ARM Simple Calculator

Luke Hackett - 17324340

CS1021 Intro to Computing

Contents

1. Introduction

2. Functionality
- 2.1. Addition
- 2.2. Subtraction
- 2.3. Multiplication
- 2.4. Division
- 2.5. Powers
- 2.6. Factorial
- 2.7. Alter input

3. Read
- 3.1. Implementation
- 3.2. Code and testing

4. Evaluate
- 4.1. Implementation
- 4.2. Testing
- 4.3. Code and testing

5. Print
- 5.1. Implementation
- 5.2. Code and testing

6. Extra Mile
- 6.1. Division
  - *6.1.1. Implementation*
  - *6.1.2. Code and testing*
- 6.2. Deleting digits
  - *6.2.1. Implementation*
  - *6.2.2. Code and testing*
- 6.3. Powers
  - *6.3.1. Implementation*
  - *6.3.2. Code and testing*
- 6.4. Formatting result
  - *6.4.1. No leading zeroes*
  - *6.4.2. Added whitespace and equal*
  - *6.4.3. Remainder printed and Goodbye when exit*

6.5. Unlimited amount of calculations

7. Broken Stuff

7.1. Negative numbers

7.2. Very large numbers

1. Introduction

The simple calculator that I have built will take an input from the user, perform any necessary computations and print out the answer correctly formatted with whitespace, no leading zeros and an equals sign. The calculator will then continue to take calculations on a new line each time until the user chooses to exit the program. At this point the calculator prints out “Goodbye!” and the program will terminate. I chose to add more possible operations and format the answer in the most simple and user friendly way . Therefore my focus was not on dealing with negatives and I chose not to add more operators.

2. Functionality

2.1 Addition:
- Takes ‘+’ as addition character
- Will add two numbers entered

2.2 Subtraction:
- Takes ‘-’ as subtraction character
- Will subtract one number from another

2.3 Multiplication:
- Takes ‘*’ as multiplication character
- Will multiply two numbers entered

2.4 Division:
- Takes ‘/’ as division character
- Will divide one number by another, and print quotient and any remainder

2.5 Power:
- Takes ‘^’ as power character
- Will calculate the first number input to the power of the second

2.6 Factorial:
- Takes ‘!’ as factorial character
- Will calculate the factorial of a number entered

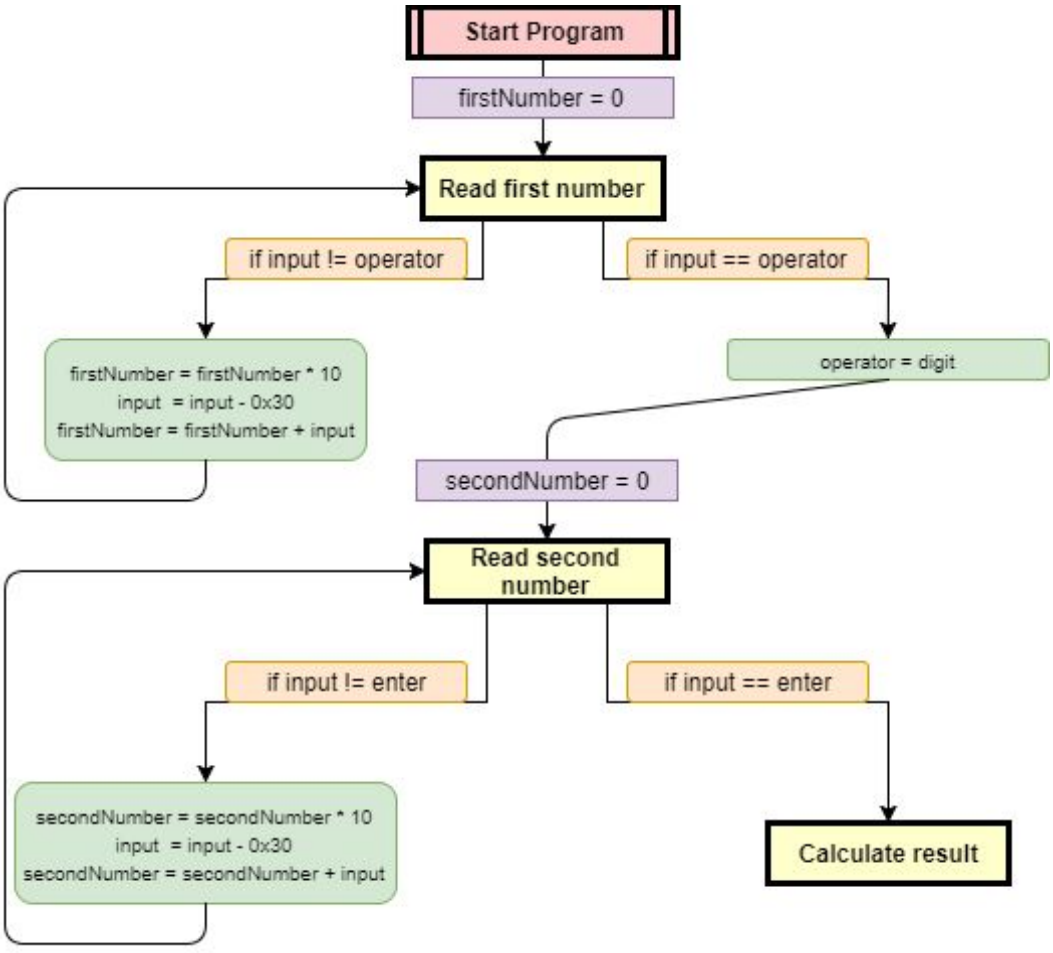
2.7 Alter Input:
- Backspace will remove the last digit
- This removes the digit from the number and allows the user to alter input

