Project 3 – Calculator

ECE251 – Computer Architecture

Prof. Billoo

The Cooper Union

Spring 2020

Daniel Mezhiborsky

Introduction

The project description required an ARM assembly program to be written to satisfy the following requirements:

- Accept input as a command-line argument containing a mathematical expression to be evaluated
- Support up to four operations total
- Support the basic operations *,/,+,-,^ as well as parentheses
- Support floating-point numbers

Given the constraints, the following behavior was implemented:

- The program will check for and will only run if it has exactly one argument passed to it (besides its filename).
  - The argument does not need to contain spaces. Any spaces included will be ignored.
- The program implements the five specified operations and supports parentheses, even where the parentheses are not necessary for the calculation.
- Floating point numbers are supported and are double-precision
- Where possible, any errors in the input are marked.
- When there is an error, a descriptive error message is returned as well as a unique return code, but when the run is successful, only the final result is returned.

The following report contains a breakdown of the source code with key methodology explained. All code referenced is from the included calc.S and ccalc.c source files.

Key Specifications

- Language: 32-bit ARM assembly, EABI5 version 1
- Assembler: cross-gcc (arm-linux-gnueabi-gcc) 7.4.0
- Environment: qemu-arm 2.11.1 on Ubuntu18.04.4 LTS

Files

- calc.S – main source file
- ccalc.c – assistive C implementation
- Makefile
- README

External Functions

Two external functions were used explicitly:

- **sscanf()**: Used to read the input data. It writes the floating-point number as a double directly into memory. The position of the first non-parsable character is recorded.
- **pow()**: From **math.h**. Used to carry out the exponentiation function. Works reliably from the compiled C version, but may not work in assembly without the correct hard-float toolchain and libraries.

C Implementation

A C implementation of the program was written in order to facilitate the development of a method for parsing and evaluating the expression. The implementation was used for rough guidance on the structure of the assembly as well, and much of the program logic is directly analogous (the if statements for parsing in particular have shared labels of the format COND #). A few key differences between them are:

