ_MUL, DX
            MOV SQ_MUL+2, AX

            ;MULTIPLICATION USING IMUL
            POP AX
            XOR DX, DX
            IMUL AX
            MOV SQ_IMUL, DX
            MOV SQ_IMUL+2, AX
            HLT

            ORG 050H
SQ_MUL DW ?, ?
            ORG 060H
SQ_IMUL DW ?, ?

CODE        ENDS
      END
```

6.10CWD Instruction

CWD stands for *convert word to double word*. When using **IDIV**, DX should be made the sign extention of AX. **CWD** will do this extension.

For example,

```
MOV AX, -1250  ;AX = FB1Eh
CWD
```

will make DX sign extension of AX, so that DX:AX = FFFF FB1E h.

In the following example it is intended to calculate the -4010d÷7d. Try to explain why the result is not correct when sign extension to DX is not performed.

```
;EXAMPLE 5
CODE        SEGMENT
            ASSUME CS:CODE

            MOV AX, -4010D
            MOV BX, 7D
            PUSH AX

            ;CASE 1: DIVISION USING DIV

            XOR DX, DX
            DIV BX
            MOV Q_DIV, AX
            MOV R_DIV, DX

            ;CASE 2: DIVISION USING IDIV WITHOUT SIGN EXTENSION TO DX
            POP AX
            PUSH AX
            XOR DX, DX
            IDIV BX
            MOV Q_IDIV1, AX
            MOV R_IDIV1, DX

            ;CASE 3: DIVISION USING IDIV WITH SIGN EXTENTION TO DX
            POP AX
            CWD
            IDIV BX
            MOV Q_IDIV2, AX
            MOV R_IDIV2, DX
            HLT

            ORG 050H
Q_DIV  DW ?
R_DIV  DW ?
            ORG 060H
Q_IDIV1 DW ?
R_IDIV1 DW ?
```

```
            ORG 070H
Q_IDIV2 DW ?
R_IDIV2 DW ?


CODE ENDS
      END
```

Problem Set 6.9

1 Consider example 3 in table 4. Explain why divide overflow occurs for **DIV** instruction, while not for **IDIV** instruction.

2 If your intended division is 105Fh÷Fh which cases will give the correct result? Which case(s) will give the correct result when the intended division is F0FFh÷Ch?

3 The equivalent of **CWD** for integer division in byte form is **CBW** (Convert byte to word). Write an assembly code perform a integer division (**IDIV**) in byte form.

4 Now try the division of AX by BL with AX=00FBh and BL=FFh. Determine for which of the instructions (**IDIV** and **DIV**) divide overflow occurs and explain?

6.11 Report (Part B)

Submit the solution of the problem sets, 6.7, 6.8, 6.9. Check the integrity of your codes by an 8086 emulator.

PART C: Arithmetic with Double Precision Numbers, BCD Numbers and Floating Point Numbers

6.12 Introducing Double-Precision Numbers

Numbers stored in the 8086 based microprocessors can be 8 or 16-bit numbers. Even for 16-bit numbers, the range is limited to 0 to 65535 for unsigned numbers and -32768 to +327678 for signed numbers. TO extend this range, a common technique is to use 2 words for each number. Such numbers are called **double-precision numbers** and he range here is 0 to 2^{32}-1 or 4,294,967,295 for unsigned and -2,147,483,648 to -2,147,483,648 for signed numbers.

A double-precision number may occupy two registers or two memory words. For example, if a 32-bit number is stored in the two memory words A and A+2, written A+2:A, then the upper 16 bits are in A+2 and the lower 16 bits are in A. If the number is signed, then the msb of A+2 is the sign bit. Negative numbers are represented in two's complement form.

6.13 Addition, Subtraction & Negation of Double-Precision Numbers

To add or subtract two 32 bit numbers, we first add or subtract the lower 16 bits and then add or subtract the higher 16 bits. In case of addition, carry generated in the addition of the lower 16 bit numbers must be added to the sum of the higher 16 bit, which can be done by **ADC** instruction. Similarly in case of subtraction, borrow generated in the subtraction of the lower 16 bit numbers must be subtracted from the subtracted result of the higher 16 bit, which can be done by **SBB** instruction. The following numerical example illustrates addition of double precision numbers, FFFF FA11h and 0A12 1001.

Carry from higher 16 bits	Higher 16 bit	Carry from lower 16 bits	Lower 16 bit
	FFFF		FA11
	+0A12		+1001
1	0A11	1	0A12
	+1		
1	0A12		0A12

Hence the final result of addition in 10A120A12.

Hence the algorithm for addition of A+2:A and B+2:B can be shown as below:

1. AX←A, BX←B
2. C←AB+BX

3. AX←A+2, BX←B+2
4. C+2←AX+BX+C(Carry)

The following example illustrates double-precision addition and subtraction.