3. Stage 1 - Read

3.1 How it works

The first read loop works by starting with 0 as the ‘first number’. Each time a number is entered the ‘first number’ register is multiplied by ten. This is because when a new digit is added to a number, the number is made bigger by a power of ten. The input is then converted from its ASCII value to its numerical value . This is done by subtracting 0x30 from the number, as this is where ASCII numbers start(e.g 0x39 - 0x30 = 0x9). We can then simply add this number to our ‘first number’ register to create the updated number.

This first read loop will continue until the character entered is an operator. At this point the program will identify the operator and assign a variable to signify what this operator is. Once the operator is assigned, the program will move to the second read loop, which will read the input in the same manner as the first loop, storing its running total in a ‘second number’ register. This loop will only terminate when the user presses the enter key, then the loop will end and any necessary calculations will begin.



The order and logic of the read loop

3.2 Code and testing

In testing both extremes of small and very large numbers were entered into the console. The values for all numbers were correctly stored and printed out to the console. The operators were also printed correctly and stored in their register.

Input	Displayed input	Stored Value 1	Stored Value 2
0 + 12	0 + 12	0x00000000	0x0000000C
321 - 34	321 - 34	0x00000141	0x00000022
12!	12!	0x0000000C	0x00000000
89^7	89^7	0x00000059	0x00000007
4294967295/6	4294967295/6	0xFFFFFFFF	0x00000006

Read loop Code:

