Implementation of a Calculator

Junpeng Lu and Jungang Fang
Prof. Billoo
ECE 251
05/06/2019

Requirements:

Our program will implement a basic calculator that takes into account the standard "PEMDAS" rule. The calculator will be invoked by the following:

$> ./calc <operation>

where operation is defined as:

operation = <operand> <arithmetic operation> <operand> , ...

up to 4 times.

For example, the program will support the following invocation:

$> ./calc 3+4*2-3

$> 8

There can be fewer than 4 operations.

For example:

$> ./calc 3+4

$> 7

The parentheses portion of PEMDAS will also be supported:

$> ./calc (3+4)*2-3

$> 11

Architecture:
Our overall structure is described as below.

1. Read input from the argument byte by byte in order to read one character (token) at a time, using “ldrb”. Read until a null character is reached (end of input).
   1.1 If the token is:
      1.1.1 A number or a dot: store it in the temporary num-char array.
      1.1.2 A left parenthesis
         1) Convert the contents in num-char array, if any, to a float (using sscanf) and store it to the value stack (s10-s14).
         2) Clear the num-char array by setting all its bytes to null characters
         2) Push the left parenthesis onto the operator stack.
      1.1.3 A right parenthesis:
         1) Convert the contents in num-char array, if any, to a float (using sscanf) and store it to the value stack (s10-s14).
         2) Clear the num-char array by setting all its bytes to null characters
         3) While the character on top of the operator stack is not a left parenthesis,
            a) Pop the operator from the operator stack.
            b) Pop the value stack twice, getting two operands.
            c) Apply the operator to the operands, in the correct order.
            d) Push the result onto the value stack.
         4) Pop the left parenthesis from the operator stack
      1.1.4 An operator (call it thisOp):
         1) Convert the contents in num-char array, if any, to a float (using sscanf) and store it to the value stack (s10-s14).
         2) Clear the num-char array by setting all its bytes to null characters
         3) While the operator stack is not empty, and the top character on the operator stack has the same or greater precedence as thisOp,
            a) Pop the operator from the operator stack.
            b) Pop the value stack twice, getting two operands.
            c) Apply the operator to the operands, in the correct order.
            d) Push the result onto the value stack.
         4) Push thisOp onto the operator stack.
2. While the operator stack is not empty,
   1) Pop the operator from the operator stack.
   2) Pop the value stack twice, getting two operands.
   3) Apply the operator to the operands, in the correct order.
   4) Push the result onto the value stack.

3. At this point the operator stack should be empty, and the value stack should have only one value in it, which is the result. Convert the result to double precision (using fcvtds), store it in two regular registers (using vmov) and print it (using printf).

Note: This algorithm allows for more than four operations. It can take a large amount of operations as long as not too many parentheses are used within another.

Challenges:

1. Large variation of input cases
   Since there are so many different combinations of inputs, it is impossible to hardcode each situation. As a result, two stacks are used, one as an array and one as a collection of registers, to store the operators and values used in the algorithm.

2. Working with floats
   Since the calculator has to deal with floating numbers, the normal registers cannot be used to store numbers. After doing research, it was decided to use v-type registers (single-precision), because this type register can handle floating numbers, and is well-documented. In addition, there was a problem with printing floats since the printf function does not accept v-type registers as its input. As a result, we decided to convert the result from single precision to double precision (using fcvtds and a d-type register) and then split double precision result into two r-type registers (using vmov). After this process, printf can be called with "%f" as the format and successfully print the result.

3. Converting strings to floats
   Since the input arguments are all chars, we have to convert the chars to a floating number. After some research, the sscanf function was used. The function takes three parameters: the addresses of input char array, the desired format (%f in this case), and the address to store the converted floating number. After using the correct parameters, the char array was converted to a floating number successfully.

4. Attempts at implementing a value stack
   At first, we decided to use an array as a stack for the floating values. During the implementation however, it was realized that there was a problem with storing floating as an array in ARM. After storing the second element in the array, the first element in the array would be overwritten, rendering the array ineffective. We tried different cases to look for where bugs are. This is likely due to the way that floating numbers are stored in memory.
   As an alternative, a collection of floating registers (s10-s14) are used and implemented to act as a stack. The stack grows from s10 to s14. The top of the stack is kept track with a counter that contains the number of values in the stack. This implementation works as intended but is less efficient than the array implementation.

5. GDB is used extensively to debug the program. Since it is the only way to check values in registers step by step.

6. Parentheses check
   While testing the program, it was realized that the program did not return the correct result when two operations appeared in a row, for example, 2*(2+2). A corner case check was created to fix this problem.