```
;EXAMPLE 6
; DOUBLE PRECISION ADDITION AND SUBTRACTION
; C+4:C+2:C = A+2:A + B+2:B
; D+2:D = A+2:A - B+2:B
CODE SEGMENT
ASSUME CS:CODE,DS:CODE

      MOV AX, A
      MOV BX, B
      ADD AX, BX
      MOV C, AX
      MOV AX, A+2
      MOV BX, B+2
      ADC AX, BX
      MOV C+2, AX
      ADC C+4, 0


      MOV AX, A
      MOV BX, B
      SUB AX, BX
      MOV D, AX
      MOV AX, A+2
      MOV BX, B+2
      SBB AX, BX
      MOV D+2, AX
      HLT

ORG 0050H
A      DW   0F056H, 4509H
ORG 0060H
B      DW   1056H, 1509H
ORG 0070H
C      DW   ?,?, ?
```

```
ORG 0080H
D       DW    ?,?

CODE   ENDS
       END
```

Problem Set 6.11

1 Write the assembly code to perform addition of a 32 bit number and a 48 bit number.

2* Write the assembly code to perform addition of two 80 bit numbers invoking a procedure. (Use CALL, RET instructions.)

3 Explain how the following instructions form the negation of A+2:A.

```
NOT A+2
NOT A
INC A
ADC A+2, 0
```

Can you negate A+2:A using **NEG** instruction?

6.14 Multiplication of Double-Precision Numbers

The following example illustrates the multiplication of a double precision number, A+2:A with a contents of the register BX. The algorithm for multiplication in shown symbolically in figure 1. The product is stored in C.

```
                    BX
            × A+2: A
-------------------------------
                P2 :AX
          Q2:   Q1 xx
-------------------------------
   Q2+Carry: P2+Q1:C
    (=C+4)   (=C+2)
```

Figure 1: Symbolical representation of algorithm for Multiplication of B and A+2:A

From figure 1 the algorithm is generated are presented below:

1. AX←A
2. DX:AX←AX×BX
3. C←AX
4. TEMP←DX
5. AX←A+2
6. DX:AX← AX×BX

7. AX←AX+TEMP
8. C+2←AX
9. DX←DX+C (Carry)
10. C+4←DX

```
;EXAMPLE 7
; DOUBLE PRECISION MULTIPLICATION
; C+4:C+2:C=BX x A+2:A

CODE        SEGMENT
            ASSUME      CS:CODE,DS:CODE

            ORG 0100H
            MOV BX, 0E56FH
            XOR DX, DX
            MOV AX, A       ;STEP 1
            MUL BX          ;STEP 2
            MOV C, AX       ;STEP 3
            MOV TEMP, DX    ;STEP 4
            MOV AX, A+2     ;STEP 5
            MUL BX          ;STEP 6
            ADD AX, TEMP    ;STEP 7
            MOV C+2, AX     ;STEP 8
            ADC DX, 0       ;STEP 9
            MOV C+4, DX     ;STEP 10
            HLT

            ORG 0150H
A       DW  0F056H, 4509H
            ORG 0160H
C       DW  ?, ?,?
TEMP    DW  ?

CODE  ENDS
      END
```

Problem Set 6.12

1 Write the assembly code to perform multiplication of two 32 bit numbers.

6.15 Division of a 48-bit number by a 16-bit number

The algorithm for performing A+4:A+2:A÷BX division is stated below:

1. DX:AX←A+4:A+2
2. Quotient AX, Remainder DX←DX:AX÷BX
3. Q+2←AX
4. AX←A
5. Quotient AX, Remainder DX←DX:AX÷BX
6. Q←AX, R←DX

The following code performs a division of 48-bit number by a 16-bit number.

