Shahreer Al Hossain, Brian Khaimov, Giuseppe Quaratino, Fakharyar Khan
ECE-251: Computer Architecture
Professor Billoo

Project #3

In this project, our group wrote a program that accepts an infix expression as a command-line argument and evaluates the expression. The program supports the basic mathematical operations addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^). It also accepts parentheses to increase the precedence of various operations. The program applies the rules of PEMDAS to evaluate the inputted expression.

Our program is separated into a data section (.data) and a code section (.text). In the (.data) section. The arguments for the various C functions imported from the C standard library are initialized in the data section. A string containing all the required operators and parentheses arranged in order of precedence is initialized in the data section. This will be used later to determine the precedence values of the operators and convert the infix expression into a postfix expression. Another string containing only the required operators arranged in order of precedence is also initialized in the data section. Lastly, the error messages for different scenarios are initialized in the data section.

Figure 1. Data Section

```
.data
.balign 4
operators: .asciz "+-o*/o()o^"

.balign 4
prefix: .space 100

.balign 4
spa: .asciz " "

.balign 4
infix: .space 100

.balign 4
operations: .asciz "+-*/^"

.balign 4
delimitor: .asciz " "

.balign 4
printnum: .asciz "%d\n"

.balign 4
printerr: .asciz "%s\n"

.balign 4
onearg: .asciz "Error: Enter one argument"

.balign 4
divzero: .asciz "Error: Divide by zero"

.balign 4
return: .word 0

.balign 4
result: .space 100

.global main
.global printf
.global strtok
.global atoi
```

C functions from the Standard C Library used in the program:

Printf: writes the C string pointed by format to the standard output (stdout)

Strtok: Scans a string and breaks up the string into a separate string based on the delimiters within
the string

Atoi: Takes string values that are numbers and converts them to ints.

The Shunting Yard Algorithm Implementation:

To evaluate the mathematical expression, Dijkstra's Shunting Yard Algorithm was implemented. The standard way we write mathematical expressions is known as the infix notation. Mathematical operators have precedence, but we can use parentheses to override this precedence. The shunting yard algorithm is a technique for parsing infix expressions containing binary operators of different precedence. Basically, the algorithm assigns to each operator its appropriate operands, and the operators are assigned to the correct operands after the order of precedence into account. Therefore, the algorithm is used to convert the infix notation to postfix notation.

Before the program can evaluate the expression, the code converts the infix notation to postfix notation where the operands are in the correct order. The following behavior is implemented in the code:

- ❏ the expression is parsed left to right.
- ❏ each time a number or operand is read, it is pushed onto the stack
- ❏ each time an operator comes up, the required operands from the stack are popped, and the operation is performed
- ❏ the intermediate result is pushed back to the stack.
- ❏ when there are no tokens (numbers, operators, or any other mathematical symbol) to read, the program stops evaluating, and the final number on the stack is the result

Figure 2. The ‘main’ branch first stores the mathematical expression that is entered by the user as a command line argument into register R0. Since the user needs to enter a command of type ./filename ‘expression’, the second argument is stored into the register R0 by using an offset. If the user doesn’t follow the convention mentioned above, the program will print out an error message.

```
main:

    mov r3, r0          /*store argc in r3*/
    ldr r0, [r1, #4]    /*store link register*/
    ldr r1, =return
    str lr, [r1]
    cmp r3, #2          /*error message if argc != 2*/
    bne err_onearg
    ldr r2, =infix
    ldr r3, =spa
    ldrb r3, [r3]
```

Figure 3. The ‘Parse‘ branch and “operation’ loop are used to parse through the mathematical expression entered by the user. The mathematical expression is scanned in as a string. Every character of the expression is compared to the required operators to see if the expression contains any operators. If a character of the expression is found to be an operator, the character (operator) is stored in the ‘infix’ array.

```
Parse:
    ldrb r7, [r0]
    cmp r7, #0 //check if reached end of input
    beq Converter //if so go to end
    ldr r1, =operations //load the address of  operations string in r1
```

```
Operator:
    ldrb r7, [r1]
    cmp r7, #0 //check if reached end of operation string
    beq Storage //means the current character in output isn't an operator
    ldrb r5, [r1] //else load value of r1 in r5, the current operator
    ldrb r6, [r0] //load current value in input string to r6
    cmp r5, r6 //compare the two values
    beq Op //if equal go to loop3
    add r1, r1, #1 //if not increment address of r1
    b Operator //start next iteration of loop2
```

