A First Course In Assembly Language Programming

80 X 86 Assembly Language Computer Architecture

Howard Dachslager, Ph.D.

TABLE OF CONTENTS

.

A First Course In Assembly Language Programming

80 X 86 Assembly Language Computer Architecture

Howard Dachslager, Ph.D.

A First Course In Assembly Language Programming

80 X 86 Assembly Language Computer Architecture

Howard Dachslager, Ph.D.

Irvine Valley College

I. Working with Integer Numbers

CHAPTER 1 - NUMBER BASES FOR INTEGERS

INTRODUCTION

In order be become a proficient assembly language programmer, one needs to have a good understanding how numbers are represented in the assembler. To accomplish this, we start with the basic ideas of integer numbers. In later chapters we will expand these numbers to the various forms that are needed. We will also later, study decimal numbers as floating point numbers.

1.1 Definition of Integers

There are three types of integer numbers: positive , negative and zero.

Definition: The positive integer numbers are represented by the following symbols: 1,2,3,4,...

Definition: The negative integer numbers are represented by the following symbols: -1, -2, - 3, - 4, ...

Definition: The integer number zero is represented by the symbol: 0.

Definition: Integers are therefore defined as the following numbers: 0, 1, -1, 2, -2,

Examples: 123, - 143, 44, 3333333333333, - 72

Although the study of these numbers will give us a greater understanding of the types of numbers we are going to be concerned with when writing assembler language program, the reality is that the only kind of numbers that the assembler can handle are integers and finite decimals numbers. Further, we need to understand that the assembler cannot work within our decimal number system. The assembler must convert all numbers to the base 2. The number system that we normally work with is in the base 10 and they will then be converted by the assembler to the base 2. In this chapter we will define and examine the various number bases including those that we need to use when programming.

Numbers in the base 10

Definition: The set of all numbers whose digits are 0,1,2,3,4,5,6,7,8, 9 are said to be of the base 10.

Representing positive integers in the base 10 in expanded form.

Definition: Decimal integers in expanded form: $a_n a_{n-1} \ldots a_1 a_0 = a_n * 10^n + a_{n-1} * 10^{n-1} + \ldots + a_1 * 10 + a_0$

where $a_k = 0,1,2,3,4,5,6,7,8,9$.

Examples:

a. $235 = 2 * 10^2 + 3 * 10 + 5$ **b.** $56{,}768 = 5 * 10^4 + 6 * 10^3 + 7 * 10^2 + 6 * 10 + 8$

Exercises:

1. Write the following integers in expanded form:

a. 56 **b.** 26,578 **c.** 23,556,891,010

■

The number system that we use is said to be in the base 10. This because we only use the above 10 digits to build are decimal number system. . For the following discussion all numbers will be integers and non- negative. The following table shows how starting with 0, we systematically create numbers from these 10 digits:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
::::::::	:::::::	::::::::	:::::::: :	:::::::: ::	:::::::: :	::::::::	:::::::: :	:::::::: :	:::::::: ::
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109
:::::::: :	:::::::: :	:::::::: :	:::::::: :	:::::::: :	:::::::: :	:::::::: :	:::::::: :	:::::::: :	:::::::: :

The way we wish to think about creating these numbers is best described as follows:

First we list the ten digits 0 - 9 (row 1):

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

At this points we have run out of digits. To continue we start over again by first writing the digit 1 and to the right place the digit 0 - 9: (row 2):

10 , 11, 12, 13, 14, 15, 16, 17, 18, 19 .

Again we have run out of digits. To continue we start over again by first writing the digit 2 and to the right place the digit 0 - 9 (row 3):

20, 21, 22, 23, 24, 25, 26, 27, 28, 29 .

Continuing this way, we can create the positive integers as shown in the above table.

1.2 Numbers in Other Bases:

Base 8 (N_8)

Definition: Octal integers in expanded form: $a_n a_{n-1} \ldots a_1 a_0 = a_n * 10^n + a_{n-1} * 10^{n-1} + \ldots + a_1 * 10 + a_0$

where $a_k = 0,1,2,3,4,5,6,7$.

Examples:

a. $235 = 2 * 10^2 + 3 * 10 + 5$ **b.** $56761 = 5 * 10^4 + 6 * 10^3 + 7 * 10^2 + 6 * 10 + 1$

This number system is called the octal number system. In the early development of computers, the octal number system was extensively used. How do we develop the octal number system? In the same way we showed how we developed the decimal system; by using only 8 digits: 0, 1, 2, 3, 4, 5, 6, 7.

Note: Integer numbers that are in a base, other than 10 will distinguished by a subscript N.

0_8	1_8	2_8	3_8	4_8	5_8	6_8	7_8
10_8	11_8	12_8	13_8	14_8	15_8	16_8	17_8
20_8	21_8	22_8	23_8	24_8	25_8	26_8	27_8
30_8	31_8	32_8	33_8	34_8	35_8	36_8	37_8
::::::::	:::::::	::::::::	:::::::::	::::::::::	:::::::::	::::::::	:::::::::

70_8	71_8	72_8	73_8	74_8	75_8	76_8	77_8
100_8	101_8	102_8	103_8	104_8	105_8	106_8	107_8
:::::::::	:::::::::	:::::::::	:::::::::	:::::::::	:::::::::	:::::::::	:::::::::

First, we list the eight digits 0 - 7 (row 1):

0, 1, 2, 3, 4, 5, 6, 7

At this points we have run out of digits. To continue we start over again by first writing the digit 1 and to the right place the digit 0 -7 : (row 2):

10 , 11, 12, 13, 14, 15, 16, 17

Again we have run out of digits. To continue we start over again by first writing the digit 2 and to the right place the digit 0 - 9 (row 3):
20, 21, 22, 23, 24, 25, 26, 27

Continuing this way, we can create the positive integers as shown in the above table.
We can easily compare the development of the decimal and octal number system:

DECIMAL NUMBERS	**OCTAL NUMBERS (Base 8)**
0	0_8
1	1_8
2	2_8
3	3_8
4	4_8
5	5_8

6	6_8
7	7_8
8	10_8
9	11_8
10	12_8
11	13_8
12	14_8
13	15_8
14	16_8
15	17_8
16	20_8
17	21_8
18	22_8
19	23_8
20	24_8
::::::::::::::::::	::::::::::::::::::::::

Exercises:

1. Write an example of a 5 digit octal integer number.

2. In the octal number system, simplify the following expressions:

a. $2361_8 + 4_8$ **b.** $33_8 - 2_8$ **c.** $777_8 + 3_8$

3. What is the largest 10 digit octal number ?

4. Add on 10 more rows to the above table .

We wish to create number system in the base 5 (N_5).

5. What digits would makeup these numbers?

6. Create a 2 column, 21 row table, where the first column will be the decimal numbers 0 - 20 and the second column will consists of the corresponding numbers in the base 5, starting with the digit 0.

7. Write out the largest 7 digit number in the base 5.

8. In the base 5 number system simplify the following expressions:

$n_5 =$ **a.** $22212_5 + 3_5$ **b.** $23333_5 + 2_5$ **c.** $12011_5 - 2_5$
■

Base 2(N_2)

Definition: Binary integers in expanded form: $a_n a_{n-1} \ldots a_1 a_0 = a_n * 10^n + a_{n-1} * 10^{n-1} + \ldots + a_1 * 10 + a_0$

where $a_k = 0,1$.

Examples:

a. $101 = 1 * 10^2 + 0 * 10 + 1$ **b.** $11011 = 1 * 10^4 + 1 * 10^3 + 0 * 10^2 + 1 * 10 + 1$

This number system is called the binary number system. Binary numbers are the most important numbers since all numbers stored in the assembler are in the base 2. The digits that make these numbers are 0,1 and are called bits. Numbers made from these bits are called the binary numbers.

How do we develop the binary number system? In the same way we showed how to developed the decimal and the octal number system; by using only the 2 bits: 0, 1:

DECIMAL NUMBERS	BINARY NUMBERS
0	0_2
1	1_2
2	10_2
3	11_2
4	100_2
5	101_2
6	110_2
7	111_2
8	1000_2
9	1001_2
10	1010_2
11	1011_2
12	1100_2
13	1101_2
14	1110_2
15	1111_2
16	10000_2
17	10001_2
18	10010_2
19	10011_2
20	10100_2
:::::::::::::::::::::	::::::::::::::::::::::::

Exercises:

9. Extend the above table for the integer numbers 21 - 30.

10. Simplify (a). $10011_2 + 1_2$ (a). $1011_2 + 11_2$ (c). $10111_2 + 111_2$

11. Complete the following table:

OCTAL NUMBERS	**BINARY NUMBERS**
0_8	
1_8	
2_8	
3_8	
:::::::::	:::::::::::::
16_8	

12. From the above table, what does it tell us about the relationship of the digits of the octal system and the binary numbers?

■

Base 16 (N_{16})

Definition: Hexadecimal integers in expanded form:

$a_n a_{n-1} \ldots a_1 a_0 = a_n * 10^n + a_{n-1} * 10^{n-1} + \ldots + a_1 * 10 + a_0$

where a_k = 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Examples:

a. $2E5 = 2*10^2 + E*10 + 5$ **b.** $56ADF = 5*10^4 + 6*10^3 + A*10^2 + D*10 + F$

The number system in the base 16 is called the hexadecimal number system. Next to be binary number system, hexadecimal numbers are very important in that these numbers are used extensively to help the programmer to interpret the binary numeric values computed by the assembler. Many assemblers will display the numbers only in hexadecimal.

We can easily compare the development of the decimal and hexadecimal number system:

DECIMAL NUMBERS	**HEXADECIMAL NUMBERS**
0	0_{16}
1	1_{16}
2	2_{16}
3	3_{16}
4	4_{16}
5	5_{16}
6	6_{16}
7	7_{16}
8	8_{16}
9	9_{16}
10	A_{16}
11	B_{16}
12	C_{16}
13	D_{16}
14	E_{16}
15	F_{16}

16	10_{16}
17	11_{16}
18	12_{16}
19	13_{16}
20	14_{16}
21	15_{16}
22	16_{16}
23	17_{16}
24	18_{16}
25	19_{16}
26	$1A_{16}$
27	$1B_{16}$
28	$1C_{16}$
29	$1D_{16}$
30	$1E_{16}$
31	$1F_{16}$
32	20_{16}
::::::::::	::::::::::::::

Exercises:

13. Extend the above table for the decimal integer numbers 33 - 50.

14. Simplify n_{16} = (a). $A_{16} + 6_{16}$ (a). $FFFF_{16} + 1_{16}$ (c). $100_{16} + E_{16}$

15. Complete the following table:

OCTAL NUMBERS	**HEXADECIMAL NUMBERS**

0_8	
1_8	
2_8	
3_8	
::::::::	:::::::::::::
26_8	

16. Complete the following table:

HEXADECIMAL NUMBERS	**BINARY NUMBERS**
0_{16}	
1_{16}	
2_{16}	
3_{16}	
:::::::::	:::::::::::::
FF_{16}	

17. What does the above table tell you about the relationship of the binary and hexadecimal numbers ?

■

Project

In assembly language the basic binary numbers are made up of eight bits. A binary number of this type is called a *byte*. Therefore, a bye is an 8 bit number. For example, the decimal number 5 can be represented as the binary number 00000101.

Complete the following table. (Hint: First complete the hexadecimal byte column .)

HEXADECIMAL BYTE		**BINARY BYTE**		**DECIMAL BYTE**
0	0	0000	0000	0
0	1	0000	0001	1
::::::::::	::::::::::	::::::::::	::::::::::	::::::::::
F	F			

CHAPTER 2 - RELATIONS BETWEEN NUMBER BASES

INTRODUCTION

In this chapter we will study the one to one correspondence that exist between the various number bases. To accomplish this we approach these number systems as sets.

2.1 Sets

Definition of a set:

A set is a well defined collection of items where

1. each item in the set is unique

and

2. the items can be listed in any order.

Examples:

1. S = {a,b,c,d}

2. A = {23, -8 , 23.3 }

3. N_{10} = {0,1,2,3,4,5, ...} (base 10)

4. N_8 = {0,1,2,3,4,5,6,7,10,11,12,13,14,15,16,17,20, ...} (base 8)

5. N_2 = {0,1,10,11,100,101,110,111,1000, ...} (base 2)

6. N_{16} = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1E,1F,20,...} (base 16)

Exercises:

1. For the following bases, write out the first 10 numbers as a set in natural order:

(a). N_3 (b). N_4 (c). N_5 (d). N_6 (e). N_7

2**.** Assume we need to define a number system in the base 20 (N_{20}) . Create N_{20} by using digits and capital letters. Write out the first 40 numbers in their natural order.

■

2.2 One to One Correspondence Between Sets

Assume we have two sets, D, R. The set D is called the domain and the set R is called the range.

Definition of a one to one correspondence between sets:

We say there is a one to one correspondence between sets if the following rules hold:

Rule 1: There exists function $f : D \Rightarrow R$

Rule 2: The function f is one to one

Rule 3: The function f is onto

Definition of a one to one function:

A function is said to be one to one, if the following is true:

if $f(x_1) = f(x_2)$ then $x_1 = x_2$ where x_1, x_2 are contained in D.

Definition of an onto function:

A function is said to be onto, if the following is true:

if for every y in R, there exists a element x in D where $f(x) = y$.

Change in notation

For such functions we will use the notation: $D \Rightarrow R$

and $x \Rightarrow y$

If $D \Rightarrow R$, we write

$D \Leftrightarrow R$,

meaning the two sets D and R are in one to one correspondence.

Examples:

1. Let D = {1,2,3,4,5,...} and R = {2,4,6,8,10,12,...}.

Show there is a one-to-one correspondence between these two sets.

Solution:

$k \Rightarrow 2k$, where

k = 1,2,3,...

2. $D = \{1,2,3,4,5,...\}$ and $R = \{1,-1,2,-2,3,-3, ...\}$

Show there is a one-to-one correspondence between these two sets.

Solution:

For the odd numbers of D:

$2k + 1 \Rightarrow k + 1$

where $k = 0,1,2,3,...$

For the even numbers of D:

$2k \Rightarrow - k$

$k = 1,2,3,...$

Combining these into one function gives:
$1 \Rightarrow 1$
$2 \Rightarrow -1$
$3 \Rightarrow 2$
$4 \Rightarrow -2$
$5 \Rightarrow 3$
$6 \Rightarrow -3$
$7 \Rightarrow 4$
$8 \Rightarrow -4$
:::::::::

Exercises:

1. If $D = \{2,4,6,8,10, ...\}$ and $R = \{1,3,5,7,9,...\}$, show that $D \Leftrightarrow R$.

Finding the one to one correspondence Between Number Bases

It is important to be able to find the functions that establishes one to one corresponding between number bases.

To assist us, we establish the following laws about one to one correspondence:

1. If $D \Leftrightarrow R$ then $R \Leftrightarrow D$ (Reflexive law)

2. If $A \Leftrightarrow B$ and $B \Leftrightarrow C$ then $A \Leftrightarrow C$. (Transitive law)

We begin by finding the formula that gives a one - to - one correspondence

$N_b \Rightarrow N_{10}$

2.3 Converting numbers in any base b to its corresponding number in the base 10 ($N_b \Rightarrow N_{10}$):

Assume $a_na_{n-1}... a_1a_0$ is a number in the base N_b. The following formula give us a one-to - correspondence

$N_b \Rightarrow N_{10}$:

$n_b = a_na_{n-1}... a_1a_0 \Rightarrow a_nb^n + a_{n-1}b^{n-1} + ... + a_1b + a_0b^0 = n_{10}$

where

all computation are performed in the base 10.

Note: The above expansion is from right to left.

Examples:

1. $n_5 = 32412_5 \Rightarrow 3*5^4 + 2*5^3 + 4*5^2 + 1*5^1 + 2*5^0 = 3(625) + 2(125) + 4(25) + 1(5) + 2 = 2232_{10}$

Therefore, $32412_5 \Rightarrow 2232_{10}$.

$n_2 = 1110101_2 \Rightarrow 1*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1 = 64 + 32 + 16 + 4 + 1 = 117_{10}$

Therefore, $1110101_2 \Rightarrow 117_{10}$.

2. $n_{16} = 9B5F2_{16} \Rightarrow 9*16^4 + 11*16^3 + 5*16^2 + 15*16^1 + 2 = 589824 + 45056 + 1280 + 240 + 2 = 636402_{10}$

Therefore,

$9B5F2_{16} \Rightarrow 636402_{10}$

Note: In the above example we needed to replace the hexadecimal digit B with the decimal number 11 and the digit F with the decimal number 15.

The reason we are able to make a correspondence is that we can show there a one to one correspondence between the hexadecimal digits and the corresponding numbers of the decimal system as shown in the following table:

BASE 16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
BASE 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Exercises:

Convert the following numbers to the base 10.

a. 2022301_6 **b.** 66061_9 **c**. 11101101_2 **d.** 756402_8 **e.** $A0CD8_{16}$

■

2.4 Converting numbers in the base 10 to its corresponding number in any base b:

To convert a number in the base 10 to its corresponding number in any base b we use the famous Euclidean division theorem:

Euclidean Division Theorem: Assume N, b are non- negative integers. There exist unique integers Q, R where

$N_{10} = Qb + R$, where $0 \leq R < b$.

To compute Q and R, we use the following algorithm:

Step 1: Divide N by b which will result in a decimal value in the form *integer*. *fraction.*

Step 2: From Step 1, Q = *integer*

Step 3: R = N - Qb.

Example:

$N_{10} = 3451$, b = 34

Step 1: 3451/34 = 101.5

Step 2: Q = 101

Step 3: R = 3451 - 101 $*$ 34 = 17

Step 4: Therefore, N = Qb + R = 101 $*$ 34 + 17.

Using the Euclidean division theorem, we now show how to convert numbers in the base 10 to its corresponding numbers in the base b.

We want to write N_{10} in the form: $N_{10} = a_n b^n + a_{n-1} b^{n-1} + ... + a_1 b + a_0$

Step 1: Factor out the number b: $N_{10} = (a_n b^{n-1} + a_{n-1} b^{n-2} + ... + a_1)b + a_0 = Qb + R$ where,

$Q = a_n b^{n-1} + a_{n-1} b^{n-2} + ... + a_2 b + a_1$

$R = a_0$

Step 2: Set $N = Q = a_n b^{n-1} + a_{n-1}b^{n-2} + ... + a_2 b + a_1$.

$Q = Q_1 b + R_1 = (a_n b^{n-2} + a_{n-1}b^{n-3} + ... + a_2)b + a_1$ where

$Q_1 = a_n b^{n-2} + a_{n-1}b^{n-3} + ... + a_2$,

$R_1 = a_1$.

Step 4: Continue in this manner, until $Q_n = 0$.

$N_{10} \Leftrightarrow (a_n \; a_{n-1} \; ... \; a_1 \, a_0)_b$

Examples:

Convert the following decimal numbers to the specified base.

1. $1625_{10} \Leftrightarrow N_8$

Step 1: 1625/8 = 203.125

$a_0 = 1625 - 203 * 8 = 1$

Step 2: 203/8 = 25.375

$a_1 = 203 - 25 * 8 = 3$

Step 3: 25/8 = 3.125

$a_2 = 25 - 3 * 8 = 1$

Step 4: 3/8 = 0.375

$a_3 = 3 - 0 * 8 = 3$

Since Q = 0, the algorithm is completed.

$1625_{10} \Leftrightarrow (a_3a_2a_1a_0)_8 = 3131_8$

2. $89629_{10} \Leftrightarrow N_{16}$

Step 1: 89629/16 = 5601.8125

$a_0 = 89629 - 5601 * 16 = 13 \Leftrightarrow D$

Step 2: 5601/16 = 350.0625

$a_1 = 5601 - 350 * 16 = 1$

Step 3: 350/16 = 21.875

$a_3 = 350 - 21*16 = 14 \Leftrightarrow E$

Step 4: 21/16 = 1.3125

$a_4 = 21 - 1*16 = 5$

Step 5: 1/16 = 0.0625

$a_5 = 1 - 0*16 = 1$

Therefore, $89629 \Leftrightarrow (a_4a_3a_2a_1a_0)_{16} = 15E1D_{16}$

Exercises:

1. Convert the following:

a. $2545601_{10} \Leftrightarrow$ base 2 **b.** $16523823_{10} \Leftrightarrow$ base 16 **c.** $5321_{10} \Leftrightarrow$ base 3 **d.** $81401_{10} \Leftrightarrow$ base 8.

2 . Convert the number $2245_6 \Leftrightarrow N_4$ (Hint: first convert 2245_6 to decimal) .
■

2.5 Expanding Numbers in the Base b (N_b).

In the base 10 system (N_{10}) ,

$a_na_{n-1}... a_1a_0 = a_n10^n + a_{n-1} 10^{n-1} + ... + a_110 + a_0$.

Does such an expansion hold for all numbers in the base b **(N_b)** ? The answer is yes and the expansion can be written as

$(a_na_{n-1}... a_1a_0)_b = a_n10_b{}^n + a_{n-1} 10_b{}^{n-1} + ... + a_110_b + a_0$.

The following explains the validity of this expansion.

First note that the digits of any number in a given base is

0,1,2,... b - 1.

Following these digits is the number 10:

0,1,2,... b - 1, 10_b

Now in the base b, the following arithmetic holds:

$0 + 0 = 0$, $0*0 = 0$, $1 + 0 = 1$, $1*1 = 1$, $a_k * 0 = 0$, $a_k *1 = a_k$, $a_k *10 = a_k 0$

Therefore the following rules holds for any given base:

$10 * 10^n = 10^{n+1}$

and

$a_n 10_b^{\,n} + a_{n-1} 10_b^{\,n-1} + \ldots + a_1 10_b + a_0 = a_n 100\ldots0_b + \ldots + a_1 10_b + a_0 = a_n 00\ldots0_b + \ldots + a_1 0_b + a_0$.

Examples:

1. 2562_8 : $2 * 1000_8 + 5 * 100_8 + 6 * 10_8 + 2_8 = 2000_8 + 500_8 + 60_8 + 2_8 = 2562_8$

2. 10111_2 : $1 * 10000_2 + 0 * 1000_2 + 1 * 100_2 + 1 * 10_2 + 1 = 10000_2 + 000_2 + 100_2 + 10_2 + 1 = 10111_2$

3. $97FA_{16}$: $9 * 1000_{16} + 7 * 100_{16} + F * 10_{16} + A_{16} = 9000_{16} + 700_{16} + F0_{16} + A_{16} = 97FA_{16}$

Exercises:

Find the expansions for the following numbers in their give bases:

(a) 4312322_5 (b) $ABCDEF_{16}$ (C) 12322_4 (b) 111101101_2

■

2.6 A Quick Method of Converting Between Binary and Hexadecimal numbers

Of primary concern is to develop an easy conversion between binary and hexadecimal numbers without multiplication and division. Later we see that the ability to convert quickly between binary and hexadecimal decimal will be critical in learn to program in assembly language.

To perform this conversion we first construct a table comparing the 16 digits of the hexadecimal number system and the corresponding binary numbers:

HEXADECIMAL DIGITS	CORRESPONDING BINARY NUMBERS
0	0000_2
1	0001_2
2	0010_2
3	0011_2
4	0100_2
5	0101_2
6	0110_2
7	0111_2

8	1000_2
9	1001_2
A	1010_2
B	1011_2
C	1100_2
D	1101_2
E	1110_2
F	1111_2

Note: Each digit of the hexadecimal system, corresponds to a number of 4 bits in the binary number system.

Now we can convert between any binary number and hexadecimal number directly by the following rules:

<u>Converting a binary number to its corresponding hexadecimal number:</u>

Given any binary number the following steps will convert the number to hexadecimal:

Step 1: Group the binary number from right to left into 4 binary bit groups.

Step 2: From the table above, match the hexadecimal digit with each of the 4 binary bit group.

Example:

$$11011011010101011101_2 = \underset{3}{\underline{0011}}\ \underset{6}{\underline{0110}}\ \underset{D}{\underline{1101}}\ \underset{5}{\underline{0101}}\ \underset{D}{\underline{1101}}_2 \Leftrightarrow 36D5D_{16}$$

<u>Converting a hexadecimal number to its corresponding binary number:</u>

Given any hexadecimal number the following steps will convert the number to binary:

From the table above, match each of digits of the hexadecimal number with the corresponding 4 bit binary number.

Example:

$$34ABC02DE0F_{16} = \underset{0011}{3}\ \underset{0100}{4}\ \underset{1010}{A}\ \underset{1011}{B}\ \underset{1100}{C}\ \underset{0000}{0}\ \underset{0010}{2}\ \underset{1101}{D}\ \underset{1110}{E}\ \underset{0000}{0}\ \underset{1111_2}{F_{16}} \Leftrightarrow$$

$$= 00110100101010111100000000101101111000001111_2$$

Exercises:

1. Complete the table below that matching the digits of the octal number system with its corresponding binary numbers:

OCTAL DIGITS	CORRESPONDING BINARY NUMBERS
0	000
1	001

2. From the tables above convert quickly the following numbers:

a. $111011011110001101010 11_2 \Leftrightarrow n_8$

b. $67574112014_8 \Leftrightarrow n_2$

c. $235621103_8 \Leftrightarrow n_{16}$

d. $A2B3C4D5E6D7F_{16} \Leftrightarrow n_2$

e. $1101110101101110 01_2 \Leftrightarrow n_{16}$

3. Create a similar table to convert numbers of the base 4 to the base 2.

4. Using the tables, convert the following:

a. $121301_4 \Leftrightarrow n_2$ **b.** $121301_8 \Leftrightarrow n_4$ **c.** $10011100110_2 \Leftrightarrow n_4$

■

2.7 Performing Arithmetic For Different Number Bases

Given any number base, one can develop arithmetic operations so that we can perform addition, subtraction, and multiplication between integers numbers. For example $ABC23_{16} + 5_{16} = ABC28_{16}$. To perform operations such as addition, subtraction and multiplication within the given number system can be very confusing and prone to errors. The best way to do such computations is to convert the numbers to the base 10 and then perform arithmetic operations only in the base 10. Finally convert the resulting computed number back to the original base. The following theorem assures us that there is a consistency in arithmetic operations when we convert any number to the base 10

Theorem: Invariant properties of arithmetic operations between bases:

1. Invariant property of addition: If $N_b \Leftrightarrow N_c$ and $M_b \Leftrightarrow M_c$ then $N_b + M_b \Leftrightarrow N_c + M_c$.

2. Invariant property of subtraction: If $N_b \Leftrightarrow N_c$ and $M_b \Leftrightarrow M_c$ then $N_b - M_b \Leftrightarrow N_c - M_c$.

3. Invariant property of multiplication: If $N_b \Leftrightarrow N_c$ and $M_b \Leftrightarrow M_c$ then $N_b * M_b \Leftrightarrow N_c * M_c$

The following algorithm will allow us to perform arithmetic operations using the above theorem.

Step 1: Convert each number to the base 10.

Step 2: Perform the arithmetic operation on the converted numbers.

Step 3: Convert the resulting number from Step 2 back to the original base.

Examples:

a. Perform $2367_8 + 471123_8$

Step 1:

$2367_8 \Leftrightarrow 2*8^3 + 3*8^2 + 6*8 + 7 = 1271_{10}$

$471123_8 = 4*8^5 + 7*8^4 + 1*8^3 + 1*8^2 + 2*8 + 3 = 160339_{10}$

Step 2: $1271_{10} + 160339_{10} = 161610_{10}$

Step 3: Through long division,

$161610_{10} \Leftrightarrow 473512_8$

Step 4: Therefore,

$2367_8 + 471123_8 = 473512_8$

b. Perform $56AF02_{16} * 682FA_{16}$

Step 1:

$56AF02_{16} \Leftrightarrow 5* 16^5 + 6*16^4 + 10*16^3 + 15*16^2 + 0*16^1 + 2 = 5680898_{10}$

$682FA_{16} \Leftrightarrow 426746_{10}$

Step 2: $5680898_{10} * 426746_{10} = 2,424,300,497,908_{10}$

Step 3: Through long division,

$2,424,300,497,908_{10} \Leftrightarrow 2347391EBF4_{16}$

Step 4: Therefore,

$56AF02_{16} * 682FA_{16} = 2347391EBF4_{16}$

c. Perform $1011101101_2 - 10101011_2$

Step 1:

$1011101101_2 \Leftrightarrow 749_{10}$

$10101011 \Leftrightarrow 171_{10}$

Step 2: $749_{10} - 171_{10} = 578_{10}$

Step 3: Through long division,

$578_{10} \Leftrightarrow 1001000010_2$

Step 4: Therefore,

$1011101101_2 - 10101011_2 = 1001000010_2$

Note: Since we are only working with integer number, we will postpone division for later chapters.

Exercise:

1. For each of the above examples, verify the result in Step 3.

2. Perform the following:

a. $(212_3 + 2222_3) * 101_3$ **b.** $(101101_2 - 1101_2) * 11101_2$ **c.** $AB2F_{16} * 23D_{16} + 2F5_{16}$

3. Using the laws of arithmetic, show that for any number in the base b, $N_b = a_n a_{n-1} \ldots a_1 a_0, \; a_k < b$

can be written in the expanded form

$N_b = a_n * 10_b^n + a_{n-1} * 10_b^{n-1} + \ldots + a_1 * 10_b + a_0$

4. Show that $10^n_b \Rightarrow b^n_{10}$

■

Project

Show that the one-to-one function $f^{-1}: N_{10} \Rightarrow N_b$ is the inverse of f: N $N_b \Rightarrow N_{10}$.
(Hint: Show $f^{-1}(f(n_b)) = n_b$)

CHAPTER - 3 PSEUDO-CODE AND WRITING ALGORITHMS

INTRODUCTION

In this chapter we will learn the basics of computer programming. This involves defining a set of instructions, called pseudo-code that when written, in a specific order, will perform desired tasks. When completed such a sequence of instructions are call a computer program. We used the word pseudo-code in that the codes are independent of any specific computer language. Finally, we then use this code as a guide to writing the desired programs in assembly language.

3.1 The Assignment Statement

The form of the assignment statement is:

VARIABLE := VALUE

where

VARIABLE is a name that begins with a letter and can be letters, digits.

VALUE is any numeric value of base 10, variable or a mathematical expressions.

Note. Frequently, instructions are referred to as statements

The assignment statement is used to assign a numeric value to a variable.

Rules of assignment statements

R1: The left-hand side of an assignment statement **must** be a variable.

R2: The assignment statement will evaluate the right-hand side of the statement first and will place the result in the variable name specified on the left-side of the assignment statement. The quantities on the right-hand side are unchanged; only the variable on the left-hand side is changed. Always read the assignment statement from right to left.

Examples:

ASSIGNMENT STATEMENTS	X	X2	XYZ	SAM	TURNS
X2 := 3		3			
XYZ := 23		3	23		
TURNS := XYZ		3	23		23
X2 := 5		5	23		23

Exercises:

1. Complete the following table:

ASSIGNMENT STATEMENTS	**T**	**YZ2**	**TABLE**	**FORM**	**TAB**
YZ2 := 3					
TABLE :=YZ2					
YZ2 := 1123					
FORM :=TABLE					
YZ2 := FORM					

2. Which of the following are illegal assignment statements. State the reason.

a. XYZ := XYZ **b.** 23 := S1 **c.** 2ZX := XZ **d.** MARY MARRIED := JOHN
■

Exchanging the Contents of Two Variables:

An important task is swapping or exchanging the contents of two variable. The following example shows how this is done:

Example:

ASSIGNMENT STATEMENTS	**X**	**Y**	**TEMP**
X := 4	4		
Y :=12	4	12	
TEMP := X	4	12	4
X := Y	12	12	4
Y := TEMP	12	4	4

Note: To perform the swap, we needed to create an additional variable TEMP.

Exercises:

3. Assume we have the following assignments:

A	**B**	**C**	**D**
10	20	30	40

Write a series of assignment statements which will rotate the values of A,B,C,D as show in the table below:

A	B	C	D
40	10	20	30

4. The instructions:

S := R
R :=T
T := S

will exchange the contents of the variables R and T. (a). True (b). False

5. The following instructions

A := 2
B := 3
Z := A
A := B
B := Z

will exchange the contents of the variables A and B. (a). True (b). False

6.

X := 5
Y := 10
Z := 2
Z := X
X := Y
Y := Z

The above sequence of commands will exchange the values in the variables __________ and _________.

■

3.2: Mathematical Expressions

Our system has the following mathematical operators that can be used to evaluate mathematical expressions:

Mathematical Operator	Symbol	Example	Restrictions
Multiplication	$x * y$	$3 * 5 = 15$	none
Integer Division	$x \div y$	$7 \div 2 = 3$	$y \neq 0$

Mod	x mod y	7 mod 2 = 1 7 = 2∗3 + 1	y ≠ 0
Addition	x + y	2 + 4 = 6	none
Subtraction	x - y	5 - 9 = - 4	none

IMPORTANT: All numbers are of type integer.
.
Order of Operations

The following are the order of operations:

- parenthesis, exponentiation, multiplication & division & integral division, addition & subtraction.
- When in doubt make use of parenthesis.

Examples:

ASSIGNMENT STATEMENTS	**X**	**Y**
X := 4	4	
Y :=5	4	5
X := 2∗X + 3∗Y + X	27	5

ASSIGNMENT STATEMENTS	**X**	**Y**
X := 4	4	
Y :=5	4	5
X := 2∗(X + Y)∗(X + Y) + X	166	5

Important: Remember to <u>always</u> evaluate assignment statements from right to left.

Iterative Addition

Addition of several numbers can be compute using repetitive addition:

S := S + X

Examples:

1. Add, using repetitive addition, the number 2, 4, 6, 8.

ASSIGNMENT STATEMENTS	S	X
S := 0	0	
X := 2	0	2
S := S + X	2	2
X := 4	2	4
S := S + X	6	4
X : = 6	6	6
S := S + X	12	6
X := 8	12	8
S := S + X	20	8

2. Add the digits of 268: 2 + 6 + 8

INSTRUCTIONS	N	R	SUM
N:= 268	**268**		
SUM := 0	268		**0**
R := N MOD 10	268	**8**	0
SUM := SUM + R	268	8	**8**
N := N - R	**260**	8	8
N := N÷10	**26**	8	8
R := N MOD 10	26	**6**	8
SUM := SUM + R	26	6	**14**
N := N - R	**20**	6	14
N := N÷10	**2**	6	14
R := N MOD 10	2	**2**	14
SUM := SUM + R	2	2	**16**
N := N - R	**0**	2	16
N := N÷10	0	2	16

Exercises:

1. Complete the table:

ASSIGNMENT STATEMENTS	X
X := 2	
X := X∗X	
X := X + X	
X := X∗X	

2. Complete the table:

ASSIGNMENT STATEMENTS	X	U	W
X:=5			
W:= 2			
U := 4			
W := W∗(W + U÷W)∗(W + U ÷W)			
X := X∗X + U			

3. Complete the table below.

ASSIGNMENT STATEMENTS	X	T1	Z
X:=3			
Z := 15			
T1:=10			
X:=Z+X∗X			
Z:=X+Z+1			
T1:=T1 + Z÷T1 + T1			

4. Evaluate the following expressions:

a. 2 + 3∗4
b. 2 + 2∗2∗2 ÷ 4 - 3
c. 2 + 2∗2∗2 ÷ (7 - 3)
d. 17 ÷ 2
e. 17 ÷ 2

f. $16 \div 2$
g. $3 + 9 \div 3$
h. $3 + 8 \div 3$
I. $3 + 79 \div 3$
j. $3 + 2 * 2 * 2 \div 8 * 2 - 5$
k. $3 + 2 * 2 * 2 \div (8 * 2 - 5)$

5. Set up a table for evaluating the following sequence of instructions.

NUM1 := 0
NUM2 := 20
NUM3 := 30
SUM1 := NUM1 + NUM2
SUM2 := NUM2 + NUM3
TOTAL := NUM1 + NUM2 + NUM3
AVG1 := SUM1 ÷ 2
AVG2 := SUM2 ÷ 2
AVG := TOTAL ÷3

6. Set up a table for evaluating the following sequence of instructions:

X := 2
X := 2∗X + X
X := 2∗X + X
X := 2∗X + X
X := 2∗X + X
X := 2∗X + X
X := 2∗X + X
■

3.3 Algorithms and Programs

Definition of an algorithm: An algorithm is a sequence of instructions that solves a given problem.

Definition of a program: A program is a sequence of instructions and algorithms.

Examples:

1. Assume N and P are positive integers. We can write

$N = QP + R$ where $R < P$.

The following algorithm and program will demonstrate how to compute and store Q and R.

Algorithm:

ASSIGNMENT STATEMENTS	EXPLANATION
Q := N÷P	COMPUTES AND STORES THE INTEGRAL PART
R := N MOD P	COMPUTES AND STORES THE REMAINDER R

Task 1: Store the number 957

Task 2: Store the number 35

Task 3: Find Q and R for 957 = Q∗35 + R

Program:

ASSIGNMENT STATEMENTS	N	P	Q	R
N := 957	957			
P := 35	957	35		
Q := N÷P	957	35	27	
R := N MOD P	957	35	27	12

2. We define n- factorial:

N! = N∗(N - 1)∗(N - 2) ...∗(1)

for N, a positive integer.

The following algorithm uses the repetitive multiplication statement to compute N!

Algorithm:

ASSIGNMENT STATEMENTS	EXPLANATION
NFACTORIAL := N	SET THE INITIAL VALUE
N:= N-1	REDUCES N BY 1
NFACTORIAL:= NFACTORIAL∗N	
N:= N-1	
NFACTORIAL:= NFACTORIAL∗N	
N:= N-1	
NFACTORIAL:= NFACTORIAL∗N	
:::::::::::::::::::	
N:= N-1	
NFACTORIAL:= NFACTORIAL∗N	TERMINATES WHEN N = 1

The following program computes 5!

Program:

ASSIGNMENT STATEMENTS	N	NFACTORIAL
N:= 5	**5**	
NFACTORIAL := N	5	**5**
N:= N-1	**4**	5
NFACTORIAL:= NFACTORIAL$*$N	4	**20**
N:= N-1	**3**	20
NFACTORIAL:= NFACTORIAL$*$N	3	**60**
N:= N-1	**2**	60
NFACTORIAL:= NFACTORIAL$*$N	2	**120**
N:= N-1	**1**	120
NFACTORIAL:= NFACTORIAL$*$N	1	120

3. The Fibonacci Sequence

To create a Finonacci sequence, we begin with the numbers

0, 1.

Step 1: Add the above 2 numbers (0 + 1 = 1) and insert the number in the above sequence:

0,1,1

Step2: Add the last 2 numbers (1 + 1 = 2) of the above sequence and insert the number in the above sequence:

0,1,1,2

Step3: Add the last 2 numbers (1 + 2 = 3) of the above sequence and insert the number in the above sequence:

0,1,1,2,3

Continue as often as desired.

The following algorithm uses the above steps will compute the Fibonacci sequence to a desired number of members of the sequence.

Algorithm:

STATEMENTS	EXPLANATION
FIBON_NUM1 := 0	FIRST VALUE OF THE SEQUENCE
FIBON_NUM2 : = 1	SECOND VALUE OF THE SEQUENCE
SUM := FIBON_NUM1 + FIBON_NUM2	SUM OF THE LAST 2 VALUES OF THE SEQUENCE
FIBON_NUM1:= FIB_NUM2	PLACE THE NUMBER IN THE SEQUENCE
FIBON_NUM2 := SUM	PLACE THE NUMBER IN THE SEQUENCE
SUM := FIBON_NUM1 + FIBON_NUM2	SUM OF THE LAST 2 VALUES OF THE SEQUENCE
FIBON_NUM1:= FIB_NUM2	PLACE THE NUMBER IN THE SEQUENCE
FIBON_NUM2 := SUM	PLACE THE NUMBER IN THE SEQUENCE
::::::::::::::::::::::::::::::;	:::::::::::::::::::::::::::
SUM := FIBON_NUM1 + FIBON_NUM2	SUM OF THE LAST 2 VALUES OF THE SEQUENCE
FIBON_NUM1:= FIB_NUM2	PLACE THE NUMBER IN THE SEQUENCE
FIBON_NUM2 := SUM	PLACE THE NUMBER IN THE SEQUENCE

The following program will generate the first 6 numbers of the Fibonacci sequence:

0,1,1,2,3,5,8

Program

ASSIGNMENT STATEMENT	FIBON_NUM1	FIBON_NUM2	SUM
FIBON_NUM1 := 0	**0**		
FIBON_NUM2 : = 1	0	**1**	
SUM := FIBON_NUM1 + FIBON_NUM2	0	1	**1**
FIBON_NUM1:= FIB_NUM2	1	1	1
FIBON_NUM2 := SUM	1	**1**	1
SUM := FIBON_NUM1 + FIBON_NUM2	1	1	**2**
FIBON_NUM1:= FIB_NUM2	**1**	1	2
FIBON_NUM2 := SUM	1	2	2
SUM := FIBON_NUM1 + FIBON_NUM2	1	2	**3**
FIBON_NUM1:= FIB_NUM2	**2**	2	3
FIBON_NUM2 := SUM	2	**3**	3

SUM := FIBON_NUM1 + FIBON_NUM2	2	3	**5**
FIBON_NUM1:= FIB_NUM2	**3**	3	5
FIBON_NUM2 := SUM	3	**5**	5
SUM := FIBON_NUM1 + FIBON_NUM2	3	5	**8**
FIBON_NUM1:= FIB_NUM2	**5**	5	8
FIBON_NUM2 := SUM	5	**8**	8

Exercises:

1. Write a program that computes 10!

2. Write a program that will compute a Fibonacci sequence where each number in the sequence is less than 50. ■

3.4 NON-EXECUTABLE STATEMENTS

All assignment statement are executable statements: when the assembler encounter the statement, it will be executed .

There are however, non-executable statements. The first one we will here introduce is the REM statement.

Definition of the REM statement: The form of the rem statement is

REM: comment; where comment can be any words made up of alfa-numeric characters.

Example:

STATEMENTS	X	Y	SUM
REM: THE FOLLOWING PROGRAM WILL ASSIGN NUMBERS TO X, Y AND THEN ADD THEM			
X := 34	**34**		
Y := 100	34	**100**	
SUM := X + Y	34	100	**134**

PROJECT:

Assume the numbers n_1 , n_2 , ... n_m

1. Write an algorithm that will perform iterative multiplication.

2. Using this algorithm write a program to compute n = $34 * 226 * 12 * 44 * 5$

3. Define a^N = a^N

Write an algorithm to perform a^N.

CHAPTER - 4 SIMPLE ALGORITHMS FOR CONVERTING BETWEEN A NUMBER BASE AND THE BASE 10

INTRODUCTION

In this chapter we will show how we write algorithms to convert a number in the base b (b <10) to its corresponding number in the base 10 and from a number base 10 to its corresponding number in the base b (b <10). These algorithms are based on the conversion methods developed in Chapter 2.To help us write these algorithms, we first create a sample program from a specific example. Once the program is written, we will use it as a guide to create the algorithm. In later chapters we will generalize these algorithms.

4.1 An Algorithm to Convert any Positive Integer Number In any Base b < 10 To Its Corresponding Number in the Base 10.

To convert between integer number in any base b to its corresponding number in the base 10, we recall from chapter 1 the following formula:

$n_b = a_n a_{n-1} \ldots a_1 a_0 \Leftrightarrow a_n b^n + a_{n-1} b^{n-1} \ldots + a_1 b + a_0$ base 10 .

Example:

The following program will convert the number 267_8 to its correspond number in the base 10:

$n_{\mathbf{8}} = 267_8 \Leftrightarrow 2*8^2 + 6*8^1 + 7*8^0 = 2(64) + 6(8) + 7 = 183_{10}$

Program

PSEUDO-CODE INSTRUCTIONS	N8	P	A	N10	BASE
N10:= 0				**0**	
N8 := 267	**267**			0	
BASE := 8	267			0	**8**
P := 1		**1**		0	8
A := N8 MOD 10	267	1	**7**	0	8
N10 := N10+ A∗P	267	1	7	**7**	8
N8 := N8 ÷ 10	**26**	1	7	7	8
P := P∗BASE	26	**8**	7	7	8
A := N8 MOD 10	26	8	**6**	7	8
N10 := N10 + A∗P	26	8	6	**55**	8
N8 := N8 ÷ 10	2	8	6	55	8
P := P∗BASE	2	**64**	6	55	8

A := N8 MOD 10	2	8	**2**	55	8
N10 := N10+ A∗P	2		2	**183**	8
N8 := N8 ÷ 10	**0**		2	183	8

Therefore, $267_8 \Leftrightarrow 183_{10}$

Using the above program as a model, the following algorithm will convert any positive integer number in the base b < 10 to its corresponding number in the base 10:

Algorithm:

PSEUDO-CODE INSTRUCTIONS
P := P∗BASE
A := NB MOD 10
N10 := N10+ A∗BASE^K
NB := NB ÷ 10
:::::::::::::::::::

Exercises:

1. Modify the above program to convert the number 5632_8 to the corresponding number in the base 10.

2. Modify the above program to convert the number 1101_2 to the corresponding number in the base 10.

■

4.2 An Algorithm to Convert any Integer Number in the Base 10 to a Corresponding Number in the Base b < 10.

Using the Euclidean division theorem explained in Chapter 1, we now review how to convert numbers in the base 10 to any in the base b < 10.

Step 1: We want to write n in the form: $n = a_n b^n + a_{n-1}b^{n-1} \ldots + a_1 b + a_0$

Step 2: $N = Qb + R = (a_n b^{n-1} + a_{n-1}b^{n-2} \ldots + a_1)b + a_0$

Here, $Q = a_n b^{n-1} + a_{n-1}b^{n-2} \ldots + a_2 b + a_1 = (a_n b^{n-2} + a_{n-1}b^{n-3} \ldots + a_2)b + a_1$ and $R = a_0$

Step 3: Set N = Q.

$Q = Q_1 b + R_1 = (a_n b^{n-2} + a_{n-1}b^{n-3} \ldots + a_2)b + a_1$ where

$Q_1 = a_n b^{n-2} + a_{n-1}b^{n-3} \ldots + a_2$,

$R_1 = a_1$.

Step 4: Continue in this manner, until $Q_n = 0$.

Example:

Convert the following decimal numbers to the specified base.

1. $1625 \Leftrightarrow n_8$

Step 1: $1625 = 203 * 8 + 1$

$a_0 = 1$

Step 2: $203 = 25 * 8 + 3$

$a_1 = 3$

Step 3: $25 = 3 * 8 + 1$

$a_2 = 1$

Step 4: $3 = 0 * 8 + 3$

$a_3 = 3$

Therefore, $n = 3 * 8^3 + 1 * 8^2 + 3 * 8 + 1 \Leftrightarrow n_8 = 3131$

Program

Task: Convert the integer number 1625 to the base 8.

PSEUDO-CODE INSTRUCTIONS	N10	Q	N8	R	BASE	P	TEN
N10 := 1625	**1625**						
BASE := 8	1625				**8**		
TEN := 10	1625						10
P := 10	1625					**10**	10
N8 := 0	1625		**0**		8	10	10
R := N10 MOD BASE	1625		0	**1**	8	10	10
Q:= (N10 - R) ÷ BASE	1625	**203**	0	1	8	10	10
N8:= N8 + R	1625	203	**1**	1	8	10	10
N10 := Q	**203**	203	1	1	8	10	10
R := N10 MOD BASE	203	203	1	**3**	8	10	10
N8 := N8 + R∗P	203	25	**31**	3	8	10	10

P := P∗TEN	203	25	31	3	8	**100**	10
N10 := Q	**25**	25	31	3	8	100	10
R := N10 MOD BASE	25	**25**	31	**1**	8	100	10
Q:= (N10 - R)÷ BASE	25	**3**	31	1	8	100	10
N8 := N8 + R∗P	25	3	**131**	1	8	100	10
P := P∗TEN	25	3	131	1	8	**1000**	10
N10 := Q	**3**	3	131	1	8	1000	10
R := N10 MOD BASE	3	3	131	**3**	8	1000	10
Q:= (N10 - R)÷BASE	3	**0**	131	3	8	1000	10
N8:= N8 + R∗P	3	0	**3131**	3	8	1000	10
N10 := Q	0	0	3131	3	8	1000	10

$1625 \Leftrightarrow 3131_8$

Algorithm:

PSEUDO-CODE INSTRUCTIONS
R := N10 MOD BASE
Q:= (N10 - R)÷ BASE
N8 := N8 + R∗P
P := P∗TEN
N10 := Q
::::::::::::::::

Exercises:

1. Use the above algorithm to write a program to convert the decimal number 2543_{10} to octal.

2. Write an algorithm to convert any decimal number a_1a_0 to the base 2.

■

PROJECT

a. Write a program that will convert the number $2356_7 \Rightarrow n_b$ where b = 9.

b. Write an algorithm that will convert a number n_b to n_c where b, c < 10.

CHAPTER - 5 THE IF-THEN CONDITIONAL STATEMENT

INTRODUCTION

The statements used so far are called unconditional statements. Each statement performs its task without any conditions placed upon them. In this chapter , we will discuss conditional statements. The manner in which these instructions are carried out will depend on various conditions in the programs and algorithms. We begin by defining and explaining conditional expressions.

5.1 Conditional Expressions

We begin with the definition of conditional values:

Definition of Conditional Values: Conditional values take on the value *TRUE* or *FALSE***.** Each conditional value is determined by six relational operators preceded and followed by numeric values or variables.

Definition of Six Relational Operators:

The six relational operators are:

	Operator	**Interpretation**
1.	=	Equality
2.	<>	Inequality
3.	<	Less Than
4.	>	Greater Than
5.	<=	Less than or equal to
6.	>=	Greater than or equal to

Examples:	**Values:**
5 = 2 + 3	*TRUE*
9 <> 3∗3	*FALSE*
4 <= 4	*TRUE*

EXERCISES:

1. Evaluate the following conditional expressions:

a. 3 + 3 = 6 **b.** 8 >= 10 **c.** 7 <> 7

■

Definition of Conditional Expressions: Conditional expressions are conditional values connected by three logical operators.

Definition of the Three Logical Operators:

Logical operators connect conditional expressions and return a value of *TRUE* or *FALSE.* The three logical operators are:

	Operator	Interpretation
1.	**NOT**	NOT conditional expression (*TRUE* if the conditional expression is *FALSE*; *FALSE* if the if the conditional expression is *TRUE).*
2.	**AND**	Conditional expression AND conditional expression (*TRUE* if all the conditional expressions are true).
3.	**OR**	Conditional expression OR conditional expression (*TRUE* if one or more of the conditional expressions are *TRUE*).

Values Returned by Operators

NOT *TRUE*	*FALSE*
NOT *FALSE*	*TRUE*
TRUE AND *TRUE*	TRUE
TRUE AND *FALSE*	*FALSE*
FALSE AND *FALSE*	*FALSE*
TRUE OR *TRUE*	*TRUE*
TRUE OR *FALSE*	*TRUE*
FALSE OR *FALSE*	*FALSE*

EXAMPLES:

CONDITIONAL EXPRESSIONS	**VALUE**
(2 < 3) OR (5 = 7)	*TRUE*
NOT (2 <= 2)	*FALSE*
NOT ((2 > 0) AND (3 <> 2 + 1))	*TRUE*

5.2 THE IF-THEN STATEMENT

Definition of the IF-THEN Statement:

The form of the IF - THEN statement is

IF *conditional expression* THEN

BEGIN
statements
END

If the conditional expression is *TRUE*, then the

BEGIN
statements
END

will be carried out.

If the conditional expression is *FALSE*, then the

BEGIN
statements
END

will NOT be carried out and the program will go to the instruction following the END.

The BEGIN and END statements are non-executable statements.

The

BEGIN
statements
END

is called a compound statement.

EXAMPLES:

1. **PROGRAM**

PSEUDO-CODE INSTRUCTIONS	**X**	**Y**
X := 5	**5**	

IF X = 5 THEN
BEGIN
X := 2$*$X
END
	10	

Y := 2	10	**2**

IF X = Y THEN
BEGIN
X := 2$*$X
END
	10	2

X := 100	**100**	2

The following program will perform the following tasks:

Task 1: Assign three numbers.

Task 2: Count the number of negative numbers.

2. **PROGRAM**

PSEUDO-CODE INSTRUCTIONS	**X1**	**X2**	**X3**	**COUNT**
X1 := 6	**6**			
X2 := -5	6	**-5**		
X3 := -25	6	-5	**-25**	
COUNT := 0	6	-5	-25	**0**

IF X1 < 0 THEN
BEGIN
COUNT := COUNT + 1
END
	6	-5	-25	0

IF X2 < 0 THEN
BEGIN
COUNT := COUNT + 1
END
	6	-5	-25	**1**

IF X3 < 0 THEN
BEGIN
COUNT := COUNT + 1
END
	6	-5	-25	**2**

EXERCISES:

1. Modify the above program so that it performs the following tasks:

Task 1: Assign 4 numbers.

Task 2: Counts the number of positive numbers entered.

Task 3: Add the positive numbers.

2. Modify the above program so that it performs the following tasks:

Task 1: Assign 4 numbers.

Task 2: Multiplies the negative numbers.

■

EXAMPLES:

1. The following algorithm will perform the following task:

Task 1: Find the largest of three numbers

ALGORITHM

PSEUDO-CODE INSTRUCTIONS	**EXPLANATION**
LARGEST := X1	We start by assuming X1 is the largest

IF X2 > LARGEST THEN
BEGIN
LARGEST := X2
END
	If the contents of X2 is larger than the contents of LARGEST replace LARGEST with the contents of X2

IF X3 > LARGEST THEN
BEGIN
LARGEST := X3
END
	If the contents of X3 is larger than the contents of LARGEST replace LARGEST with the contents of X3

The following program will perform the following tasks:

Task 1: Assign 3 numbers

Task 2: Find the largest of these three numbers.

PROGRAM

PSEUDO-CODE INSTRUCTIONS	X1	X2	X3	LARGEST
X1 := 5	**5**			
X2 := 6	5	**6**		
X3 := 10	5	6	**10**	
LARGEST := X1	5	6	10	**5**

IF X2 > LARGEST THEN
BEGIN
LARGEST := X2
END
	5	6	10	**6**

IF X3 > LARGEST THEN
BEGIN
LARGEST := X3
END
	5	6	10	**10**

2. The following program will perform the following tasks:

Task1 : Assign 2 numbers to variables.

Task2: If the number is negative, change it to its absolute value.

PROGRAM

PSEUDO-CODE INSTRUCTIONS	X	Y
X := 23	**23**	
Y := -17	23	**-17**

IF X < 0 THEN
BEGIN
X := -1∗X
END
	23	-17

IF Y < 0 THEN
BEGIN
Y := -1∗Y
END
	23	**17**

EXERCISE:

3. Complete the table below

PSEUDO-CODE INSTRUCTIONS	X	Y	Z
X := 2			
Y := 5			
Z := -4			

IF (X + Y + Z) <> X∗Y THEN
BEGIN
X : = (X - Y)÷X
Y := X + 2∗Y
Z : = X - 2
END			

IF (X - Y + Z) <> X +Y THEN
BEGIN
X : = 2∗(X - Y)÷X
Y := X - 3∗Z
Z : = X + 2
END			

4. Assume X is an integer Explain what the following algorithm does:

IF 2∗(X÷2) = X THEN
BEGIN
X : = 3∗X - 1
END
IF 2∗(X÷2) <> X THEN
BEGIN
X := 2∗X + 1
END

5. Write an algorithm to find the second largest number amongst 4 numbers.

■

5.3: THE IF-THEN- ELSE STATEMENT

Definition of the IF-THEN-ELSE Statement:

The form of the IF-THEN -ELSE statement is:

IF *conditional expression* THEN
  BEGIN
  statements 1
  END
ELSE
  BEGIN
  statements 2
  END

If the conditional expression is *TRUE*, statements 1 following the THEN will be carried out and the program will skip statements 2.

If the conditional expression is *FALSE*, statements 1 following the THEN will not be carried out and the program will execute statements 2.

EXAMPLES:

1. The following program will perform the following tasks:

Task1 : Assign 2 positive integer numbers to variables.

Task2: If the number is even, add a 1 to the number

Task3: If the number is odd, subtract a 1 to the number.

PROGRAM

<table>
<tr><th>PSEUDO-CODE INSTRUCTIONS</th><th>X</th><th>Y</th></tr>
<tr><td>X := 23</td><td>23</td><td></td></tr>
<tr><td>Y := 44</td><td>23</td><td>44</td></tr>
<tr><td rowspan="2">IF 2∗(X÷2) = X THEN
BEGIN
X := X + 1
END
ELSE
X := X - 1
END</td><td>22</td><td>44</td></tr>
<tr><td></td><td></td></tr>
<tr><td rowspan="2">IF 2∗(Y÷2) = Y THEN
BEGIN
Y := Y + 1
END
ELSE
Y := Y - 1
END</td><td>22</td><td>45</td></tr>
<tr><td></td><td></td></tr>
</table>

2. The following program will perform the following tasks:

Task1: Assign two numbers.

Task2: Find the smallest of the two number.

PROGRAM

PSEUDO-CODE INSTRUCTIONS	X	Y	SMALLEST
X := 723	**723**		
Y := 54	723	**54**	
IF X < Y THEN	723	54	
BEGIN	723	54	
SMALLEST := X	723	54	
END	723	54	
ELSE	723	54	
BEGIN	723	54	
SMALLEST : = Y	723	54	**54**
END	723	54	54

PROJECT

The Bubble Sort Algorithm

Perhaps the most important application of computers is the ability to sort data. Data is either sorted in ascending or descending order. For the following 4 numbers, we will state the tasks that show how the bubble sort algorithm is applied using the IF-THEN statement to move the highest remaining numbers to the right:

List of numbers (unsorted).

X1	X2	X3	X4
w	x	y	z

Task 1: Move the highest number to variable X4:

Task 2: Move the next highest number to variable X3:

Task 3: Move the next highest number to variable X2:

Write a program using the bubble sort tasks to sort the numbers below in ascending order.

X1	X2	X3	X4
23	17	3	1

CHAPTER - 6 THE WHILE CONDITIONAL STATEMENT

INTRODUCTION

So far, in our programs, we have not had the ability to perform repetitive operations. In this chapter we will define the WHILE statement which will allow us to make such repetitive operations.

6.1 The While Statement

Definition of the WHILE statement

The form of the WHILE statement is

WHILE *conditional statement*
BEGIN
statements
END

where the statements enclosed in the BEGIN - END are repeated as long as the conditional expression is true. If the conditional statement is false then the statement following the END will be executed.

Examples:

1. The following is an algorithm that will compute the sum of the numbers 1 to R.

Algorithm

PSEUDO-CODE INSTRUCTIONS	EXPLANATION
N := 1	N ⇐ 1
SUM := 0	SUM ⇐ 0
WHILE N <= R	
BEGIN	
SUM := SUM + N	SUM ⇐ SUM + N
N := N + 1	N ⇐ N + 1
END	

Program: **will compute the sum of the numbers 1 to 5.**

PSEUDO-CODE INSTRUCTIONS	CYCLE OF INSTRUCTIONS	SUM	N
N := 1	N := 1		**1**
SUM := 0	SUM := 0	**0**	1
WHILE N <= 5	WHILE N <= 5	0	1
BEGIN	**BEGIN**	0	1
SUM := SUM + N	SUM := SUM + N	**1**	1
N := N + 1	N := N + 1	1	**2**
	SUM := SUM + N	**3**	2
	N := N + 1	3	**3**
	SUM := SUM + N	**6**	3
	N := N + 1	6	4
	SUM := SUM + N	**10**	4
	N := N + 1	10	**5**
	SUM := SUM + N	**15**	5
	N := N + 1	15	**6**
END	**END**	15	6

2. The following algorithm will sum all of the proper divisors of a positive integer number N > 1. A proper divisor d of a integer number N is a number where 1 < d < N and N MOD d = 0. To find all the proper divisors we only need to check all values of d ≤ N÷2.

Algorithm

PSEUDO-CODE INSTRUCTIONS	EXPLANATION
SUM := 0	
D := 2	A DIVISOR
WHILE D <= N÷2	
BEGIN	
IF N MOD D = 0	CHECK TO SEE IF D DIVIDES N
BEGIN	
SUM := SUM + D	IF D DIVIDES N ADD D TO SUM
END	
D := D + 1	
END	

Program: finds and adds the sum of all proper divisors of 18.

PSEUDO-CODE INSTRUCTIONS	CYCLE OF INSTRUCTIONS	N	SUM	D
N:= 18	N := 18	**18**		
SUM := 0	SUM := 0	18	0	
D := 2	D := 2	18	0	**2**
WHILE D <= N÷2	WHILE D <= N÷2	18	0	2
BEGIN	**BEGIN**	18	0	2
IF N MOD D = 0	IF N MOD D = 0	18	0	2
BEGIN	BEGIN	18	0	2
SUM := SUM + D	SUM := SUM + D	18	**2**	2
END	END	18	2	2
D := D + 1	D := D + 1	18	2	**3**
	IF N MOD D = 0	18	2	3
	BEGIN	18	2	3
	SUM := SUM + D	18	**5**	3
	END	18	5	3
	D := D + 1	18	5	**4**
	IF N MOD D = 0	18	5	4
	BEGIN	18	5	4
	SUM := SUM + D	18	5	4
	END	18	5	4
	D := D + 1	18	5	**5**
	IF N MOD D = 0	18	5	5
	BEGIN	18	5	5
	SUM := SUM + D	18	5	5
	END	18	5	5
	D := D + 1	18	5	**6**
	IF N MOD D = 0	18	5	6
	BEGIN	18	5	6
	SUM := SUM + D	18	**11**	6
	END	18	11	6

	D := D + 1	18	11	**7**
	IF N MOD D = 0	18	11	7
	BEGIN	18	11	7
	SUM := SUM + D	18	11	7
	END	18	11	7
	D := D + 1	18	11	**8**
	IF N MOD D = 0	18	11	8
	BEGIN	18	11	8
	SUM := SUM + D	18	11	8
	END	18	11	8
	D := D + 1	18	11	**9**
	IF N MOD D = 0	18	11	9
	BEGIN	18	11	9
	SUM := SUM + D	18	**20**	9
	END	18	20	9
	D := D + 1	18	20	**10**
END	**END**	18	20	10

3. **Length of numbers:**

Definition of the length of a number :

The length of a number is the number of digits that define the number.