- the C version has been more extensively tested with edge cases
- the C version has a reliably-working pow() implementation
- the assembly version has additional edge-case checking related to parentheses
- the assembly version uses different methods for addressing and interacting with the data; most notably the use of pointers instead of array counters

```c
/*  ECE251-Computer Architecture
            Daniel Mezhiborsky
           Prof. Billoo
        Calculator

filename: calc.c - Generic C implementation of 5-function PEMDAS calculator (+,-,*,/,^)

*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void badInputExit();
void eval_single();
void parensexit();
void tooManyOpsExit();

double operands[5] = {0.0,0.0,0.0,0.0,0.0}; //operands stack
char operators[5];  //operators stack
int operand_count=-1;
int operator_count=-1;
int offset;
int lastType=-1;
int next_negative=0;
int operations_count=0;

int main(int argc, char *argv[]) {
    char *input = argv[1];

//  printf("%c\n", *argv[1]);
    if (argc!=2) {
        printf("Error! Invalid number of arguments. Please put your expression in quotes!");
        exit(9);
    }
    while(1) {
        //COND 1
        if (*input==0) break;
        //COND 2
        if (*input==' ') {
            input++;
            continue;
        }
        //COND 3
```

```
        else if (*input=='.' || (*input>='0' && *input<='9')) {
            operand_count++;
            if ((sscanf(input, "%lf%n", &operands[operand_count], &offset) != 1)) {
                printf("oof\n");
            }
            if (next_negative==1) {
                operands[operand_count]=-operands[operand_count];
                next_negative=0;
            }
            lastType=1;
            input += offset;
            continue;
        }
        //COND 4
        else if (*input=='(') {
            input++;
            operator_count++;
            operators[operator_count]='(';
            lastType=0;
            continue;
        }
        //COND 5
        else if (*input==')') {
            while (operators[operator_count] != '(') {
                if (operator_count==-1) parensexit();
                eval_single();
            }
            operator_count--;
            input++;
        }
        //COND 6
        else if (*input=='-' && lastType<1 && next_negative==0) {
            next_negative=1;
            input++;
            continue;
        }
        //COND 7
        else if (operations_count==4) {
            printf("Max 4 operations allowed, stopping early at %d\n", (int)(input-argv[1]));
            break;
        }
        //COND 8
        else if (lastType<1) badInputExit(argv[1], input);

        //COND 9
        else if (*input=='+'||*input=='-') {
            while(operator_count!=-1 && operators[operator_count]!='(') {
                eval_single();
            }
            operator_count++;
            operations_count++;
            operators[operator_count]=*input;
            input++;
            lastType=0;
            continue;
        }
        else if (*input=='/'||*input=='*') {
            while(operator_count!=-1 && (operators[operator_count]!='+' && operators[operator_count]!='-
') && operators[operator_count]!='(') {
                eval_single();
            }
            operator_count++;
            operations_count++;
            operators[operator_count]=*input;
            input++;
            lastType=0;
            continue;
        }
```

```
        else if (*input=='^') {
            operator_count++;
            operations_count++;
            operators[operator_count]='^';
            input++;
            lastType=0;
            continue;
        }
        else badInputExit(argv[1], input);      //means the input is unrecognized
    }
    if (lastType<1 /*operators[operator_count]!='('*/) badInputExit(argv[1], input-1);
    while(operator_count!=-1) eval_single();
    printf("%lf\n", operands[0]);
}

void badInputExit(char* full, char* input) {
    printf("Error! Invalid input: %s\n", full);
    printf("%*c\n", input-full+23, '^');
    exit(1);
}

void parensexit() {
    printf("Error: mismatched parentheses!\n");
    exit(2);
}

void tooManyOpsExit() {
    printf("Error: Input too long; too many operations (max 4).\n");
    exit(3);
}

void eval_single() {
    char op = operators[operator_count];
    if (op=='(') parensexit();
    double op1 = operands[operand_count-1];
    double op2 = operands[operand_count];
    operands[operand_count] = 0.0;
    switch (op) {
        case '+':
            operands[operand_count-1] = op1 + op2; break;
        case '-':
            operands[operand_count-1] = op1 - op2; break;
        case '/':
            operands[operand_count-1] = op1 / op2; break;
        case '*':
            operands[operand_count-1] = op1 * op2; break;
        case '^':
            operands[operand_count-1] = pow(op1, op2); break;
        default:
            break;
    }
    operand_count--;
    operator_count--;
}
```

Data Section

```
.data
/* ------------------------------------------------------ */
argv1_addr:             .space 4
operands:               .space 40
operators:              .space 20
bail:                   .space 4
operand_scan_fmt:       .asciz "%lf%n"
error_bad_parens_msg:   .asciz "Error! Mismatched parentheses!"
error_arg_count_msg:    .asciz "Error! Invalid number of arguments. Please put your expressio
n in quotes!\n"
error_bad_input_msg:    .asciz "Error! Invalid input: %s\n%*c\n"
print_result_msg:       .asciz "%lf\n"
operations_exceeded_msg:.asciz "Error: Input too long; too many operations (max 4).\n"
/* ------------------------------------------------------ */
```

The names in the data section are expressive. Clarification:

- bail: return address

Breakdown: main

```
.global main
main:
    push {r4, r5, r6, lr}           @ 16 byte aligned, but still doesn't fix pow

    ldr r3, =bail                   @ save return addr
    str lr, [r3]

    bl prepare                      @ prepare data (addresses, etc) for the parsing
    bl parse_loop_start             @ parse the data
    bl print_result                 @ eval what's left and print the result

    pop {r4, r5, r6, lr}
    bx lr
```

-> The stack is 8-byte aligned. The stack is kept 16-byte aligned here for troubleshooting pow (unsuccessful!). The return address is saved and routines are called to process and output the data.