Figure 4. The 'PrecedenceCheck' loop checks if the operator is placed in the right order of precedence in the infix array. By comparing the operators in the input expression with the characters (operators) in the operators[] string array `"+-o*/o()o^"`, the precedence of the operator is determined. Using multiple loops, the operator is then placed in the right position in the infix array.

```
PrecedenceCheck:
    ldrb r8, [r4]
    cmp r8, #0
    beq Popping
    cmp r8, r7
    beq Popping
    add r6, r6, #1
    add r4, r4, #1
    b PrecedenceCheck
```

Figure 5. The 'Popping' segment compares the precedence numbers of the character on the top of the stack and of the current character in the infix expression. The operators string (see Figure 1) separates operators of equal precedence by the character 'o'. This ensures that if two operators have different precedence numbers, their distance in the operations string will be greater than one. Until it either reaches the end of the stack or it finds a character with a lower precedence than the current character in the infix expression, the OtherOperator loop will continue to pop off the stack and append it to the output string array.

```
Popping:
    cmp r6, #9
    moveq r6, #6
    cmp r8, #0
    moveq r6, #12
    sub r6, r5, r6
    cmp r6, #1
    bgt Pushing
    cmp r9, #0
    beq Pushing
    ldrb r6, [sp], #4
    strb r6, [r2]
    add r2, r2, #1
    sub r1, r1, #1
    b OtherOperator
```

Figure 6. Once the infix expression has been converted to postfix, more spaces are added to the expression to ensure that the operands and operations in the string are separated by spaces. The strtok C function is used to divide the postfix expression, which is a string containing operators and operands, into single operators and operands using the space character ‘ ’ as the delimiter. The ‘operator’ characters in the postfix expression are compared with ascii value of the +, -, *, /, and ^ characters to identify what operator it is. Based on the identity of the operator, the code branches to different branches (‘add’/ ‘subtract’/ ‘multiply’/ ‘divide’/ ‘power’) where the code will perform the operations.

```
evalloop:

   mov r0, #0
   ldr r1, =delimitor
   bl strtok

   mov r1, r0
   cmp r0, #0
   beq answer

   ldrb r0, [r0]        /*checks for '+'*/
   cmp r0, #43
   beq add

   cmp r0, #45          /*checks for '-'*/
   beq subtract

   cmp r0, #42          /*checks for '*'*/
   beq multiply

   cmp r0, #47          /*checks for '/'*/
   beq divide

   cmp r0, #94          /*checks for '^'*/
   beq power

   mov r0, r1
   bl atoi
   push {r0}

   b evalloop
```

Figure 7. If an expression contains the addition, subtraction, or a multiplication sign, it will be moved into its branch accordingly. The two operands will go through the operation given using the basic assembly function for addition, subtraction and multiplication, and the result will be pushed onto the stack.

```
add:

   pop {r2,r3}
   add r1, r2, r3
   push {r1}

   b evalloop

subtract:

   pop {r2,r3}
   sub r1, r3, r2
   push {r1}

   b evalloop

multiply:

   pop {r2,r3}
   mul r1, r2, r3
   push {r1}

   b evalloop
```

Figure 8. If an expression contains the division operation the two numbers being divided will be popped off the stack and run through this loop. The dividend is stored in r3 and the divisor is stored in r2. The divisor is compared to zero to see if the operation is valid. If it's not zero the operation will begin. The divisor will repeatedly be subtracted from the dividend until the dividend reaches zero. The number of times the loop was run is the quotient of the operation and it gets stored onto the stack.

```
divide:

  pop {r2,r3}
  cmp r2, #0
  beq err_divzero
  mov r1, #0

divloop:

  sub r3, r3, r2
  add r1, r1, #1
  cmp r3, #0
  bge divloop
  sub r1, r1, #1
  push {r1}

  b evalloop
```