Example:

2654 is of length 4

The following algorithm computes the length of any positive integer:

Algorithm

PSEUDO-CODE INSTRUCTIONS	EXPLANATION
COUNT := 0	WILL COUNT # OF DIGITS
WHILE N <> 0	N IS THE POSITIVE INTEGER
BEGIN	
COUNT := COUNT + 1	WILL COUNT # OF DIGITS
N := N÷10	REDUCES THE LENGTH OF N
END	

Program: **will compute the length of the number 431:**

PSEUDO-CODE INSTRUCTIONS	CYCLE OF INSTRUCTIONS	N	COUNT
N: = 431	N := 431	**431**	
COUNT := 0	COUNT := 0	431	**0**
WHILE N <> 0	WHILE N <>0	431	0
BEGIN	**BEGIN**	431	0
COUNT := COUNT + 1	COUNT := COUNT + 1	431	**1**
N := N÷10	N := N÷10	**43**	1
	COUNT := COUNT + 1	43	**2**
	N := N÷10	**4**	2
	COUNT := COUNT + 1	4	**3**
	N := N÷10	0	3
END	**END**	0	3

4. **Adding digits**

1. The following algorithm will sum the digits of an integer $a_n a_{n-1} \ldots a_0$: $a_n + a_{n-1} + \ldots + a_0$.

Algorithm

PSEUDO-CODE INSTRUCTIONS	EXPLANATION
SUM := 0	USED TO ADD THE DIGITS
WHILE N <>0	
BEGIN	
R := N MOD 10	R $\Leftarrow a_k$
SUM := SUM + R	SUM $\Leftarrow a_n + a_{n-1} + \ldots + a_k$.
N := N - R	NUMBER $\Leftarrow a_n \ldots a_r 0$
N := N÷10	
END	

Program: will add the digits of the number 579:

PSEUDO-CODE INSTRUCTIONS	CYCLE OF INSTRUCTIONS	N	R	SUM
N := 579	N := 579	**579**		
SUM := 0	SUM := 0	579		**0**
WHILE N <>0	WHILE N <>0	579		0
BEGIN	**BEGIN**	579		0
R := N MOD10	R := N MOD 10	579	**9**	0
SUM := SUM + R	SUM := SUM + R	579	9	**9**
N := N - R	N := N - R	**570**	9	9
N := N÷10	N := N÷10	**57**	9	9
	R := N MOD10	57	**7**	9
	SUM := SUM + R	57	7	**16**
	N := N - R	**50**	7	16
	N := N÷10	**5**	7	16
	R := N MOD10	5	**5**	16
	SUM := SUM + R	5	5	**21**
	N := N - R	**0**	5	21
END	**END**	0	5	21

■

Exercises:

1. Write an algorithm that performs the following tasks:

Task 1: Finds the proper divisors of a positive integer N

Task 2: Sum the proper divisors.

2. Write an algorithm that will multiply all of the proper divisors of a positive integer number N > 1.

3. A factorial number, written as N!, is defined as

N! = N(N - 1)(N –2)...(2)(1)

where N is a positive integer > 1.

Write a program that will perform the following tasks:

Task 1: Enter a positive integer number N > 1

Task 2: Compute N!

4. For the following program, what is the final value assigned to S?

```
K := 2
S := 0
WHILE K < 10
BEGIN
S := S + 2∗K + 1
K := K + 1
END
```

5. A positive integer greater than 1 is prime if it has no proper divisors. Write a program that will find all prime numbers less than 25.

6. Find the final value R computed in the following program:

```
K := 0
R := 2258 - K∗55
WHILE R > 0
BEGIN
K := K +1
R :=2258 -K∗55
END
R := R + 55
```

7. For the following program below, what is the final value X:

```
K := 1
X : = 2
WHILE K <= 6
BEGIN
X := X + 3
K : = K + 1
END
```

8. For the following program below, what is the final value X:

```
K := 1
X : = 2
WHILE K <= 6
BEGIN
X := X∗ 3
K : = K + 1
END
```

■

PROJECT:

1. A polynomial is defined as $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ where x is any number.

One way of evaluating P(x) without using exponents is to write

$P_n(x) = (.... (((a_n x + a_{n-1})x + a_{n-2}) x + a_{n-2}) x + ...+ a_1)x + a_0$

Example:

$P_3(x) = ((a_3 x + a_2)x + a_1)x + a_0$

$P_6(x) = (((((a_6 x + a_5)x + a_4)x + a_3)x + a_2)x + a_1)x + a_0$

Write an algorithm which will perform $P_n(x)$ using the evaluation of P(x) without using exponents with the following restrictions:

a_k are integers and $0 \le a_k \le 9$.

CHAPTER - 7 COMPUTING NUMBER BASIS WITH ALGORITHMS

INTRODUCTION

In this chapter we will show how to write algorithms and programs that will convert numbers from one base to another. The methods used are based on the conversion formulas that have been developed in several of the previous chapters.

7.1 Writing a Program and Algorithm to Convert numbers in the Base b < 10 to the Base 10:

From chapter 2, we saw that to convert numbers in any base b to its corresponding number in the base 10, we use the following formula :

$N_b = a_n a_{n-1} \ldots a_1 a_0 \Rightarrow a_n b^n + a_{n-1} b^{n-1} \ldots + a_1 b + a_0$

Example:

$N_8 = 4671 \Rightarrow 4*8^3 + 6*8^2 + 7*8 + 1 = 2048 + 384 + 56 + 1 = 2489_{10}$

Program: will convert the number 4671_8 to the base 10.

INSTRUCTIONS	CYCLE OF INSTRUCTIONS	N8	N10	R	P
N8 := 4671	N8 := 4671	**4671**			
P := 0	P : = 0	4671			**0**
N10 := 0	N10 := 0	4671	**0**		0
WHILE N8 <> 0	WHILE N8 <> 0	4671	0		0
BEGIN	**BEGIN**	4671	0		0
R := N8 MOD 10	R := N8 MOD 10	4671	0	**1**	0
N8 := N8 - R	N8 := N8 - R	**4670**	0	1	0
N8:= N8÷10	N8:= N8÷10	**467**	0	1	0
N10 := N10 + R∗8^P	N10 := N10 + R∗8^P	467	**1**	1	0
P := P + 1	P := P + 1	467	1	1	**1**
	R := N8 MOD 10	467	1	**7**	1
	N8 := N8 - R	**460**	1	7	1
	N8:= N8÷10	**46**	1	7	1
	N10 := N10 + R∗8^P	46	**57**	7	1
	P := P + 1	46	57	7	**2**

	R := N8 MOD 10	46	57	**6**	2
	N8:= N8 - R	**40**	57	6	2
	N8:= N8÷10	**4**	57	6	2
	N10 := N10 + R∗8^P	4	**441**	6	2
	P := P + 1	4	441	6	**3**
	R := N8 MOD 10	4	441	**4**	3
	N8:= N8 - R	**0**	441	4	3
	N8:= N8÷10	0	441	4	3
	N10 := N10 + R∗8^P	0	**2489**	4	3
	P := P + 1	0	2489	4	**4**
END	**END**	0	2489	4	4

Algorithm: will convert a number in the base b < 10 to the base 10

INSTRUCTIONS
P := 0
N10 := 0
WHILE N8 <> 0
BEGIN
R := N8 MOD 10
N8 := N - R
N8:= N÷10
N10 := N10 + R∗b^P
P := P + 1
END

Exercise:

1. Write a program and complete the table that will convert the number 231_4 to the base 10 and complete a table as above.

■

7.2 Writing an Algorithm to Convert Numbers in the base 10 to its Corresponding Number in the Base b <10.

Example:

The following method will convert the number 523 to the base 8:

a_0 = 523 mod 8 = 3
523 ÷ 8 = 65

a_1 = 65 mod 8 = 1
65 ÷ 8 = 8

a_2 = 8 mod 8 = 0
8÷8 = 1

a_3 = 1 mod 8 = 1
1÷8 = 0

523 ⇒ 1013_8

The following algorithm will convert any positive integer to any number to the base b < 10.

INSTRUCTIONS	EXPLANATION
K : = 0	
SUM := 0	
WHILE N <> 0	
BEGIN	
A := N MOD B	THE REMAINDER WHICH IS TO BE ADDED
SUM := SUM + A ∗10^K	$a_nb^n + a_{n-1}b^{n-1} \ldots + a_1b + a_0$
N := N÷B	B is the base
K := K + 1	
END	

Program:

The following program will convert the number 523 to the base 8.

INSTRUCTIONS	CYCLE OF INSTRUCTIONS	N10	A	N8	K
N10 := 523	N10 := 523	**523**			
K : = 0	K : = 0	523			**0**
N8 := 0	N8 := 0	523		**0**	0
WHILE N10 <> 0	WHILE N10 <> 0	523		0	0
BEGIN	**BEGIN**	523		0	0
A := N10 MOD 8	A := N10 MOD 8	523	**3**	0	0
N8 := N8 + A∗10^K	N8 := N8 + A∗10^K	523	3	**3**	0
N10 := N10÷8	N10 := N10÷8	**65**	3	3	0
K := K + 1	K := K + 1	65	3	3	**1**
	A := N10 MOD 8	65	**1**	3	1
	N8 := N8 + A∗10^K	65	1	**13**	1
	N10 := N10÷8	**8**	1	13	1
	K := K + 1	8	1	13	**2**
	A := N10 MOD 8	8	**0**	13	2
	N8 := N8 + A∗10^K	8	0	**13**	2
	N10 := N10÷8	**1**	0	13	2
	K := K + 1	1	0	13	**3**
	A := N10 MOD 8	1	**1**	13	3
	N8 := N8 + A∗10^K	1	1	**1013**	3
	N10 := N10÷8	**0**	1	1013	3
	K := K + 1	0	1	1013	**4**
END	**END**	0	1	1013	4

Exercises:

1. Write a program and complete the table that converts the decimal number 25 to base 2.

2. Write a program and complete the table that will print the first 100 numbers in the base 8.

■

PROJECT

Write a program that will convert the number 23_8 to the base 5.

CHAPTER 8 - RINGS AND MODULAR ARITHMETIC

INTRODUCTION

Modular arithmetic plays a major role when doing arithmetic in assembly language. We will see in the next Chapter that the number systems we will be working with are not infinite in number. To perform arithmetic on finite systems, we need to use modular arithmetic. We start with the definition of rings.

8.1 Rings

Definition of a ring:

A ring *R* is a set of numbers having two binary operations: addition $\oplus$ and multiplication $\otimes$ with the following rules:

Rule 1: Closure under addition.

Rule 2: Closure under multiplication.

Rule 3: Contains an additive identity.

Rule 4: Contains a multiplicative identity.

Rule 5: For every number n there is an additive inverse ~n.

Definition of the above rules:

Rule 1: If n, m are numbers in *R*, then c = n $\oplus$ m is in *R*.

Rule 2: If n, m are numbers in *R*, then c = n$\otimes$m is in *R*.

Rule 3: Contains a number Θ in R, where for every number n in *R*, n$\oplus\Theta$ = n.

Rule 4: Contains a number 1 in R, where for every number n in *R*, n$\otimes$1 = n.

Rule 5: For every number n in *R*, there is a number ~n in R where n $\oplus$~n = Θ.

There are two general type of rings: infinite and finite.

Example of an infinite ring:

1. All integers: *R* = {0, 1, -1, 2, -2, 3, -3, ... }

Rule 1: Let $\oplus$ = +. The sum of 2 integer numbers is an integer number.

Rule 2: Let $\otimes$ = $*$. The product 2 integer numbers is an integer number.

Rule 3: Let Θ = 0. If n is a integer number then n + 0 = n.

Rule 4: The number 1 is an integer and n ∗ 1 = n

Rule 5: Assume n is in *R* . Let ~n = -n. Therefore, n + -n = 0.

Important: For rings, there is no subtraction operation.

Example of a finite ring :

The following is a well known finite ring: the hourly clock time:
R = {1,2,3,4,5,6,7,8,9,10,11,12}

For addition or multiplication, we use the traditional system. For example:
1⊕5 = 6, 2⊕11 = 1, 3 ⊕12 = 3, 5 ⊗2 = 10, 6 ⊗3 = 6, etc.

Now we show that the *R* is a ring, by verifying the 5 rules:

Rule 1: If n, m are numbers in *R*, then c = n ⊕ m is in *R*.

Rule 2: If n, m are numbers in *R*, then c = n⊗m is in *R*.

Rule 3: Contains a number Θ = 12where for every number n in *R*, n⊕12 = n.

Rule 4: Contains a number 1 where for every number n in *R*, n⊗1 = n.

Clearly this rule is correct.

Rule 5: For every number n in *R*, there is a number ~n where n ⊕~n = 12.

To verify this rule, we the following table shows that every number of *R* has an additive inverse: n ⊕~n = 12.

hour	1	2	3	4	5	6	7	8	9	10	11	12
~ hour	11	10	9	8	7	6	5	4	3	2	1	12
hour ⊕ ~ hour	12	12	12	12	12	12	12	12	12	12	12	12

Exercises:

1. Assume *R* is clock time. Simplify the following:

a. 7⊕8⊕~7⊕11⊕~4.

b. 2⊗(6⊕ ~10)

c. ~11⊗[(2 ⊗~11) ⊗(11⊕ ~9)]

2. Assume R is military time: R = {1,2,3,..., 24}

a. $7 \oplus 18 \oplus \sim 7 \oplus 21 \oplus \sim 23$.

b. $22 \otimes (16 \oplus \sim 10)$

c. $\sim 21 \otimes [(2 \otimes \sim 21) \otimes (11 \oplus \sim 19)]$

3. Show that the set R = {0, 1,-1, 2, -2, 4, -4, 6, -6, ...± 2n, ...} is not ring.

4. Show that the set R = {0, 1,3, -3, 5, - 7, ..., ± 2n + 1} is not a ring.

5. Assume R = {0, 1,2, -2, 3, -3, 4, -4, ...}. Define $\oplus$ and $\otimes$ are defined under the following rules:

R1. : $n \oplus m = n + m + 2$.

R2: $n \otimes m = n * m$

a. Find Θ.

b. For n in R find $\sim n$, the additive inverse of n.

c. Show R is a ring.

■

8.2: The Finite Ring R

For assembly language, the most important set of numbers are

R = {0,1,2,3,...,N - 1}, where N > 1.

We want R to be a ring. To do this we need to define operations of addition and multiplication:

Definition of addition $a \oplus b$: If a, b are members of R, then $a \oplus b = (a + b) \bmod N$.

Definition of multiplication $a \otimes b$: If a, b are members of R, then $a \otimes b = (a * b) \bmod N$.

Note: The mod operator is defined in chapter 3.

Examples:

R = {0,1,2,3,4,5,6,7}.

$5 \oplus 7 = (5 + 7)\bmod(8) = 12\bmod(8) = 4$

$5 \otimes 6 = (5 * 6)\bmod(8) = 30\bmod(8) = 6$

$2 \oplus 5 = (2 + 5) \bmod(8) = 7 \bmod(8) = 7$

$(6 \otimes 7) \oplus 6 = [(42) \bmod(8)] \oplus 6 = 0$

Exercises:

1. For $R = \{0,1,2,3,4\}$, simplify:

a. $4 \otimes 4$

b. $[(4 \oplus 2) \otimes 4 \oplus 4] \otimes 3$

c. $3 \otimes (3 \oplus 4)$

2. For $R = \{0,1,2,..., 7\}$, verify if the following are true:

a. $6 \otimes (7 \oplus 5) = (6 \otimes 7) \oplus (6 \otimes 5)$

b. $(4 \otimes 3) \otimes 7 = 4 \otimes (3 \otimes 7)$

c. $(4 \oplus 3) \oplus 7 = 4 \oplus (3 \oplus 7)$

3. For $R = \{0,1,2,..., N-1\}$, what is the additive identity? What is the multiplicative identity?

4. For $R = \{0,1,2,..., 15\}$, find the additive identity of each of its numbers.

5. For R = {0,1}, find the additive identity of each of its numbers.

6. Show that R = {0,1,2,..., N} is a ring under the operations of $a \oplus b$, $a \oplus b$.

8.3 Subtraction for *R*

How then do we subtract 2 numbers in *R* ? We accomplish this using the following definition:

Definition of subtraction $a \ominus b$ for a, b in *R*:

$a \ominus b = (a + {\sim}b) \bmod(N)$, where

a and $\sim b$ are values in the ring $R_N = \{0,1,2,..., N - 1\}$

Examples:

$6 \ominus 3 = (6 + {\sim}3) \bmod(8) = (6 + 5) \bmod(8) = 11 \bmod(8) = 3$

$5 \ominus 7 = (5 + {\sim}7) \bmod(8) = (5 + 1) \bmod(8) = 6 \bmod(8) = 6$

${\sim}4 \ominus 3 = ({\sim}4 + {\sim}3) \bmod(8) = (4 + 5) \bmod(8) = 9 \bmod(8) = 1$

Exercises:

1. Are the following true or false for numbers in R_N. Show examples of each.

a. $\sim\sim a = a$?

b. $\sim(a \sim b) = b \sim a$

c. $\sim a + \sim b = \sim(a + b)$
■

8.4 Rings in Different Bases

So far we have built our finite rings in the decimal number system. We will now define binary and hexadecimal rings which plays an important role in the assembly language:

Definition of a binary finite ring: Assume we are in a binary number system. We define

$R_2 = \{0,1,10,11,100,..., N\}$

Examples:

a. $R_2 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

b. $R_2 = \{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010,1011,1100,1101,1110, 1111\}$

Definition of a hexadecimal finite ring: Assume we are in a hexadecimal number system. We define

$R_{16} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11..,. N\}$.

Examples:

a. $R_{16} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$

b. $R_{16} =$

$\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,15,16,17,18,19, 1A,1B,1C,1D,1E,1F\}$

Exercises:

1. For the finite ring $R_{16} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ find:

a. $9 \oplus 8$

b. $5 \otimes B$

2. For the finite ring $R_2 = \{00000000, 00000001, ..., 11111111\}$ find:

a. $10010110 \oplus 01010111$

b. $11010111 \ominus 10101010$

c. $11010111 \otimes 10101010$

Modular arithmetic in the base b.

As in the decimal number system we define

$r_b = a_b \bmod(n_b) =$ where

$a_b = q_b * n_b + r_b$

and

$r_b < n$.

To easily perform such modular arithmetic, we will use the following results:

$r_b = (a_b) \bmod n_b \Leftrightarrow r_{10} = (a_{10}) \bmod n_{10}$

Similarly we have

$a_b \oplus c_b = (a_b + c_b) \bmod n_b \Leftrightarrow (a_{10} + c_{10}) \bmod n_{10}$

$a_b \otimes c_b = (a_b * c_b) \bmod n_b \Leftrightarrow (a_{10} * c_{10}) \bmod n_{10}$

Examples:

1. Octal numbers:

a. $762_8 \bmod (52_8) \Leftrightarrow 498_{10} \bmod (42_{10}) = 36_{10} \Leftrightarrow 44_8$

Therefore, $762_8 \bmod (52_8) = 44_8$

b. $(771_8 + 236_8) \bmod (106_8) \Leftrightarrow (505_{10} + 158_{10}) \bmod (70_{10}) = (663_{10}) \bmod (70_{10}) = 33_{10} \Leftrightarrow 41_8$

Therefore, $(771_8 + 236_8) \bmod (106_8) = 41_8$

c. $(771_8 * 236_8) \bmod (106_8) \Leftrightarrow (505_{10} * 158_{10}) \bmod (70_{10}) = (79790_{10}) \bmod (70_{10}) = 60_{10} \Leftrightarrow 74_8$

Therefore, $(771_8 * 236_8) \bmod (106_8) = 74_8$

2. Binary numbers:

a. 100110_2 mod (1101_2) $\Leftrightarrow$ 38_{10} mod (13_{10}) = 12_{10} $\Leftrightarrow$ 1100_2

Therefore, 100110_2 mod (1101_2) == 1100_2

b. (110111_2 + 11011_2) mod (1111_2) $\Leftrightarrow$ (55_{10} + 27_{10})mod (15_{10}) = (82_{10}) mod (15_{10}) =
7_{10} $\Leftrightarrow$ 111_2

Therefore, (110111_2 + 11011_2) mod (1111_2) = 111_2 .

c. (110111_2 $*11011_2$) mod (1111_2) $\Leftrightarrow$ (55_{10} $*$ 27_{10})mod (15_{10}) = (1485_{10}) mod (15_{10}) =
0_{10} $\Leftrightarrow$ 0_2

Therefore, (110111_2 $*11011_2$) mod (1111_2) = 0.

3. Hexadecimal numbers:

a. $9A23F_{16}$ mod (AD_{16}) $\Leftrightarrow$ 631359_{10} mod (173_{10}) = 82_{10} $\Leftrightarrow$ 52_{16}

Therefore, $9A23F_{16}$ mod (AD_{16}) = 52_{16}

b. ($AC2301F_{16}$ + $27DD1_{16}$) mod (AD_{16}) $\Leftrightarrow$ (180498463_{10} +163281_{10})mod (173_{10}) = (180661744_{10}) mod (173_{10})

93_{10} $\Leftrightarrow$ $5D_{16}$

Therefore, ($AC2301F_{16}$ + $27DD1_{16}$) mod (AD_{16}) = $5D_{16}$.

c. ($AC2301F_{16}*27DD1_{16}$) mod (AD_{16}) $\Leftrightarrow$ (180498463_{10} $*163281_{10}$)mod (173_{10}) =

(29471969537103_{10}) mod (173_{10}) = 135_{10} $\Leftrightarrow$ 87_{16}

Therefore, ($AC2301F_{16}*27DD1_{16}$) mod (AD_{16}) = 87_{16}

Exercises:

Simplify the following:

a. 251_6 mod($301F_6$) b. (235432 + 251_6)mod($301F_6$) c. (235432$*$ 251_6)mod($301F_6$)

The additive inverse of a number

Recall the definition of an additive inverse:

Definition of an additive inverse: Assume a is a number in a ring. The additive inverse is a number ~ a in the ring where ~a $\oplus$ a = 0 .

Example:

1. Assume we have the following ring:

R = {0,1,2,3,4,5,6, 7}

a. If a = 5, then ~ a = 3

since

5⊕- 5 =5⊕3 = 8 Mod 8 = 0.

8.5 The Additive Inverse of Numbers for the Rings R_b = {0...0, 0...1, 0...2, ..., $\beta_1\beta_2$ ..., β_n}

Definition of $\beta_1\beta_2$ ..., β_n :The number is a positive integer $\beta_1\beta_2$ ...β_n where the digits are all equal and β_k = b - 1.

Examples:

a. R_{10} = {0000, 0001, 0002, 0003, 0004,..., 9999}

b. R_2 = {0000, 0001, 00010, 0011, 0100, ..., 1111}

c. R_8 = {000, 001,002,003,004,..., 777}

d. R_{16} = {00, 01,02,03,04, ... , FF}

For these types of rings, we can easily compute the additive inverse of a number by taking the compliment of a number. The following is the definition of a compliment of a number:

Definition of a complement of a number $a' = a_1a_2a_3 \ldots a_n'$ *in R:*

Let R = {0...0,0...1,0...2 ,..., βββ...β}. The compliment of a number a in R is

$a' = a_1'a_2'a_3' \ldots a_n'$

where $a_k' = \beta - a_k$

The following tables give the digit compliments of important number systems for the assembly language:

binary

a_k	0	1
a_k'	1	0
$a_k + a_k'$	1	1

decimal

a_k	0	1	2	3	4	5	6	**7**	8	9
a_k'	9	8	7	6	5	4	3	2	1	0
$a_k + a_k'$	9	9	9	9	9	9	9	9	9	9

octal

a_k	0	1	2	3	4	5	6	**7**
a_k'	7	6	5	4	3	2	1	0
$a_k + a_k'$	7	7	7	7	7	7	**7**	7

hexadecimal

a_k	0	1	2	3	4	5	6	**7**	8	9	A	B	C	D	E	F
a_k'	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
$a_k + a_k'$	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Examples:

a. R_{10} = {00,01,02,03,..., 99}

25′ = 74

b. R_8 = {00,01,02,03,..., 77}

42′ = 35

c. R_{16} = {000,001,002,003,..., FFF}

0C4′ = F3B

d. R_2 = {000, 001, 010, 011, 100,101,110,111}

101′ = 010

The following rule, can be useful to compute the inverse of a number:

Rule: ~a = a′ + 1

Examples:

1. R_2 = {000000,000001, ..., 111111}

$a = \ 100101_2$

$a' = 011010_2$

$\sim 100101_2 = 011010_2 + 1 = 011011_2$

$a \oplus \sim a = (100101 + 011010 + 1) \bmod (1000000) = (111111 + 1) \bmod (1000000) = (1000000) \bmod (1000000) = 0$

2. $R_{16} = \{00,01,02,03,..., FF\}$

$a = 9C$

$9C\,' = 63$

$\sim 9C = 63 + 1 = 64$

$9C \oplus 64 = (9C + 63 + 1) \bmod 100 = (FF + 1) \bmod 100 = 0$

Question: Why doesn't the assembly language allow us to do normal subtraction? It is not the assembly language that prevents this, it is the way the computer circuitry is designed. To allow subtraction, would require to double the circuitry. Since subtraction can be accomplished by the adding the additive inverse, the design of computers are more simple and faster. Also, since only binary numbers are used to represent numbers, the complement of a binary number is simply changing the 0s' to 1s' and the 1s' to 0s' . Therefore, the additive inverse of a binary number is the complement plus 1.

Exercises:

1. For each of the following binary numbers, find their additive inverses:

a. 10011100110 b. 11011011 c. 10101010

2. For the octal ring $R_8 = \{0,1,2,3,4,5,6,7,10,...,77\}$, compute the following:

a. $43 \ominus 56$ b. $55 \ominus 55$ c. $\sim 10 \ominus 56$ d. $\sim 43 \ominus \sim 56$

3. Assume we have the hexadecimal ring: $R_{16} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,...,FF\}$. Find the following:

a. $\sim AC$ b. $A9 \ominus \sim 55$ c. $\sim 10 \ominus 5E$ d. c. $\sim 10 \ominus \sim 5E$

■

Modular arithmetic for rings $R_b = \{0...0, 0...1, 0...2, ..., \beta_1\beta_2 ... \beta_n\}$, $\beta_k = b - 1$.

In this section, we will study the modular arithmetic $a_b \mod(\beta_1\beta_2 ... \beta_n + 1)$.

First observe that $\beta_1\beta_2 ... \beta_n + 1 = {10_b}^n$

Examples:

1. $\mathbf{77_8} + 1 = 100_8 = {10_8}^2$
2. $FFFFF_{16} + 1 = 100000_{16} = {10_{16}}^5$
3. $11111111_2 + 1 = 100000000_2 = {10_2}^8$

Therefore, for R_b , the following examples will show how to evaluate

$a_b \mod(\beta_1\beta_2 ... \beta_n + 1) = a_b \mod({10_b}^n)$.

Examples:

1. $253_8 \mod(77 + 1) = 253_8 \mod({10_8}^2) = 253_8 \mod(100_8)$

Solution:

$253_8 = 2 * 100_8 + 53_8$

Therefore,

$253_8 \mod(77_8 + 1) = 53_8$

2. $AC23D_{16} \mod(FFF + 1) = AC23D_{16} \mod(1000_{16})$

Solution:

$AC23D_{16} = AC_{16} * 1000_{16} + 23D_{16}$

Therefore,

$AC23D_{16} \mod(FFF_{16} + 1) = 23D_{16}$

3. $111001101_2 \mod(1111_2 + 1) = 111000101_2 \mod(10000_2)$

Solution:

$111001101_2 = 11100_2 * 10000_2 + 1101_2$

Therefore, $111001101_2 \mod(1111_2 + 1) = 1101_2$

From these examples the following formula evolves:

$(a_n a_{n-1} a_{n-2} \ldots a_1\ a_0)_b\ = (a_n a_{n-1 \ldots} a_{k+1})10^k\ + (\ a_k a_{k-1-1} \ldots a_1\ a_0)_b$

Therefore, $(a_n a_{n-1} a_{n-2} \ldots a_1\ a_0)_b \bmod(10^k)\ =\ (\ a_k a_{k-1-1} \ldots a_1\ a_0)_b$

8.6 Special Binary Rings For Assembly Language

In assembly language we will need to be concerned about following three special binary rings:

THE BYTE RING (8 bits)	**THE WORD RING (16 bits)**	**THE DWORD (32 bits)**
00000000	0000000000000000	00000000000000000000000000000000
00000001	0000000000000001	00000000000000000000000000000001
00000010	0000000000000010	00000000000000000000000000000010
00000011	0000000000000011	00000000000000000000000000000011
00000100	0000000000000100	00000000000000000000000000000100
00000101	0000000000000101	00000000000000000000000000000101
00000110	0000000000000110	00000000000000000000000000000110
00000111	0000000000000111	00000000000000000000000000000111
00001000	0000000000001000	00000000000000000000000000001000
:::::::::::::::	:::::::::::::::::::::::::::	:::
11111111	1111111111111111	11111111111111111111111111111111

To better understand these three rings, we will now study them as equivalent rings in the base 10:

THE BYTE RING (8 bits)	**THE WORD RING (16 bits)**	**THE DWORD (32 bits)**
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
:::::::::::::::	:::::::::::::::	:::::::::::::::
255	65,535	4,294,967,295

Exercises:

1. Convert the above each of the binary table to hexadecimal.

2. Assume we have a binary number $n_2 = a_1\ a_2\ a_3\ \ \ldots a_n$ = 111...1, consisting of n, 1 bits.

Show $n_2 \Rightarrow N_{10} = 2^n - 1$

Hint: Show $(2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots 2 + 1)(2 - 1) = 2^n - 1$

3. Using exercise 2 , show that

a. the largest decimal number in the byte ring is 255.

b. the largest decimal number in the word ring is 65,535.

c. the largest decimal number in the dword ring is 4,294,967,295.
■

Modular arithmetic for the byte ring (in decimal).

The modulus formula is r = m mod (256)

Examples:

1. $5 \oplus 254 = (5 + 254) \bmod(256) = 259 \bmod(256) = 3$

2. $164 \otimes 21 = (164 * 21) \bmod(256) = 5{,}442{,}444 \bmod(256) = 140$

$100 \ominus 253 = (100 - 253) \bmod(256) = -153 \bmod(256) = 103 \bmod(256) = 103$

Exercises:

1. Compute:

a. $122 \oplus 122$ **b.** $162 \otimes 31$ c. $175 \otimes 222 \otimes 13$ d**.** $(175 \oplus 222) \otimes 13$

2. Find the additive inverse for the following:

a. 214 b. 0 c. 128
■

Modular arithmetic for the word ring (in decimal).

The modulus formula is r = m mod (65,536)

1. $5 \oplus 254 = (5 + 254) \bmod(65{,}536) = 259 \bmod(65{,}536) = 259$

2. $23{,}641 \otimes 500 = (23{,}641 * 500) \bmod(65{,}536) = 11{,}820{,}500 \bmod(65{,}536) = 24{,}020$

Exercises:

1. Compute:

a. $122 \oplus 122$ b. $162 \otimes 31$ c. $175 \otimes 222 \otimes 13$ d. $(175 \oplus 222) \otimes 13$

2. Find the additive inverse for the following:

a. 214 b. 0 **c.** 128
■

Modular arithmetic for the dword ring (in decimal).

The modulus formula is r = m mod (4,294,967,296)

1. 3,000,000,000,$\oplus$ 4,254,256,111 = (7,254,256,111) mod(4,294,967,296) = 2,959,288,815

2. 2,323,641$\otimes$3,200,241,001 = (2,323,641$\otimes$3,200,241,001) mod(4,294,967,296) =

465,288,199,804,641mod(4,294,967,296) = 1,507,727,073

Exercises:

1. Compute:

a. 127,567,222$\oplus$ 2,123,567,222 b. 127,567,222$\otimes$ 2,123,567,222 c. 175$\otimes$222$\otimes$13,000

d. (175$\oplus$222)$\otimes$13

2. Find the additive inverse for the following:

a. 214 b. 0 c. 128

3. Convert the decimal number - 202 to a binary number in a

a. byte ring b. word ring c. dword ring .
■

8.7 Ordered Relations of Rings

Definition of an ordered relationship of a ring:

Assume we have the following ring R_{10} = {0,1,2,..., N - 1} containing N numbers. A set of ordered pairs of these numbers is defined as {(a,b)}, where a and b are numbers in R and the order is defined by some given rule. Such a set of ordered pairs of numbers is defined as an ordered relationship of the ring R_{10}.

Examples:

R = {0,1,2,3,4}

A natural set of ordered pairs

Definition: A natural set of ordered pairs is where the numbers (a,b) are defined in the their order of magnitude:

A natural set of ordered pairs for ring R_{10} would be

{(0,0) (0,1), (0,2), (0,3), (0,4), (1,1), (1,2), (1,3), (1,4), (2,2), (2,3),(2,4), (3,3), (3,4), (4,4) }

Note in this example the ordered pair is defined as (a, b) where b is greater than a or b is equal to a.

For all ordered pairs of this type we will use the following symbols:

a equals to b: $a = b$

b is greater than a or a is less than b: $a < b$

These symbols will be used to describe the ordered pair relationships of the number in the ring:
The pair (a, a) will be written as $a = a$.

If $a \neq b$, the pair (a, b) will be written as $a < b$.

For example, the pair (2,2) will be written as $2 = 2$ but the pair (3,4) will be written as $3 < 4$.

Therefore we have $0 < 1 < 2 < 3 < 4$

Other sets of ordered pairs:

The following is another example of a set of ordered pairs of the ring R:

{(4,0), (4,1), (4,2), (4,3), (4,4), (3,0), (3,1), (3,2), (3,3), (2,0), (2,1), (2,2),(1,0), (1,1),(0,0)}

Using our special symbols

=, <

we will still have (a,b) where

$a = a$, and $a < b$ where $a \neq b$.

Therefore for our ordered pair the following will hold true:

4 < 0, 4 < 1, 4 < 2, 4 < 3, 4 = 4, 3 < 0, 3 < 1, 3 < 2, 3 = 3, 2 < 0, 2 < 1, 2 = 2),1 < 0, 1 < 1,0 =)

Laws of ordered relations

For the above special sets of ordered pairs, the following two laws apply:

1. *Reflexive law:* For each number a in the ring, $a = a$.

2. *Transitive law:* If a < b and b < c then a < c.

Exercises:

1. For the ring R = {0,1,2,3,4}, using the special symbols, write out the relations of the ordered pair:

{(0,0), (1,1), (1,0), (2,2), (2,1), (2,0), (3,3), (3,2), (3,1), (3,0), (4,4), (4,3),(4,2), (4,1),(4,0)}

2. Show for the ring R = {0,1,2,3,4}, that the above 2 laws hold for both the natural and the ordered pairs:

{(0,0), (1,1), (1,0), (2,2), (2,1), (2,0), (3,3), (3,2), (3,1), (3,0), (4,4), (4,3),(4,2), (4,1),(4,0)}
■

8.8 Special Ordering of Rings For Assembly Language

In assembly language we will need to be concerned about following three special binary rings: Bytes, words and dwords. For each of these rings the assembly language will recognize two types of ordered pairs:

1. The natural order pairs

2. The signed order pairs.

For demonstration purposes all the rings will be represented as decimal integer numbers.

Ordered pairs for the byte ring

In decimal we will write the byte ring as R = {0,1,2,3, ..., 255}.

The natural order:

The natural order for R is {(0,0), (0,1), ..,(0,255), (1,1), (1,2), ..., (1,255), (2,2), (2,3) ..., (2,255), ... (255,255)}

which can be written as

0	1	2	3	4	5	6	-----	251	252	253	254	255

where the order pairs can be seen as a list of numbers in their increasing order:

0 < 1 < 2 < 3 < ... < 254 < 255

For an example, we can write

5 < 214, 211 < 244, 255 = 255

The signed order:

128	129	...	253	254	255	0	1	2	3	...	126	127

where the order pairs can be seen as a list of numbers in their increasing order:

128 < 129 < 130 < ... < 255 < 0 < 1 < 2 < ... < 126 < 127

The following table give in the second row the “traditional” representation of additive inverse of the numbers

0, 1, 2, 3,... 126, 127.

128	129	...	253	254	255	0	1	2	3	...	126	127
-128	-127	....	-3	-2	-1	0	1	2	3	...	126	127

The next table gives the binary representation:

128	129	...	254	255	0	1	2	---	126	127
10000000	10000001	...	11111110	11111111	00000000	00000001	00000010	--- .	01111110	01111111

Therefore, sticking to our rules on ordered relationships we have for example:

251 = 251,

251 < 0

5 < 122

254 < 15

Therefore, in decimal we have

128 < 129 < 130 < ... < 254 < 255 < 0 < 1 < 2 < 3 < ... < 126 < 127 .

Exercises:

1. Construct a natural order table for the values the word ring.

2. Construct a signed order table for the values of the word ring.

3. Construct a natural order table for the values the dword ring.

4. Construct a signed order table for the values of the dword ring.

PROJECT

Assume we want a program that will perform arithmetic in finite ring R = {0,1,2,..., N} base 10.

Write a program that given any two numbers x, y in R will perform $x \oplus y$, $x \ominus y$, $x \otimes y$.

CHAPTER 9 - ASSEMBLY LANGUAGE BASICS

INTRODUCTION

A close examination of our pseudo-language programs reveals that such programs are made up of four major components: numbers, arithmetic expressions, variables, and instructions. In this chapter we will study at an elementary level how these four components are defined and used in the assembly language. Also for this chapter, as well as several subsequent chapters, all numbers will be integers.

9.1 Data Types of Integer Binary Numbers

First we must understand that when programming in assembly language all numbers are converted by the assembler into binary numbers of a well defined data type. Most assemblers will only recognize the following three data types of binary integer numbers:

1. Eight bit binary numbers.

2. Sixteen bit binary numbers.

3. Thirty- two bit binary numbers.

Special names are given to each of these data types: bytes, words, and dwords.

Definition: A byte is a eight bit binary number.

Definition: A word is a sixteen bit binary number.

Definition: A dword (i.e. double word) is a thirty-two bit binary number.

Important: All numbers must be defined as a given data type by the programmer in order for the assembler to process the program.

Examples:

1. byte (8 bits):
a.

0	1	1	0	1	1	1	0

b.

0	0	0	0	0	1	0	1

2. word: (16 bits)

a.

1 0 0 1	1 1 0 0	0 0 1 0	1 0 1 0

b.

0 0 1 1	0 0 1 1	1 0 0 1	0 1 0 1

3. dword:(32 bits)

a.

1 0 0 1	1 0 1 0	1 1 1 0	0 1 1 1	1 0 1 0	1 0 1 1	0 1 1 1	0 0 1 1

b.

0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 0 1 1	0 1 1 0

Exercises:

For the examples above,

1. find the binary complements.

2. find the binary additive inverses.

3. find the equivalent numbers in the hexadecimal base.

■

9.2 Other Integers

Besides binary numbers, the assembler recognize three other number basis: decimal, octal, hexadecimal . Except for the decimal numbers, all numbers must be followed by the following suffixes:

NUMBER SYSTEM	BASE	SUFFIX
hexadecimal	16	h
binary	2	b
octal	8	o
decimal	10	none (or d)

Examples:

a. e239ch **b.** 101101b **c.** 23771o **d.** 3499

Exercises:

1. For the examples above, convert each to decimal.

2. Which of the following are valid numbers:

a. 2397h **b.** 1011011o **c.** 01101101h
■

9.3 Variables

As in the pseudo language code, variables are names that will contain numbers. The following rules are required when defining a variable name in assembly language:

1. The first character of the variable name must begin with either a letter (A, B, ..., Z, a, b, ..., z), a underscore (_), @, ? or $.

The other characters can also be digits.

2. They are not case- sensitive.

3. The maximum number of characters in the name is 247.

Examples:

a. apple_of_my_eye b. S23x c. $money2 d. hdachslager@ivc **e.** X1_or_X2

f. X g. y h. $124 I. _@yahoo j. z2

Variable types

As in binary numbers, variables are of three data types: BYTE, WORD, DWORD.

We will identify the data types as follows:

variable name byte

variable name word

variable name dword

Examples:

1. x byte

2. Number word

3. Large_Number_dword

Exercises:

Which of the following are legal variable names:

a. _apple_of_my_eye **b.** S_23x **c.** $money2& **d.** hdachslager@ivc.edu **e.** 1XorX2

9.4 Assigning Integers to Variables

There are two ways to assign an integer to a variable:

- By initialing the variable when the variable's data type is defined.
- By using the *mov* assignment instruction.

Initialing the variable

To initialize the variable we use the form:

variable name *data type* *integer*

Examples:

1.
x byte 1101101b

0110	1101

2.
y byte 5Ah

0101	1010

3.
z byte 250

1111	1010

4.
x word 10101101101b

0000	0101	0110	1101

.

y word 1D5Ah

0001	1101	0101	1010

6.
z word 65500

1111	1111	1101	1100

7.

x dword 11010111101010100011010110110 1b

0011	0101	1110	1010	1000	1101	0110	1101

8.
y dword 2ABC1D5Ah

0010	1010	1011	1100	0001	1101	0101	1010

9.
z dword 4294967216

1111	1111	1111	1111	1111	1111	1011	0000

Exercises:

1. Verify that the conversions to binary are correct for examples 1 - 9.

2. For the above 9 examples above, convert each data type to their hexadecimal values.
■

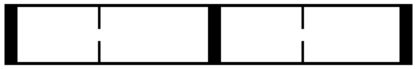
Defining a variable without initialization

If you do not wish to initialize the variable, use the symbol ? in place of the integer.

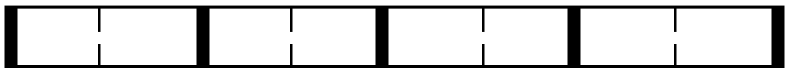
Examples:

x byte ?

y word ?



z dword ?



Using the *mov* assignment instruction

The *mov* instruction is of the general form:

mov *destination, source*

where the destination must be a variable or register (discussed below) and the source can be an integer, variable or register.

The mov instruction can be used in five ways:

MOVE INSTRUCTION	ORDER OF ASSIGNMENT
mov *register1, register2*	*register1* ← *register2*
mov *register, variable*	*register* ← *variable*
mov *variable, register*	*variable* ← *register*
mov *register, integer*	*register* ← *integer*
mov *variable, integer*	*variable* ← *integer*

Note: The definition of registers are defined in the next section.

Important: You cannot use the mov instruction to move data contained in one variable directly into another variable: mov *variable, variable* **is not a legal statement.**

The following rules apply:

Rule 1. The destination and the source <u>cannot</u> both be variables.

Rule 2. If the source is a variable, then both the destination and the source must be of the <u>same data type</u>.

Rule 3: All hexadecimal numbers must begin with a digit (0 - 9)

Examples:

1.
x byte ?
 mov x, 1011010b

0101	1010

2.
z byte ?
mov z, 8Fh

1000	1111

3.
y byte ?
mov y, 252

1111	1100

4.
x word ?
 mov x, 10011001011010b

0010	0110	0101	1010

5.
z word ?
mov z, 1D8Fh

0001	1101	1000	1111

6.
y word ?
mov y, 65010

1111	1101	1111	0010

7.
x dword ?
 mov x, 10101110101010011001011010b

0000	0010	1011	1010	1010	0110	0101	1010

8.
z dword ?
mov z, 0ACEF1D8Fh

1010	1100	1110	1111	0001	1101	1000	1111

9.
y dword ?
mov y, 4194967096

1111	1010	0000	1010	0001	1110	0011	1000

Note:

• mov x, A23F h is not valid by Rule 3. However mov x, 0A23F h is valid.

• mov x,y is not valid by Rule 1.

Exercises:

1. Verify that the conversions to binary are correct for examples 1 - 9.

2. For the above 9 examples above, convert each data type to their hexadecimal values.
■

9.5 Registers

Registers are used by the programmer for storing data and performing arithmetic operations.

There are three types of registers that are used for arithmetic operations and storage: 32 - bit, 16 bit and 8-bit.

Important: All three types of registers are rings.

The 32- bit registers:

The 32-bit registers that we have are EAX, EBX, ECX, EDX

These 4 registers are used to store 32- bit binary numbers. They all can be used to perform arithmetic operations. However, the recommended convention is to use only the EAX for arithmetic operations and the other three 32-bit registers for temporary storage. These registers will be broken into 4 bytes sections:

32	25	24	17	16	9	8	1

where each bye is divided into two 4 bits

Examples:

1.
mov eax, 5

EAX

0000	0000	0000	0000	0000	0000	0000	0101

2.
mov ebx, 101 01010010b

EBX

0000	0000	0000	0000	0000	0101	0101	0010

3.
mov ecx, 0A93F2CAh

ECX

0000	1010	1001	0011	1111	0010	1100	1010

4.
mov edx , 34577111o

EDX

0011	0100	0101	0111	0111	0001	0001	0001

Exercises:

1. Explain why the follow instructions will cause an error:

a. mov eax, D2h

b. x byte ?
mov eax, x

c. mov eax, 3ABDD12E1h

2. For exercise 1, what can be done so D2h can be stored in EAX?

3. Complete the following table, using only binary numbers in EAX:

ASSEMBLY CODE **EAX**

ASSEMBLY CODE								
mov eax, 2D3Fh								
mov eax, 3h								
mov eax, 1010101b								
mov eax, 434789								
mov eax, 4DFA1101h								
mov eax 2675411o								

■

It is important to realize, as we demonstrated, that only binary numbers are stored in the variables and registers, irrespective of the number system we are using. However, since binary numbers are difficult to read, most debuggers for the assembly language will display the contents of the registers as well as the variables in the equivalent hexadecimal number system (base 16). The following table gives the equivalent values between the binary digits and the hexadecimal digits:

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Examples:

1. mov edx, 9AB120h

EDX

BASE 2:	0000	0000	1001	1010	1011	0001	0010	0000
BASE 16:	0	0	9	A	B	1	2	0

2. mov ecx, 5953189d

ECX

BASE 2:	0000	0000	0101	1010	1101	0110	1010	0101
BASE 16:	0	0	5	A	D	6	A	5

Most of our mathematical experiences has been working with numbers in the base 10. Therefore, if our debugger returns the numbers in our registers as well as variables in hexadecimal, frequently we will need to translate these numbers into base 10. How do we do this? Well, we could use the methods we have learned so far to find the equivalent hexadecimal numbers in the base 10. However, doing this is not practical. It would be better to use a calculator that will quickly go from one base to another. Microsoft Windows XP and Vista provides such a calculator.

Examples:

1. mov eax, 10001100b

EAX

BASE 2:	0000	0000	0000	0000	0000	0000	1000	1100
BASE 16:	0	0	0	0	0	0	8	C
BASE 10:	140							

2. mov ebx, 0DF3h

EBX

BASE 2:	0000	0000	0000	0000	0000	1101	1111	0011
BASE 16:	0	0	0	0	0	D	F	3
BASE 10	3,571							

3. mov ecx, 0111 0111 1101 1110 1110 1110 1011 0111 b

ECX

BASE 2:	0111	0111	1101	1110	1110	1110	1011	0111
BASE 16:	7	7	D	E	E	E	B	7
BASE 10:	2,011,098,80							

Exercises:

1. Complete the following:

a. mov eax , 278901

EAX

BASE 2:								
BASE 16:								
BASE 10:								

b. mov eax , 3ABCD10Fh

EAX

BASE 2:								
BASE 16:								
BASE 10:								

c. mov edx , 2772101o

EDX

BASE 2:								
BASE 16:								
BASE 10:								

d. mov eax , 278901

EAX

BASE 2:								
BASE 8:								
BASE 10:								

e. mov ecx , 3ABCD10Fh

ECX

BASE 2:								
BASE 8:								
BASE 10:								

f. mov edx , 2772101o

EDX

BASE 2:								
BASE 8:								
BASE 10:								

2. What is the largest number?:

a. binary integer of type BYTE ?

b. octal integer of type BYTE ?

c. decimal integer associated with type BYTE ?

3. What is the largest:

a. binary integer of type WORD ?

b. octal integer of type WORD ?

c. decimal integer associated with type WORD ?

4. What is the largest:

a. binary integer of type DWORD ?

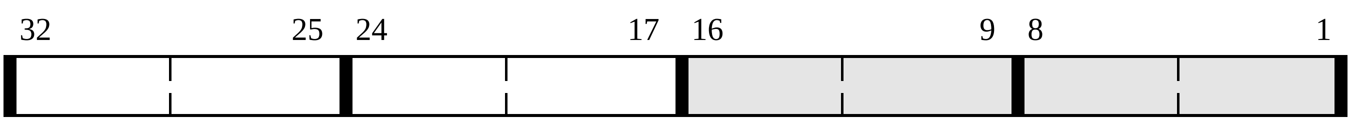
b. octal integer of type DWORD ?

c. decimal integer associated with type DWORD ?
■

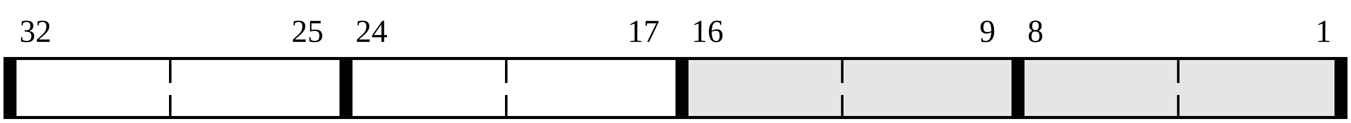
The 16- bit registers:

The 16-bit registers are AX, BX, CX, DX. Each of these registers occupy the right-most part of there corresponding 32 bit - registers:

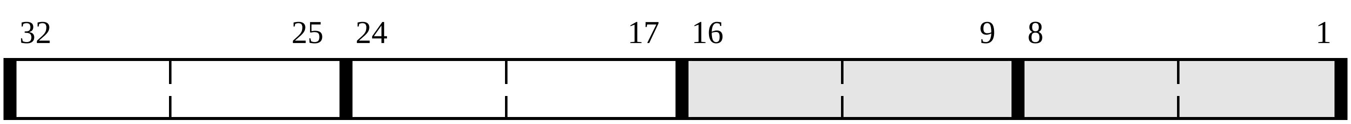
EAX AX



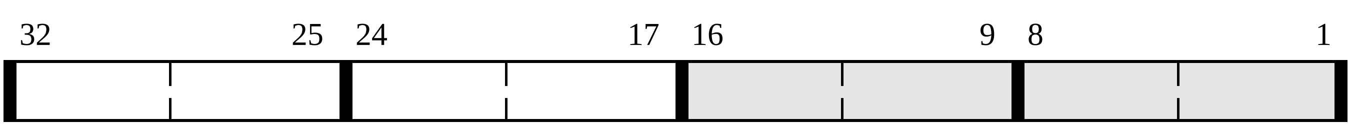
EBX BX



ECX CX



EDX DX



Example:

INSTRUCTIONS	32	25	24	17	16	9	8	1
mov eax, 3C293567h	3	C	2	9	**3**	**5**	**6**	**7**
mov ax, 9BCh	3	C	2	9	**0**	**9**	**B**	**C**
mov ax, 56325d	3	C	2	9	**D**	**C**	**0**	**F**

Note: When working with a 16-bit register, the other bits of the 32-bit register are not affected.

The 8- bit registers:

The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, DL . AH occupies the left most bits of AX and AL occupies the right most 8 bits of AX, etc.:

EAX AX

32	25	24	17	16 9	8 1
				AH	AL

EBX BX

32	25	24	17	16 9	8 1
				BH	BL

ECX CX

32	25	24	17	16 9	8 1
				CH	CL

EDX DX

32	25	24	17	16 9	8 1
				DH	DL

Examples:

INSTRUCTIONS	32	25	24	17	16	9	8	1
mov eax, 7293567h	0	7	2	9	**3**	**5**	**6**	**7**
mov ax, 9BCh	0	7	2	9	**0**	**9**	**B**	**C**

mov ah, 5	0	7	2	9	**0**	**5**	**B**	**C**
mov al, 0Eh	0	7	2	9	**0**	**5**	**0**	**E**
mov al, 251	0	7	2	9	**0**	**5**	**F**	**B**

Note: When working with a 8-bit register, the other bits of the 16-bit and the 32-bit registers are not affected

Mixing Registers

Rule: The assembler will not allow mixing of registers of different data types. The following are examples of errors in programming:

mov eax, bx

mov cx, eax

mov dx, al

Exercises:

1. Complete the following tables using hexadecimal numbers only :

	32	25	24	17	16	9	8	1
INSTRUCTIONS								
mov eax, 293567h								
mov ax, 9BCh								
mov ax, 3D32h								
mov ax, 5h								
mov ax, 3h								
mov eax, 1267								

2.

	32	25	24	17	16	9	8	1
INSTRUCTIONS								
mov eax, 112937234								
mov ax, 9BCh								
mov al, 5								
mov ah, 0Eh								
mov al, 2								

9.6 Transferring data between registers and variables

The following examples demonstrate how integer data is transferred using the *mov* instruction:

Examples:

1.

```
 x dword   23
mov eax, x
```

2.

```
x dword 23
y dword ?
mov ebx, x
mov y, ebx
```

3.

```
 x word   3A7Fh
mov ax, x
```

4.

```
x word 3A7Fh
y word ?
mov bx, x
mov y, bx
```

5.

```
 x byte   3Ah
mov ah, x
```

6.

```
x byte 3Ah
y byte ?
mov bl, x
mov y, bl
```

Transferring data from one variable to another variable

The above examples show how to transfer the contents of one variable to another variable. The following algorithm demonstrates: x := y .

PSEUDO CODE	AL PSEUDO CODE	ASSEMBLY LANGUAGE CODE
X:= Y	EAX := Y	mov eax, y
	X:= EAX	mov x,eax

The following program will perform the following tasks:

Task 1: Store the number 23 into x
Task 2: Store the number 59 into y
Task 3: Store the contents of x into y.

AL PSEUDO CODE	ASSEMBLY LANGUAGE CODE
X := 23	mov x, 23
Y := 59	mov y, 59
EAX := X	mov eax, x
Y := EAX	mov y,eax

EXERCISE:

1. Modify the above program by initializing the values in x, y without using the *mov* instruction.

2. Complete the following table:

AL PSEUDO CODE	AL CODE	X	Y	EAX	EBX
X := 23	mov x, 23				
Y := 59	mov y, 59				
EAX := X	mov eax, x				
EBX := Y	mov ebx, y				
X := EBX	mov x, ebx				
Y := EAX	mov y, eax				

2. In exercise 1, what does the code accomplish ?
■

9.7 Assembly Language Statements

In assembly language there are three basic statements: *instructions, directives, and macros.*

Definition of instructions:

An instruction is translated by the assembler into one or more bytes of object code which will be translated into machine language. The general form of an instruction is:

label: (optional) mnemonic   operand(s)   ; comment (optional) where

mnemonic is an instruction and operands can be numeric value, variable, register.

Example:

label: mov eax, 23h ; This is an instruction.

There are two kinds of instructions:

1. non-executable codes

2. executable codes.

Example of a non-executable instruction: The comment

Definition of a comment: A comment is any string of characters preceded by a semicolon (;)

The comment is ignored by the assembler.

Example:

```
  mov eax, 2 ;   Transfer the number 2 into the register EAX.
```

The instruction *mov eax, 2* will be executed by the assembler but the string following the semicolon will be ignored by the assembler.

The label:

All instructions can be preceded by a label ending in a colon (:). The rules for the label are basically the same as variables.

Example:

```
xyz:  mov eax ,  -4
```

Later we will se how labels are used in programing.

Definition of a Directive:

A directive instructs the assembler to take a certain action.

Variable Data Type Declarations

A Variable has to be designated as one of the following type types: *BYTE, WORD, DWORD.*

Definition of a BYTE: A byte consists of 8 bits.

Definition of a WORD: A word consists of 2 bytes (16 bits).

Definition of a DWORD **:** A double word (DWORD) consists of 4 bytes (32 bits).

The form of the variable data type declarations is the following:

variable name *data type* *numeric value assigned* or ?

Examples:

Num BYTE 23 ;will define Num as a 8 bit byte and will convert the number 23 to binary and store it into the variable Num.

Num WORD ? ;will define Num as a 16 bit word but will not assign a value to Num.

Num DWORD 0ACD35h ;will define Num as a 32 bits dword and will convert the number 0ACDE5h to binary and store it into the variable Num.

Note: You may place a label in front of the variable declaration but the colon (:) is not allowed.

Exercises:

1. What is the largest integer number base 10 that can be store in a variable of type BYTE.
2. What is the largest integer number base 10 that can be store in a variable of type WORD.
3. What is the largest integer number base 10 that can be store in a variable of type DWORD.
4. What is the largest integer number base 16 that can be store in a variable of type BYTE.
5. What is the largest integer number base 16 that can be store in a variable of type WORD.
6. What is the largest integer number base 16 that can be store in a variable of type DWORD.
7. What is the largest integer number base 8 that can be store in a variable of type BYTE.
8. What is the largest integer number base 8 that can be store in a variable of type WORD.
9. What is the largest integer number base 8 that can be store in a variable of type DWORD.

■

Exercise:
Assume the above program is run. For the table below, fill in the final values stored.

EAX	EBX	A	B	C	D	E	F

■

9.8 A SAMPLE ASSEMBLY LANGUAGE WRITTEN FOR MASMA (Microsoft Assembler)

The following is a complete assembly language program written for the MASMA

(Microsoft Assembler)

```
; This program assigns values to registers
; Last update: 2/1/02

.386

.MODEL FLAT

.STACK 4096

.DATA
a     byte   40
b     byte  30
d     dword  10
e     byte   50
f     word   20

.CODE                        ;  start of main program code

_start:
          ;
          ; code inserted here
          ;
      mov eax, 10h
      mov ebx, 15h
      mov eax, d
      mov ax, f
       mov ah, e
PUBLIC_start

END                         ; end of source code
```

PROJECT

Write an assembly language program that will rearrange the numbers so that they are in increasing order as shown below::

	A	**B**	**C**	**D**	**E**
BEFORE	40	30	10	50	20
AFTER	10	20	30	40	50

Do not add any additional variables.

CHAPTER 10 - ARITHMETIC EXPRESSIONS

INTRODUCTION

Our next step in becoming assembly language programmers is to learn how to create arithmetic expressions. Those who have studied higher level programming languages know that assigning arithmetic expressions to variables generally follow the normal assignment statements. For example, in pseudo- code we can write such instructions as X =: 2 + 3. However, in assembly language, it is not possible to directly write such an assignment statement.
To be able to create arithmetic expressions in assembly language, we first study what are unsigned/signed integer numbers, followed by the arithmetic operations that are available to us. We then learn how to build arithmetic expressions using these types of numbers as needed.

10.1 Ring Registers

In Chapter 9, (9.6) we saw that there are three important rings in the assembly language: byte rings, word rings and dword rings. The three type of registers EAX,(EBX,ECX, EDX), AX (BX,CX,DX) and AH, AL (BH,BL, CH,CL, DH,DL) are rings, they conform to the modular rules of arithmetic. The modular formula is

r = m mod N where

N = 256_{10} for the byte rings: AH, AL (BH,BL, CH,CL, DH,DL),

N = $65{,}536_{10}$ for the word rings: AX (BX,CX,DX),

N = $4{,}294{,}967{,}296_{10}$ for the dword rings: EAX,(EBX,ECX, EDX),

Additive Inverses

Since the rings do not have negative numbers, as we have in ordinary numbers in the base 10, we need to approach the creation of “negative” numbers in these rings by the following reasoning: in the ordinary base 10 number system, negative numbers are additive inverses of non negative numbers and non negative numbers are additive inverses of negative numbers. Therefore, we can create additive numbers in the rings by associating each number of the ring with its corresponding additive inverse. To accomplish this, we begin with the definition of unsigned and signed integers. (See chapter 8, for the definition of additive inverse for a ring and section 8.8 where we first introduce the concept of unsigned and signed binary integers).

Unsigned and signed binary integers

We start with a arbitrary ring of binary integer numbers:

R = {0...00 , 0...01, 0...10, 0...11,....011...1, 10...00, 10...01, 10...10, 10...011, ..., 11...1}

For rings of this type we have the following definitions:

Definition of an unsigned binary integer number: An unsigned binary integer number has as its extreme left most bit the bit number zero (0).

Definition of a signed binary integer: A signed binary integer number has as its extreme left most bit the bit number one (1).

We see above that the ring R can be divided into two subsets consisting of those binary number that are unsigned:

{0...00 , 0...01, 0...10, 0...11,....011...1}

and those that signed:

{10...00, 10...01, 10...10, 10...011, ..., 11...1}

The 8 bit ring as unsigned binary and integer numbers.

The following table contains the integer numbers base 10 and their 8 bit unsigned binary representation:

NON-NEGATIVE INTEGERS BASE 10	UNSIGNED BINARY REPRESENTATION
0	00 00 00 00
1	00 00 00 01
2	00 00 00 10
3	00 00 00 11
4	00 00 01 00
5	00 00 01 01
6	00 00 01 10
7	00 00 01 11
8	00 00 10 00
9	00 00 10 01
:::::	:::::
127	01 11 11 11

Next we need to convert the 8 bit binary numbers to their 8 bit additive inverse numbers. From chapter 8, we use the following formula:

The additive inverse of $a_1a_2a_3a_4a_5a_6a_7\,a_8$ equals $(a_1a_2a_3a_4a_5a_6a_7a_8)' + 1 = a_1'a_2'a_3'a_4'a_5'a_6'a_7'a_8' + 1$
where

$a_k' = 1$ if $a_k = 0$
and
$a_k' = 0$ if $a_k = 1$.

The following table of unsigned and signed binary numbers are listed so that each of two columns are additive inverses of each other:

INTEGERS BASE 10	BINARY REPRESENTATION	INTEGERS BASE 10	BINARY REPRESENTATION
0	00 00 00 00	0	00 00 00 00
1	00 00 00 01	255	11 11 11 11
2	00 00 00 10	254	11 11 11 10
3	00 00 00 11	253	11 11 11 01
4	00 00 01 00	252	11 11 11 00
5	00 00 01 01	251	11 11 10 11
6	00 00 01 10	250	11 11 10 10
7	00 00 01 11	249	11 11 10 01
8	00 0010 00	248	11 11 10 00
9	00 00 10 01	247	11 11 01 11
:::::	:::::	::::	::::
127	01 11 11 11	129	10 00 00 01
128	10 00 00 00	128	10 00 00 00

Note: in the above table, the binary numbers in each of the columns are additive inverses of each other.

Examples:

1. Convert the binary number representing 5 to its additive inverse.

Step 1: The integer number 5: 00000101

Step 2: The additive inverse of 00000101 equals 11111010 + 1 = 11111011

2. Convert the binary number representing 3 to its additive inverse.

Step 1: The integer number 3: 00000011

Step 2: The additive inverse of 00000011 equals 11111100 + 1 = 11111101

The following table gives the representation of the above table as hexadecimal numbers. Most assemblers will display the binary numbers in registers as their corresponding hexadecimal values.

INTEGERS BASE 10	HEXADECIMAL NUMBERS	INTEGERS BASE 10	HEXADECIMAL NUMBERS
0	00	0	00
1	01	255	FF
2	02	254	FE
3	03	253	FD
4	04	252	FC
5	05	251	FB
6	06	250	FA
7	07	249	F9
8	08	248	F8
9	09	247	F7
:::::	:::::	::::	::::
127	7F	129	81
128	80	128	80

Note: in the above table, the hexadecimal numbers in each of the columns are additive inverses of each other.

Exercises:

1. Find the additive inverse of the following numbers in binary as well as the number system given:

a. 100101b b. 2E h c. 222 d

2. Find in binary representation of the following numbers :

a. - 81h b. - 1010111b c. -28h

■

The 16 bit rings

The following table contains the 16 bit ring divided into columns which are additive inverses of each other.

INTEGERS BASE 10	BINARY REPRESENTATION	INTEGERS BASE 10	BINARY REPRESENTATION
0	00 00 00 00 00 00 00 00	0	00 00 00 00 00 00 00 00
1	00 00 00 00 00 00 00 01	65535	11 11 11 11 11 11 11 11
2	00 00 00 00 00 00 00 10	65534	11 11 11 11 11 11 11 10
3	00 00 00 00 00 00 00 11	65533	11 11 11 11 11 11 11 01
4	00 00 00 00 00 00 01 00	65532	11 11 11 11 11 11 11 00
5	00 00 00 00 00 00 01 01	65531	11 11 11 11 11 11 10 11
6	00 00 00 00 00 00 01 10	65530	11 11 11 11 11 11 10 10
7	00 00 00 00 00 00 01 11	65529	11 11 11 11 11 11 10 01
8	00 00 00 00 00 00 10 00	65528	11 11 11 11 11 11 10 00
9	00 00 00 00 00 00 10 01	65527	11 11 11 11 11 11 01 11
::::	::::	::::	:::
32767	01 11 11 11 11 11 11 11	32769	10 00 00 00 00 00 00 01
32768	10 00 00 00 00 00 00 00	32768	10 00 00 00 00 00 00 00

Note: in the above table, the binary numbers in each of the columns are additive inverses of each other.