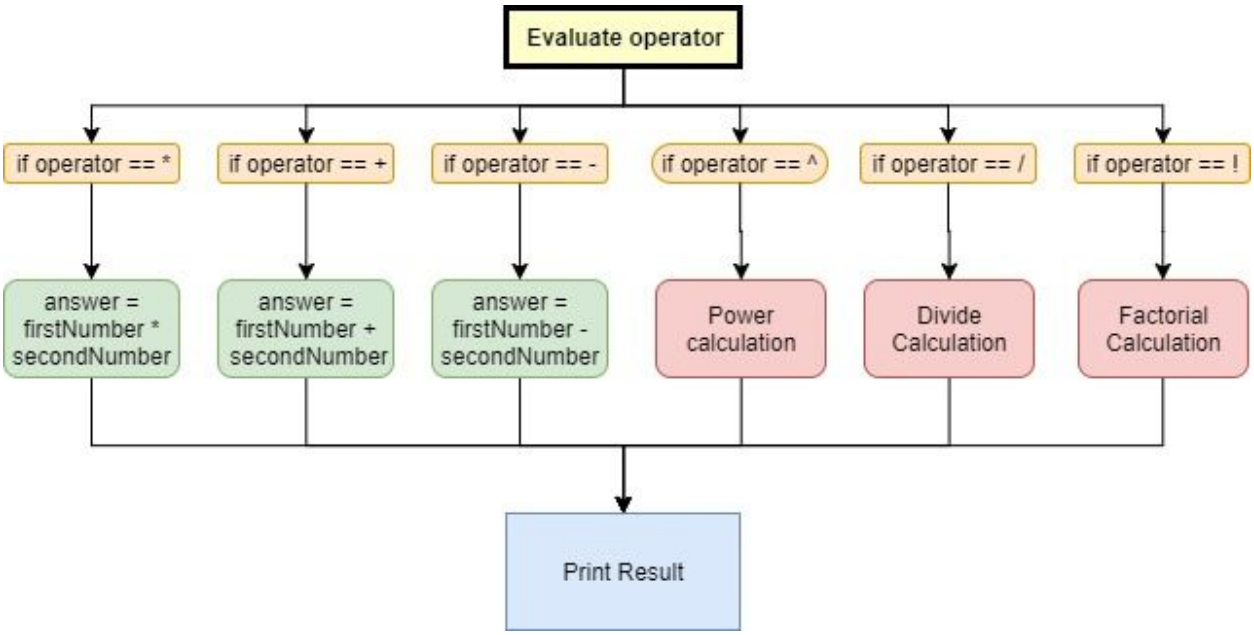
```
read
    BL  getkey      ; read key from console
    CMP R0, #0x1B   ;
    BEQ endProgram  ;
    BL  sendchar    ;   echo key back to console

    ;
    ; do any necessary processing of the key
    ;

    CMP R0, #'*'            ;if(input == '*')
    BEQ multiplyOperator    ;   multiplyOperator()
    CMP R0, #'+'            ;else if(input == '+')
    BEQ addOperator         ;   addOperator()
    CMP R0, #'-'            ;else if(input == '-')
    BEQ subtractOperator    ;   subtractOperator()
    CMP R0, #'/'            ;else if(input == '/')
    BEQ divideOperator      ;   divideOperator()
    CMP R0, #'^'            ;else if(input == '/')
    BEQ powerOperator       ;   powerOperator()
    CMP R0, #'!'            ;else if(input == '!')
    BEQ factorialOperator   ;   factorialOperator()

    MUL R4, R3, R4      ; number1 *= 10
    SUB R0, R0, #0x30   ; input -= ASCII Offset
    MOV R5, R0          ; lastDigit = input
    ADD R4, R0, R4      ; number1 += input

    B read  ; read next digit

multiplyOperator    ; int multiplyOperator()
    LDR R7,= 1      ;   operator = 1;
    B endRead

addOperator         ; int addOperator()
    LDR R7,= 2      ;   operator = 2
    B endRead

subtractOperator    ; subtractOperator()
    LDR R7,= 3      ;   operator = 3
    B endRead

divideOperator      ; divideOperator()
    LDR R7,= 4      ;   operator = 4
    B endRead

powerOperator       ; divideOperator()
    LDR R7,= 5      ;   operator = 4
    B endRead

factorialOperator     ; factorialOperator()
    LDR R7,= 6      ;   operator = 5
    B endReadAgain

endRead
```

```
readAgain

    BL  getkey      ; read key from console
    CMP R0, #0x0D       ; while (key != enter)
    BEQ endReadAgain    ; {
    BL  sendchar    ; echo key back to console

    ;
    ; do any necessary processing of the key
    ;

    MUL R6, R3, R6      ; number2 *= 10
    SUB R0, R0, #0x30   ; input -= ASCII offset
    MOV R5, R0          ; lastDigit = input
    ADD R6, R0, R6      ; number2 += input

    B readAgain

endReadAgain
```

4. Stage 2 - Evaluate

4.1 Implementation

In the evaluation stage of the program the operator is checked and the program branches to the appropriate section of code. A register is allocated as the result and the outcome of any calculation is stored there. The possible operations for this calculator have been outlined in the functionality section. For addition, subtraction and multiplication the standard commands are used, however as there are no commands for division, powers or factorials these have been implemented within the evaluate section but will not be discussed much further until the **Extra mile** section.



Visualisation of Evaluator logic

4.2 Code and Testing

Once again for testing a variation of large and small numbers were used, as well as some operations that would evaluate to zero.

