Lab Session 01

Introduction to Assembly Language Programming

ASSEMBLY LANGUAGE SYNTAX

name operation operand (s) comment

Assembly language statement is classified in two types

Instruction

Assembler translates into machine code.

Example:

```
        START:      MOV   CX, 5   ; initialize counter
```

Comparing with the syntax of the Assembly statement, name field consists of the label START:. The operation is MOV, operands are CX and 5 and the comment is ;initialize counter.

Assembler Directive

Instructs the assembler to perform some specific task, and are not converted into machine code.

Example:

```
        MAIN        PROC
```

MAIN is the name, and operation field contains PROC. This particular directive creates a procedure called MAIN.

Name field

Assembler translate name into memory addresses. It can be 31 characters long.

Operation field

It contains symbolic operation code (opcode). The assembler translates symbolic opcode into machine language opcode. In assembler directive, the operation field contains a pseudo-operation code (pseudo-op). Pseudo-op are not translated into machine code, rather they simply tell the assembler to do something.

Operand field

It specifies the data that are to be acted on by the operation. An instruction may have a zero, one or two operands.

Comment field

A semicolon marks the beginning of a comment. Good programming practice dictates comment on every line

Examples: MOVCX, 0 ;move 0 to CX
Do not say something obvious; so:
MOV CX, 0 ;CX counts terms, initially 0

Put instruction in context of program
; initialize registers

DATA REPRESENTATION

Numbers

11011	decimal
11011B	binary
64223	decimal
-21843D	decimal
1,234	illegal, contains a non-digit character
1B4DH	hexadecimal number
1B4D	illegal hex number, does not end with
FFFFH	illegal hex number, does not begin with digit
OFFFFH	hexadecimal number

Signed numbers represented using 2's complement.

Characters

- Must be enclosed in single or double quotes, e.g. “Hello”, ‘Hello’, “A”, ‘B’
- encoded by ASCII code
  - 'A' has ASCII code 41H
  - 'a' has ASCII code 61H
  - '0' has ASCII code 30H
  - Line feed has ASCII code OAH
  - Carriage Return has ASCII code
  - Back Space has ASCII code 08H
  - Horizontal tab has ASCII code 09H

VARIABLE DECLARATION

Each variable has a type and assigned a memory address.
Data-defining pseudo-ops

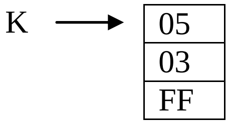
DB	define byte
DW	define word
DD	define double word (two consecutive words)
DQ	define quad word (four consecutive words)
DT	define ten bytes (five consecutive words)

Each pseudo-op can be used to define one or more data items of given type.

Byte Variables

Assembler directive format assigning a byte variable
Name DB initial value
A question mark ("?") place in initial value leaves variable uninitialized

```
I       DB   4          ;define variable I with initial value 4
J       DB   ?          ;Define variable J with uninitialized value
Name    DB   "Course"   ;allocate 6 bytes for name
K       DB   5, 3,-1    ;allocate 3 bytes
```



Other data type variables have the same format for defining the variables.
Like:
Name DW initial value

NAMED CONSTANTS

- EQU pseudo-op used to assign a name to constant.
- Makes assembly language easier to understand.
- No memory allocated for EQU names.

LF EQU 0AH

- MOV DL, 0AH
- MOV DL, LF

PROMPT EQU "Type your name"

- MSG DB "Type your name"
- MDC DB PROMPT

INPUT AND OUTPUT USING DOS ROUTINES

CPU communicates with peripherals through I/O registers called I/O ports. Two instructions access I/O ports directly: IN and OUT. These are used when fast I/O is essential, e.g. games.

Most programs do not use IN/OUT instructions. Since port addresses vary among computer models and it is much easier to program I/O with service routines provided by manufacturer.

Two categories of I/O service routines are Basic input & output system (BIOS) routines and Disk operating system (DOS) routines. Both DOS and BIOS routines are invoked by INT (interrupt) instruction.

Disk operating system (DOS) routines

INT 21 H is used to invoke a large number of DOS function. The type of called function is specified by pulling a number in AH register.

For example

AH=1	input with echo
AH=2	single-character output
AH=9	character string output
AH=8	single-key input without echo
AH=0Ah	character string input

Single-Key Input

Input: AH=1
Output: AL= ASCII code if character key is pressed, otherwise 0.

