

Project 6 - String Primitives and Macros

Due Aug 12 by 11:59pm **Points** 50 **Submitting** a file upload **File Types** asm
Available after Jul 31 at 12am

Introduction



This program, the portfolio project for the class, is the final step up in difficulty. Once again the Rubric (see below) now has a number of point deductions for not meeting requirements. It is not uncommon for a student to generate a program that meets the **Program Description** but violates several **Program Requirements**, causing a *significant* loss in points. Please carefully review the Rubric to avoid this circumstance.

The purpose of this assignment is to reinforce concepts related to string primitive instructions and macros (CLO 3, 4, 5).

1. Designing, implementing, and calling *low-level I/O procedures*
2. Implementing and using *macros*

What you must do

Program Description

Write and test a MASM program to perform the following tasks (check the Requirements section for specifics on program modularization):

- Implement and test two **macros** for string processing. These macros should use Irvine's `ReadString` to get input from the user, and `WriteString` procedures to display output.
 - `mGetString`: Display a prompt (*input parameter, by reference*), then get the user's keyboard input into a memory location (*output parameter, by reference*). You may also need to provide a count (*input parameter, by value*) for the length of input string you can accommodate and a provide a number of bytes read (*output parameter, by reference*) by the macro.
 - `mDisplayString`: Print the string which is stored in a specified memory location (*input parameter, by reference*).
- Implement and test two **procedures** for signed integers which use string primitive instructions
 - `ReadVal`:
 1. Invoke the `mGetString` macro (see parameter requirements above) to get user input in the form of a string of digits.
 2. Convert (using string primitives) the string of ASCII digits to its numeric value representation (SDWORD), validating the user's input is a valid number (no letters, symbols, etc).
 3. Store this one value in a memory variable (*output parameter, by reference*).

- `WriteVal` :
 1. Convert a numeric SDWORD value (*input parameter, by value*) to a string of ASCII digits
 2. Invoke the `mDisplayString` macro to print the ASCII representation of the SDWORD value to the output.
- Write a test program (in `main`) which uses the `ReadVal` and `WriteVal` procedures above to:
 1. Get 10 valid integers from the user. Your `ReadVal` will be called within the loop in `main`. Do not put your counted loop within `ReadVal`.
 2. Stores these numeric values in an array.
 3. Display the integers, their sum, and their average (truncated to its integer part).
- Your `ReadVal` will be called within the loop in `main`. Do not put your counted loop within `ReadVal`.

Program Requirements


1. User's numeric input **must** be validated the hard way:
 - a. Read the user's input as a string and convert the string to numeric form.
 - b. If the user enters non-digits other than something which will indicate sign (e.g. '+' or '-'), or the number is too large for 32-bit registers, an error message should be displayed and the number should be discarded.
 - c. If the user enters nothing (empty input), display an error and re-prompt.
2. `ReadInt`, `ReadDec`, `WriteInt`, and `WriteDec` are **not allowed** in this program.
3. Conversion routines **must** appropriately use the `LODSB` and/or `STOSB` operators for dealing with strings.
4. All procedure parameters **must** be passed on the runtime stack using the **STDCall** calling convention (see [Module 6, Exploration 2 - Passing Parameters on the Stack](https://canvas.oregonstate.edu/courses/1879149/pages/exploration-2-passing-parameters-on-the-stack) (<https://canvas.oregonstate.edu/courses/1879149/pages/exploration-2-passing-parameters-on-the-stack>)). Strings also **must** be passed by reference.
5. Prompts, identifying strings, and other memory locations **must** be passed by address to the macros.
6. Used registers **must** be saved and restored by the called procedures and macros.
7. The stack frame **must** be cleaned up by the **called** procedure.
8. Procedures (except `main`) **must not** reference data segment variables by name. There is a **significant** penalty attached to violations of this rule. Some global constants (properly defined using EQU, =, or TEXTEQU and not redefined) are allowed. These **must** fit the proper role of a constant in a program (master values used throughout a program which, similar to `HI` and `LO` in Project 5)
9. The program **must** use *Register Indirect* addressing for integer (SDWORD) array elements, and *Base+Offset* addressing for accessing parameters on the runtime stack.
10. Procedures **may** use local variables when appropriate.
11. The program **must** be fully documented and laid out according to the [CS271 Style Guide](https://canvas.oregonstate.edu/courses/1879149/files/94053695/download?wrap=1) (<https://canvas.oregonstate.edu/courses/1879149/files/94053695/download?wrap=1>). This includes a complete header block for identification, description, etc., a comment outline to explain each section of code, and proper procedure headers/documentation.