The following table gives the representation of the binary numbers as hexadecimal numbers.

INTEGERS BASE 10	HEXADECIMAL NUMBERS	INTEGERS BASE 10	HEXADECIMAL NUMBERS
0	00 00	0	00 00
1	00 01	65535	FF FF
2	00 02	65534	FF FE
3	00 03	65533	FF FD
4	00 04	65532	FF FC
5	00 05	65531	FF FB
6	00 06	65530	FF FA
7	00 07	65529	FF F9

8	00 08	65528	FF F8
9	00 09	65527	FF F7
::::	::::	::::	::::
32767	7F FF	32769	80 01
32768	80 00	32768	80 00

Note: in the above table, the hexadecimal numbers in each of the columns are additive inverses of each other.

Exercises:

1. Find the additive inverse of the following numbers in binary as well as the number system given:

a. 100101b   b. 2E h   c. 222 d

2. Find in binary representation of the following numbers :

a. - 81h   b. - 1010111b   c. -28h

■

The 32 bit rings

The following table contains the 32 bit ring divided into columns which are additive inverses of each other.

INTEGERS BASE 10	BINARY REPRESENTATION 32 bits	INTEGERS BASE 10	BINARY REPRESENTATION 32 BITS
0	00---00 00 00 00	0	00---00 00 00 00
1	00---00 00 00 01	4,294,967,295	11---11 11 11 11
2	00---00 00 00 10	4,294,967,294	11---11 11 11 10
3	00---00 00 00 11	4,294,967,293	11---11 11 11 01
4	00---00 00 01 00	4,294,967,292	11---11 11 11 00
5	00---00 00 01 01	4,294,967,291	11---11 11 10 11
6	00---00 00 01 10	4,294,967,290	11---11 11 10 10
7	00---00 00 01 11	4,294,967,289	11---11 11 10 01
8	00---00 00 10 00	4,294,967,288	11--- 11 11 00 00
9	00---00 00 10 01	4,294,967,287	11--- 11 11 01 11
::::	::::	::::	::::
2,147,483,647	01---11 11 11 11	2,147,483,649	10---00 00 00 01
2,147,483,648	10---00 00 00 00	2,147,483,648	10---00 00 00 00

Note: in the above table, the binary numbers in each of the columns are additive inverses of each other.

The following table gives the representation of the binary numbers as hexadecimal numbers

INTEGERS BASE 10	HEXADECIMAL NUMBERS	INTEGERS BASE 10	HEXADECIMAL NUMBERS
0	00 00 00 00	0	00 00 00 00
1	00 00 00 01	4,294,967,295	FF FF FF FF
2	00 00 00 02	4,294,967,294	FF FF FF FE
3	00 00 00 03	4,294,967,293	FF FF FF FD
4	00 00 00 04	4,294,967,292	FF FF FF FC
5	00 00 00 05	4,294,967,291	FF FF FF FB
6	00 00 00 06	4,294,967,290	FF FF FF FA
7	00 00 00 07	4,294,967,289	FF FF FF F9
8	00 00 00 08	4,294,967,288	FF FF FF F8
9	00 00 00 09	4,294,967,287	FF FF FF F7
::::	**::::**	::::	:::::::::
2,147,483,647	7F FF FF FF	2,147,483,649	80 00 00 01
2,147,483,648	80 00 00 00	2,147,483,648	80 00 00 00

Note: in the above table, the hexadecimal numbers in each of the columns are additive inverses of each other.

Exercises:

1. Find the additive inverse of the following numbers in binary as well as the number system given:

a. 100101b b. 2E h c. 222 d

2. Find in binary representation of the following numbers :

a. - 81h b. - 1010111b c. -28h

■

Computing a - b

In order to see how subtraction of numbers is handled by the assembler, we need to interpret a - b as addition:

a - b = a + - b

We will interpret -b as the additive inverse of b.

For the following examples, we assume that the numbers are represented in 8 bit registers.

Examples:

1. Show 5 - 2 = 3

Solution:

Step 1: 5 - 2 = 5 + -2

Step 2: The binary representation of 5 is 00000101

Step 3: The binary representation of - 2 is the additive inverse of 2. The binary representation of -2 is

(00000010)′ + 1 = 11111101 + 1 = 11111110

Step 4: The binary representation of 5 + - 2 is 00000101 + 11111110 = 00000011

Step 5: Since the leading bit is 0, 00000011 is the binary representation of 3.

2. Show 2 - 5 = - 3

Solution:

Step 1: 2 - 5 = 2 + -5

Step 2: The binary representation of 2 is 00000010

Step 3: The binary representation of - 5 is the additive inverse of 5. The binary representation of -5 is

(00000101)′ + 1 = 11111010 + 1 = 11111011

Step 4: The binary representation of 2 + - 5 is 00000010 + 11111011 = 11111101

Step 5: To find The binary representation of 2 + - 5 we compute the additive inverse of 11111101:

(11111101)′ + 1 = 00000010 + 1 = 00000011

which is the binary representation of 3.

Step 6: Therefore the binary representation of 11111101 is -3

3. Show - 2 - 5 = - 7

Solution:

Step 1: -2 - 5 = -2 + -5

Step 2: From the above table, the binary representation of -2 is 11111110.

Step 3: From the above table, the binary representation o of - 5 is 11111011

Step 4: The binary representation of - 2 + - 5 is 11111110 + 11111011 = 11111001

Step 5: From the above table we find that the additive inverse of 11111001 is -7 .

Exercises:

1. Perform the following operations:

a. 10011010b + 1010110b b. 244d + 177d + 8d c. 5Ah + FEh d. 78h - EBh

2. From example 1, we found

00000101 + 11111110 = 00000011

When adding how does one justify that when performing addition on these two numbers, the extreme left digit 1 disappears.

3. A mathematical system is said to be an integral domain if for all numbers a,b in the system where $a * b = 0$, it follows that a = 0 or b = 0.

Is the 8 -bit ring an integral domain? Explain.

4. Which numbers, if any, are equal to its own additive inverse ?
■

Computing $a * b$

As we did for addition and subtraction, we will show by example how multiplication of binary numbers is accomplished.

Examples:

1. Show $2 * 3 = 6$

Step 1: The binary representation of 2 is 00000010 .

Step 2: The binary representation of 3 is 00000011 .

Step 3: Using the standard method of multiplication:

```
  00000010
  00000011
  --------
  00000010
 00000010x
 =========
  00000110
```

From the above table, we see that

00000110

is the binary representation of 6.

2. Show $-2 * 3 = -6$

Step 1: The binary representation of -2 is 11111110 .

Step 2: The binary representation of 3 is 00000011 .

Step 3: Using the standard method of multiplication:

```
   11111110
   00000011
   11111110
  11111110x
 1011111010
```

$1011111010_2 = \Rightarrow 762_{10} \bmod 256 = 250_{10}$

From the above table, we see that 250 is the additive inverse of 6

Exercises:

1. Perform the following operations:

a. 10011010b $*$ 1010110b   b. 244d $*$ 177d + 8d   c. 5Ah $*$ FEh   d. (78h - EBh)$*$2h

2. For each of the examples above, convert the final answers to hexadecimal.

■

10.2 ASSEMBLY LANGUAGE ARITHMETIC OPERATIONS FOR INTEGERS

The following is a list of the important arithmetic operations for integers:

Addition (+):

Definition: Form of the assembly language add instruction*: add register, source*

where the following rules apply:

Rule 1: The integers may be unsigned or signed.

Rule 2: The source can be a register, variable, or numeric value.

Rule 3: The resulting sum will be stored in the register.

Rule 4: Data types for the register and source must always be the same.

Examples :

1.

PSEUDO-CODE	AL PSEUDO-CODE	ASSEMBLY CODE
Z := 2h + 3h	EAX := 2h	mov eax, 2h
	EAX := EAX + 3h	add eax, 3h
	Z := EAX	mov z, eax

2.

ASSEMBLY CODE	EAX		AX	AH	AL	X
x dword 2h						**2**
mov eax, 12345h	**00 01**	**23 45**	**23 45**	**23**	**45**	2
add eax, x	00 01	**23 47**	23 **47**	23	**47**	2

3.

ASSEMBLY CODE	EAX		AX	AH	AL	X
x word 1h						**1**
mov ax, 0ffffh	**00 00**	**ff ff**	**ff ff**	**ff**	**ff**	1
add ax, x	00 00	**00 00**	**00 00**	**00**	**00**	1

4.

ASSEMBLY CODE	EAX		AX	AH	AL	X
x byte 2h						**2**
mov eax, 0	**00 00**	**00 00**	**00 00**	**00**	**00**	**2**
mov al, 0ffh	**00 00**	**00 ff**	**00 ff**	00	**ff**	2
add al, x	00 00	00 **01**	**00 01**	00	**01**	2

5.

ASSEMBLY CODE	EAX		AX	AH	AL	X
x byte 2h						**2**
mov eax,0	**00 00**	**00 00**	**00 00**	**00**	**00**	**2**
mov ah, 0ffh	00 00	**ff 00**	**ff 00**	**ff**	00	2
add ah, x	00 00	**01 00**	**01 00**	**01**	00	2

6.

PSEUDO_CODE	AL PSEUDO-CODE	AL CODE	EAX	AX	X	Y	W
		x word ?					
Y := 223h	Y := 223h	y dword 223h				**223**	
W := 79223h	W := 79223h	w dword 79223h				223	**79223**
X:= 2h + 3h	AX:= 2h	mov ax, 2h	**00 00 00 02**	**00 02**		223	79223
	AX := AX + 3h	add ax, 3h	**00 00 00 05**	**00 05**		223	79223
	X := AX	mov, x, ax	**00 00 00 05**	00 05	**5**	223	79223
W := W + Y	EAX:= W	mov eax, w	**00 07 92 23**	**92 23**	5	223	79223
	EAX := EAX +Y	add eax, y	**00 07 94 46**	**9446**	5	223	79223
	W := EAX	mov w, eax	00 07 94 46	9446	5	223	**79446**

Exercises:

1. Complete the following tables:

ASSEMBLY CODE	EAX	AX	AH	AL	X
x dword 2h					
mov eax, 12345h					
add eax, x					

ASSEMBLY CODE	EAX	AX	AH	AL	X
x word 1h					
mov eax, 0ffffh					
add ax, x					

ASSEMBLY CODE	EAX	AX	AH	AL	X
x byte2h					
mov eax, 12345h					
add eax, x					

2. Complete the table below:

PSEUDO CODE	AL PSEUDO-CODE	AL CODE	EAX	AX	X	Y	W
		x word ? y dword 223h w dword 79223h					
W := W + Y							
X := 2 + 3							

■

Subtraction (-):

Definition: Form of the subtraction instruction*: sub register, source*

where the following rules apply:

Rule 1: The integers may be signed or unsigned.

Rule 2: The source can be a register, variable, or numeric value.

Rule 3: The resulting subtraction will be stored in the register.

Rule 4: Data types for the register and source must always be the same.

Examples :

1.

PSEUDO-CODE	AL PSEUDO-CODE	ASSEMBLY CODE
Z := 2h - 3h	EAX := 2h	mov eax, 2h
	EAX := EAX - 3h	sub eax, 3h
	Z := EAX	mov z, eax

2.

ASSEMBLY CODE	EAX		AX	AH	AL	X
x dword 10h						**10**
mov eax , 12345678h	**1234**	**5678**	**5678**	**56**	**78**	10
sub eax, x	**1234**	**5668**	**5668**	**56**	**68**	10

3.

ASSEMBLY CODE	EAX		AX	AH	AL	X
x word 5000h						**5000**
mov eax, 12345678h	**1234**	**5678**	**5678**	**56**	**78**	5000
sub ax, x	**1234**	**0678**	**0678**	**06**	**78**	5000

4.

ASSEMBLY CODE	EAX		AX	AH	AL	X
x byte 70h						**70**
mov eax, 12345678h	**1234**	**5678**	**5678**	**56**	**78**	70
sub al, x	**1234**	**5608**	5**608**	56	**08**	70

Exercises:

1. Complete the following table:

PSEUDO-CODE	AL PSEUDO-CODE	AL CODE	EAX	X	Y	Z
		x dword ? y dword ? z dword ?				
X := CD2h - 2h	EAX := OCD2h					
	EAX := EAX - 2h					
	X := EAX					
X := 421h	X := 421h					
Y := 4E75h	Y := 4E75h					
Z:= X - Y	EAX := X					
	EAX := EAX - Y					
	Z := EAX					

2.

ASSEMBLY CODE	EAX	AX	AH	AL	X
x dword 5677h					
mov eax , 0C1234h					
sub eax, x					

3.

ASSEMBLY CODE	EAX	AX	AH	AL	X
x word 0ab9h					
mov eax, 0cca18h					
sub ax, x					

4.

ASSEMBLY CODE	EAX	AX	AH	AL	X
x byte 0dh					
mov eax, 12345678h					
sub al, x					

■

Multiplication (∗):

Definition: There are 2 multiplication instructions we can use: mul and imul.

• Form of the mul instruction*:* *mul source*

•Form of the imul instruction: *imul source*

where the following rules apply:

Rule 1: The register used for multiplication is alway EAX.

Rule 2a: For the mul instruction, the integers that are multiplied must be unsigned.

Rule 2b: For the imul instruction, the integers can be either unsigned, signed order or both.

Rule 3: The source can be a register or a variable. The source cannot be a numeric value .

Rule 4: The location of the other number (accumulator) to be multiplied it is in one of the following registers:

• AL, if the source is a byte.

• AX,   if the source is a word.

• EAX,   if the source is a double word.

Rule 5: The resulting product will be located in the accumulator under the following rules:

• If the data type is a byte (8 bits) , then the resulting product (8 bits) will be located in AL.

• If the data type is a word (16 bits), then the resulting product (16) bits it will be located in AX;

• If the data type is a dword (32 bits), then the resulting product (32) bits has its lower 16 bits will be located in AX and higher order bits in the DX register.

• If the data type is a qword (64 bits)[1], then the resulting product (64) bits its lower 32 bits will be located in EAX and its higher order bits in the EDX register.

Examples :

1.

PSEUDO-CODE	AL PSEUDO-CODE	ASSEMBLY CODE
Z := 2h∗3h	EAX := 2h	mov eax, 2h
	EBX:= 3h	mov ebx, 3h
	EAX := EAX∗EBX	mul ebx
	Z := EAX	mov z, eax

2.

ASSEMBLY CODE	EAX	AX	EDX	X
x dword 10h				**10**
mov eax , 1234567h	**01 23 45 67**			10
mul x		**45 67**	**01 23**	10

3.

ASSEMBLY CODE	AL	AX	X
x byte 10h			**10**
mov al, 23h	**23**	**00 23**	10
mul x	**30**	**02 30**	10

[1] The qword will be discussed in chapter 20.

4.

ASSEMBLY CODE	AX	DX	X
x word 100h			**100**
mov ax 1234h	**12 34**		100
mul x	**34 00**	**00 12**	100

5.

ASSEMBLY CODE	EAX	EDX	X
x dword 100h			**100**
mov eax , 1234567h	**01 23 45 67**		10
mul x	**23 45 67 00**	**01**	10

Exercises:

1. Complete the following tables:

.

ASSEMBLY CODE	EAX	AX	AH	AL	X
x dword 0edh					
mov eax , 77bd55h					
imul x					

ASSEMBLY CODE	EAX	AX	AH	AL	X
x byte 0bh					
mov al, 2ch					
imul x					

■

Division (÷):

For this type of division, we are only performing integer division. The following is the definition of integer division:

Definition of integer division n÷m : Given unsigned integers n, m, we say n is divided by m where

$n = q * m + r$, where

$0 \leq r < m$.

$n \div m = q$

$n = (n \div m) * m + r$

Note: The general terminology is:

n: dividend
m:divisor
q: quotient
r: remainder

Examples:

a. $9 \div 4$: $9 = 2 * 4 + 1$ where $q = 2$ and $r = 1$

$9 \div 4 = q = 2$

b. $356 \div 7$: $356 = 50 * 7 + 6$ where $q = 50$ and $r = 6$

$356 \div 7 = q = 50$

c. $78 \div 99$: $78 = 0 * 99 + 78$ where $q = 0$ and $r = 78$

$78 \div 99 = 0$

Exercises:

1. For the following integer division, find the division form: $n = q * m + r$:

a. $143 \div 3$ b. $3,457 \div 55$ c. $579 \div 2$ d. $23 \div 40$

■

There are 2 division instructions we will use: div and idiv.

- Form of the div instruction*:* *div source*

•Form of the idiv instruction: *idiv* *source*

where the following rules apply:

Rule 1: The register used for integer division is alway EAX.

Rule 2: The source is the divisor (m).

Rule 3: The source can be in a register or variable, but cannot be a numeric value.

Rule 4: The following gives us the locations of n,m,q,r.

• If the source (m) is a byte, then the dividend (n) is stored in the AX register. After execution, the quotient (q) will be stored in the AL register and the remainder (r) in the AH register.

• If the source (m) is a word, then the dividend (n) is stored in the AX register. Before executing, the EDX must be assign a numeric value. After execution, the quotient(q) will be stored in AX and the remainder (r) in DX.

• If the source(m) is a double word, then the dividend (n) is stored in the EAX register. Before executing, the EDX must be assign a numeric value. After execution, the quotient (q = n÷m) will be stored in the EAX register and the remainder(r) in the EDX register.

Rule 5:

• The div instruction should only be used when the dividend and divisor are both unsigned.

• The idiv instruction can be used when the dividend and divisor can be either signed or unsigned or both .

The following table summaries Rule 3:

DIVIDEND (N)	DIVISOR (M)	N÷M	REMAINDER
AX	byte: register or variable	AL	AH
AX	word: register or variable	AX	DX
EAX	dword: register or variable	EAX	EDX

Important: When programming in Visual Studio, one must assign the number 0 to the EDX register before each div or idiv instruction.

Examples:
1.

ASSEMBLY CODE	EAX	AX	AH	AL	EDX	X
x dword 10h						**10**
mov edx, 0h					**00 00 00 00**	10
mov eax, 378h	**00 00 03 78**	**03 78**	**03**	**78**	00 00 00 00	10
div x	**00 00 00 37**	**00 37**	**00**	**37**	**00 00 00 08**	10

2.

ASSEMBLY CODE	EAX	AX	AH	AL	EDX	X
x word 100h						**100**
mov edx,0					**00 00 00 00**	100
mov ax, 9378h	**00 00 93 78**	**93 78**	**93**	**78**	00 00 00 00	100
div x	**00 00 00 93**	**00 93**	**00**	**93**	**00 00 00 78**	100

3.

ASSEMBLY CODE	EAX	AX	AH	AL	X
x byte 10h					**10**
mov ax, 456h	**04 56**	**04 56**	**04**	**56**	10
div x	**06 45**	**06 45**	**06**	**45**	10

Exercises:

Complete the following table:

1. The following program will cause an overflow. Explain why ?

```
x byte 10h
mov ax,1456
idiv x
```

2. complete the following tables:

ASSEMBLY CODE	EAX	AX	AH	AL	EDX	X
x dword 22ch						
mov edx, 0						
mov eax, 0f3aah						
div x						

ASSEMBLY CODE	EAX	AX	AH	AL	EDX	X
x word 567h						
mov edx,0						
mov ax, 0d378h						

div x						

ASSEMBLY CODE	EAX	AX	AH	AL	X
x byte 0fdh					
mov ax, 0a56h					
div x					

■

10.3 Special Numeric Algorithms

In this section we will study how we can write assembly language algorithms for special numeric expressions. To assist us, we will first use pseudo-codes as our guide. The following are several important algorithms :

- *Interchanging values*:

Algorithm:

PSEUDO CODE	AL PSEUDO CODE	AL CODE
TEMP:= X	EAX:= X	mov eax, x
	TEMP:= EAX	mov temp,eax
X:= Y	EAX:= Y	mov eax, y
	X:= EAX	mov x, eax
Y:= TEMP	EAX:= TEMP	mov eax, temp
	Y := EAX	mov y, eax

Example :

PSEUDO CODE	AL PSEUDO CODE	AL CODE	X	Y	EAX	T
X := 254d	X := 254d	mov x, 254d	**254**			
Y := 100d	Y := 100d	mov y, 100d	254	**100**		
T:= X	EAX := X	mov eax, x	254	100	**254**	
	T := EAX	mov t, eax	254	100	254	**254**
X:= Y	EAX:= Y	mov eax, y	254	100	**100**	254
	X:= EAX	mov x, eax	**100**	100	100	254

Y:= T	EAX:= T	mov eax, t	100	100	**254**	254
	Y := EAX	mov y, eax	100	254	254	254

• *The exponential operator:* Although we define an exponential operator in assembly, the exponential operator does not exist in the assembly language.

One way to create an exponential operation in assembly language is to perform repetitive multiplication of the same number. The following algorithm will perform such a task:

Algorithm:

PSEUDO-CODE	**AL PSEUDO-CODE**	**AL - CODE**
P:= 1	P:= 1	mov p, 1
P:= X∗P	EAX:= X	mov eax,x
	EAX:= EAX∗P	mul p
	P:=EAX	mov p,eax
P:= X∗P	EAX:= X	mov eax,x
	EAX:= EAX∗P	mul p
	P:=EAX	mov p,eax
::::::::::	::::::::::	::::::::::::::::
P:= X∗P	EAX:= X	mov eax,x
	EAX:= EAX∗P	mul p
	P:=EAX	mov p,eax

Example: :

Compute x := 10^4

AL Pseudo-Code	**AL - CODE**	**X**	**EAX**	**P**
P := 1	mov p, 1			1
X:=10	mov x, 10	10		1
EAX:= X	mov eax, x	10	**10**	1
EAX:= EAX∗P	mul p	10	10	1
P:= EAX	mov p,eax	10	10	**10**
EAX:= X	mov eax, x	10	**10**	10
EAX:= EAX∗P	mul p	10	**100**	10
P:= EAX	mov p,eax	10	100	**100**

EAX:= X	mov eax, x	10	**10**	100
EAX:= EAX∗P	mov p,eax	10	**1000**	100
P:= EAX	mov eax, x	10	1000	**1000**
EAX:= X	mov eax, x	10	**10**	1000
EAX:= EAX∗P	mov p,eax	10	**10000**	1000
P:= EAX	mov eax, x	10	10000	10000

- *Sum the digits of a positive integer* $a_1a_2a_3...a_n$

Example:
Sum the digits of 268.

PSEUDO-CODE	AL PSEUDO-CODE	N	R	EAX	SUM	EDX	TEN
TEN := 10d	TEN := 10d						**10**
N:= 268d	N := 268d	**268**					10
SUM := 0d	SUM := 0d	268			0		10
R := N MOD TEN	EAX:= N	268		**268**	0		10
	EAX:=EAX ÷ TEN EDX:= EAX MOD TEN	268		**26**	0	**8**	10
	R:= EDX	268	**8**	26	0	8	10
N:= N÷10d	N:= EAX	**26**	8	26	0	8	10
SUM:= SUM + R	EDX:= EDX + SUM	26	8	26	0	**8**	10
	SUM:= EDX	26	8	26	**8**	8	10
R:= N MOD TEN	EAX:= EAX÷ TEN EDX:= EAX MOD TEN	26	8	**2**	8	**6**	10
	R:= EDX	26	**6**	2	8	6	10
N:= N÷10d	N:= EAX	**2**	6	2	8	6	10
SUM:= SUM + R	EDX:= EDX + SUM	2	6	2	8	**14**	10
	SUM:= EDX	2	6	2	**14**	14	10
R:= N MOD TEN	EAX:= EAX÷ TEN EDX:= EAX MOD TEN	2	6	**0**	14	**2**	10
	R:= EDX	2	**2**	0	14	2	10

N:= N÷10d	N:= EAX	**0**	2	0	14	2	10
SUM:= SUM + R	EDX:= EDX + SUM	0	2	0	14	**16**	10
	SUM:= EDX	0	2	0	**16**	16	10

PSEUDO-CODE	**AL PSEUDO-CODE**	**AL CODE**
TEN := 10d	TEN := 10d	mov ten,10
N:= 268d	N := 268d	mov n,268
SUM := 0	SUM := 0	mov sum,0
R := N MOD TEN	EAX:= N	mov eax,n
	EDX:= 0 EAX:= EAX÷ TEN	mov edx,0 div ten
	R:= EDX	mov r,edx
N:= N÷10	N:= EAX	mov n,eax
SUM:= SUM + R	EDX:= EDX + SUM	add edx,sum
	SUM:= EDX	mov sum, edx
R:= N MOD TEN	EDX:= 0 EAX:= EAX÷ TEN	mov edx,0 div ten
	R:= EDX	mov r,edx
N:= N÷10	N:= EAX	mov n,eax
SUM:= SUM + R	EDX:= EDX + SUM	add edx,sum
	SUM:= EDX	mov sum, edx
R:= N MOD TEN	EDX:= 0 EAX:= EAX÷ TEN	mov edx,0 div ten
	R:= EDX	mov r,edx
N:= N÷10	N:= EAX	mov n,eax
SUM:= SUM + R	EDX:= EDX + SUM	add edx,sum
	SUM:= EDX	mov sum, edx

Algorithm:

PSEUDO-CODE	AL PSEUDO-CODE	AL CODE
SUM := 0	SUM := 0	mov sum,0
R := N MOD TEN	EAX:= N	mov eax,n
	EDX:= 0 EAX:= EAX÷ TEN	mov edx,0 div ten
	R:= EDX	mov r,edx
N:= N÷10	N:= EAX	mov n,eax
SUM:= SUM + R	EDX:= EDX + SUM	add edx,sum
	SUM:= EDX	mov sum, edx
R:= N MOD TEN	EDX:= 0 EAX:= EAX÷ TEN	mov edx,0 div ten
	R:= EDX	mov r,edx
N:= N÷10	N:= EAX	mov n,eax
SUM:= SUM + R	EDX:= EDX + SUM	add edx,sum
	SUM:= EDX	mov sum, edx
:::::::::	:::::::::	:::::::::::
R:= N MOD TEN	EDX:= 0 EAX:= EAX÷ TEN	mov edx,0 div ten
	R:= EDX	mov r,edx
N:= N÷10	N:= EAX	mov n,eax
SUM:= SUM + R	EDX:= EDX + SUM	add edx,sum
	SUM:= EDX	mov sum, edx

• Factorial $n! = n(n - 1)(n - 2)... (1)$

Example :

5! = 5(4)(3)(2)(1) = 120

AL PSEUDO-CODE	AL CODE	EAX	EBX
EAX := 5d	mov eax , 5	**5**	
EBX := 5d	mov ebx , 5	5	**5**
EBX := EBX - 1d	sub ebx , 1	5	**4**
EAX := EAX∗EBX	mul ebx	**20**	4
EBX := EBX - 1d	sub ebx , 1	20	**3**
EAX := EAX∗EBX	mul ebx	**60**	3
EBX := EBX - 1d	sub ebx , 1	60	**2**
EAX := EAX∗EBX	mul ebx	**120**	2
EBX := EBX - 1d	sub ebx , 1	120	**1**
EAX := EAX∗EBX	mul ebx	120	1

Note: See last page for the complete assembly language program.

Algorithm

PSEUDO-CODE	ASSEMBLY LANGUAGE
EAX := N	mov eax , n
EBX : = N	mov ebx , n
EBX := EBX - 1	sub ebx , 1
EAX := EAX∗EBX	mul ebx
:::::::::::::::::::::	:::::::::::::::::::

• $P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$

For simplicity, we will evaluate P(x) where n = 5 using the following formula:

$P(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = ((((a_5 x + a_4)x + a_3)x + a_2)x + a1)x + a_0$

$P(x) = a_nx^n + a_{n-1}x^{n-1} +... + a_3x^3 + a_2x^2 + a_1x + a_0 = (...(((a_n x + a_{4n-1})x +... + a_3)x + a_2)x + a1)x + a_0$

Example:

$p(2) = 7*2^5 + 4*2^4 + 2*2^3 + 10*2^2 + 8*2 + 3 = ((((7*2 + 4)*2 + 2)*2 + 10)*2 + 8)*2 + 3 = 363$

PSEUDO-CODE	AL PSEUDO-CODE	AL CODE	P	EAX	X
X:= 2d	X := 2d	mov x, 2			**2**
P:= 7d	P:= 7d	mov p, 7	**7**		2
P:= P∗X + 4d	EAX:= P	mov eax, p	7	7	2
	EAX:= EAX∗X	mul x	7	**14**	2
	EAX:= EAX + 4d	add eax, 4	7	**18**	2
	P:= EAX	mov p, eax	**18**	18	2
P:= P∗X + 2d	EAX:= P	mov eax, p	18	**18**	2
	EAX:= EAX∗X	mul x	18	**36**	2
	EAX:= EAX + 2d	add eax, 2	18	**38**	2
	P:= EAX	mov p, eax	**38**	38	2
P:= P∗X + 10d	EAX:= P	mov eax, p	38	**38**	2
	EAX:= EAX∗X	mul x	38	**76**	2
	EAX:= EAX + 10d	add eax, 10	38	**86**	2
	P:= EAX	mov p, eax	**86**	86	2
P:= P∗X + 8d	EAX:= P	mov eax, p	86	**86**	2
	EAX:= EAX∗X	mul x	86	**172**	2
	EAX:= EAX + 8d	add eax, 8	86	**180**	2
	P:= EAX	mov p, eax	**180**	180	2
P:= P∗X + 3d	EAX:= P	mov eax, p	180	**180**	2
	EAX:= EAX∗X	mul x	180	**360**	2
	EAX:= EAX + 3d	add eax, 3	180	**363**	2
	P:= EAX	mov p, eax	**363**	363	2

Exercises:

1. In this section, change all the numbers in the tables to hexadecimal.

2. Write assembly language algorithms that will compute:

a. sum = $a_1 + a_2 + ... + a_n$

b. Cn,m = n!/[(m!)(n - m)!]

■

Model Program

```
; This program computes 5!
.386
.model flat
.stack 4096
.data
factorial dword ?
.CODE
_start:
mov eax , 5
mov ebx, 5
sub ebx, 1
mul ebx
sub ebx, 1
mul ebx
sub ebx,1
mul ebx
sub ebx,1
mul ebx
mov factorial, eax
public _start
end
```

PROJECT

Write a general algorithm that can be used to convert any integer number $N_{10} \Rightarrow N_b$ where $b < 10$.

CHAPTER - 11 CONSTRUCTING PROGRAMS IN ASSEMBLY LANGUAGE PART 1

INTRODUCTION

In chapters 9 and 10 were given the basic of the assembly language code. From these basics we need to use the syntax to construct complete programs in assembly language. Professional programmers use several different methods for writing programs such as flow diagrams, pseudo-code, and several others. In this chapter we will use pseudo-code to guide us in writing assembly language program. We will employ a three step process:

Step 1: Analysis the objectives of the program.

Step 2: Convert the objectives of the program into pseudo-code.

Step 3: Convert the pseudo-code into assembly language pseudo-code (AL pseudo-code).

Step 4: Convert the AL pseudo-code into assembly language code.

To demonstrate these four steps, we will write programs to convert integer numbers from one base to another . In chapter 2, we developed the mathematics to convert bases. From chapter 2, we see that to convert numbers from an arbitrary base to the base 10, we need to evaluate

$a_nb^n + a_{n-1}b^{n-1} + ... + a_3b^3 + a_2b^2 + a_1b + a_0$

which is a polynomial of one variable.

However, in assembly language, there is no syntax that will directly allow us the perform exponential operations. The easiest way to evaluate the above expression is to linearize the polynomial.

Definition of linearizing a polynomial: Given a polynomial of one variable, we write:

$a_nb^n + a_{n-1}b^{n-1} + ... + a_3b^3 + a_2b^2 + a_1b + a_0 = (...(((a_nx + a_{4n-1})b + ... + a_3)b + a_2)b + a_1)b + a_0$

In the following number base conversions we will use the four steps, mentioned above.

11.1 An Assembly Language Program to Convert a Positive Integer Number In any Base b < 10 to its Corresponding Number in the Base 10.

Step 1: Analysis the objectives of the program.

To convert between integer number in any base b to its corresponding number in the base 10, we recall from chapter 2 the following formula:

$N_b = a_na_{n-1}... a_1a_0 \Leftrightarrow a_nb^n + a_{n-1}b^{n-1} + ... + a_1b + a_0$ base 10 .

Example:

The following manual method will convert the number 2567_8 to its correspond number in the base 10:

$N_8 = 2567_8 \Leftrightarrow ((2*8 + 5)*8 + 6)*8 + 7 = ((21)*8 + 6)*8 + 7 = 174*8 + 7 = 1399$

To convert the number 2567_8 to the base 10, we first need to write a sample program in pseudo-code and assembly language to capture the digits 2,5, 6,7 from the number. The following programs will perform such a task:

STEP 2: Convert the objectives of the program into pseudo-code.

Program: Capture the digits of 2567_8 .

PSEUDO-CODE	N	A	D
N:= 2567	**2567**		
D:= 1000	2567		**1000**
A:= N÷D	2567	**2**	1000
N:= N MOD D	**567**	2	1000
D:= 100	567	2	**100**
A:= N÷D	567	**5**	100
N:= N MOD D	**67**	5	100
D:= 10	67	5	**10**
A:= N÷D	67	**6**	10
N:= N MOD D	**7**	6	10
D:= 1	7	6	**1**
A:= N÷D	7	**7**	1
N:= N MOD D	0	7	0

Step 3: Convert the pseudo-code into assembly language pseudo-code (AL pseudo-code).

PSEUDO-CODE	AL PSEUDO-CODE	N	A	D	EAX	EDX
N:= 2567	N:= 2567	**2567**				
D:= 1000	D:=1000	2567		**1000**		
A:= N÷D	EAX:= N	2567		1000	**2567**	
	EAX:= EAX÷D	2567		1000	**2**	
	EDX:= EAX MOD D	2567		1000	2	**567**
	A:= EAX	2567	**2**	1000	2	567
N:= N MOD D	N:= EDX	**567**	2	1000	2	567
D:= 100	D:= 100	567	2	**100**	2	567

A:= N÷D	EAX:=N	567	2	100	**567**	567
	EAX:=EAX÷D	567	2	100	**5**	567
	EDX:= EAX MOD D	567	2	100	5	**67**
	A:=EAX	567	**5**	100	5	67
N:= N MOD D	N:= EDX	**67**	5	100	5	67
D:= 10	D:= 10	67	5	**10**	5	67
A:= N÷D	EAX:=N	67	5	10	**67**	67
	EAX:=EAX÷D	67	5	10	**6**	67
	EDX:= EAX MOD D	67	5	10	6	**7**
	A:=EAX	67	**6**	10	6	7
N:= N MOD D	N:= EDX	**7**	6	10	6	7
D:= 1	D:= 1	7	6	**1**	6	7
A:= N÷D	EAX:=N	7	6	1	**7**	7
	EAX:=EAX÷D	7	6	1	**7**	**7**
	EDX:= EAX MOD D	7	6	1	**7**	**0**
	A:=EAX	7	**7**	1	7	0
N:= N MOD D	N:= EAX	**7**	7	1	7	0

Step 4: Convert the AL pseudo-code into assembly language code.

PSEUDO-CODE	AL PSEUDO-CODE	ASSEMBLY LANGUAGE
N:= 2567	N:= 2567	mov n,2567
D:= 1000	D:=1000	mov d,1000
A:= N÷D	EAX:= N	mov eax,n
	EAX:= EAX÷D	mov edx,0
	EDX:= EAX MOD D	div d
	A:= EAX	mov a,eax
N:= N MOD D	N:= EDX	mov n,edx
D:= 100	D:= 100	mov d,100
A:= N÷D	EAX:=N	mov eax,n
	EAX:=EAX÷D	mov edx, 0

	EDX:= EAX MOD D	div d
	A:=EAX	mov a,eax
N:= N MOD D	N:= EDX	mov n,edx
D:= 10	D:= 10	mov d,10
A:= N÷D	EAX:=N	mov eax,n
	EAX:=EAX÷D	mov edx,0
	EDX:= EAX MOD D	div d
	A:=EAX	mov a,eax
N:= N MOD D	N:= EDX	mov n,edx
D:= 1	D:= 1	mov d,1
A:= N÷D	EAX:=N	mov eax,n
	EAX:=EAX÷D	mov edx,0
	EDX:= EAX MOD D	div d
	A:=EAX	mov a,eax
N:= N MOD D	N:= EDX	mov n,edx

Note: See model assembly language program.

Step 1: Analysis the objectives of the program.

Program: Writing a sample program to compute

$N_8 = 2567_8 \Rightarrow N = ((2*8+5)*8+6)*8+7 = 1399$

Step 2: Convert the objectives of the program into pseudo-code.

PSEUDO-CODE	**N**	**A**	**SUM**	**D**
N:= 2567	**2567**			
SUM:= 0	2567		**0**	
D:= 1000	2567		0	**1000**
A:= N÷D	2567	**2**	0	1000
N:= N MOD D	**567**	2	0	1000
SUM:= SUM + A	567	2	**2**	1000
SUM:= SUM∗8	567	2	**16**	1000

D:= 100	567	2	16	**100**
A:= N÷D	567	**5**	16	100
N:= N MOD D	**67**	5	16	100
SUM:= SUM + A	67	5	**21**	100
SUM:= SUM∗8	67	5	**168**	100
D:= 10	67	5	168	**10**
A:= N÷D	67	**6**	168	10
N:= N MOD D	**7**	6	168	10
SUM:= SUM + A	7	6	**174**	10
SUM:= SUM∗8	7	6	**1392**	10
DIVISOR:= 1	7	6	1392	**1**
A:= N÷D	7	**7**	1392	1
SUM:= SUM + A	7	7	**1399**	1

Step 3: Convert the pseudo-code into assembly language pseudo-code (AL pseudo-code).

PSEUDO-CODE	AL PSEUDO-CODE	N	A	SUM	D	EAX	EDX	E
N:= 2567	N:= 2567	**2567**						
E:= 8	E:= 8	2567						**8**
SUM:= 0	SUM:= 0	2567		**0**				8
D:= 1000	D:= 1000	2567		0	**1000**			8
A:= N÷D	EAX:= N	2567		0	1000	**2567**		8
	EAX:= EAX÷D	2567		0	1000	**2**		8
	EDX:= EAX MOD D	2567		0	100	2	567	8
	A:= EAX	2567	**2**	0	1000	2	567	8
N:= N MOD D	N:= EDX	**567**	2	0	1000	2	567	8
SUM:= SUM + A	EAX:= SUM	567	2	0	1000	**0**	567	8
	EAX:= EAX + A	567	2	0	1000	**2**	567	8
	SUM:= EAX	567	2	**2**	1000	2	567	8
SUM:= SUM∗E	EAX:= SUM	567	2	2	1000	**2**	567	8
	EAX:= EAX∗E	567	2	2	1000	**16**	**0**	8

	SUM:= EAX	567	2	**16**	1000	16	0	8
D:= 100	D:= 100	567	2	16	**100**	16	0	8
A:= N÷D	EAX:= N	567	2	16	100	**567**	0	8
	EAX:= EAX÷D	567	2	16	100	**5**	0	8
	EDX:= EAX MOD D	567	2	16	100	**5**	**67**	8
	A:= EAX	567	**5**	16	100	5	67	8
N:= N MOD D	N:= EDX	**67**	5	16	100	5	67	8
SUM:= SUM + A	EAX:= SUM	67	5	16	100	**16**	67	8
	EAX:= EAX + A	67	5	16	100	**21**	67	8
	SUM:= EAX	67	5	**21**	100	21	67	8
SUM:= SUM∗E	EAX:= SUM	67	5	21	100	**21**	67	8
	EAX:= EAX∗E	67	5	21	100	**168**	**0**	8
	SUM:= EAX	67	5	**168**	100	168	0	8
D:= 10	D:= 10	67	5	168	**10**	168	0	8
A:= N÷D	EAX:= N	67	5	168	10	**67**	0	8
	EAX:= EAX÷D	67	5	168	10	**6**	0	8
	EDX:= EAX MOD D	67	5	168	10	6	**7**	8
	A:= EAX	67	**6**	168	10	6	7	8
N:= N MOD D	N:= EDX	**7**	6	168	10	6	7	8
SUM:= SUM + A	EAX:= SUM	7	6	168	10	**168**	7	8
	EAX:= EAX + A	7	6	168	10	**174**	7	8
	SUM:= EAX	7	6	**174**	10	174	7	8
SUM:= SUM∗E	EAX:= SUM	7	6	174	10	**174**	7	8
	EAX:= EAX∗E	7	6	174	10	**1392**	7	8
	SUM:= EAX	7	6	**1392**	10	1392	7	8
D:= 1	D:= 1	7	6	1392	**1**	1392	7	8
A:= N÷D	EAX:= N	7	6	1392	1	**7**	0	8
	EAX:= EAX÷D	7	6	1392	1	**7**	0	8
	EDX:= EAX MOD D	7	6	1392	1	7	**0**	8
	A:= EAX	7	**7**	1392	1	7	0	8

SUM:= SUM + A	EAX:= SUM	7	7	1392	1	**1392**	0	8
	EAX:= EAX + A	7	7	1392	1	**1399**	0	8
	SUM:= EAX	7	7	**1399**	1	1399	0	8

Step 4: Convert the AL pseudo-code into assembly language code.

PSEUDO-CODE	**AL PSEUDO-CODE**	**AL CODE**
N:= 2567	N:= 2567	mov n, 2567
E:= 8	E:= 8	mov e, 8
SUM:= 0	SUM:= 0	mov sum, 0
D:= 1000	D:= 1000	mov d, 1000
A:= N÷D	EAX:= N	mov eax, n
	EAX:= EAX÷D	mov edx,0
	EDX:= EAX MOD D	div d
	A:= EAX	mov a, eax
N:= N MOD D	N:= EDX	mov n, edx
SUM:= SUM + A	EAX:= SUM	mov eax, sum
	EAX:= EAX + A	add eax, a
	SUM:= EAX	mov sum, eax
SUM:= SUM∗E	EAX:= SUM	mov eax, sum
	EAX:= EAX∗E	mul e
	SUM:= EAX	mov sum, eax
D:= 100	D:= 100	mov d, 100
A:= N÷D	EAX:= N	mov eax, n
	EAX:= EAX÷D	mov edx,0
	EDX:= EAX MOD D	div d
	A:= EAX	mov a, eax
N:= N MOD D	N:= EDX	mov n, edx
SUM:= SUM + A	EAX:= SUM	mov eax, sum
	EAX:= EAX + A	add eax, a
	SUM:= EAX	mov sum, eax

SUM:= SUM∗E	EAX:= SUM	mov eax, sum
	EAX:= EAX∗E	mul e
	SUM:= EAX	mov sum, eax
D:= 10	D:= 10	mov d, 10
A:= N÷D	EAX:= N	mov eax, n
	EAX:= EAX÷D	mov edx,0
	EDX:= EAX MOD D	div d
	A:= EAX	mov a, eax
N:= N MOD D	N:= EDX	mov n, edx
SUM:= SUM + A	EAX:= SUM	mov eax, sum
	EAX:= EAX + A	add eax, a
	SUM:= EAX	mov sum, eax
SUM:= SUM∗E	EAX:= SUM	mov eax, sum
	EAX:= EAX∗E	mul e
	SUM:= EAX	mov sum, eax
D:= 1	D:= 1000	mov d, 100
A:= N÷D	EAX:= N	mov eax, n
	EAX:= EAX÷D	mov edx,0
	EDX:= EAX MOD D	div d
	A:= EAX	mov a, eax
SUM:= SUM + A	N:= EDX	mov eax, sum
	EAX:= SUM	add eax, a
	EAX:= EAX + A	mov sum,eax

Exercises:

1. Modify the above assembly language program to convert the number 5632_8 to the corresponding number in the base 10.

2. Modify the above assembly language program to convert the number 1101_2 to the corresponding number in the base 10.

■

11.2 An Algorithm to Convert any Integer Number in the Base 10 to a Corresponding Number in the Base b < 10.

Step 1: Analysis the objectives of the program.

Using the Euclidean division theorem, we now review how, using the manual method to convert numbers in the base 10 to any in the base b.

Step 1: We want to write N in the form: $N = a_n b^n + a_{n-1} b^{n-1} \ldots + a_1 b + a_0$

Step 2: $N = Qb + R = (a_n b^{n-1} + a_{n-1} b^{n-2} \ldots + a_1)b + a_0$

Here, $Q = a_n b^{n-1} + a_{n-1} b^{n-2} \ldots + a_2 b + a_1 = (a_n b^{n-2} + a_{n-1} b^{n-3} \ldots + a_2)b + a_1$ and $R = a_0$

Step 3: Set N = Q.

$Q = Q_1 b + R_1 = (a_n b^{n-2} + a_{n-1} b^{n-3} \ldots + a_2)b + a_1$ where

$Q_1 = a_n b^{n-2} + a_{n-1} b^{n-3} \ldots + a_2$,

$R_1 = a_1$.

Step 4: Continue in this manner, until $Q_n = 0$.

Example:

Convert the following decimal numbers to the specified base.

1. $1625 \Rightarrow N_8$

Step 1: $1625 = (1625 \div 8) * 8 + 1 = 203 * 8 + 1$

$a_0 = 1$

Step 2: $203 = (203 \div 8) * 8 + 3 = 25 * 8 + 3$

$a_1 = 3$

Step 3: $25 = (25 \div 8) * 8 + 1 = 3 * 8 + 1$

$a_2 = 1$

Step 4: $3 = (3 \div 8) * 8 + 3 = 3$

$a_3 = 3$

Therefore, $1625 \Rightarrow N_8 = 3 * 8^3 + 1 * 8^2 + 3 * 8 + 1 \Leftrightarrow N_8 = 3131_8$

Program: To convert the integer number 1625 to the base 8.

Step 2: Convert the objectives of the program into pseudo-code.

PSEUDO-CODE	N	SUM	TEN	MUL	BASE	R
BASE := 8					**8**	
N := 1625	**1625**				8	
SUM := 0	1625	**0**			8	
MUL := 1	1625	0		**1**	8	
TEN := 10	1625	0	**10**	1	8	
R := N MOD BASE	1625	0	10	1	8	**1**
N:= N÷BASE	**203**	0	10	1	8	1
R := R∗MUL	203	0	10	1	8	**1**
SUM:= SUM + R	203	**1**	10	1	8	1
MUL:= MUL∗TEN	203	1	10	**10**	8	1
R := N MOD BASE	203	1	10	10	8	**3**
N:= N÷BASE	**25**	1	10	10	8	3
R := R∗MUL	25	1	10	10	8	**30**
SUM:= SUM + R	25	**31**	10	10	8	30
MUL:= MUL∗TEN	25	31	10	**100**	8	30
R := N MOD BASE	25	31	10	100	8	**1**
N:= N÷BASE	**3**	31	10	100	8	1
R := R∗MUL	3	31	10	100	8	**100**
SUM:= SUM + R	3	**131**	10	100	8	100
MUL:= MUL∗TEN	3	131	10	**1000**	8	100
R := N MOD BASE	3	131	10	1000	8	**3**
N:= N÷BASE	**0**	131	10	1000	8	3
R := R∗MUL	0	131	10	1000	8	**3000**
SUM:= SUM + R	0	**3131**	10	1000	8	3000

Step 3: Convert the pseudo-code into assembly language pseudo-code (AL pseudo-code).

PSEUDO-CODE	AL PSEUDO-CODE	N	S	M	R	EAX	EDX	B	T
B := 8	B := 8							**8**	
N := 1625	N := 1625	**1625**						8	
S:= 0	S:= 0	1625	**0**					8	
M:= 1	M:= 1	1625	0	**1**				8	
T:= 10	T:= 10	1625	0	1				8	**10**
R := N MOD B	EAX:= N	1625	0	1		**1625**		8	10
	EAX:= EAX÷B	1625	0	1		**203**		8	10
	EDX:= EAX MOD B	1625	0	1		203		8	10
	R:= EDX	1625	0	1	**1**	203	1	8	10
N:= N÷B	N:= EAX	**203**	0	1	1	203	1	8	10
R := R∗M	EAX:= R	203	0	1	1	**1**	1	8	10
	EAX:= EAX∗M	203	0	1	1	**1**	1	8	10
	R:= EAX	203	0	1	**1**	1	1	8	10
S:= S + R	EAX:= S	203	0	1	1	**0**	1	8	10
	EAX:= EAX+ R	203	0	1	1	**1**	1	8	10
	S:= EAX	203	**1**	1	1	1	1	8	10
M:= M∗T	EAX:= M	203	1	1	1	**1**	1	8	10
	EAX:= EAX∗T	203	1	1	1	**10**	1	8	10
	M:= EAX	203	1	**10**	1	10	1	8	10
R := N MOD B	EAX:= N	203	1	10	1	**203**	1	8	10
	EAX:= EAX÷B	203	1	10	1	**25**	1	8	10
	EDX:= EAX MOD B	203	1	10	1	25	3	8	10
	R:= EDX	203	1	10	**3**	25	3	8	10
N:= N÷B	N:= EAX	**25**	1	10	3	25	3	8	10
R := R∗M	EAX:= R	25	1	10	3	**3**	**1**	8	10
	EAX:= EAX∗M	25	1	10	3	**30**	1	8	10
	R:= EAX	25	1	10	**30**	30	1	8	10
S:= S + R	EAX:= S	25	1	10	30	**1**	1	8	10

	EAX:= EAX + R	25	1	10	30	**31**	1	8	10
	S:= EAX	25	**31**	10	30	31	1	8	10
M:= M∗T	EAX:= M	25	31	10	30	**10**	1	8	10
	EAX:= EAX∗T	25	31	10	30	**100**	**0**	8	10
	M:= EAX	25	31	**100**	30	100	0	8	10
R := N MOD B	EAX:= N	25	31	100	30	**25**	0	8	10
	EAX:= EAX÷B	25	31	100	30	**3**	0	8	10
	EDX:= EAX MOD B	25	31	100	30	3	**1**	8	10
	R:= EDX	25	31	100	**1**	3	1	8	10
N:= N÷B	N:= EAX	**3**	31	100	1	3	1	8	10
R := R∗M	EAX:= R	3	31	100	1	**1**	1	8	10
	EAX:= EAX∗M	3	31	100	1	**100**	**0**	8	10
	R:= EAX	3	31	100	**100**	100	0	8	10
S:= S + R	EAX:= S	3	31	100	100	**31**	1	8	10
	EAX:= EAX + R	3	31	100	100	**131**	1	8	10
	S:= EAX	3	**131**	100	100	131	1	8	10
M:= M∗T	EAX:= M	3	131	100	100	**100**	1	8	10
	EAX:= EAX∗T	3	131	100	100	**1000**	**0**	8	10
	M:= EAX	3	131	**1000**	100	1000	0	8	10
R := N MOD B	EAX:= N	3	131	1000	100	**3**	0	8	10
	EAX:= EAX÷B	3	131	1000	100	**0**	0	8	10
	EDX:= EAX MOD B	3	131	1000	100	0	**3**	8	10
	R:= EDX	3	131	1000	**3**	**0**	3	8	10
N:= N÷B	N:= EAX	**0**	131	1000	3	0	3	8	10
R := R∗M	EAX:= R	0	131	1000	3	**3**	3	8	10
	EAX:= EAX∗M	0	131	1000	3	**3000**	**0**	8	10
	R:= EAX	0	131	1000	**3000**	3000	3	8	10
S:= S + R	EAX:= S	0	131	1000	3000	**131**	3	8	10
	EAX:= EAX + R	0	131	1000	3000	**3131**	3	8	10
	S:= EAX	0	**3131**	1000	3000	3131	3	8	10

$1625 \Rightarrow 3131_8$

Step 4: Convert the AL pseudo-code into assembly language code.

PSEUDO-CODE	**AL PSEUDO-CODE**	**AL CODE**
B := 8	B := 8	mov b, 8
N := 1625	N := 1625	mov n, 1625
S:= 0	S:= 0	mov s, 0
M:= 1	M:= 1	mov m, 1
T:= 10	T:= 10	mov t, 10
R := N MOD B	EAX:= N	mov eax, n
	EAX:= EAX÷B EDX:= EAX MOD B	mov edx,0 div b
	R:= EDX	mov r, edx
N:= N÷B	N:= EAX	mov n, eax
R := R∗M	EAX:= R	mov eax, r
	EAX:= EAX∗M	mul m
	R:= EAX	mov r, eax
S:= S + R	EAX:= S	mov eax, s
	EAX:= EAX + R	add eax, r
	S:= EAX	mov s, eax
M:= M∗T	EAX:= M	mov eax, m
	EAX:= EAX∗T	mul t
	M:= EAX	mov m eax
R := N MOD B	EAX:= N	mov eax, n
	EAX:= EAX÷B EDX:= EAX MOD B	mov edx,0 div b
	R:= EDX	mov r, edx
N:= N÷B	N:= EAX	mov n, eax
R := R∗M	EAX:= R	mov eax, r
	EAX:= EAX∗M	mul m

	R:= EAX	mov r, eax
S:= S + R	EAX:= S	mov eax, s
	EAX:= EAX + R	add eax, r
	S:= EAX	mov s, eax
M:= M∗T	EAX:= M	mov eax, m
	EAX:= EAX∗T	mul t
	M:= EAX	mov m eax
R := N MOD B	EAX:= N	mov eax, n
	EAX:= EAX÷B EDX:= EAX MOD B	mov edx,0 div b
	R:= EDX	mov r, edx
N:= N÷B	N:= EAX	mov n, eax
R := R∗M	EAX:= R	mov eax, r
	EAX:= EAX∗M	mul m
	R:= EAX	mov r, eax
S:= S + R	EAX:= S	mov eax, s
	EAX:= EAX + R	add eax, r
	S:= EAX	mov s, eax
M:= M∗T	EAX:= M	mov eax, m
	EAX:= EAX∗T	mul t
	M:= EAX	mov m eax
R := N MOD B	EAX:= N	mov eax, n
	EAX:=EAX÷B EDX:= EAX MOD B	mov edx,0 div b
	R:= EDX	mov r, edx
N:= N÷B	N:= EAX	mov n, eax
R := R∗M	EAX:= R	mov eax, r
	EAX:= EAX∗M	mul m
	R:= EAX	mov r, eax
S:= S + R	EAX:= S	mov eax, s

	EAX:= EAX + R	add eax, r
	S:= EAX	mov s, eax

Exercises:

1. Use the above algorithm to write a program to convert the decimal number 2543_{10} to octal.

2. Write an algorithm to convert decimal number a_1a_0 to the base 2.

■

Model Assembly Language Program: Capture the digits of 2567_8 (See program in 11.1)

```
; This program Capture the digits of 2567₈

.386
.model flat

.stack 4096
.data
n dword ?
d dwoprd ?
a dword ?

.code
_start:
mov n, 2567
mov d, 1000
mov eax, n
div d
mov a, eax
mov n, edx
mov d, 100
mov eax, n
div d
mov a, eax
mov n, edx
mov d, 10
mov eax, n
div d
mov a, eax
mov n, edx
```

```
mov d, 1
mov eax, n
div d
mov a, eax
mov n, edx

public _start

end
```

Project

Modify the above pseudo-code programs with appropriate WHILE statements to make the programs as general as possible.

We are now ready to study the necessary assembly language instructions to convert the While-Conditional and If-Then pseudo-codes, defined in chapter 5, to assembly code. To do this conversion, we need two types of jump instructions: conditional jump instructions and a unconditional jump instruction.

12.1 Conditional Jump Instructions for Signed Order:

The basic form in assembly language consists[1] of 2 instructions:

• **The compare instructions:**

cmp *operand1*, *operand2,*

• **The conditional jump instructions:**

jump *j* <u>*condition*</u> *label*

The above instruction are always written in the above order.

The operands can be numeric values, registers, variables.

The Compare(cmp) Instructions

The following table gives the type of operand1, operand2 that are allowed.

OPERAND1	OPERAND2
register 8 bits (byte)	numeric byte register 8 bits variable byte
register 16 bits (word)	numeric byte numeric word register 16 bits (word) variable word
register 32 bits (dword)	numeric byte numeric dword register 32 bits (word) variable dword

[1] There exists additional jump instructions in assembly language which will be discussed in later chapters.

variable byte: 8 bits (byte)	numeric byte register 8 bits (byte)
variable word :16 bits (word)	numeric byte numeric word register word
variable dword: 32 bits	numeric byte numeric dword register 32 bits
AL AX EAX	numeric byte numeric word numeric dword

Note: The instruction cmp x,y are not valid in assembly language.

Examples:

1.
x dword 236
cmp eax, x

2.
cmp ebx, eax

3.
cmp x, eax

4.
cmp x, 25767h

Exercises:

1. Which of the following are valid. If not indicate why.

a.	**b.**	**c.**	**d.**	**e.**
x dword 456h y dword 44444h cmp x,y	cmp eax, x	cmp x, eax	cmp x, 235	cmp 235, x

■

The conditional jump instructions for signed order numbers.

To perform the pseudo-code WHILE statement in assembly language, we now introduce for signed order numbers, the conditional jump instructions.

From Chapter 8, the following are the signed order of the numbers for the three types of rings:

• The binary ring (8 bits)

$\Longrightarrow$

R_{10}	128 <	129 <	130 <	---	255 <	0 <	1 <	2 <	---	126 <	127
R_8	80 <	81 <	82 <	---	FF <	00 <	01 <	02 <	---	7E <	7F

• The word ring (16 bits)

$\Longrightarrow$

R_{10}	32768 <	32769 <	32770 <	---	65535 <	0 <	1 <	2 <	---	32766<	32767
R_{16}	80 00 <	80 01 <	80 02 <	---	FF FF <	00 00 <	00 01 <	00 02 <	---	7F FE	7F FF

• The dword ring (32 bits)

$\Longrightarrow$

R_{10}	2147483648<	2147483649<	---	4,294,967,295<	0 <	1 <	---	2147483647
R_{32}	80 00 00 00 <	80 00 00 01 <	---	FF FF FF FF<	00 00 00 00 00<	00 00 00 01<	---	7F FF FF FF

The following is a table of the conditional jumps for the signed order of rings in assembly language:

Mnemonic[1]	**Description**
je	jump to the label if *operand1* = *operand* 2; *jump if equal to*
jne	jump to the label if *operand1* ≠ *operand 2; jump if not equal to*
jnge	jump to the label if *operand1* < *operand* 2; *jump if not greater or equal*
jnle	jump to the label if *operand1* > *operand* 2; *jump if not less than or equal*
jge	jump to the label if *operand1* ≥ *operand* 2; *jump if greater than or equal*
jle	jump to the label if *operand1* ≤ *operand* 2; *jump if less than or equal*

jl	jump to the label if *operand1* < *operand* 2; *jump if less than*
jnl	jump to the label if *operand1* ≥ *operand* 2; *jump if not less than*
jg	jump to the label if *operand1* > *operand* 2; *jump if greater than*
jng	jump to the label if *operand1* ≤ *operand* 2; *jump if not greater than*

All of the above jump instructions **must** be preceded by the cmp instruction.

Examples:

1.
mov al,10 ; al is operand1
cmp al,2;   2 is operand2
je xyz ; since the contents of al is not equal to 2, a jump does not occur.
::::::::::::::::: ; instructions
xyz: ; a label

2.
mov al, 10; al is operand1
cmp al,2 ; 2 is operand2
jne xyz ; since the contents of al is not equal to 2, a jump occurs.
::::::::::::::::: ; instructions
xyz: ; a label

3.
mov ax,32770 ; ax is operand1
cmp ax,2; 2 is operand2
jnge xyz ; since the contents of ax is not greater than 2, a jump does occur.

::::::::::::::::: ; instructions
xyz: ; a label

4a.
mov eax,80000000h; eax is operand1,
cmp al,2;   2 is operand2
jge xyz ; since the contents of al is not greater than or equal to 2, a jump does not occur occurs.
::::::::::::::::: ; instructions

xyz: ; a label

4b.
mov al,0 ; al is operand1
cmp al,129;   129 is operand2
jge xyz ; since the contents of al is greater than or equal to 129, a jump occurs.
::::::::::::::::: ; instructions
xyz: ; a label

5a.
mov al,255 ; al is operand1
cmp al,2;   2 is operand2
jle xyz ; since the contents of al is less than or equal to 2, a jump occurs.
::::::::::::::::: ; instructions
xyz: ; a label

5b.
mov al,2 ; al is operand1
cmp al,255;   255 is operand2
jle xyz ; since the contents of al is greater than 255, a jump does not occurs.
::::::::::::::::: ; instructions
xyz: ; a label

6.
mov al,10 ; al is operand1
cmp al,2;   2 is operand2
jnle al ; since the contents of al is not less than or equal to 2, a jump occurs.
::::::::::::::::: ; instructions
xyz: ; a label

7.
mov al,128 ; al is operand1
cmp al,255;   255 is operand2
jl xyz ; since the contents of al is less than 255, a jump occurs.
::::::::::::::::: ; instructions
xyz: ; a label

8.
mov al,10 ; al is operand1
cmp al,2;   2 is operand2

jnl xyz ; since the contents of al is not less than 2, a jump occurs.
:::::::::::::::: ; instructions
xyz: ; a label

9.
mov al,10 ; al is operand1
cmp al,2; 2 is operand2
jg xyz ; since the contents of al is greater than 2, a jump occurs.
:::::::::::::::: ; instructions
xyz: ; a label

10.

mov al,10 ; al is operand1
cmp al,2; 2 is operand2
jng xyz ; since the contents of al is greater than 2, a jump does not occur.
:::::::::::::::: ; instructions
xyz: ; a label

Exercises: Assume al contains the number 5 and n also contains 5. Which of the following incomplete programs will cause a jump:

1. cmp al,n je xyz xyz:	**2.** cmp al,n jne xyz xyz:	**3.** cmp al,n jnge xyz xyz:
4. cmp al,n jge xyz. xyz:	**5.** cmp al,n jle xyz . xyz	**6.** cmp al,n jnle al xyz
7. cmp al,n jl xyz xyz	**8.** cmp al,n jnl xyz xyz:	**9.** cmp al,n jg xyz xyz:

10.
cmp al,n;
jng xyz.

■
The unconditional jump instruction:

The form of the unconditional jump instruction is

jmp *label;* a jump will automatically occur.

Example:

```
jmp xyz ;
::::::::::::::::: ; instructions
xyz: ; a label
```

The conditional jump instructions for the natural order (unsigned) .