```
; EXAMPLE 8
;THIS EXAMPLE PERFORMS A 48 BIT NUMBER BY 16 BIT NUMBER DIVISION
;Q+2:Q=A+4:A+2:A/BX, R = REMAINDER
CODE SEGMENT
      ASSUME CS:CODE,DS:CODE

      MOV BX, 0F015H
      MOV DX, A+4
      MOV AX, A+2
      DIV BX
      MOV Q+2, AX
      MOV AX, A
      DIV BX
      MOV Q, AX
      MOV R, DX
      HLT
ORG 50H
A DW 1536H, 4563H, 1234H
ORG 60H
Q DW ?, ?
ORG 70H
R DW ?
CODE ENDS
END
```

Problem Set 6.13

1 Write an assembly program to perform a 64 bit by 16 bit division.

2* Can you extend this algorithm to perform a 48 bit by 32 bit division by using **DIV** instruction? If not, why?

6.16 BCD Arithmetic

The BCD (binary coded decimal) number system uses four bits to code each decimal digit, from 0000 to 1001. The combinations 1010 to 1111 are illegal in BCD.

Since only 4 bits are required to represent a BCD, two digits can be placed in a byte. This is known as packed BCD form. In unpacked BCD form, only one digit is contained in a byte. The 8086 has addition and multiplication instructions to perform with both forms, but for multiplication and division, the digits must be unpacked.

6.17 BCD Addition

In BCD addition, we perform the addition on one digit at a time. Let us consider the addition of the BCD numbers contained in AL and BL. When addition performed using **ADD** instruction, it is possible to obtain a non-BCD result. For example if AL= 6d and BL=7d, the sum of 13 is in AL which is no longer a valid BCD digit. To adjust it is required to subtract 10 from AL and place 1 in AH, then AX will contain the correct sum.

AH	0000 0000		AL		0000 0110
			BL	+	0000 0111
			AL		0000 1101
	+ 1			-	0000 1010
AH	0000 0001		AL		0000 0011

This adjustment is performed in 8086 if we add the instruction **AAA** (ASCII Adjust for addition).

For example, the following assembly code performs decimal addition on the unpacked BCD numbers in AL and BL.

```
MOV AH, 0
ADD AL, BL
AAA
```

Example 9 performs the addition of two two-digit numbers stored in A+1:A and B+1:B.

```
;Example 9
;ADDITION OF TWO TWO-DIGIT NUMBERS
;C+2:C+1:C = A+1:A + B+1:B
```

```
CODE        SEGMENT
            ASSUME      CS:CODE,DS:CODE

            ORG 0100H
            XOR AH, AH
            MOV AL, A
            ADD AL, B
            AAA

            MOV C, AL
            MOV AL, AH
            ADD AL, A+1

            ADD AL, B+1
            MOV AH, 0
            AAA
            MOV C+1, AL
            MOV C+2, AH
            HLT

            ORG 0150H
A    DB     7, 9  ;A+1:A=09 07
B    DB     5, 6  ;B+1:B=06 05
            ORG 0160H
C    DB     ?, ?, ?

CODE ENDS
     END
```

6.18BCD Subtraction

As in BCD addition, BCD subtraction is also performed on one digit at a time. Let us consider the subtraction of the BCD numbers contained in AL and BL. When subtraction performed using **SUB** instruction, it is possible to obtain a non-BCD result. For example to subtract 26 from 7, we put AH=2, AL= 6, BL = 7. After subtracting BL from AL, we obtain a incorrect result in AL. To adjust it is required to subtract 6 from AL, clear high nibble (most significant 4 bits) and subtract 1 from AH, then AX will contain the correct sum.

AH	0000 0010	AL		0000 0110
		BL	-	0000 0111
	0000 0010	AL		1111 1111
	1 1		-	0000 0110
AH	0000 0001	AL		0000 1001

This adjustment is performed in 8086 if we add the instruction **AAS** (ASCII Adjust for subtraction) after subtraction. The usage of **AAS** is shown below.

```
MOV AH, 0
SUB AL, BL
AAS
```

Problem Set 6.16

1* Write down the code to subtract the two-digit number in bytes B+1:B from the one contained in A+1:A.

6.19 BCD Multiplication

Let us consider an example of single digit BCD multiplication. To multiply 7 and 9, we put 7 in AL and 9 in BL. After multiplying them using **MUL** instruction, AH will contain 00 3Fh= 63d. To convert the content in AX to 06 03, the instruction **AAM** (ASCII adjustment for multiplication) should follow. The usage of **AAM** for BCD multiplication is shown below.

