**Namal University**

**Department of Computer Science**

**Faculty Member: Dr. Muhammad Sadiq Amin Dated: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Course/Section:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Semester: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Computer Organization and Assembly Language (CS233)**

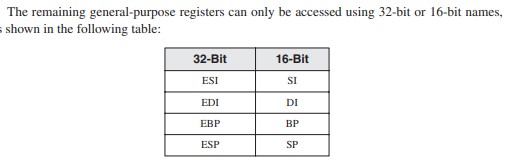
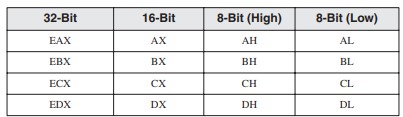
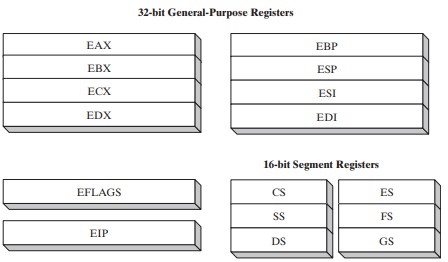
**Lab #2 Data Types in Assembly Language**

**Grading**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Registration No.** | **Report Marks (Max. 8)** | **Viva Marks (Max. 7)** | **Total (Max. 15)** |
| MUHAMMAD SHAHBAZ | BSCS-2022-30 |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 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

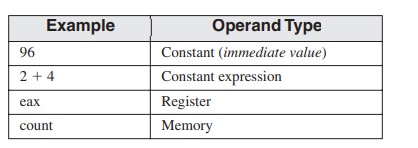
**Operands:**

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

**2. 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

...

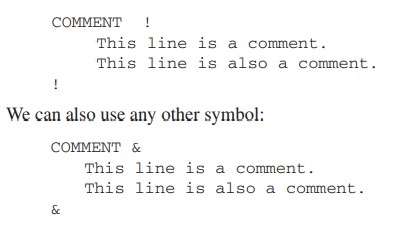
jmp target

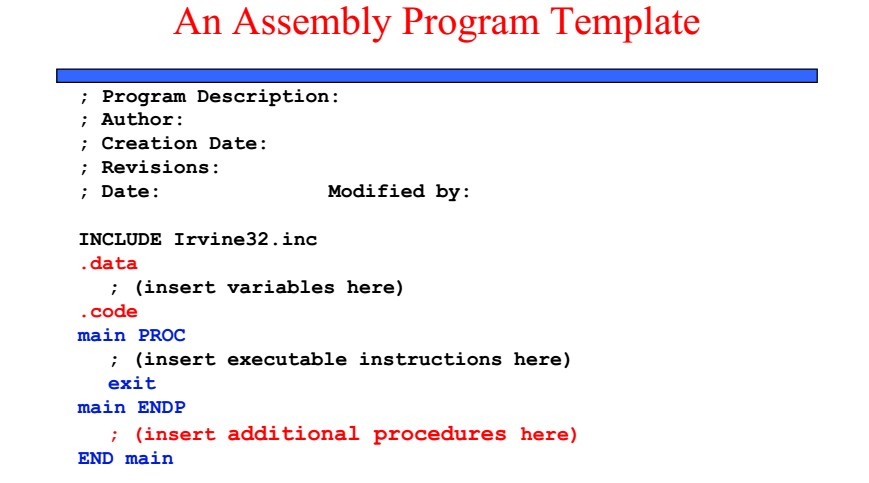
A code label can share the same line with an instruction, or it can be on a line by itself: L1: mov ax,bx L2:

**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.

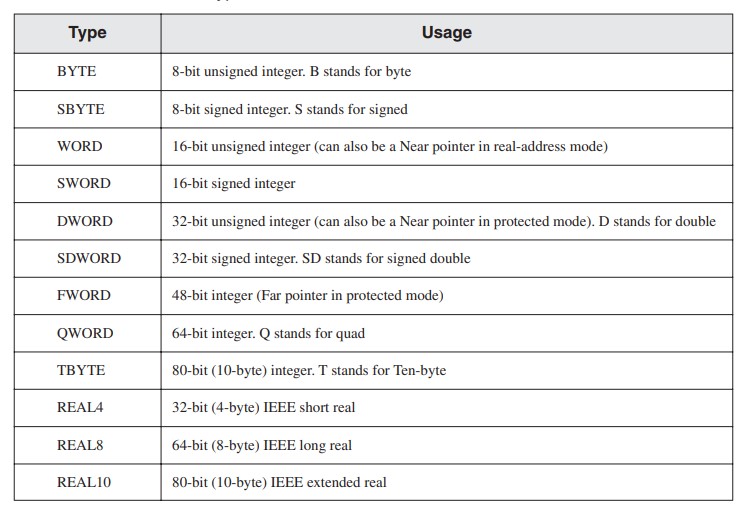




**Data Types:**

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:**

* 1. , +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:**

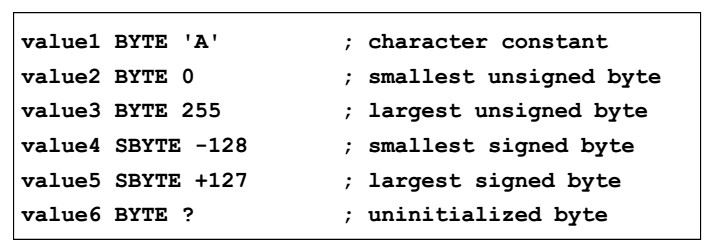
‘A’ , "d"