Notes

1. For this assignment you are allowed to assume that the total sum of the valid numbers will fit inside a 32 bit register.
2. We will be testing this program with positive ***and*** negative values.
3. When displaying the average, only display the integer part (that is, drop/truncate any fractional part).
4. Check the [Course Syllabus](#) (<https://canvas.oregonstate.edu/courses/1879149/files/94254851/download?wrap=1>) for late submission guidelines.
5. Find the assembly language instruction syntax and help in the [CS271 Instructions Guide](#) (<https://canvas.oregonstate.edu/courses/1879149/files/94053691/download?wrap=1>).
6. To create, assemble, run, and modify your program, follow the instructions on the course [Syllabus Page](#) (<https://canvas.oregonstate.edu/courses/1879149/assignments/syllabus>)'s "Tools" tab.

Resources

Additional resources for this assignment

- [Project Shell with Template.asm](#)
(<https://canvas.oregonstate.edu/courses/1879149/files/94047843/download?wrap=1>) 
(https://canvas.oregonstate.edu/courses/1879149/files/94047843/download?download_frd=1)
- [CS271 Style Guide](#) (<https://canvas.oregonstate.edu/courses/1879149/files/94053695/download?wrap=1>)
- [CS271 Instructions Reference](#)
(<https://canvas.oregonstate.edu/courses/1879149/files/94053691/download?wrap=1>)
- [CS271 Irvine Procedure Reference](#)
(<https://canvas.oregonstate.edu/courses/1879149/files/94053692/download?wrap=1>)

What to turn in

Turn in a single .asm file (the actual Assembly Language Program file, not the Visual Studio solution file). File must be named "Proj6_ONID.asm" where ONID is your ONID username. Failure to name files according to this convention may result in reduced scores (or ungraded work). When you resubmit a file in Canvas, Canvas can attach a suffix to the file, e.g., the file name may become Proj6_ONID-1.asm. Don't worry about this name change as no points will be deducted because of this.

Example Execution

User input in this example is shown in ***boldface italics***.

Please provide 10 signed decimal integers.
Each number needs to be small enough to fit inside a 32 bit register. After you have finished inputting the raw numbers I will display a list of the integers, their sum, and their average value.

Please enter an signed number: *156*
Please enter an signed number: *51d6fd*
ERROR: You did not enter a signed number or your number was too big.
Please try again: *34*
Please enter a signed number: *-180*
Please enter a signed number: *115616148561615630*
ERROR: You did not enter an signed number or your number was too big.
Please try again: *-145*
Please enter a signed number: *5*
Please enter a signed number: *+23*
Please enter a signed number: *51*
Please enter a signed number: *0*
Please enter a signed number: *56*
Please enter a signed number: *11*

You entered the following numbers:
156, 34, -180, -145, 5, 23, 51, 0, 56, 11
The sum of these numbers is: 11
The truncated average is: 1

Thanks for playing!

Extra Credit (Original Project Definition must be Fulfilled)

To receive points for any extra credit options, you **must** add one print statement to your program output **per extra credit** which describes the extra credit you chose to work on. You **will not receive extra credit points** unless you do this. The statement must be formatted as follows...

--Program Intro--

**EC: DESCRIPTION

--Program prompts, etc--

Extra Credit Options

1. Number each line of user input and display a running subtotal of the user's valid numbers. These displays must use `WriteVal`. (1 pt)
2. Implement procedures `ReadFloatVal` and `WriteFloatVal` for floating point values, using the FPU. These must be in addition to `ReadVal` and `WriteVal` and you must have a separate code block to demo them (one 10-valid entry loop to demo `ReadVal` / `WriteVal` and one 10-valid-entry loop to demo `ReadFloatVal` and `WriteFloatVal`). (4pts)

Grading criteria

Please view the rubric attached to this assignment to understand how your assignment will be graded. If you have any questions please ask on the course discussion board.

Project 6 Rubric