Namal University.

Department of Computer Science

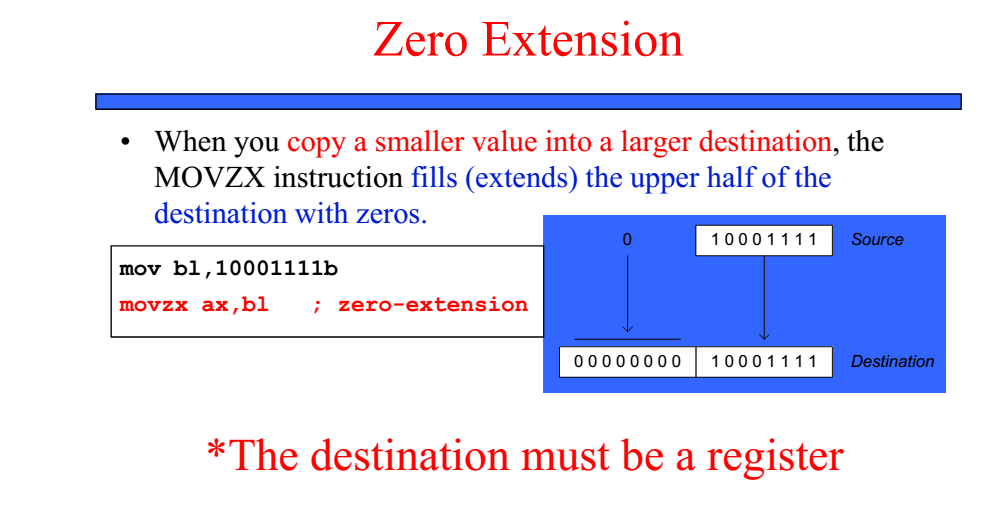
|  |  |
| --- | --- |
| **CS-235: Computer Organization and Assembly Language** | |
| **Faculty Member** | **Semester** |
| Dr Sadiq Amin | 4th Semester |
| **Class/Section** | **Date** |
| BSCS | 3/28/2024 |

**Lab 3: Memory Access in Assembly Language**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Grading** | | | | |
| **Name** | **Registration No.** | **Report Marks**  **(Max. 8)** | **Viva Marks (Max. 7)** | **Total**  **(Max. 15)** |
| Touseef Ahmed | NUM-BSCS-2022-18 |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

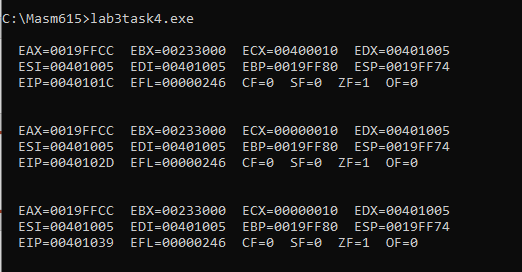
**Lab 3: Memory Access in Assembly Language**