From Chapter 8, the following are the natural order of the numbers for the three types of rings:

- The binary ring (8 bits)

$\Longrightarrow$

R_{10}	0 <	1 <	2 < ...	15 <	16 <	17 < ...	240 < ...	254 <	255
R_8	00 <	01 <	02 < ...	0F <	10 <	11 < ...	F0 < ---	FE <	FF

- The word ring (16 bits)

$\Longrightarrow$

R_{10}	0 <	1 <	2 < ...	255 <	256 < ...	511 <...	65280 <	---	65534 <	65535
R_{16}	00 00 <	00 01 <	00 02 < ...	00 FF <	01 00 < ...	01 FF <...	FF 00 <	---	FF FE	FF FF

- The dword ring (32 bits)

$\Longrightarrow$

R_{10}	0<	15< ---	255 < ---	65535 <	16777215 <	2147483647
R_{32}	00 00 00 00 <	00 00 00 0F <---	00 00 00 FF < ---	00 00 FF FF< ---	00 FF FF FF<---	FF FF FF FF

The following is a table of the conditional jumps for the natural order of rings (unsigned) in assembly language:

Mnemonic	Description
je	jump to the label if *operand1* = *operand* 2; *jump if equal to*
jne	jump to the label if *operand1* ≠ *operand 2; jump if not equal to*
jae	jump to the label if *operand1* ≥ *operand 2; jump if greater than or eqaul*

ja	jump to the label if *operand1 > operand 2; jump if greater than*
jbe	jump to the label if *operand1 ≤ operand 2; jump if less than or equal*
jna	jump to the label if *operand1 ≤ operand 2; jump if less than or equal*
jb	jump to the label if *operand1 < operand 2; jump if less than*
jnb	jump to the label if *operand1 ≥ operand 2; jump if greater than or equal*
jnae	jump to the label if *operand1 < operand 2; jump if less than*
jnbe	jump to the label if *operand1 > operand 2; jump if greater than*

Examples:

1.
mov al,10 ; al is operand1
cmp al,2; 2 is operand2
je xyz ; since the contents of al is not equal to 2, a jump does not occur.
:::::::::::::::::: ; instructions
xyz: ; a label

2.
mov al,10 ; al is operand1
cmp al,2; 2 is operand2
jne xyz ; since the contents of al is not equal to 2, a jump occurs.
:::::::::::::::::: ; instructions
xyz: ; a label

3.
mov al,210 ; al is operand1
cmp al,2; 2 is operand2
ja xyz ; since the contents of al is greater than 2, a jump occurs.

:::::::::::::::::: ; instructions
xyz: ; a label

4.
mov al,10 ; al is operand1
cmp al,2; 2 is operand2
jae xyz ; since the contents of al is greater than or equal to 2, a jump occurs.

::::::::::::::::: ; instructions
xyz: ; a label

5.
mov al,2 ; al is operand1
cmp al,2;   255 is operand2
jbe xyz ; since the contents of al is less than or equal to 2, a jump occurs.
::::::::::::::::: ; instructions
xyz: ; a label

7.
mov al,128 ; al is operand1
cmp al,255;   255 is operand2
jbe xyz ; since the contents of al is less than 255, a jump occurs.
::::::::::::::::: ; instructions
xyz: ; a label

8.
mov al,10 ; al is operand1
cmp al,2;   2 is operand2
je xyz ; since the contents of al is not equal to   2, a jump does not occurs.
::::::::::::::::: ; instructions
xyz: ; a label

9.
mov al,10 ; al is operand1
cmp al,2;   2 is operand2
jne xyz ; since the contents of al is not equal to 2, a jump occurs.
::::::::::::::::: ; instructions
xyz: ;   a label

12.2: Converting the While-Conditional Statements to Assembly Language

We will use the pseudo-code examples from Chapter 6 to demonstrate how the jump instructions can be used to convert While Statements.

Example:

Write a partial program that will sum the numbers from 1 to 6.

PSEUDO CODE	CYCLE OF INSTRUCTIONS	TOTAL	N
N := 1	N := 1		**1**
TOTAL := 0	TOTAL := 0	**0**	1
WHILE N <= 6	WHILE N <= 6	0	1

BEGIN	**BEGIN**	0	1
TOTAL := TOTAL + N	TOTAL := TOTAL + N	**1**	1
N := N + 1	N := N + 1	1	**2**
	TOTAL := TOTAL + N	**3**	2
	N := N + 1	3	**3**
	TOTAL := TOTAL + N	**6**	3
	N := N + 1	6	**4**
	TOTAL := TOTAL + N	**10**	4
	N := N + 1	10	**5**
	TOTAL := TOTAL + N	**15**	5
	N := N + 1	15	**6**
	TOTAL := TOTAL + N	**21**	6
	N := N + 1	21	**7**
END	**END**	21	7

PSEUDO CODE	AL PSEUDO CODE CYCLE OF INSTRUCTION	TOTAL	N	EAX
N := 1	N := 1		**1**	
TOTAL := 0	TOTAL := 0	**0**	1	
WHILE N <= 6	WHILE N <= 6	0	1	
BEGIN	**BEGIN**	0	1	
TOTAL := TOTAL + N	EAX := TOTAL	0	1	**0**
	EAX:= EAX + N	0	**1**	**1**
	TOTAL := EAX	**1**	1	1
N := N + 1	EAX := N	1	1	**1**
	EAX := EAX + 1	1	2	**2**
	N:= EAX	1	**2**	2
	EAX := TOTAL	1	2	**1**
	EAX:= EAX + N	2	2	**3**
	TOTAL := EAX	**3**	2	3

	EAX := N	3	2	**2**
	EAX := EAX + 1	3	2	**3**
	N:= EAX	3	**3**	3
	EAX := TOTAL	3	3	**3**
	EAX:= EAX + N	3	3	**6**
	TOTAL := EAX	**6**	3	6
	EAX := N	6	3	**3**
	EAX := EAX + 1	6	3	**4**
	N:= EAX	6	**4**	4
	EAX := TOTAL	6	4	**6**
	EAX:= EAX + N	6	4	**10**
	TOTAL := EAX	**10**	4	10
	EAX := N	10	4	**4**
	EAX := EAX + 1	10	4	**5**
	N:= EAX	10	**5**	5
	EAX := TOTAL	10	5	**10**
	EAX:= EAX + N	10	5	**15**
	TOTAL := EAX	**15**	5	15
	EAX := N	15	5	**5**
	EAX := EAX + 1	15	5	**6**
	N:= EAX	15	**6**	6
	EAX := TOTAL	15	6	**15**
	EAX:= EAX + N	15	6	**21**
	TOTAL := EAX	**21**	6	21
	EAX := N	21	6	**6**
	EAX := EAX + 1	21	6	**7**
	N:= EAX	21	**7**	7
END	**END**	21	7	7

PSEUDO CODE	**AL PSEUDO CODE**	**ASSEMBLY CODE**
N:= 1	N := 1	mov n, 1
TOTAL:= 0	TOTAL := 0	mov total, 0
WHILE N <= 6	WHILE N <= 6	while: cmp n, 6
BEGIN	**BEGIN**	jg end
TOTAL:= TOTAL + N	EAX := TOTAL	mov eax, total
	EAX:= EAX + N	add eax, n
	TOTAL := EAX	mov total, eax
N:= N + 1	EAX := N	mov eax, n
	EAX := EAX + 1	add eax, 1
	N:= EAX	mov n, eax
END	**END**	jmp while end:

Exercises:

1. Rewrite the above program in a AL pseudo-code where only registers (not variables) are used.

2. Modify the above program by replacing jg with jle .

3. Modify the above program by changing the pseudo code

T0TAL := TOTAL + N

with

N := N - 1

4. Modify the above program that would allow the user to sum an arbitrary number.

5. The number 1 + 2+ 3+ ... + n = n(n + 1)/2. Modify the above program to check if the program is add correctly and inform the user if it is or is not working correctly.

6. Write a program to compute

$1^2 + 2^2 + 3^2 + ... + N^2$

for a given positive integer N.
■

Example:

Program: will compute the length of the number 431

INSTRUCTIONS	CYCLE OF INSTRUCTIONS	N	COUNT
N: = 431	N := 431	**431**	
COUNT := 0	COUNT := 0	431	**0**
WHILE N <> 0	WHILE N <>0	431	0
BEGIN	**BEGIN**	431	0
COUNT := COUNT + 1	COUNT := COUNT + 1	431	**1**
N := N÷10	N := N÷10	**43**	1
	COUNT := COUNT + 1	43	**2**
	N := N÷10	**4**	2
	COUNT := COUNT + 1	4	**3**
	N := N÷10	0	3
END	**END**	0	3

PSEUDO CODE	AL PSEUDO CODE CYCLE	N	COUNT	EAX	EDX	TEN
TEN:= 10	TEN:= 10					**10**
N: = 431	N := 431	**431**				10
COUNT := 0	COUNT := 0	431	**0**			10
WHILE N <> 0	WHILE N <>0	431	0			10
BEGIN	**BEGIN**	431	0			10
COUNT := COUNT + 1	EAX: = COUNT	431	0	**0**		10
	EAX:= EAX + 1	431	0	**1**		10
	COUNT:= EAX	431	**1**	1		10
N := N÷TEN	EAX:= N	431	1	**431**		10
	EAX:= EAX ÷TEN	431	1	**43**		10
	EDX:= EAX MOD10	431	1	43	**1**	10
	N:= EAX	**43**	1	43	1	10
	EAX: = COUNT	43	1	**1**	1	10
	EAX:= EAX + 1	43	1	**2**	1	10
	COUNT:= EAX	43	**2**	2	1	10

	EAX:= N	43	2	**43**	1	10
	EAX:= EAX ÷TEN	43	2	**4**	1	10
	EDX:= EAX MOD10	43	2	4	**3**	10
	N:= EAX	**4**	2	4	3	10
	EAX: = COUNT	4	2	2	3	10
	EAX:= EAX + 1	4	2	**3**	3	10
	COUNT:= EAX	4	**3**	3	3	10
	EAX:= N	4	3	**4**	3	10
	EAX:= EAX ÷TEN	4	3	**0**	**4**	10
	N:= EAX	**0**	3	0	4	10
END	**END**	0	3	0	4	10

PSEUDO INSTRUCTIONS	AL PSEUDO CODE	ASSEMBLY CODE
TEN:= 10	TEN:= 10	mov ten, 10
N: = 431	N: = 431	mov n, 431
COUNT := 0	COUNT := 0	mov count, 0
WHILE N <> 0	WHILE N <> 0	while: cmp n, 0
BEGIN	**BEGIN**	begin: je end
COUNT:= COUNT + 1	EAX: = COUNT	mov eax, count
	EAX:= EAX + 1	add eax, 1
	COUNT:= EAX	mov count, eax
N÷TEN	EAX:= N	mov eax, n
	EAX:= EAX ÷TEN	mov edx, 0
		div ten
	N:= EAX	mov n, eax
END	**END**	jmp while end:

Exercises:

1. Modify the above program so that it will perform the following tasks:

Task 1: The user will enter a positive integer.

Task 2: The program will count the number digits of the integer.

Task 3: The number of digits will be outputted to the monitor.

2. Write a program that will perform the following tasks:

Task 1: The user will enter a positive integer N.

Task 2: The program will computer the sum: $1 + 2^2 + 3^2 + ... + N^2$.

Task 3: The sum will be outputted to the monitor.

■

12.3: IF-THEN STATEMENTS

The assembly language does not have an If-THEN statement as defined in higher programming languages. However, we can obtain many of the same results by using the jump instructions as defined above. The following table, gives the instructions on how to emulate in many of the IF-THEN statements:

PSEUDO IF-THEN INSTRUCTIONS	**JUMP INSTRUCTIONS**

IF *operand1* > *oprand* 2 THEN
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin: jng end
	(instructions)
	end:

IF *operand1* ≥ *oprand* 2 THEN
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin: jnge end
	(instructions)
	end:

IF *operand1* = *oprand* 2 THEN
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin: jne end
	(instructions)
	end:

IF *operand1* ≠ *oprand* 2 THEN
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin: je end
	(instructions)
	end:

IF *operand1* < *oprand* 2 THEN
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin: jnl end
	(instructions)
	end:

IF *operand1* ≤ *oprand* 2 THEN
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin: jg end
	(instructions)
	end:

Example:

1.

The following program will perform the following tasks:

Task 1: Check if the number 12103 is divisible by 7.

Task 2: If 7 divides, then place 0 in x

PSEUDO- INSTRUCTIONS	**Y**	**X**	**S**
X := 12103		**12103**	
S := 7		12103	**7**
Y := (X÷S)*S	**12103**	12103	7
IF X = Y THEN	12103	12103	7
BEGIN	12103	12103	7
X := 0	12103	**0**	7
END	1203	0	7

PSEUDO- INSTRUCTIONS	**AL PSEUDO-CODE**	**Y**	**X**	**S**	**EAX**	**EDX**
X := 12103	X:= 12103		**12103**			
S:= 7	S := 7		12103	**7**		
Y := (X÷S)*S	EAX:= X		12103	7	**12103**	
	EAX:= EAX÷S		12103	7	**1729**	
	EDX:= EAX MOD S		12103	7	1729	**0**
	EAX:= EAX∗S		12103	7	**12103**	0
	Y:= EAX	**12103**	12103	7	12103	0
IF X = Y THEN	EAX:= X	12103	12103	7	1729	0
	CMP EAX, Y	12103	12103	7	1729	0
	JNE END	12103	12103	7	1729	0
BEGIN	**BEGIN**	12103	12103	7	1729	0
X := 0	X := 0	12103	**0**	7	1729	0
END	**END**	1203	0	7	1729	0

PSEUDO- INSTRUCTIONS	AL PSEUDO-CODE	AL INSTRUCTIONS
X := 12103	X:= 12103	mov x, 12103
S:= 7	S := 7	mov s, 7
Y := (X÷S)*S	EAX:= X	mov eax, x
	EAX:= EAX÷S EAX := EAX*S	mov edx, 0
		div s
		mul s
IF X = Y THEN	Y:= EAX	mov y,eax
	EAX:= X	mov eax,x
	CMP EAX, Y	cmp eax, y
	JNE END	jne end
BEGIN	**BEGIN**	;begin
X := 0	X := 0	mov x, 0
END	**END**	end:

Exercises:

1.
From Chapter 5, we have the following algorithm.

PSEUDO - INSTRUCTIONS	EXPLANATION
LARGEST := X1	We start by assuming X1 is the largest

IF X2 > LARGEST THEN
BEGIN
LARGEST := X2
END
	If the contents of X2 is larger than the contents of LARGEST replace LARGEST with the contents of X2

IF X3 > LARGEST THEN
BEGIN
LARGEST := X3
END
	If the contents of X3 is larger than the contents of LARGEST replace LARGEST with the contents of X3

Using the above algorithm, write an assembly language program that will perform the following tasks:

Task1: Assign 2 positive integer numbers.

Task2: Find the largest of the 2 numbers entered.

Task3: Output the largest number.

Write the assembly language code to replicate the pseudo-code:

2.
IF a < x $\leq$ b THEN
BEGIN
:::::::::::::::
END

3.
IF x = a or x = b THEN
BEGIN
::::::::::::::::
END
■

12.4: IF-THEN - ELSE STATEMENTS

Recall from Chapter 5 the form of this conditional statement:

IF *conditional expression* THEN
BEGIN statements 1
END
ELSE
BEGIN
statements 2
END

If the conditional expression is *TRUE,* statements1 following the THEN will be carried out and the program will skip statements 2.

If the conditional expression is *FALSE,* statements 1 following the THEN will not be carried out and the program will execute statements 2.

Since the assembly language does not have the IF-THEN-ELSE statements, the following table shows how the jumps can be used to simulate this type of instruction:

IF-THEN-ELSE PSEUDO - INSTRUCTIONS	JUMP INSTRUCTIONS

IF *operand1* > *operand* 2 THEN
BEGIN
(instructions)
END
ELSE
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin1: jng end1
	(instructions)
	end1: jg end2
	(instructions)
	end2:

IF *operand1* ≥ *operand* 2 THEN
BEGIN
(instructions)
END
ELSE
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin1: jnge end1
	(instructions)
	end1:jge end2
	(instructions)
	end2:

IF *operand1* = *operand* 2 THEN
BEGIN
(instructions)
END
ELSE
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin1: jne end1
	(instructions)
	end1:je end2
	(instructions)
	end2:

IF *operand1* ≠ *operand* 2 THEN
BEGIN
(instructions)
END
ELSE
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin1: je end1
	(instructions)
	end1:jne end2
	(instructions)
	end2:

IF *operand1* < *operand* 2 THEN
BEGIN
(instructions)
END
ELSE
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin1: jnl end1
	(instructions)
	end1:jl end2
	(instructions)
	end2:

IF *operand1* $\le$ *operand* 2 THEN
BEGIN
(instructions)
END
ELSE
BEGIN
(instructions)
END
	cmp *operand1*,*operand* 2
	begin1: jg end1
	(instructions)
	end1: jng end2
	(instructions)
	end2:

Example:

PSEUDO- INSTRUCTIONS	ASSEMBLY CODE
N := 7	mov n, 7
M := 5	mov m, 5

IF N = 2 THEN
BEGIN
N = N + 5
END
	begin1: cmp n, 2

	jne end1
	mov eax, n
	add eax, 5
	mov n, eax
	end1:

ELSE
BEGIN
M = N + 5
END
	begin2: je end2

	mov eax, n
	add eax, 5
	mov m , eax
	end2:

Exercise:

1. Assume n is a non-negative integer. We define n factorial as: n! = n(n -1)(n -2)... (2)(1) for n > 0 and 0! = 1.

2. Write an assembly language program that will compute the value 10! .

3. Modify the above problem for an arbitrary n integer program.

Application: Assume we have N distinct objects and r of these objects are randomly selected.

4. The number of ways that this can be done, where order is important is

${}_{N}P_{r}$ = N!/(N - r)!.

Write an assembly language program that will perform the following tasks:

Task1: Assign the integer N and r.

Task2: compute $_{N}P_{r}$ = N!/(N - r)!.

Task3: Output $_{N}P_{r}$.

5. The number of ways that this can be done, where order is not important is

$$\binom{N}{r} = \frac{N!}{r!(N - r)!}$$

Write an assembly language program that will perform the following tasks:

Task1:Assign the integer N and r.

Task2: compute $\binom{N}{r}$.

Task3: Output $\binom{N}{r}$.

■

6. Write an assembly language program that will compute the absolute value of $|x - y|$.

12.5 Top Down Structured Modular Programming

To program using top down structured modular programming, we first begin with a list of tasks that we want to process in the specified order:

Task 1: ------

Task 2: -------

Task3: --------

:::::::::::::::::::

Task n: -------

Next we write pseudo-code for each task in a given module as follows:

Task 1: Module 1
::::::::::::::::::::::::

⇓

Task 2: Module 2
::::::::::::::::::::::::

⇓

Task 3: Module 3
::::::::::::::::::::::::

⇓

::

Task n: Module n
::::::::::::::::::::::::

Finally, we re-write the pseudo-code to assembly language.

Basic Rules:

1. After writing the tasks, first we write the code for Module 1 and check for errors. Once all errors, if any, are corrected, we write module 2 and check for errors. We continue in this manner.

2. We only use to performs branching within the same module. If we need to jump to outside the module, we either branch down to another module or if the program is menu driven we can jump to the module which contains the menu.

Exercise:

Write a structured program that will perform the following tasks:

Task 1: Assign a arbitrary positive integer.

Task 2: Count the number of digits that the integer is made of.

Task 3: Sum the digits of the integer.

PROJECT:

Write an assembly language program that will perform the following tasks:

Task 1: Assign an arbitrary positive integer and a integer 2 through 9.

Task 2: Convert the positive integer into a base 2 through 9.

Task 3: Store the converted integer as a single number

CHAPTER - 13 CONSTRUCTING PROGRAMS IN ASSEMBLY LANGUAGE PART II

Introduction

Now that we can create in assembly language, logical and while statements, we return to the programs and algorithms in chapter 11, to rewrite them in the most general form. Therefore, the following algorithms and programs will be modeled after those in chapter 11.

13.1 An Assembly Language Program to Convert a Positive Integer Number In any Base b < 10 to its Corresponding Number in the Base 10.

Examples:

1. The following method will convert the number 2567_8 to its correspond number in the base 10:

$N_8 = 2567_8 \Leftrightarrow ((2*8+5)*8+6)*8 + 7 = ((21)*8+6)*8 + 7 = 174*8 + 7 = 1399$

To convert the number 2567_8 to the base 10, we first need to write a sample program in pseudo-code and assembly language to capture the digits 2,5, 6,7 from the number. The following programs will perform such a task:

Program: Capture the digits of 2567_8 .

PSEUDO-INSTRUCTIONS	N	A	D
N:= 2567	**2567**		
D:= 1000	2567		**1000**
A:= N÷D	2567	**2**	1000
N:= N MOD D	**567**	2	1000
D:= D÷10	567	2	**100**
A:= N÷D	567	**5**	100
N:= N MOD D	**67**	5	100
D:= D÷10	67	5	**10**
A:= N÷D	67	**6**	10
N:= N MOD D	**7**	6	10
D:= D÷10	7	6	**1**
A:= N÷D	7	**7**	1
N:= N MOD D	**0**	7	**0**

PSEUDO-CODE	CYCLE OF CODES	AL PSEUDO CODE	N	A	EAX	EDX	D	T
N:= 2567	N:= 2567	N:= 2567	**2567**					
T:= 10	T:= 10	T:= 10	2567					**10**
D:= 1000	D:= 1000	D:= 1000	2567				**1000**	10
WHILE N <> 0	WHILE N <> 0	WHILE N <> 0	2567				1000	10
BEGIN	**BEGIN**	**BEGIN**	2567				1000	10
A:= N÷D	A:= N÷D	EAX := N	2567		**2567**		1000	10
		EAX := EAX ÷D EDX:= EAX MOD D	2567		**2**	**567**	1000	10
		A := EAX	2567	**2**	2	567	1000	10
N:= N MOD D	N:= N MOD D	N:= EDX	567	2	2	567	1000	10
D:= D÷T	D:= D÷T	EAX := D	567	2	**1000**	567	1000	10
		EAX:= EAX÷T EDX := EAX MOD T	567	2	**100**	**0**	1000	10
		D := EAX	567	2	100	0	**100**	10
	A:= N÷D	EAX := N	567	2	**567**	0	100	10
		EAX := EAX ÷D EDX:= EAX MOD D	567	2	**5**	**67**	100	10
		A := EAX	567	**5**	5	**67**	100	10
	N:= N MOD D	N:= EDX	**67**	5	5	67	100	10
	D:= D÷10	EAX := D	67	5	**100**	67	100	10
		EAX := EAX ÷T EDX := EAX MOD T	67	5	**10**	**0**	100	10
		D := EAX	67	5	10	0	**10**	10
	A:= N÷D	EAX := N	67	5	**67**	0	10	10
		EAX := EAX ÷D EDX := EAX MOD D	67	5	**6**	**7**	10	10
		A:= EAX	67	**6**	6	7	10	10
	N:= N MOD D	N:= EDX	**7**	6	6	7	10	10
	D:= D÷10	EAX := D	7	6	**10**	7	**10**	**10**

		EAX := EAX ÷T EDX := EAX MOD T	7	6	**1**	**0**	10	10
		D:= EAX	7	6	1	0	**1**	10
	A:= N÷D	EAX := N	7	6	**7**	0	1	10
		EAX := EAX ÷D EDX := EAX MOD D	7	6	**7**	**0**	1	10
		A := EAX	7	**7**	7	0	1	10
	N:= N MOD D	N:= EDX	**0**	7	7	0	1	10
END	**END**	**END**	0	7	7	0	1	10

PSEUDO-CODE	**AL PSEUDO CODE**	**AL CODE**
N:= 2567	N:= 2567	mov n, 2567
T:= 10	T:= 10	mov t, 10
D:= 1000	D:= 1000	mov d, 1000
WHILE N <> 0	WHILE N <> 0	while: cmp n, 0
		je end
BEGIN	**BEGIN**	;begin
A:= N÷D	EAX := N	mov eax, n
	EAX := EAX ÷D EDX := EAX MOD D	mov edx,0
		div d
	A := EAX	mov a, eax
N:= N MOD D	N:= EDX	mov n, edx
D:= D÷T	EAX := D	mov eax, t
	EAX := EAX ÷T EDX := EAX MOD T	mov edx,0
		div t
	D := EAX	mov d,eax
		jmp while
END	**END**	end:

Exercise:

1. Write an assembly language program that will capture the digits of 4578
■

2. **Program:** Writing a sample program to compute

$N_8 = 2567_8 \Rightarrow N_{10} = ((2*8+5)*8+6)*8 + 7 = 1399$

PSEUDO-INSTRUCTIONS	N	A	SUM	D	T
N:= 2567	**2567**				
SUM:= 0	2567		**0**		
T:= 10	T:= 10		0		**10**
D:= 1000	2567		0	**1000**	10
A:= N÷D	2567	**2**	0	1000	10
SUM:= SUM + A	2567	2	**2**	1000	10
SUM:= SUM∗8	2567	2	**16**	1000	10
N:= N MOD D	**567**	2	16	1000	10
D:= D÷T	567	2	16	**100**	10
A:= N÷D	567	**5**	16	100	10
SUM:= SUM + A	567	5	**21**	100	10
SUM:= SUM∗8	567	5	**168**	100	10
N:= N MOD D	**67**	5	168	100	10
D:= D÷T	67	5	168	**10**	10
A:= N÷D	67	**6**	168	10	10
SUM:= SUM + A	67	6	**174**	10	10
SUM:= SUM∗8	67	6	**1392**	10	10
N:= N MOD D	7	6	1392	10	10
D:= D÷T	7	6	1392	**1**	10
A:= N÷D	7	**7**	1392	1	10
SUM:= SUM + A	7	7	**1399**	1	10

PSEUDO-CODE	**CYCLE OF CODES**	**AL PSEUDO CODES**	**N**	**A**	**S**	**EAX**	**EDX**	**D**	**E**	**T**
N := 2567	N:= 2567	N:= 2567	**2567**						**8**	
E:= 8	E:= 8	E:= 8	2567						**8**	
S := 0	S := 0	S := 0	2567		**0**				8	
T:= 10	T:=10	T:= 10	2567		0				8	**10**
D := 1000	D := 1000	D := 1000	2567		0			**1000**	8	10
WHILE D <> 1	WHILE D <> 1	WHILE D <> 1	2567		0			1000	8	10
BEGIN	**BEGIN**	**BEGIN**	2567		0			1000	8	10
A:= N÷D	A:= N÷D	EAX := N	2567		0	**2567**		1000	8	10
		EAX := EAX ÷D	2567		0	**2**	**567**	1000	8	10
		A := EAX	2567	**2**	0	2	567	1000	8	10
S:= S + A	S:= S + A	EAX := S	2567	2	0	**0**	567	1000	8	10
		EAX := EAX + A	2567	2	0	**2**	567	1000	8	10
		S := EAX	2567	2	**2**	2	567	1000	8	10
S:= S∗E	S:= S∗E	EAX := S	2567	2	2	**2**	567	1000	8	10
		EAX := EAX ∗E	2567	2	2	**16**	567	1000	8	10
		S := EAX	2567	2	**16**	16	567	1000	8	10
N:= N MOD D	N:= N MOD D	EAX := N	2567	2	16	**2567**	567	1000	8	10
		EAX := EAX ÷ D	2567	2	16	**2**	**567**	1000	8	10
		N := EDX	**567**	2	16	2	567	1000	8	10
D:= D÷10	D:= D÷10	EAX :=D	567	2	16	**1000**	567	**1000**	8	10
		EAX:= EAX ÷T	567	2	16	**100**	**0**	1000	8	10
		D:= EAX	567	2	16	100	0	**100**	8	10
	A:= N÷D	EAX := N	567	2	16	**567**	0	100	8	10
		EAX := EAX ÷D	567	2	16	**5**	**67**	100	8	10
		A := EAX	2567	**5**	16	5	67	100	8	10
	S:= S + A	EAX := S	2567	5	16	**16**	67	100	8	10
		EAX:= EAX + A	567	5	16	**21**	67	100	8	10
		S:= EAX	567	5	**21**	21	67	100	8	10

	S:= S∗E	EAX := S	567	5	21	**21**	67	100	8	10
		EAX := EAX ∗E	567	5	21	**168**	**0**	100	8	10
		S := EAX	567	5	**168**	168	0	100	8	10
	N:= N MOD D	EAX := N	567	5	168	**567**	0	100	8	10
		EAX := EAX ÷ D	567	5	168	**5**	**67**	100	8	10
		N := EDX	**67**	5	168	**5**	67	100	8	10
	D:= D÷10	EAX :=D	67	5	168	**100**	67	100	8	10
		EAX:= EAX ÷T	67	5	168	**10**	**0**	100	8	10
		D:= EAX	67	5	168	10	0	**10**	8	10
	A:= N÷D	EAX := N	67	5	168	**67**	0	10	8	10
		EAX := EAX ÷D	67	5	168	**6**	**7**	10	8	10
		A := EAX	67	**6**	168	6	7	10	8	10
	S:= S + A	EAX:= S	67	6	168	**168**	7	10	8	10
		EAX:= EAX + A	67	6	168	**174**	7	10	8	10
		S:= EAX	67	6	**174**	174	7	10	8	10
	S:= S∗E	EAX := S	67	6	174	**174**	7	10	8	10
		EAX := EAX ∗E	67	6	174	**1392**	**0**	10	8	10
		S := EAX	67	6	**1392**	1392	0	10	8	10
	N:= N MOD D	EAX := N	67	6	1392	**67**	0	10	8	10
		EAX := EAX ÷ D	7	6	1392	**6**	**7**	10	8	10
		N := EDX	**7**	6	1392	6	7	10	8	10
	D:= D÷10	EAX :=D	7	6	1392	**10**	7	10	8	10
		EAX:= EAX ÷T	7	6	1392	**1**	**0**	10	8	10
		D:= EAX	7	6	1392	1	0	**1**	8	10
END	**END**	**END**	7	6	1392	1	0	1	8	10
S:= S + A	S:= S + A	EAX := S	7	**7**	1392	**1392**	0	1	8	10
		EAX:= EAX + A	7	7	1392	**1399**	0	1	8	10
		S:= EAX	7	7	**1399**	1399	0	1	8	10

PSEUDO-CODE	AL PSEUDO - CODES	AL CODE
N := 2567	N:= 2567	mov n, 2567
E:=8	E:= 8	mov e, 8
S := 0	S := 0	mov sum, 0
T:= 10	T:= 10	mov t,10
D := 1000	D := 1000	mov d, 1000
WHILE D <> 1	WHILE D <> 1	while: cmp d,1 je end
BEGIN	**BEGIN**	**;begin**
A:= N÷D	EAX := N	mov eax,n
	EAX := EAX ÷D	mov edx,0
		div d
	A := EAX	mov a,eax
S:= S + A	EAX := S	mov eax,s
	EAX := EAX + A	add eax, a
	S := EAX	mov s,eax
S:= S∗E	EAX := S	mov eax, s
	EAX := EAX ∗E	mul e
	S := EAX	mov s, eax
N:= N MOD D	EAX := N	mov eax, n
	EAX := N ÷ D	mov edx,0 div d
	N := EDX	mov n,edx
D:= D÷10	EAX :=D	mov eax,d
	EAX:= EAX ÷T	mov edx,0
		div t
	D:= EAX	mov d,eax
END	**END**	jmp while
S:= S + A	EAX := S	end: mov eax, s
	EAX:= EAX + A	add eax, a

	S:= EAX	mov s, eax

Exercise:

1. Write a assembly language program that will convert $N_4 = 2312_4 \Rightarrow N$ $N_8 = 2567_8 \Rightarrow N_{10}$
■

13.2 An Algorithm to Convert any Integer Number in the Base 10 to a Corresponding Number in the Base b < 10.

Using the Euclidean division theorem, we now review how, using the manual method to convert numbers in the base 10 to any in the base b.

Step 1: We want to write N in the form: $N = a_n b^n + a_{n-1} b^{n-1} \ldots + a_1 b + a_0$

Step 2: $N = Qb + R = (a_n b^{n-1} + a_{n-1} b^{n-2} \ldots + a_1)b + a_0$

Here, $Q = a_n b^{n-1} + a_{n-1} b^{n-2} \ldots + a_2 b + a_1 = (a_n b^{n-2} + a_{n-1} b^{n-3} \ldots + a_2)b + a_1$ and $R = a_0$

Step 3: Set N = Q.

$Q = Q_1 b + R_1 = (a_n b^{n-2} + a_{n-1} b^{n-3} \ldots + a_2)b + a_1$ where

$Q_1 = a_n b^{n-2} + a_{n-1} b^{n-3} \ldots + a_2$,

$R_1 = a_1$.

Step 4: Continue in this manner, until $Q_n = 0$.

Example:

Convert the following decimal numbers to the specified base.

1. $1625 \Rightarrow N_8$

Step 1: $1625 = (1625 \div 8) * 8 + 1 = 203 * 8 + 1$

$a_0 = 1$

Step 2: $203 = (203 \div 8) * 8 + 3 = 25 * 8 + 3$

$a_1 = 3$

Step 3: $25 = (25 \div 8) * 8 + 1 = 3 * 8 + 1$

$a_2 = 1$

Step 4: $3 = (3 \div 8) * 8 + 3 = 3$

$a_3 = 3$

Therefore, $1625 \Rightarrow N_8 = 3 * 8^3 + 1 * 8^2 + 3 * 8 + 1 \Leftrightarrow N_8 = 3131_8$

Program: Pseudo-Code to convert the integer number 1625 to the base 8.

PSEUDO-CODE	N	SUM	TEN	MUL	BASE	R
BASE := 8					**8**	
N := 1625	**1625**				8	
SUM := 0	1625	**0**			8	
MUL := 1	1625	0		**1**	8	
TEN := 10	1625	0	**10**	1	8	
R := N MOD BASE	1625	0	10	1	8	**1**
N:= N÷BASE	**203**	0	10	1	8	1
R := R∗MUL	203	0	10	1	8	1
SUM:= SUM + R	203	**1**	10	1	8	1
MUL:= MUL∗TEN	203	1	10	**10**	8	1
R := N MOD BASE	203	1	10	10	8	**3**
N:= N÷BASE	**25**	1	10	10	8	3
R := R∗MUL	25	1	10	10	8	**30**
SUM:= SUM + R	25	**31**	10	10	8	30
MUL:= MUL∗TEN	25	31	10	**100**	8	30
R := N MOD BASE	25	31	10	100	8	**1**
N:= N÷BASE	**3**	31	10	100	8	1
R := R∗MUL	3	31	10	100	8	**100**
SUM:= SUM + R	3	**131**	10	100	8	100
MUL:= MUL∗TEN	3	131	10	**1000**	8	100
R := N MOD BASE	3	131	10	1000	8	**3**
N:= N÷BASE	0	131	10	1000	8	3
R := R∗MUL	0	131	10	1000	8	**3000**
SUM:= SUM + R	0	**3131**	10	1000	8	3000

PSEUDO-CODE	CYCLE OF CODES	AL PSEUDO-CODE	N	S	M	R	EAX	EDX	B	T
B := 8	B := 8	B := 8							**8**	
N := 1625	N := 1625	N := 1625	**162**						8	
S:= 0	S:= 0	S:= 0	162	**0**					8	
M:= 1	M:= 1	M:= 1	162	0	**1**				8	
T:= 10	T:= 10	T:= 10	162	0	1				8	**10**
WHILE N <>	WHILE N <> 0	WHILE N <> 0	162	0	1				8	10
BEGIN	**BEGIN**	**BEGIN**	162	0	1				8	10
R := N MOD B	R := N MOD B	EAX:= N	162	0	1		**1625**		8	10
		EAX:= EAX÷B	162	0	1		**203**		8	10
		EDX:= EAX MOD B	162	0	1		203	1	8	10
		R:= EDX	162	0	1	**1**	203	1	8	10
N:= N÷B	N:= N÷B	N:= EAX	**203**	0	1	1	203	1	8	10
R := R∗M	R := R∗M	EAX:= R	203	0	1	1	**1**	1	8	10
		EAX:= EAX∗M	203	0	1	1	1	**0**	8	10
		R:= EAX	203	0	1	**1**	1	0	8	10
S:= S + R	S:= S + R	EAX:= S	203	0	1	1	0	0	8	10
		EAX:= EAX + R	203	0	1	1	1	0	8	10
		S:= EAX	203	**1**	1	1	1	0	8	10
M:= M∗T	M:= M∗T	EAX:= M	203	1	1	1	1	0	8	10
		EAX:= EAX∗T	203	1	1	1	**10**	**0**	8	10
		M:= EAX	203	1	**10**	1	10	0	8	10
	R := N MOD B	EAX:= N	203	1	1	1	**203**	0	8	10
		EAX:= EAX:÷B	203	1	1	1	**25**	**3**	8	10
		R:= EDX	203	1	1	**3**	25	3	8	10
	N:= N÷B	N:= EAX	**25**	1	10	3	25	3	8	10
	R := R∗M	EAX:= R	25	1	10	3	**3**	3	8	10
		EAX:= EAX∗M	25	1	10	3	**30**	**0**	8	10

		R:= EAX	25	1	10	**30**	30	0	8	10
	S:= S + R	EAX:= S	25	1	10	30	**1**	0	8	10
		EAX:= EAX + R	25	1	10	30	**31**	0	8	10
		S:= EAX	25	**31**	10	30	31	0	8	10
	M:= M∗T	EAX:= M	25	31	10	**1**	**10**	0	8	10
		EAX:= EAX∗T	25	31	10	1	**100**	**0**	8	10
		M:= EAX	25	31	**100**	1	100	0	8	10
	R := N MOD B	EAX:= N	25	31	100	1	**25**	0	8	10
		EAX:= EAX÷B	25	31	100	1	**3**	0	8	10
		EDX:= EAX MOD B	25	31	100	1	3	**1**	8	10
		R:= EDX	25	31	100	**1**	3	1	8	10
	N:= N÷B	N:= EAX	**3**	31	100	1	3	1	8	10
	R := R∗M	EAX:= R	3	31	100	1	**1**	1	8	10
		EAX:= EAX∗M	3	31	100	1	**100**	0	8	10
		R:= EAX	3	31	10	**100**	100	0	8	10
	S:= S + R	EAX:= S	3	31	100	100	**31**	0	8	10
		EAX:= EAX + R	3	31	100	100	**131**	0	8	10
		S:= EAX	3	**131**	100	100	131	0	8	10
	M:= M∗T	EAX:= M	3	131	100	100	**100**	0	8	10
		EAX:= EAX∗T	3	131	100	**1**	**1000**	0	8	10
		M:= EAX	3	131	**1000**	1	1000	0	8	10
	R := N MOD B	EAX:= N	3	131	1000	1	**3**	0	8	10
		EAX:= EAX÷B	3	131	1000	1	**0**	0	8	10
		EDX:= EAX MOD B	3	131	1000	1	**0**	3	8	10
		R:= EDX	3	131	1000	**3**	**0**	3	8	10
	N:= N÷B	N:= EAX	**0**	131	1000	3	0	3	8	10
	R := R∗M	EAX:= R	0	131	1000	3	**3**	3	8	10
		EAX:= EAX∗M	0	131	1000	3	**3000**	**0**	8	10
		R:= EAX	0	131	1000	**3000**	3000	0	8	10
	S:= S + R	EAX:= S	0	131	1000	3000	**131**	0	8	10

		EAX:= EAX + R	0	131	1000	3000	**3131**	0	8	10
		S:= EAX	0	**3131**	1000	3000	3131	0	8	10
	M:= M∗T	EAX:= M	0	3131	1000	3000	**1000**	0	8	10
		EAX:= EAX∗T	0	3131	1000	3000	**10000**	**0**	8	10
		M:= EAX	0	3131	**10000**	3000	10000	0	8	10
END	**END**	**END**	0	3131	10000	3000	10000	0	8	10

$1625 \Rightarrow 3131_8$

PSEUDO-CODE	AL PSEUDO-CODE	AL CODE
B := 8	B := 8	mov b, 8
N := 1625	N := 1625	mov n, 1625
S:= 0	S:= 0	mov s, 0
M:= 1	M:= 1	mov m, 1
T:= 10	T:= 10	mov t, 10
WHILE N <> 0	WHILE N <> 0	while: cmp n, 0
BEGIN	**BEGIN**	begin: je end
R := N MOD B	EAX:= N	mov eax, n
	EAX:= EAX÷B EDX:= EAX MOD B	mov edx,0
		div b
	R:= EDX	mov r, edx
N:= N÷B	N:= EAX	mov n, eax
R := R∗M	EAX:= R	mov eax, r
	EAX:= EAX∗M	mul m
	R:= EAX	mov r, eax
S:= S + R	EAX:= S	mov eax, s
	EAX:= EAX + R	add eax, r
	S:= EAX	mov s, eax
M:= M∗T	EAX:= M	mov eax, m
	EAX:= EAX∗T	mul t

	M:= EAX	mov m , eax
		jmp while
END	**END**	end:

$1625 \Rightarrow 3131_8$

Exercise:

1. Write a assembly language program that will convert 2567 $\Rightarrow N_5$

Note: See model program.

```
; This program converts 1625 ⇒ 3131₈
.386

.MODEL FLAT

.STACK 4096

.DATA
n dword ?
s dword ?
m dword ?
r dword ?
b dword ?
t dword ?

.CODE

_start:


;start assembly language code

mov b, 8

mov n, 1625

mov s, 0

mov m, 1

mov t, 10

while: cmp n, 0

begin: je end
```

```
mov eax, n
mov edx,0
div b
mov r, edx
mov n, eax
mov eax, r
mul m
mov r, eax
mov eax, s
add eax, r
mov s, eax
mov eax, m
mul t
mov m , eax

jmp while
end:
;end of assembly language code
PUBLIC_start
END
```

PROJECTS

1. An integer number N is said to be prime if it is only dividable evenly by 1 or N.

Write an assembly Language program to determine if 2,346,799 is prime.

2. *The Fibonacci Numbers*

The Fibonacci numbers is a sequence of integer numbers generated as follows:

Step 1: Start with 0,1

Step 2: The next number is generated by adding the last 2 numbers: 0,1,1 .

Step 3: To generate the next number, continue by adding the last 2 numbers:

0,1,1,2,3,5, 8, 13, 21, ...

Write an assembly language program that will generate a sequence N Fibonacci numbers.

CHAPTER- 14 LOGICAL EXPRESSIONS, MASKS, AND SHIFTING

14.1: Logical Expressions

Logical expressions and values are similar to conditional expressions as defined in Chapters 5 and 6. However, due to the nature of the applications, we will use a different terminology in this chapter.

Definition of logical values: Logical values are of two types: *true*, *false.*

Definition of logical identifiers: Logical identifiers are identifiers (variables) that are assigned only values *true, false.*

Definition of logical operators: There are three binary logical operators and one unary logical operator:

The binary logical operators are .AND., .OR., .XOR. .

The unary logical operator is .NOT. .

Definition of Logical Expressions: A logical expression is made up of logical values, logical identifiers connected by logical operators.

The following table gives the logical values which result from the four logical operators:

OPERATORS	RESULTING VALUE
.OR.	*true* .OR. *true* = *true* *true* .OR. *false* = *true* *false* .OR. *true* = *true* *false* .OR. *false* = *false*
.AND.	*true* .AND. *true* = *true* *true* .AND. *false* = *false* *false.* AND. *true* = *false* *false* .AND. *false* = *false*
.XOR.	*true* .XOR. *true* = *false* *true* .XOR. *false* =*true* *false* .XOR. *true* = *true* *false* .XOR. *false* = *false*
.NOT.	.NOT. *true* = *false* .NOT. *false* = *true*

Examples:

a. *logical value:*

5 = 2 + 3

takes on the value *true.*

b. *logical identifiers:* X

where

X := (5 = 1 - 4)

X takes on the value *false.*

c. *true* .AND. (X = *false*)

takes on the value *false.*

d. Y:= 5

VALUE := *true*

(.NOT. (VALUE = *true*)) .OR. (Y < 3)

The above expression takes on the value *false* .OR. *false* = *false.*

e. Z := 0

Y = *true*

NOT. ((Z < 2) .XOR. (Y = *false*))

takes on the value *false.*

Relational Operators
The following six relational operators connect the logical values and identifiers:

Definition of Six Relational Operators

The six relational operators are:

	Operator	**Interpretation**
1.	=	Equality
2.	<>	Inequality
3.	<	Less Than
4.	>	Greater Than
5.	<=	Less than or equal to
6.	>=	Greater than or equal to

Examples: **Values:**

5 = 2 + 3 *true*
9 <> 3∗3 *false*
4 <= 4 *true*
-17 < -7 *true*
(7 = 2 + 3) .OR. (4 < 1) *false*

LOGICAL EXPRESSIONS	VALUES
(5 = 2 - 4) .OR. (2 <> 3)	*false* .OR. *true = true*
(5 = 2 - 4) .AND. (2 <> 3)	*false* .AND. *true = false*
(5 = 2 - 4) .XOR. (2 <> 3)	*false* .XOR. *true = true*
.NOT. (5 = 2 - 4)	.NOT. (5 = 2 - 4) = *true*

Logical Statements

Definition of logical statements: A logical statement is a an instruction where the variables are declared to be logical identifiers and these variables can be assigned logical values resulting from logical expressions.

Example:

PSEUDO - CODE	X	Y	L	Z
X := 4	**4**			
Y := 6	4	**6**		
L:= (X + Y = 10)	4	6	***true***	
Z := L .XOR. (X - Y <> 0)	10	6	*true*	***false***
Z := Z .AND. L	10	6	*true*	***false***

Exercise:

1. Complete the following:

PSEUDO - CODE	X	Y	L	Z
X := 2				
Y := 5				
L:= (X + 2∗Y > 2)				
Z := .NOT. (L .OR. (.NOT. (X - Y <> 0)))				
Z := (.NOT.(L .AND. (Z .OR. L)) .XOR. Z				

■

Example:

The following program demonstrates how these logical expressions can be used in a program.

Task1: Assign three integer numbers

Task2: If the sum of these numbers is greater than 10 but less than 20 divide the sum by 2 ; otherwise compute the average of these numbers.

For the following program, assume the numbers 3,4,9 are assigned.

PSEUDO - CODE	**X**	**Y**	**Z**	**S**	**L**
X:= 3	**3**				
Y:= 4	3	**4**			
Z:= 9	3	4	**9**		
S := X + Y + Z	3	4	9	**16**	
L: = (S > 10) .AND. (S < 20)	3	4	9	16	***true***
IF L = *true* THEN	3	4	9	16	*true*
BEGIN	3	4	9	16	*true*
S:= S÷2	3	4	9	**8**	*true*
END	3	4	9	8	*true*
ELSE	3	4	9	8	*true*
BEGIN	3	4	9	8	*true*
S:= S÷3	3	4	9	8	*true*
END	3	4	9	8	*true*

Exercises:

1. In the following program, indicate if the following statements are correct or incorrect.

X: = 2

Z := *true*

V := .NOT. (*true* .OR. *false*)

V:= (.NOT.(V .OR. V)) .AND. V

2. Evaluate the following expressions:

a. (.NOT.(*true* .XOR *true*)) .AND. (.NOT.(*false* .OR. *true*))

b. (. NOT.(*true* .XOR *false*)) .OR. (.NOT.(*true* .OR. *false*))

c. .NOT. ((NOT.(*true* .XOR. *false*)) .AND. ((*true* .OR. *false*)))

3. Evaluate the following expressions:

a. (.NOT. (*true* .AND. *true*) = *false*) .OR. *false*

b. (.NOT. (*false* .AND. *true*) = *true*).XOR. *false*

c. (.NOT. (*false* .AND. *false*) = *true*) .OR. *true*

d. (.NOT. (*true* .OR. *true*) = *false*).AND. *false*

e (.NOT. (*false* .OR. *true*) = *true*) .AND. *false*

f. (.NOT. (*false* .OR *false*) = *true*) .AND. *true*

4. Is the following statement true or false: (.NOT. (*false* .XOR. *true*) = *true*) .AND. *false* ?

■

14.2: Logical Expressions In Assembly Language.

In assembly language the value *true* is associated with the integer number 1 and the value *false* is associated with the integer number 0. The four logical operations in assembly are given by the following table:

PSEUDO-LANGUAGE LOGICAL OPERATORS	**ASSEMBLY LANGUAGE LOGICAL OPERATORS**
.AND.	and
.OR.	or
.XOR.	xor
.NOT.	not

The following table gives the logical values in the assembly language which result from the above four logical operators:

ASSEMBLY LANGUAGE LOGICAL OPERATORS	RESULTING VALUE
and	1 and 1 = 1
	1 and 0 = 0
	0 and 1 = 0
	0 and 0 = 0
or	1 or 1 = 1
	1 or 0 = 1
	0 or 1 = 1
	0 or 0 = 0
xor	1 xor 1 = 0
	1 xor 0 = 1
	0 xor 1 = 1
	0 xor 0 = 0
not	not 1 = 0 not 0 = 1

The format of the assembly language logical operators

The following are the formats of the four assembly language logical operators:

and *destination, source*

or *destination, source*

xor *destination, source*

not *destination*

where *destination* is a register where the logical value is assigned and *source* is a logical identifier, logical value (0 or 1), or register containing a logical value. If the source is a identifier (variable), the register and the identifier must be of the same data type.

Important: The *not* logical instruction will change, in the register, the o bits to the 1 bits and the 1 bits to the 0 bits.

Examples:

The and operator

ASSEMBLY LANGUAGE	Al
mov al, 1	**00 00 00 01**
and al,1	00 00 00 01
and al, 0	00 00 00 00
mov al,0	00 00 00 00
and al,0	00 00 00 00

The or operator

ASSEMBLY LANGUAGE	AL
mov al,1	00 00 00 01
or al,1	00 00 00 01
or al,0	00 00 00 01
mov al,0	00 00 00 00
or al,0	00 00 00 00

The xor operator

ASSEMBLY LANGUAGE	AL
mov al,1	00 00 00 01
xor al ,1	00 00 00 00
xor al,0	00 00 00 00
xor al,1	00 00 00 01

The not operator

ASSEMBLY LANGUAGE	AL
mov al,1	00 00 00 01
not al	11 11 11 10
not al	00 00 00 01
mov al,0	00 00 00 00
not al	11 11 11 11
not al	00 00 00 00

Exercise:

1. Change the following pseudo-code program to a partial assembly program .

EAX := *true*
EBX := *false*
EAX := (.NOT. (EAX .AND EBX)) .XOR. (EAX)

PSEUDO-LANGUAGE	ASSEMBLY LANGUAGE	EAX	EBX
EAX := *true*			
EBX := *false*			
EAX := (.NOT. (EAX .AND EBX)) .XOR. (EAX)			

■

14.3: Assigning to Logical Expressions a Logical Value in Assembly Language.

When programming in assembly language, we can not use logical statements directly. To perform logical statements, we need to use the compare and jump statements described in Chapter 12. This is done by assigning values 1 or 0 so that the compare and the appropriate jump statements can properly evaluate and carry out the logical statements desired. The following examples show how this is done.

Example:

We wish to write an assembly language program that will perform the following tasks:

Task1: Assign two numbers into x, y.

Task2: If both numbers are greater than 10, compute the sum of the two numbers.

Task3: If at least one of the numbers is less than or equal to 10, compute the product of the two numbers.

PSEUDO-CODE	X	Y	Z	LOG
X:= 5	**5**			
Y:= 60	5	**60**		
LOG:= (X > 10) .AND. (Y > 10)	5	60		***false***
IF LOG = *true* THEN	5	60		*false*
BEGIN	5	60		*false*
Z:= X + Y	5	60		*false*
END	5	60		*false*
ELSE	5	60		*false*
BEGIN	5	60		*false*
Z:= X $*$ Y	5	60	**300**	*false*
END	5	60	300	*false*

PSEUDO-CODE	AL	X	Y	Z	LOG	EAX	EBX
X:= 5	mov x, 5	**5**					
Y:= 60	mov y, 60	5	**60**				
LOG := (X > 10) .AND. (Y > 10)	mov eax, 0	5	60			**0**	
	mov ebx, 0	5	60			0	**0**
	cmp x, 10	5	60			0	0
	jng L1	5	60			0	0
	mov eax, 1	5	60			0	0
	L1: cmp y,10	5	60			0	0
	jng L2	5	60			0	0
	mov ebx, 1	5	60			0	1
	L2: and eax, ebx	5	60			**0**	1
	mov log, eax	5	60		**0**	0	1

IF LOG = *true* THEN	cmp log, 1	5	60		0	0	1
BEGIN	begin1: jne end1	5	60		0	0	1
Z:= X + Y	mov eax, x	5	60		0	0	1
	add eax, y	5	60		0	0	1
	mov z, eax	5	60		0	0	1
END	end1:	5	60		0	0	1
ELSE	je end2	5	60		0	0	1
BEGIN	begin2:	5	60		0	0	1
Z:= X∗Y	mov eax, x	5	60		5	0	1
	mul y	5	60		5	**300**	1
	mov z, eax	5	60	**300**	5	300	1
END	end2:	5	60	300	0	300	1

Exercises:

1. For the above program, assume x = 20 and y = 30. With these values, change the above table.

2. For the above program, assume x = 2 and y = 3. With these values, change the above table.

3. Write an assembly language program that will perform the following tasks:

Task1: Assign two positive integer numbers x, y.

Task2: If $x > 10$ and $y > 10$ than compute $x + y$.

Task3: If $x > 10$ and $y \le 10$ than compute $x * y$.

Task4: If $x \le 10$ and $y > 10$ than compute $2 * (x + y)$.

Task5: If $x \le 10$ and $y \le 10$ than compute $3 * (x + y)$.
■

14.4: Masks

Definition of a mask: A mask is a binary integer number (BYTE, WORD, DWORD) used with a selected logical operator (and, or, xor) that will be matched bit-by-bit with binary number contained in a selected register.

The mask instruction

Definition of the mask instruction:

logical operator   destination, source

where the destination and source is defined above. If the source is an identifier, the destination and the source must be of the same data type.

For this matching the following resulting values will hold:

ASSEMBLY LANGUAGE LOGICAL OPERATORS	RESULTING VALUE
and	1 and 1 = 1 1 and 0 = 0 0 and 1 = 0 0 and 0 = 0
or	1 or 1 = 1 1 or 0 = 1 0 or 0 = 0
xor	1 xor 1 = 0 1 xor 0 = 1 0 xor 1 = 1 0 xor 0 = 0

Examples:

Assume AX and BX contains the following binary numbers:
AX: 0110 1110 1100 0011

BX: 1001 1100 0101 1011

<u>Here BX will be the mask.</u>

We will now show, by the following examples, how the mask works, resulting in changing of bits in AX:

and ax, bx;   AX: 0110 1110 1100 0011

BX: 1001 1100 0101 1011

⇓

AX: 0000 1100 0100 0011

::::::::::::::::::::::::::

or ax, bx; AX: 0110 1110 1100 0011

BX: 1001 1100 0101 1011

⇓

AX: 1111 1100 1101 1011

:::::::::::::::::::::::::

xor ax, bx; AX: 0110 1110 1100 0011

BX: 1001 1100 0101 1011

⇓

AX: 1111 0010 1001 1000

Exercises:

Assume CX contains an arbitrary number. For the following assembly instructions, explain what changes to CX, if any, result from the following masks:

1. and cx, cx

2. or cx, cx

3. xor cx, cx

4. and cx, (not cx)

5. or cx, (not cx)

6. xor cx, (not cx)

■

14.5: Shifting Instructions

There are two types of shifting instructions: the shift instructions and the rotation instructions.

The shift instructions

The shift instructions move the bits in a register to the left or to the right by a designated number. The following are the shift instructions:

shl *register*, n; will shift the bits in the register to the left by n places. The extreme left bits will fall out of the register. Added bits will be the bit 0. The added bit(s) will be in bold.

shr *register*, n; will shift the bits in the register to the right by n places. The extreme right bits will fall out of the register but the left added bits will be the bit 0 . The added bit(s) will be in bold.

Examples:

For the following examples assume the register AX contains 1011 0100 1110 1011 .

shl ax, 1 ; 1011 0100 1110 1011
⇐
0110 1001 1101 011**0**
:::::::::::::

shl ax, 4 1011 0100 1110 1011
⇐
0100 1110 1011 **0000**
::::::::::::

shr ax, 1 ; 1011 0100 1110 1011
⇒
0101 1010 0111 0101
:::::::::::

shr ax, 4 1011 0100 1110 1011
⇒
0000 1011 0100 1110

Multiplication and division applications.

One important application of the left shift results in multiplying the original number by a power of 2.

Examples:

1. Assume AX contains 0000 0000 0000 0011 which is equal to the number 3d.

shl ax, 1 will result in AX changed to 000 0000 0000 00110 which is equal to the number 6d.

2. Assume AX contains 0000 0000 0000 0011 which is equal to the number 3d.

shl ax, 2 will result in AX changed to 0000 0000 0000 1100 which is equal to the number 12d.

One important application of the right shift results in dividing the original number by a power of 2.

3. Assume AX contains 0000 0000 0000 0110 which is equal to the number 6d.

shr ax, 1 will result in AX changed to 0000 0000 0000 0011 which is equal to the number 3d.

The rotation instructions

There are two types rotation instructions:

rol *destination,* n; rotate the bits to the left n places. The bits that are shifted off the left hand side replace the bits that are added on the right hand side.

ror *destination,* n; rotate the bits to the right n places. The bits that are shifted off the right hand side replace the bits that are added on the left hand side.

Examples:

1: Assume AX contains 1100 0000 0000 0101.

rol ax, 2 will result in AX changed to 0000 0000 0001 0111

2: Assume AX contains 1100 0000 0000 0101

ror ax, 3 will result in AX changed to 1011 1000 0000 0000

```
;This is the above program.
.386

.model flat

.stack 4096

.data
n dword ?
s dword ?
m dword ?
r dword ?
b dword ?
t dword ?

.code

_start:


;start assembly language code
mov x, 5
mov y, 60
mov eax, 0
mov ebx, 0
cmp x, 10
jng L1
mov eax, 1
L1: cmp y,10
jng L2
mov ebx, 1
L2: and eax, ebx
mov log, eax
cmp log, 1
begin1: jne end1
```

```
 mov eax, x
add eax, y
mov z, eax
end1:
je end2
begin2:
mov eax, x
mul y
mov z, eax
end2:

;end of assembly language code

public _start

End
```

Project

Two positive different integer numbers are said to be relatively prime if both numbers have no common divisors other than the number 1.

Examples:

The numbers 51, 32 are relatively prime since they have no common divisors.

The numbers 22, 40 are not relatively prime since 2 divides both numbers.

Write an assembly language program that will perform the following tasks:

Task1: Enter a positive number N > 1.

Task2: Find the number of relatively prime numbers ≤ N

CHAPTER 15 - INTEGER ARRAYS

INTRODUCTION

So far we have seen that we can save integer numeric values in variables such as x, y, z, etc. Restricting ourselves to only variables of this type do not allow us to effectively store large amount of data. To accomplish this we need to define arrays (tables). We first introduce one dimensional arrays in pseudo-code.

15.1 Representing One-Dimensional Arrays in Pseudo-Code.

Definition of a one-dimensional arrays

A one dimensional array is a collection of cells all of which have the same name, but are distinguished from one another by the use of subscripts. A subscript is a positive integer number in parentheses which follows the array's name.

Examples:

1. a(1), a(2), a(3), ..., a(99), a(100)

2. num(1), num(2), ..., num(999), num(1000)

In the first example, the array named a can store 100 pieces of data and the in the second example, the array named num can store 1,000 pieces of data.

Rules for arrays

1. The array name is a valid identifier

2. Each subscript must be a positive integer

3. Integer numeric values can be stored in these array cells.

Examples:

a(10) := 3

num(100) := -7

sum := a(10) + num(100)

Programming examples:

The following program, in pseudo-code, will perform the following tasks:

Task1: Stores the numbers 2, 4, 6, ..., 1000 in array cells.

Task2: Add the numbers in the cells.

Task3: Compute the average

Task4: Store all the numbers that are greater than the average.

```
TASK1:
k : = 1
j := 0
WHILE j ≤ 1000
BEGIN
j := 2∗k
num(k) = j
k := k+1
END

TASK2:
total:= 0
k := 0
WHILE k ≤ 500
k := k + 1
total := total + num(k)
END

TASK3:
average : = total/500

TASK4:
k := 0
WHILE k ≤ 500
k := k + 1
IF num(k) > average THEN
Store(k) :=  num(k)
END
```

Exercises:

1. Write a pseudo- code program that will perform the following tasks:

Task1: Stores the numbers 2, 2^2, 2^3, ..., 2^n in array cells.

Task2: Add the numbers in the cells.

Task3: Compute the integer average. (The average without the remainder.)

2. Finding the largest value.

Write a pseudo- code algorithm that will perform the following tasks:

Task1: Store n non-negative integers into an array.

Task2: Find the largest value.

3. Converting positive decimal integers into binary.

Write a pseudo- code algorithm that will perform the following tasks:

Task1: Store a non-negative integer number.

Task2: Convert this number into binary and store the binary digits into an array.

4. Writing numbers backward.

Task1: Store a positive integer number.

Task2: Store the digits into an array backward.

5. A proper divisor of a positive integer N is an integer that is not equal to 1 or N and divides N without a remainder.

For example the proper divisors of 210 are 2, 3,5,7 .

Write a program that perform the following tasks:

Task1: Store a positive integer number N.

Task2: Find and store in array all the proper divisors of N.

6. The Fibonacci number sequence

The Fibonacci numbers are the following:

0,1,1,2,3,5,8,13,...

where 0 + 1 = 1, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, etc.

The general rule is to add the last two numbers in the sequence to get the next number.

Write a pseudo-code program that will perform the following tasks:

Task1: Store a positive integer N.

Task2: Compute and store in an array, all Fibonacci numbers less than or equal to N.
■

15. 2 Creating One Dimensional Integer Arrays In Assembly Language.

Ther are several ways to create a one dimensional integer array. We begin by starting an array at the location of a given variable. We define an array using the directive instructive in the data portion of the program. We will use the directive

***variable name* byte ?**

to establish the location in memory of the cell a(1).

Since the assembler will determine the beginning location of the first cell of the array, we can capture the location with the *lea* instruction. The following is the definition of the *lea* instruction in the instruction portion of the program:

The lea *32 bit register, variable name of the array instruction.*

Definition of the lea instruction:

The lea instruction will store into any 32 bit register, the first byte location of a variable.

Example:

x byte ?
lea ebx, x

In this example the lea instruction will store into ebx, the first byte location of the variable x.

Before we discuss arrays in assembly language, we need to better understand how data is stored in main memory. All integer data are represented as bytes, words or dwords. All of these are made up of bytes: the double word (DWORD) is made up of 4 bytes (32 bits); the word (WORD) is made up of 2 bytes and the byte (BYTE) is made up of 1 byte. We can think of the main memory as large memory table made up of columns and rows and where each cell of the table is a byte, each identified with a numeric location :

1		2		3		4	
5		6		7		8	
9		10		11		12	
13		14		15		16	
:::::::::::::::	:::::::::::::::	:::::::::::::::	:::::::::::::::	:::::::::::::::	:::::::::::::::	:::::::::::::::	:::::::::::::::

For example, assume the identifiers x, y are defined as double words and assigned the values 3h and 5875h respectively:

x dword 3h

y dword 5875h

Assume the assembler selects in memory cell locations 1-4 for x and 13-16 for y. Our memory table would look something like:

1 0	0	2 0	0	3 0	0	4 0	3
::::	::::	::::	::::	::::	::::	::::	::::
13 0	0	14 0	0	15 5	8	16 7	5
::::	::::	::::	::::	::::	::::	::::	::::

Creating a one dimensional array of a given data type.

When we create an array, we can store the array elements as three types of data: bye, word, dword.

The following steps will define and set up the array.

Step 1: Define the variable name and its data type byte.

Step 2: Using the lea instruction, store the first byte location in a 32 bit register.

Examples:

1.
x byte ?
lea ebx, x

2. y word ?
lea eax, y

3. z dword ?
lea edx, z

Storing data in the array using a variable's location.

The following definition is the assignment statement that will allow us to perform data assignments to and from memory cells:

mov [register], source instruction.

where

the **register** must be a 32 bit register and the **source** can be a register of the same data type as the variable. .

