Experiment 2

2 Assembly Language Structure and Data Representation

Introduction

In this experiment, students are exposed to the general structure of an assembly language program, and deal with data representation of in a computer system.

Objectives:

- Assembly language program structure
- Instructions and Directives
- Data representation: Variable declaration and Constant definition
- Some basic assembly instructions

2.1 Structure of an Assembly Language Program

An assembly language program is a sequence of instructions and directives. Only one statement is written per line. An assembly language program has the structure shown in the following table:

TITLE "Optional: Write here the Title of your program"
.MODEL SMALL
This directive defines the memory model used in the program.

.STACK
This directive specifies the memory space reserved for the stack

.DATA
Assembler directive that reserves a memory space for constants and variables

.CODE
Assembler directive that defines the program instructions

END
Assembler directive that finishes the assembler program

Table 2.1: Assembly Language Program Structure

2.1.1 Program Statement

A line of a program is generally known as a statement. A statement contains the fields as shown in the syntax below. Some of the fields are optional.

Name Operation operand(s) ;comment

2.2 Instructions and Directives

There exist two types of statements used in assembly language programming: Instructions and Directives.

2.2.1 Instruction

An instruction is meant for the processor. The format of an assembly instruction closely mirrors the structure of a machine instruction. The assembler translates this instruction into machine code

Example

```
label_1     MOV AX, BX      ; Load AX to prepare for multiplication
            ADD AX, MEM16   ; AX = AX + MEM16
```

2.2.2 Directive

Pseudo-instructions or **assembler directives** are instructions that are directed to the assembler. They will affect the machine code generated by the assembler and are not be translated into machine code. Directives are used to declare variables, constants, segments, macros, and procedures as well as supporting conditional assembly

Model Directive

The model directive determines the size of the code, stack and data segments of the program. Each of the segments is called a logical segment. Depending on the model used, the code and data segments may be in the same or in different physical segments as shown in table 2.2.

In most programs, the model small is sufficient. The tiny model is usually used to generate **command** files (files with extension **.com**). This type of files is smaller in size than the executable files with extension **.exe**.

Stack Directive

The stack directive is used to declare the stack segment. The stack is used to temporarily save registers or variable contents. The value after the stack directive tells the program how many bytes are initially reserved for the stack. The stack directive should be used even if the program itself does not use the stack, as the latter is needed for subroutine calling (return address) and possibly passing parameters. More details will be given on the stack when considering procedures and interrupts.

Data Directive

At this level, all variables must be declared and constants must be defined. Variables are declared using: DB, DW ,… etc, while constants are defined using: the directive **equ**.

Memory Model	Size of Code and Data		
	Code	**Data**	**Note**
TINY	= 64KB	= 64KB	Code + Data = 64KB
SMALL	= 64KB	= 64KB	
MEDIUM	may be = 64KB	= 64KB	
COMPACT	= 64KB	may be = 64KB	
LARGE	may be = 64KB	may be = 64KB	no array = 64KB
HUGE	may be = 64KB	may be = 64KB	arrays can be = 64KB

Table 2.2: Memory Models

Code Directive

The directive **.code** is used to declare the code segment. Any program code resides at this level.

End of Program Directive

The Directive **End** is used to tell the assembler that this is the end of the program source file. One has to make sure that no more instructions appear after this directive.

STARTUP and EXIT Directives

The following sequence of instructions is always used at the beginning of a program to force the system assume the right data segment:

```
MOV AX, @DATA
MOV DS, AX
```

This sequence may be replaced by the **.STARTUP** directive which assigns the right DATA segment and hence the assembler will issue no warning. However, it should be noted that your code will start at address **CS:0017H**. The Startup directive occupies the bytes CS:0000 to CS:0017H.

Identically, the sequence used to terminate and exit to DOS:

```
MOV AH, 4CH
INT 21H
```

can be replaced by the **.EXIT** directive.

2.3 Data Representation

Two main types of data commonly used in programming: *numbers* and *characters*.