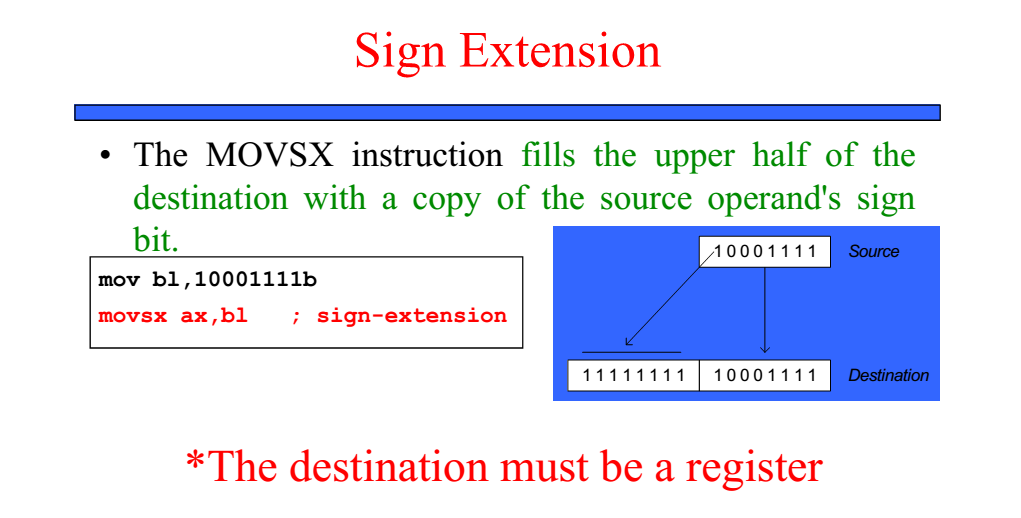
**Objective**

The aim of this lab is to use some of the data transfer and manipulation instructions, and to use some assembler operators.

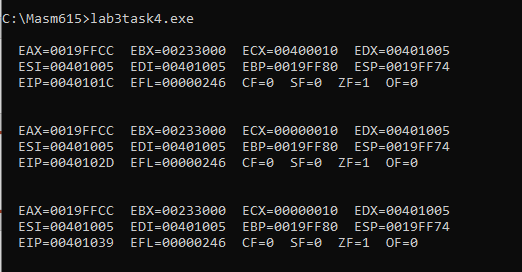
**Comparison of mov and movzx:**

Suppose count (unsigned, 16 bits) must be moved to ECX (32 bits).

* .data
* count WORD 16
* .code
* Main Proc
* **mov cx,count**
* Call dumpregs
* **mov ecx,0**
* **mov cx, count**
* Call dumpregs
* **movzx ecx,count**
* Call dumpregs

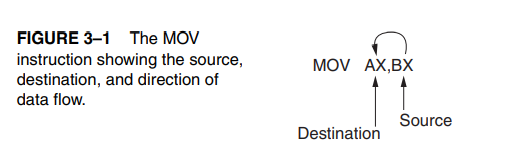


**Comparison of mov and movsx:**

Suppose count (unsigned, 16 bits) must be moved to ECX (32 bits).

* .data
* count SWORD 16
* .code
* Main Proc
* **mov cx,count**
* Call dumpregs
* **mov ecx,0**
* **mov cx, count**
* Call dumpregs
* **movsx ecx,count**
* Call dumpregs

**Data Addressing Modes:**

The MOV instruction is a very common and flexible instruction, it provides a basis for the explanation of the data-addressing modes. Figure 3–1 illustrates the MOV instruction and defines the direction of data flow. The source is to the right and the destination is to the left, next to the opcode MOV. (An opcode, or operation code, tells the microprocessor which operation to perform.)

**The data-addressing modes are as follows:**

1. **Register addressing:**

Register addressing transfers a copy of a byte or word from the source addressing register or contents of a memory location to the destination register or memory location.

**Example:** The MOV CX, DX instruction copies the word-sized contents of register DX into register CX.

MOV CS, DX (Not Allowed).

Note: Code segment (CS) register is not normally changed by a MOV instruction because the address of the next instruction is found by both IP/EIP and CS. If only CS were changed, the address of the next instruction would be unpredictable. Therefore, changing the CS register with a MOV instruction is not allowed.

1. **Immediate addressing:**

Immediate addressing transfers the source, an immediate byte, word, addressing doubleword, or quadword of data, into the destination register or memory location.

**Example:** The MOV AL, 22H instruction copies a byte-sized 22H into register AL.

1. **Direct addressing:**

Direct addressing moves a byte or word between a memory location addressing and a register. The instruction set does not support a memory-to memory transfer, except with the MOVS instruction.

**Example:** The MOV CX, LIST instruction copies the word-sized contents of memory location LIST into register CX.

MOV ESI, LIST instruction copies a 32-bit number, stored in four consecutive bytes of memory, from location LIST into register ESI.