Breakdown: Prepare

```
prepare:
    cmp r0, #2
    bne error_arg_count_exit
    ldr r0, =argv1_addr
    add r1, #4                      @ get second arg
    ldr r1, [r1]
    str r1, [r0]
    mov r4, r1                      @ r4 = input pointer
//  sub r4, #1
    ldr r5, =operands               @ r5 = operands pointer
    sub r5, #8
    ldr r6, =operators              @ r6 = operators pointer
    sub r6, #1
    //r7 loaded in parse_loop       @ r7 = current input char
    mov r8, #-1                     @ r8 = last_type
    //likewise loaded later         @ r9 = current operator
    mov r10, #0                     @ r10 = next_negative
    mov r11, #0                     @ r11 = operations_count
    bx lr
```

-> Given the limited scope and external calls of this program, registers r4–r11 were used as storage for internal variables instead of memory. Besides having the benefit of fewer memory accesses, this also reduces the length of the routines and eliminates some opportunity for error.

-> The address to the argument passed in argv is saved to memory. R4, R5,and R6 are pointers that will be incremented across the program instead of maintaining separate counter values. R8 (last_type) indicates the last type of input parsed (–1, 0, 1 -> none, operator, and operand, respectively). R10 is a flag for whether the next operand should be negative. Registers R0–R3 are left as volatile registers for the subroutines and any external function calls.

```
error_arg_count_exit:
    ldr r0, =error_arg_count_msg
    bl printf
    pop {r4, r5, r6, lr}
    mov r0, #9
    bx lr
    bx lr
```

-> Error routine for an incorrect number of arguments.

Breakdown: Parsing

```
parse_loop_start:
    push {r0, r1, r2, lr}           @ 8-byte alignment
parse_loop:
    ldrb r7, [r4]                   @ r7 = input character
    //COND 1                        @ end of string; break loop
        cmp r7, #0
            popeq {r0, r1, r2, lr}
            bxeq lr                 @ break loop
    //COND 2
        cmp r7, #' '                @ space, just increment and skip!
            addeq r4, #1
            beq parse_loop
    //COND 3
        cmp r7, #'.'                @ operand! parse and continue
            beq parse_loop_isoperand
        mov r0, #0
        cmp r7, #'0'
            addge r0, #1
        cmp r7, #'9'
            addle r0, #10
        cmp r0, #11
            beq parse_loop_isoperand
    //COND 4                        @ open parenthesis
        cmp r7, #'('
            beq parse_loop_openparen
    //COND 5
        cmp r7, #')'
            beq parse_loop_closeparen
    //COND 6
        mov r0, #0                  @ negative sign (not minus!)
        cmp r7, #'-'
            addeq r0, #1
        cmp r8, #1
            addlt r0, #10
        cmp r10, #0
            addeq r0, #100
        cmp r0, #111
            moveq r10, #1
            addeq r4, #1
            beq parse_loop
    //COND 7
        cmp r11, #4                 @ too many ops; throw error
            beq parse_loop_operations_exceeded

    //COND 8
        cmp r8, #1                  @ if the last type was 1, that means we're about to have
2 operands in a row
        blt error_bad_input_exit
```

```
    //COND 9
        mov r0, #0                  @ + & -
        cmp r7, #'+'
            addeq r0, #1
        cmp r7, #'-'
            addeq r0, #1
        cmp r0, #1
            bge parse_loop_isplusminus
    //COND 10
        mov r0, #0                  @ * & /
        cmp r7, #'/'
            addeq r0, #1
        cmp r7, #'*'
            addeq r0, #1
        cmp r0, #1
            bge parse_loop_ismuldiv


    //COND 11
        cmp r7, #'^'                @ ^
            beq parse_loop_ispow

    b error_bad_input_exit
```

-> This is a large if/then-style decision tree for parsing the input. It calls subroutines to perform the actions to enhance readability. This is based directly on the included C implementation, which is based on Dijkstra's Shunting Yard Algorithm. Each condition is labeled.

-> Each of those subroutines decides how to act upon the input and whether (and for how long) to evaluate the existing input. This does not use RPN. The same criteria for when to push operators and operations onto an output stack from the common Shunting Yard implementation is used to decide when to evaluate.

-> Criteria for negative numbers, spaces, parentheses, and unrecognized characters are included.

-> Parentheses are treated and counted as operators, but negative signs are not.

