**Namal University**

**Department of Computer Science**

|  |  |
| --- | --- |
| **Faculty Member: Dr. Muhammad Sadiq Amin** | **Dated: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |
|  |  |
| **Course/Section:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Semester: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |
|  |  |

**Computer Organization and**

**Assembly Language (CS233)**

**Lab #2 Data Types in Assembly Language**

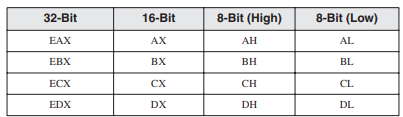
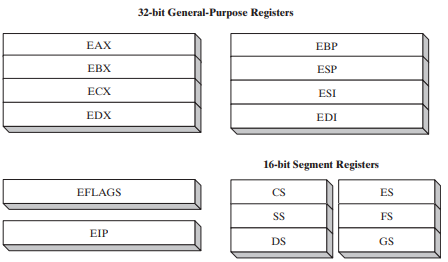
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Grading** | | | | |
| **Name** | **Registration No.** | **Report Marks**  **(Max. 8)** | **Viva Marks (Max. 7)** | **Total**  **(Max. 15)** |
| Muhammad Waqas | NUM-BSCS-2022-23 |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

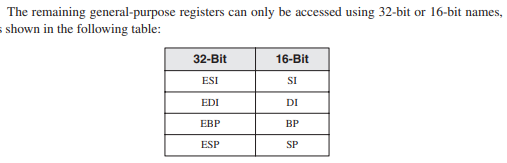
**Data Types in Assembly Language**

**Objective:** The aim of this lab is to understand specialized purpose of general purpose registers and to practice declaring and manipulating variables in assembly language programs and verifying the outputs.









**Assignment 1 of coal Lab:**

Write small detail of each of CPU registers. Write in detail the purpose of general purpose registers and segment registers. Assignment submission is in group.

**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:

• Label (optional)

• Instruction mnemonic (required)

• Operand(s) (usually required)

• Comment (optional)

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

**Intruction Mnemonic:**

An instruction mnemonic is a short word that identifies an instruction. In English, a mnemonic is a device that assists memory. Similarly, assembly language instruction mnemonics such as mov, add, and sub provide hints about the type of operation they perform.

* + mov Move (assign) one value to another
  + add Add two values.
  + sub Subtract one value from another
  + mul Multiply two values
  + jmp Jump to a new location
  + call Call a procedure

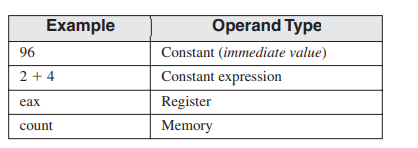
**Operands:**

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.

**Examples:**

* stc, inc eax, mov count,ebx, imul eax,ebx,5

A memory operand is specified by the name of a variable or by one or more registers containing the address of a variable.



**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.

Directives can define

* + variables and procedures
  + program sections, or segments.

In MASM, directives are case insensitive. .data, .DATA, and .Data as equivalent.

**Identifier:**

An identifier is a programmer-chosen name. It might identify a

* + variable
  + constant
  + procedure
  + code label.

**Label:**

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

1. **Data Label**

A data label identifies the location of a variable, providing a convenient way to reference the variable in code.

* + count DWORD 100
  + array DWORD 1024, 2048

1. **Code Label:**

A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions.

target:

mov ax,bx

...

jmp target

A code label can share the same line with an instruction, or it can be on a line by itself:

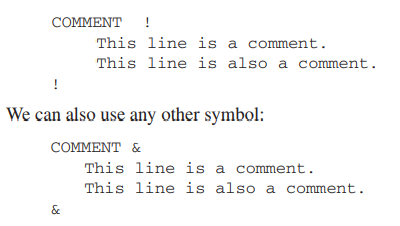
L1: mov ax,bx

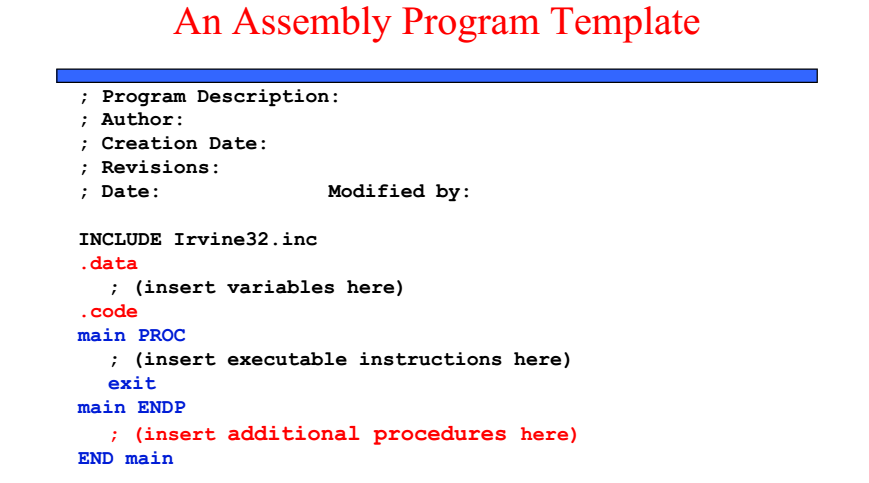
L2:

**Comments:**

Single-line comments, beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.