Definition: *mov [register], source*

The mov [*register*], *source* instruction will store the number in the source register directly into the memory location indicated by the contents of the register,

where the following rules apply:

Rule 1: The lea instruction will establish the first byte location.

Rule 2: The register must be EAX, EBX, ECX, or EDX.

Rule3: The *source* can be a register of the same data type as the variable.

Rule4: The *[register]* indicates the cell location where the bytes are to be located.

The [*register]* is call the indirect register.

For all examples in this chapter, we assume all numbers are represented as hexadecimals.

Examples:

The following examples show how arrays of different data types are created and data is stored.

1.

AL CODE	**AL**	**X**
x byte 68h		**68**
lea ebx, x		68
mov al, 9Ah	**9A**	68
mov [ebx], al	9A	**9A**

2.

AL CODE	AX	X
x word 35A8h		**35A8**
lea ebx, x		35A8
mov ax, 237Ah	**237A**	35A8
mov [ebx], ax	237A	**237A**

3.

AL CODE	EAX	X
x dword 17223FDh		**17223FD**
lea ebx, x		17223FD
mov eax, 0A637Ah	**A637A**	17223FD
mov [ebx], eax	A637A	**A637A**

4. The following program will store numbers 13h, 29h,25h into the array X of type BYTE.

PSEUDO CODE	AL CODE	AL	X		
Array X	x byte ? lea ebx,x		**byte 1**	**byte 2**	**byte 3**
X(1) := 13h	mov al, 13h	**13**			
	mov [ebx], al	13	**13**		
	add ebx,1	13	13		
X(2):= 29h	mov al,29h	**29**	13		
	mov [ebx],al	29	13	**29**	
	add ebx,1	29	13	29	
X(3):= 25h	mov al,25h	**25**	13	29	
	mov [ebx],al	25	13	29	**25**

Important: Since we are storing into individual byes, <u>we increment by 1.</u>

5. The following program will store numbers 13h, 29h,25h into the array of type WORD.

PSEUDO CODE	AL CODE	AX	X		
Array X	x word ? lea ebx,x		**word 1**	**word 2**	**word 3**
X(1) := 13h	mov ax, 13h	**0013**			
	mov [ebx], ax	0013	**00 13**		
	add ebx,2	0013	00 13		
X(2):= 29h	mov ax,29h	**0029**	00 13		
	mov [ebx],ax	0029	00 13	**00 29**	
	add ebx,2	0029	00 13	00 29	
X(3):= 25h	mov ax,25h	**0025**	00 13	00 29	
	mov [ebx],ax	0025	00 13	00 29	**00 25**

Important: Since we are storing into individual byes for each word, <u>we increment by by 2.</u>

6. The following program will store numbers 13h, 29h,25h into the array of type DWORD.

PSEUDO CODE	AL CODE	EAX	X		
Array X	x dword ? lea ebx,x		**dword 1**	**dword 2**	**dword 3**
X(1) := 13h	mov eax, 13h	**00 00 00 13**			
	mov [ebx], eax	00 00 00 13	**00 00 00 13**		
	add ebx,4	00 00 00 13	00 00 00 13		
X(2):= 29h	mov eax,29h	**00 00 00 29**	00 00 00 13		
	mov [ebx],eax	00 00 00 29	00 00 00 13	**00 00 00 29**	
	add ebx,4	00 00 00 29	00 00 00 13	00 00 00 29	
X(3):= 25h	mov eax,25h	**00 00 00 25**	00 00 00 13	00 00 00 29	
	mov [ebx],eax	00 00 00 25	00 00 00 13	00 00 00 29	**00 00 00 25**

Important: Since we are storing into individual byes for each dword, <u>we increment by 4.</u>

Exercise:

Write a assembly language program that will store the first 50 positive odd numbers.

■

Storing data in the array without a variable's location.

Arrays can also be created without using a variable location by simply using the

mov [register], source instruction

where the source is a register, containing the location where the first byte of

the array is to be stored.

For this instruction the following rules apply:

Rule 1: The register must be EAX, EBX, ECX, or EDX.

Rule2: The *source* can be a register of any data type.

Rule3: The *[register]* indicates the cell location where the bytes are to be located.

The [*register]* is call the indirect register.

Examples:

1.

AL CODE	EBX	AL	[EBX]
mov ebx,403030h	**403030**		
mov al, 9Ah	403030	**9A**	
mov [ebx], al	403030	9A	**9A**

2.

AL CODE	EBX	AX	[EBX]
mov ebx,403030h	**403030**		
mov ax, 569Ah	403030	**569A**	
mov [ebx], ax	403030	569A	**569A**

3.

AL CODE	EBX	EAX	[EBX]
mov ebx,403030h	**403030**		
mov eax, 2AC67569Ah	403030	**2AC67569A**	
mov [ebx], eax	403030	2AC67569A	**2AC67569A**

4.

AL	EAX	EBX	BYTE	1	2	3	4	5	6	7	8
mov eax, 1h	**1**										
mov ebx, 7D712Eh	1	**0007D712E**									
mov [eax], ebx	1	007D712E		**0 0**	**7 D**	**7 1**	**2 E**				
mov eax, 5h	**5**	007D712E		0 0	7 D	7 1	2 E				
mov ebx, 568923h	5	**00568923**		0 0	7 D	7 1	2 E				
mov [eax], ebx	5	00568923		0 0	7 D	7 1	2 E	**0 0**	**5 6**	**8 9**	**2 3**
mov ebx, 3h	5	**00000003**		0 0	7 D	7 1	2 E	0 0	5 6	8 9	2 3
mov [eax], ebx	5	00000003		0 0	7 D	7 1	2 E	**0 0**	**0 0**	**0 0**	**0 3**

Exercise:

Complete the table below.

AL INSTRUCTIONS	eax	ebx	BYTES	1	2	3	4	5	6	7	8
mov eax, 2											
mov ebx, 7D12Eh											
mov [eax], ebx											
mov eax, 4											
mov ebx, 568923h											
mov [eax], ebx											
mov ebx, 3											
mov [eax], ebx											

2. Write a assembly language program that will perform the following tasks:

Task 1: store the first 50 positive odd numbers.

Task 2: retrieve the first 50 positive odd numbers stored in task 1.

■

Retrieving data from an array.

The array elements of an array can be retrieved using the following instruction:

mov source, [register]

The *mov source, [register]* instruction will retrieve the number in the array at its beginning location and store it into the source where the following rules apply:

Rule 1: The register must be EAX, EBX, ECX, or EDX.

Rule2: The *source* must be a register of the same data type as the original array.

Rule3: The *[register]* indicates the cell location where the bytes are to be located.

The [*register]* is call the indirect register.

Examples:

1.

AL CODE	EBX	AL	[EBX]	CL
mov ebx,403030h	**403030**			
mov al, 9Ah	403030	**9A**		
mov [ebx], al	403030	9A	**9A**	
mov cl, [ebx]	403030	9a	9A	**9A**

2.

AL CODE	EBX	AX	[EBX]	CX
mov ebx,403030h	**403030**			
mov ax, 569Ah	403030	**569A**		
mov [ebx], ax	403030	569A	**569A**	
mov cx,[ebx]	403030	569A	569A	**569A**

3.

AL CODE	EBX	EAX	[EBX]	ECX
mov ebx,403030h	**403030**			
mov eax, 2AC67569Ah	403030	**2AC67569A**		
mov [ebx], eax	403030	2AC67569A	**2AC67569A**	
mov ecx, [ebx]	403030	2AC67569A	2AC67569A	**2AC67569A**

The following example is an extension of the above example and shows how the data from the array can be retrieved.
4.

AL CODE	AL	X		
x byte ? lea ebx,x		**byte 1**	**byte 2**	**byte 3**
mov al, 13h	**13**			
mov [ebx], al	13	**13**		
add ebx,1	13	13		
mov al,29h	**29**	13		
mov [ebx],al	29	13	**29**	
add ebx,1	29	13	29	
mov al,25h	**25**	13	29	
mov [ebx],al	25	13	29	**25**
sub ebx,2; Retrieving data	25	13	**29**	25
mov al,[ebx]	**13**	13	29	25
add ebx,1	13	13	29	25
mov al,[ebx]	**29**	13	29	25
add ebx,1	29	13	29	25
mov al,[ebx]	**25**	13	29	25

Exercise:

Extend the following program so that the array data stored can be retrieved in the register bx.

AL CODE	EAX	X		
x dword ? lea ebx,x		**dword 1**	**dword 2**	**dword 3**
mov eax, 13h	**00 00 00 13**			
mov [ebx], eax	00 00 00 13	**00 00 00 13**		
add ebx,4	00 00 00 13	00 00 00 13		
mov eax,29h	**00 00 00 29**	00 00 00 13		
mov [ebx],eax	00 00 00 29	00 00 00 13	**00 00 00 29**	
add ebx,4	00 00 00 29	00 00 00 13	00 00 00 29	

mov eax,25h	**00 00 00 25**	00 00 00 13	00 00 00 29	
mov [ebx],eax	00 00 00 25	00 00 00 13	00 00 00 29	**00 00 00 25**

■

Array lists

An alternative way to create one dimensional arrays is to list the array elements in the following directive:

variable name data type $n_1, n_2, \ldots, n_m$,

where the list is of the same data type.

There are 3 directives of this type:

variable name byte type $n_1, n_2, \ldots, n_m$

variable name word type $n_1, n_2, \ldots, n_m$

variable name dword type $n_1, n_2, \ldots, n_m$

Examples:

The following examples show how to retrieve listed arrays.

1.

AL CODE	**AL**	**X**		
		byte 1	**byte 2**	**byte 3**
x byte 3h, 7dh, 99h		**3**	**7d**	**99**
lea ebx,x		3	7d	99
mov al, [ebx]	**3**	3	7d	99
add ebx,1	3	3	7d	99
mov al, [ebx]	**7**	3	7d	99
add ebx,1	**7**	3	7d	99
mov al, [ebx]	**99**	3	7d	99

2.

AL CODE	**AX**	**X**		
		word 1	**word 2**	**word 3**
x word 37f2h,723dh, 0defah		**37f2**	**723d**	**defa**
lea ebx, x		37f2	723d	defa
mov ax, [ebx]	**37f2**	37f2	723d	defa
add ebx,2	37f2	37f2	723d	defa
mov ax, [ebx]	**723d**	37f2	723d	defa
add ebx,2	723d	37f2	723d	defa
mov ax, [ebx]	**defa**	37f2	723d	defa

3.

AL CODE	**EAX**	**X**		
		dword 1	**dword 2**	**dword 3**
x dword 4437f2h,21723dh, 0d276efah		**4437f2**	**21723d**	**d276efa**
lea ebx, x		4437f2	21723d	d276efa
mov eax, [ebx]	**4437f2**	4437f2	21723d	d276efa
add ebx,4	4437f2	4437f2	21723d	d276efa
mov eax, [ebx]	**21723d**	4437f2	21723d	d276efa
add ebx,4	21723d	4437f2	21723d	d276efa
mov eax, [ebx]	**d276efa**	4437f2	21723d	d276efa

15.3: Reserving Storage for an Array Using the DUP Directive.

There are times when it is important to set aside a block of memory that array values will be stored in. The reason is that without reserving a block of memory, data or code can be destroyed when cells are fill by an array. In fact it is recommended ,where possible, that the DUP directive always be used when creating arrays. To accomplish this we define an array A(dimension) using the following directive instructive in the data portion of the program:

variable name type dimension DUP (?)

Examples:

1. x byte 100 dup (?)

will create an array with a dimension of 100 byte cells:

2. x word 100 dup (?)

will create an array with a dimension of 100 WORD cells, consisting of 200 bytes.

3. x dword 100 dup (?)

will create an array with a dimension of 100 DWORD cells, consisting of 400 bytes.

Note: The *lea* instruction will still be use to determine the first byte position of the array.

Exercise:

Write a program that will perform the following tasks:

Task 1: Store in a dimensioned array the first 50 positive odd numbers.

Task 2: Store in another dimensioned array the first 50 positive even numbers.

Note: See model program below.
■

15.4 Working with Data

The following instruction will allow data to be directly stored into an array cell:

mov *DATA TYPE* PTR.

In order to avoid ambiguity about the data type, this instruction informs the assembler that the numeric value to be stored is to be identified as a given data type.

This instruction is defined as

mov *data type* PTR [register], *numeric value.*

For this move instruction, the following are the three different forms of the instruction: :

- mov *byte* PTR *[register]*, *numeric value*;

will define the size *of the numeric value* to be stored as a byte.

- mov *word* PTR *[register]*, *numeric value*;

will define the *size of the numeric value* to be stored as a word.

- mov *dword* PTR *[register]*, *numeric value*;

will define the *size of the numeric value* to be stored a dword.

Note: mov [*register*],source does not modify the contents of the register in question.

Examples:

1.

AL CODE	EBX	[EBX]
mov ebx,403030h	**403030**	
mov byte ptr [ebx], 9ah	403030	**9a**

2.

AL CODE	EBX	[EBX]
mov ebx,403030h	**403030**	
mov word ptr [ebx], 679ah	403030	**679a**

3.

AL CODE	EBX	[EBX]
mov ebx,403030h	**403030**	
mov dword ptr [ebx], 231abc9ah	403030	**231abc9a**

Arithmetic operators using [*register*]

For the following two integer arithmetic operators: addition, subtraction: the indirect register *[register]* can be a source for the following arithmetic instructions:

- add register, *[register]*

- add *[register]*, register

• sub register, *[register]*

• sub *[register]*, register

Examples:
1.

AL CODE	EAX	X
x byte 6		**6**
lea ebx, x		6
mov eax, 2	**2**	6
add eax, [ebx]	**8**	6

2.

AL CODE	EAX	X
x byte 2		**2**
lea ebx, x		2
mov eax, 8	**8**	2
sub eax, [ebx]	**6**	6

Exercises:

1. Complete the following table:

AL INSTRUCTIONS	eax	ebx	BYTES:	9	10	11	12	13	14	15	16	17
mov eax, 2ACD16 h												
mov ebx, 10												
add ebx, 1												
mov [ebx], eax												
add [ebx], ebx												
add eax, ebx												

2. Assume we have two arrays x, y containing the elements:

x: 2, 7, 9, 10

y: 123, 56, 11, 9

Write an assembly language program that will multiply the corresponding array elements and store the resulting product in an array z.
■

The cmp using [*register*]

The cmp instruction can be used to compare array elements. The instruction is of the following forms:

cmp [register], register

cmp register, [register]

Example:
.

AL CODE	EAX	X
x byte 6		**6**
lea ebx, x		6
mov eax, 7	**7**	6
cmp eax, [ebx]	7	6
ja bigger	7	6
jp not_bigger	7	6
bigger: mov eax, 0	**0**	6
jmp finished	0	6
not_bigger: mov eax, 1	0	6
finished:	0	6

15.5 Representing Two-Dimensional Arrays in Pseudo-Code.

Definition of a two-dimensional arrays *name*(r,c)

A two dimensional array is a collection of cells all of which have the same name, but are distinguished from one another by the use of 2 subscripts. A subscript is a positive integer number in parentheses which follows the array's name. The two dimensional array can be indicated by *name*(r,c) where r is the number of rows and c the number of columns.

Example: a(1,1), a(1,2), a(1,3) ,..., a(1,50),
a(2,1), a(2,2), a(2,3),...,   a(2,50),
::
a(100,1), a(100,2), a(100,3), ...a (100,50)

Such an array is said to have

r = 100 rows and

c = 50 columns.

Programming example:

The following program in pseudo - code will perform the following task:

Task: Assign array values a(j ,k) = j + k, for $1 \le j \le 100$; $1 \le k \le 10$

Program:

```
j : = 1
WHILE j ≤ 100
BEGIN
k := 1
WHILE k ≤ 10
   BEGIN
  a(j , k) := j + k
  k := k + 1
  END
j := j + 1
END
```

The following table shows the values stored in the array:

row/col	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	2	3	4	5	6	7	8	9	10	11
2	3	4	5	6	7	8	9	10	11	12
3	4	5	6	7	8	9	10	11	12	13
:::::::::	::::	::::	::::	::::	::::	::::	::::	::::	::::	::::
j	j + 1	j + 2	j + 3	j + 4	j + 5	j + 6	j + 7	j + 8	j + 9	j + 10
:::::::::	::::	::::	::::	::::	::::	::::	::::	::::	::::	::::
100	101	102	103	104	105	106	107	108	109	110

However, we have one small problem: the assembly language really only provides storing of data for one - dimensional arrays. Therefore, to program two dimensional arrays, we need to go back to the pseudo- code for one dimensional arrays where we can create the same results of a two dimensional array. To do this we define the two dimensional array a(j , k) as a one dimensional array as:

$a(j, k) := a(4*c*(j - 1) + 4*k - 3)$

where

$1 \le j \le r = 100$,

$1 \le k \le c = 10$,

Example:

For the above table and a(j, k) = j + k, where

r = 100

c = 10

$1 \le j \le 100$

$1 \le k \le 10$

a(j,k) := a(4∗10∗(j - 1) + 4k -3) = a(40∗(j - 1) + 4∗k - 3)

a(1,1) := a(1) a(1,2) := a(5) a(1,3) := a(9) a(1,4) := a(13) a(1,5) := a(17) a(1,6) := a(21) a(1,7) := a(25) a(1,8) := a(29) a(1,9) := a(33) a(1,10) := a(37)	a(2,1) := a(41) a(2,2) := a(45) a(2,3) := a(49) a(2,4) := a(53) a(2,5) := a(57) a(2,6) := a(61) a(2,7) := a(65) a(2,8) := a(69) a(2,9) := a(73) a(2,10) := a(77)	:::::::::::::::: :::::::::::::::: ::::::::::::::::	a(100,1) := a(3961) a(100,2) := a(3965) a(100,3) := a(3969) a(100,4) := a(3973) a(100,5) := a(3977) a(100,6) := a(3981) a(100,7) := a(3985) a(100,8) := a(3989) a(100,9) := a(3993) a(100,10) := a(3997)

Our pseudo-code program will now be changed to:

Program:

```
j : = 1
WHILE j ≤ 100
BEGIN
k := 1
WHILE k ≤ 10
BEGIN
a(40∗(j - 1) + 4∗k - 3) := j + k
 k := k + 1
   END
j := j + 1
END
```

Program

PSEUDO-CODE	AL PSEUDO-CODE	ASSEMBLY LANGUAGE CODE
J:= 1	J:= 1	mov j, 1
WHILE J ≤ 100	WHILE J ≤ 100	begin_row: cmp j,100 jg end_row
BEGIN	**BEGIN**	begin:
K := 1	K:= 1	mov k, 1
WHILE K ≤ 10	WHILE K ≤ 10	begin_col: cmp k, 10 jg end_col
BEGIN	**BEGIN**	begin_col:
a(40∗(J - 1) + 4∗K - 3) := J + K	EAX:= J	mov eax, j
	EAX := EAX -1	sub eax, 1
	EAX := 40∗EAX	mov f, 40
		mul f
	TEMP:= EAX	mov temp, eax
	EAX:= 4	mov eax, 4
	EAX:= EAX∗K	mul k
	EAX:= EAX - 3	sub eax, 3
	EAX:= EAX + TEMP	add eax, temp
	EBX:= J	mov ebx, j
	EBX:= EBX + K	add ebx, k
	[EAX] := EBX	mov [eax], ebx
K := K + 1	EAX: = K	mov eax, k
	EAX:= EAX + 1	add eax, 1
	K:= EAX	mov k, eax
END	**END**	jmp begin_col
		end_col: mov eax, j

J := J + 1	EAX: = J
	EAX:= EAX + 1
	J:= EAX
		mov eax, j
		add eax, 1
		mov j, eax
		jmp begin_row

END	**END**	end_row :

Exercise:

1. Modify the above program to allow the creation of the two dimensional array using the directive

a dword *dimension* dup (?)

■

Model Program

```
; The following program is a partial program that will store numbers 2,4,6,..., 10,000 into an array a.

.386
.MODEL FLAT
.STACK 4096
.DATA

a dword 5000 dup (?) ; Array a(dim 5000)

.CODE
_start:
lea ebx, a
mov k, 1
while: cmp k, 5000
begin: jg  end
; begin
mov eax, k
mul  2
mov [ebx], eax
mov eax, k
add eax, 1
mov k, eax
add ebx, 4
jmp while
end:
;end of assembly language code

PUBLIC_start

end
```

PROJECTS

1. A numeric conversion table is a table made up of decimal , hexadecimal and octal numbers in increasing order:

DECIMAL	OCTAL
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12
::::::::	:::::::::::
N	N_8

Write a program in assembly language that will create and a store the above table for any given value N.

2. Write a program that will find and store the first 50 prime numbers in an array.

CHAPTER 16 PROCEDURES

16.1 Pseudo-code Procedures

As in higher programming languages, we will need to use procedures (subroutines), repeatedly in many of our assembly language programs. These procedures in a sense can be thought as algorithms, in that they can stand alone and be used repeatedly in different programs. For pseudo-code, the following will be our the definition of the main body of the procedure:

Definition of pseud-code procedures:

PROCEDURE *name of procedure*
BEGIN

(*instructions)*

END

We will assume the following rules will apply to procedures:

Rule1: All procedures will be local to the main program.

Rule2: All procedures will be located at the end of the main program.

Rule3: All variables are global.

Rule4: The procedure will be ignored by the assembler, unless it is called by the Call instruction

Definition of the Call instruction:

CALL *name of procedure*

We will assume the following rules will apply to the call instruction:

Rule1: All call instructions can be inserted anywhere inside the main program.

Rule2: When the call instruction is activated, transfer is made to the first instruction of the procedure.

Rule3: The END at the end of the procedure, will transfer back to the instruction immediately following the call instruction.

Examples :

1. *The exponential operator* $p = a^N$. Although we define an exponential operator in pseudo-code, the exponential operator does not exist in the assembly language. Therefore we need to create a procedure that will perform the exponential operator that we have in our pseudo-code. For the following procedure we will compute $p = a^n$, where

$a > 0$
$n \geq 0$

```
PROCEDURE exponential
BEGIN
P := 1
K:= 1
WHILE K ≤ N
BEGIN
P:= A∗P
K:= K + 1
END
IF N:= 0 THEN
BEGIN
P:= A
END
```

The following program will use the above procedure and will perform the following task:

Task: Compute and store 5^7, 2^{10} .

PSEUDO-CODE	A	N	EXP1	EXP2	EXP3
A:= 5	**5**				
N:= 7	5	**7**			
CALL EXPONENTIAL	5	7			
EXP1:= P	5	7	**78125**		
A:= 2	**2**	7	78125		
N:= 10	2	**10**	78125		
CALL EXPONENTIAL	2	10	78125		
EXP2:= P	2	10	78125	**1024**	

PROCEDURE EXPONENTIAL
BEGIN
P := 1
K:= 1
WHILE K ≤ N
BEGIN
P:= A∗P
K:= K + 1
END
IF N:= 0 THEN
BEGIN
P:= 1
END					

2. The following procedure will perform the following tasks:

Task1: Compare the relative size of two different integer numbers x, y.

Task2: Returns the larger of the two numbers.

```
PROCEDURE compare
BEGIN
IF x > y THEN
BEGIN
larger := x
ELSE
BEGIN
larger := y
END
```

Write a program using the above procedure that will perform the following task:

Task1: Compare two pair of different integer numbers and store the larger in different varialbes.

PSEUDO-CODE	X	Y	LARGER	LARGER1	LARGER2
X := 5	5				
Y := 10	5	10			
CALL COMPARE	5	10			
LARGER1:= LARGER	5	10	**10**	**10**	
X : = 12	12	10	10	10	
Y : = 7	12	7	10	10	
CALL COMPARE	12	7	10	10	
LARGER2:= LARGER	12	7	**12**	10	**12**

PROCEDURE COMPARE
BEGIN
IF X > Y THEN
BEGIN
LARGER := X
ELSE
BEGIN
LARGER := Y
END					

3. The following procedure will perform the following task:

Task: For any positive integer N, compute the value sum = 1 + 2 + 3 + ... + N.

```
PROCEDURE sum
BEGIN
total := 0
k : = 1
WHILE k ≤ N
BEGIN
total := total + k
k := k + 1
END
END
```

Write a program using the above the procedure that will perform the following tasks:

Task1: Store the sum of the numbers 1,2,3,..., 100

Task2: Store the sum of the numbers 1,2,3,..., 150

Task3: Store the sum of the numbers 1,2,3,...., 250

PSEUDO-CODE	**N**	**TOTAL**	**TOTAL1**	**TOTAL2**	**TOTAL3**
N := 100	**100**				
CALL SUM	100				
TOTAL1:= TOTAL	100	**5050**	**5050**		
N:= 150	150	5050	5050		
CALL SUM	150	**11325**	5050		
TOTAL2:= TOTAL	150	11325	5050	**11325**	
N := 250	250	11325	5050	11325	
CALL SUM	250	11325	5050	11325	
TOTAL3:= TOTAL	250	**125500**	5050	11325	**125500**
PROCEDURE SUM BEGIN TOTAL := 0 K : = 1 WHILE K ≤ N BEGIN TOTAL := TOTAL + K K := K + 1 END END					

4. The following procedure will perform the following tasks:

Task1: Compare four array integer values.

Task2: Find and return the smallest integer value.

```
PROCEDURE array

BEGIN
smallest := a(1)

IF a(2) < smallest THEN
  BEGIN
  smallest := a(2)
  END

IF a(3) < smallest THEN
  BEGIN smallest := a(3)
  END

IF a(4) < smallest THEN
   BEGIN
   smallest := a(4)
  END

END
```

Write a program using the above the procedure that will perform the following tasks:

Task1: Find and store the smallest of the number: 5, 7, 2, 10

Task2: Find and store the smallest of the number, 57, 1001, 2222, 43

PSEUDO-CODE	**A(1)**	**A(2)**	**A(3)**	**A(4)**	**SMALLEST**	**S1**	**S2**
A(1) := 5	**5**						
A(2) := 7	5	**7**					
A(3) := 2	5	7	**2**				
A(4) := 10	5	7	2	**10**			
CALL ARRAY	5	7	2	10			
S1:= SMALLEST	5	7	2	10	**2**		
A(1) := 57	57	7	2	10	2		
A(2) := 1001	57	1001	2	10	2		

A(3) := 2222	57	1001	2222	10	2		
A(4) := 43	57	1001	2222	43	2		
CALL ARRAY	57	1001	2222	43	2		
S2:= SMALLEST	57	1001	2222	43	43		

```
PROCEDURE ARRAY
BEGIN
  SMALLEST := A(1)

IF A(2) < SMALLEST THEN
        BEGIN
   SMALLEST := A(2)
        END

IF A(3) < SMALLEST THEN
        BEGIN
  SMALLEST := A(3)
        END

IF A(4) < SMALLEST THEN
        BEGIN
  SMALLEST := A(4)
        END
END
```

EXERCISES:

1. Write a procedure that will perform the following tasks:

Task1: Store the following positive integer numbers in an array:

n, n + 1, n + 2, n + 3, ..., n + m,   m > 0.

Task2: Add the numbers stored in the array.

2. Write a procedure that will perform the following tasks:

Task1: Store n integers in an array.

Task2: Find the largest number of this array.

■

16.2 Writing procedures in Assembly Language

The assembly language syntax is very similar to pseudo-code:

Body of the procedure:

identifier PROC NEAR 32 ; *identifier:* the procedure's name

(*instructions)*

ret *;* will jump to the code following the call instruction.

identifier ENDP ; Terminates the body of the procedure.

The call instruction is simply :

call *identifier*

Examples:

1. From example 1 above, complete the table below:

PSEUDO-CODE	AL PSEUDO-CODE	ASSEMBLY LANGUAGE CODE
A:= 5	A:= 5	mov a, 5
N:= 7	N:= 7	mov n, 7
CALL EXPONENTIAL	CALL EXPONENTIAL	call exponential
EXP1:= P	EAX:= P	mov eax, p
	EXP1:= EAX	mov exp1, eax
A:= 2	A:= 2	mov a, 2
N:= 10	N:= 10	mov n,10
CALL EXPONENTIAL	CALL EXPONENTIAL	call exponential
EXP2:= P	EAX:= P	mov eax,p
	EXP2:= EAX	mov exp2, eax
PROCEDURE EXPONENTIAL	**PROCEDURE EXPONENTIAL**	exponential PROC NEAR 32
BEGIN	BEGIN	begin:
P := 1	P:= 1	mov p, 1
K:= 1	K:= 1	mov k, 1
WHILE K $\leq$ N	WHILE K$\leq$ N	while: cmp k, n
		jg end1
BEGIN	BEGIN	begin1:
P:= A$*$P	EAX:= P	mov eax, p

	MUL A	mul a
	P:= EAX	mov p, eax
K:= K + 1	EAX:= K	mov eax, k
	EAX:= EAX + 1	add eax, 1
	K:= EAX	mov k, eax
END	END	jmp while
		end1:
IF N:= 0 THEN	IF N:= 0 THEN	cmp ebx,0
		jg end2
BEGIN	BEGIN	begin2:
P:= 1	P:= 1	mov p, 1
END	END	end2:
		ret expontential ENDP

2. From example 3 above, complete the table below:

PSEUDO-CODE	**AL PSEUDO-CODE**	**ASSEMBLY LANGUAGE CODE**
N := 100	N:= 100	mov n, 100
CALL SUM	CALL SUM	call sum
TOTAL1:= TOTAL	TOTAL1 := TOTAL	mov eax, total mov total1, eax
N:= 150	N:= 150	mov n, 150
CALL SUM	CALL SUM	call sum
TOTAL2:= TOTAL	TOTAL2:= TOTAL	mov eax, total mov total2, eax
N := 250	N:= 250	mov n,250
CALL SUM	CALL SUM	call sum
TOTAL3:= TOTAL	TOTAL3:= EBX	mov eax, total mov total2, eax

PROCEDURE SUM
BEGIN
TOTAL := 0
K : = 1
WHILE K ≤ N
BEGIN
TOTAL := TOTAL + K
K := K + 1
END
END
	PROCEDURE SUM
	BEGIN
	TOTAL := 0
	K:= 1
	WHILE TOTAL ≤ N
	BEGIN
	EAX:= TOTAL
	EAX:= EAX + 1
	TOTAL:= EAX
	END
	END
		sum PROC NEAR 32
		mov total, 0
		mov k, 1
		cmp total, n
		begin:
		mov eax, total
		add eax, 1
		mov total, eax
		jle begin
		ret
		sum ENDP

Exercises:

1. For the remaining examples in 16.1, write appropriate assembly language codes.

2. Modify the a procedure that will compute a^n where n is a integer and the value a is a non-negative floating point number. .

■

PROJECT:

Write a program to compute

$P(x) = a_n x^n + a_{n-1}x^{n-1} +\dots + a_3x^3 + a_2x^2 + a_1x + a_0 = (\dots(((a_n x + a_{4n-1})x +\dots + a_3)x + a_2)x + a1)x + a_0$

using the following tasks:

Task 1: Store the integers a_k in an array.

Task 2: In a procedure, compute P(x).

II. WORKING WITH DECIMAL NUMBERS

CHAPTER 17 - DECIMAL NUMBERS

INTRODUCTION

So far we have only worked with integers in assembly language. For many assembly language compilers, decimal numbers are also available. In order be become a proficient assembly language programmer, one needs to have a good understanding how decimal numbers are represented in the assembler. To accomplish this, we start with the basic ideas of decimal numbers in the base 10. In later chapters we will expand these numbers to the various forms that are needed.

17.1 Definition of Decimal Numbers and Fractions.

Definition of Decimal Numbers Base 10: Decimal numbers are numbers of the following forms

$m.a_1a_2 a_3 .. a_n$

or

$m.a_1a_2a_3 ... a_n a_1a_2a_3 ... a_n ... a_1a_2a_3 ... a_n ...$

where m is a integer and $a_{1,}a_{2,}a_3$ ... are non-negative integers.

There are three types of decimal numbers: positive , negative and zero.

Examples: 0.123, - 0.06143, 4.54, 33.248248..., - 72.77777777777

Definition of Fractions: Fractions are defined as ± N/M, where N and M represent arbitrary integers, with the restriction that M ≠ 0.

2/3, - 4/7, 1/3, 124/456, - 7/7, 0/4, 400/200

There are two types of fractions: proper and improper.

Definition: A proper positive fraction N/M is a fraction where 0 < N < M.

Examples:

2/3, - 4/7, 1/3, 124/456

Definition: An improper positive fraction N/M is a fraction where N ≥ M > 0.

5/2, - 7/6, 10/ 5

Note: In this chapter we are primarily interested in positive proper fractions.

Exercises:

1. Which of the following fractions can be reduced to integer numbers:

a. 1446/558 **b.** 12356/2333 **c.** 458/3206 **d.** 1138/569

2. Rewrite the following numbers as fractions:

a. (1/2)/(5/7) **b.** (212/124)/(5) **c.** (1/3)/(2/3)

3. Which of the following fractions are proper:

a. 3/2 **b.** 234/567 **c.** 1/2
■

Note: For the following presentation, we will only consider decimal numbers that are generated from positive fractions.

17.2 Representing positive decimal numbers corresponding to proper fractions in expanded form.

Any fraction can be represented by a decimal number. Since we are mainly interested in fractions that are proper, this means that all corresponding decimal numbers we study will be less than 1.

There are two types of decimal numbers: finite and infinite:

Definition of finite decimal numbers: Finite decimal numbers are written in the form: $0.a_1a_2a_3 .. a_n$

where

$0.a_1a_2a_3 .. a_n = a_1/10 + a_2/10^2 + a_3/10^3 + ... + a_n/10^n$

and

a_k (k = 1, 2, ..., n) are non- negative integers.

Note: Finite decimal numbers can also be negative numbers.

Examples:

0.579 = 5/10 + 7/100 + 9/1000

0.3579 = 0.3579 = 3/10 + 5/100 + 7/1000 + 9/10000

0.49607 = 4/10 + 9/100 + 6/1000 + 0/10000 + 7/100000

0.005411 = 0/10 + 0/100 + 5/1000 + 4/10000 + 1/100000 + 1/1000000 =

5/1000 + 4/10000 + 1/100000 + 1/1000000

Definition of infinite decimal numbers : Infinite decimal numbers are written in the form:

$0.a_1a_2a_3 \ldots a_n a_1a_2a_3 \ldots a_n \ldots a_1a_2a_3 \ldots a_n \ldots$

where

$0.a_1a_2a_3 \ldots a_n a_1a_2a_3 \ldots a_n \ldots a_1a_2a_3 \ldots a_n \ldots =$

$a_1/10 + a_2/10^2 + a_3/10^3 + \ldots + a_n/10^n + a_1/10^{n+1} + a_2/10^{n+2} + a_3/10^{n+3} + \ldots + a_n/10^{2n} + \ldots$

and

a_k (k = 1, 2, ...,) are non- negative integers.

To avoid the complications of working with infinite expansions, we will use the following notation:

$0.a_1a_2a_3 \ldots a_n\ a_1a_2a_3 \ldots a_n \ldots a_1a_2a_3 \ldots a_n \ldots = 0.\overline{a_1a_2\ \ldots\ a_n}$

Also, we will assume that all the laws of arithmetic work when applied to infinite decimal numbers.

Examples:

$0.798798\ldots = 0.\overline{798}$

$0.015981598\ \ldots = 0.0\overline{1598}$

$0.66\ldots = 0.\overline{6}$,

$0.13241324\ldots = 0.0\overline{1324}$

$0.25897897897\ldots = 25.\overline{897}$

Examples:

$1/2 = 0.5$, $2/3 = 0.666\ldots = 0.\overline{6}$ $1/4 = 0.25$, $1/3 = 0.333\ldots = 0.\overline{3}$, $1/2 = 0.5$

$213/999 = 0.213213213\ldots = 0.\overline{213}$, $16/3 = 5.333\ldots = 5.\overline{3}$

Exercises:

1. Expand the following in the form: $0.\overline{a_1a_2\ \ldots\ a_n} = 0.a_1a_2a_3 \ldots a_na_1a_2a_3 \ldots a_n \ldots a_1a_2a_3 \ldots a_n \ldots$

a. $0.\overline{2357}$ **b.** $0.\overline{0097}$

2. Expand the following in the form $0.a_1a_2a_3 \ldots a_n\ a_1a_2a_3 \ldots a_n \ldots a_1a_2a_3 \ldots a_n .. = 0.\overline{a_1a_2\ \ldots\ a_n}$

a. 0.0768907689 ... **b.** 0.00235559055590 ...

3. Write the following fractions as decimal numbers using the upper bar notation where necessary:

a. 5/12 **b.** - 7/8 **c.** 5/6 **d.** 1/7 **e.** - 3/7

■

17.3 Converting Decimal Numbers to Fractions:

Finite decimal numbers can easily be converted to fractions by writing them first in the form:

$0.a_1a_2a_3 .. a_n = a_1/10 + a_2/10^2 + a_3/10^3 + \ldots + a_n/10^n = (a_1 * 10^{n-1} + a_2 * 10^{n-2} + \ldots + a_k * 10^{n-k} + \ldots + a_1)/10^n$.

and then sum the terms with a common denominator.

Examples:

0.5 = 5/10

0.579 = 5/10 + 7/100 + 9/1000 = (5 ∗ 100 + 7 ∗ 10 + 9)/1000 = 579/1,000

0.3579 = 0.3579 = 3/10 + 5/100 + 7/1000 + 9/10000 = (3 ∗ 1000 + 5 ∗ 100 + 7 ∗ 10 + 9)/10000 = 3,579/10,000

0.49607 = (4/10 + 9/100 + 6/1000 + 0/10000 + 7/100,000) = - 49607/100,000

0.005411 = 0/10 + 0/100 + 5/1000 + 4/10000 + 1/100000 + 1/1000000 =

(5 ∗ 1000 + 4 ∗ 100 + 1/10 + 1)/1000000 = 5411/1,000,000

Exercises:

1. Write the decimal numbers as fractions:

a. 0.0235 **b.** 0.1111215 **c.** 0.999999

■

Infinite decimal numbers of type $0.\overline{a_1a_2\ \ldots\ a_n}$ can also be converted into a fraction. The following algorithm[1] will demonstrate how this is done:

Step 1: Let $x = 0.\overline{a_1a_2\ \ldots\ a_n}$

[1]. An algorithm is a finite set of rules to compute a specific result.

Step 2: $10^n * x = a_1 a_2 a_3 . \ldots a_n . \overline{a_1 a_2 \ \ldots \ a_n}$

Step 3: $10^n * x - x = a_1 a_2 a_3 . \ldots a_n . \overline{a_1 a_2 \ \ldots \ a_n} - 0.\overline{a_1 a_2 \ \ldots \ a_n} = a_1 a_2 a_3 . \ldots a_n$

Step 4: $10^n * x - x = 99\ldots9x = a_1 a_2 a_3 . \ldots a_n$

Step 5: $x = a_1 a_2 a_3 . \ldots a_n / 99\ldots9$

Step 6: $0.\overline{a_1 a_2 \ \ldots \ a_n} = = a_1 a_2 a_3 . \ldots a_n / 99\ldots9$

Example:

Convert $0.\overline{21657}$ to a fraction :

Step 1: Let x = $0.\overline{21657}$ = 0.216572165721657 ...

Step 2: $10^5 * x = 100{,}000 * 0.\ 2165721657\ldots = 21657.\overline{21657}$

Step 3: Subtract the equation in step 1 from the equation in step 2:

$100{,}000 * x - x = 21657.\ 2165721657\ldots - 0.216572165721657\ \ldots = 21657$

Step 4: $100{,}000 * x - x = 99{,}999x = 21657$

Step 5: x = 21657/99999

Step 6: $0.\overline{21657}$ = 21657/99999

We can incorporate the above algorithm into a single basic formula:

$$0.\overline{a_1 a_2 \ \ldots \ a_n} = \frac{a_1 a_2 \ \ldots \ a_n}{10^n - 1}$$

Example:

Convert $0.\overline{21657}$ to a fraction:

$$0.\overline{21657} = \frac{21657}{10^5 - 1} = \frac{21657}{100000 - 1} = \frac{21657}{99999}$$

Exercises:

1. Write the following decimal numbers as fractions:

a. $0.\overline{23}$ **b.** $0.\overline{73}$ **c.** $0.\overline{8}$ **d.** $0.\overline{101}$ **e.** $0.\overline{3}$

g. 23.468 **h.** 2.0078 **I.** $0.246\overline{79852}$

2. Write the following decimal numbers as a single fraction p/q where p, q are integers:

a. $0.\overline{7323} + 0.\overline{83}$ **b.** $0.\overline{7323} - 0.\overline{83}$ **c.** $0.\overline{7323} * 0.\overline{83}$ **d.** $0.\overline{7323} / 0.\overline{83}$

3. Write the following decimal numbers as a decimal number $0.\overline{a_1 a_2 \ldots a_n}$:

a. $0.\overline{7323} + 0.\overline{0083}$ **b.** $0.\overline{7323} - 0.\overline{0083}$ **c.** $0.\overline{7323} * 0.\overline{83}$ **d.** $0.\overline{7323} / 0.\overline{83}$

■

17.4 Converting Fractions to Decimal Numbers:

Assume that N/M is a positive proper fraction. We define the decimal representation of N/M as

$M/N = a_1/10 + a_2/10^2 + a_3/10^3 + \ldots.$

where a_k are non-negative integers.

The following example will demonstrate the conversion from a fraction to a decimal number:

Example:

Convert 3/7 to its decimal representation.

$3/7 = a_1/10 + a_2/10^2 + a_3/10^3 + a_4/10^4 + a_5/10^5 + a_6/10^6 + a_7/10^7 + \ldots$

Step 1: $10(3/7) = 30/7 = (28 + 2)/7 = 4 + 2/7 = a_1 + a_2/10 + a_3/10^2 + a_4/10^3 + a_5/10^4 + a_6/10^5 + a_7/10^6 + \ldots$

$a_1 = 4$

$2/7 = a_2/10 + a_3/10^2 + a_4/10^3 + a_5/10^4 + a_6/10^5 + a_7/10^6 + \ldots$

Step 2: $10(2/7) = 20/7 = (14 + 6)/ 7 = 2 + 6/7 = a_2 + a_3/10 + a_4/10^2 + a_5/10^3 + a_6/10^4 + a_7/10^5 + \ldots$

$a_2 = 2$

$6/7 = a_3/10 + a_4/10^2 + a_5/10^3 + a_6/10^4 + a_7/10^5 + \ldots$

Step 3: $10(6/7) = 60/7 = (56 + 4)/ 7 = 8 + 4/7 = a_3 + .a_4/10 + a_5/10^2 + a_6/10^3 + a_7/10^4 + \ldots$

$a_3 = 8$

$4/7 = a_4/10 + a_5/10^2 + a_6/10^3 + \ldots$

Step 5: $10(4/7) = 40/7 = (35 + 5)/ 7 = 5 + 5/7 = a_4 + a_5/10 + a_6/10^2 + \ldots$

$a_4 = 5$

$5/7 = a_5/10 + a_6/10^2 + \ldots$

Step 6: $10(5/7) = 50/7 = (49 + 1)/7 = 7 + 1/7 = a_5 + a_6/10 + a_7/10^2 + \ldots$

$a_5 = 7$

$1/7 = a_6/10 + a_7/10^2 + \ldots$

Step 7: $10(1/7) = 10/7 + (7 + 3)/7 = 1 + 3/7 = a_6 + a_7/10 + \ldots$

$a_6 = 1$

$3/7 = a_7/10 + \ldots$

Since we cycled back to 3/7 we can write:

$3/7 = 0.42857142857142857142857142857143 \ldots = 0.\overline{428571}$

Exercise:

Convert the following fractions to decimal:

1. 4/9

2. 3/8

3. 67/5

■

17.5 Representation of Decimal Numbers

Every finite decimal number has 2 representations.

Examples:

a. $0.\overline{9}$

Step 1: $x = 0.\overline{9} = 0.99....$

Step 2: $10x = 9.99....$

Step 3: Subtract the equation in step 1 from the equation in step 2:

$9x = 9$

Step 4: $x = 0.\overline{9} = 1.$

b. $0.00\overline{9}$

Step 1: $0.00\overline{9} = \overline{9}/100 = 1/100 = 0.01$

c. $24.\overline{9}$

$24.\overline{9} = 24 + 0.\overline{9} = 24 + 1 = 25$

d. $0.2354\overline{9}$

$0.2354\overline{9} = 0.2354 + 0.0000\overline{9} = 0.2354 + 0.0001 = 0.2355$

Exercises:

1. Convert the following into integer form:

a. $281.\overline{9}$ b. $41256.\overline{9}$

2. Write the following into fraction form:

a. $0.23\overline{8}$ **b.** $0.00\overline{791}$ **c.** $0.\overline{1110000}$

3. Explain why we cannot convert, using our above algorithm, the following number into a fraction:

0.272772777277772777772...

From your analysis, does such a number exist ?
■

Project

We assume in this chapter that we can apply the ordinary rules of decimal arithmetic to infinite decimal numbers. For the following infinite decimal numbers, find the fraction that presents p/q where p, q are integers.

1.
a. $10^n * 0.\overline{a_1 a_2 \ldots a_n} = p/q$

b. $0.\overline{a_1 a_2 \ldots a_n} + 0.\overline{b_1 b_2 \ldots b_m} = p/q$

c. $0.\overline{a_1a_2\ \dots\ a_n} - 0.\overline{b_1b_2\ \dots\ b_m}$ = p/q

d. $0.\overline{a_1a_2\ \dots\ a_n} * 0.\overline{b_1b_2\ \dots\ b_m}$ = p/q

e. $0.\overline{a_1a_2\ \dots\ a_n} / 0.\overline{b_1b_2\ \dots\ b_m}$ = p/q

2. Show the following is true:

a. $a_1/10 + a_2/10^2 + a_3/10^3 + \dots + a_n/10^n = (a_1a_2 \dots a_n)/10^n$

b. $a_1/10 + a_2/10^2 + a_3/10^{3} + \dots + a_n/10^n + a_1/10^{n+1} + a_2/10^{n+2} + a_3/10^{n+3} + \dots + a_n/10^{2n} + \dots =$

$(a_1a_2 \dots a_n)/10^n + (a_1a_2 \dots a_n)/10^{2n} + (a_1a_2 \dots a_n)/10^{3n} + \dots$

3. Write an assembly language program that will perform the following tasks:

Task1: Assume n/m = $= 0.\overline{a_1a_2\ \dots\ a_n}$

Compute the values a_k of $0.\overline{a_1a_2\ \dots\ a_n}$ and store them in an array.

CHAPTER 18 - DIFFERENT NUMBER BASIS FOR FRACTIONS (OPTIONAL)

INTRODUCTION

In Chapter 2, we restricted our studies to integer numbers of different bases. We now move on to the study of decimal numbers of different bases. It is important to understand that to become a successful assembly programmer one has to have a complete understanding how both integer and decimal numbers work within the assembler system.

18.1 Definition of Decimal and Fractions

In Chapter 17, we defined finite and infinite decimal numbers in the base 10 as

$0.a_1a_2a_3...a_n = a_1/10 + a_2/10^2 + ... + a_n/10^n$

$0.a_1a_2a_3...a_n\ a_1a_2a_3...a_n ... = 0.\overline{a_1a_2\ ...\ a_n} = a_1/10 + a_2/10^2 + ... + a_n/10^n + a_1/10^{n+1} + a_2/10^{n+2} + ... + a_n/10^{2n} + ...$

Examples:

$0.25 = 2/10 + 5/10^2$

$0.0625 = 6/10^2 + 2/10^3 + 5/10^4$

$0.3333... = 3/10 + 3/10^2 + 3/10^3 + ...$

$0.\overline{285714} = 2/10 + 8/10^2 + 5/10^3 + 7/10^4 + 1/10^5 + 4/10^6 + 2/10^7 + 8/10^8 + 5/10^9 + 7/10^{10} + 1/10^{11} + 4/10^{12} + ... +$

In a similar manner we can define finite and infinite decimal numbers, less than 1, for any base b in expanded form:

Definition: A finite non negative decimal number less than 1 can be written in the base b as*:*

$(0.a_1a_2a_3...a_n)_b = a_1/10 + a_2/10^2 + ... + a_n/10^n$

where

$0 \leq a_k < b$ (k = 1, 2, ...,n),

$a_1/10_b + a_2/10_b{}^2 + ... + a_n/10_b{}^n = 0.a_1 + 0.0a_2 + ... + 0.00...0a_n$

Definition: An infinite decimal number less than 1 can be written in the base b as *:*

$(0.a_1a_2a_3...a_n ...)_b = a_1/10_b + a_2/10_b{}^2 + ... + a_n/10_b{}^n + ...$

where

$0 \leq a_k < b$ (k = 1, 2, ...)

$a_1/10_b + a_2/10_b{}^2 + ... + a_n/10_b{}^n + ... = 0.a_1 + 0.0a_2 + ... + 0.00...0a_n + ...$

Note: We are only using these decimal expansions to indicate the various position of the decimal point; not for computational values.

Examples:

$0.11101_2 = 1/10_2 + 1/10_2{}^2 + 1/10_2{}^3 + 0/10_2{}^4 + 1/10_2{}^5$

$0.02756_8 = 0/10_8 + 2/10_8{}^2 + 7/10_8{}^3 + 5/10_8{}^4 + 6/10_8{}^5$

$0.\overline{98C7DF}_{16} = 9/10 + 8/10^2 + C/10^3 + 7/10^4 + D/10^5 + F/10^6 + ...$

Exercises:

1. Write the following numbers in expanded form:

a. 0.231120_4 **b.** 0.11111101_2 **c.** 0.232323_8 **d.** $0.ABC2_{16}$
■

18.2 Converting Decimal Numbers Between The base 10 and an Arbitrary Base

As we stated in Chapter 4, it is important to be able to convert integer numbers from a given number base to corresponding integer numbers in any other base. Similarly , we wish to do the same for fractions. First we will define the corresponding decimal number ($N_b < 1$) that corresponds to a unique decimal number in the base 10.

Converting finite decimal numbers in any base b to its corresponding decimal numbers in the base 10:

The following formula gives a one - to - one correspondence from a finite decimal number in the base b to a unique finite decimal number in the base 10:

$N_b = 0.a_1a_2... a_n \Leftrightarrow a_1/b + a_2/b^2 + ... + a_n/b^n = N_{10}$

Note: All computation is done in decimal.

Examples:

$0.321_4 \Leftrightarrow 3/4 + 2/4^2 + 1/4^3 = 3/4 + 2/16 + 1/64 = 0.75 + 0.125 + 0.015625 = 0.890625_{10}$

$0.11011_2 \Leftrightarrow 1/2 + 1/2^2 + 1/2^4 + 1/2^5 = 0.5 + 0.25 + 0.0625 + 0.03125 = 0.84375_{10}$

$0.9AF_{16} \Leftrightarrow 9/16 + 10/16^2 + 15/16^3 = 0.5625 + 0.0390625 + 0.003662109375 =$

0.605224609375_{10}

Exercises:

1. Convert the following numbers to the base 10:

a. 0.231120_4 **b.** 0.11111101_2 **c.** 0.232323_8 **d.** $ABC2_{16}$
■

Converting infinite decimal numbers in any base b to its corresponding decimal numbers in the base 10:

The following formula will convert any infinite decimal number in the base b to its corresponding decimal number in the base 10:

Assume $\bar{a}_b = \overline{0.a_1a_2...a_n}$

Let $a_b = 0.a_1a_2...a_n \Leftrightarrow a_{10} = a_1/b + a_2/b^2 + ... + a_n/b^n$ then

$$\bar{a}_b \Leftrightarrow a_{10}\frac{b^n}{b^n - 1}$$

Examples:

a. Find $0.\bar{3}_4 \Leftrightarrow N_{10}$

Step 1: b = 4

Step 2: n = 1

Step 3: $\bar{a}_4 = 0.\bar{3}$

Step 4: $a_4 = 0.3$

Step 5: $a_{10} = 3/4$

Step 6: Substituting in the above formula gives

$0.\overline{3} \Leftrightarrow (3/4)*[4/(4 - 1)] = 3/4*4/3 = 1$

b. Find $0.\overline{101}_2 \Leftrightarrow N_{10}$

Step 1: b = 2

Step 2: n = 3

Step 3: $\overline{a_2} = 0.\overline{101}$

Step 4: $a_{2.} = 0.101$

Step 5: $a_{10} = 1/2 + 0/2^2 + 1/2^3 = 4/8 + 1/8 = 5/8$

Step 6: Substituting in the above formula gives

$0.\overline{101}_2 = (5/8) * [2^3/(2^3 - 1)] = 5/7$

Exercises:

1. Convert the following numbers to the base 10:

a. $0.\overline{6}_8$ **b.** $0.\overline{01001}_2$ **c.** $0.\overline{A5C}_{16}$ **d.** $0.\overline{00365}_8$

■

Converting finite decimal numbers in the base 10 to its corresponding decimal numbers in any base b:

$N_{10} = (a_1/b + a_2/b^2 + ... + a_n/b^n) + (a_1/b^{n+1} + a_2/b^{n+2} + ... + a_n/b^{2n}) + ... + \Leftrightarrow (.a_1a_2... a_n ...)_b$

The following examples will demonstrate how to solve the values a_k:

Examples:

Convert the following decimal numbers to the indicated base.

a. Convert 0.2 to the base 4.

Step 1: $0.2 = a_1/4 + a_2/4^2 + a_n/4^3 + ...$

Step 2: $4*(0.2) = 0.8 = a_1 + a_2/4 + a_3/4^2 + ...$

Step 3: Since a_1 is an integer, $a_1 = 0$.

Step 4: $0.8 = a_2/4 + a_3/4^2 + ...$

Step 5: $4*(0.8) = 3.2 = a_2 + a_3/4 + ...$

Step 6: $a_2 = 3$

Step 7: $0.2 = a_3/4 + a_4/4^2 + ...$

Since we are back to Step 1, the decimal number in the base 4 can be written as

$0.2_{10} \Leftrightarrow N_4 = 0.0303... = 0.\overline{a_1 a_2}_2 = 0.\overline{03}$

b. Convert 0.9 to the base 16.

Step 1: $0.9 = a_1/16 + a_2/16^2 + a_n/16^3 + ...$

Step 2: $16*(0.9) = 14.4 = a_1 + a_2/16 + a_3/16^2 + ...$

Step 3: Since a_1 is an integer, $14 \Leftrightarrow a_1 = E$

Step 4: $0.4 = a_2/16 + a_3/16^2 + ...$

Step 5: $16*(0.4) = 6.4 = a_2 + a_3/16 + ...$

Step 6: $a_2 = 6$

Step 7: $0.4 = a_3/16 + a_4/16^2 + ...$

Step 8: Since we are back to Step 4, the decimal number can be written as

$0.9_{10} \Leftrightarrow N_{16} = 0.E666... = 0.E\overline{6}$

c. Convert 0.8 to the base 2.

Step 1: $0.8 = a_1/2 + a_2/2^2 + a_n/2^3 + ...$

Step 2: $2*(0.8) = 1.6 = a_1 + a_2/2 + a_3/2^2 + ...$

Step 3: $a_1 = 1$

Step 4: $0.6 = a_2/2 + a_3/2^2 + ...$

Step 5: $2*(0.6) = 1.2 = a_2 + a_3/2 + ...$

Step 6: $a_2 = 1$

Step 7: $0.2= a_3/2 + a_4/2^2 + ...$

Step 8: $2*(0.2) = 0.4 = a_3 + a_4/2 + ..$

Step 9: $a_3 = 0$

Step 10: $0.4= a_4/2 + a_5/2^2 + ...$

Step 11: $2*(0.4) = 0.8 = a_4 + a_5/2 + ..$

Step 12: $a_4 = 0$

Step 13: $0.8 = a_5/2 + a_6/2^2 + ...$

At this point we are back to Step 1:

Step 12: Therefore $0.8 \Leftrightarrow 0.\overline{a_1a_2a_3a_4}_2 = 0.\overline{1100}_2$

Checking out computation.

By applying the above formula :

$$\bar{a}_b \Leftrightarrow a_{10}\frac{b^n}{b^n - 1}$$

we can check to see if we correctly converted the finite decimal number.

Example:

Let us check to see that we correctly converted 0.8_{10} to binary $0.\overline{1100}_2$.

Step 1: $\bar{a}_2 = 0.\overline{1100}$

Step 2: $a_{10} = 1/2 + 1/2^2 = 1/2 + 1/4 = 3/4$

Step 3: b = 2

Step 4: n = 4

Step 5: Substituting in the above formula gives

$0.\overline{11}_2 \Leftrightarrow (3/4) * [2^4/(2^4 -1)] = (3/4)(16/15) = 0.8$

Exercises:

1. Convert 0.55 to the

a. base 2. **b.** base 4 **c.** base 8 **d.** base 16

and check your results.

4. Show $0.0\overline{21}_4 \Leftrightarrow 0.15_{10}$

5. Show $0.111001\overline{100}_2 \Leftrightarrow 0.9_{10}$

■

Converting infinite decimal numbers in the base 10 to its corresponding decimal numbers in any base b:

We will use the same method of converting a finite decimal number in the base 10 to any number in the base b
by replacing the finite decimal number by an infinite decimal number.

Example.

Convert $\mathbf{0.\overline{8}_{10} \Leftrightarrow N_2}$

Step 1: $\mathbf{0.\overline{8}_{10}}$ = 0.888888... = $a_1/2 + a_2/2^2 + a_n/2^3 + ...$

Step 2: $2*(0.8888...)$ =1.77777... = $a_1 + a_2/2 + a_3/2^2 + ...$

Step 3: $a_1 = 1$

0.77777 ... = $a_2/2 + a_3/2^2 + ...$

Step 4: $2*(0.77777...)$ = 1.55555... = $a_2 + a_3/2 + a_4/2^2$ ...

Step 5: $a_2 = 1$

$0.55555... = a_3/2 + a_4/2^2 + ...$

Step 6: $2*(0.55555...) = 1.1111... = a_3 + a_4/2 + a_5/2^2..$

Step 7: $a_3 = 1$

$0.1111... = a_4/2 + a_5/2^2 + ...$

Step 8: $2*(0.1111...) = 0.2222... = a_4 + a_5/2 + a_6/2^2 + ...$

Step 9: $a_4 = 0$

$0.2222... = a_5/2 + a_6/2^2 + ...$

Step 10: $2*(0.2222...) = 0.4444... = a_5 + a_6/2 + a_7/2^2 + ...$

Step 11: $a_5 = 0$

$0.4444... = a_6/2 + a_7/2^2 + ...$

Step 12: $2*(0.4444...) = 0.8888... = a_6 + a_7/2 + ...$

Step 13: $a_6 = 0$

$0.8888... = a_7/2 + ...$

Step 14: Since step 13 returns to the original value in step 1, we are finished and conclude:

$$0.\overline{8}_{10} \Leftrightarrow 0.a_1a_2\,a_3a_4a_5a_6\ 0.a_1a_2\,a_3a_4a_5a_6 ... = (111000111000...)_2 = 0.\overline{111000}_2$$

Checking out computation.

By applying the above formula :

$$\overline{a}_b \Leftrightarrow a_{10}\frac{b^n}{b^n - 1}$$

we can check to see if we correctly converted the infinite decimal number $0.\overline{8}_{10}$.

Assume $\bar{a}_b = 0.\overline{111000}_2$

Let $a_2 = 0.111000 \Leftrightarrow a_{10} = 1/2 + 1/2^2 + 1/2^3 = 0.5 + 0.25 + 0.125 = 0.875$ then

$b = 2$

$n = 6$

$0.\overline{111000}_2 \Leftrightarrow (0.875)*2^6/(2^6 - 1) = 0.\overline{8}_{10}$

Exercises:

6. Convert and check your results:

Convert: $0.\overline{2}_{10}$ to **a.** base 2 **b.** base 8 **c.** base 16
■

Alternative Method for Converting Infinite Decimal Numbers in the Base 10 to its Corresponding Decimal Numbers in any base b:

Rather then working with infinite decimal numbers in the base 10, first convert the number to a fraction N/M and then write

$N/M = a_1/b + a_2/b^2 + ... + a_n/b^n + ... \Leftrightarrow (.a_1a_2... a_n)_b$

and solve for a_k (k = 1,2,3,...).

Example:

$0.\overline{3}_{10} = 1/3$ to the base 4.

Step 1: $1/3 = a_1/4 + a_2/4^2 + ... + a_n/4^n + ...$

Step 2: $4*(1/3) = 1 + 1/3 = a_1 + a_2/4 + ... + a_n/4^n + ...$

$a_1 = 1$ and $1/3 = a_2/4 + ... + a_n/4^n + ...$

Since the fraction 1/3 has repeated:

$0.\overline{3}_{10} \Leftrightarrow 0.\overline{1}_4$

Exercises:

1. Using the alternative method, convert $0.\overline{8}_{10} \Leftrightarrow \mathbf{N_2}$.
■

18.3 Converting Decimal Numbers In a Given Base To Fractions In the Same Base:

Finite decimal numbers in a base b can easily be converted to fractions by writing them first in the form:

$(0.a_1a_2a_3 .. a_n)_b = a_1/10 + a_2/10^2 + a_3/10^3 + ... + a_n/10^n = [(a_1 *10^{n-1} + a_2 *10^{n-2} + ... + a_k *10^{n-k}$

$+... a_1)/10^n]_b$.

Examples:

$0.5_8 = (5/10)_8$

$0.1011_2 = 1/10 + 0/100 + 1/1000 + 1/10000 = (1*1000 + 1*10 + 1)/10000 = (1011/10000)_2$

$0.3DF2_{16} = 3/10 + D/100 + F/1000 + 2/10000 = (3*1000 + D*100 + F*10 + 2)/10000 =$

$(3DF2/10000)_{16}$

Exercise:

1. Write the decimal numbers as fractions:

a. 0.0235_8 **b.** 0.110111_2 **c.** 0.999999_{16}
■

Infinite decimal numbers of type $0.\overline{a_1a_2 \ ... \ a_n}_{\ b}$ can also be converted into a fraction by using the basic formula developed in Chapter 17:

$$0.\overline{a_1a_2 \ ... \ a_n}_{\ b} = \left(\frac{a_1a_2 \ ... \ a_n}{10^n - 1}\right)_b$$

where $10^n - 1 = d_1d_2 ... d_n$

where d_k = b - 1 (k = 1,2, ..., n), the largest digit in the base b.

Examples:

$0.\overline{723}_8$ = 723/(1000 - 1) = 723/777 $_8$

$0.\overline{10}_2$ = 10/(100 - 1) = 10/11 $_2$

$0.\overline{3FA9}_{16}$ = 3FA9/(10000 - 1) = 3FA9/FFFF $_{16}$

Exercise:

1. Write the decimal numbers as fractions in the same base:

a. 0.0101_2 **b.** $0.\overline{000723}_8$ **c.** $0235.\overline{7237}_8$ **d.** $02C5.\overline{7239}_{16}$

■

18.4 Converting Numbers Between Different Bases

There exists a one to one correspondence between different bases. This can be shown by converting a number in one base to the base 10 and then convert this number to the other base.