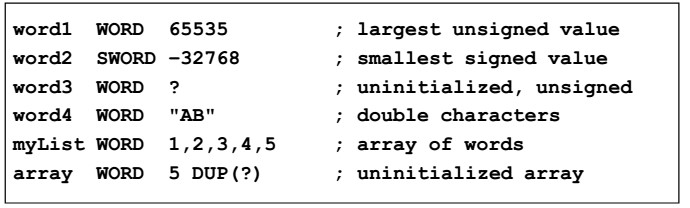
**4. String Constant:**

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

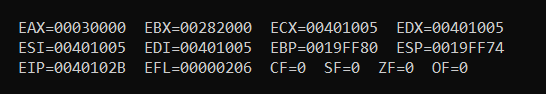
|  |  |
| --- | --- |
| mov eax,val1 | ; start with 10000h |
| add eax,val2 | ; add 40000h |
| sub eax,val3 | ; subtract 20000h |
| mov finalVal,eax | ; store the result (30000h) |
| call DumpRegs  exit | ; display the registers |

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?

ANS: Yes the content of the registers match the expected results.



**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 Variable Swap (VarSwap.asm)

; This program swaps the contents of two 8-bit variables

; and displays the contents of registers.

INCLUDE Irvine32.inc

.data

var1 BYTE 20h ; Define two 8-bit variables and initialize them

var2 BYTE 30h

.code

main PROC

; Variable swap operation

mov eax,0h

mov ebx,0h

mov al, var1 ; move contents of var1 into al register

mov bl, var2 ; move contents of var2 into bl register

; Display contents of registers before swapping

call DumpRegs

mov var1, bl ; move contents of bl (var2) into var1

mov var2, al ; move contents of al (var1) into var2

mov al, var1 ; move contents of var1 into al register

mov bl, var2 ; move contents of var2 into bl register

; Display contents of registers after swapping

call DumpRegs

exit ; exit the program

main ENDP

END main

A black background with white text

Description automatically generated

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

1. Using one register only
2. Using two registers only

Write down the source codes below.

TITLE Arithmetic Expression Evaluation (OneRegister.asm)

; This program evaluates the arithmetic expression "5+(6-2)" using one register only.

INCLUDE Irvine32.inc

.data

.code

main PROC

mov eax, 6h ; Move 5 into the eax register

sub eax, 2h ; Subtract 2 from eax (6 - 2)

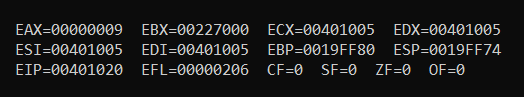
add eax, 5h ; Add 5 to eax (5 + (6 - 2))

call DumpRegs ; Display the contents of the registers

exit ; Exit the program

main ENDP

END main



TITLE Arithmetic Expression Evaluation (TwoRegisters.asm)

; This program evaluates the arithmetic expression "5+(6-2)" using two registers only.

INCLUDE Irvine32.inc

.data

.code

main PROC

mov eax, 2h ; Move 2 into the eax register

mov ebx, 6h ; Move 6 into the ebx register

sub ebx,eax ; Subtract 2 from eax (6 - 2), store the result in ebx

mov eax, 5 ; Move 5 into the ebx register

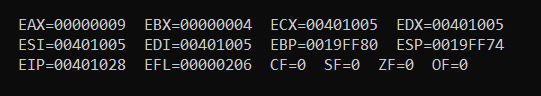
add eax, ebx ; Add the contents of eax (5) to ebx ((6 - 2) + 5)

call DumpRegs ; Display the contents of the registers

exit ; Exit the program

main ENDP

END main



**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

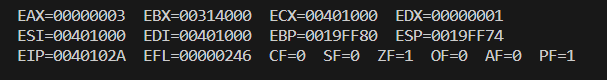
mov eax, quotient

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

Exit

Main endp

End main



**Conclusion:**

In this lab session focused on data types in assembly language, students delved into the intricate details of CPU registers, including their specialized purposes within the computer architecture. They learned about both general-purpose registers and segment registers, understanding how these components facilitate various operations and data management tasks within the CPU. With a comprehensive overview of instruction syntax, students grasped the fundamental structure of assembly language statements, including mnemonic codes, operands, labels, and comments. Through practical examples, they gained insights into data types and definitions in MASM, discovering how to reserve memory space for variables and initialize them with specific values. The lab also provided hands-on exercises in arithmetic expression evaluation, employing essential instructions such as mov, add, and sub to perform mathematical operations. Students honed their skills in variable manipulation, including swapping the contents of variables, and learned how to display register contents and arithmetic operation results in both decimal and hexadecimal formats. Overall, this lab served as a valuable opportunity for students to solidify their understanding of assembly language fundamentals, empowering them with practical experience in programming and verifying program outputs.