Input	Expected Result	Stored Result
0 + 12	0x0000000C	0x0000000C
321 - 321	0x00000000	0x00000000
12!	0x1C8F66C0	0x1C8F66C0
5^7	0x1312D	0x1312D
5342234/7	0x000BA910 remainder 0x00000002	0x000BA910 remainder 0x00000002

This is the code used to determine the operator and evaluate appropriately, the division, power and factorial code will be demonstrated in later sections.

Evaluate code:

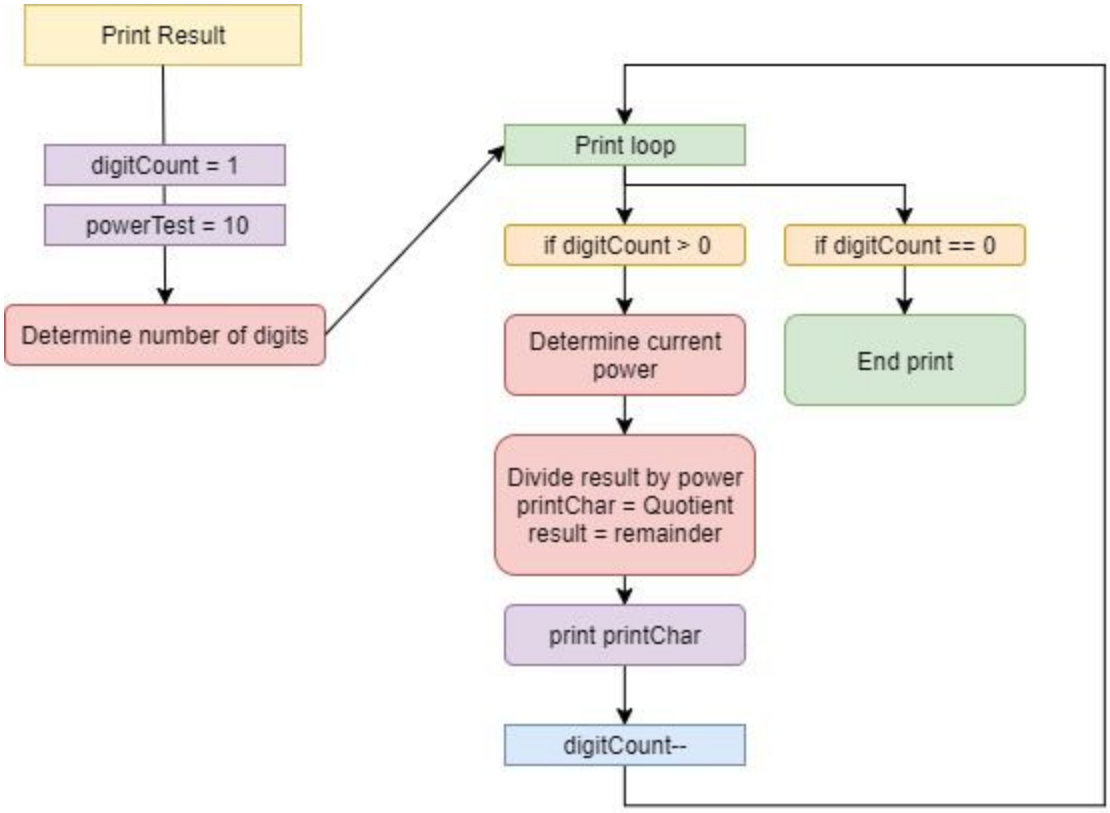
```
    CMP R7,#1        ;if(operator == 1)
    BEQ multiplyExp  ;   multiplyExp()
    CMP R7,#2        ;else if(operator == 2)
    BEQ addExp       ;   addExp()
    CMP R7,#3        ;else if(operator == 3)
    BEQ subtractExp  ;   subtractExp()
    CMP R7,#4        ;else if(operator == 4)
    BEQ divideExp    ;   divideExp()
    CMP R7,#5        ;else if(operator == 5)
    BEQ powerExp     ;   powerExp()
    CMP R7,#6        ;else if(operator == 6)
    BEQ factorialExp;   factorialExp()

multiplyExp
    MUL R5, R4, R6  ; result = number1 * number2
    B endCalculate

addExp
    ADD R5, R4, R6  ; result = number1 + number2
    B endCalculate

subtractExp
    SUB R5, R4, R6  ; result = number1 - number2
    B endCalculate
```