Examples:

a. $0.2_4 \Leftrightarrow N_6$

$0.2_4 \Leftrightarrow N_{10} = 2/4 = 0.5$

$0.5 = a_1/6 + a_2/6^2 + a_3/6^3 + \ldots$

$6 * 0.5 = a_1 + a_2/6 + a/6^2 + \ldots = 3.0$

$a_1 = 3,\ a_2 = 0,\ a_3 = 0,\ ..$

$0.5 \Leftrightarrow 0.3_6$

$0.2_4 \Leftrightarrow 0.3_6$

b. $0.6_8 \Leftrightarrow N_2$

$0.6 \Leftrightarrow N_{10} = 6/8 = .75$

$0.75 = a_1/2 + a_2/2^2 + a_3/2^3 + ...$

$2*(0.75) = a_1 + a_2/2 + a_3/2^2 + ... = 1.5$

$a_1 = 1$

$0.5 = a_2/2 + a_3/2^2 + ...$

$2*0.5 = a_2 + a_3/2 + ... = 1$

$a_2 = 1, a_3 = 0, a_4 = 0, ...$

$0.5 \Leftrightarrow 0.11_2$

$0.6_8 \Leftrightarrow 0.11_2$

c. $0.A_{16} \Leftrightarrow N_2$

$0.A \Leftrightarrow N_{10} = 10/16 = 0.625_{10}$

$0.625 = a_1/2 + a_2/2^2 + a_3/2^3 + ...$

$2*(0.625) = a_1 + a_2/2 + a_3/2^2 + ... = 1.25$

$a_1 = 1$

$2*(0.25) = a_2 + a_3/2 + ... = 0.5$

$a_2 = 0$

$2*(0.5) = a_3 + a_4/2 + ... = 1$

$a_3 = 1$

$0.625_{10} \Leftrightarrow 0.101_2$

$0.A_{16} \Leftrightarrow 0.101_2$

Exercises:

1. Convert the following:

a. $0.AB_{16} \Leftrightarrow N_4$ **b.** $0.258_{16} \Leftrightarrow N_8$ **c.** $0.01_2 \Leftrightarrow N_{16}$

■

Quick conversions between the base 2 and base 16.

With no computation we can convert a number in the base 2 to its corresponding number in the base 16.

To convert from base 2 to base 16 or conversely, we need to construct the following table:

BASE2 DIGITS Binary	**BASE 16DIGITS** Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Converting a finite decimal number less than one

The following 2 rules show how to convert a finite binary number to a hexadecimal number:

1. From left to right, group the digits of the binary number in groups of 4; adding zeros at the end if necessary .

2. Match each group of these 4 digits with the corresponding hexadecimal digits from the about table.

Example: Convert 1101111011_2 to its corresponding hexadecimal digit.

We first write: $0.1101111011_2 = 0.\underline{1101}\ \underline{1110}\ \underline{1100}$

Next we match from the above table the corresponding hexadecimal digit:

$$0.1101111011_2 = 0.\underline{1101}\ \underline{1110}\ \underline{1100} \Leftrightarrow 0.DEC_{16}$$
$$0.\ \ D\ \ \ E\ \ \ C_{16}$$

To convert a finite hexadecimal number to a binary number just match each hexadecimal digit with the corresponding binary digits in the above table:

Example: Convert $0.F3DB_{16}$ to its corresponding binary number.

$$0.F3DB_{16} = 0.\ \ F\ \ \ 3\ \ \ D\ \ \ B \Leftrightarrow 0.1111\ 0011\ 1101\ 1011_2$$
$$0.\ 1111\ 0011\ 1101\ \ 1011$$

Exercises:

1. Using this quick conversion, convert the following binary numbers to hexadecimal:

a. 0.011010101_2 **b.** 0.0001111101_2

2. Using this quick conversion, convert the following hexadecimal numbers to binary:

a. 0.5623_{16} **b.** $0.ACF230A_2$

3. In the example above, we converted $0.1101111011_2 \Leftrightarrow 0.DEC_{16}$**.** Use another conversion method. Is the result the same.

5. Set up a quick conversion system between the base 2 and base 8.

6. Convert **a.** 0.110111011_2 to the base 8. **b.** Convert 0.23461_8 to the base 2.

7. Use quick conversion, to convert 0.76123_8 to the base 16.
■

Converting an infinite decimal number less than one

When converting an infinite binary number to hexadecimal we to use the following rules:

1. From left to right, group the digits of the binary number in groups of 4; adding zeros at the end if necessary .
If we cannot group the digits in groups of 4, expand the binary number to a minimal number of digits that wil allow the grouping.

2. Match each group of these 4 digits with the corresponding hexadecimal digits from the about table.

Example:

Convert $0.\overline{1001}_2$ to hexadecimal.

$$0.\overline{1001}_2 = 0.\underline{1001}\ \ \underline{1001}\ \ \underline{1001}\ \ldots \Leftrightarrow 0.\overline{9}_{16}$$

0. 9 9 9 ...

Example: Convert $0.\overline{11011011}_2$ to hexadecimal.

$$0.\overline{11011011} = 0.1101\ \ 1011\ \ 1101\ \ 1011\ \ 1101\ \ 1011\ \ldots \Leftrightarrow 0.\overline{DB}_{16}$$

0. D B D B D B ...

Example:

Convert $0.\overline{10}_2$ to hexadecimal.

Since we don't have a multiple of 4 digits, we expand:

$$0.\overline{10}_2 = 0.\overline{1010}_2 = 0.1010\ 1010\ 1010\ 1010\ \ldots \Leftrightarrow 0.\overline{A}_{16}$$

0. A A A A ....

Example:

Convert $0.\overline{101}_2$ to hexadecimal.

Since we don't have a multiple of 4 digits, we expand:

$$0.\overline{101} = 0.101101101101\ \ldots = 0.1011\ \ 0110\ \ 1101\ \ldots \Leftrightarrow 0.\overline{B6D}_{16}$$

0. B 6 D

Example:

Convert $0.\overline{9A3DD}_{16}$ to binary.

$$0.\overline{9A3DD}_{16} \Leftrightarrow 0.\underline{1001}\ \underline{1010}\ \underline{0011}\ \underline{1101}\ \underline{1101} = \overline{0.1001\ 1010\ 0011\ 1101\ 1101}_2$$

Exercises:

1. Let $\bar{a} = 0.\overline{139}_{10}$. Write each of the following as a single infinite decimal number:

a. $2 * \bar{a}$ **b.** $3 * \bar{a}$ **c.** $4 * \bar{a}$ **d.** $5 * \bar{a}$ **e.** $6 * \bar{a}$ **f.** $7 * \bar{a}$ **g.** $8 * \bar{a}$ **h.** $9 * \bar{a}$

2. Convert the following binary numbers to hexadecimal:

a. $0.\overline{1}_2$

b. $0.\overline{10111}_2$

c. $0.\overline{101101101}_2$

3. Convert the following hexadecimal numbers to binary.

a. $0.\overline{F}_{16}$ **b.** $0.\overline{23ADF}_{16}$

4. Show that the largest positive 32 bit number $0.1111...1_2$ corresponds to the decimal number $1 - 1/2^{32}$.

■

18.5 Performing Arithmetic On Finite Decimal Numbers In Different Number Bases

From Chapter 2, we can extend the invariant theorem to apply to decimal numbers. This theorem can also be extended to division:

Theorem: Invariant properties of arithmetic operations between bases:

1. Invariant property of addition: If $N_b \Leftrightarrow N_c$ and $M_b \Leftrightarrow M_c$ then $N_b + M_b \Leftrightarrow N_c + M_c$.

2. Invariant property of subtraction: If $N_b \Leftrightarrow N_c$ and $M_b \Leftrightarrow M_c$ then $N_b - M_b \Leftrightarrow N_c - M_c$.

3. Invariant property of multiplication: If $N_b \Leftrightarrow N_c$ and $M_b \Leftrightarrow M_c$ then $N_b * M_b \Leftrightarrow N_c * M_c$

4. Invariant property of division : Assume $M_b \neq 0$, $N_b \Leftrightarrow N_c$, and $M_b \Leftrightarrow M_c$ then

$N_b/M_b \Leftrightarrow N_c/M_c$

Working with finite decimal numbers in the base given can be confusing and difficult. A better way is use the following algorithm:

Step 1: Write $(0.a_1 a_2 \ldots a_n)_b = (a_1 a_2 \ldots a_n)_b/10^n$

Step 2: Write $(0.b_1 b_2 \ldots b_m)_b = (b_1 b_2 \ldots b_m)_b/10^m$

Step 3: $(a_1 a_2 \ldots a_n)_b/10^n \Leftrightarrow N_{10}/b^n$

Step 4: $(b_1 b_2 \ldots b_m)_b/10^m \Leftrightarrow M_{10}/b^m$

Step 5: $N_{10}/b^n \odot M_{10}/n^m$, where $\odot$ is one of the above operations.

Step 6: Convert $N_{10}/b^n \odot M_{10}/n^m$ to the corresponding decimal number in the base b.

Examples:

a. Perform $0.237_8 + 0.33_8$

Step 1: $0.237_8 = (237/1000)_8$

Step 2: $0.33_8 = (33/100)_8$

Step 3: $(237/1000)_8 \Leftrightarrow 159/8^3$

Step 4: $(33/100)_8 \Leftrightarrow 27/8^2$

Step 5: $159/8^3 + 27/8^2 = 159/8^3 + 27*8/8^3 = 375/8^3$

Step 6: $375/8^3 \Leftrightarrow 567_8/1000 = 0.567_8$

Step 7: $0.237_8 + 0.33_8 = 0.567_8$

b. Perform $0.237_8 * 0.33_8$

Step 1: $0.237_8 = (237/1000)_8$

Step 2: $0.33_8 = (33/100)_8$

Step 3: $(237/1000)_8 \Leftrightarrow 159/8^3$

Step 4: $(33/100)_8 \Leftrightarrow 27/8^2$

Step 5: $(159/8^3) * (27/8^2) = 4293/8^5$

Step 6: $4293/8^5 \Leftrightarrow 10305_8/100000 = 0.0.10305_8$

Step 7: $0.237_8 * 0.33_8 = 0.10305_8$

c. Perform $0.237_8 - 0.33_8$

Step 1: $0.237_8 = (237/1000)_8$

Step 2: $0.33_8 = (33/100)_8$

Step 3: $(237/1000)_8 \Leftrightarrow 159/8^3$

Step 4: $(33/100)_8 \Leftrightarrow 27/8^2$

Step 5: $159/8^3 - 27/8^2 = 159/8^3 - 27 * 8/8^3 = - 57/8^3$

Step 6: $- 57/8^3 \Leftrightarrow - 57_8/1000 = - 0.57_8$

Step 7: $0.237_8 - 0.33_8 = - 0.57_8$

d. Perform : $0.237_8 / 0.33_8$

Step 1: $0.237_8 = (237/1000)_8$

Step 2: $0.33_8 = (33/100)_8$

Step 3: $(237/1000)_8 \Leftrightarrow 159/8^3$

Step 4: $(33/100)_8 \Leftrightarrow 27/8^2$

Step 5: $(159/8^3)_{10} / (27/8^2)_{10} = 5.\overline{8}_{10}/8$

Step 6: We now convert $5.\overline{8}_{10}$ to octal:

step 1: $0.\overline{8}_{10} = 0.888888... = a_1/8 + a_2/8^2 + a_n/8^3 + ...$

step 2: $8*(0.8888...) = 7.1111111111 = a_1 + a_2/2 + a_3/2^2 + ...$

step 3: $a_1 = 7$

step 4: $0.111111 ... = a_2/8 + a_3/8^2 + ...$

step 5: $8*(0.11111111...) = ... 0.888888888888 = a_2 + a_3/8 + a_4/8^2 ...$

step 6: $a_2 = 0$

Step 7: Since step 5 returns to the original value in step 1, we are finished and conclude:

$$0.\overline{8}_{10} \Leftrightarrow 0.\overline{70}_8$$

$$5.\overline{8}_{10}/8 \Leftrightarrow (5.\overline{70}_8/10)_8 = 0.5\overline{70}_8$$

Step 8: $0.237_8 / 0.33_8 = 0.5\overline{70}_8$

Exercises:

Perform the following operations:

1. **a.** $0.1011_2 + 0.00111_2$ **b.** $0.1011_2 - 0.00111_2$ **c.** $0.1011_2 * 0.00111_2$

d. 0.1011/ 0.00111

2. **a.** $0.9AB2_{16} + 0.029E_{16}$ **b.** $0.9AB2_{16} - 0.029E_{16}$ **c.** $0.9AB2_{16} * 0.029E_{16}$

e. $0.9AB2_{16} / 0.029E_{16}$

■

PROJECT

Develop the formula: $\overline{a}_b \Leftrightarrow a_{10}\dfrac{b^n}{b^n - 1}$

CHAPTER - 19 SIMPLE ALGORITHMS FOR CONVERTING BETWEEN DECIMAL NUMBER BASES (OPTIONAL)

INTRODUCTION

In this chapter we will show how we write algorithms to convert decimal numbers from one base to another by writing algorithms . At this time we will only write these algorithms for specific types of numbers. To write an algorithm, we first create a sample program from a specific example. Once the program is written, we will use it to create the algorithm.

19.1 An Algorithm to Convert a Positive Finite Decimal Number in any Base b < 10 to its Corresponding Number in the Base 10.

To convert between decimal numbers in any base b to its corresponding number in the base 10, we recall from chapter 2 the following formula:

$N_b = 0.a_1a_2... a_n \Leftrightarrow a_1/b + a_2/b^2 + ... + a_n/b^n = N_{10}$

Example:

$N_4 = 0.321 \Leftrightarrow 3/4 + 2/4^2 + 1/4^3 = 3/4 + 1/8 + 1/64 = 0.75 + 0.125 + 0.015625 = 0.890625_{10}$

Program: Convert the number the number 0.321_4 to the base 10.

INSTRUCTIONS	SUM	A	FRACTION	TEMP	E	BASE
FRACTION := 0.321			**0.321**			
BASE:= 4			0.321			**4**
SUM := 0	**0**		0.321			4
E := 0	0		0.321		**0**	4
TEMP := FRACTION∗10	0		0.321	**3.21**	0	4
E := E + 1	0		0.321	3.21	**1**	4
A := TEMP÷1	0	**3**	0.321	3.21	1	4
SUM := SUM + A/(BASE)^E	**0.75**	3	0.321	3.21	1	4
FRACTION := TEMP - A	0.75	3	**0.21**	3.21	1	4
TEMP := FRACTION∗10	0.75	3	0.21	**2.1**	1	4
E := E + 1	0.75	3	0.21	2.1	**2**	4
A := TEMP÷1	0.75	**2**	0.21	2.1	2	4
SUM := SUM + A/BASE^E	**0.875**	2	0.21	2.1	2	4

FRACTION := TEMP - A	0.875	2	**0.1**	2.1	2	4
TEMP := FRACTION $*$10	0.875	2	0.1	**1**	2	4
E := E + 1	0.875	2	0.1	1	**3**	4
A := TEMP÷1	0.875	**1**	0.1	1	3	4
SUM := SUM + A/BASE^E	**0.8906**	1	0.1	1	3	4

$0.321_4 \Leftrightarrow 0.890625$

Algorithm:

INSTRUCTIONS
SUM := 0
E:= 0
TEMP := FRACTION $*$10
E := E + 1
A := TEMP÷1
SUM := SUM + A/(BASE)^E
FRACTION := TEMP - A
::::::::::::::::

Exercises:

1. Using this algorithm, write a program to convert the following numbers to the base 10.

a. 0.7777_8 **b.** 0.1101_2
■

19.2 An Algorithm to Convert any Decimal Number in the Base 10 to a Corresponding Number in the Base b < 10.

In Chapter 3 we saw to convert a decimal number in the base 10 to its corresponding number in a given base b we use the following formula:

$N_{10} = a_1/b + a_2/b^2 + ... + a_n/b^n \Leftrightarrow (0.a_1a_2... a_n)_b$

The following example will demonstrate how to solve the values a_k:

Convert 0.8 to the base 2.

Step 1: $0.8 = a_1/2 + a_2/2^2 + a_n/2^3 + ...$

Step 2: $2*(0.8) = 1.6 = a_1 + a_2/2 + a_3/2^2 + ...$

Step 3: Since a_1 is an integer, $a_1 = 1$ and $0.6 = a_2/2 + a_3/2^2 + ...$

Step 4: $2*(0.6) = 1.2 = a_2 + a_3/2 + ...$

Step 5: $a_2 = 1$ and $0.2= a_3/2 + a_4/2^2 + ...$

Step 6: $2*(0.2) = 0.4 = a_3 + a_4/2 + ..$

Step 7: $a_3 = 0$ and $0.4= a_4/2 + a_5/2^2 + ...$

Step 8: $2*(0.4) = 0.8 = a_4 + a_5/2 + ..$

Step 9: $a_4 = 0$ and $0.8 = a_5/2 + a_6/2^2 + ...$

Step 10: $2*(.8) = 1.6 = a_5 + a_6/2 + ..$

Therefore, $0.8 \Leftrightarrow 0.\overline{1100}_2$

Example:

Program: Convert to the base 2 the decimal number 0.8 to 4 places.

INSTRUCTIONS	N	SUM	DIGIT	TEMP	E	BASE
N := 0.8	**0.8**					
BASE := 2	0.8					**2**
SUM := 0	0.8	**0**				2
E := 1	0.8	0				2
TEMP := N∗BASE	0.8	0		**1.6**	**0**	2
DIGIT := TEMP÷1	0.8	0	**1**	1.6	0	2
SUM := SUM + DIGIT/(10^E)	0.8	**0.1**	1	1.6	1	2
N := TEMP - DIGIT	**0.6**	0.1	1	1.6	1	2
E := E + 1	0.6	0.1	1	`1.6	**2**	2
TEMP := N∗BASE	0.6	0.1	1	**1.2**	2	2
DIGIT := TEMP÷1	0.6	0.1	**1**	1.2	2	2
SUM := SUM + DIGIT/(10^E)	0.6	**0.11**	1	1.2	2	2
N := TEMP - DIGIT	**0.2**	0.11	1	1.2	2	2

E := E + 1	0.2	0.11	1	1.2	**3**	2
TEMP := N∗BASE	0.2	0.11	1	**0.4**	3	2
DIGIT := TEMP÷1	0.2	0.11	**0**	0.4	3	2
SUM := SUM + DIGIT/(10^E)	0.2	**0.110**	0	0.4	3	2
N := TEMP - DIGIT	**0.4**	0.110	0	0.4	3	2
E := E + 1	0.4	0.110	0	0.4	**4**	2
TEMP := N∗BASE	0.4	0.110	0	**0.8**	4	2
DIGIT := TEMP÷1	0.4	0.110	**0**	0.8	4	2
SUM := SUM + DIGIT/(10^E)	0.4	**0.1100**	0	0.8	4	2
N := TEMP - DIGIT	**0.8**	0.1100	0	0.8	4	2

$0.8 \Leftrightarrow 0.\overline{1100}_2$

Algorithm:

INSTRUCTIONS
SUM := 0
E := 1
TEMP := N∗BASE
DIGIT := TEMP÷1
SUM := SUM + DIGIT/(10^E)
N := TEMP - DIGIT
E := E + 1
:::::::::::::::::::

Exercise:

1. Write a program that convert, to 4 places, the number 0.9 to the base 8.
■

PROJECT

1. Using iterative addition, write an algorithm that will convert to 4 places any decimal number in base b $(0.a_1a_2a_3a_4)_b$ to its corresponding value $(d_1d_2d_3d_4)_c$ where $c \neq b$ and c, b < 10.

2. Using this algorithm, write a program that will convert to 4 places 0.2546_8 to $(d_1d_2d_3d_4)_2$.

CHAPTER 20 - WORKING WITH DECIMAL NUMBERS IN ASSEMBLY

20.1: Representation of Decimal Numbers

So far in assembly language, we have only worked with integer numbers. We will now study how we can represent and work with fractions represented as numbers with a decimal point. These numbers will be called decimal numbers. When such numbers are used in assembly language programming they are frequently represented as ordinary decimal numbers or scientific notation.

Definition: Ordinary decimal numbers

An ordinary decimal number is of the form $\pm\, a_0.a_1a_2 \ldots\; a_n$,

where a_k are non negative integers

Examples:

23.4, -55.0101, 0.00154 9.0

Definition: Scientific Representation of Decimal Numbers:

The representation of a decimal number in a scientific format is of the form $\pm\; n * 10^{k}$.

where n is an integer, $*$ represents the multiplication operation and k is always a non-positive integer. The value k is called the exponent and the integer n is called the mantissa.

Definition: Floating Point Representation of Decimal Numbers:

In assembly language, decimal numbers represented in the form

$\pm\, a_0.a_1a_2 \ldots\; a_n .\times E \pm n$

are called floating point numbers

where a_0 is a positive digit.

Examples:

ORDINARY DECIMAL NUMBER REPRESENTATION	SCIENTIFIC REPRESENTATION	FLOATING POINT REPRESENTATION
23.4	$234 * 10^{-1}$	2.34 E1
-55.0101	$-550101 * 10^{-4}$	-5.50101E 1
0.00154	$154 * 10^{-5}$	1.54 E -3
- 79.0	$-79 * 10^{0}$	-7.9 E -1
9.0	$9 * 10^{0}$	9 E 0

Exercises

Write the following in scientific and floating point representation:

0.00234 45.356 - 32

■

20.2: Arithmetic Operations Using Scientific Representation.

Multiplication

To multiply two numbers in scientific notation, we simply multiply the integer numbers and add the exponents :

$(N\ E\ n_1)\ (M\ E\ n_2) = N*M\ E\ (n_1 + n_2)$

Examples:

$(0.234)(0.05667) = (234*10^{-3})(5667*10^{-5}) = (234)(5667)*10^{-8} = 1326078*10^{-8} = 1326078\ E\ -8$

The following partial assembly language code will compute (0.234)(0.05667):

```
mov eax, 234
mov ebx, -3
mul 5667
add ebx, -5
```

Exercises:

1. Write the following using scientific representation.

$-\ 575.345*0.00234$ $678*0.03*2.135$ $0.0034*0.221$

2. Write assembly language codes that will compute the above.

■

Addition and Subtraction

To add or subtract two numbers using scientific representation, the exponents must be equal:

$N\ E\ n \pm M\ E\ n = (N \pm M)\ E\ n$

Example:

$0.234 + 0.05667 = 234*10^{-3} + 5667*10^{-5} = 23400*10^{-5} + 5667*10^{-5} = (23400+ 5667)*10^{-5} =$

$29067 * 10^{-5} = 29067$ E -5

The following assembly language code will compute 0.234 + 0.05667:

```
mov eax, 23400
mov ebx, -5
add eax, 5667
```

Exercises:

Write the following using scientific representation :

1. - 575.345 + 0.00234   678 + 0.03 + 2.135   0.0034 - 0.221

2. Write assembly language codes that will compute the above.

■

Long Division

To divide two decimal numbers N/M using scientific representation, we have the following form :

$(N * 10^n)/M * 10^m = (N/M) * 10^{n-m} = (N/M)$ E $(n - m)$

Example:

$1/0.6 = 1/(6 * 10^{-1}) = 10/6 = (1/6) * 10^1 \approx (0.16666) * 10^1 = (16666 * 10^{-5}) * 10^1 = 16666 * 10^{-4}$

Since N/M is not always an integer, we need to use the long division algorithm to convert N/M to a decimal value. We first show how to develop an algorithm to convert 1/N to a decimal value.

We define the decimal representation of 1/N as

$1/N = a_1/10 + a_2/10^2 + a_3/10^3 + ... = 0.a_1a_2a_3 .... \approx a_1/10 + a_2/10^2 + ... a_3/10^n$

where $N \neq 0$.

Example:

Convert 5/6 to a 5 place decimal representation.

$1/6 = a_1/10 + a_2/10^2 + a_3/10^3 + a_4/10^4 + a_5/10^5 + (a_6/10^6 + a_7/10^7 + ...)$

Step 1: $10(1/6) = 10/6 = 1 + 4/6 = a_1 + a_2/10 + a_3/10^2 + a_4/10^3 + a_5/10^4 + (a_6/10^5 + a_7/10^6 + ...)$

$a_1 = 1$

$4/6 = a_2/10 + a_3/10^2 + a_4/10^3 + a_5/10^4 + (a_6/10^5 + a_7/10^6 + ...)$

Step 2: $10(4/6) = 40/6 = (36 + 4)/ 6 = 6 + 4/6 = a_2 + a_3/10 + a_4/10^2 + a_5/10^3 + (a_6/10^4 + a_7/10^5 + ...)$

$a_2 = 6$

$4/6 = a_3/10^1 + a_4/10^2 + a_5/10^3 + (a_6/10^4 + a_7/10^5 + ...)$

Step 3: $10(4/6) = 40/6 = (36 + 4)/ 6 = 6 + 4/6 = a_3 + a_4/10 + a_5/10^2 + (a_6/10^3 + a_7/10^4 + ...)$

$a_3 = 6$

$4/6 = a_4/10 + a_5/10^2 + (a_6/10^3 + a_7/10^4 + ...)$

Step 4: $10(4/6) = 40/6 = (36 + 4)/ 6 = 6 + 4/6 = a_4 + a_5/10 + (a_6/10^2 + a_7/10^3 + ...)$

$a_4 = 6$

$4/6 = a_5/10 + (a_6/10 + a_7/10^2 + ...)$

Step 5: $10(4/6) = 40/6 = (36 + 4)/ 6 = 6 + 4/6 = a_5 + (a_6 + a_7/10 + ...)$
$a_5 = 6$

Therefore,

$1/6 = 1/10 + 6/10^2 + 6/10^3 + 6/10^4 + 6/10^5 + (a_6/10^6 + a_7/10^7 + ...) \approx 1/10 + 6/10^2 + 6/10^3 + 6/10^4 + 6/10^5 =$

$0.1 + 0.06 + 0.006 + 0.0006 + 0.00006 = 0.16666$.

Now,

$5/6 = 5*(1/6) \approx 5*(0.16666) = (5*10^0) * 16666*10^{-5} = 83330*10^{-5} = 83330E -5$

Example:

The following pseudo-language program that will compute $1/6 \approx 16666*10^{-5}$.

PSEUDO-CODES	N	A	SUM	E	MUL	R
E:= 0				**0**		
SUM:= 0			**0**	0		
MUL:= 10000			0	0	**10000**	
N:= 1	**1**		0	0	10000	

N:= N∗10	**10**		0	0	10000	
A:= N÷6	10	**1**	0	0	10000	
R:= N MOD 6	10	1	0	0	10000	**4**
SUM:= SUM + A∗MUL	10	1	**10000**	0	10000	4
E:= E - 1	10	1	10000	**-1**	1000	4
MUL:= MUL÷10	10	1	10000	-1	**1000**	4
N:= R	**4**	1	10000	-1	1000	4
N:= N∗10	**40**	**1**	10000	-1	1000	4
A:= N÷6	40	**6**	10000	-1	1000	4
R:= N MOD 6	40	6	10000	-1	1000	**4**
SUM:= SUM + A∗MUL	40	6	**16000**	-1	1000	4
E:= E -1	40	6	16000	**-2**	1000	4
MUL:= MUL÷10	40	6	16000	-2	**100**	4
N:= R	**4**	6	16000	-2	100	4
N:= N∗10	**40**	6	16000	-2	100	4
A:= N÷6	40	**6**	16000	-2	100	4
R:= N MOD 6	40	6	16000	-2	100	**4**
SUM:= SUM + A∗MUL	40	6	**16600**	-2	100	4
E:= E - 1	40	6	16600	**-3**	100	4
MUL:= MUL÷10	40	6	16600	-3	**10**	4
N:= R	**4**	6	16600	-3	10	4
N:= N∗10	**40**	6	16600	-3	10	4
A:= N÷6	40	**6**	16600	-3	10	4
R:= N MOD 6	40	6	16600	-3	10	**4**
SUM:= SUM + A∗MUL	40	6	**16660**	-3	10	4
E:= E - 1	40	6	16660	**-4**	10	4
MUL:= MUL÷10	40	6	16660	-4	**1**	4
N:= R	**4**	6	16660	-4	1	4
N:= N∗10	**40**	6	16660	-4	1	4
A:= N÷6	40	**6**	16660	-4	1	4

R:= N MOD 6	4	6	16660	-4	1	**4**
SUM:= SUM + A ∗MUL	4	6	**16666**	-4	1	4
E:= E - 1	4	6	16666	**-5**	1	4
MUL:= MUL÷10	4	6	16666	-5	**0**	4
N:= R	**4**	6	16666	-5	0	4

Exercises:

1. Rewrite the above program in pseudo-code using a while statement. From this program write an assembly language.

2. Using the above algorithm, convert 1/7 to a 7 place decimal representation.

3. Write the following in a scientific notation form.

a. 5/7 b. 0.23/0.035

4. Convert 1/3 to binary.

■

20.3: 80X86 Floating-Point Architecture

The MASM compiler has the ability to handle ordinary and floating **-** point decimal numbers.
The following are definitions of the representation given by MASM for decimal numbers:

Definition float: An ordinary decimal representation. The number is represented as a 32 bit number.

Definition double-decimal: An ordinary decimal representation. The number is represented as a 64 bit number.

Definition long-double: floating point representation. The number is represented as a 80 bit number.

The following are data - type registers that are available : *TBYTE, REAL4, REAL8, REAL10.* The table below gives the specifications for each of these data-types:

DIRECTIVE	**# OF BYTES**	**Number type**
REAL4	4	float - decimal
REAL8	8	double - decimal
REAL10	10	long double - floating-point
QWORD	8	integer
TBYTE	10	long double - floating-point

Along with these data-types, we still can use the integer data-types: *BYTE, WORD, DWORD* .

Important: Except the QWORD data type, all the above data types are only represented in the base 10. The QWORD follows the data type representation for integer numbers.

Examples:

.DATA

w TBYTE 0.236; will assign the number 2.36 to the identifier w as 2.36E-1.

x real4 2.34; will assign the number 2.34 to the identifier x as 2.34 .

y real8 0.00678; will assign 0.00678 to the identifier y as 0.00678

z real10 23554.5678 will assign 23554.5678 to the identifier z as 2.35545678E 4

q qword 10 will assign 10 to the identifier q as a .

Rules for Assigning floating point numbers.

The following rules for assigning floating point numbers:

• All identifiers are initially assigned floating point numbers, where they are defined in the data part of the program.

• All other assignments are done by passing the contents of the variables to the various floating-point registers.

Floating-point registers

The registers EAX, EBX, ECX, EDX can not be used directly when working with floating-points numbers. Instead, we have eight data registers, each 80 bits long. Their names are ST or ST(0) , ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7). These eight registers are shown stacked vertically top down and should be visualized as following:

ST
ST(1)
ST(2)
ST(3)
ST(4)
ST(5)

ST(6)
ST(7)

Exercise:

1. What is the largest value (base 10) that can be stored in ST(k)?
■

The operands of all floating-point instructions begin with the letter f. The following will give the most important floating-point instructions according to their general functions. Additional floating-point instructions will be discussed in a later chapter of this book.

Storing data from memory to the registers

For demonstration purposes, we will assume the registers have the numbers:

ST	10.0
ST(1)	15.0
ST(2)	20.0
ST(3)	25.0
ST(4)	
ST(5)	
ST(6)	
ST(7)	

The following are the floating-point instructions that will store data from memory to a given register.

- **fld**

MNEMONIC	OPERAND	ACTION
fld	memory variable (real)	The real number from memory is stored in ST and data is pushed down.

Example:

.DATA

x REAL4 30.0

fld x; stores the content of x into register ST and pushes the other values down.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**30.0**
ST(1)	15.0	10.0
ST(2)	20.0	15.0
ST(3)	25.0	20.0
ST(4)		25.0
ST(5)		
ST(6)		
ST(7)		

- **fild**

MNEMONIC	OPERAND	ACTION
fild	variable memory (integer)	The integer number from memory is stored in ST, converted to floating- point and data is pushed down.

Example:

```
.DATA
x    DWORD   50
```

fild x;   stores the content of x (integer value) into register ST and pushes the other values down.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**50.0**
ST(1)	15.0	10.0
ST(2)	20.0	15.0
ST(3)	25.0	20.0
ST(4)		25.0
ST(5)		
ST(6)		
ST(7)		

- **fld**

MNEMONIC	OPERAND	ACTION
fld	st(k)	The number in st(k) is stored in ST and data is pushed down.

Example:

fld st(2) ; stores the contents of register st(2) into register ST and pushes the other values down.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**20.0**
ST(1)	15.0	10.0
ST(2)	20.0	15.0
ST(3)	25.0	20.0
ST(4)		25.0
ST(5)		
ST(6)		
ST(7)		

Important: Once the stack is full, additional stored data will cause the bottom values to be lost. Also the *finit* instruction will clear all the values in the register.

Copying data from the stack

We will assume the registers have the numbers:

ST	10.0
ST(1)	15.0
ST(2)	20.0
ST(3)	25.0
ST(4)	
ST(5)	
ST(6)	
ST(7)	

The following are the floating-point instructions that will copy data from stack.

- **fst**

MNEMONIC	OPERAND	ACTION
fst	st(k)	Makes a copy of ST and stores the value in ST(k).

Example:

fst ST(2) ; stores the content of ST into ST(2)

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	10.0
ST(1)	15.0	15.0
ST(2)	20.0	**10.0**
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

- **fst**

MNEMONIC	OPERAND	ACTION
fst	memory variable (real)	Makes a copy of ST and stores the value in a real memory location

Example:

```
.DATA
x     real4     ?
```

fst x ; stores the content of ST into x. The stack is not affected.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	10.0
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

- **fist**

MNEMONIC	OPERAND	ACTION
fist	memory variable (integer)	Converts to integer a copy of ST and stores the rounded value in a integer memory location .

Example:

.DATA
x DWORD ?

fist x ; stores the content of ST as an integer number into x.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	10.0
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Exchanging the contents of the two floating-point registers.

We will assume the registers have the numbers

ST	10.0
ST(1)	15.0
ST(2)	20.0
ST(3)	25.0
ST(4)	
ST(5)	
ST(6)	
ST(7)	

The following are the floating-point instructions that will exchange the contents of two floating-point registers.

• fxch

MNEMONIC	OPERAND	ACTION
fxch	none	Exchanges the content of ST and ST(1).

Example:

fxch ; exchanges the content of ST and ST(1).

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**15.0**
ST(1)	15.0	**10.0**
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

- **fxch**

MNEMONIC	OPERAND	ACTION
fxch	st(k)	Exchanges the content of ST and ST(k).

Example:

fxch st(3) ; exchanges the content of ST and ST(3).

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**25.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	**10.0**
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Adding contents of the two floating-point registers.

We will assume the registers have the numbers

ST	10.0
ST(1)	15.0
ST(2)	20.0
ST(3)	25.0
ST(4)	
ST(5)	
ST(6)	
ST(7)	

The following are the floating-point instructions that will add the contents of two floating-point registers.

• **fadd**

MNEMONIC	OPERAND	ACTION
fadd	st(k), st	Adds ST(k) and ST; then ST(k) is replaced by the sum.

Example:

fadd st(3), st ; adds ST(3) and ST; then ST(3) is replaced by the sum.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	10.0
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	**35.0**
ST(4)		
ST(5)		
ST(6)		
ST(7)		

• **fadd**

MNEMONIC	OPERAND	ACTION
fadd	st, st(k)	Adds ST and ST(k); then ST is replaced by the sum.

Example:

fadd st, st(3) ; adds the content of ST and ST(3) then ST is replaced by the sum.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**35.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0

ST(4)		
ST(5)		
ST(6)		
ST(7)		

• fadd

MNEMONIC	OPERAND	ACTION
fadd	memory variable (real)	Adds ST and the contents of a real variable; then ST is replaced by the sum.

Example:

x REAL4 12.0

fadd x ; adds the content of ST and x; then ST is replaced by the sum.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**22.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

• fiadd

MNEMONIC	OPERAND	ACTION
fiadd	memory variable (integer)	Adds ST and the contents of a integer variable; then ST is replaced by the sum.

Example:

x DWORD 70

fadd x ; adds the content of ST and x then ST is replaced by the sum.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**80.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Subtracting the contents of the two floating-point registers.

The following are the floating-point instructions that will subtract the contents of two floating-point registers.

- **fsub**
- **fsbur**

MNEMONIC	OPERAND	ACTION
fsub	st(k), st	Computes ST(k) - ST; then ST(k) is replaced by the difference.
fsbur	st(k), st	Computes ST - ST(k); then ST(k) is replaced by the difference.

Example:

fsub st(3), st ; computes ST(3) - ST; then ST(3) is replaced by the difference.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	10.0
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	**15.0**
ST(4)		

ST(5)		
ST(6)		
ST(7)		

- **fsub**
- **fsubr**

MNEMONIC	OPERAND	ACTION
fsub	st, st(k)	Computes ST - ST(k) ; then ST is replaced by the difference.
fsubr	st, st(k)	Computes ST(k) - ST; then ST is replaced by the difference

Example:

fsub st , st(1) ; computes st - st(1) ; then st is replaced by the difference.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**- 5.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

- **fsub**
- **fsubr**

MNEMONIC	OPERAND	ACTION
fsub	memory (real)	Calculates ST - real number ; then ST is replaced by the difference.
fsubr	memory (real)	Calculates real number - ST; then ST is replaced by the difference.

Example:

x REAL4   12.0

fsub   x ;   calculates st - x   ; then st is replaced by the difference.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**- 2.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

- **fisub**
- **fisubr**

MNEMONIC	OPERAND	ACTION
fisub	memory (integer)	Calculates ST - integer number; then ST is replaced by the difference
fisubr	memory (integer)	Calculates integer number - ST; then ST is replaced by the difference

Example:

x DWORD   70

fisub   x ; calculates st - x; then st is replaced by the difference

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**- 60.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0

ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Multiplying the contents of the two floating-point registers.

The following are the floating-point instructions that will multiply the contents of two floating-point registers.

• fmul

MNEMONIC	OPERAND	ACTION
fmul	st(k), st	Multiplies ST(k) and ST; then ST(k) is replaced by the product.

Example:

fmul st(3), st ; multiplies st(3) and st; then st(3) is replaced by the product.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	10.0
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	**250.0**
ST(4)		
ST(5)		
ST(6)		
ST(7)		

• fmul

MNEMONIC	OPERAND	ACTION
fmul	st, st(k)	Multiplies ST(k) and ST; then ST is replaced by the product.

Example:

fmul st, st(3) ; multiplies st(3) and st; then st is replaced by the product.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**250.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

• fmul

MNEMONIC	OPERAND	ACTION
fmul	memory variable (real)	Multiplies ST and real variable ; then ST is replaced by the product.

Example:

x REAL4 35.0

fmul x ; multiplies x and st; then st is replaced by the product.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**350.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		

ST(7)		

• **fmul**

MNEMONIC	OPERAND	ACTION
fmul	memory variable (integer)	Multiplies integer variable and ST; then ST is replaced by the product.

Example:

x DWORD 45

fmul x ; multiplies x and st; then st is replaced by the product.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**450.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Dividing the contents of floating-point registers.

The following are the floating-point instructions that will divide the contents of floating-point registers.

• **fdiv**
• **fdivr**

MNEMONIC	OPERAND	ACTION
fdiv	st(k), st	Computes ST(k)/ ST; then ST(k) is replaced by the quotient.

fdivr	st(k), st	Computes ST/ ST(k); then ST(k) is replaced by the quotient.

Example:

fdiv st(1), st ; computes st(1)/st; then st(1) is replaced by the quotient.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	5.0	5.0
ST(1)	15.0	**3.0**
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

- **fdiv**
- **fdivr**

MNEMONIC	OPERAND	ACTION
fdiv	st, st(k)	Computes ST/ ST(k) ; then ST is replaced by the quotient.
fdivr	st, st(k)	Computes ST(k)/ ST; then ST is replaced by the quotient.

Example:

fdiv st, st(2) ; computes st/ st(2) ; then st is replaced by the quotient.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	5.0	**0.25**
ST(1)	15.0	15.0
ST(2)	20.0	20.0

ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

- **fdiv**
- **fdivr**

MNEMONIC	OPERAND	ACTION
fdiv	memory variable (real) e	Computes ST/ real variable ; then ST is replaced by the quotient.
fdivr	memory variable (real)	Computes real variable/ST ; then ST is replaced by the quotient.

Example:

x real4 10.0

fdiv x ; computes st/ x ; then st is replaced by the quotient.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	5.0	**0.5**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

- **fidv**
- **fidvr**

MNEMONIC	OPERAND	ACTION

fidiv	memory (integer)	Computes ST/ integer variable ; then ST is replaced by the quotient.
fidivr	memory (integer)	Computes integer variable /ST ; then ST is replaced by the quotient.

Example:

x DWORD 5

fdiv x; computes st/ x ; then st is replaced by the quotient.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	5.0	**1.0**
ST(1)	15.0	15.0
ST(2)	20.0	20.0
ST(3)	25.0	25.0
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Summary Tables of Floating Point Arithmetic Operations

Store data from memory to a given register

MNEMONIC	OPERAND	ACTION
fld	variable memory (real)	The real number from memory is stored in ST and data is pushed down.
fild	variable memory (integer)	The integer number from memory is stored in ST, converted to floating- point and data is pushed down.
fld	st(k)	The number in st(k) is stored in ST and data is pushed down.

Copying data from the stack

MNEMONIC	OPERAND	ACTION
fst	st(k)	makes a copy of ST and stores the value in ST(k).

fst	memory variable (real)	makes a copy of ST and stores the value in a real memory location.
fist	memory (integer)	Converts to integer a copy of ST and stores the rounded value in a integer memory location.

Exchanging the contents of the two floating-point registers

MNEMONIC	OPERAND	ACTION
fxch	(none)	Exchanges the content of ST and ST(1).
fxch	st(k)	Exchanges the content of ST and ST(k).

Adding contents of the two floating-point registers

MNEMONIC	OPERAND	ACTION
fadd	st(k), st	Adds ST(k) and ST; then ST(k) is replaced by the sum.
fadd	st, st(k)	Adds ST and ST(k); then ST is replaced by the sum.
fadd	memory variable (real)	Adds ST and the contents of a real variable; then ST is replaced by the sum.
fiadd	memory variable (integer)	Adds ST and the contents of a integer variable; then ST is replaced by the sum.

Subtracting the contents of the two floating-point registers.

MNEMONIC	OPERAND	ACTION
fisub	memory (integer)	Calculates ST - integer number; then ST is replaced by the difference
fisubr	memory (integer)	Calculates integer number - ST; then ST is replaced by the difference
fsbur	st(k), st	Computes ST - ST(k); then ST(k) is replaced by the difference.
fsub	memory (real)	Calculates ST - real number ; then ST is replaced by the difference.
fsub	st, st(k)	Computes ST - ST(k) ; then ST is replaced by the difference.

fsub	st(k), st	Computes ST(k) - ST; then ST(k) is replaced by the difference.
fsubr	st, st(k)	Computes ST(k) - ST; then ST is replaced by the difference
fsubr	memory (real)	Calculates real number - ST; then ST is replaced by the difference.

Multiplying the contents of the two floating-point registers

MNEMONIC	OPERAND	ACTION
fmul	st, st(k)	Multiplies ST(k) and ST; then ST is replaced by the product.
fmul	st(k), st	Multiplies ST(k) and ST; then ST(k) is replaced by the product.
fmul	memory variable (real)	Multiplies ST and real variable ; then ST is replaced by the product.
fmul	memory variable (integer)	Multiplies integer variable and ST; then ST is replaced by the product.

Dividing the contents of floating-point registers

MNEMONIC	OPERAND	ACTION
fdiv	st(k), st	Computes ST(k) / ST; then ST(k) is replaced by the quotient.
fdiv	st, st(k)	Computes ST / ST(k) ; then ST is replaced by the quotient.
fdiv	memory variable (real)	Computes ST / real variable ; then ST is replaced by the quotient.
fdivr	st(k), st	Computes ST / ST(k); then ST(k) is replaced by the quotient.
fdivr	st, st(k)	Computes ST(k) / ST; then ST is replaced by the quotient.
fdivr	memory variable (real)	Computes real variable / ST; then ST is replaced by the quotient.
fidiv	memory variable (integer)	Computes ST / integer variable ; then ST is replaced by the quotient.

fidivr	memory variable (integer)	Computes integer variable / ST; then ST is replaced by the quotient.

Miscellaneous floating point instructions

1.

MNEMONIC	OPERAND	ACTION
fabs	(none)	replaces the contents of ST with it absolute value.

Example:

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	- 10.0	**10.0**

2.

MNEMONIC	OPERAND	ACTION
fchs	(none)	replaces the contents of ST with - ST

Example:

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**- 10.0**

3.

MNEMONIC	OPERAND	ACTION
frndint	(none)	rounds ST to an integer value

Example:

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	12.754	**12.0**

Example:

A harmonic sum is defined by the sum

1 + 1/2 + 1/3 + ... + ... + 1/n

The following pseudo-code programs will compute

1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6.

PSEUDO CODE	CYCLE OF INSTRUCTIONS	SUM	N	ONE
SUM := 0.0	SUM:= 0.0	**0.0**	1	
N := 1	N := 1	0.0	**1**	
ONE:= 1	ONE:= 1	**0.0**	1	**1**
WHILE N <= 6	WHILE N <= 6	0.0	1	1
BEGIN	**BEGIN**	0.0	1	1
SUM := SUM + 1/N	SUM:= SUM+1/ N	**1**	1	1
N := N + 1	N := N + 1	1	**2**	1
	SUM:= SUM+1/ N	**1.5**	2	1
	N := N + 1	1.5	**3**	1
	SUM:= SUM+ 1/N	**1.8333...33333**	3	1
	N := N + 1	1.833...33333	**4**	1
	SUM:= SUM+ 1/N	**2.0833....33333**	4	1
	N := N + 1	2.0833....33333	**5**	1
	SUM:= SUM+ 1/N	**2.2833...33333**	5	1
	N := N + 1	2.2833...33333	**6**	1
	SUM:= SUM+ 1/N	**2.45**	6	1
	N := N + 1	2.45	**7**	1
END	**END**	2.45	7	1

PSEUDO CODE	AL PSEUDO CODE	ASSEMBLY CODE
SUM := 0.0	SUM := 0.0	sum real4 0.0
N := 1	N := 1	n byte 1
ONE:= 1	0NE:= 1	one byte 1
WHILE N <= 6	WHILE N ≤ 6	while: cmp n, 6 jg end
SUM := SUM + 1/N N:= N + 1	ST := ONE	fld one
	ST:= ST/N	fidiv n
	ST:= SUM + ST	fadd sum

	SUM:= ST	fst sum
	EAX := N	mov eax, n
	EAX := EAX + 1	add eax, 1
	N:= EAX	mov n, eax
END	**END**	jmp while
		end:

Exercises:

1. Modify the above three tables to compute the sum:

$1^2 + 1/2^2 + 1/3^2 + 1/4^2 + 1/5^2 + 1/6^2$.
■

For problems 2 - 5, assume ST(k) (k = 0 , 1, ..., 7) contain pre-assigned values. Write assembly language programs that will perform the following tasks:

2. Task: Compute and store the value ST(0) + TS(1) + ... + TS(7).

3. Task: Compute and store the value $ST(0)^2 + 2ST(1)^2 + 3ST(3)^2 + 4ST(4)^2 + 5ST(5)^2 + 6ST(6)^2 + 7ST(7)^2$

4. Task: Find and store the largest value.

5. Task: Find and store the smallest value.

6. Write an algorithm that will compute and store the number: 1 + 2 + ... + N .

7. Write an algorithm that will compute and store the $1 + 2^2 + ... + N^2$.

8. The determinate of a square table plays a major rule in mathematics. The following is a definition of a 2 by 2 determinate:

$$\Delta = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

Write an algorithm that will compute an arbitrary 2 by 2 determinate.

Cramer's Rule

Assume we wish to solve the following 2 by 2 system of equations:

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

The following Cramer's Rule's give us a solution of the above system of equations:

$$x = \frac{\begin{vmatrix} b_1 & a_{12} \\ b_2 & a_{22} \end{vmatrix}}{\Delta}$$

$$y = \frac{\begin{vmatrix} a_{11} & b_1 \\ a_{21} & b_2 \end{vmatrix}}{\Delta}$$

8. Write a algorithm that will compute an arbitrary 2 by 2 system of equations. Make certain that $\Delta \neq 0$.

A 3 by 3 determinate is defined as

$$\Delta = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

9. Write a algorithm that will compute an arbitrary 3 by 3 determinate.
■

Interchanging integer and floating point numbers.

The following table will demonstrate how integer numbers and floating pont numbers are interchanged (all numbers are decimal).

AS CODE	N	X	Y	Z	ST(0)
n dword ?					
x real4 2.0		**2.0**			
y real4 23.7		2.0	**23.7**		
z real4 55.4		2.0	23.7	**55.4**	
fld x		2.0	23.7	55.4	**2.0**
fist n	**2**	2.0	23.7	55.4	2.0
fld y	2	2.0	23.7	55.4	**23.7**
fist n	**24**	2.0	23.7	55.4	23.7
fld z	24	20	23.7	55.4	**55.4**
fist n	**55**	20	23.7	55.4	55.4

Model Program

```
; This program will compute the harmonic sum

;1 + 1/2 + 1/3 +  1/4 + 1/5 + 1/6

.386

.model flat

.stack 4096

.data

sum real4  0.0
n byte  1
one byte 1

.code

_start:

;start assembly language code

while: cmp n, 6

jg end

fild one

fidiv n

fadd sum

fst sum

mov eax, n

add eax, 1

mov n, eax

jmp while

end:



;end of assembly language code

public _start

end
```

Project:

The solution of a 3 by 3 system of equations

$$a_{11}x + a_{12}y + a_{13}z = b_1$$

$$a_{21}x + a_{22}y + a_{23}z = b_2$$

$$a_{31}x + a_{32}y + a_{33}z = b_3$$

$$x = \frac{\begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix}}{\Delta}$$

$$y = \frac{\begin{vmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{vmatrix}}{\Delta}$$

$$z = \frac{\begin{vmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{32} & b_3 \end{vmatrix}}{\Delta}$$

Write an algorithm that solves any 3 by 3 system of equations. Make certain to check that $\Delta \neq 0$.

CHAPTER 21 COMPARING AND ROUNDING FLOATING - POINT NUMBERS

21.1: Instructions that Compare Floating-Point Numbers

When we are comparing floating-point numbers, we cannot use the instruction *cmp*. Instead we have the following instructions that allow us to compare the register ST to a second operand:

MNEMONIC	OPERAND	ACTION
fcom	(none)	compares ST and ST(1)
fcom	st(k)	compares ST and ST(k)
fcom	variable memory (real)	compares ST and a real number in memory
ficom	variable memory (integer)	compares ST and a integer number in memory
ftst	(none)	compares ST and 0.0

The status word register

When one of the comparison instructions is made , the contents of a special 16-bit register, called the *status word* register is modified. The comparison instruction will assign bits (0 or 1) to the bits 9, 11, 15 of the status word.

The status word register cannot be directly accessed. In order the evaluate the bits in the status word, we can with the following two instructions, copy the contents of the status word to a memory variable or the AX register:

MNEMONIC	OPERAND	ACTION
fstsw	variable (word) memory (integer)	copies status register into memory
fstsw	AX	copies status register into AX

Examples:

x dword ?

fstsw x

fstsw ax

Interpretation of the contents of the status word

When a comparison is made, the table below give the bit values that are assigned to the status word by the comparison instructions:

COMPARISON	STATUS WORD															
BIT POSITION	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
ST > second operand	x	**0**	x	x	x	**0**	x	**0**	x	x	x	x	x	x	x	x

ST < second operand	x	**0**	x	x	x	**0**	x	**1**	x	x	x	x	x	x	x	x
ST = second operand	x	**1**	x	x	x	**0**	x	**0**	x	x	x	x	x	x	x	x

where the values x are either 0 or 1.

Since we are not sure what the other bits are in the status word, we need to create a mask that will convert the bits represented above by xs' to the bit 0. By doing this we can make correct comparisons. The following mask will be used:

BIT POSITION	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
MASK (binary)	0	**1**	0	0	0	**1**	0	**1**	0	0	0	0	0	0	0	0
MASK (hex)	**4500h**															

The following codes will show the effect of the mask on the possible contents of the status word resulting from a comparison instruction:

ST > second operand

AL CODE	AX	MASK
mov mask, 0100010100000000 b		0100010100000000
fstsw ax ; stores the contents of the status word into ax.	x0xxx0x0xxxxxxxx	0100010100000000
and ax, mask	0000000000000000	0100010100000000

or

AL CODE	AX	MASK
mov mask, 4500h		4500h
fstsw ax ; stores the contents of the status word into ax.	x0xxx0x0xxxxxxxx	4500h
and ax, mask	0h	4500h

ST < second operand

AL CODE	AX	MASK
mov mask, 010001010000000 b		0100010100000000
fstsw ax ; stores the contents of the status word into ax.	x0xxx0x1xxxxxxxx	0100010100000000
and ax, mask	0000000100000000	0100010100000000

or

AL CODE	AX	MASK
mov mask, 4500h		4500h
fstsw ax ; stores the contents of the status word into ax.	x0xxx0x1xxxxxxxx	4500h
and ax, mask	100h	4500h

ST = second operand

AL CODE	AX	MASK
mov mask, 0100010100000000 b		0100010100000000
fstsw ax ; stores the contents of the status word into ax.	x1xxx0x0xxxxxxxx	0100010100000000
and ax, mask 100000000	0100000000000000	0100010100000000

or

AL CODE	AX	MASK
mov mask, 4500h		4500h
fstsw ax ; stores the contents of the status word into ax.	x1xxx0x0xxxxxxxx	4500h
and ax, mask 100000000	4000h	4500h

Performing Jumps

From above, we see that comparison instructions only sets the status word. Therefore, to make our jump instructions from Chapter 12 work, we need to check the contents of the status word. In order to make the comparison we must first store the status word into a variable (word) or the ax register and then use the above mask, as shown above. The following example should give us a clear idea of how this is done:

EXAMPLES:

1. Assume each of the registers in the stack have been previously assigned values.

The following pseudo-code, and al pseudo-code will perform the following tasks:

Task1: If y is larger than x , then assign the contents of y to the memory location z.

Task2: If y is smaller than x, then assign contents of x to the memory location z.

Task3: If y is equal to x , then assign zero to the memory location z.

PSEUDO-CODE	AL PSEUDO-CODE
MASK:= 4500h	MASK:= 4500h
IF Y > X THEN	ST:= Y
	COMPARE ST, X
	AX:= STATUS- WORD
	AX:= AX .AND. MASK
	IF AX = 0h THEN
BEGIN	**BEGIN**
Z:= Y	EAX:= Y
	Z:= EAX
END	**END**
IF Y < X THEN	IF AX = 100h THEN
BEGIN	**BEGIN**
Z:= X	EAX:= X
	Z:= EAX
END	**END**
IF Y = X THEN	IF AX:= 4000h THEN
BEGIN	**BEGIN**
Z:= 0	Z:= 0
END	**END**

Using the above pseudo-code and al pseudo-code, the below partial assembly language program will find the larger of x, y where x = 7 and y = 2.

AL PSEUDO-CODE	AL CODE	Y	X	Z	ST	AX	EAX
M:= 4500h	mov m, 4500h						
X:= 7	mov x, 7		7				
Y:= 2	mov y, 2	2	7				
ST:= Y	fild y	2	7		2		
COMPARE ST, X	fcom x	2	7		2		

AX:= STATUS- WORD	fstsw ax	2	7		2	x0xxx0x1xxxxxxxx	
AX:= AX .AND. M	and ax,m	2	7		2	100h	
IF AX = 0h THEN	cmp ax, 0h	2	7		2	100h	
	jne L1	2	7		2	100h	
BEGIN	begin:	2	7		2	100h	
EAX:= Y	mov eax, y	2	7		2	100h	
Z:= EAX	mov z, eax	2	7		2	100h	
END	end: jmp end2	2	7		2	100h	
IF AX =100h THEN	L1: cmp ax, 100h	2	7		2	100h	
	jne begin2	2	7		2	100h	
BEGIN	begin:	2	7		2	100h	
EAX:= X	mov eax, x	2	7		2	100h	7
Z:= EAX	mov z, eax	2	7	7	2	100h	7
END	end: jmp end2	2	7	7	2	100h	7
IF AX:= 4000h THEN		2	7	7	2	100h	7
BEGIN	begin2:	2	7	7	2	100h	7
Z:= 0	mov z, 0	2	7	7	2	100h	7
END	end2:	2	7	7	2	100h	7

2. The following program will compute the harmonic sum

1 + 1/2 + 1/3 + ... + 1/n
until
1/n < e,
where 0 < e < 1

Assume e = 0.00001 .

Note: See Model Program below.

PSEUDO CODE	AL PSEUDO CODE	ASSEMBLY CODE
E:= 0.00001	E:= 0.00001	e real4 0.00001
F:= 1.0	F:= 1.0	f real4 1.0
SUM := 0.0	SUM:= 0.0	sum real4 0.0
N := 1	N:= 1.0	n real4 1.0
ONE:= 1	ONE:= 1.0	one real4, 1.0
MASK:= 4500h	MASK:= 4500h	mov mask ,4500h
WHILE F ≥ E	WHILE: SK:= F	while: fld f
	FCOM E	fcom e
	AX:= STATUS WORD	fstsw ax
	AX:= AX .AND. MASK	and ax, mask
	IF AX = 100h THEN JUMP END	comp ax, 100h je end
BEGIN	**BEGIN:**	**begin:**
SUM:= SUM + F	SK:= SUM	fld sum
	SK:= SK + F	fadd f
	SUM:= SK	fst sum
N:= N + ONE	SK:= N	fld n
	SK:= SK + ONE	fadd one
	N:= SK	fst n
F:= ONE/N	SK:= ONE	fld one
	SK:= SK/N	fdiv n
	F:= SK	fst f
	JUMP WHILE	jmp while
END	**END**	end:

■

21.2 - Rounding Floating Point Numbers

In order to write such programs we need to be able to truncate decimal values. The contents of the control register (see below) determines how data is to be rounded when data in the ST register is transferred to an integer variable. There are four types of rounding:

• normal rounding of the number to an integer

• rounding the number up to the nearest integer

• rounding the number down to the nearest integer

• truncating the number to its integer value.

The following table gives the hexadecimal representation of the contents of the control register that is needed to perform rounding in ST:

BYTE POSITION	2	1
Round the number to the nearest integer	00	00
Round the number up to the nearest integer	08	00
Round the number down to the nearest integer	04	00
Truncate the number to its integer value	06	00

Examples:

1. 23.678 $\Rightarrow$ 24, normal rounding to an integer.

2. 23.678 $\Rightarrow$ 24, rounded up to the nearest integer

3. 23.678 $\Rightarrow$ 23, rounded down to the nearest integer

4. 23.678 $\Rightarrow$ 23, truncated to its integer value.

The control register

The control register is a 16 bit register that determines the kind of rounding that is to take place. When copying a value from the ST register to an integer variable, the 11th and 12th bits of the control register has to be modified to determine what type of rounding is to take place. This can be accomplished by transferring to the control register one of the bytes in the table above.

The table below are the instructions that will copy the contents of an integer variable from and to the control register:

MNEMONIC	OPERAND	ACTION
fstcw	memory variable (integer)	Copies the contents of the control register to a memory variable
fldcw	memory variable (integer)	Copies the contents of the memory variable to the control register

To round a number to the desired type, the following order has to be followed.

1. Copy the desired byte, from the table above, to the control register..

2. Copy the contents of ST to a given integer variable.

Examples:

1. Normal Rounding

```
; 2.9 ⇒ 3
.data
n word ?
x real4  2.9
round word 0h
.code

_start :

fld x ;           2.9 ⇒ st(0)
fldcw round;      0h  ⇒ control
register
fist n;             3 ⇒   n
public _start
```

2.Rounding Down

```
; 2.9 ⇒ 2
.data
n word ?
x real4  2.9
round word 0400h
.code

_start :

fld x ;            2.9 ⇒ st(0)
fldcw round;       0400h  ⇒ control
register
fist n ;              2 ⇒   n
public _start
```

3. Rounding Up

```
; 2.1 ⇒ 3
.data
n word ?
x real4  2.1
round word 0800h
.code

_start :

fld x ;            2.1 ⇒ st(0)
fldcw round;       0800h  ⇒ control
register
fist n ;              3 ⇒   n
public _start
```

4.Truncating

```
; 2.9 ⇒ 2
.data
n word ?
x real4  2.9
round word 0600h
.code

_start :

fld x ;           2.9 ⇒ st(0)
fldcw round;      0600h  ⇒ control
register
fist n ;             2 ⇒   n
public _start
```

Exercise:

1. Write a AL program that will perform the following:

1. Store in a variable the decimal representation of the number 1/7

2. Round the number to 5 places of accuracy.
■

Model Program

```
; This program will compute the harmonic sum

;1 + 1/2 + 1/3 + ... + 1/n
;until
;1/n < e,
;where 0 < e < 1

;Assume e = 0.00001

.386
.MODEL FLAT

.STACK 4096

.DATA
```

2

```
.CODE
 e real4   0.00001
 f real4   1.0
 sum real4  0.0
 n real4  1.0
 one real4, 1.0

_start:

;start assembly language code
mov mask ,010001010000000 b

while: fld f
fcom e
fstsw ax
and ax, mask
comp ax,  0000000010000000b
je end
begin:
fld sum
fadd f
fst sum
fld n
fadd one
fst n
fld one
fdiv n
fst f
jmp while
end:
;end of assembly language code

PUBLIC_start

END
```

Project:

1. Write a program that will round the floating-point representation of the fraction n/m to k places of accuracy.

2. Write a program that will convert a decimal - point number to its scientific representation.

INTRODUCTION

In Chapter 15, we say how arrays in assembly language allows the programmer to store a large amount of integer numeric data sequentially in memory locations. In this chapter we will study two other types of instructions in assembly that performs dynamic storage for decimal numbers: the push and pop instructions.

Definition of the push instructions: Push instructions will insert data into registers or memory locations.

Definition of the pop instructions; Pop instructions may remove data from registers or memory locations and insert data into registers or memory locations. .

22.1 Floating Point Push and Pop Instructions.

The following instructions will bring about push and pops' that are used in floating point programming. They are part of the instruction set which were first introduced in Chapter 18.

As you will recall, the operands of all floating-point instructions begin with the letter f. When storing or changing data in the registers, the following floating point instructions will cause the data that is replaced in the register to be pushed down to the registers below or up to the registers above.

Storing data from memory to the registers

We will assume the registers have the numbers

ST	10.0
ST(1)	15.0
ST(2)	20.0
ST(3)	25.0
ST(4)	
ST(5)	
ST(6)	
ST(7)	

The following are the floating-point instructions that will store data from memory to a given register.

MNEMONIC	OPERAND	ACTION
fld	memory (real)	the real number from memory is stored in ST and *data is pushed down*.

Example:

.DATA
x   REAL4   30.0

fld x; stores the content of x (real) into register ST and pushes the other values down.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**30.0**
ST(1)	15.0	10.0
ST(2)	20.0	15.0
ST(3)	25.0	20.0
ST(4)		25.0
ST(5)		
ST(6)		
ST(7)		

MNEMONIC	OPERAND	ACTION
fild	memory (integer)	the integer number from memory is stored in ST, converted to floating- point and *data is pushed down*.

Example:

.DATA
x   DWORD   50

fild x; stores the content of x (integer)   into register ST and pushes the other values down.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**50.0**
ST(1)	15.0	10.0
ST(2)	20.0	15.0
ST(3)	25.0	20.0
ST(4)		25.0
ST(5)		

ST(6)		
ST(7)		

MNEMONIC	OPERAND	ACTION
fld	st(k)	the number in st(k) is stored in ST and *data is pushed down.*

Example:

fld st(2) ; stores the number 20.0 into register ST and pushes the other values down.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**20.0**
ST(1)	15.0	10.0
ST(2)	20.0	15.0
ST(3)	25.0	20.0
ST(4)		25.0
ST(5)		
ST(6)		
ST(7)		

Important: Once the stack is full, additional stored data will cause the bottom values to be lost. Also the *finit* instruction will clear all the values in the register.

