CS 1022 ASSIGNMENT #3

WILLIAM BROWNE: 09389822

ASSIGNMENT OUTLINE:

CREATE A PROGRAM THAT EVALUATES A REVERSE POLISH NOTATION EXPRESSION

Explanation of how **Reverse Polish Notation** (RPN) works:

RPN is a mathematical expression where the operations follow the operands (or arguments being worked on). It is referred to as *postfix notation*, unlike our commonly used expression which is known as *infix notation.* To get a better understanding, examples work best.

Infix notation	Postfix notation
2 + 3	2 3 +
(33 + - (4!)) * (-100 + (5^3))	33 4 ! n + 100 n 5 3 ^ + *

Beginning my solution:

Previously we had already made a quite simple RPN calculator. However, it was restricted to values greater than 0, but less than 9, and had only 3 operators – plus, minus and multiplication. So for this assignment we had to expand this program by adding in more operators, and so it would have the capability to work with much greater (and negative!) values. Furthermore, we had to design it as a subroutine. Just like in the assignment outlines, it is noted that RPN can be evaluated conveniently using a stack.

So I had to make a few adjustments to my original program.

I began by first trying to solve a way of pushing values greater than 9 onto the stack. This involved the introduction of a variable (I used R10) which would be set to one after every time it encountered a digit in the string. Say for the value 33 – I first push 3 onto the stack, the variable is set to 1. I move onto the next character which is 3. I check if the variable is 1. If it is, I pop the first value off the stack, multiply it by 10 and then add the 3. Which results as (3 * 10) + 3

Further explanation: I keep pushing until I reach a space character. And every time I reach a digit in memory, I let the variable = 1. When I loop back around and find if the variable is equal to 1, I pop the last value off the stack, multiply it by ten, add the new digit and pop the result back on. If there is still an extra digit, it will repeat the last step. Once it reaches a space character (0x20), the variable will become 0 to note that the value is finished with and I can move onto the next. Every time I reach a space character, I just move onto the next character in memory.

These following operators are the ones in which I had to implement in my new program.

(Previous included were addition, subtraction and multiplication)

! - factorial (unary)

n - negation (unary)

^ - power (binary)

/ - integer division (binary)

Note: Dealing with my operators took the form of if-else parts.

To implement the factorial operator I had to first of all ensure the character was the correct one (!). The method begins by popping the value off of the stack. 1 is then subtracted from this value and the result is stored in a new temp value. This is done to create the first value to be multiplied.

Example – 4 is popped off of the stack. 1 is subtracted from 4, resulting in 3, which is stored in a new register.

```
CMP R0, #'!'
BNE else5
LDR R5, [sp], #4
SUB R6, R5, #1
MUL R0, R5, R6

CMP R6, #1
BLE endwh1
SUB R6, R6, #1
MUL R0, R6, R0
B wh1
```

I then created a while loop which will continually multiply the original value by the new temporary value until the temp value is less than or equal to 1. The temp value is decremented by 1 each time which will result in the following structure

4 * 3 * 2 * 1

To implement the negation operator I had to make use of 2's complement. This is a useful way to change a binary number to its negative. The process involves inverting the bits and adding 1. Once I had reached the negation character (n) in the string I popped the value off of the stack. I inverted the value using the ARM instruction **MVN.** This only inverted the bits, so to complete the negation I had to add 1 bit. I could then push the value back onto the stack.

```
    CMP R0, #'n'
    BNE else6
    LDR R0, [sp], #4
Implementing 2's complement
    MVN R2, R0
    ADD R2, R2, #1
    STR R2, [sp, #-4]!
    B endif
```

To implement the power operator I had to pop twice from the stack. With these two values stored in registers, I made a temp storage each.

I then created a while loop that would multiply the value by itself as many times as it would have to take the temp storage containing the power value to reach 1.

IE – If the sum was 5 ^ 3, once I had created temp values for the both of them I would multiply 5 * 5 in a while loop and decrement the temp power value each time 5 * 5 was computed. When 3 equalled 1 or was less than 1, we could move onto the next character in the string.

```
wh2
    CMP R8, #1
    BLE endwh2
    MUL R2, R9, R6
    SUB R8, R8, #1
    MOV R9, R2
    B wh2
endwh2
    STR R2, [sp, #-4]!
    B endif
```

To implement the division operator I had to pop twice from the stack. These values would act as the divisor and dividend. The first would be the divisor, and the second would be the dividend. I also had to use a register to store the quotient, so I picked an arbitrary register and let it equal 0.

The actual calculation of division consisted of subtracting the divisor from the dividend until the divisor was lower. Every time this was computed, the quotient was incremented and stored.

```
; R7 = quotient
; A = dividend
; B = divisor
wh3
    CMP R5, R6        ; If (a < b)
    BLO endwh3        ; {
    ADD R7, R7, #1    ; quotient = quotient + 1
    SUB R5, R5, R6    ; a = a - b
    B wh3             ;}

endwh3
    STR R7, [sp, #-4]!    ; Push quotient value onto stack
    B endif               ; Move onto next character
```

The operators for addition, subtraction and multiplication were very similar. They all took the form of popping twice from the stack. With these two popped values:

For addition we had to use was the **ADD** operator

For subtraction “ “ **SUB** operator

For multiplication “ “ **MUL** operator

```
else1
    CMP R0, #'+'
    BNE else2
    LDR R5, [sp], #4
    LDR R6, [sp], #4
    ADD R2, R5, R6
    STR R2, [sp, #-4]!
    B endif
```

```
else2
    CMP R0, #'-'
    BNE else3
    LDR R5, [sp], #4
    LDR R6, [sp], #4
    SUB R2, R5, R6
    STR R2, [sp, #-4]!
    B endif
```

```
else3
    CMP R0, #'*'
    BNE else4
    LDR R5, [sp], #4
    LDR R6, [sp], #4
    MUL R2, R5, R6
    STR R2, [sp, #-4]!
    B endif
```

Conclusion: I am very happy with how my program functions. Upon completion, I tested the subroutine with varied RPN expressions in order to be sure it works correctly. Since the expression supplied in memory did not include the division operator, I decided to make simple expressions to ensure it worked. IE (“8 4 /”,0), (“13 6 /”, 0) etc.

RPN Expression	Expected Result	My result
30 400 * 15 60 * +	12900	12900 or 0x00003264
100 2 + 30 *	3060	3060 or 0x00000bf4

My program is very responsive and I haven’t encountered any problems in any calculations upon its final draft.