Breakdown: “Action” Subroutines

-> The aforementioned subroutines that implement the second piece of the Shunting Yard algorithm. These are for when operators are encountered:

```
parse_loop_ispow:                   @ there are no evaluation criteria because nothing is of
higher precedence than ^ and it's right-associative
    add r6, #1
    add r11, #1
    str r7, [r6]
    add r4, #1
    mov r8, #0
    b parse_loop

parse_loop_ismuldiv:                @ evaluate the existing list if the priority is less than
 or the same
    mov r1, #0
    ldrb r0, [r6]                   @ eval while we have the the correct precedence, so eval
*,/,^ but not +,-
    cmp r0, #'('                    @ stop at open parentheses
        addne r1, #1
    cmp r0, #'+'
        addne r1, #1
    cmp r0, #'-'
        addne r1, #1
    ldr r0, =operators
    cmp r6, r0
        addge r1, #1
    cmp r1, #4
        beq parse_loop_evalmuldiv

    add r6, #1
    add r11, #1
    str r7, [r6]
    add r4, #1
    mov r8, #0
    b parse_loop
parse_loop_evalmuldiv:
    bl eval_single
    b parse_loop_ismuldiv

parse_loop_isplusminus:             @ similar to above but we just evaluate so long as we don
't reach an open parenthesis due to lowest precedence
    mov r1, #0
    ldrb r0, [r6]
    cmp r0, #'('
        addne r1, #1
    ldr r0, =operators
    cmp r6, r0
        addge r1, #1
```

```
    cmp r1, #2
        beq parse_loop_evalplusminus

    add r6, #1
    add r11, #1
    str r7, [r6]
    add r4, #1
    mov r8, #0
    b parse_loop
parse_loop_evalplusminus:
    bl eval_single
    b parse_loop_isplusminus
```

-> And this is for when an operand is encountered:

```
parse_loop_isoperand:               @ record the operand using sscanf.
    add r5, #8
    mov r0, r4
    ldr r1, =operand_scan_fmt
    mov r2, r5
    ldr r3, =offset
    bl sscanf

    cmp r10,#1
    bleq parse_loop_flip_operand

    mov r8, #1
    ldr r1, =offset
    ldr r1, [r1]
    add r4, r1
    b parse_loop
```

-> Negative numbers are supported. This handles flipping the sign with vneg (negate):

```
parse_loop_flip_operand:            @ flip the sign if indicated by a preceding negative sign
.
    vldr.f64 d0, [r5]
    vneg.f64 d0, d0
    vstr.f64 d0, [r5]
    mov r10, #0
    bx lr
```

```
parse_loop_openparen:               @ record open parenthesis
    cmp r8, #1
    beq error_bad_input_exit
    add r4, #1
    add r6, #1
    mov r1, #'('
    strb r1, [r6]
    mov r8, #0
    b parse_loop

parse_loop_closeparen:
    ldrb r0, [r6]                   @evaluate until we reach the open parenthesis, then disca
rd both
    cmp r0, #'('
    bne parse_loop_evalparen
    sub r6, #1
    add r4, #1                      @skip the close paren in input
    b parse_loop
parse_loop_evalparen:
    sub r1, r0, r6
    cmp r1, #-1
    beq error_bad_parens
    bl eval_single
    b parse_loop_closeparen
```

-> Open parentheses are placed on the operator stack, but no action is taken until a close parenthesis is encountered, at which point the existing expression will be evaluated until the open parenthesis is found. Both are then discarded.

```
parse_loop_operations_exceeded:     @too many ops, time head out
    ldr r0, =operations_exceeded_msg
    bl printf
    ldr r0, =bail
    ldr lr, [r0]
    mov r0, #5
    bx lr

error_bad_parens:                   @ bad parentheseses error message routine
    ldr r0, =error_bad_parens_msg
    bl printf
    ldr r0, =bail
    ldr lr, [r0]
    bx lr
```

-> These routines print errors in case of too many operations or mismatched parentheses.