5. Stage 3 - Display

5.1 Implementation

This is the most complex section of code so far in the program. To display the result in the console, it was necessary to first find out the number of digits in the result. This was achieved by comparing the number to increasing powers of 10, starting with ten. If this power of ten was less than the result, this meant that the result had more digits than the current power, so 1 was added to a count that started at 1 each time that this occurred. When the power of ten was finally bigger than the result, the loop was finished and the program moved on.

The next step is to step into a loop that runs as many times as there are digits in the result. This loop will get the current result and determine its power of ten. It will then divide the result by that power and display the quotient, while setting the remainder to the result. It does this because when dividing a number by its power of ten the quotient will be the first digit in the number, and any remainder will be the digits after the first digit. Once this loop repeats for the amount of digits in the number it will have printed out each digit in the number.



Flowchart of logic for print loop

4.2 Code and testing

Once again for testing a variation of large and small numbers were used, as well as some operations that would evaluate to zero.

Input	**Expected printed result**	**Actual printed result**
0 + 12	12	12
321 - 321	0	0
12!	479160000	479160000
5^7	78125	78125
5342234/7	763176 remainder 2	763176 remainder 2

This print code is limited as its is adapted in the case of division to account for printing the remainder as well as the quotient. This will be demonstrated as part of the next section.

Print function code:

```
digits                  ;
    CMP R5, R8          ;
    BLO endDigits       ;    while(result > testPower)
                        ;    {
    MUL R8, R3, R8      ;       testPower *= 10
    ADD R12, R12, #1    ;       numberOfDigits += 1
    B digits            ;    }
endDigits

print
    MOV R8, #1          ; testPower = 1
    MOV R9, #1          ; realPower = 1

power
    CMP R5, R8          ; if(result <= testPower)
    BLE endPower        ;    end division
                        ; else
    MOV R9, R8          ;    realPower = testPower
    MUL R8, R3, R8      ;    testPower *= 10
    B power             ;
endPower

    LDR R11, =0 ; quotient

divide
    CMP R5, R9          ; while(remainder >= power)
    BLO endDivide       ; {
    SUB R5, R5, R9      ;    result = result - realPower
    ADD R11, R11, #1    ;    quotient++
    B divide            ; }
endDivide

    CMP R12, #0         ; if(numberOfDigits == 0)
    BEQ endPrint        ;    end print
                        ; else
    ADD R0, R10, R11    ;    character = ASCII offset + quotient
    BL sendchar         ;    print character
    SUB R12, R12, #1    ;    numberOfDigits--
    B print             ;

endPrint
```

