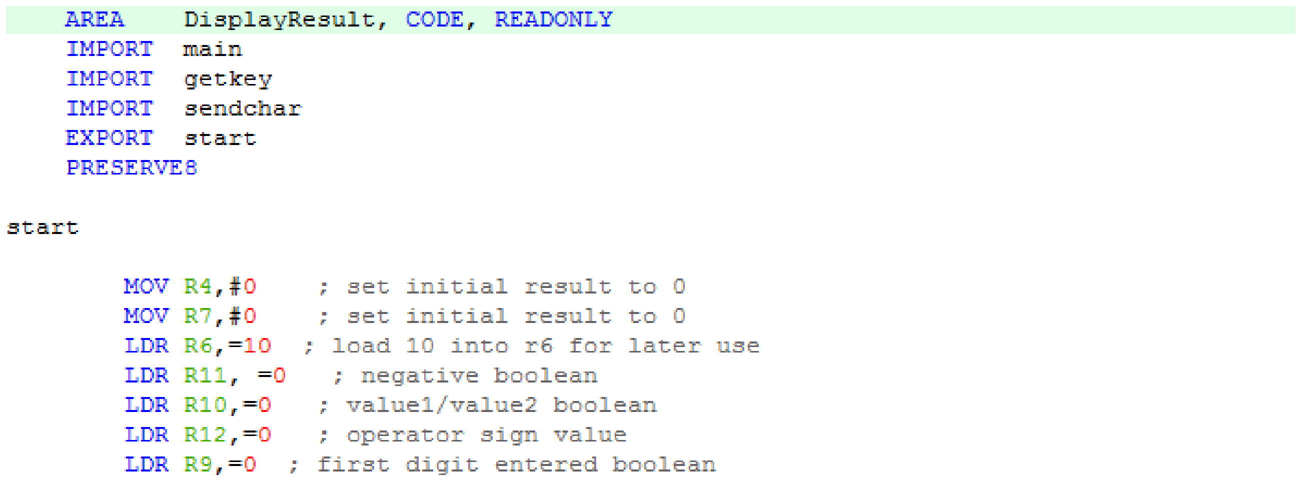
CS1021 Assignment #1 Calculator program is ARM assembly

Student : Anton Yamkovoy.

Console Input : Step #1

The Arm Assembly code will be posted in the next pages, before an in-depth description of its flow and the operations that it carries out, It should be easier to follow the explanation with the code on the screen.



```
	AREA	DisplayResult, CODE, READONLY
	IMPORT	main
	IMPORT	getkey
	IMPORT	sendchar
	EXPORT	start
	PRESERVE8

start

		MOV R4,#0	; set initial result to 0
		MOV R7,#0	; set initial result to 0
		LDR R6,=10	; load 10 into r6 for later use
		LDR R11, =0	 ; negative boolean
		LDR R10,=0	; value1/value2 boolean
		LDR R12,=0	; operator sign value
		LDR R9,=0  ; first digit entered boolean
```

Firstly the Boolean variables and different registers that act like flags are set, and some registers are set to zero so that the operations do not show any unexpected values, the registers from R1-R3, aren't used at all due to the send char function changing their contents in its operation, and the registers R13-R15 aren't touched as usual.

The R4 register will be used throughout the program to store the value of firstly the 1st value of an operation and if there are more than 2 operands the R4 register will store the result of all the previous operations together as a number. It is set to 0.

The R7 register is used for storing the second value inputted, and when there are more than 2 operands it stores the last value to be added at this point.

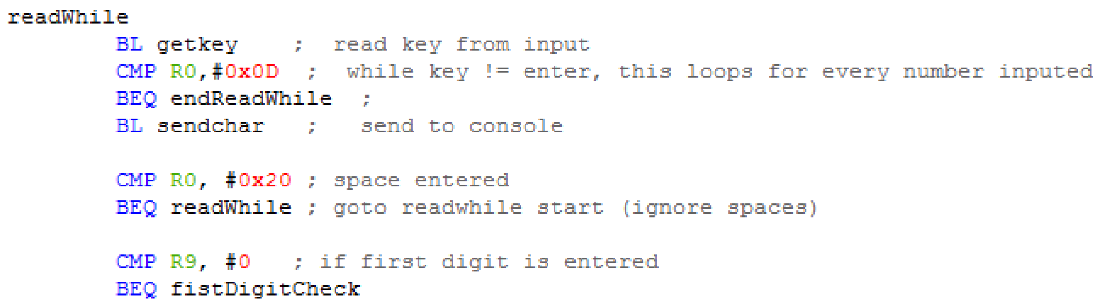
The R6 register is storing a value of 10, which is used in the algorithm that converts the inputted digits into the console 1-by-1 into a value that is stored in the register R4 at first.