1. **Register indirect addressing**:

Register indirect Register indirect addressing transfers a byte or word between a addressing register and a memory location addressed by an index or base register. The index and base registers are BP, BX, DI, and S1.

**Example:** The MOV AX, [BX] instruction copies the word-sized data from the data segment offset address indexed by BX into register AX.

1. **Base-plus-index addressing**:

Base-plus-index addressing transfers a byte or word between a addressing register and the memory location addressed by a base register (BP or BX) plus an index register (DI or SI).

**Example:** The MOV [BX + DI ] , CL instruction copies the byte-sized contents of register CL into the data segment memory location addressed by BX plus DI.

1. **Register relative addressing**:

Register relative addressing moves a byte or word between a register addressing and the memory location addressed by an index or base register plus a displacement.

**Example:** MOV AX,[ BX plus 4] or MOV AX,ARRAY[BX]. The first instruction loads AX from the data segment address formed by BX plus 4. The second instruction loads AX from the data segment memory location in ARRAY plus the contents of BX.)

1. **Base relative-plus-index addressing**:

Base relative-plus-index addressing transfers a byte or word between a index addressing register and the memory location addressed by a base and an index register plus a displacement.

**Example:** MOV AX, ARRAY[BX + DI ] or MOV AX, [BX+DI+4 ]. These instructions load AX from a data segment memory location. The first instruction uses an address formed by adding ARRAY, BX, and DI and the second by adding BX, DI, and 4.

1. **Scaled-index addressing**:

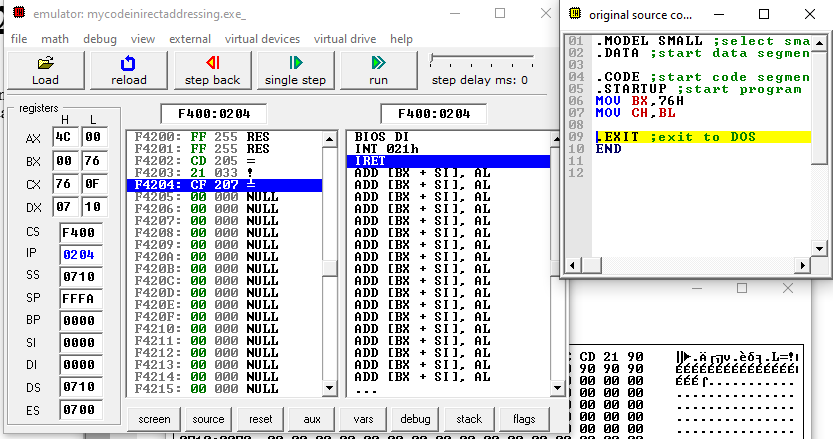
Scaled-index addressing is available only in the 80386 through the addressing Pentium 4 microprocessor. The second register of a pair of registers is modified by the scale factor of to generate the operand memory address.

**Example:** A MOV EDX, [EAX+ 4\*EBX ] instruction loads EDX from the data segment memory location addressed by EAX plus four times EBX.

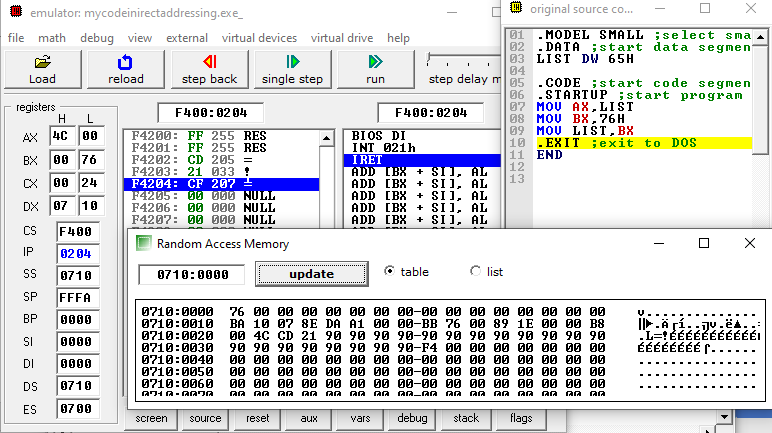
**Data-addressing modes Code Examples:**