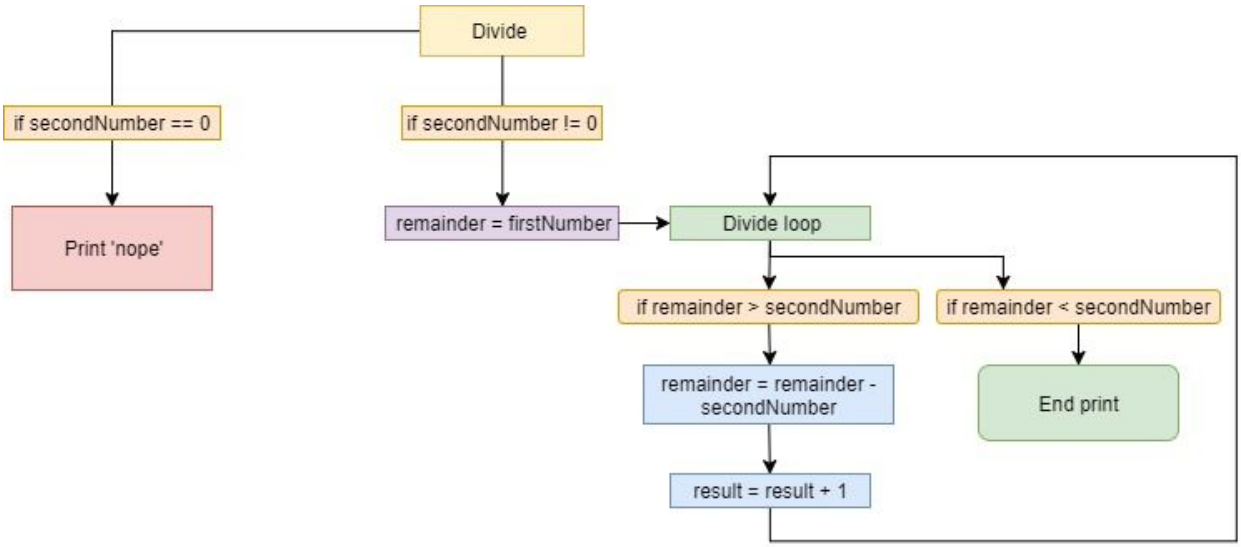
6. Extra Mile

6.1 Division

6.1.1 Implementation

To implement division, it was necessary to allocate another variable to store the remainder, then set that remainder equal to the number to divide. The division was done by making a loop that repeatedly subtracted the remainder by the number that the user wants to divide by, then adding one to the quotient. This loop will end when the remainder is less than the number that it is being divided into. The program also checks if the number to divide by is 0, and if so it will print an appropriate response.

Next it was necessary to print out the quotient and the remainder. As the quotient is stored in the same register as the result so it is not necessary to alter the print code until it is finished printing the first number. At this point the program checks to see if the operator is a '/', and if so it will set the result equal to the remainder, print out ' Remainder ' and loop back to the top of the print loop to print the quotient. The value of the operator is also changed so that the program will not branch back into the code that checks for division more than once.



Divide loop

6.2.2 Code and Testing

In testing i tested values where the second number was bigger than the first, where the second number was zero as well as normal divisions with and without remainders.

Input	Expected printed result	Actual printed result
0 /12	0	0
321/43	7 Remainder 20	7 Remainder 20
123/0	aghh!	nope
12/4	3	3
7/14	0 Remainder 7	0 Remainder 7

Code for the division:

```
divideExp
    LDR R2, =0x0     ; remainder = 0

    MOV R2, R4       ; remainder = number1

    CMP R6, #0
    BEQ nope

subDivide            ;
    CMP R2, R6       ; while(remainder >= number2)
    BLO endCalculate; {
    SUB R2, R2, R6   ;    remainder -= number2
    ADD R5, R5, #1   ;    result += 1
    B subDivide      ; }
```

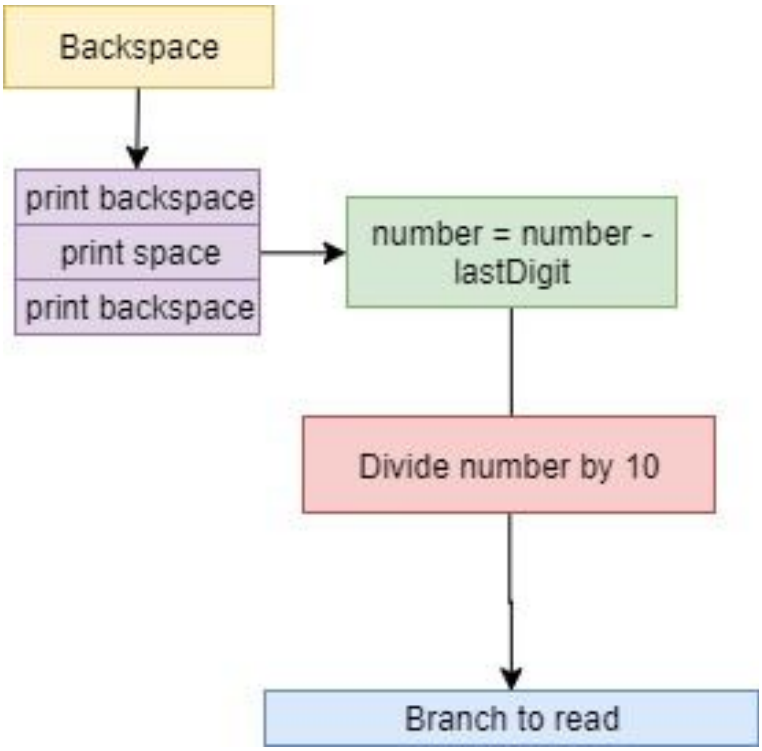
Extra code for printing:

```
    CMP R7, #4
    BNE notDiv           ; if(operator == /)

    LDR R7, =0           ; operator - 0

    CMP R2, #0
    BEQ notDiv

    LDR R0, =' '
    BL sendchar
    LDR R0, ='R'
    BL sendchar
    LDR R0, ='e'
    BL sendchar
    LDR R0, ='m'
    BL sendchar
    LDR R0, ='a'
    BL sendchar
    LDR R0, ='i'
    BL sendchar
    LDR R0, ='n'
    BL sendchar
    LDR R0, ='d'
    BL sendchar
    LDR R0, ='e'
    BL sendchar
    LDR R0, ='r'
    BL sendchar
    LDR R0, =' '
    BL sendchar
    MOV R5, R2
    B remainderPrint

nope
    LDR R0, ='n'         ; print 'n'
    BL sendchar
    LDR R0, ='o'         ; print 'o'
    BL sendchar
    LDR R0, ='p'         ; print 'p'
    BL sendchar
    LDR R0, ='e'         ; print 'e'
    BL sendchar

notDiv
    LDR R0, =0xA         ;
    BL sendchar          ; print new line

    B startProgram       ; } while (input != escape key)
```