The R11 register stores a value of either 1 or 0 and is used to check if a value is negative when converting it to 2s complement form.

The R10 register acts as a 1st / 2nd value Boolean, it is used to check if the 1st and or second values have been inputted and whether to the program can continue with its operation or whether it will go out of the cycle and head onto the calculation steps.

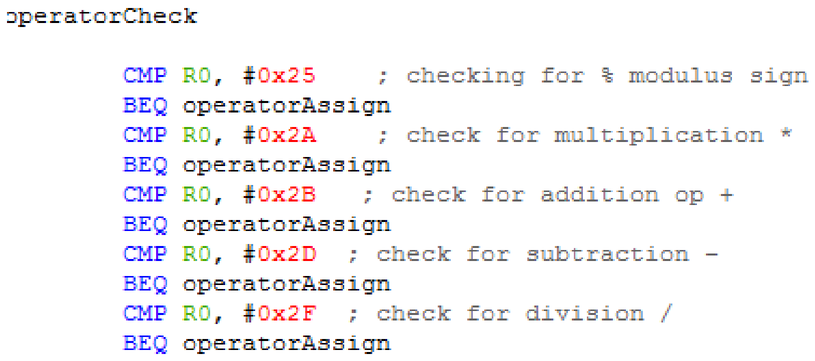
The R12 register holds the ASCii codes for the different operators so they are stored somewhere and can be checked later during calculation stages.

Register R9 is used to check if the first digit of a value has been entered to progress past steps in the loop eg checking for negative numbers.



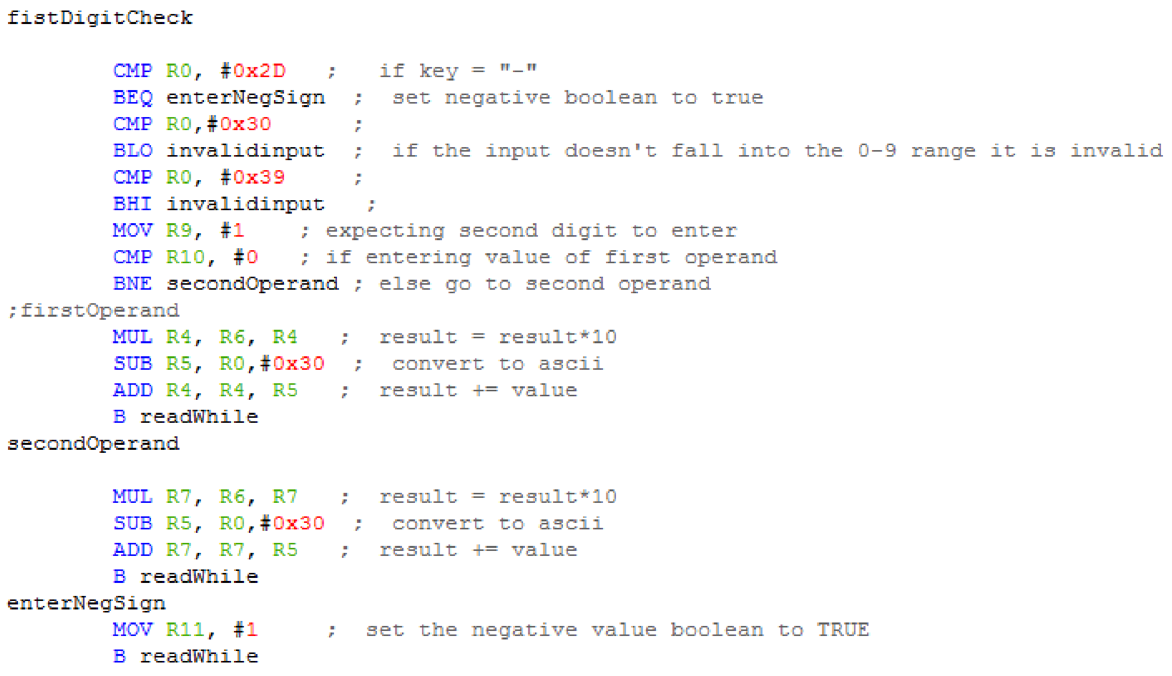
```
readWhile
        BL getkey    ;  read key from input
        CMP R0,#0x0D  ;  while key != enter, this loops for every number inputed
        BEQ endReadWhile  ;
        BL sendchar   ;   send to console

        CMP R0, #0x20 ; space entered
        BEQ readWhile ; goto readwhile start (ignore spaces)

        CMP R9, #0   ; if first digit is entered
        BEQ fistDigitCheck
```