Numbers: Three numbering systems are used in assembly programming: binary, decimal and hexadecimal. Signed numbers are represented using 2's complement notation

Legal number representations		**Illegal number representations**	
Number	**Type**	**Number**	**Reason for being illegal**
11011B	binary	1,234	contains a non-digit character
11011	decimal	1B4D	hex number not ending with H
64223	decimal	FFFFH	hex number not beginning with a digit
-21843D	decimal	11_90	contains a non-digit character
1B4DH	hexadecimal number	119A	decimal with a non-digit character
0FFFFH	hexadecimal number	0A,34H	contains a non-digit character

Table 2.3: Number Representation

Characters: A character, or string of characters, must be enclosed in single or double quotes: e.g. "Hello", 'Hello', "A", 'B'. Characters are encoded using ASCII code.

Examples:

- 'A' has ASCII code 41H
- 'a' has ASCII code 61H
- '0' has ASCII code 30H
- Line feed has ASCII code 0AH
- Carriage Return has ASCII code 0DH
- Back Space has ASCII code 08H
- Horizontal tab has ASCII code 09H

Note:

The value of a variable, the content of registers or memory is based on the programmer interpretation:

AL = FFH	represents the unsigned number 255 represents the signed number -1 (in 2's complement)
AH = 30H	represents the decimal number 48 represents the character '0'
BL = 80H	represents the unsigned number +128 represents the signed number -128

2.4 Variable Declaration

Each variable has a type. Based on its definition, a variable is assigned a memory location. The location is defined by its address and number of bytes. Different data definition directives are used for different data types and variable sizes.

Directive	Effect
DB	define byte
DW	define word
DD	define double word (two consecutive words)
DQ	define quad word (four consecutive words)
DT	define ten bytes (five consecutive words)

Table 2.4: Data Declaration Directives

Each pseudo-op can be used to define one or more data items of given type. DQ and DT are mostly used to declare variables in floating point format.

2.4.1 Byte Variables

The following directive defines a variable of size byte**:**

Var_name DB initial_value

a question mark (?) instead of 'initial_value' leaves the variable non-initialized. However, the assembler will assume the default value in memory, and will not generate a "non initialized variable" warning as in high level languages.

Examples

a. I DB 4 ; define variable I with initial value 4
b. J DB ? ; define variable J with no initial value
c. Name DB "Course" ; allocate 6 bytes for the variable Name
d. K DB 5, 3, -1 ; allocates 3 bytes, as shown in the table

K →	**05**
	03
	FF

2.4.2 Word Variables

The following directive defines a variable of size word**:**

Var_name DW initial value

I DW 4	**I →**	**04**
		00
J DW -2	**J →**	**FE**
		FF
K DW 1ABCH	**K →**	**BC**
		1A
L DW "01"	**L →**	**31**
		30

2.4.3 Double Word Variables

The following directive defines a variable of size double word**:**

Var_name DD initial value

I DD 1FE2AB20H	**I ➔**	**20**
		AB
		E2
		1F
J DD -4	**J ➔**	**FE**
		FF
		FF
		FF

2.5 Constant Definition:

The use of constants makes assembly language programs easier to understand. The EQU directive is used to assign a name to a constant:

Cst_name EQU Cst_Value

No memory is allocated for constants defined using the EQU directive. However, the constant will be replaced by its value during assembly time.

Examples

```
Declaration:  LF      EQU   0AH
Instruction:  MOV     DL, LF
Code View:    MOV     DL, 0AH

Declaration:  PROMPT  EQU   "Type your name"
Directive:    MSG     DB    PROMPT
```

2.6 ASCII Table

The ASCII table is a double entry table, which contains all alphanumeric characters and symbols. The hexadecimal ASCII code of a given character is found by concatenating the column number with the row number. The row number is the least significant digit. For the same code to be expressed in decimal, the row number is added to the column number.