Figure 9. If an operation has a carrot symbol the operands will be put into the power branch. Here r2 will store the exponent and r3 will store the base number. If the exponent equals zero then the program will branch into “powzero” where the result will just be stored as 1. To evaluate the operation the base will be multiplied with itself and the exponent will be subtracted by 1. This will repeat until the exponent is equal to 1. The result will then be pushed onto the stack.

```
power:

    pop {r2, r3}
    cmp r2, #0
    beq powzero
    mov r4, r3

powloop:

    mul r1, r3, r4
    mov r3, r1
    sub r2, r2, #1
    cmp r2, #1
    bgt powloop
    push {r1}

    b evalloop

powzero:

    mov r1, #1
    push {r1}

    b evalloop
```

Basic Four Mathematics Operations with Floating Point Numbers in ARM Assembly

Figure 10. This is the data section for the ARM Assembly program that was written to perform the basic four math function with floating point numbers. The arguments for the various C functions imported from the C standard library are initialized in the data section.

```
.data
.balign 4
        enterFirst:  .asciz "Enter the first number: "
.balign 4
        enterSecond: .asciz "Enter the second number: "
.balign 4
        inputFormat: .asciz "%f"
.balign 4
        addition:      .asciz "The result of adding is: %f\n"
.balign 4
        subtraction:      .asciz "The result of subtracting is: %f\n"
.balign 4
        multiply:      .asciz "The result of multiplying is: %f\n"
.balign 4
        division:      .asciz "The result of diviving is: %f\n"

.text
.global main
.global printf
.global scanf
```

Figure 11. The floating-point extension , also called the VFP Extension, can be used to work with floating point numbers in ARM assembly code. The VFP also makes use of a Load / Store architecture. It has its own sets of registers apart from the registers provided by the processor itself. There are 32 registers used for single precision storage or 16 registers for double precision storage. The VADD, VSUB, VMUL, VDIV instructions were used to add, subtract, multiply, and divide two floating point numbers. Note that the VMOV instruction requires to ARM registers to move a double precision floating point number from the VFP registers to ARM registers.

```
main:
        PUSH {R0, LR}
        LDR R0, =enterFirst
        BL printf

        SUB SP, SP, #8
        LDR R0, =inputFormat
        MOV R1, SP
        BL scanf
        VLDR S1, [SP]
        VCVT.F64.F32 D1, S1

        LDR R0, =enterSecond
        BL printf

        SUB SP, SP, #8
        LDR R0, =inputFormat
        MOV R1, SP
        BL scanf
        VLDR S2, [SP]
        VCVT.F64.F32 D2, S2

        VADD.F64 D0, D1, D2
        VMOV R2, R3, D0
        LDR R0, =addition
        BL printf

        VSUB.F64 D0, D1, D2
        VMOV R2, R3, D0
        LDR R0, =subtraction
        BL printf

        VMUL.F64 D0, D1, D2
        VMOV R2, R3, D0
        LDR R0, =multiply
        BL printf

        VDIV.F64 D0, D1, D2
        VMOV R2, R3, D0
        LDR R0, =division
        BL printf

        ADD SP, SP, #16
        POP {R0, LR}
        BX LR
```

Challenges Faced

Support for floating point numbers: We were unable to integrate the VFP extension into our code. Our ARM code calculator works great for math expressions with whole numbers. However, while trying to include VFP instructions in the code written for the project, we encountered various types of errors. Additionally, some parts that work fine in ARM assembly code sometimes stop working when VFP instructions are used. When we have a double precision VFP register (which contains 8 bytes) some processes are different from what they are for ARM.

Evaluating postfix expression: While we were testing our code, we found that there was a bug in the postfix converter. When an infix expression such as 3+5*2+3 is given where a high precedence operator is "sandwiched" between two lower precedence operators, the converter didn't add spaces properly and gave an expression similar to 3 5 2 * +3 +. This was solved by iterating over the expression and adding a space between any operation and operand that were together.