Block comments, beginning with the COMMENT directive and a user-specified symbol. All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears.



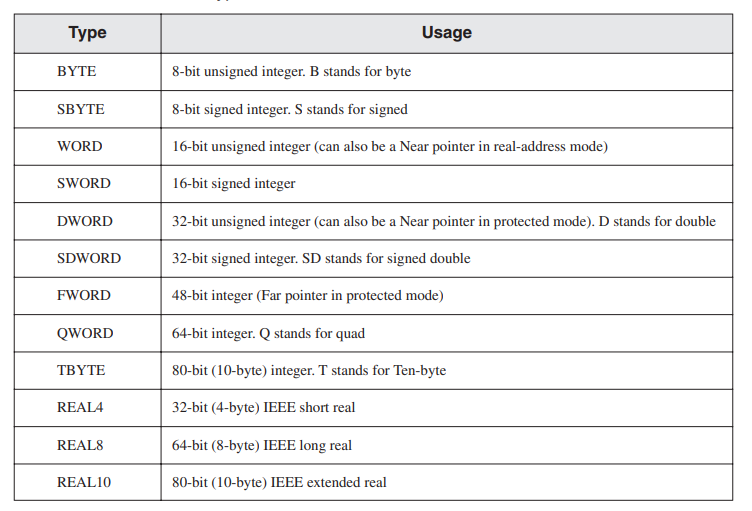


**Data Types:**

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. The essential characteristic of each type is its size in bits: 8, 16, 32, 48, 64, and 80.

A variable declared as DWORD, for example, logically holds an unsigned 32-bit integer.

The assembler is not case sensitive, so a directive such as DWORD can be written as dword, Dword, dWord, and so on.



**Defining data:**

A data definition statement sets aside storage in memory for a variable, with an optional name. Data definition statements create variables based on intrinsic data types.

A data definition has the following syntax:

[name] directive initializer [,initializer]

**Initializers:**

All the data types you have seen are integer and real. For integer data types, initializer is an integer constant or expression matching the size of the variable’s type, such as BYTE or WORD.

If you prefer to leave the variable uninitialized (assigned a random value), the ? symbol can be used as the initializer.

All initializers, regardless of their format, are converted to binary data by the assembler.

Initializers such as 00110010b, 32h, and 50d all end up being having the same binary value.

1. **Integer Constant:**

An integer constant (or integer literal) is made up of an optional leading sign, one or more digits, and an optional suffix character (called a radix) indicating the number’s base:

[{+ | −}] digits [radix]

|  |  |
| --- | --- |
| Type | Representation |
| Decimal | d, t |
| Binary | b, y |
| Octal | q, o |
| Hexadecimal | h |

1. **Real Constant:**

Real number constants are represented as decimal reals or encoded (hexadecimal) reals. A decimal real contains an optional sign followed by an integer, a decimal point, an optional integer that expresses a fraction, and an optional exponent:

[sign]integer.[integer][exponent]

Following are the syntax for the sign and exponent:

sign {+,-} exponent E[{+,-}]

**Examples:**

2. , +3.0, -44.2E+05, 26.E5

1. **Character Constant:**

A character constant is a single character enclosed in single or double quotes. MASM stores the value in memory as the character’s binary ASCII code.

**Examples:**

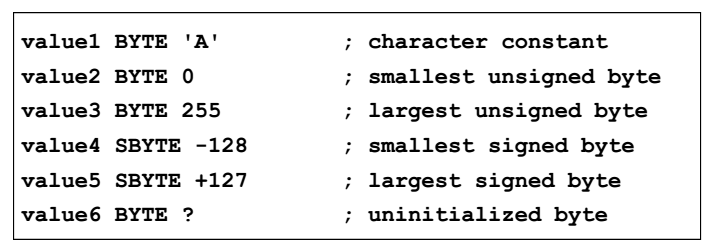
‘A’ , "d"

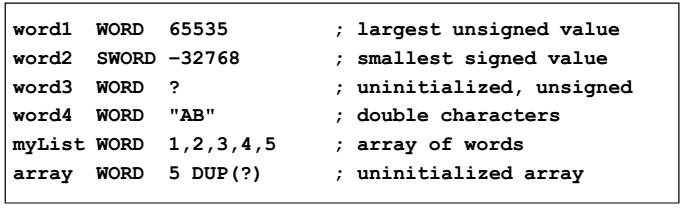
1. **String Constant:**

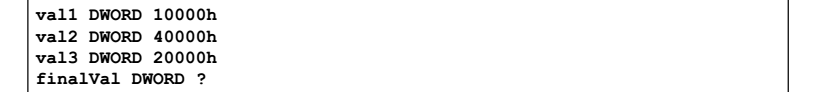
A string constant is a sequence of characters (including spaces) enclosed in single or double quotes:

**Examples:**

'ABC’, 'X’, "Good night, Gracie", '4096'







**Exercise 1:** Assemble and run the following program.

**Program**

TITLE Add and Subtract, (AddSub2.asm)

; This program adds and subtracts 32-bit unsigned

; integers and stores the sum in a variable.