```
MUL BL
AAM
```

The following example illustrates the effect **AAM** on AX and performs a single digit BCD multiplication. Run the code in single step mode.

```
;EXAMPLE 10
CODE        SEGMENT
            ASSUME      CS:CODE,DS:CODE
            ;OBSERVE THE EFFECTS OF AAM ON THE CONTENTS IN AX IN SINGLE
            ;STEP MODE
            MOV AX, 97H
            AAM

            MOV AX, 97D
            AAM
```

```
            MOV AX, 9804H
            AAM

            MOV AL, 9D
            MOV BL, 8D
            MUL BL
            AAM
            HLT

CODE ENDS
     END
```

6.20 BCD Division

Let us consider an example of division of a two digit BCD number by a single digit BCD number. To multiply divide 97 and 9, we put 0907 in AX and 9 in BL. Before dividing using **DIV**, the contents in AH is changed from 0907 to 97d=61h by **AAD** (ASCII adjust for division). After ordinary binary division, **AAM** instruction must follow to convert the contents to BCD format. The following example shows a BCD division.

The following example illustrates a BCD division. Run the code in single step mode.

```
;EXAMPLE 11
;BCD DIVISION
;95 BY 8
; Q=QUOTIENT, R=REMAINDER

CODE        SEGMENT
            ASSUME      CS:CODE,DS:CODE

MOV AL, 05
MOV AH, 09   ;AX CONTAINS DIVIDEND BCD 95
MOV BL, 8    ;BL CONTAINS DIVISOR BCD 8
AAD
DIV BL
MOV R, AH
AAM           ; CONVERTING THE QUOTIENT TO BCD FORMAT
MOV Q, AL
MOV Q+1, AH
            HLT
```

```
Q DB ?, ?
R DB ?
CODE ENDS
     END
```

Problem Set 6.17

1 Write down the code to perform a multiplication of a 2 digit BCD number by a single digit BCD number.

2 Write down the code to perform a division of a 3 digit BCD number by a 1 digit BCD number.

6.21 Introducing Floating-Point Numbers

In floating representation, each number is represented in two parts, a mantissa, which contains the loading significant bits in a numbers and an exponent, which is used to adjust the position of the binary point.

For example,

100000000b can be represented as 1×2^{8}; hence in floating point representation, it has mantissa 1 and exponent 8.

0.0001b can be represented as 1×2^{-4}; hence in floating point representation, it has mantissa 1 and exponent -4.

Floating-point numbers are convenient for handling numbers, some which are very large and some are fraction. Example 12 illustrates how floating point representation can perform the multiplication of two numbers one very large and another fraction.

```
;EXAMPLE 12
;MULTIPLICATION OF TWO NUMBERS IN FLOATING POINT REPRESENTATION
;A = 8000 0000 0000h    = 8x16^(C)
;B = 0.0000 0000 00F0h  = Fx16^(-B)
;C=AxB

CODE        SEGMENT
            ASSUME      CS:CODE,DS:CODE

            ORG 0100H
            MOV AX, A
            MUL B
```

```
            MOV C, AX


            MOV AX, A+2
            ADD AX, B+2
            MOV C+2, AX
            HLT
            ORG 0150H
A     DW    8H, 0CH   ;A = MANTISSA, A+2 = EXPONENT
B     DW    0FH, -0BH ;B = MANTISSA, B+2 = EXPONENT
            ORG 0160H
C     DW    ?, ? ;C = MANTISSA, C+2 = EXPONENT


CODE  ENDS
      END
```

6.22 Report (Part C)

Submit the solution of the problem sets, 6.11, 6.12, 6.13, 6.16, 6.17. Check the integrity of your codes by an 8086 emulator.

6.23 References

Ytha Yu, Charles Marut, ***Assembly Language Programming and Organization of the IBM PC,*** Mitchell McGraw-Hill, 1992.

→ Chapter 4: Introduction to IBM PC and Assembly Language (Part A)

→ Chapter 9: Multiplication and Division Instructions (Part B)

→ Chapter 18: Advanced Arithmetic (Part C)