MP2

MP 2 - Stack calculator

In this programming assignment, you will write an LC-3 program capable of evaluating postfix expressions (aka **reverse polish notation (http://en.wikipedia.org/wiki/Reverse_Polish_notation)**.) using a stack. A postfix expression is a sequence of numbers (`1`, `5`, etc.) and operators (`+`, `*`, `-` , etc.) where every operator comes after its pair of operands. For example `3 + 2` would be represented as `3 2 +` in postfix. The expression `(3 - 4) + 5` with 2 operators would be `3 4 - 5 +` in postfix. Notice that a nice feature of postfix is that the parentheses are not necessary, which makes the expressions more compact.

In Friday's Lab, we will work on the code to handle the input typed in by the user at the keyboard. You will have to develop the subroutines for evaluating the expressions using a stack yourself and printing it out in hexadecimal format.

The Pieces

Evaluating the Expression:

In your Lab assignment, you should have parsed the input expression from the keyboard, **echoed it to the screen** and called a subroutine with the input. In this MP you will write code to use a stack to evaluate the expression. If the read value is an operand, push it onto the stack. If the read value is an operator, pop two values from the stack, apply the operator on the two values and push the result back on the stack. Keep repeating this for every value read from keyboard and **stop when you reach the ASCII value of the `=` (equal sign) character**. If the last value is an operand, that means the expression was invalid. If there is a stack underflow during any point in this process, then the expression was invalid. Do not worry about stack overflows. Look at the flowchart below for a detailed description of the algorithm (click it to see in full detail).

For example if the input the user entered on the keyboard was `3 4 - =`, then the stack through the evaluation of the expression will be:

Start	Read "3"	Read "4"	Read "-"
	(push x0003)	(push x0004)	(pop two, calculate and push)
		x0004	

	x0003	x0003	xFFFF

Examples

Input	Solution in decimal (stored in R5 as binary)	Hexadecimal output on screen
4 5 + 3 * 7 -=	20 (0000000000010100)	0014
5 2 8 * + 3 -=	18 (0000000000010010)	0012
5 1 2 + 4 * + 3 -=	14 (0000000000001110)	000E

Printing in Hexadecimal:

You will need to write the `PRINT_HEX` subroutine to print the value of the evaluated expression stored in `R5` to the screen in hexadecimal format. Follow the algorithm in the Hex Printing Flowchart (see Lab 1). The flowchart assumes `R3` as the input, you can use any register. However, the correct output value should still be intact in `R5` to get full credit. **You can reuse the code you developed in Lab1/MP1**. Continuing with our example `3 4 +` , the solution of the expression is `7`, so the output printed to the screen should be `0007`.

The exact output format should be

```
1 2 4 + 5 + + =000C
```

```
1 2 3 4 + = Invalid Expression
```

Noted that the inputs like `1 2 4 + 5 + +` should also be printed.

Details

You must use subroutines with `JSR` and `RET` (`JSRR` and `JMP` could be used instead, but are not recommended unless necessary).

If you find yourself in a nested subroutine(calling a subroutine from within another subroutine) remember to save `R7`. Here (https://mediaspace.illinois.edu/media/1_mkpxg1se) is a short video explaining why.

You have been provided with the implementation of a stack in `prog2.asm`. Use the given push and pop subroutines to carry out the push and pop operations. Implement the `EVALUATE` subroutine and all the code necessary to make your calculator work.

- Your code will be tested with only non-negative single digit input integers (0-9) in the expression. The output and intermediate values on the stack could be positive or negative integers.
- You do not have to handle overflow or underflow of registers.

- You may assume that the input expressions will have a space between operators and operands. Your code should not be outputting a space to the screen unless the user inputs a space!
- The output of the expression should be stored in R5.

These subroutines are suggested. They are listed to provide you with guidance on how to solve this MP. You can implement the MP any way you want. However, as mentioned above you must have subroutines with `JSR` and `RET`.

- `EVALUATE`: This subroutine for each input value (echo to screen), calls the push, pop, add, subtract, multiply, divide and power after evaluating if the value read from the memory is an operator or operand and if it is an operator, which kind. If the stack underflows or if after evaluating the whole expression the stack has more than one value, print "Invalid Expression" to the screen and halt.
- `PUSH`: This subroutine has been given to you.
- `POP`: This subroutine has been given to you.
- `ADD`: Adds the two operands.
- `SUBTRACT`: Performs subtraction.
- `MULTIPLY`: Performs multiplication.
- `DIVIDE`: Performs division. In case the division results in a remainder, just return the integer quotient. The input values will always be positive integers.
- `POWER`: Performs power operation. You may assume that the register storing the result will not overflow. The input values will always be positive integers.
- `PRINT_HEX`: This subroutine prints the output of the input expression in its hexadecimal representation.

Grading and Testing

To test, assemble your code with `lc3as` and run it with `lc3sim`. The final answer should be stored in `R5`.

Functionality (90%)

The program should have the correct contents in the `R5` for the test cases as well as print the "`Invalid Expression`" message when an invalid expression is provided as input.

Style, comments, clarity, and write-up (10%)

- Comments and Style - 5%
- Intro Paragraph - 5%

If your code does not assemble you will get 0.

Common errors and their point deductions:

- Not storing result in R5: -10
- Not handling spaces correctly: -45
- Incorrect prompt method: -10

Tips and Resources

- Input should be echoed to the screen
- Test cases: MP2_strong_testcases.docx (https://canvas.illinois.edu/courses/47191/files/12877579?wrap=1) ↓ (https://canvas.illinois.edu/courses/47191/files/12877579/download?download_frd=1)
- Flowcharts:
  - MP2_mainflow-1.pdf (https://canvas.illinois.edu/courses/47191/files/12877578?wrap=1) ↓ (https://canvas.illinois.edu/courses/47191/files/12877578/download?download_frd=1)
    - SP23+MP2+Old+Flowchart-1.doc (https://canvas.illinois.edu/courses/47191/files/12877577?wrap=1) ↓ (https://canvas.illinois.edu/courses/47191/files/12877577/download?download_frd=1)