INCLUDE Irvine32.inc

.data

val1 DWORD 10000h ;val1 declared as a variable of type DWORD and initialized

val2 DWORD 40000h

val3 DWORD 20000h

finalVal DWORD ?

.code

main PROC

mov eax,val1 ; start with 10000h

add eax,val2 ; add 40000h

sub eax,val3 ; subtract 20000h

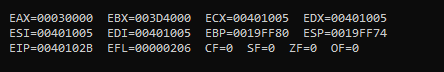
mov finalVal,eax ; store the result (30000h)

call DumpRegs ; display the registers

exit

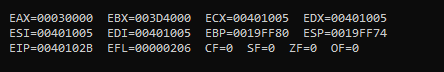
main ENDP

END main



**Exercise 2:** Note down the contents of registers EAX, EBX and ECX as displayed by the program. Do the registers contents match the expected results?

Yes it match the expected result



**Exercise 3:** Write code to achieve the following:

1. Define two 8 bit variables var1, and var2, and initialize these to 20, and 30.
2. Swap the contents of var1 and var2 variables using registers.
3. Display the contents of the registers. (Use “call dumpregs” instruction twice, First display variable before swapping, then display variable after swapping)

TITLE Add and Subtract, (AddSub2.asm)

; This program adds and subtracts 32-bit unsigned

; integers and stores the sum in a variable.

INCLUDE Irvine32.inc

.data

val1 BYTE 20h ;val1 declared as a variable of type DWORD and initialized

val2 BYTE 30h

.code

main PROC

mov al,val1

mov bl,val2

call DumpRegs

mov cl,al

mov al,bl

mov bl,cl

mov val1,al

mov val2,bl

call DumpRegs

;to cechk that varaible values are changed so we make val1 to another register and show its output output

mov dl,val1

call DumpRegs

exit

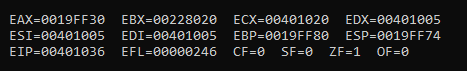
main ENDP

END main

**Before swapping**

1. 

**After swapping**



**Exercise 4:** Write codes to evaluate the arithmetic expression “5+(6-2)”, by:

1. Using one register only

TITLE Add and Subtract, (AddSub2.asm)

; This program adds and subtracts 32-bit unsigned

; integers and stores the sum in a variable.

INCLUDE Irvine32.inc

.data

.code

main PROC

mov al,0h

mov al,6h

sub al,2h

add al,5h

call DumpRegs

exit

main ENDP

END main

1. 
2. Using two registers only

TITLE Add and Subtract, (AddSub2.asm)

; This program adds and subtracts 32-bit unsigned

; integers and stores the sum in a variable.

INCLUDE Irvine32.inc

.data

.code

main PROC

mov al,6h

mov bl,5h

sub al,2h

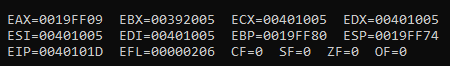
add al,bl

call DumpRegs

exit

main ENDP

END main



Write down the source codes below.

**Exercise 5:** Write a constant expression that divides -10 by 3

1. display the integer remainder and quotient.
2. display the remainder and quotient in hexadecimal.

TITLE Displaying the values of quotient and Remainder (Divide.asm)

INCLUDE Irvine32.inc

.data

dividend DWORD -10

divisor DWORD 3

quotient DWORD ?

remainder DWORD ?

.code

main PROC

;Evaluate quotient and remainder

mov eax,0

mov edx,0

mov eax, dividend ; Moving dividend (10) into eax

div divisor ; Dividing edx:eax by divisor (3)

mov quotient, eax ; Moving quotient into the quotient variable

mov remainder, edx ; Moving remainder into the remainder variable

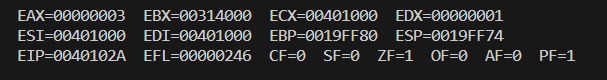
mov eax, quotient

Call dumpregs ;displays the result on the screen by displaying all register values

Exit

Main endp

End main



**Conclusion**

In this lab we learn about general purpose register of 32 bit,16 bit and 8 bit. We learn about the four parts of instruction or we can say that syntax of instruction .we learn about commenting the lines in our assembly programs.we learn about the data types.we also learn how to make assembly program file and run the code given in lab. In our exercises we add 2 registers,swap the values of variable and solve arithmetic operation