6.2 Deleting Digits

6.3.1 Implementation

This code was added on to the end of both read functions to deal with the alteration of a number. If a backspace is entered, the program will print the backspace, then a space to clear the last digit entered then another backspace to move the cursor back to the right position. The program at this point also subtracts the last digit entered from the total number and divides the number by ten to negate the last digit entered. It then branched back to read again.



Backspace logic

6.3.2 Code and Testing

For this the only testing was to verify that if a backspace was pressed then the number would not be affected by the number that was erased. For all cases the result was unaltered by the removed digit.

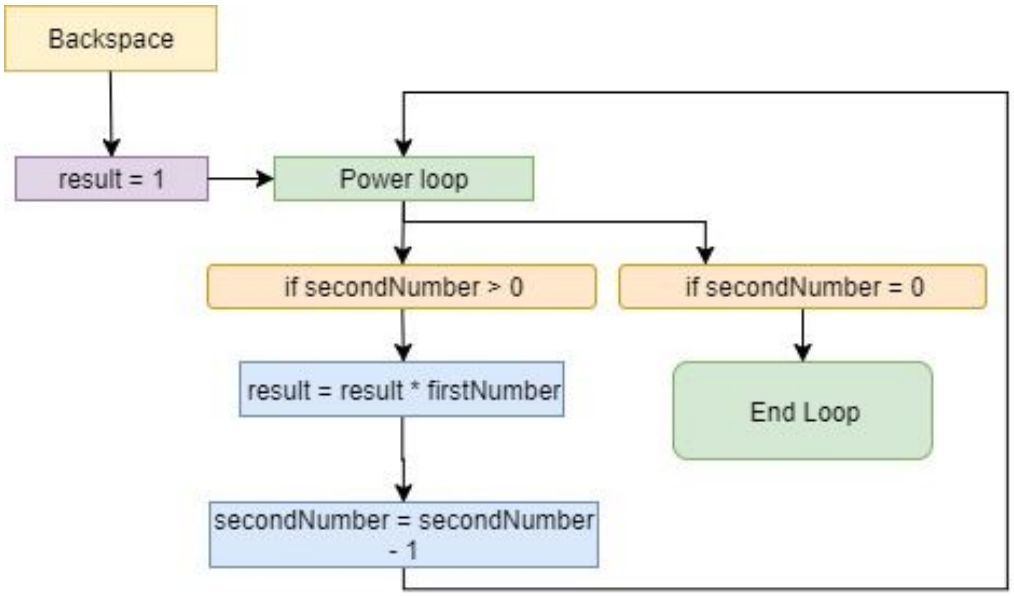
Code for backspace check:

```
backspace
    BL sendchar      ; print backspace
    LDR R0, =0x20    ;
    BL sendchar      ; print space
    LDR R0, =0x8     ;
    BL sendchar      ; print backspace
    SUB R4, R4, R5   ; number2 -= lastDigit

    MOV R5, R4       ; R5 is now equal to the number to divide

    LDR R4, =0       ; number1 = quotient

removePower
    CMP R5, #10          ; while(remainder >= power)
    BLO read             ; {
    SUB R5, R5, #10      ;     remainder = remainder - 10
    ADD R4, R4, #1       ;     number1++
    B removePower        ; }
```