Copying data from the stack

We will assume the registers have the numbers

ST	10.0
ST(1)	15.0
ST(2)	20.0
ST(3)	25.0
ST(4)	
ST(5)	
ST(6)	

ST(7)	

MNEMONIC	OPERAND	ACTION
fstp	st(k)	makes a copy of ST and stores the value in ST(k) and then *ST is popped off the stack by moving the data up.*

Example:

fstp ST(2) ; stores the content of ST into ST(2) and then pops ST off the stack by moving the data up.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	15.0
ST(1)	15.0	**10.0**
ST(2)	20.0	25.0
ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

MNEMONIC	OPERAND	ACTION
fstp	memory (real)	makes a copy of ST and stores the value in a real memory location . *ST is popped off the stack.*

Example:

.DATA
x real4 ?
fstp x ; stores the content of ST into x. ST is popped off the stack.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	15.0
ST(1)	15.0	20.0
ST(2)	20.0	25.0
ST(3)	25.0	

ST(4)		
ST(5)		
ST(6)		
ST(7)		

MNEMONIC	OPERAND	ACTION
fistp	memory (integer)	Converts to integer a copy of ST and stores the value in a integer memory location. *ST is popped off the stack.*

Example:

.DATA
x DWORD ?

fistp x ; stores the content of ST as an integer number into x.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	15.0
ST(1)	15.0	20.0
ST(2)	20.0	25.0
ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Adding contents of the two floating-point registers.

We will assume the registers have the numbers

ST	10.0
ST(1)	15.0
ST(2)	20.0
ST(3)	25.0
ST(4)	

ST(5)	
ST(6)	
ST(7)	

The following are the floating-point instructions that will add the contents of two floating-point registers.

MNEMONIC	OPERAND	ACTION
fadd	none	First it *pops both ST and ST(1);* next it adds ST and ST(1); finally *the sum is pushed onto the stack.*

Example:

fadd ; first it pops both st and st(1); next it adds st and st(1); finally the sum is pushed onto the stack.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**45.0**
ST(1)	15.0	20.0
ST(2)	20.0	25.0
ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

MNEMONIC	OPERAND	ACTION
faddp	st(k), st	Adds ST(k) and ST; ST(k) is replaced by the sum and *ST is popped from the stack.*

Example:

faddp st(2), st ; adds ST(2) and ST; ST(2) is replaced by the sum and ST is popped from the stack.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	15.0
ST(1)	15.0	**30.0**
ST(2)	20.0	25.0

ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Subtracting the contents of the two floating-point registers.

The following are the floating-point instructions that will subtract the contents of two floating-point registers.

MNEMONIC	OPERAND	ACTION
fsub	none	*First it pops ST and ST(1)* ;next is calculates ST(1) - ST; next *it pushes the difference into ST.*
fsubr	none	*First it pops ST and ST(1)* ;next is calculates ST - ST(1) ; *next it pushes the difference into ST.*

Example:

fsub ; first it pops st and st(1) ;next is calculates st(1) - st; next it pushes the difference into st.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**- 2.0**
ST(1)	15.0	27.0
ST(2)	27.0	25.0
ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

MNEMONIC	OPERAND	ACTION
fsubp	st(k), st	computes ST(k) - ST; replaces ST(k) by the difference; finally *pops ST from the stack*
fsubpr	st(k), st	computes ST - ST(k) ; replaces ST(k) by the difference; finally *pops ST from the stack*

Example:

fsubp st(1), st ; computes st(1) - st; replaces st(1) by the difference; finally pops ST from the stack

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**5.0**
ST(1)	15.0	20.0
ST(2)	20.0	25.0
ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Multiplying the contents of the two floating-point registers.

The following are the floating-point instructions that will multiply the contents of two floating-point registers.

MNEMONIC	OPERAND	ACTION
fmul	none	First it *pops both ST and ST(1)* ; next it multiplies ST and ST(1); finally the *product is pushed onto the stack*.

Example:

fmul ; first it pops both st and st(1); next it multiplies st and st(1); finally the product is pushed onto the stack.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	**500.0**
ST(1)	15.0	20.0
ST(2)	20.0	25.0
ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

MNEMONIC	OPERAND	ACTION
fmulp	st(k) , st	multiplies ST(k) and ST; ST(k) is replaced by the product and *ST is popped from the stack.*

Example:

fmulp st(3), st ; multiplies st(3) and st; then st(k) is replaced by the product and st is popped from the stack.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	10.0	15.0
ST(1)	15.0	20.0
ST(2)	20.0	**250.0**
ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Dividing the contents of floating-point registers.

The following are the floating-point instructions that will divide the contents of floating-point registers.

MNEMONIC	OPERAND	ACTION
fdiv	none	First it *pops both ST and ST(1) ;* next it computes ST(1)/ ST; finally *the quotient is pushed onto the stack.*
fdivr	none	*First it pops both ST and ST(1) ;* next it computes ST/ ST(1); finally *the quotient is pushed onto the stack.*

Example:

fdiv ; first it pops both st and st(1); next it computes ST(1)/ ST ; finally the quotient is pushed onto the stack.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	5.0	**1.25**
ST(1)	15.0	20.0
ST(2)	20.0	25.0

ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

MNEMONIC	OPERAND	ACTION
fdivp	st(k), st	computes ST(k) /ST ; then ST(k) is replaced by the quotient. Next *ST is popped from the stack.*
fdivpr	st(k), st	computes ST /ST(k) ; then ST(k) is replaced by the quotient. Next *ST is popped from the stack.*

Example:
fidivp st(2) ; computes st(2) /st ; then st(2) is replaced by the quotient and ST is popped from the stack.

REGISTER	BEFORE EXECUTION	AFTER EXECUTION
ST	5.0	15.0
ST(1)	15.0	**4.0**
ST(2)	20.0	25.0
ST(3)	25.0	
ST(4)		
ST(5)		
ST(6)		
ST(7)		

Instructions that Compare Floating-Point numbers

MNEMONIC	OPERAND	ACTION
fcomp	(none)	compares ST and ST(1); then *pops the stack*
fcomp	st(k)	compares ST and ST(k); then *pops the stack*
fcomp	memory (real)	compares ST and a real number in memory; then *pops the stack*
fcomp	memory (integer)	compares ST and a integer number in memory; then *pops the stack*
fcompp	(none)	compares ST and ST(1) and then *pops the stack twice*

22.2 The 80x86 Stack

The directive

.STACK 4096

in the assembly language has the assembler reserve 4096 byes of storage. This will allow the programmer to temporarily store integer data in this location. The instruction to store data sequentially is the push instruction.

The push instruction

The syntax of the push instruction is

push *source*

where the source can be any of the following:

- 16 bit register (AX, BX, CX, DX)
- 32 bit register (EAX, EBX, EDX, EDX)
- a declared word or doubleword variable
- a numeric byte, word or doubleword

The push instruction will sequentially store data in the stack starting at the initial location.

Note: For simplicity, we will only push 32 bit registers or numeric values.

EXAMPLE

AL CODE	EAX	STACK											
mov eax, 52B6h	**52B6**												
push eax	52B6	**00**	**00**	**52**	**B6**								
mov eax, 23A7h	**23A7**	00	00	52	B6								
push eax	23A7	**00**	**00**	**23**	**A7**	00	00	52	B6				
mov eax, 72346711h	**72346711**	00	00	23	A7	00	00	52	B6				
push eax	72346711	**72**	**34**	**67**	**11**	00	00	23	A7	00	00	52	B6

Other push instructions

• **pushw**

When a numeric integer is to be pushed into the stack, to prevent confusion, the assembler needs to be informed as to its data type. The following push instructions perform this task:

• pushw *source*

where

source is a numeric value.

This push instruction will identify the numeric value to be stored as a word.

• pushd *source*

where

source is a numeric value.

This push instruction will identify the numeric value to be stored as a doubleword.

The pop instruction

The pop instruction will copy data from the stack, using the rule: “ last in first copied”, and store the data at the designated destination. The data copied will be popped from the stack and the remaining data will be push up the stack. .

The syntax of the pop instruction is

pop *destination*

where the destination can be any of the following:

• 16 bit register (AX, BX, CX, DX)

• 32 bit register (EAX, EBX, EDX, EDX)

• a declared word or doubleword variable

EXAMPLE:

AL CODE	EAX	EBX	STACK							
mov eax, 52B6h	**52B6**									
push eax	52B6		**00**	**00**	**52**	**B6**				
mov eax, 23A7h	**23A7**		00	00	52	B6				
push eax	23A7		**00**	**00**	**23**	**A7**	00	00	52	B6
pop ebx	23A7	**000023A7**	00	00	52	B6				
pop ebx	23A7	**000052B6**								

Note: Perhaps the best use of the push, pop instructions is to give the programmer additional temporary storage.

PROJECT:

Write an assembly language program that will find and store in the stack all positive integer numbers between 1 and N that are prime.

WORKING WITH STRINGS

CHAPTER 23 - DYNAMIC STORAGE: STRINGS

INTRODUCTION

So far in this book, we have only been working with numeric data. In this chapter we will define and work with string data . Strings are very important in that they can be used to communicate with the programmer and user. We start with the definition of a string and its numeric representation: the ASCII code.

23.1 The ASCII Code

Definition of a string: A string is a sequence of printable characters such as numbers, letters, spaces and special symbols : $, *, etc enclosed in single quotes: ′ ′.

Examples:

' Hello!' , 'Sam lives here' , ' To Be Or Not To Be.' , ' x = 2y + 3z.'

Now all data entered must be represented as numeric values. In assembly language, as well as many computer languages the numeric representation of the ASCII code is used.

ASCII (*American Standard Code for Information Interchange*), is a character encoding based on the English alphabet. ASCII codes represent text in computers, communications equipment, and other devices that work with text. Most modern character encoding systems have a historical basis in ASCII.

ASCII was first published as a standard in 1967 and was last updated in 1986. It currently defines codes for 33-non-printing, mostly obsolete control characters that affect how text is processed, plus 95 printable characters (starting with the space character).

ASCII is strictly a seven-bit code; meaning that it uses the bit patterns representable with seven binary digits (a range of 0 to 127 decimal) to represent character information. For example three important codes are the null code (00), carriage return (0D) and line feed (0A).”

The following is a table of the ASCII code along with each string’s symbol associated with its hexadecimal number value:

ASCII Table

ASCII SYMBOL	HEX	DECIMAL	NAME	ASCII SYMBOL	HEX	DECIMAL	NAME
	00	0	Null	@	40	64	At
SOH	01	1	Start of Header	A	41	65	
STX	02	2	Start of Text	B	42	66	
ETX	03	3	End of Text	C	43	67	

EOT	04	4	End of Transmission	D	44	68	
ENG	05	5	Enquire	E	45	69	
ACK	06	6	Acknowledge	F	46	70	
BEL	07	7	Bell	G	47	71	
BS	08	8	Backspace	H	48	72	
HT	09	9	Horizontal Tab	I	49	73	
LF	0A	10	Line Feed	J	4A	74	
VT	0B	11	Vertical Tab	K	4B	75	
FF	0C	12	Form Feed	L	4C	76	
CR	0D	13	Carriage Return	M	4D	77	
SO	0E	14	Shift Out	N	4E	78	
SI	0F	15	Shift In	O	4F	79	
DLE	10	16	Data Link Escape	P	50	80	
DC1	11	17	Device Control 1	Q	51	81	
DC2	12	18	Device Control 2	R	52	82	
DC3	13	19	Device Control 3	S	53	83	
DC4	14	20	Device Control 4	T	54	84	
NAK	15	21	Negative Acknowledge	U	55	85	
SYN	16	22	Synchronous Idle	V	56	86	
ETB	17	23	End of Transmission Block	W	57	87	
CAN	18	24	Cancel	X	58	88	
EM	19	25	End of Medium	Y	59	89	
SUB	1A	26	Substitute	Z	5A	90	
ESC	1B	27	Escape	[	5B	91	Open Square Bracket
FS	1C	28	File Separator	\	5C	92	Back Slash

GS	1D	29	Group Separator	]	5D	93	Close Square Bracket
RS	1E	30	Record Separator	^	5E	94	Circumflex
US	1F	31	Unit Separator	_	5F	95	Underscore
SP	20	32	Space or Blank	'	60	96	Single Quote
!	21	33	Exclamation Point	a	61	97	
"	22	34	Quotation Mark	b	62	98	
#	23	35	Number sign (Pound sign)	c	63	99	
$	24	36	Dollar Sign	d	64	100	
%	25	37	Percent Sign	e	65	101	
&	26	38	Ampersand	f	66	102	
'	27	39	Apostrophe (Single quote)	g	67	103	
(	28	40	Opening Parenthesis	h	68	104	
)	29	41	Close Parenthesis	i	69	105	
*	2A	42	Asterisk(Star sign)	j	6A	106	
+	2B	43	Plus Sign	k	6B	107	
,	2C	44	Comma	l	6C	108	
-	2D	45	Hyphen (Minus)	m	6D	109	
.	2E	46	Dot (Period)	n	6E	110	
/	2F	47	Forward Slash	o	6F	111	
0	30	48	Zero	p	70	112	
1	31	49		q	71	113	
2	32	50		r	72	114	
3	33	51		s	73	115	
4	34	52		t	74	116	
5	35	53		u	75	117	

6	36	54		v	76	118	
7	37	55		w	77	119	
8	38	56		x	78	120	
9	39	57		y	79	121	
:	3A	58	Colon	z	7A	122	
;	3B	59	Semi Colon	{	7B	123	Open Curly Bracket
<	3C	60	Less Than	|	7C	124	OR (Pipe)
=	3D	61	Equality	}	7D	125	Close Curly Bracket
>	3E	62	Greater Than	~	7E	126	Equivalence (tilde)
?	3F	63	Question Mark	DEL	7F	127	Delete

Note: The associated ASCII codes are always in hexadecimal.

23.2 Storing Strings

In this chapter we will find that there are several instructions to store strings in registers as well as defined variables.

- **mov *register* , *string***
- **mov *variable* , *string***

The register and the variable can be of any data type.

When a string is stored, each character of the string is converted to its hexadecimal ASCII code. For example the string ′- x3′ is made up of 4 characters (counting the space but not the single quotes).The assembler will convert the 4 characters into its corresponding ASII code:

Examples:

ASSEMBLY CODE	**EAX**			
mov eax, '- x3'	2D	20	78	33

ASSEMBLY CODE	X
x byte ?	
mov x , '/ '	2F

Exercise:

Convert the following strings to its ASCII codes:

ASSEMBLY CODE	EAX			
mov eax, '+ YZ '				
mov eax, '/'				
mov eax, '* %'				

• **The string variables:**

Since all strings are converted by the assembler into integer bytes, we use the normal directives to define the variables as bytes, words or double words.

Examples:

1.

x BYTE 20 DUP (?) ; This directive will assign 20 blank bytes to the variable x.

2.

Hamlet BYTE 'To be or not to be'; The assembler will set aside 18 bytes containing the ASC codes.

54	65	20	62	65	20	65	72	20	6E	65	54	20	54	65	20	62	65

3.

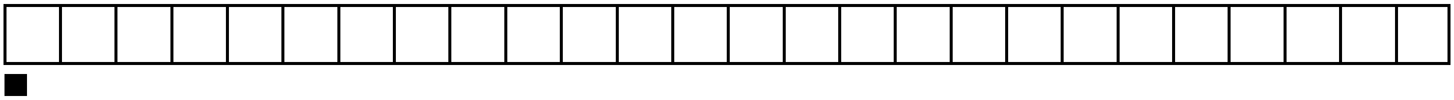
array_x DWORD 5 DUP '- 23'; The assembler will set aside 5 dwords containing the ASC code '- 23' .

2D	20	32	33	2D	20	32	33	2D	20	32	33	2D	20	32	33	2D	20	32	33

Exercise:

Complete the following tables:

Hamlet BYTE 'Brevity is the soul of wit'



A natural question should be raised: how does the programmer assign strings to registers and variables without using directly the above type of directives ? For example the above x variable has 20 blank bytes assigned to it for storage. Therefore, we should be able to assign any string of length 20 characters or less to the variable x. Since string data are changed to ASCII code by the assembler, we can use, as shown above, the *mov* instruction to assign a string to a register or a variable. However, there are times when we want to copy strings stored in one variable to another variable. We should note that transferring some strings through a register may not be possible due to the size of the string. The following sections will give the necessary instructions to perform such tasks.

23.2 The movs instructions.

To move strings from one variable to another variable, we define the following 3 movs instructions:

Definition *movsb**:* The movsb will move the byes of a variable, byte by byte to another variable. The movsb instruction has no operands.

Definition *movsw**:* The movsw will move the words of a variable, word by word to another variable. The movsw instruction has no operands.

Definition *movsd**:* The movsd will move the dwords of a variable, dword by dword to another variable. The movsd instruction has no operands.

*S*ince the three movs instructions have no operands, the assembler has to know which variable is the source of the string and which variable is the destination. The location of these variables are to be stored in the ESI and the EDI registers.

The ESI and EDI registers

Definition ESI: The ESI register must contain the location of the source variable.

Definition EDI: The EDI register must contain the location of the destination variable.

•The lea instruction

In order to store the locations in these two registers, we use the *lea* instruction:

Definition lea: The form of the lea instruction is

lea register, variable name

where, for this application, the registers are esi or edi.

Once the esi and edi are initialized the movs instructions will increment these register under the following rule:

1. The movsb will cause the esi and edi to be incremented to the next byte.

2. The movsw will cause the esi and edi to be incremented to the next word.

3. The movsd will cause the esi and edi to be incremented to the next dword.

Example:

ASSEMBLY CODE	X				Y			
x dword '- x3'	2D	20	78	33				
y dword ?	2D	20	78	33				
lea esi, x	2D	20	78	33				
lea edi, y	2D	20	78	33				
movsb	2D	20	78	33	33			
movsb	2D	20	78	33	78	33		
movsb	2D	20	78	33	20	78	33	
movsb	2D	20	78	33	2D	20	78	33

■

Exercises:

1. Hamlet DWORD 'To be or not to be'

Write a program that will move the string in variable Hamlet to the variable Shakespeare DWORD ?

■

23.3 More String Instructions

The following are additional string instructions that can be very useful when working with strings.

The stos instruction

There are three stos instruction:

• *Definition:* stosb copies a byte stored in the AL register to the destination variable.

Example:

AL CODE	AL (Byte is in ASCII symbols)	X (Byte is in ASCII symbols)
x byte ?		

mov al, '&'	**&**	
lea edi, x	&	
stosb	&	**&**

• *Definition:* stosw copies a word stored in the AX register to the destination variable.

Example:

AL CODE	AX (Word is in ASCII symbols)	X (Word is in ASCII symbols)
x word ?		
mov ax, '-9'	**-9**	
lea edi, x	-9	
stosw	-9	**-9**

• *Definition:* stosd copies a word stored in the EAX register to the destination variable.

Example

AL CODE	EAX (DWord is in ASCII symbols)	X (DWord is in ASCII symbols)
x dword ?		
mov eax, 'home'	**home**	
lea edi, x	home	
stosd	home	**home**

1. In the above 3 examples change the ASCII symbols to their corresponding hexadecimal values.
■

The lods instruction

There are three lods instruction:

• *Definition:* lodsb copies a source stored in the byte variable to the AL register .

Example:

AL CODE	AL (Byte is in ASCII symbols)	X (Byte is in ASCII symbols)
x byte '#'		**#**
lea esi, x		#

lodsb	**#**	#

• *Definition:* lodsw copies a source stored in the word variable to the AX register .
Example:

AL CODE	AX (Word is in ASCII symbols)	X (Word is in ASCII symbols)
x word '$7 '		**$7**
lea esi, x		$7
lodsw	**$7**	

• *Definition:* lodsd copies a source stored in the word variable to the EAX register

Example:

AL CODE	EAX (Word is in ASCII symbols)	X (DWord is in ASCII symbols)
xdword 'Bach '		**Bach**
lea esi, x		Bach
lodsd	**Bach**	

Exercise:

1. In the above 3 examples change the ASCII symbols to their corresponding hexadecimal values.
■

The rep instruction

• *Definition:* The rep instruction is a prefix to several other instructions to perform a given repetitive task. The number of repetitions is a given number stored in the ECX register. When completed the ECX register will contain zero (0).

Examples:

1.

AL CODE	ECX	AL (Byte is in ASCII symbols)	X (Dword is in ASCII symbols)			
x dword ?						
mov al, '^'		^				
lea edi, x		^				

mov ecx, 4	**4**	^				
rep stosb	4	^	^	^	^	^

2.

AL CODE	ECX	AX (Words in ASCII symbols)	X (Words in ASCII symbols)				
x word 5 dup (?)							
mov ax, 'WA'		WA					
lea edi, x		WA					
mov ecx, 5	**5**	WA					
rep stosw	5	WA	**WA**	**WA**	**WA**	**WA**	**WA**

3.

AL CODE	ECX	EAX (Words in ASCII symbols)	X (DWords in ASCII symbols)			
x dword 4 dup (?)						
mov eax, '1234'		**1234**				
lea edi, x		1234				
mov ecx, 4	**4**	1234				
rep stosd	4		**1234**	**1234**	**1234**	**1234**

Exercises:

1. In the above 3 examples change the ASCII symbols to their corresponding hexadecimal values.

2. Complete the table below:

AL CODE	ECX	Y (DWords in ASCIIsymbols)				X (DWords in ASCIIsymbols)			
x dword 4 dup (?)									
Y dword '123456789abcde'									
mov ecx, 4									
lea esi, y									
lea edi, x									

rep movsd									

3. Complete the table below:

AL CODE	**ECX**	**Y (DWords in hex symbols)**				**X (DWords in hex symbols)**			
X dword 4 DUP (?)									
Y dword '123456789abcde'									
mov ecx, 4									
lea esi, y									
lea edi, x									
rep movsd									

Other repeat instructions

Depending on the suffix, the following are additional versions of the rep instruction:

• Definition: the repe prefix is to repeat while ECX > 0 and the suffix operation compute a value equal 0.

• Definition: the repz prefix is to repeat while ECX > 0 and the suffix operation compute a value equal 0.

• Definition: the repne prefix is to repeat while ECX > 0 and the suffix operation compute a value not equal 0.

• Definition: the repnz prefix is to repeat while ECX > 0 and the suffix operation compute a value not equal 0.

Note: repz/repe and repnz/repne pairs are equivalent instructions. Also all repeat instructions can be used in conjunction will procedures. In this way multiple instructions can be repeated.

The cmps instruction

There are three cmps instructions:

• *Definition:* cmpsb compares the binary source and binary designation strings. It does not have operands.

• *Definition:* cmpsw compares the word source and word designation strings . It does not have operands.

• *Definition:* cmpsd compares the double word and double word designation strings . It does not have operands.

Note: The cmps instructions should be use in conjunction with the jump instructions of Chapter 11.
The following is a table of the conditional jumps for the signed order of rings in assembly language:

Mnemonic[1]	Description
je	jump to the label if *source* = *destination;* *jump if equal to*
jne	jump to the label if *source* ≠ *destination;* *jump if not equal to*
jnge	jump to the label if *source* < *destination*; *jump if not greater or equal*
jnle	jump to the label if *source* > *destination*; *jump if not less than or equal*
jge	jump to the label if *source* ≥ *destination*; *jump if greater than or equal*
jle	jump to the label if *source* ≤ *destination*; *jump if less than or equal*
jl	jump to the label if *source* < *destination*; *jump if less than*
jnl	jump to the label if *source* ≥ *destination;* *jump if not less than*
jg	jump to the label if *source* > *destination*; *jump if greater than*
jng	jump to the label if *source* ≤ *destination*; *jump if not greater than*

Note. Remember, that the string comparisons are actually the comparisons of the numeric values associated with the strings.

The scas instruction

The scan string instruction is used to scan a string for the presence of a given string element. The scan string is the designation string and the element that is being searched for is in a given register .

There are three scas instructions:

• *Definition*: The scasb requires the element being searched for is in the AL register.

• *Definition*: The scasw requires the element being searched for is in the AX register.

• *Definition:* The scasd requires the element being search for is in the EAX register.

Note: To scan the entire string for the given elements, the *repne* prefix is used with the scas instruction.

Algorithm: Checks to see if a string has a given element of a byte size.

ASSEMBLY LANGUAGE CODE	COMMENTS
stringlocation byte ‘string’	
mov al, 'byte element'	
lea edi,stringlocation	' string' is the sting to check if it contains the byte element
mov ecx, n	the number of bytes containing sting.
mov eax,ecx	will contain the location of the element
repne scasb	checks byte by byte . Will stop checking if the byte is found.
sub eax,ecx	location of the element if it exists in the string.

Exercise:

1. Write a program that will find the position location of “f” in the of the string 'I live in California '

PROJECT

1. Write an assembly language program that will convert an arbitrary string “$a_1a_2a_3...a_n$” to it number value $a_1a_2a_3...a_n$.

2. Write an assembly language program that will convert an arbitrary integer number $a_1a_2a_3...a_n$ to the string “$a_1a_2a_3...a_n$”.

CHAPTER 24 - STRING ARRAYS

INTRODUCTION

In Chapter 15, we created one and two dimensional integer arrays. In this chapter we will create arrays that contain strings. We will see that the string arrays and integer arrays share many of the same rules.

24.1 Storing Stings in the Directive

The following are the ways string(s) can be stored using the directive in the data portion of the program. We can use the following directives:

- *variable name data type ?*
- *variable name data type string*
- *variable name data type string_1, string_2, ..., string_n*

variable name data type dimension dup*(?)*

Examples:

variable name data type ?

1. x byte ?

will allow a one character string to be stored in x.

2. x word ?

will allow a two character string to be stored in x.

3. x dword ?

will allow a four character string to be stored in x.

variable name data type string

1. x byte *a string of any length*

will allow any size string to be stored in an array starting in location x.

x byte ‘abcde’

2. x word *string*

will allow a string of 2 characters to be stored in x x word ‘ab’

3. x dword *string*

will allow a string of 4 characters to be stored in x

x dword ‘abcd’

variable name data type   string_1, string_2, ..., string_n

1. x byte   *string_1, string_2, ..., string_n*

will allow a list of strings of any length starting in location x.

x byte ‘a’, ‘b’, ‘c’, ‘d’

2. x word   *string_1, string_2, ..., string_n*

will allow a list of strings of 2 characters each starting in location x.

x word ‘ab’, ‘cd’, ‘ef’, ‘gh’

3. x dword   *string_1, string_2, ..., string_n*

will allow a list of strings of 4 characters each starting in location x.

x dword ‘abcd’, ‘efgh’, ‘ijkl’, ‘mnop’

variable name data type   dimension dup*(?)*

will create a string array with a given dimension and data type.

Note: As in Chapter 14, The *lea* instruction will still be use to determine the first byte position of the array.

Retrieving strings stored in the variable

The following examples will demonstrate how strings are retrieved from the variables:

Examples:

1.

AL CODE	AL	byte 1	byte 2	byte 3
x byte ‘abc’		**a**	**b**	**c**
lea ebx,x		a	b	c
mov al,[ebx]	**a**	a	b	c

add ebx,1	a	a	b	c
mov al,[ebx]	**b**	a	b	c
add ebx,1	b	a	b	c
mov al,[ebx]	**c**	a	b	c

2.

AL CODE	AL	byte 1	byte 2	byte 3
x byte ‘a’, ‘b’, ‘c’		**a**	**b**	**c**
lea ebx,x		a	b	c
mov al,[ebx]	**a**	a	b	c
add ebx,1	a	a	b	c
mov al,[ebx]	**b**	a	b	c
add ebx,1	b	a	b	c
mov al,[ebx]	**c**	a	b	c

3.

AL CODE	AX	word 1	word 2	word 3
x word ‘ab’, ‘cd’, ‘ef ’		**ab**	**cd**	**ef**
lea ebx,x		ab	cd	ef
mov ax, [ebx]	**ab**	ab	cd	ef
add ebx,2	ab	ab	cd	ef
mov ax, [ebx]	**cd**	ab	cd	ef
add ebx,2	cd	ab	cd	ef
mov ax, [ebx]	**ef**	ab	cd	ef

4.

AL CODE	EAX	dword 1	dword 2	dword 3
x dword ‘abcd’, ‘ef’, ‘ghi’		**abcd**	**ef**	**ghi**

lea ebx,x		ab	cd	ef
mov ,[ebx]	**abcd**	ab	cd	ef
add ebx,4	abcd	ab	cd	ef
mov al,[ebx]	**ef**	ab	cd	ef
add ebx,4	ef	ab	cd	ef
mov al,[ebx]	**ghi**	ab	cd	ef

Exercise:

For the four examples above, fill in the appropriate cells with the ASCII code.

Creating a one dimensional string array using the dup(?) directive .

The following steps will define and set up the array.

Step 1: Define the directive *variable name data type   dimension* dup*(?)*

Step 2: Useing the lea instruction, store the first byte location in a 32 bit register.

Example:

x byte 10 (?)

lea ebx, x

Storing data in the array.

In the assembler we can use any of the registers EAX, EBX, ECX and EDX. The following definition is the assignment statement that will allow us to perform data assignments to and from memory cells:

mov [register], source instruction.

Definition: *mov [register], source*

where the following rules apply:

Rule1: The registers must be EAX, EBX, ECX, or EDX.

Rule2: The *source* can be any register, or variable.

Rule3: The *[register]* indicates the cell location where the bytes are to be located.

The [*register]* is call the indirect register.

Rule4: The lea instruction will establish the first byte location.

The mov [*register*], *source* instruction will store the sting in the source register or variable into the memory location indicated by the contents of the register.

For all examples in this chapter, we assume all numbers are represented as hexadecimal.

Examples:

The following examples show how string arrays are created and stored.

1. The following program will store the strings ‘a’, ‘b’, ‘c’ into the array of type BYTE.

PSEUDO CODE	AL CODE	AL	X		
Array X	x byte 10 dup(?) lea ebx,x		**byte 1**	**byte 2**	**byte 3**
X(1) := ‘a’	mov al, ‘a’	**a**			
	mov [ebx], al	a	**a**		
	add ebx,1	a	a		
X(2):= ‘b’	mov al, ‘b’	**b**	a		
	mov [ebx],al	b	a	**b**	
	add ebx,1	b	a	b	
X(3):= ‘c’	mov al,’c’	**c**	a	b	
	mov [ebx],al	c	a	b	**c**

Important: Since we are storing into individual byes, we increment by 1.

2. The following program will store numbers ‘ab’, ‘cd’, ‘ef’ into the array of type WORD.

PSEUDO CODE	AL CODE	AX	X		
Array X	x word ? lea ebx,x		**word 1**	**word 2**	**word 3**
X(1) := ‘ab’	mov ax, ‘ab’	**ab**			
	mov [ebx], ax	ab	**ab**		
	add ebx,2	ab	ab		
X(2):= ‘cd’	mov ax, ‘cd’	**cd**	ab		

	mov [ebx],ax	cd	ab	**cd**	
	add ebx,2	cd	ab	cd	
X(3):= ‘ef’	mov ax, ‘ef’	**ef**	ab	cd	
	mov [ebx],ax	ef	ab	cd	**ef**

Important: Since we are storing into individual byes for each word, <u>we increment by by 2.</u>

3. The following program will store numbers ‘abcd’, ‘efgh’, ‘ijk’ into the array of type DWORD.

PSEUDO CODE	**AL CODE**	**EAX**	**X**		
Array X	x dword ? lea ebx,x		**dword 1**	**dword 2**	**dword 3**
X(1) := ‘abcd’	mov eax, ‘abcd’	**abcd**			
	mov [ebx], eax	abcd	abcd		
	add ebx,4	abcd	abcd		
X(2):= ‘efgh’	mov eax,’efgh’	**efgh**	abcd		
	mov [ebx],eax	efgh	abcd	**efgh**	
	add ebx,4	efgh	abcd	efgh	
X(3):= ‘ijk’	mov eax,’ijk’	**ijk**	abcd	efgh	
	mov [ebx],eax	ijk	abcd	efgh	**ijk**

Important: Since we are storing into individual byes for each dword, <u>we increment by 4.</u>

mov register, [register]

Definition: *mov register, [register]*

where the following rules apply:

Rule1: The registers can be EAX, EBX, ECX, and EDX.

Rule2: The *[register]* indicates the cell location where the bytes are located.

where the following rules apply:

The mov register, [*register*] instruction will store the number contained in the address location in *[register]* into the register.

Example:

AL INSTRUCTIONS	eax	cl	X
x byte ‘abcdef’			**abcdef**
lea ebx,x			abcdef
mov eax, [ebx]	**abcd**		abcdef
mov cl, [ebx]	abcd	**a**	abcdef
add ebx,1	abcd	a	abcdef
mov cl,[ebx]	abcd	**b**	abcdef
add ebx,1	abcd	b	abcdef
mov cl , [ebx]	abcd	**c**	abcdef
add ebx,1	abcd	c	abcdef
mov cl, [ebx]	abcd	**d**	abcdef
add ebx,1	abcd	d	abcdef
mov cl, [ebx]	abcd	**e**	abcdef
add ebx,1	abcd	e	abcdef
mov cl, [ebx]	abcd	**f**	abcdef

AL INSTRUCTIONS	eax	ebx	BYTES:	1	2	3	4	5	6	7	8
mov eax, 2											
mov ebx, 7D12Eh											
mov [eax], ebx											
mov eax, 4											
mov ebx, 568923h											
mov [eax], ebx											
mov ebx, 3											
mov eax, [ebx]											

2. Write a assembly language program that will perform the following tasks:

Task 1: store the ASC codes of the alphabet a to z into an array x.

Task 2: retrieve the ASC codes from the array x.
■

PROJECTS

Write an assembly language program that will store all the ASC codes into an array x.

INTRODUCTION

The 80/86 MASM assembler provides the Kernel32 library of program utilities which includes input/out instructions. In this chapter we will examine programs that will perform the following functions:

• Output strings to the monitor

• Input strings from the keyboard

25.1 Outputting Strings to the Monitor

The following is a complete program that will output to the screen the message: “Good morning America!”

The following directives are used to input and output string data:

• ExitProcess PROTO NEAR32 stdcall, dwExitCode: DWORD

where

PROTO is a directive that prototypes the function ExitProcess
and
ExitProcess is a directive that is used to terminate a program.

• GetStdHandle

The *GetStdHandle* returns in EAX a handle for the I/O device.

Examples:

Program:

```
;   A complete program that will output to the screen the message: “Good morning America!”

.386
.MODEL FLAT
ExitProcess PROTO NEAR32 stdcall, dwExitCode: DWORD
```

```
;Setup for Writing to the Monitor

GetStdHandle PROTO NEAR32 stdcall,  nStdHandle:DWORD
WriteFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOfCharsToWrite:DWORD,
lpNumberOfBytesWritten:NEAR32, lpOverlapped:NEAR32
STD_OUTPUT EQU -11
cr     EQU 0dh               ; carriage return character
lf      EQU 0ah              ; line feed
```

```
.STACK 4096
.DATA
message  BYTE     'Good morning America!'; This is the message that will be displayed on the monitor
size DWORD 21 ;                                  Number of characters in message
written    DWORD ?
message_out    DWORD ?
```

```
.CODE
; The following instructions will print the message “Good morning America!”
_start:

INVOKE GetStdHandle,           ;                   Prepare output

    STD_OUTPUT                        ;                    -- to screen

      mov message_out, eax

    INVOKE WriteFile,               ;                    Initial output

           message_out,                   ;                    screen hardware location

          NEAR32 PTR message,  size,   ;            size of message

          NEAR32 PTR written,        ;             bytes written
           0 ;                                          overlapped mode
INVOKE ExitProcess, o                  ;      exit with return code o

PUBLIC _start

END
```

25.2 Inputting Strings from the keyboard

The following complete program will perform the following tasks.

Task 1: A message to the monitor will prompt the user to enter a message.

Task 2: Allow the user to enter a message.

Example:

```
;   A complete program that will allow the user  to enter a message and enter data from the keyboard.
.386

.MODEL FLAT
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

GetStdHandle PROTO NEAR32 stdcall,
   nStdHandle:DWORD

ReadFile PROTO NEAR32 stdcall,
     hFile:DWORD, lpBuffer:NEAR32, nNumberOFCharsToRead:DWORD,
    lpNumberOfBytesRead:NEAR32, lpOverlapped:NEAR32

WriteFile PROTO NEAR32 stdcall,
    hFile:DWORD, lpBuffer:NEAR32, nNumberOFCharsToWrite:DWORD,
   lpNumberOfBytesWritten:NEAR32, lpOverlapped:NEAR32

STD_INPUT EQU -10
STD_OUTPUT EQU -11

.STACK 4096
.DATA
request  BYTE  "Please enter a message ? "
CrLf    BYTE  0ah, 0dh
Enter_message   BYTE  80 DUP (?)
read_in    DWORD   ?
written_out DWORD  ?
handle_Out  DWORD  ?
handle_In  DWORD  ?
```

```
.code
; The following instructions will print the message "Please enter a message"

_start:
;  WRITE REQUEST
        INVOKE GetStdHandle,    ; get handle for console output
             STD_OUTPUT
      mov handle_In, eax

     INVOKE WriteFile,
     handle_In,
    NEAR32 PTR request,
      80,
             NEAR32 PTR written_out,
              0
```

```
; The following instructions will allow a message to be entered from the keyboard.
; INPUT DATA

 INVOKE GetStdHandle,    ; get handle for console output
             STD_INPUT
       mov handle_In, eax
       INVOKE ReadFile,
        handle_In,
        NEAR32 PTR Enter_message,
        80,
        NEAR32 PTR read_in ,
        0

                INVOKE ExitProcess, 0

INVOKE ExitProcess, o                    ;      exit with return code o

 PUBLIC _start
END
```

PROJECT

Write a program that will perform the following two tasks:

- an arbitrary number of hexadecimal numbers can be entered from the keyboard and stored in a array.
- the numbers can be retrieved from the array, converted to decimal and display onto the monitor.

CHAPTER 26 SIGNED NUMBERS AND THE EFLAG SIGNALS

INTRODUCTION

It is important to keep in mind that when working with integers numbers, that the numbers are contained in a ring of a given data type. When we preform arithmetic operations, it is possible that the resulting computations do not always return the expected value as they would appear in the ordinary integer number system. For example we would expect the simple expression 2 - 3 to return a value of -1. But if our number system is a 8 bit ring we will obtain the result 255 which is the additive inverse of -1. Let us assume for further discussion that the register we will work with in this chapter is the register AL, which is a 8 bit ring. Further we will assume the following table is a signed order representation of this ring in decimal (See chapter 8)

128	129	...	253	254	255	0	1	2	3	...	126	127
-128	-127	...	-3	-2	-1	0	1	2	3	...	126	127

where the bottom row represents the additive inverse of the above values.

If we wish to write a program that will print out the true value -1, how is this done when the instructions

move al, 2
sub al, 3

will return the value 255 in the register al?

To print the correct -1 we need to write al instructions that will perform the following tasks:

Task 1: Test what value resulted in the subtraction: 255.

Task2: Convert 255 into is additive inverse: 1

Task3: Store in a variable the ASCII code for -1: 2D31 (See Chapter 23).

Task4: Print this ASCU code (See Chapter 25).

Performing operations such as Task1 is the main emphasis of ths chapter.

26.1 THE EFLAGS

The EFLAG is a 32 bit register where some of its 32 bits indicated three types of flag signals resulting from arithmetic or logical operations:

• the sign flag

• the carry flag

• the overflow flag.

Before defining these important flags, we make the following observation: when performing arithmetic or logical operations we first assign a numeric integer to a byte register that has a 0 or 1 bit at its left most bit position . If after the operation, the resulting binary value will have a 0 or 1 in its left most bit position. If this bit is the same or different than the left most bit of the original value, a change may occur in the various flags listed above.

Addition and Subtract

Examples:

1.

AL CODE	**AL (decimal)**	**AL (binary)**	
move al, 1	1	**0**000	0001
add al, 2	3	**0**000	0011

Comment: The left-most bit of the original number 0000 0001 is a 0 and the left-most bit the resulting number 0000 0011 is also 0, Therefore the left-most bit has not changed.

2.

AL CODE	**AL (decimal)**	**AL (binary)**	
move al, 1	1	**0**000	0001
sub al, 2	255	**1**111	1111

Comment: The left-most bit of the original number 0000 0001 is a 0 and the left-most bit the resulting number 1111 1111 is 1, Therefore the left-most bit has changed.

3
.

AL CODE	**AL (decimal)**	**AL (binary)**	
move al, 254	254	**1**111	1110
add al, 10	8	**0**000	1000

Comment: The left-most bit of the original number 1111 1110 is a 1 and the left-most bit the resulting number 0000 1000 is 0, Therefore the left-most bit has changed.

4.

AL CODE	**AL (decimal)**	**AL (binary)**	
move al, 128	128	**1**000	0000
add al, 128	0	**0**000	0000

Comment: The left-most bit of the original number 1000 0000 is a 1 and the left-most bit the resulting number 0000 0000 is 0, Therefore the left-most bit has changed.

Exercises:

Complete the tables below:

1.

AL CODE	AL (decimal)	AL (hex)		AL (binary)	
move al, 1	1			**0**000	0001
add al, 2	3			**0**000	0011

2.

AL CODE	AL (decimal)	AL (hex)		AL (binary)	
move al, 1	1			**0**000	0001
sub al, 2	255			**1**111	1111

3.

AL CODE	AL (decimal)	AL (hex)		AL (binary)	
move al, 254	254			**1**111	1110
add al, 10	8			**0**000	1000

4..

AL CODE	AL (decimal)	AL (hex)		AL (binary)	
move al, 128	128			**1**000	0000
add al, 128	0			**0**000	0000

■

Definition of the sign flag:

After an arithmetic or logical operation on a integer value in a byte register, if the resulting binary number has at its left-most position a 1, then the sign flag will be assigned a value 1; otherwise a number 0.

Examples:

1.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 250	250	**1**1111010	

add al,1	251	**1**1111011	
mov sign,'yes'	251	**1**1111011	yes

Comment: The resulting number 251 has as its associated binary number a 1 as it left most bit.

2.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 120	120	**0**1111000	
add al,10	130	**1**0000010	
mov sign, 'yes'	150	**1**0000010	yes

Comment: The resulting number 150 has as its associated binary number a 1 as it left most bit.

3.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 220	220	**1**1011100	
add al,40	260	**0**0000100	
mov sign, 'no'	260	**0**0000100	no

Comment: The resulting number 260 has as its associated binary number a 0 as it left most bit.

4.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 22	22	**0**0010110	
add al,40	62	**0**0111110	
mov sign, 'no'	62	**0**0111110	no

Comment: The resulting number 260 has as its associated binary number a 0 as it left most bit.

Exercises

Complete the tables below.
1.

AL CODE	AL (decimal)	AL (binary)	AL (hex)	SIGN
mov al, 250	250	**1**1111010		

add al,1	251	**1**1111011		
mov sign,’yes’	251	**1**1111011		**yes**

2.

AL CODE	AL (decimal)	AL (binary)	AL (hex)	SIGN
mov al, 120	120	**0**1111000		
add al,10	130	**1**0000010		
mov sign, ‘yes’	150	**1**0000010		**yes**

3.

AL CODE	AL (decimal)	AL (binary)	AL (hex)	SIGN
mov al, 220	220	**1**1011100		
add al,40	260	**0**0000100		
mov sign,’no’	260	**0**0000100		**no**

4.

AL CODE	AL (decimal)	AL (binary)	AL(hex)	SIGN
mov al, 22	22	**0**0010110		
add al,40	62	**0**0111110		
mov sign, ‘no’	62	**0**0111110		**no**

Definition of the carry flag:

Assume the original number has a left-most 0 binary bit. After an arithmetic or logical operation if the resulting binary number has at its left-most position a 0, then the carry flag will be assigned a value 1; otherwise it will be assigned a numeric bit 0.

Examples:

1.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 250	250	**1**1111010	

add al,1	251	**1**1111011	
mov sign,'no'	251	**1**1111011	**no**

Comment: 250 has a 1 as its left-most binary bit. The resulting number 251 has as its associated binary number a 1 as it left most bit. Therefore, the left most bit it is still a 1.

2.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 120	120	**0**1111000	
add al,10	130	**1**0000010	
mov sign, 'no'	150	**1**0000010	**no**

Comment: The original binary number has as its left-most bit a 0. Therefore ,the carry flag is set to 0.

3.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 220	220	**1**1011100	
add al,40	260	**0**0000100	
mov sign, 'yes'	260	**0**0000100	**yes**

Comment: 220 has a 1 as its left-most binary bit. The resulting number 260 has as its associated binary number a 0 as it left most bit. Therefore, Therefore, the carry bit is set to 1..

.

4.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 22	22	**0**0010110	
add al,40	62	**0**0111110	
mov sign, 'no'	62	**0**0111110	**no**

Comment: The original binary number has as its left-most bit a 0. Therefore ,the carry flag is set to 0.

Exercises:

Complete the tables below.

1.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 250			
sub al,1			
mov sign,’no’			

2.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 120			
sub al,10			
mov sign, ‘no’			

3.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 220			
sub al,40			
mov sign, ‘yes’			

4.

AL CODE	AL (decimal)	AL (binary)	SIGN
mov al, 22			
add al,40			
mov sign, ‘no’			

Definition of the overflow flag:

<table>
<tr><td colspan="14">• ------------>---------------- > ------------------------> -------->---------> • OF</td></tr>
<tr><td>X</td><td>X</td><td>128</td><td>129</td><td>...</td><td>254</td><td>255</td><td>0</td><td>1</td><td>2</td><td>3</td><td>...</td><td>126</td><td>127</td><td>X</td><td>X</td></tr>
<tr><td colspan="14">OF • < ---------<----------------<--------------------------<-----------------------•</td></tr>
</table>

The above table shows that when performing arithmetic or logical operations, if the resulting value falls in the X areas an overflow(OF) will occur. If it falls between the X areas no overflow will occur.

Examples:

1.

AL CODE	AL (decimal)	SIGN
mov al, 127	127	
add al,1	128	
mov sign,’yes’	128	**yes**

Comment: Even though 128 is a value in al, it resulted by adding 1 to 127 and 128 fell in the X area. Therefore, an overflow occurred.

2.

AL CODE	AL (decimal)	SIGN
mov al, 128	128	
sub al,1	127	
mov sign, ‘yes’	127	**yes**

Comment: Even though 127 is a value in al, it resulted by subtracting 1 from 128 and 127 fell in the X area. Therefore, an overflow occurred.

.

3.

AL CODE	AL (decimal)	SIGN
mov al, 127	127	
sub al, 200	83	
mov sign, ‘no’	83	**no**

Comment: Since 83 is a value in al, therefore, no overflow occurred.

4.

AL CODE	AL (decimal)	SIGN
mov al, 255	255	
add al,1	0	
mov sign, ‘no’	0	**no**

Comment: Since 0 is a value in al, therefore, no overflow occurred. .

26.2 EFLAG JUMP INSTRUCTIONS

The eflag bits cannot be directly accessed. However, the following jump instructions can be used to jump to a designated instruction:

Jump Instruction	Result
js	Jump if sign bit is turned on
jns	Jump if sing bit is turned off
jc	Jump if carry bit is turned on
jnc	Jump if carry bit is turned off
jo	Jump if overflow bit is turned on
jno	Jump if overflow bit is turned off

Examples:

1.

AL CODE	AL (decimal)	SIGN
mov al, 254	254	
add al, 3	1	
js label1	1	
mov sign, ‘no’	1	**no**
jmp end	1	no
label1: mov sign, ‘yes’	1	no
end:	1	no

2.

AL CODE	AL (decimal)	SIGN
mov al, 253	253	
add al, 1	154	
js label1	154	
mov sign, ‘no’	154	
jmp end	154	
label1: mov sign, ‘yes’	154	**yes**
end:	154	yes

3.

AL CODE	AL (decimal)	CARRY
mov al, 254	254	
add al, 3	1	
jc label1	1	
mov carry, ‘no’	1	
jmp end	1	
label1: mov carry, ‘yes’	1	**yes**
end:	1	yes

4.

AL CODE	AL (decimal)	OVERFLOW
mov al, 254	254	
add al, 3	1	
jo label1	1	
mov overflow, ‘no’	1	**no**
jmp end	1	no
label1: mov sign, ‘yes’	1	no
end:	1	no

Exercises:

Complete the table below:

AL CODE	AL (decimal)	SIGN	CARRY	OVERFLOW
mov al, 258				
sub al, 7				
js sign				
mov sign, ‘no’				
jmp endsign				
sign: mov sign, ‘yes’				
endsign:				
mov al, 4				
add al, 127				
jc carry				
mov carry, ‘no’				
jmp endcarry				
carry: mov carry, ‘yes’				
endcarry:				
mov al, 254				
sub al, 234				
jof overflow				
mov overflow, ‘no’				
jmp endoverflow				
overflow: mov overflow, ‘yes’				
endoverflow:				

Multiplication

There are 2 types of multiplication operations: mul and imul (See Chapter 10). The mul instruction is when the numbers are considered as unsigned (natural order) and the imul instruction is when the numbers are considered as signed. The mul instruction will set the carry and overflow flags depending on the value of the left most bit. The imul instruction will set the carry if the resulting number is too large. This will result in the edx resgister no being equal to zero.

MUL

Examples:

1.

AL CODE	AL (decimal)	CARRY
mov al, 7	7	
mov x,12	7	
mul x	84	
mov carry, ‘no’	84	**no**

Comment: Since left most bit for 7 is a 0, there is no carry.

2.

AL CODE	AL (decimal)	CARRY
mov al, 128	128	
mov x,2	128	
mul x	0	
mov carry, ‘yes’	0	**yes**

Comment: Since left most bit for 128 is a 1 and after multiplication the left most bit changed to 0, there is a carry .

3.

AL CODE	AL (decimal)	CARRY
mov al, 255	255	
mov x,255	255	
mul x	36	
mov carry, ‘yes’	36	**yes**

Comment: Since left most bit for 255 is a 1 and after the left most bit changed to 0, there is a carry .

4.

AL CODE	AL (decimal)	OVERFLOW
mov al, 255	255	
mov x,255	255	
mul x	36	
mov overflow, ‘yes’	36	**yes**

Comment: Since 255$*$255 mod 256 > 0 , there is an overflow from the natural order .

4.

AL CODE	AL (decimal)	OVERFLOW
mov al, 110	110	
mov x,2	110	
mul x	220	
mov overflow, ‘no’	220	**no**

Comment: Since 110$*$2 mod 256 = 0 , there is no overflow from the natural order .

IMUL

1.

AL CODE	AL (decimal)	CARRY
mov al, 7	7	
mov x,12	7	
imul x	84	
mov carry, ‘no’	84	**no**

Comment: Since left most bit for 7 is a 0, there is no carry.

2.

AL CODE	AL (decimal)	CARRY
mov al, 128	128	
mov x,2	128	
imul x	0	
mov carry, ‘yes’	0	**yes**

Comment: Since left most bit for 128 is a 1 and after multiplication the left most bit changed to 0, there is a carry .

3.

AL CODE	AL (decimal)	CARRY
mov al, 129	129	
mov x,2	129	
imul x	2	
mov carry, ‘yes’	2	**yes**

Comment: Since left most bit for 129 is a 1 and after the left most bit of 2 is 0, there is a carry .

4.

ASSEMBLY CODE	EAX	AX	AH	AL	DX	X
x word 100h						**100**
mov eax 123456h	**00 12 34 56**	**34 56**	**34**	**56**		100
imul x	**00 12 56 00**	**56 00**	**56**	**00**	**34**	100

Comment: Since DX > 0 this resulted in a carry (see example 5, Chapter 10).

5.

AL CODE	AL (decimal)	OVERFLOW
mov al, 2	255	
mov x,255	255	
imul x	36	
mov overflow, ‘yes’	36	**yes**

Comment: The multiplication resulted in moving from 2 past 255 resulting in an overflow.

6.

AL CODE	AL (decimal)	OVERFLOW
mov al, 110	110	
mov x,2	110	
imul x	220	
mov overflow, ‘no’	220	**no**

Comment: Going from 110 to 220 does not result in an overflow. .

Project

Write a procedure in assembly language that will perform the following task:

In a program, assume an operation is performed. Convert the resulting value to its appropriate decimal value.

For example the following code will generate the decimal value 252:

mov al, 4

sub al, 8

However, the correct value in ordinary arithmetic is 4 - 8 = -4.
The subroutine needs to find the ASCII code for the symbol - and 4:

- : 2Dh

4: 34h

The number 2D34 is stored.

The procedure will perform this conversion on any integer number.

CHAPTER 27- NUMERIC APPROXIMATIONS (OPTIONAL)

INTRODUCTION

Numeric approximations play an important rule in assembly language programming. The assembler that you use will provide some numeric algorithms but in most cases the programmer will have to program several necessary numeric algorithms. For, example, at this point we cannot even approximate the square root of a number. Unless the assembler provides a square root approximation algorithm, the programmer will have to write such an algorithm in the usually in the form of a procedure. At this point, in passing, we should note the following additional floating-point instructions that are provided by the 80x86 assembler language:

27.1 Assembler Floating Point Numeric Approximations

The following floating point instructions are provided by the assembler to compute approximations for a specific functions:

1.

MNEMONIC	OPERAND	ACTION
fsin	(none)	Replaces the contents of ST by sin(ST) .

2.

MNEMONIC	OPERAND	ACTION
fcos	(none)	Replaces the contents of ST by cos(ST).

3.

MNEMONIC	OPERAND	ACTION
fsincos	(none)	Replaces the contents of ST by sin(ST), pushes the stack down and then replaces the contents of ST by cos(ST)

4.

MNEMONIC	OPERAND	ACTION
fptan	(none)	Replaces the contents of ST by tan(ST)

5.

MNEMONIC	OPERAND	ACTION
fldpi	(none)	Replaces the contents of ST by π.

6.

MNEMONIC	OPERAND	ACTION
fld12e	(none)	Replaces the contents of ST by $\log_2(e)$.

7.

MNEMONIC	OPERAND	ACTION
fld12t	(none)	Replaces the contents of ST by $\log_2(10)$.

8.

MNEMONIC	OPERAND	ACTION
fldlog2	(none)	Replaces the contents of ST by $\log_{10}(2)$.

9.

MNEMONIC	OPERAND	ACTION
fldln2	(none)	Replaces the contents of ST by $\log_e(2)$.

10.

MNEMONIC	OPERAND	ACTION
fsqrt	(none)	replaces the contents of ST by its square root

27.2 Special Approximations

Although the above are useful, we will need more powerful algorithms that we can call as procedures in our assembly language. We begin with the Newton Interpolation Method.

Newton Interpolation Method

The Newton interpolation method is a powerful method for approximation solving solutions of equations. First we will show, how it can be used to write an algorithm f or computing an approximating the square root of any non-negative number. Then we will apply the Newton's method to approximate the n-root of any appropriate number.

Roots of an equation.

Assume you have an equation $y = f(x)$, represented by the graph below. The root(s) of the equation is (are) the value(s) of x where the graph crosses the x-axis ($f(x) = 0$). . First we start with an initial value x_0 . Next, we compute the tangent line of the curve at x_0. We next find the point x_1 where the tangent line crosses the x-axis. Continuing, we compute the tangent line of the curve at x_1 and we find the point x_2 where the tangent line crosses the x-axis. From he graph we see that t his will lead a sequence to numbers x_0, x_1, x_2, ..., x_n , ... that will converge to one of the roots of the equation.

The Newton interpolation method gives us the following sequential formulas:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

where $f'(x_k)$ are the slopes of the tangent lines.

Using the Newton Interpolation Method to approximate $\sqrt[n]{a}$ of a number where a > 0 .

Assume we wish to approximate the nth root of a number a, $\sqrt[n]{a}$, using Newton's interpolation method. We start by defining f(x) as

$f(x) = x^n - a$

which has a root $\sqrt[n]{a}$

It can be shown that $f'(x) = n x^{n-1}$ which gives use a formula for the slopes of the tangent lines.

We therefore have:

$f(x_k) = x_k^n - a$

$f'(x_k) = nx_k^{n-1}$

$$x_{k+1} = x_k - \frac{x_k^n - a}{nx_k^{n-1}}$$

Example:

Assume we wish to approximate the $\sqrt{5}$ using Newton's approximation method.

Step 1: $f(x) = x^2 - 5$

Step 2: $f'(x) = 2x$

Step 3: $x_{k+1} = x_k - \dfrac{x_k^2 - 5}{2x_k}$; k = 0, 1,2, ...

Step 4: First we set $x_0 = 3$.

$$x_1 = x_0 - \frac{x_0^2 - 5}{2x_0} = 3 - \frac{3^2 - 5}{2(3)} = 3 - 2/3 = 7/3 = 2.333...$$

$$x_2 = x_1 - \frac{x_1^2 - 5}{2x_1} = 2.\overline{3} - \frac{2.\overline{3}^{\,2} - 5}{2(2.\overline{3})} \approx 2.236067978...$$

Since $\sqrt{5} \approx 2.236067978$ is accurate to 8 places we see that if we let $x_2 = 2.236067978...$

will give us at least 8 places of accuracy.

A pseudo-code algorithm for approximating the square root $\sqrt{a}$, where $a \geq 0$.

INSTRUCTIONS	EXPLANATION
X := A + 1	X IS LARGER THAN ROOT OF A.
WHILE N > 0	N IS THE POSITIVE INTEGER
BEGIN	
$X := X - \frac{X^2 - A}{2 * X}$	$x_{k+1} = x_k - \frac{x_k^2 - a}{2x_k}$
N := N - 1	
END	

Exercises:

1. Using the above pseudo-code algorithm for approximating the square root $\sqrt{a}$, where $a \geq 0$, write an assembly language program that will approximate the square root.

2. Modify the above pseudo-code algorithm by replacing the number a by its absolute value.

3. We say that two numbers x, y are at least equal to the nth place if $| x - y | < 1/ 10^n$

For example, the 2 numbers 7.12567890435656 and 7.12567890438905 are at least equal to the 10^{th} place since

$$| 7.12567890435656 - 7.12567890438905 | = 0.00000000003249 < 1/10^{10}$$

4. Modify the above pseudo-code algorithm that will terminate the computation x_{n+1} when

$| x_{n+1} - x_n | < 1/10^n$.

Explain why this would be the better way of estimating the square root $\sqrt{a}$.

5. For problem 4, write an assembly language program.

6. Write a pseudo-code algorithm that will approximate the nth root $\sqrt[n]{a}$.

7. From problem 6, write an assembly language program.

8. Write an assembly language program that will approximate $a^{m/n}$, where m, n are positive integers.

■

Using Polynomials to Approximate Transcendental Functions and Numbers

As you may recall, real polynomials are of the form $a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$, where a_k are real numbers (k = 0,1,...n).

The following important transcendental functions and numbers can often play an important part in any assembly language program:

Transcendental functions:

e^x, ln(x), sin(x), cos(x), $\tan^{-1}(x)$.

Transcendental numbers: e, π.

The following are polynomial approximations of transcendental functions:

$$e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!}, \quad -\infty < x < \infty; \quad n = 0, 1, 2, ...$$

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ... + (-1)^n \frac{x^{2n+1}}{(2n+1)!}, \quad -\infty < x < \infty; \quad n = 0,1,2, ...$$

$$\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... + (-1)^n \frac{x^{2n}}{(2n)!}, \quad -\infty < x < \infty; \quad n = 0,1,2, ...$$

$$\tan^{-1}(x) \approx x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + ... + (-1)^n \frac{x^{2n+1}}{2n+1}, \quad -1 \le x \le 1; \quad n = 0,1,2,...$$

$$\ln(x) \approx -\{1 - x + \frac{(1-x)^2}{2} + \frac{(1-x)^3}{3} + ... + \frac{(1-x)^n}{n}\}; \quad 0 < x < 1; \quad n = 1,2,...$$

$$\ln(x) \approx (1 - \frac{1}{x}) + \frac{1}{2}(1 - \frac{1}{x})^2 + \frac{1}{3}(1 - \frac{1}{x})^3 + \ldots + \frac{1}{n}(1 - \frac{1}{x})^n \quad 1 \le x; \quad n = 1,2,\ldots$$

Using the above approximations, the following transcendental numbers e, π can be approximated:

$$e = e^1 \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}; \quad n = 0,1,2,\ldots$$

$$\frac{\pi}{4} = \tan(1) \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + (-1)^n \frac{1}{2n + 1}; \quad n = 0,1,2,\ldots$$

Therefore,

$$\pi \approx 4\{1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + (-1)^n \frac{1}{2n + 1}\}$$

Pseudo-code Algorithms for Approximating Transcendental Functions and Numbers.

The following pseudo-code algorithm will estimate $e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$:

INSTRUCTIONS	EXPLANATION
K := 0	COUNTER
SUM_EX := 0	WILL SUM POLYNOMIAL
WHILE K ≤ N	WILL COMPUTE N TIMES
BEGIN	
SUM_EX := SUM_EX + X^K/K!	X^K WRITTEN AS A PROCEDURE K! WRITTEN AS A PROCEDURE
K := K + 1	
END	

The following pseudo-code algorithm will estimate :

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^n \frac{x^{2n + 1}}{(2n + 1)!}$$

INSTRUCTIONS	EXPLANATION
K := 0	COUNTER
SUM_SIN := 0	WILL SUM POLYNOMIAL
WHILE K ≤ N	WILL COMPUTE N TIMES

BEGIN	
SUM_SIN := SUM_SIN + $(-1)^{K}*X^{2K+1}/(2K+1)!$	X^K WRITTEN AS A PROCEDURE K! WRITTEN AS A PROCEDURE
K := K + 1	
END	

The following pseudo-code algorithm will estimate :

$$\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots + (-1)^n \frac{x^{2n}}{(2n)!}$$

INSTRUCTIONS	EXPLANATION
K := 0	COUNTER
SUM_COS := 0	WILL SUM POLYNOMIAL
WHILE K $\leq$ N	WILL COMPUTE N TIMES
BEGIN	
SUM_COS := SUM_COS + $(-1)^{K}*X^{2K}/(2K)!$	X^K WRITTEN AS A PROCEDURE K! WRITTEN AS A PROCEDURE
K := K + 1	
END	

The following pseudo-code algorithm will estimate :

$$\tan^{-1}(x) \approx x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots + (-1)^n \frac{x^{2n+1}}{2n+1}$$

INSTRUCTIONS	EXPLANATION
K := 0	COUNTER
SUM_INTAN := 0	WILL SUM POLYNOMIAL
WHILE K $\leq$ N	WILL COMPUTE N TIMES
BEGIN	
SUM_INTAN := SUM_INTAN + $(-1)^{K}*X^{2K+1}/(2K+1)\,)!$	X^K WRITTEN AS A PROCEDURE K! WRITTEN AS A PROCEDURE
K := K + 1	

END	

The following pseudo-code algorithm will estimate :

$$\ln(x) \approx -\{(1 - x) + \frac{(1 - x)^2}{2} + \frac{(1 - x)^3}{3} + \dots + \frac{(1 - x)^n}{n}\}; \quad 0 < x < 1; \quad n = 1,2,\dots$$

$$\ln(x) \approx (1 - \frac{1}{x}) + \frac{1}{2}(1 - \frac{1}{x})^2 + \frac{1}{3}(1 - \frac{1}{x})^3 + \dots + \frac{1}{n}(1 - \frac{1}{x})^n \quad 1 \le x; \quad n = 1,2,\dots$$

INSTRUCTIONS	EXPLANATION
IF 0 < X < 1 THEN	
BEGIN	
K := 1	COUNTER
SUM_LN := 0	WILL SUM POLYNOMIAL
WHILE K ≤ N	WILL COMPUTE N TIMES
BEGIN	
SUM_LN := SUM_LN - $(1 - X)^K$/K	X^K WRITTEN AS A PROCEDURE
K := K + 1	
END	
ELSE	
BEGIN	
SUM_LN := 0	WILL SUM POLYNOMIAL
WHILE K ≤ N	WILL COMPUTE N TIMES
BEGIN	
SUM_LN := SUM_LN + $(1 - 1/X)^K$/K	X^K WRITTEN AS A PROCEDURE
K := K + 1	
END	
END	

Exercises:

1. Using the above algorithm, write a pseudo-code to estimate the number e.

2. Using the above algorithm, write a pseudo-code to estimate the number π.

3. For each of the above algorithms, write an assembly language program.

4. The error created by using the above polynomial approximation is written as

E(x) = transcendental function - polynomial

For the sin(x), cos(x), $\tan^{-1}(x)$ functions, $|E(x)| \le \dfrac{|x|^{n+1}}{n!}$

5. Modify the above algorithms so that the program terminates when $|E(x)| \le 1/10^n$.

Also, write an assembly language program for each of these algorithms.