Breakdown: Finish and Print

```
print_result:                       @ if we have an operand OR an open parenthesis at the en,
, we're good to go
    cmp r8, #1
        bge print_result_eval_remaining
    ldrb r9, [r6]
    cmp r9, #'('
        beq print_result_eval_remaining
    sub r4, r4, #1
    b error_bad_input_exit          @ otherwise there's a mistake -> error out


print_result_eval_remaining:        @ evaluate the remaining operands
    ldr r0, =operators
    sub r0, r6, r0
    cmp r0, #0
    blt print_result_print
    bl eval_single
    b print_result_eval_remaining

print_result_print:                 @ finally print the result
    ldr r0, =print_result_msg
    ldr r1, =operands
    vldr d0, [r1]
    vmov r2, r3, d0
    bl printf
    ldr r0, =bail
    ldr lr, [r0]
    mov r0, #0
    bx lr
```

-> The end of the input is checked for validity (no operator at the end). The rest of the operands and operators are evaluated until none remain, and then the result is printed.

Breakdown: Evaluation

```
eval_single:
    push {r0, r1, r2, lr}
eval_single_cont:
    ldrb r0, [r6]                   @ get top operator
    cmp r0, #'('
        beq error_bad_parens
    vldr.f64 d0, [r5, #-8]
    vldr.f64 d1, [r5]

    cmp r0, #'+'
        bleq eval_add
        vstreq.f64 d3, [r5, #-8]
        beq eval_single_end
    cmp r0, #'-'
        bleq eval_sub
        vstreq.f64 d3, [r5, #-8]
        beq eval_single_end
    cmp r0, #'/'
        bleq eval_div
        vstreq.f64 d3, [r5, #-8]
        beq eval_single_end
    cmp r0, #'*'
        bleq eval_mul
        vstreq.f64 d3, [r5, #-8]
        beq eval_single_end
    cmp r0, #'^'
        bleq pow                    @ !!!! pow doesn't work! seems to be an issue with the ha
rd/soft float calling conventions, don't have hf frontend to test with
        vstreq.f64 d0, [r5, #-8]
        beq eval_single_end
eval_single_end:
    sub r5, #8
    sub r6 ,#1
    pop {r0, r1, r2, lr}
    bx lr

eval_add:   vadd.f64 d3, d0, d1
        bx lr
eval_sub:   vsub.f64 d3, d0, d1
        bx lr
eval_div:   vdiv.f64 d3, d0, d1
        bx lr
eval_mul:   vmul.f64 d3, d0, d1
        bx lr
```

-> Evaluation is done one operation at a time. eval_single takes the top operator on the operator stack, two topmost two operands, and executes one

of five subroutine calls to complete the evaluation. It then decrements the pointers and returns.

-> As discussed, pow() doesn't appear to work properly. This may be because of the lack of a hf frontend/target triple. The arguments to pow are passed in the same way as done in a test file compiled with the same options, and the stack is kept aligned throughout the program.

Challenges Faced and Notes

- **RPN** implementation
  - I found it unintuitive that RPN had to be used as an intermediary for the Shunting Yard algorithm. After some in-depth research and discussion with Jon Lu, I discovered that the algorithm could indeed by easily implemented without a separate output stack and RPN evaluator.
    - All operations are performed on the operand stack, with a small separate stack for operators.
- VFP instructions
  - VFP instructions are more limited in some ways than their normal ARM counterparts and have unique syntax; notably that most of them can't be conditionally executed with condition flags directly.
- printf() and pow() argument passing
  - Finding the specification for the calling conventions for these was accomplished through looking at the assembly output of small test programs in C.
- printf() stack alignment
  - printf() would not function properly unless the stack was 8-byte aligned. The ARM reference website indicates that 8-byte stack alignment is convention on external function calls. http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka4127.html
- pow() not working
  - While GCC assembly output included a calling convention for pow(), it does not function properly. This may be due to incorrect flags, or potentially issues with the toolchain (incorrect target triple; hf needed).

- ldr pseudo-instruction
  - the form ldr, r0, =some_data_item is used frequently for brevity. It is expanded by the compiler, and is described here: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0100a/armasm_babbfdih.htm

Acknowledgements

- Jonathan Lu, for testing assistance and discussion on RPN/Shunting Yard
- cppreference.com
- infocenter.arm.com
- ARMv7 Architecture Reference Manual