Graphical user interface

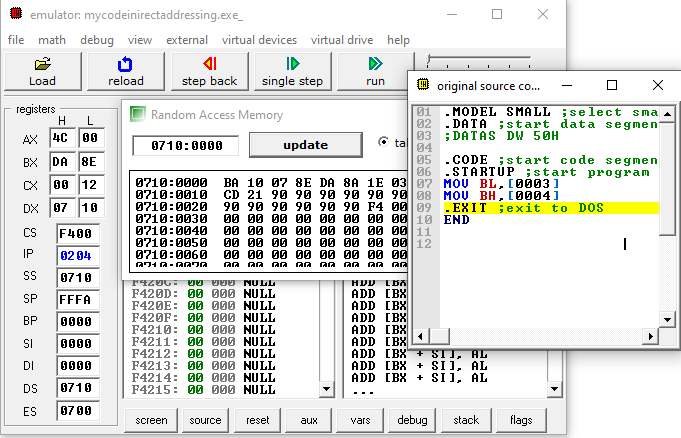
Description automatically generated



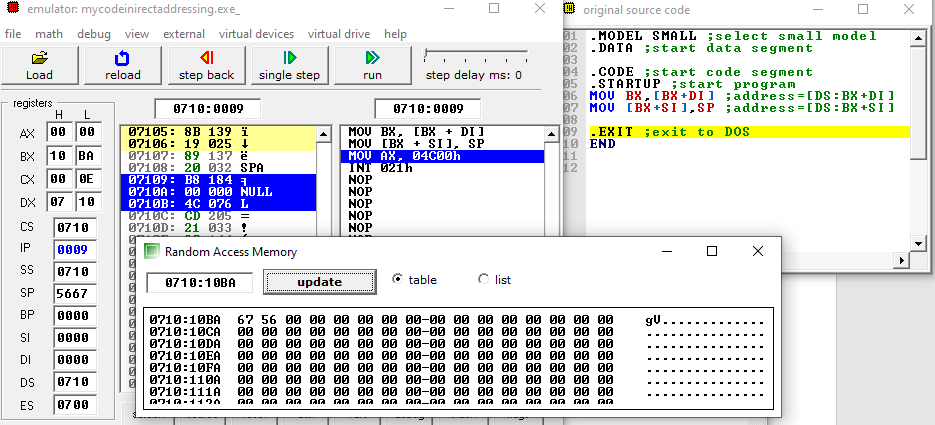
**Figure 1:Register and Immediate Addressing**