binary	MSN	0000	0001	0010	0011	0100	0101	0110	0111
LSN	hex	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0000	*0*	**NUL**	**DLE**	**SP**	**0**	**@**	**P**	**`**	**p**
0001	*1*	**SOH**	**XON**	**!**	**1**	**A**	**Q**	**a**	**q**
0010	*2*	**STX**	**DC2**	**"**	**2**	**B**	**R**	**b**	**r**
0011	*3*	**ETX**	**XOFF**	**#**	**3**	**C**	**S**	**c**	**s**
0100	*4*	**EOT**	**DC4**	**$**	**4**	**D**	**T**	**d**	**t**
0101	*5*	**ENQ**	**NAK**	**%**	**5**	**E**	**U**	**e**	**u**
0110	*6*	**ACK**	**SYN**	**&**	**6**	**F**	**V**	**f**	**v**
0111	*7*	**BEL**	**ETB**	**'**	**7**	**G**	**W**	**g**	**w**
1000	*8*	**BS**	**CAN**	**(**	**8**	**H**	**X**	**h**	**x**
1001	*9*	**HT**	**EM**	**)**	**9**	**I**	**Y**	**i**	**y**
1010	*A*	**LF**	**SUB**	*****	**:**	**J**	**Z**	**j**	**z**
1011	*B*	**VT**	**ESC**	**+**	**;**	**K**	**[**	**k**	**{**
1100	*C*	**FF**	**FS**	**,**	**<**	**L**	****	**l**	**|**
1101	*D*	**CR**	**GS**	**-**	**=**	**M**	**]**	**m**	**}**
1110	*E*	**SO**	**RS**	**.**	**>**	**N**	**^**	**n**	**~**
1111	*F*	**SI**	**US**	**/**	**?**	**O**	**_**	**o**	**DEL**

Table 2.4: The ASCII table

Example on the use of the ASCII table

Character	Column #	Row #	Code	
			Hex	binary
a	6	1	61H	0110 0001B
A	4	1	41H	0100 0001B
β	E	1	E1H	1110 0001B
%	2	5	25H	0010 0101B

Table 2.5: Using the ASCII table

2.7 Some Basic Assembly Instructions

In this experiment and the next one, a number of basic instructions will be introduced. The aim of which is to provide the student with a set of basic tools that allows him write basic programs.

2.7.1 ADD & SUB instructions

In this experiment, only ADD and SUB instructions are introduced. In the following table (Table 2.5) basic arithmetic instructions are summarized. The effect of each instruction on the flags is also shown together with a brief example. The "*" symbol indicates that the corresponding flag may change as a result of executing the instruction. The "-"indicates that the corresponding flag is not affected by the instruction, whereas the "?"means that the flag is undefined after the instruction is executed.

Type	Inst.	Example	Meaning	Flags Affected					
				OF	SF	ZF	AF	PF	CF
Addition	ADD	ADD AX, 7BH	AX ← AX + 7Bh	*	*	*	*	*	*
	ADC	ADC AX, 7BH	AX ← AX + 7Bh + CF	*	*	*	*	*	*
	INC	INC BX	BX ← BX + 1	*	*	*	*	*	-
	DAA	DAA	Decimal Adjust after ADD	?	*	*	*	*	*
Subtraction	SUB	SUB CL,AH	CL ← CL – AH	*	*	*	*	*	*
	SBB	SBB CL,AH	CL ← CL – AH – CF	*	*	*	*	*	*
	DEC	DEC DAT	[DAT] ← [DAT] – 1	*	*	*	*	*	-
	DAS	DAS	Decimal Adjust after SUB	?	*	*	*	*	*
	NEG	NEG CX	CX ← 0 – CX	*	*	*	*	*	*

Table 2. 6: Summary of basic arithemtic instructions

2.8 The Binary Coded Decimal system

The Binary Coded Decimal system or BCD is used to represent the binary equivalent of the decimal system. In BCD, the binary patterns 1010 through 1111 do not represent valid BCD numbers, and cannot be used.

Conversion from Decimal to BCD

Thousands	Hundreds	Tens	Units
5	3	1	9
00000101	00000011	00000001	00001001

Table 2. 7: Unpacked BCD

Since computer storage requires minimum of 1 byte, the upper nibble of each BCD number is wasted. But BCD is a weighted position number system so you may perform mathematics, but we must use special techniques in order to obtain a correct answer.

