Calculator Project

Cooper Union

ECE-251

Professor Billoo

Ali Ghuman, Xue Ru Zhou, Husam Almanakly, Layth Yassin

May 2021

Contents

1 Objective

The objective is to sort a given set of unsigned ints that are a maximum of 32 bits. One hundred values is the maximum number that can be inputted. The unsorted input is to be read in from an input file declared by the user, and the sorted output placed into another file declared by the user. The output will be a list of newline separated values.

2 Architecture

2.1 Reading in the input

Input reading was based entirely on reading in the command-line argument that gives the expression to be evaluated. This was done by obtaining a pointer to the second argument that is stored in the second set of four bytes in r1 at the beginning of the program. This null terminated string was then loaded into an array for use later in the program.

2.2 Infix to Postfix Algorithm

To account for order of operations, an algorithm to convert from an infix expression to a postfix expression was implemented first.

The general idea of this algorithm is to convert and express such as "3+4*2" into "342*+" so that the evaluation algorithm could evaluate the expression inside out.

First, in `initialize_postfix`, the index counters were initialized to the value 0. The null character was pushed into the stack as a marker which we can retrieve to confirm that the stack is empty.

The first step in the algorithm is to iterate over the input array and detect whether the element is an operator or an operand. In `read_exp`, the element is compared with the operator symbols. If the element is detected to be any of these operators, it would branch to the corresponding branch, which contains instructions that differ depending on the operation precedence.

For example, if the detected element is a "+" or "-" operator, then it will branch to `add_sub`. Likewise, if the detected element is "(", it will branch to `left_parenthesis`. If the element is not any of the specified operators, then it gets automatically appended to the postfix array and the counters for both the input array and postfix array are incremented.

In the smaller branches, a set of instructions are carried out depending on the operator precedence.

Exponents do not have their own branch because they get pushed to the stack as soon as they get detected.

If a left parenthesis is detected, branch to `left_parenthesis`, where the infix array index is incremented and then the current element is pushed onto the stack. Branch back to `read_exp` to read in the next element. This also did not require a branch by itself (just like the exponent), but originally, we were thinking of adding an error for missing operator between sets of parentheses. That idea was ultimately thrown away because ")", right parenthesis, never gets pushed to the stack so this is more complicated to implement then expected.

If a right parenthesis is detected, branch to `right_parenthesis`. If the stack is empty (first element of stack is null char), then print the missing parenthesis error as there is no left parenthesis in the stack. Otherwise, pop the operands off the stack and load them into the next empty element in the postfix array. Increment the index of the postfix array. Loop until "(" is detected and discard both parentheses. In this way, the expression inside the parentheses are always evaluated first.

If "*" or "/" is detected, branch to `mul_div`. "*" and "/" always gets pushed to the stack unless the first element in the stack is a carrot or "*", "/". In the exponent case, the carrot gets popped off first and appended to the postfix array before the current element ("*" or "/") gets pushed to the stack. This makes sense because the exponents need to be evaluated before multiplication or division. Similarly, because multiplication and division is evaluated from left to right, if an operator of the same precedence is found on top of the stack, then it gets popped off and appended first.

If "+" or "-" is detected, branch to `add_sub`. Only push the current element if the stack is empty or contains a "(" as the top element. Otherwise, pop off all the elements in the stack before pushing the current element. This works because addition and subtraction have the lowest precedence. It also accounts for left to right addition and subtraction. Lastly, branch back to `read_exp` to read in the next element of the infix expression.

Once all the elements of the infix array are read in, the stack is emptied in `empty_stack` and the program moves on to evaluate the expression.

2.3 Print Error Message

The error `err_missing_parentheses` prints "ERROR: *Parenthesis Syntax*" if there parentheses are unbalanced.

2.4 Evaluation Algorithm

The conversion to postfix notation allowed for a quick and simple method of evaluation. The algorithm invovled parsing through the postfix expression and evaluating "inside out". The call stack was utilized to easily keep track of the operands. The process was as follows: if an operand was read, it was immediately pushed to the call stack. If an operator was read, two values would be popped from the stack and their sum evaluated, pushing the result back onto the stack.

When an operator was read, firstly the algorithm determined which operator it was. Then the "evaluation" would follow. Two items would be popped from the stack into the double registers (d0 and d1 for example) and the operation that was predetermined could thus be conducted. The result would be pushed back onto the stack and the iteration would continue until the end of the array was reached. After completing the iteration, the value at the top of the call stack would be popped and printed as the final result of the calculation.

3 Design Considerations

3.1 Errors

Some numbers such as 4.29 will return a segment fault because parts of the binary representation of 4.29 coincides with that of the character "(". There may be other numbers that will throw a similar error. As far as our design goes, we did not account for these rare coincidences.

3.2 C Functions

Throughout the program, existing C std library functions are leveraged to complete difficult tasks. To output the expression that the user inputs, error messages, and the evaluation of the user-input expression, the std library function `printf()` is used. The `printf()` function outputs to the stdout (i.e., the terminal), which allows the user to see the printed outputs. Writing the code for a function that accomplishes the same tasks as `printf()` in assembly is complex and impractical. This is mainly because several cases and test conditions would have to be accounted for. In addition, writing a `printf()` function without the use of any libraries would require knowledge of machine code.