**Figure 2: Direct Addressing**



**Figure 3: Register Indirect Addressing**

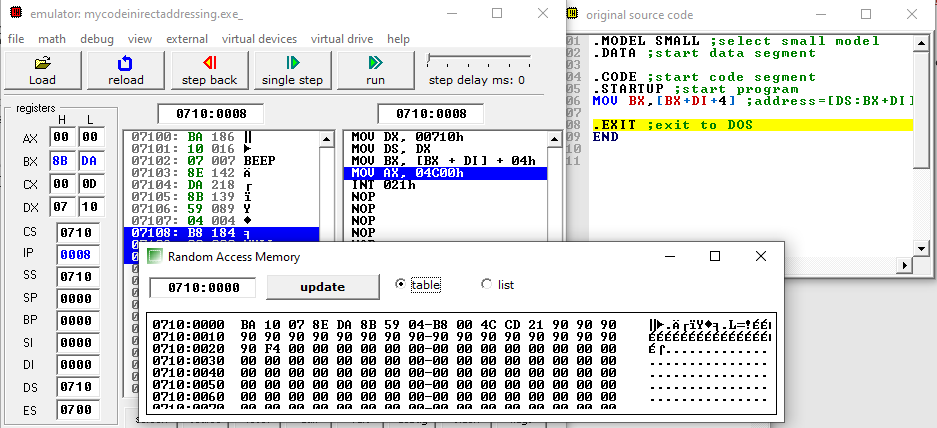


**Figure 4: Base Plus Index Addressing**

Graphical user interface, text, application

Description automatically generated

**Figure 5: Register Relative Addressing**



**Figure 6: Base Relative Plus Index Addressing**

**Exercise 1:** In the memory list shown, insert the values of the variables as declared below, in hexadecimal format:

|  |  |  |  |
| --- | --- | --- | --- |
| **offset** | **Content** | **Offset** | **Content** |
| 00 | 05 | 10 | 68 |
| 01 | 0C | 11 | 65 |
| 02 | 64 | 12 | 72 |
| 03 |  | 13 | 65 |
| 04 | 00 | 14 | 00 |
| 05 | 50 | 15 | 48 |
| 06 | 00 | 16 | 69 |
| 07 | 60 | 17 | 2C |
| 08 | A0 | 18 | 20 |
| 09 | 90 | 19 | 54 |
| 0A | 80 | 1A | 68 |
| 0B | 70 | 1B | 61 |
| 0C | 48 | 1C | 6E |
| 0D | 69 | 1D | 6B |
| 0E | 20 | 1E | 73 |
| 0F | 54 | 1F | 2E |
|  |  | 20 |  |

.data

mbyte BYTE 05,12,100

Align 2

mword WORD 50h, 60h

mdouble DWORD 0A0908070h

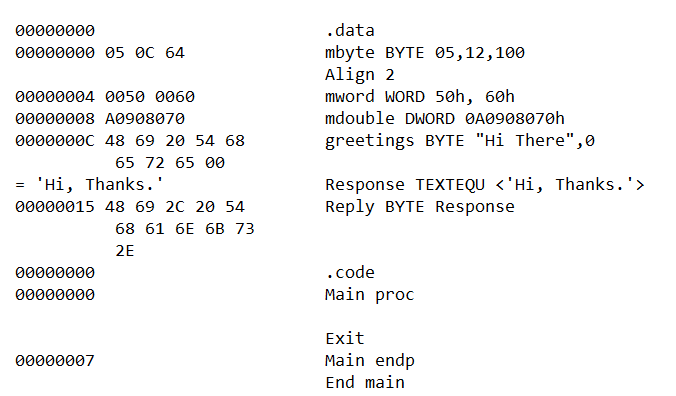
greetings BYTE "Hi There",0

Response TEXTEQU <'Hi, Thanks.'>

Reply BYTE Response

Note: No ASCII table is to be used, wait to fill the text character codes after the exercise where the textstrings have been used, and the .lst file can be used to see these codes). **DO NOT FORGET**

**Values from .lst file(OUTPUT)**



**Exercise 2:** Without writing any code, write down the expected contents of the register after the instruction is executed:

1. Mov al,mbyte ;AL=05
2. Movsx ax, mbyte+1 ;AX=12

**Exercise 3**: In this exercise we will learn about and use some new procedures that can be called to display register or memory contents.

**Writing strings to display:**

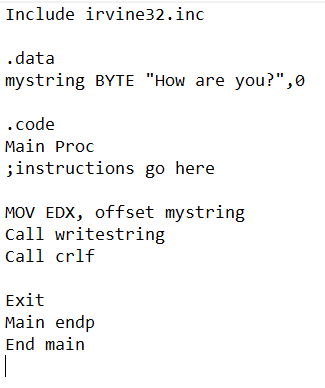
Declare/define the string: mystring BYTE “How are you?”,0

Get the offset of string into EDX: mov edx, offset mystring

Call the procedure to display: call **writestring**

(**Note** that writestring only works with EDX holding the offset)

**CODE**



**OUTPUT**



**Writing register constents to display:**

Get the contents to display in AL,AX, or EAX.