2.8.1 Packed BCD

To eliminate wasted storage BCD numbers are represented in packed format. In a packed BCD number, each nibble has a weighted position starting from the decimal point. Therefore, instead of requiring 4 bytes to store the BCD number 5319, we would require 2 bytes. The upper nibble of the upper byte of our number would store the thousands value while the lower nibble of the upper byte would store the hundreds value. Likewise, the lower byte would store the tens value in the upper nibble and the units digit in the lower nibble. Therefore, our previous example would be:

Thousands-Hundreds	**Tens-Units**	
53	**1**	**9**
00000101 00000011	**00000001 00001001**	

Table 2. 8: Packed BCD

2.8.2 Decimal After Add and Subtract

The DAA (Decimal Adjust after Addition) instruction, used immediately after normal addition instruction, allows addition of numbers represented in 8-bit packed BCD code. The sum in AL is adjusted to packed BCD format. The AL register is the source and the destination and hence no operand is required. The effect of DAS (Decimal Adjust after Subtraction) instruction is similar to that of DAA, except that it is used after a subtraction operation. For more details on BCD, refer to your class notes or the COE 200 course. You can also refer to the Appendix to this experiment.

2.9 Lab Work

Part 1:

1- Study the attached programs and review the material related to data representation arithmetic instructions, and BCD code.
2- Write both programs and see how program 2.1 manipulates the variables in internal registers, and how program 2.2 uses memory for the same purpose.
3- Modify program 2.1 so that it adds two numbers of two digits each. Use only registers, and make sure to take care of the carry when adding the two most significant digits. Call this program 2.3.
Note: In this case try to understand how the program reads the numbers and how it manipulates them. This will help you in writing your program. As a **hint**, one should know that numbers are given in decimal to the program.

Program 1:

1. Write the following program and use the CV to answer the following questions?
2. What is the starting address of the memory where the code of this program is stored?
3. What is the starting address of the data segment?
4. What is the equivalent binary code for the instruction#3? What is its size?
5. How much memory is required to store the program?

```
TITLE "programB2"
.MODEL SMALL
.STACK 100
.DATA

   NUM1 DB 9
   NUM2 DB 8
   X DB 'A'

.CODE

   MOV AX,@DATA    ; 1
   MOV DS,AX       ; 2
   MOV AL,NUM1     ; 3
   ADD NUM2,AL     ; 4
   MOV BL,X        ; 5
   MOV AX,4C00H    ; 6
   INT 21H         ; 7

END
```

6. By looking at the binary code, what type of instruction do you think the opcode "B8" refers to?
7. What is the address of the location storing the variables NUM1 & NUM2?
8. What is the value stored in the memory location of NUM2 before and after executing instruction#4?

Before:

After:

9. Write the status of the flags and their meanings after executing instruction #4?

Overflow	**Direction**	**Interrupt**	**Sign**
Zero	**Auxiliary**	**Parity**	**Carry**

10. Run the program step-by-step and write the values of the *source* and *destination* before and after each instruction.

	Source		**Destination**	
Instruction	**Before**	**After**	**Before**	**After**
MOV AX, @DATA				
MOV DS, AX				
MOV AL, NUM1				
ADD NUM2 ,AL				
MOV BL ,C				
MOV AX, 4C00H				
INT 21H				

Program 2

```
.Model Small
.STACK  200
.DATA
        NUM1        DB     235
        NUM2        DB     0AFH
        RES         DB       ?
.CODE
.STARTUP
            MOV NUM1, AL  ; save num1
            MOV NUM2, AL  ; save num2
            ADD AL, NUM1  ; perform addition
            MOV RES, AL   ; save result in RES
            MOV DL, RES   ; retrieve RES from memory
.EXIT
END
```

Appendix

Packed BCD Arithmetic

If the addition of any two digits results in a binary number between 1010 and 1111, which are not valid BCD digits, or there is a carry into the next digit, then 6 (0110) is to be added to the current digit.

- Case 1

Carry from previous digit ➔	0		0	
	7		0111	
	+ 6		+ 0110	
	13		1101	
			110 ←	Add 6 because 1101 is not a valid
Carry to next digit ➔		1	**0011**	BCD digit

- Case 2

Carry from previous digit ➔	**1**		**1**	
	9		1001	
	+ 9		+ 1001	
	19	1	0011	
			110 ←	Add 6 because of carry to next digit
Carry to next digit ➔		1	**1001**	

Essentially, the rule is needed to "skip over" the six bit combinations that are unused by the BCD format whenever such a skip is warranted.