The user-input expression is taken in as a string. It is then processed to convert the chars representing numbers to double precision floating point numbers. The `sscanf()` function is ideal for this system because it takes in an input string and format specifiers. In this system, the function took in the user-input expression, the %lf format specifier, a variable to temporarily store the double, and a variable to keep track of the index of what was last processed in from the input. This allowed for the smooth conversion of the chars to doubles. Writing a function to accomplish this task in assembly would be difficult for similar reasons as the `printf()` function.

To complete the task of evaluating an exponent, the `pow()` function is used. The `pow()` function takes in two parameters: a floating point base value and a floating point exponent value. The function returns the result of raising the base to the power of the exponent value as a double floating-point number. This could have been implemented from scratch by creating a branch where the value is multiplied by itself n times, and looping through the branch n times, where n is the value of the exponent. However, this would only work for exponents with integer values. The code would become complicated to implement if floating-point numbers were to be supported. Since it is required for the system to support floating-point numbers, the use of `pow()` is necessary.

4 Problems and Solutions

4.1 Conversion to Floating Point Numbers

The first major issue throughout this project was implementing a way to support floating point numbers. This involved devising a way to convert the inputted numbers from a string to a floating point number. It was ultimately decided that every number would be converted to a float, to allow for any case when a float was input. The C functions `atof()` and `strtof()` were considered, however proved difficult in parsing the input and converting each individual number. This was mainly due to the fact that these functions would not return a value indicating how many characters it converted to a float before hitting an operator.

As a result, the C function `sscanf()` was chosen. The arguments passed in would include the array holding the characters input by the user, the scan pattern used "%s%n", a temporary variable to hold the converted float, and a variable to hold the number of characters read in. This proved useful as the initial input was stored as a string of characters, `sscanf()` provided an easy way to keep track of the index in the original array.

4.2 Double Precision Numbers

The next major issue involved handling and managing double precision numbers. As double precision floating point numbers are 8 bytes rather than 4 bytes, this introduced a conflict with the standard ARM registers as they can only hold 32 bits. In order to get around this issue, the ARM commands "ldmia", "stmia", as well as "vldr" and "vstr" were all used to handle and move the numbers. The first two commands involved splitting the memory among two registers ("ldmia" and "stmia") while the second were mad specifically for double precision numbers.

This further created an issue with the command line used to build the assembly code. New flags were needed to be included for the processor to allow the floating point instructions in the code.

4.3 Storing Items in One Array

The last major issue in coding the calculator involved the methods for storing and handling the expression. As the program was intended to convert the expression to postfix, to allow for an easy evaluation, having the operators and operands in the same array proved helpful. This however became a slight issue once the numbers were converted to floating point doubles, as they are 8 bytes long, while the operators were each just 1 byte.

This was ultimately resolved by simply storing both in the same space in memory with 8 byte slots in the array. Whenever an operator was needed to be accessed, the commands "ldrb" and "strb" were used to only utilize the first byte, where the operators were ensured to be stored. While this worked and allowed for the conversion to postfix notation to run smoothly, it made it difficult to error check throughout the project. Rather than simply printing the entire array, a separate branch was needed to simply loop through the array and print each item (separate scan patterns for char's and floating point doubles).

5 Deliverables

5.1 README

To build and run the assembly code, use command "make all" while on the micro-lab server, and it should create an executable named calc. To execute, in the current directory, run ./calc followed by the calculation requested. For example, ./calc 4+2/10 would evaluate to 4.2000. If parenthesis are involved in the calculation, include quotation marks around the arguments as well, ie ./calc "20/(4+3)". Spaces should be omitted between characters in the expression, and the program limits input to 4 operations.

5.2 MakeFile

The Makefile automatically removes a pre-existing executable named "calc" and builds the given source code. If no file named calc exists, the remove command does nothing and the source code continues to build the executable normally.

```
# Makefile

all: calc

calc: proj3.S .clean
   arm-linux-gnueabi-gcc $< -o $@ -ggdb3 -static -mfloat-abi=hard -mfpu=vfp -lm

.clean:
   rm -f calc
```

The flags "-mfloat-abi=hard -mfpu=vfp -lm" were added to support floating point calculations and commands in ARM.

6 Conclusion

The input was successfully sorted and performed error checking to make sure the program functioned appropriately. An error was returned if the files could not be opened, if the names of the files were too long, or if more than 100 values were inputted. Both positive and negative integer values were sorted, as unsigned ints were utilized. Deeper knowledge was attained concerning the workings of registers and C functions utilized in assembly. As many C functions were used, testing with different values and researching various methods of implementation led to greater understanding of how these functions were employed and which tools were the best for the given task. Lastly, by experimenting with various methods of reading input and writing output, debugging numerous segmentation faults, and finally arriving at success, understanding of computer architecture was heightened.