6.3 Powers

6.4.1 Implementation

This was another fairly simple implementation that will compute the result by multiplying the first number input by the result which starts at 1. It will do this in a loop subtracting 1 from the second number input as it goes until the second number is equal to zero, at which point it is done with the calculation.



6.4.2 Code and Testing

To test this I used various small values of power and first number , as well as setting each of the numbers equal to zero.

Input	Expected printed result	Actual printed result
0^2	0	0
123^0	1	1
12^4	27360	27360

Power code implemented:

```
powerExp
    MOV R5, #1      ; result = 1

calcPower
    CMP R6, #0
    BEQ endCalculate; while(number2 != 0) {
    MUL R5, R4, R5  ;     result = result * number1
    SUB R6, R6, #1  ;     number2 -= 1
    B calcPower     ; }
```

6.4 Formatting results

6.4.1 No leading zeroes

Due to my implementation of the print function where I checked the amount of digits that I needed to print there were no leading zeroes in my printed result that I needed to deal with.

6.4.2 Printing whitespace and '='

To present the result in a way that is easily readable for users I printed a space after their input, followed by an equals, then another space before the printed speed

Code:

```
LDR R0, =' '
BL sendchar
LDR R0, ='='
BL sendchar
LDR R0, =' '
BL sendchar
```

6.4.3 Remainder printed and Goodbye when exit

In the case of a division, a remainder will sometimes be present. In the case that a remainder is present the program will print ' Remainder ' followed by the second number. Similarly when the user exits the program by pressing esc, the program will print a Goodbye message

Code for remainder:

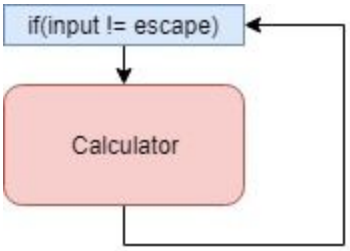
```
LDR R0, =' '
BL sendchar
LDR R0, ='R'
BL sendchar
LDR R0, ='e'
BL sendchar
LDR R0, ='m'
BL sendchar
LDR R0, ='a'
BL sendchar
LDR R0, ='i'
BL sendchar
LDR R0, ='n'
BL sendchar
LDR R0, ='d'
BL sendchar
LDR R0, ='e'
BL sendchar
LDR R0, ='r'
BL sendchar
LDR R0, =' '
BL sendchar
MOV R5, R2
B remainderPrint
```

Code for Goodbye:

```
endProgram

    LDR R0, ='G'
    BL sendchar
    LDR R0, ='o'
    BL sendchar
    LDR R0, ='o'
    BL sendchar
    LDR R0, ='d'
    BL sendchar
    LDR R0, ='b'
    BL sendchar
    LDR R0, ='y'
    BL sendchar
    LDR R0, ='e'
    BL sendchar
    LDR R0, ='!'
    BL sendchar

stop    B   stop

    END
```

6.5 Unlimited amount of calculations

The program will continue to run, and the user will be able to enter a new calculation on a new line until they hit escape, at which point the program will terminate and ‘Goodbye!’ will display. This is done by adding a start label at the top of the program and at the end the program simply prints a new line and branches back to the top. In the read loop there is a check for the escape key being pressed, and if it is it will branch straight to the end.



Logic flow

```
LDR R0, =0xA        ;
BL sendchar         ; print new line

B startProgram      ; } while (input != escape key)
```

Code block

7 Broken Stuff

7.1 Negative Numbers

Since the program does not make use of signed operations there is no functionality to deal with negative numbers. Any attempt to have a negative answer will result in the program hanging. There is no output in this case and no further calculations can be made

7.2 Very Large Numbers

The program uses registers to hold values. The max value of these registers is 0xFFFFFFFF or 4294967295. Any answer that exceeds that number will cause the program to crash and prevent any further input.