KAIST

EE 209: Programming Structures for EE

Assignment 4: Assembly Language Programming

Purpose

The purpose of this assignment is to help you learn about computer architecture, assembly language programming, and testing strategies. It also will give you the opportunity to learn more about the GNU/Unix programming tools, especially `bash`, `emacs`, `gcc209`, and `gdb` for assembly language programs.

A Desk Calculator Program in Assembly Language

Part a: Implement basic functions of dc

`dc` (*d*esk *c*alculator) is a tool on Unix-like operating systems. In its simplest form, `dc` reads a list of numbers from the standard input (stdin) and uses a set of command keys to display results of user-specified operations on the standard output (stdout).

In `dc`, the operands (numbers) and operators are added in reverse-polish (also known as postfix) notation. In this scheme, the operator follows the operands. The following example execution run explains how `dc` is used.

```
343223
+
p
343790
q
```

`dc` uses a stack to store numbers in LIFO order (last-in, first-out). Whenever it encounters an arithmetic operator, it first pops out the last 2 operands from the stack, runs the operation on those numbers and then pushes the result back into the stack. In the example above, `567` and `343223` are pushed in the stack one after the other. Once the operator `'+'` is entered, `dc` first pops `343223` and then `567` from the stack. It then adds the two integers and finally pushes the result (`343790`) back in the stack. The command `p` is used to print the value that sits on the top of the stack. Please note that `p` only retrieves the value without popping (this is also known as a peek operation). The user can either quit the program by entering `q` or `EOF` character to the program. In other words, if the annotated text mentioned above is stored in a file named values.txt then `dc` can also be executed in the following manner:

```
$ dc < values.txt
```

which will print the result to the standard output stream as:

```
343790
```

The `dc` tool supports a number of operators and subsidiary commands which you can study on the man page. For this assignment, you are required to implement only the following operations.

- Printing operator: `p`
- Arithmetic operators: `+, -, *, /, %, ^`
    - `%` performs remainder operation.

    - ^ performs exponentiation. (e.g., 2^4 = 16). You don't need to implement negative exponent.
- Terminating operator: `q`

To make the assignment tractable in assembly programming, we make some simplifying assumptions:

- You can assume all operands are 32-bit signed integers.
- The result of an arithmetic operation is in the range of a 32-bit signed integer.
- You do not need to handle an overflow/underflow. That is, it is OK not to handle it correctly even if the result of an operation exceeds the range of a 32-bit signed integer.
- You can assume that each input line has only 1 operand/operator.
- You do not need to handle input errors (but your program should not crash with invalid input). See below for more detail on error handling.

You might want to use `esp` and `ebp` registers to simulate the stack in the main function.

We are providing a startup file which contains the pseudo-code of dc.s file. Please go through the pseudo-code before you begin writing the program. It is acceptable to use global (i.e. bss section or data section) variables in `mydc.s`. Please make sure that you create your own function to implement the power (^) arithmetic operator. In `dc`, negative numbers can be added by pre-appending `'_'` symbol to the number. For example

```
_4
3
-
p
```

```
-7
q
```

calculates "-4 - 3", prints the top value (p), and quit the program (q).

Part b: Advanced functions

The dc tool also provides additional operations that manipulate the input. You are required to implement the following operators for this assignment.

Advanced Operations	**Short decription**
f	Prints the contents of the stack in LIFO order. This is a useful command to use if the user wants to keep track of the numbers he/she has pushed in the stack.
c	Clears the contents of the stack.
d	Duplicates the top-most entry of the stack and pushes it in the stack.
r	Reverses the order of (swaps) the top two values on the stack.

Please note that 'f' does **not** pop out any numbers out of the stack. The following example run of dc shows how a combination of different dc operators can be used:

```
53
48
35
+
+
343223
43
56
76
35
98
1
f
1
```

```
98
35
76
56
43
343223
136
q
```

dc keeps on pushing the integers on the stack (53, 48, 35) till it encounters the first '+' operator. It pops out 35 and 48, computes the addition and inserts 83 back in the top of the stack. When the second '+' is inserted, it repeats the same process with the integers 83 and 53 and inserts back 136 in the empty stack. Later when 'f' is entered, dc prints out all the contents of the stack in LIFO order.

The following self-explanatory example shows how one can use 'd' in dc

```
4
d
*
p
16
q
```

Finally, 'r' is used to reverse the order of (swaps) the top two values on the stack as is shown in the example below:

```
4
8
f
8
4
r
f
4
8
q
```

Error Handling

You are required to implement basic error handling and ensure that your program does not crash with any given input (except for one case: it is OK to crash if `dc` has to divide by 0). Your program should ignore those input values that have mixed alphanumeric characters. You should check whether the stack has at least two operands for `+, -, *, /, %, ^` operations. In case there are not enough operands, `dc` should print out '`dc: stack empty`' to standard output. For `p, d, r` operators, `dc` should again print '`dc: stack empty`' if the stack does not contain at least one operand (two for `r`). For all other operators, `dc` should do nothing if the stack is empty.

Logistics

Develop on lab machines. Use `emacs` to create source code. Use `gdb` to debug.

Do not use a C compiler to produce any of your assembly language code. Doing so would be considered an instance of academic dishonesty. Instead produce your assembly language code manually.

We encourage you to develop "flattened" pseudo-code (as described in precepts) to bridge the gap between the given pseudo-code and your assembly language code. Using flattened pseudo-code as a bridge can eliminate *logic* errors from your assembly language code, leaving only the possibility of *translation* errors.

We also encourage you to use your flattened pseudo-code as comments in your assembly language code. Such comments can clarify your assembly language code substantially.

Your `readme` file should contain:

- Your name and student ID.
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated, as prescribed by the course "Policy" Web page.
- (Optionally) An indication of how much time you spent doing the assignment.
- (Optionally) Your assessment of the assignment.
- (Optionally) Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Submission

Use KAIST KLMS to submit your assignments. Your submission should be one gzipped tar file whose name is

YourStudentID_assign4.tar.gz

Your submission need to include the following files:

- Your `mydc.s` file.
- A `readme` file.
- NEW: observance of ethics document
- (Optionally) Any data file you tested your `mydc` program with.

Please do not submit `emacs` backup files, that is, files that end with '~'.

Grading

As always, we will grade your work on quality from the user's and programmer's points of view. To encourage good coding practices, we will deduct points if `gcc209` generates warning messages.

Comments in your assembly language programs are especially important. Each assembly language function -- especially the `main` function -- should have a comment that describes what the function does. Local comments within your assembly language functions are equally important. Comments copied from corresponding "flattened" C code are particularly helpful.

Your assembly language code should use `.equ` directives to avoid "magic numbers." In particular, you should use `.equ` directives to give meaningful names to:

- Enumerated constants, for example: `.equ TRUE, 1`.
- Parameter stack offsets, for example: `.equ OADDEND1, 8`.
- Local variable stack offsets, for example: `.equ UICARRY, -4`.
- Stack offsets at which callee-save registers are stored, for example: `.equ EBXOFFSET, -4`.