Monte Carlo Simulations

Monte Carlo simulations solve certain types of problems through the use of random numbers. These problems can be broken down into sampling models which will give us an approximation to the solution of the given problem. In order apply these simulation techniques, we need to develop algorithms that will generate random numbers. In most cases these generated random numbers will have a uniform distribution.

Definition: A uniform distribution of random numbers is a sequence of numbers, where each has equal probability of occurring and the numbers are generated independently of each other.

Example: if we toss a die 100 times, we will generate a sequence of 100 numbers where each number (1,2,3,4,5,6) has the probability of 1/6 of appearing.

Since we have to generate the random sequence internally in the assembler, we cannot generate independently the numbers. The best we can do is generate sequences that correlate very closely to independent uniform distributions. These types of generated sequences are called pseudo random number generators (PRNG).

For our Monte Carlo simulation problems, we will use two types of pseudo random number generators:

- John Von Neumann's Middle Square method

- D.H. Lehmer's Linear Congruence method

John Von Neumann's Middle Square Method

Description This method was very simple: take any given number, square it, and remove the middle digits of the resulting number as your "random number", then use it as the seed for the next iteration. For example, assume we start with the number "seed" number 1111. Squaring the number 1111 would result in 1234321, which we can write as 01234321, an 8-digit number. From this number, we extract the middle 4 digits 2343 as the "random" number. Repeating this process again would give $2343^2 =$ 05489649. Again extracting the middle 4 digits will be 4896. Repeating this process will give a sequence of pseudo random numbers.

To write an assembly language program, we will follow the following steps:

Step 1: Store a 4 digit decimal number into EAX

Step 2: Square this number.

Step 3: Integer divide the number in EAX by 1,000

Step 4: Integer divide the number in EAX by 100000 .

Step 5: Move the remainder in EDX to EAX

Step 6: Repeat Steps 2 - 5

The following partial assembly language program, will perform these steps an undetermined number of times:

ASSEMBLY LANGUAGE	EAX	EDX
mov eax, 6511	**6511**	
mul eax	**42393121**	
div 100	**423931**	**21**
div 10000	**42**	**3931**
mov eax, edx	**3931**	**3931**
(repeat above following instructions)		

Example: The following pseudo-code will simulate the tossing of a die 100 times.

INSTRUCTIONS	EXPLANATION
N:= 100	NUMBER OF TOSSES
EAX := 6511	SEED
LABEL: EAX := EAX* EAX	SQUARE SEED
EAX := EAX/100	
EDX := EAX/10000	SEED
SEED := EDX	
DIE := EDX / 6 + 1	
N := N - 1	COUNT
EAX := SEED	
IF N <> 0 THEN	
BEGIN	
JUMP LABEL	
END	

Exercises:

1. Write a partial assembly language program from the die pseudo-code program.

2. Write a partial assembly language program that will perform the following tasks:

Task1: Toss a die 100 times

Task2: compute the number of times the number 6 occurs.

3. Write a partial assembly language program that will perform the following tasks:

Task1: Toss a pair of dice 100 times.

Task2: Sum the resulting numbers for each toss.

Task3: Compute the number of times the number 7 occurs.

4. Write a partial assembly language program that will perform the following tasks:

Task1: Toss a coin 100 times

Task2: Count the number of times "heads" appear.

5. Write a partial assembly language program that will compute 100 random numbers x where $0 \le x \le 1$.
■

D.H. Lehmer's Linear Congruence Method

The linear congruence method for generating pseudo random numbers uses the linear recurrence relation:

$x_{n+1} = ax_n + b \pmod{m}$ where n = 0,1,2,....

Lehmer proposed the following values:

$m = 10^8 + 1$
$a = 23$
$b = 0$
$x_0 = 47594118$

These values will result a repetition period of 5,882,352

Using these values the following partial program will compute a undermined number of random numbers x where $0 \le x \le 10^8 + 1$:

ASSEMBLY LANGUAGE CODE
mov m, 100000001; number m = $10^8 + 1$
mov x , 47594118

mov a, 23
mov eax, x
mul a
div m;   remainder stored in edx
mov eax, edx
mul a
div m
::::::::::::::
(repeat the above in bold)

Monte Carlo Approximations

Random sampling from a population can be applied in solving simple and complex mathematics and scientific problems . This type of applications are known as Monte Carlos approximations. To best illustrate this method, assume we wish to approximate by random sampling the number π. One method is to use a unit square that contains a circle of radius 1.

We know that the area of a circle of radius 1 is π. However, for simplicity we will only examine one quadrant as shown in the figure below , where r = 1 and the area is $\pi/4$.

The following steps will approximation π.

Step 1: Generate a pair of random numbers (x,y) where $0 \leq x, y \leq 1$. To generate these numbers we will use linear congruence method in the following form:

$x_{n+1} = a_1 x_n + b_1 \pmod{m_1}$ where n = 0,1,2,....

$y_{n+1} = a_2 \; y_n + b_2 \pmod{m_2}$ where n = 0,1,2,....

$x = x_{n+1}/m_1$

$y = y_{n+1}/m_2$

Step 2: If $x^2 + y^2 \leq 1$ then (x,y) lies in the circle of the first quadrant and we will assume success.

Step 3: Generating N pairs (x,y), the law of large number states that #successes/N $\Rightarrow$ $\pi/4$, for large values of N.

The following pseudo-code algorithm will perform this sampling and approximate π:

INSTRUCTIONS
K := 1
SUCCESS := 0
WHILE K ≤ N
BEGIN
X := (A1*X + B1) MOD M1
Y := (A2*Y + B2) MOD M2
IF $(X^2 + Y^2) \leq 1$ THEN
BEGIN
SUCCESS := SUCCESS + 1
END
K := K + 1
END
PIE := 4*(SUCCESS/ N)

Exercises:

1. From the above pseudo-code algorithm, write a assembly language algorithm.

2. To test the above assembly language algorithm, write a assembly language program for different values of m, a, b.
■

3. The Gambler's Ruin

Assume a gambler with initial capital of n dollars plays against game against a casino. Assume the following rules of the game:

- For each bet, he bets one dollar
- The gambler will play until he wins m dollars where $m > n$ or goes broke.
- For each bet, the gambler's chance of winning is p where $0 < p < 1$.

For different values of p, write a assembly language program that will compute the number of times he bets.
■

PROJECT *Bose-Einstein Statistics.*

In physics, the Bose-Einstein statistics deals with the number of ways of placing m indistinguishable particles into n distinguishable cells. This is analogous of placing m indistinguishable balls into n distinguishable urns.

The number of distinguishable arrangement is

$\binom{n+m-1}{m}$, where each distinguishable arrangement has equal probability.

Assume that m < n. Write a assembly language program, using Monte Carlo approximations, that will approximate the probability that at each cell has at most one particle.

Note: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$,

About The Author

Howard Dachslager received a Ph.D. in mathematics from the University of California, Berkeley where he specialized in real analysis and probability theory. Prior to beginning his doctoral studies at the University of California, Berkeley, he earned a masters degree in economics from the University of Wisconsin.

After graduation from the University of Wisconsin in 1956, he went to work for Remington Rand Co. as a machine language program. For the next two years he worked on various mathematical applications such as missile guidance systems, and tracking systems of naval sea vessels. In 1958, he was admitted as a graduate student to the department of mathematics, UC Berkeley. To finance his education, he worked for the first year as a programmer and programming consultant for the astronomy department, at UC Berkeley. During that year he also work, during the summer, as a machine language programmer for Lockheed Corp., Palo Alto, Calif.

His main duty was to find and correct errors in existing programs. Starting his second year at UC Berkeley, he received a teaching assistantship in the mathematics department. His main duties was to teach courses in numerical analysis and programming. He also worked several professors in this field. .

Since completing his Ph.D. in mathematics, he has taught mathematics and programming to a diverse student population on many levels. As a faculty member of the Department of Mathematics at the University of Toronto he prepared and presented undergraduate level courses in mathematics. Later he returned to the mathematics and computer science department, UC Berkeley, where he taught for several years undergraduate mathematics and programming courses.

While working in the State Department's Alliance for Progress program, he taught advanced mathematics courses at a statistics institute in Santiago, Chile. Other teaching experience includes presenting undergraduate and community college mathematics courses.

Throughout his teaching career in mathematics and computer science , he has always attempted to find and use the most effective teaching methodologies to communicate an understanding mathematics and programming. . Unable to find an appropriate text for use in his courses in assembly language programming, and drawing on his own extensive teaching experience, education and training, he developed an assembly language text that has significantly improved the understanding and performance of students in this language..

APPENDEX

To support Assembly language programming in Visual Studio 2005/2008 or Visual C++ Express Edition 2005/2008 you will need to configure the project properties by following the instructions provided.

Visual C++ Express Edition 2008 is available for free from Microsoft and includes both SP1 for Visual C++ Express Edition and MASM 9.0 in the installation of Visual C++ Express Edition 2008 to support assembly language program development.

If you are using Visual Studio 2005:

If you have not already done so, install SP1 for Visual Studio 2005. MASM 8.0 will be installed with this service pack.

If you already have Visual C++ Express Edition 2005 installed but have not installed MASM 8.0, download and install the following resources:

SP 1 for Visual C++ Express Edition 2005:

- Search Microsoft.com for file "VS80sp1-KB926748-X86-INTL.exe"
- Select Download details: Visual Studio® 2005 Express Editions SP1 from search results
- Follow instructions to download/install file VS80sp1-KB926748-X86-INTL.exe

MASM 8.0:

- Search at Microsoft.com for "MASM 8.0"
- Select Download Details: Microsoft Macro Assembler 8.0 (MASM) Package (x86) from search results
- Follow download/installation instructions

If you have installed Visual Studio 2008, MASM 9.0 will also have been installed and you may begin.

If you wish to use Visual C++ Express Edition 2008:

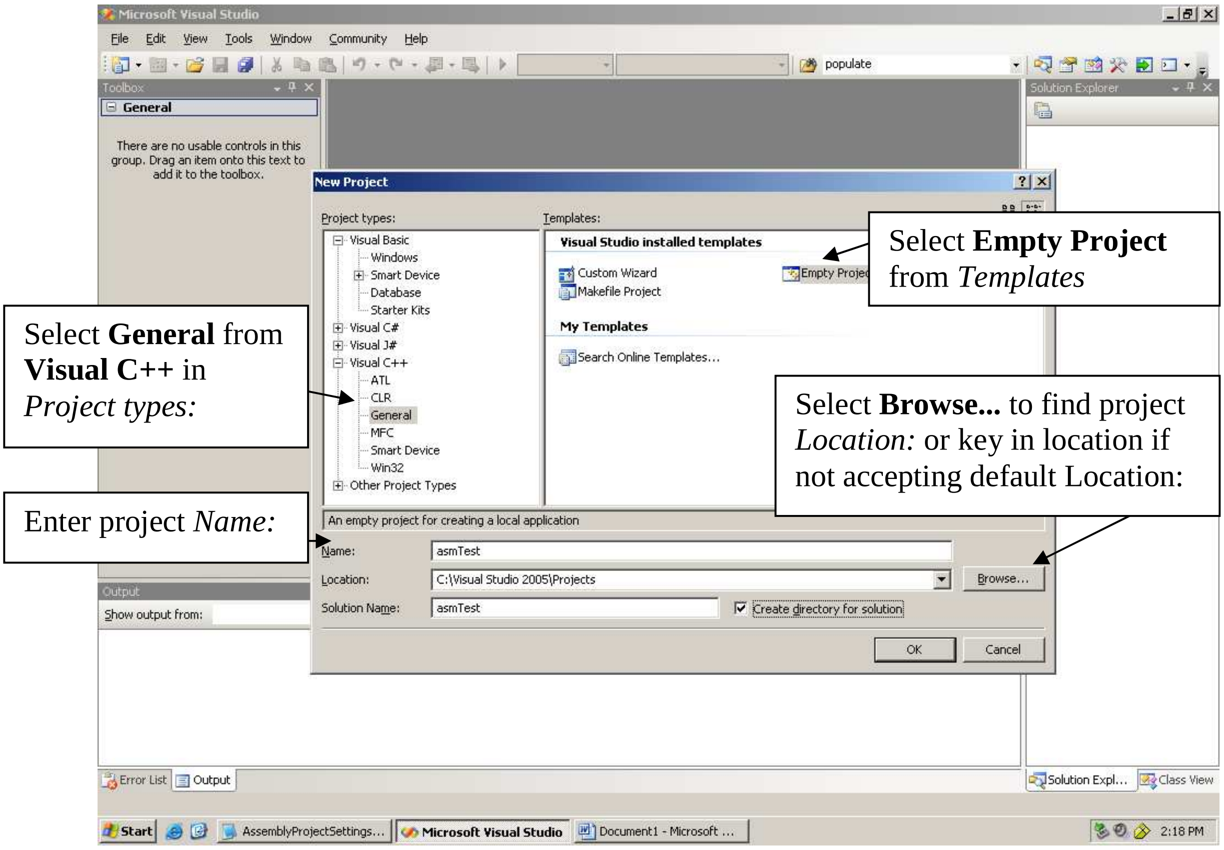
- Navigate to http://www.microsoft.com/express/download/
- Select Visual C++ Express Edition 2008. Both SP1 and MASM 9.0 will automatically install with this edition.

Start Visual Studio

If you are using Visual C++ Express Edition 2005/2008 select File from the menu, then select New, then select Project.

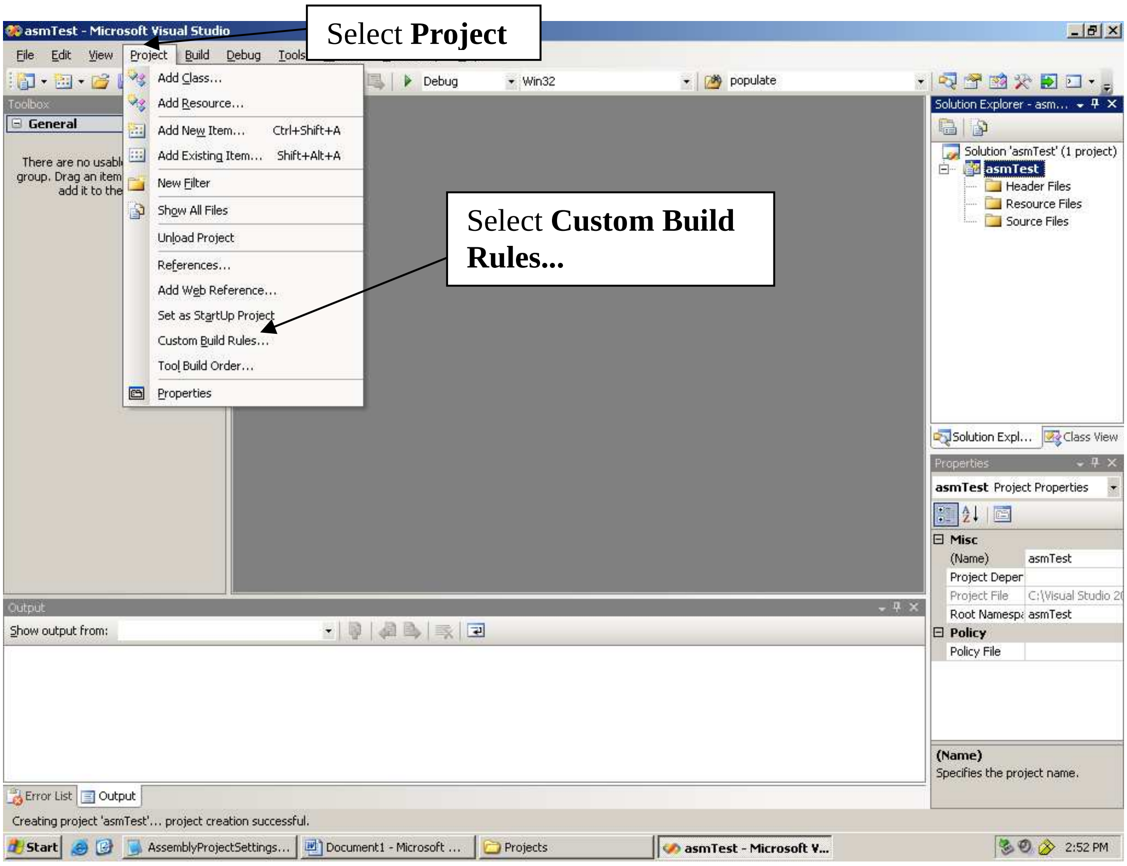
When the *New Project* dialogue box appears, select or enter the following:

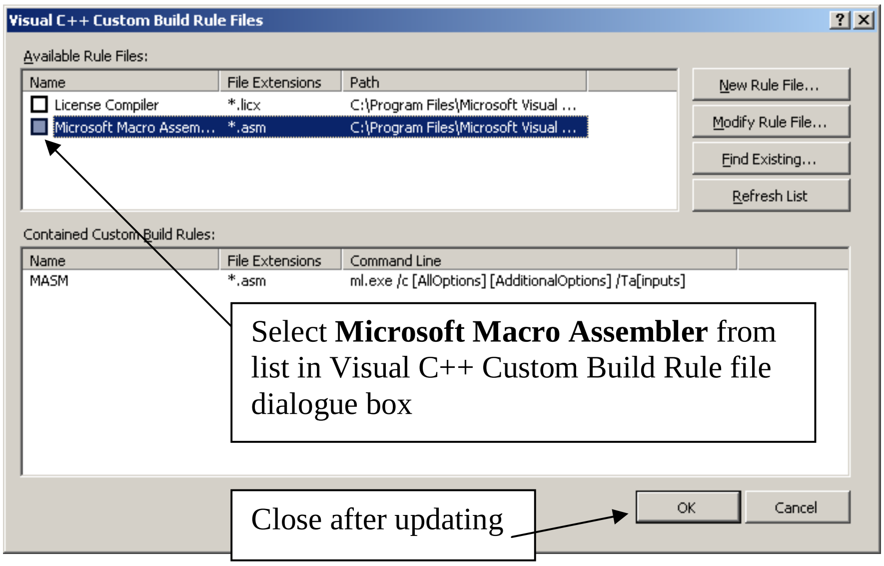
> Select **General** from **Visual C++** in *Project types:*
> Select **Empty Project** from *Templates*
> Enter project *Name:*
> Select **Browse...** to find project *Location:* or key in location if not accepting default Location:
> Close *New Project* Dialogue box

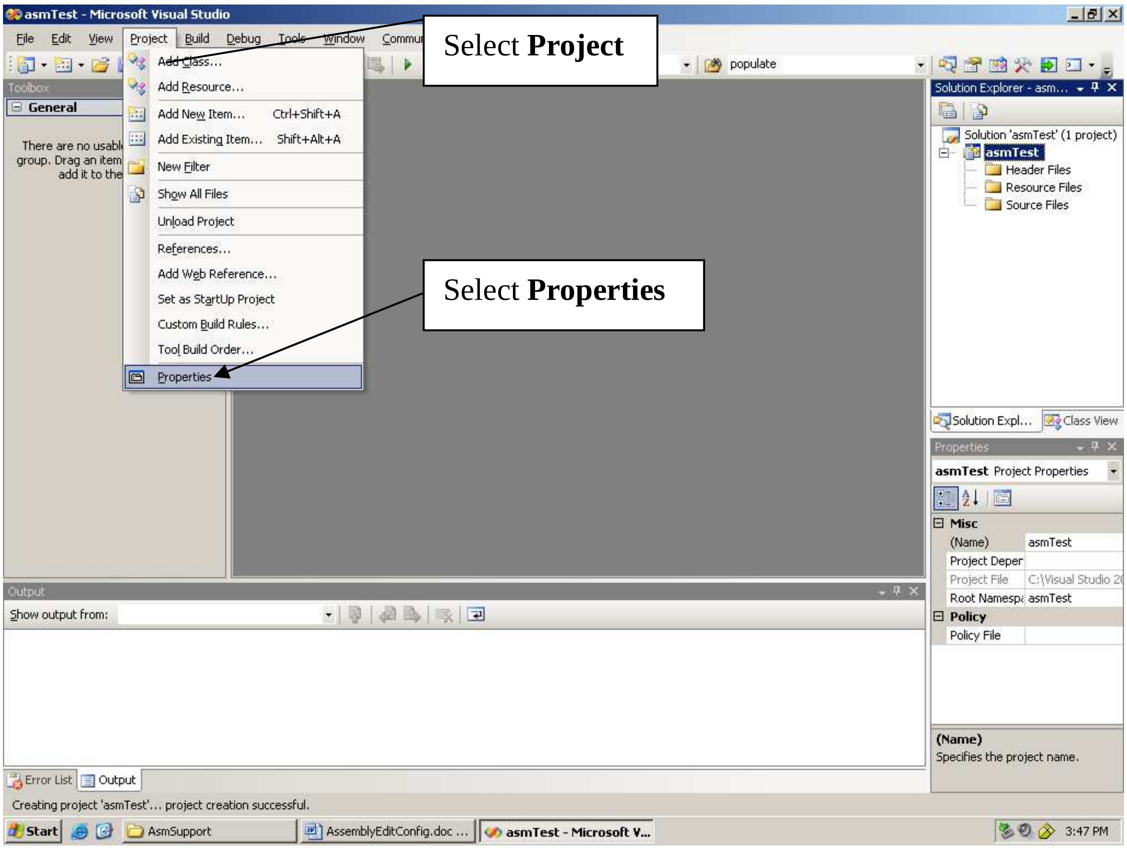


From the VS Editor main menu:

Select **Project**
Select **Custom Build Rules…**
Select **Microsoft Macro Assembler** from list in Visual C++ Custom Build Rule file dialogue box

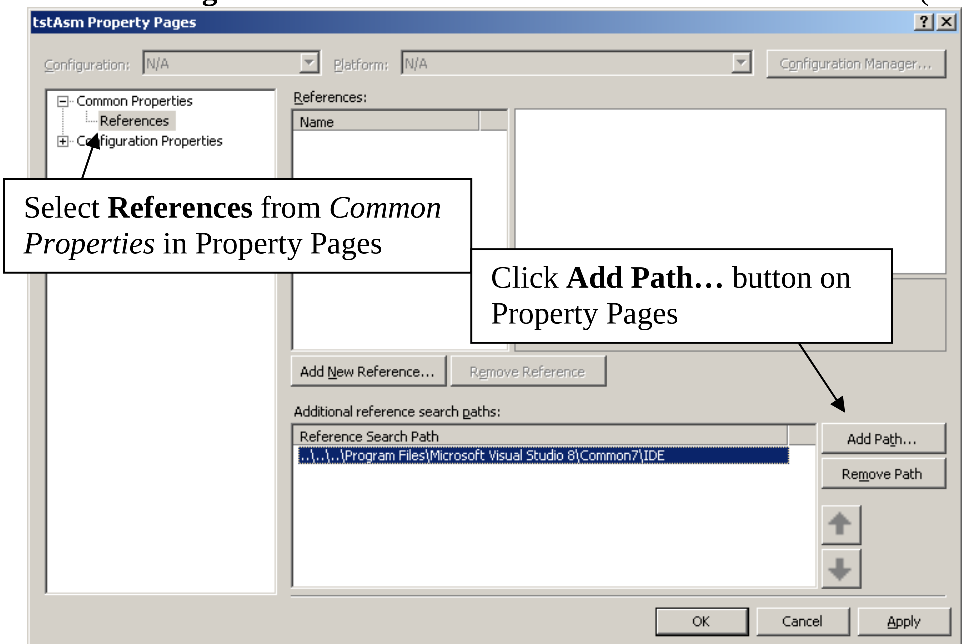


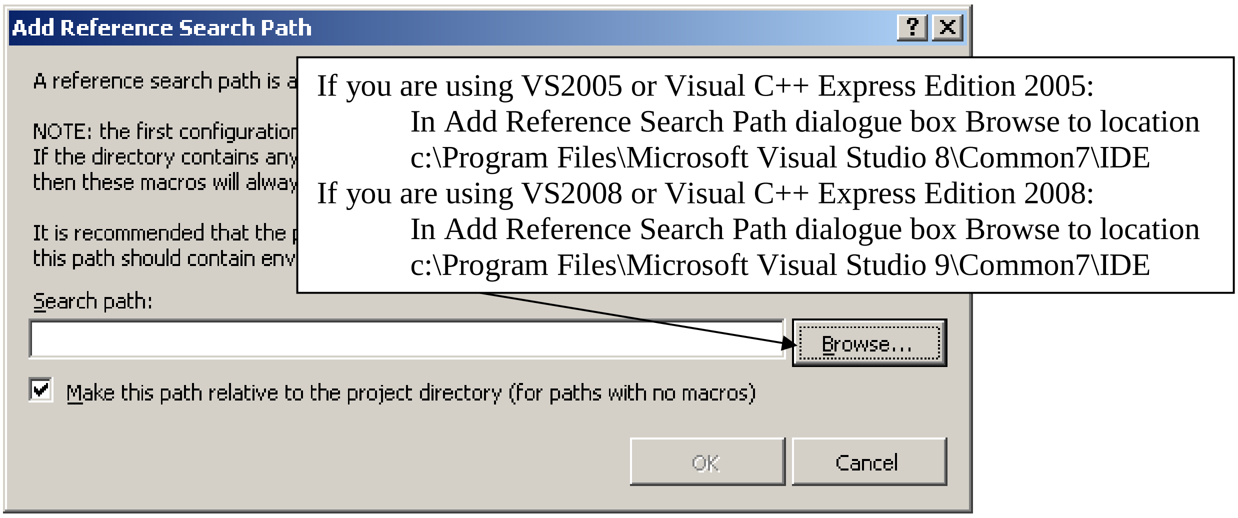




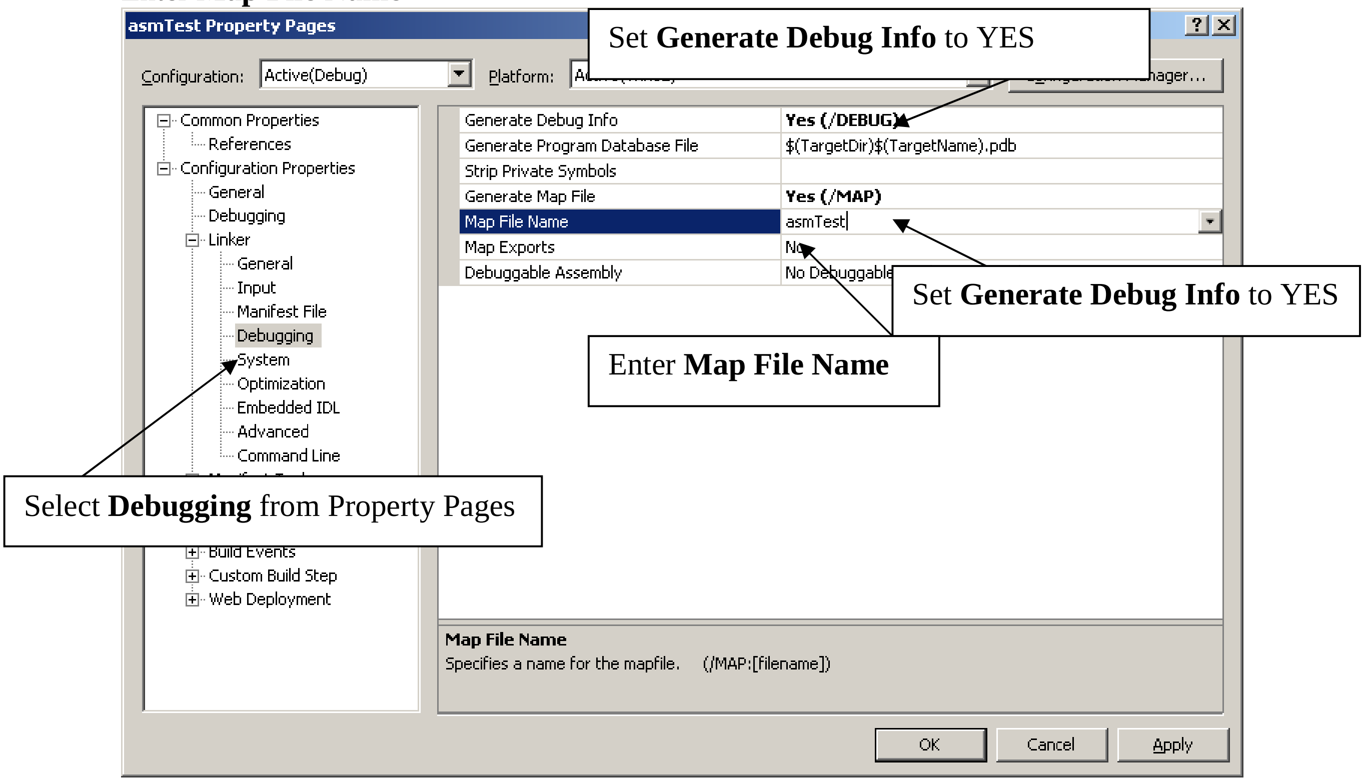
Select **References** from *Common Properties* in Property Pages
Click **Add Path…** button on Property Pages
When Add Reference Search Path dialogue box appears, browse to the appropriate location for Visual Studio 2005 or Visual Studio 2008 editions.

c:\Program Files\Microsoft Visual Studio 8\Common7\IDE (VS2005 editions)
c:\Program Files\Microsoft Visual Studio 9\Common7\IDE (VS2008 editions)

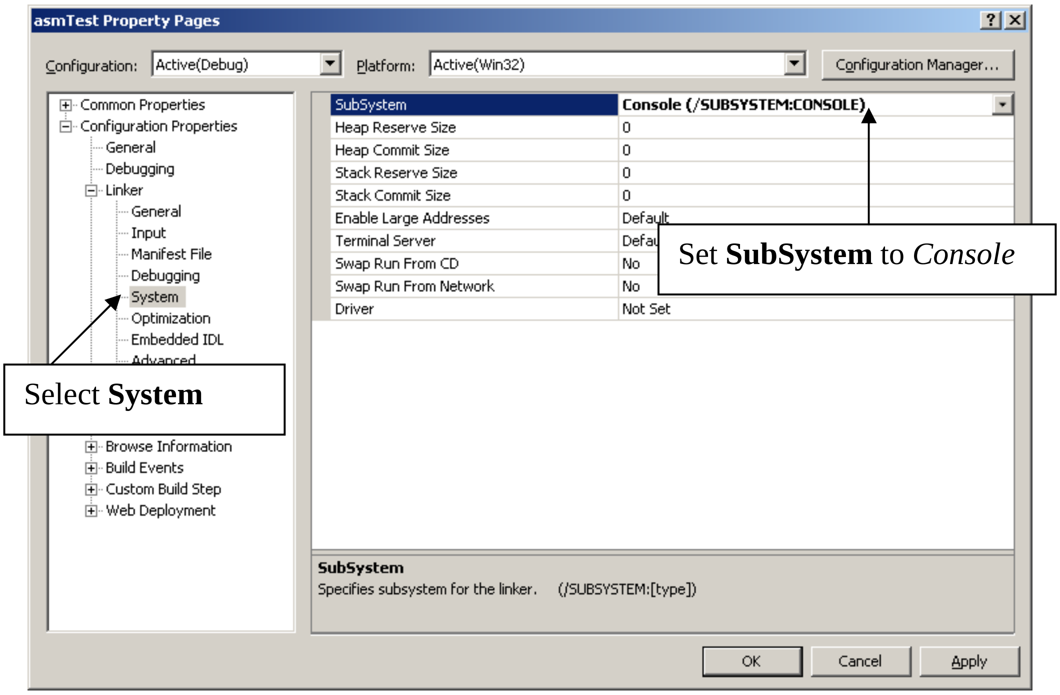




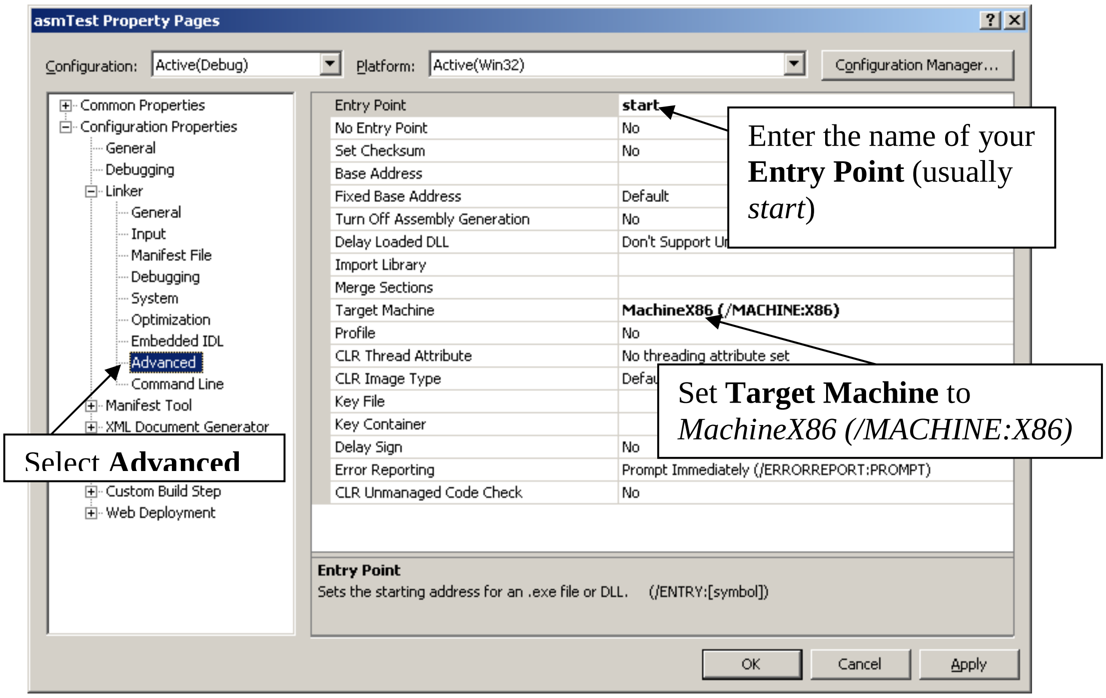
Select **Debugging** from Linker menu on Property Pages
Set **Generate Debug Info** to YES
Set **Generate Map File** to YES
Enter **Map File Name**



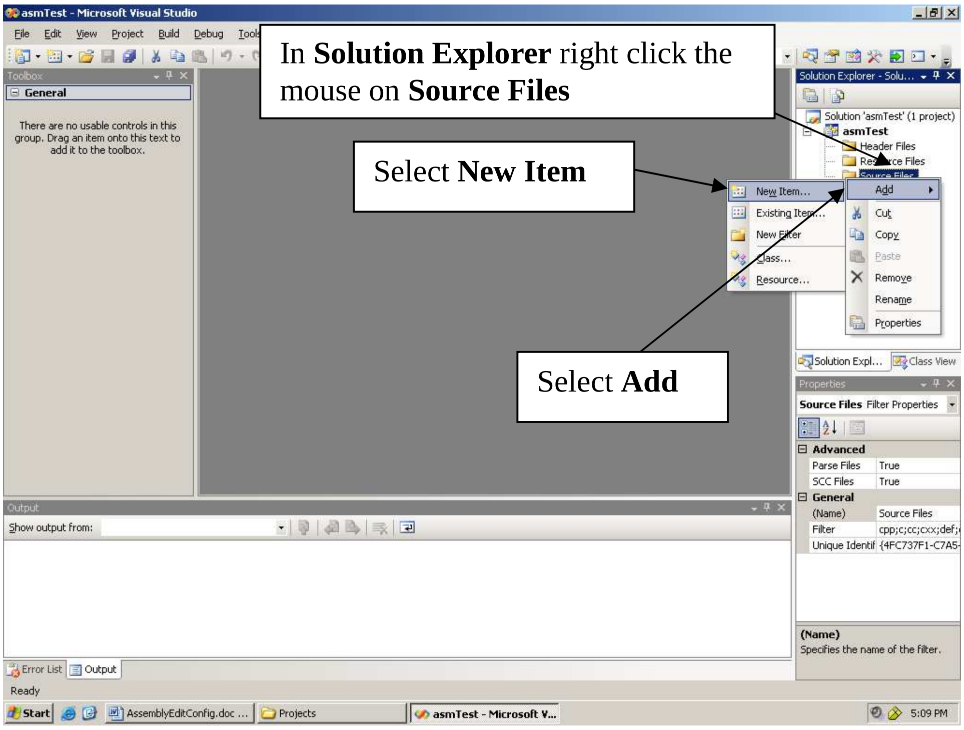
Select **System** in Property Pages
Set **SubSystem** to *Console*



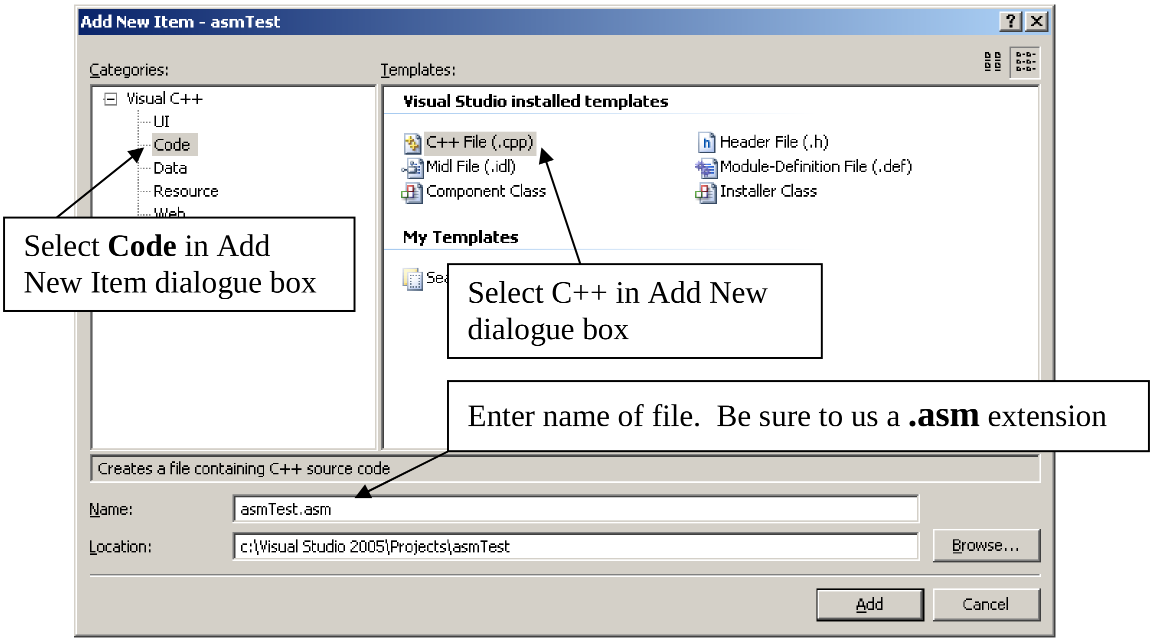
Select **Advanced**
Enter the name of your **Entry Point** (usually *start*)
Set **Target Machine** to *MachineX86 (/MACHINE:X86)*



In **Solution Explorer** right click the mouse on **Source Files**
Select **Add**
Select **New Item**



Select **Code** in Add New Item dialogue box
Select C++ in Add New dialogue box
Enter name of file. Be sure to us a **.asm** extension



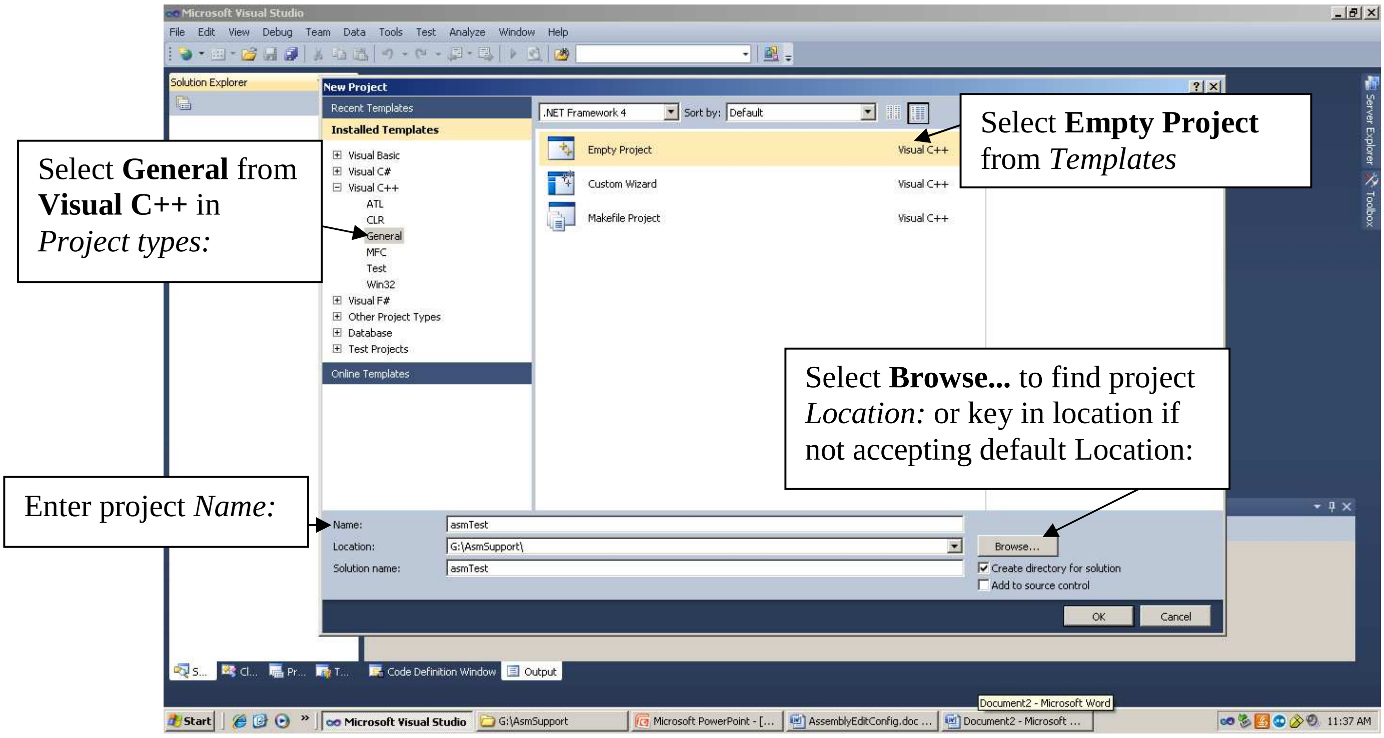
APPENDEX

Start Visual Studio 2010

Select File from the menu, then select New Project.

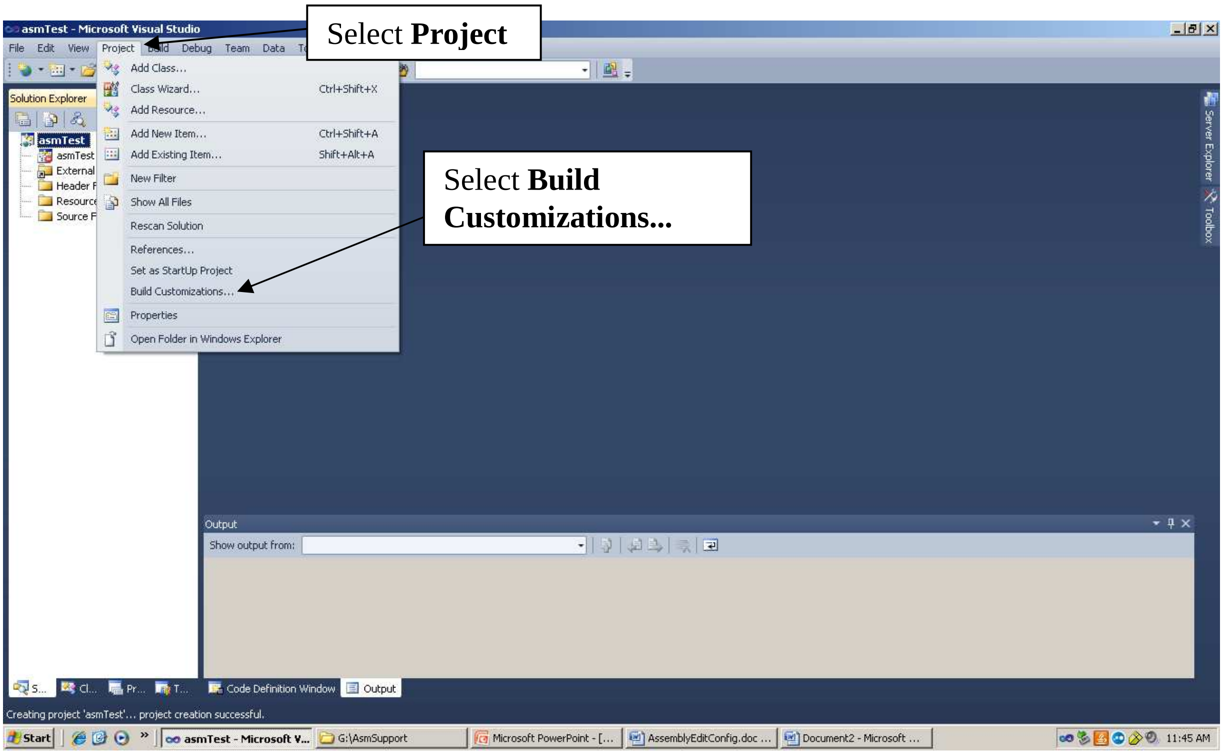
When the *New Project* dialogue box appears, select or enter the following:

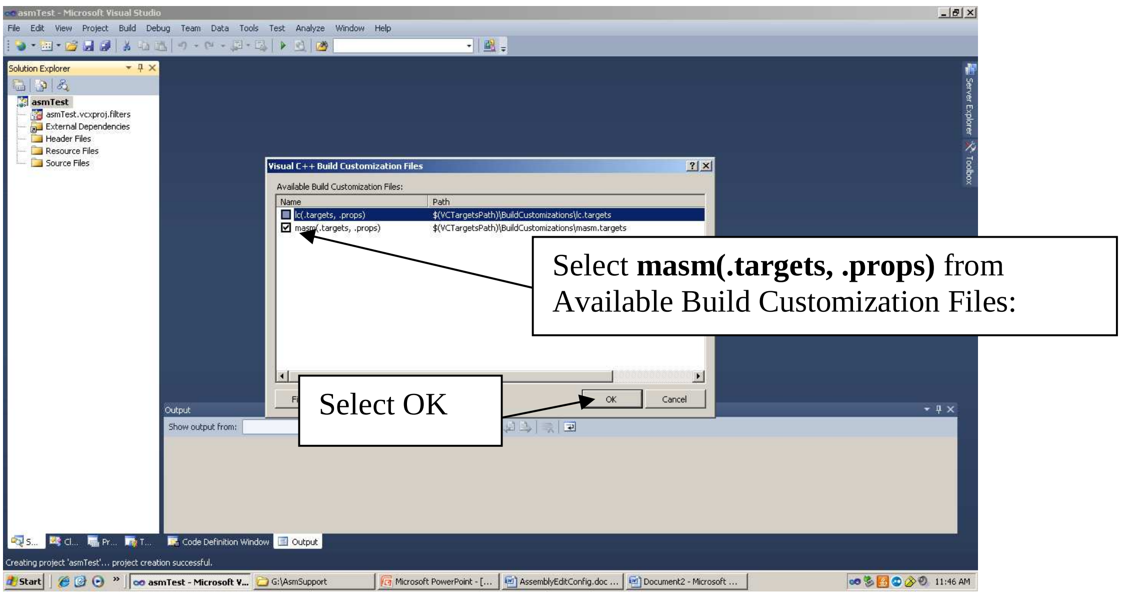
> Select **General** from **Visual C++** in *Installed Templates:*
> Select **Empty Project**
> Enter project *Name:*
> Select **Browse...** to find project *Location:* or key in location if not accepting default Location:
> Select OK in *New Project* Dialogue box

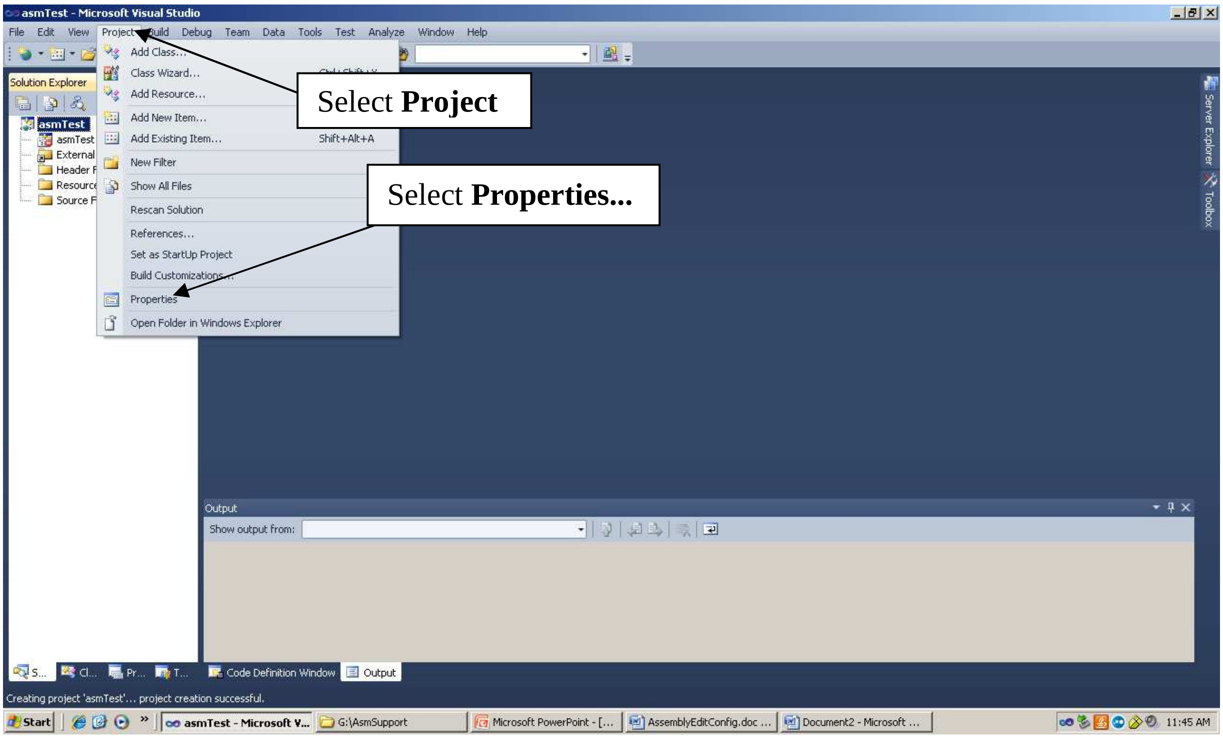


From the VS Editor main menu:

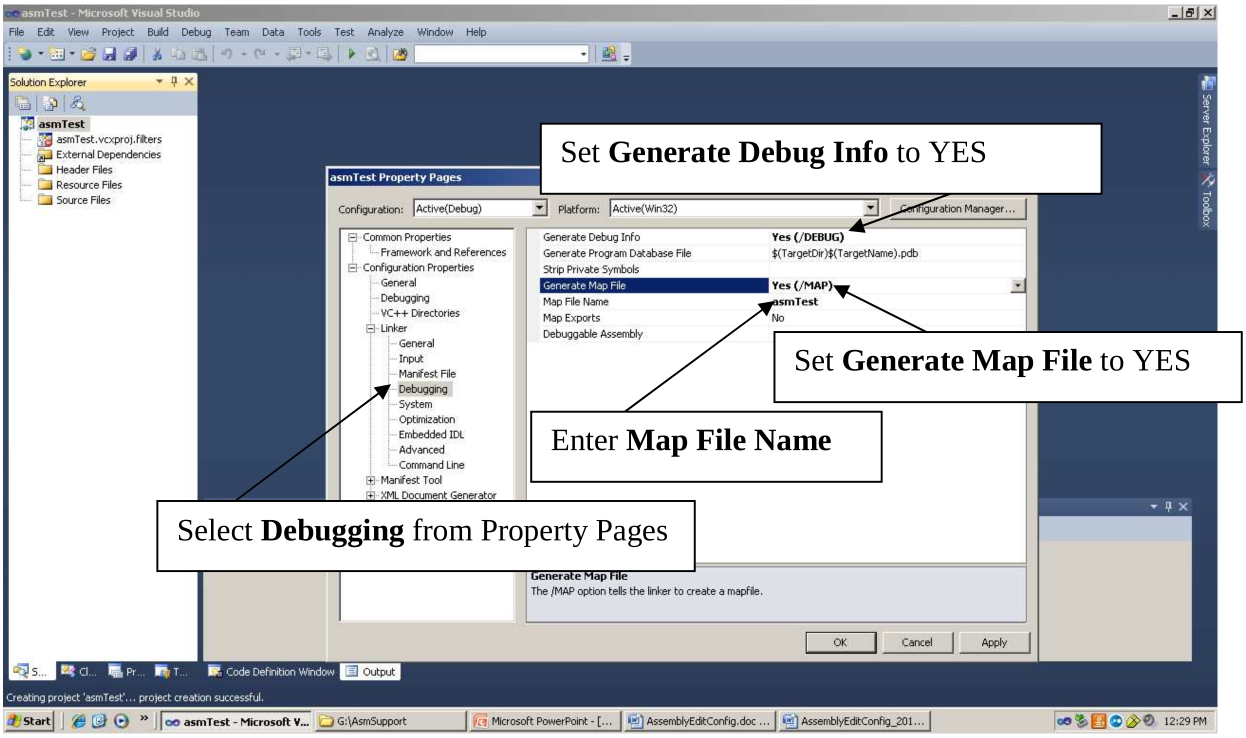
Select **Project**
Select **Build Customizations…**
Select **masm(.targets, .props)** from Available Build Customization Files:



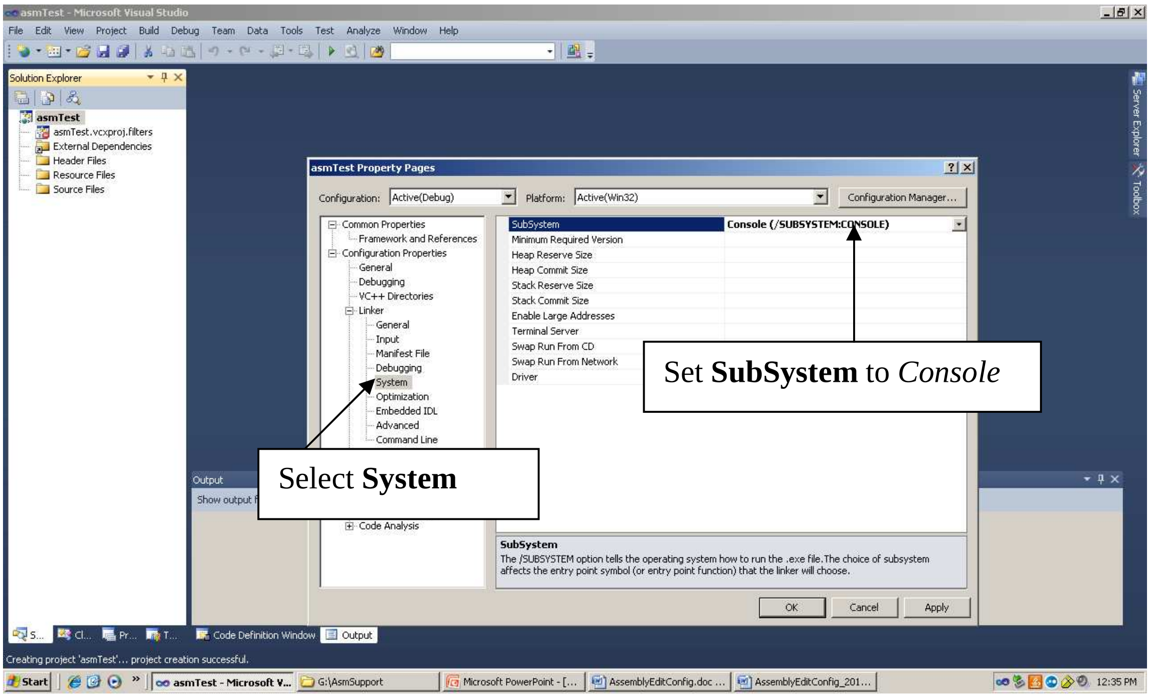




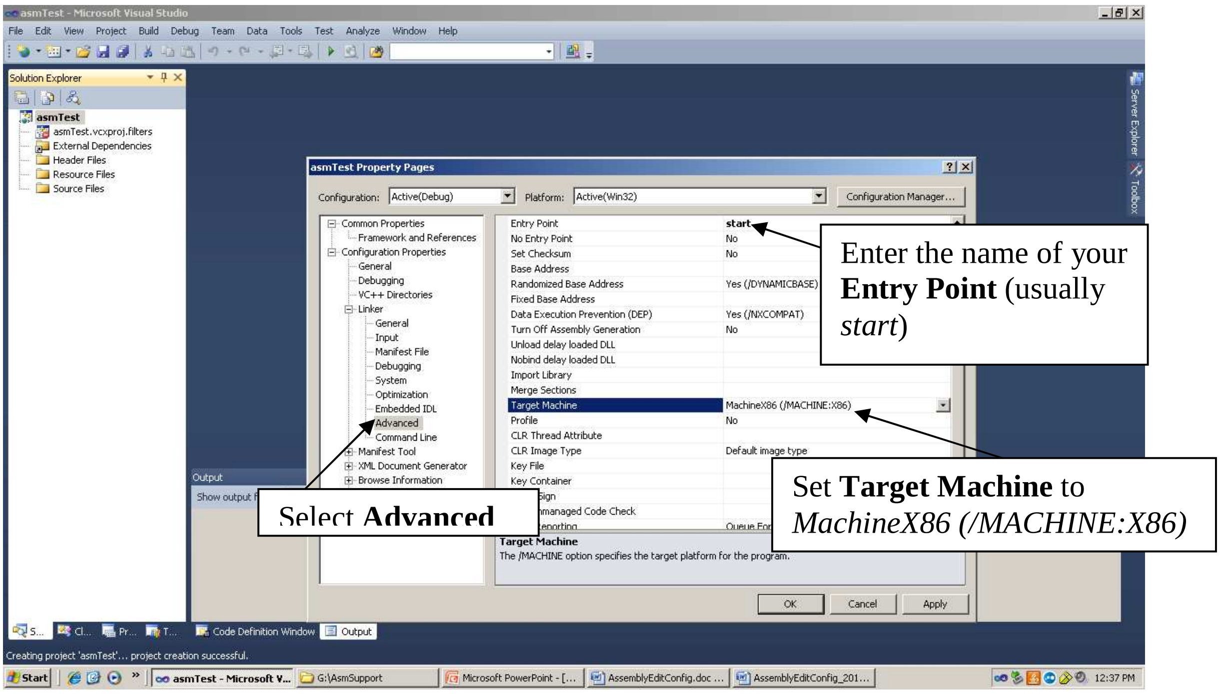
Select **Debugging** from Linker menu on Property Pages
Set **Generate Debug Info** to YES
Set **Generate Map File** to YES
Enter **Map File Name**



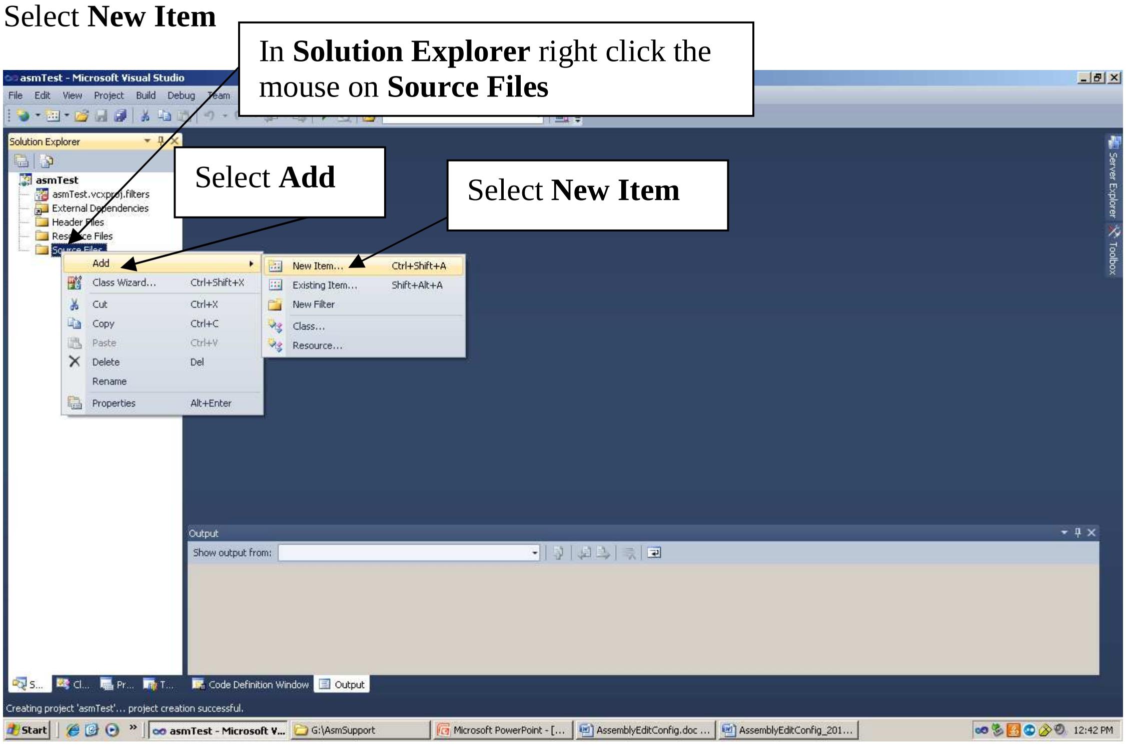
Select **System** in Property Pages
Set **SubSystem** to *Console*



Select **Advanced**
Enter the name of your **Entry Point** (usually *start*)
Set **Target Machine** to *MachineX86 (/MACHINE:X86)*



In **Solution Explorer** right click the mouse on **Source Files**
Select **Add**
Select **New Item**



Select **Code** in Add New Item dialogue box
Select C++ in Add New dialogue box
Enter name of file. Be sure to us a **.asm** extension