Ensure the bits not used are set/reset so as to improve readability.

Call the procedure to display: a. **writeint** to print in decimal format

b. **writehex** to print in hexadecimal format

c. **writebin** to print in binary format

d. **writechar** to print a character, the LSD of EAX

(**Note** that all these writexxx display EAX contents)

A call to **crlf** adds carraige return followed by a linefeed, eg., call crlf.

**Step1:** Write code to get Byte No 2 of mbyte into AL and byte No 1 of mbyte into AH. Ensure that the higher order bits of EAX are cleared. Display EAX to verify that the correct bytes are in the locations specified. Use all four write procedures to see the various output formats, with a call to crlf after each writexxx to make the output easy to read.

(**Note** Use the data given in Exercise No. 1)

Note down the outputs below:

**Part(a)**

|  |  |
| --- | --- |
| **For call writeint Procedure:** | **Code:** |
| **Value of EAX Register:**  EAX=0000050C | **Output:** |

**Part(b)**

|  |  |
| --- | --- |
| **For call writehex Procedure:** | **Code:** |
| **Value of EAX Register:**  EAX=0000050C | **Output:** |

**Part(c)**

|  |  |
| --- | --- |
| **For call writebin Procedure:** | **Code:** |
| **Value of EAX Register:**  EAX=0000050C | **Output:** |

**Part(d)**

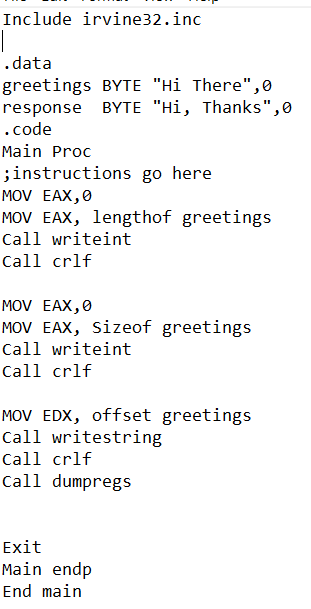
|  |  |
| --- | --- |
| **For call writechar Procedure:** | **Code:** |
| **Value of EAX Register:**  EAX=0000050C | **Output:** |

**Explanation:**

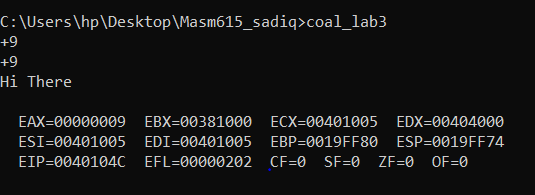
When we use the **call writechar** procedure , it reads the value provided and matches it with the ASCII table to find the corresponding character. For example, in ASCII, the hexadecimal value 12 represents the symbol ♀. When we **MOV AH, mbyte** to move the first byte of mbyte into the AH register, intending to display it using writechar. However, writechar interprets AH as a control code rather than a printable character.It displays the symbol ♀ as provided in ASCII table.

**Step2:** Extend program to display the length and size of the string variable “greetings” in decimal format, and then print the first string

**CODE**

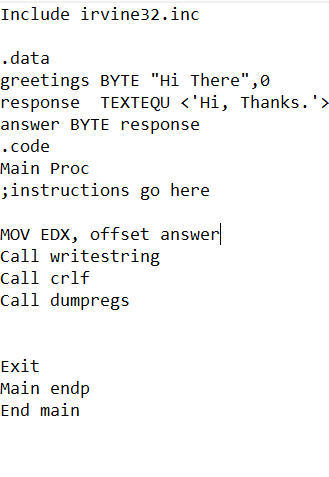


**OUTPUT**



**Step3:** Display the second string defined by the TEXTEQU operator.

**CODE**



**OUTPUT**

A computer screen with white text

Description automatically generated