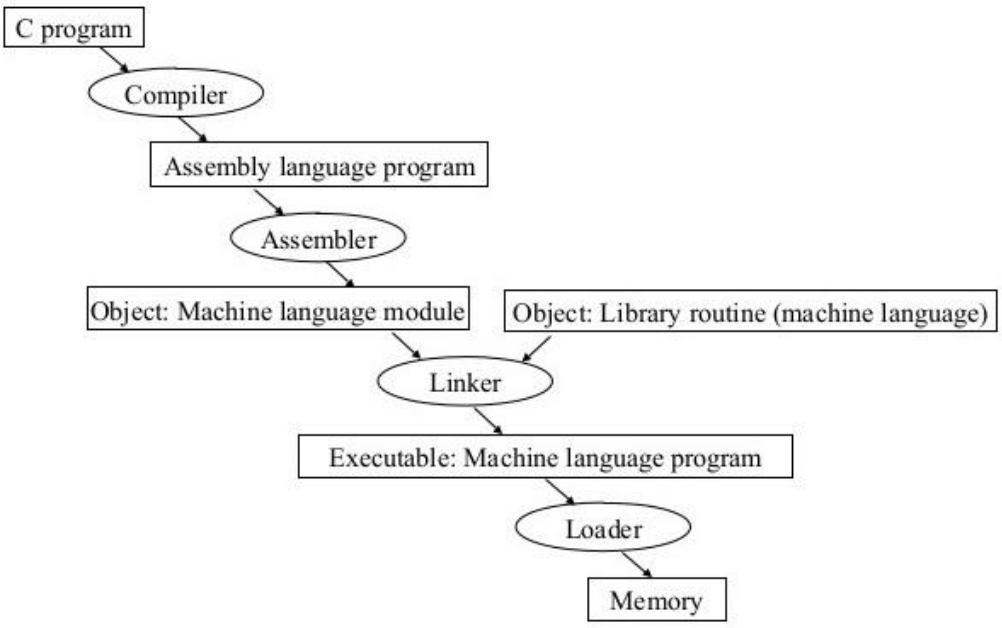
Lab 2: Assembly Language Fundamentals	**Session:** Fall 2019
Instructor(s): Faheem Ahmad, Miss Sumaiyah	

Steps in creating & running code



Program Template

```
TITLE Program Template (Template.asm)
; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date:
INCLUDE Irvine32.inc
.data
        ; (insert variables here)
.code
        main PROC
                ; (insert executable instructions here)
        exit
        main ENDP
        ; (insert additional procedures here)
END main
```

Integer Constant

Integer constants are made up of an optional leading sign, one or more digits and an optional suffix character.

Format:
[{+ | -}] digits radix

Examples:

26	for decimal
26d	for decimal
10111110b	for binary
42o	for octal
1Ah	for Hexadecimal
0A3h	for Hexadecimal

Character Constant

Character constants are made up of a single character enclosed in either single or double quotes.

Example:
'A'
"d"

String Constant

A string of characters enclosed in either single or double quotes.

Example:
"Hello World"

Identifiers

An identifier is a programmer-defined name of a variable, procedure or code label.

Format:
They may contain between 1 and 247 characters.
They are not case sensitive.
The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $. Subsequent characters may also be digits.
An identifier cannot be the same as an assembler reserved word.

Examples:
myVar
_abc
hello2

Data Types

MASM defines intrinsic data types, each of which describes a set of values that can be assigned to variables and expressions of the given type.

BYTE	8-bit unsigned integer
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned. D stands for double
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit integer. T stands for ten

Examples:

```
value1 BYTE 'A'                    ; character constant
value2 BYTE 0                      ; smallest unsigned byte
value3 BYTE 255                    ; largest unsigned byte
value4 SBYTE −128                  ; smallest signed byte
value5 SBYTE +127                  ; largest signed byte
greeting1 BYTE "Good afternoon”, 0 ; String constant
greeting2 BYTE 'Good night'        ; String constant
list BYTE 10,20,30,40              ; Multiple initializers
```

Note: A question mark (?) initializer leaves the variable uninitialized, implying it will be assigned a value at runtime:

```
value6 BYTE ?
```

Activity:
Write a data declaration for an 8-bit unsigned integer and store 10 in it. Move this value to AL and add 40 to it.

Memory Segments

A segmented memory model divides the system memory into groups of independent segments referenced by pointers located in the segment registers. Each segment defines the area of our program that contains data variables, code and stack, respectively.

Data segment: It is the memory region, where data elements are stored for the program. This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

Code segment: This section defines an area in memory that stores the instruction codes. This is also a fixed area.

Stack segment: This segment contains data values passed to procedures within the program.

Directives

A *directive* is a command embedded in the source code that is recognized and acted upon by the assembler. Directives do not execute at runtime. They can assign names to memory segments. In MASM, directives are case insensitive. For example, it recognizes .data, .DATA and .Data as equivalent.

Let us see what different directives we can use to define segments of our program:

The **.DATA** directive identifies the area of a program containing variables:
Syntax: .data

The **.CODE** directive identifies the area of a program containing executable instructions:
Syntax: .code

The **.STACK** directive identifies the area of a program holding the runtime stack, setting its size:
Syntax: .stack 100h

Instructions

An *instruction* is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. An instruction contains four basic parts:

1. Label (optional)
2. Instruction mnemonic (required)
3. Operand(s) (usually required)
4. Comment (optional)

The basic syntax of an Assembly Language instruction is as:

[*label*:] *mnemonic* [*operands*] [;*comment*]

where elements in square brackets are optional.

We will now see what each of these elements.

Label: A *label* is an identifier that acts as a place marker for instructions and data.

Operands: Assembly language instructions can have between zero and three operands, each of which can be a register, memory operand, constant expression, or input-output port.

Mnemonics: An instruction mnemonic is a short word that identifies an instruction to perform an operation. Following are examples of instruction mnemonics:

- **mov**: Moves (assigns) one value to another.
- **add**: Adds two values
- **sub**: Subtracts one value from another
- **mul**: Multiplies two values
- **jmp**: Jumps to a new location
- **call**: Calls a procedure

Activity:
Create an uninitialized data declaration for 64-bit integer

Legacy Data Directives

Following are some examples of using define directives:

```
choice      DB   'Y'
number      DW   12345
neg_number  DW   -12345
big_number  DQ 123456789
```

Exercises:

1. Write an uninitialized data declaration for an 8-bit signed integer *val1* and also initialize another 8-bit signed integer *val2* with -10. Now use the value of *val2* to initialize *val1*.
2. Create an uninitialized data declaration for a 16-bit unsigned integer. Copy whatever is in the BX to this integer.
3. Declare a 32-bit signed integer *val3* and initialize it with the smallest possible negative decimal value.
4. Declare an unsigned 16-bit integer variable named **wArray** that uses three initializers
5. Declare a string variable containing the name of your favorite color. Initialize it as a null terminated string.
6. Initialize five 16-bit unsigned integers A, B, C, D & E with the following values: *12, 2, 13, 8, 14*. Create another uninitialized unsigned integer called *value*. Now write a program to evaluate the expression A * B + C * D – E and store the result in *value*.
(Note: For this example, expression should be resolved from left to right)