To input character with echo:

```
MOV     AH, 1
INT     21H           read character will be in AL register
```

To input a character without echo:

```
MOV     AH, 8
INT     21H           read character will be in AL register
```

Single-Character Output

Input: AH=2,
DL= ASCII code of character to be output
Output: AL=ASCII code of character

To display a character

```
MOV     AH, 2
MOV     DL, ‘?’
INT     21H            displaying character'?'
```

Combining it together:

```
MOV     AH, 1
INT     21H
MOV     AH, 2
MOV     DL, AL
INT     21H             read a character and display it
```

To Display a String

Input: AH=9,
DX= offset address of a string.
String must end with a '$' character.

To display the message Hello!

```
MSG      DB      "Hello!"
MOV      AH, 9
MOV      DX, offset MSG
INT      2IH
```

OFFSET operator returns the address of a variable The instruction LEA (load effective address) loads destination with address of source
LEA DX, MSG

PROGRAM STRUCTURE

Machine language programs consist of code, data and stack. Each part occupies a memory segment. Each program segment is translated into a memory segment by the assembler.

Memory models

The size of code and data a program can have is determined by specifying a memory model using the .MODEL directive. The format is:

```
.MODEL   memory-model
```

Unless there is lot of code or data, the appropriate model is SMALL

memory-model	description
SMALL	One code-segment. One data-segment.
MEDIUM	More than one code-segment. One data-segment. Thus code may be greater than 64K
COMPACT	One code-segment. More than one data-segment.
LARGE	More than one code-segment. More than one data-segment. No array larger than 64K.
HUGE	More than one code-segment. More than one data-segment. Arrays may be larger than 64K.

Data segment

A program's DATA SEGMENT contains all the variable definitions.
To declare a data segment, we use the directive .DATA, followed by variable and constants declarations.

```
.DATA
WORD1           DW          2
MASK            EQU         10010010B
```

Stack segment

It sets aside a block of memory for storing the stack contents.

```
.STACK          100H        ;this reserves 256 bytes for the stack
```

If size is omitted then by-default size is 1KB.

Code segment

Contain program's instructions.

```
.CODE           name
```

Where name is the optional name of the segment

There is no need for a name in a SMALL program, because the assembler will generate an error). Inside a code segment, instructions are organised as procedures. The simplest procedure definition is

```
name      PROC
;body of message
name      ENDP
```

An example

```
MAIN PROC
;main procedure instructions
MAIN ENDP
;other procedures go here
```

Putting it together

```
.MODEL          SMALL
.STACK          100H
.DATA
;data definition go here
.CODE
MAIN     PROC
;instructions go here
MAIN     ENDP
;other procedures go here
END             MAIN
```

The last line in the program should be the END directive followed by name of the main procedure.

A Case Conversion Program

Prompt the user to enter a lowercase letter, and on next line displays another message with letter in uppercase, as:
Enter a lowercase letter: a
In upper case it is: A

```
TITLE PGM4_1:  CASE CONVERSION PROGRAM
.MODEL  SMALL
.STACK   100H
.DATA
        CR        EQU           0DH
        LF        EQU           0AH
        MSG1      DB            'ENTER A LOWER CASE LETTER:  $'
        MSG2      DB            CR, LF, 'IN UPPER CASE IT IS:  '
        CHAR      DB            ?,'$'

.CODE
MAIN PROC
;initialize DS
        MOV       AX,@DATA  ; get data segment
        MOV       DS,AX         ; initialize DS
;print user prompt
        LEA       DX,MSG1       ; get first message
        MOV       AH,9          ; display string function
        INT       21H           ; display first message
;input a character and convert to upper case
        MOV       AH,1          ; read character function
        INT       21H           ; read a small letter into AL
        SUB       AL,20H        ; convert it to upper case
        MOV     CHAR,AL     ; and store it
;display on the next line
        LEA     DX,MSG2     ; get second message
        MOV     AH,9        ; display string function
        INT     21H         ; display message and upper case letter in front
;DOS exit
        MOV     AH,4CH      ; DOS exit
        INT     21H
MAIN    ENDP
        END     MAIN
```

Save your program with (.asm) extension.
If “**first**” is the name of program then save it as “**first.asm**”