- The first loop that is initiated is the main reading loop of the console input section, this loop starts at the beginning of the program and ends when all operands, operators and signs are already typed into the console, the end of the readWhile loop is triggered by pressing the enter key.
- The next block after the initial reading of the inputted key using BL sendchar, is comparing the Ro register where input is stored with a space character in hex, a space is inputted, the program loops back to readWhile starting all over.
- The R9 register Boolean is checked to see if the first digit has been entered.



```
operatorCheck

        CMP R0, #0x25    ; checking for % modulus sign
        BEQ operatorAssign
        CMP R0, #0x2A    ; check for multiplication *
        BEQ operatorAssign
        CMP R0, #0x2B   ; check for addition op +
        BEQ operatorAssign
        CMP R0, #0x2D  ; check for subtraction -
        BEQ operatorAssign
        CMP R0, #0x2F  ; check for division /
        BEQ operatorAssign
```

- This section checks the Ro, register after the first digit has been inputted, for a sign ascii code value, if the inputte char is not one of these values the program heads on otherwise it goes to the operatorAssign section, which will be mentioned soon. This is not the linear movement in the program, we can assume this section is skipped the first time because, the R9 first digit bool is not set, the program skips this part on the first run of the loop.

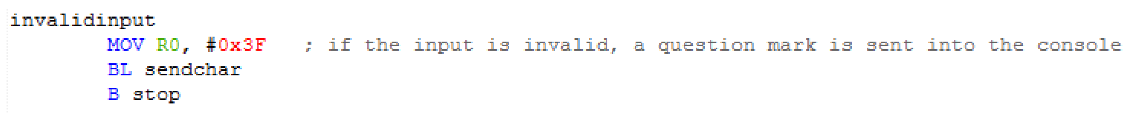


```
fistDigitCheck

        CMP R0, #0x2D   ;   if key = "-"
        BEQ enterNegSign  ;  set negative boolean to true
        CMP R0,#0x30      ;
        BLO invalidinput  ;  if the input doesn't fall into the 0-9 range it is invalid
        CMP R0, #0x39     ;
        BHI invalidinput   ;
        MOV R9, #1     ; expecting second digit to enter
        CMP R10, #0    ; if entering value of first operand
        BNE secondOperand ; else go to second operand
;firstOperand
        MUL R4, R6, R4    ;  result = result*10
        SUB R5, R0,#0x30  ;  convert to ascii
        ADD R4, R4, R5    ;  result += value
        B readWhile
secondOperand

        MUL R7, R6, R7    ;  result = result*10
        SUB R5, R0,#0x30  ;  convert to ascii
        ADD R7, R7, R5    ;  result += value
        B readWhile
enterNegSign
        MOV R11, #1     ;  set the negative value boolean to TRUE
        B readWhile
```

Assuming we skip the operatorCheck, the program arrives at this block, this is the basic operation of the consoleInput stage,.

Firstly there is a check for a negative sign which leads to its own branch. If there is a negative sign the negative Boolean R11 is set to true.

The next checks check if the number falls into the 0-9 range by comparing the input ascii char to first 0x30 and then 0x39 and 0x30 is taken away giving the ascii value to be checked, if it is not a number the program takes you to the invalid input section at the very end where it prints out a question mark in the console.



```
invalidinput
        MOV R0, #0x3F   ; if the input is invalid, a question mark is sent into the console
        BL sendchar
        B stop
```

The question mark code is moved to the R0 sendchar register and it is printed to the screen.

When we have checked the validity of the input, The R9 digit flag it set to 0 expecting a second digit to be inputted, and the R10 flag for the 1st /2nd value is set to 1st value

The program continues and performs the algorithm which when looped converts the inputted digits one by one into a value in the R4 register.

The result which is initially 0 is multiplied by 10, 0x30 is added to the value to convert it to ascii value form, then the result is added to 0, this process continues with each increasing magnitude eg the numbers 123 inputted one by one by keyboard, would transform into a value like this

0 = 0 * 10 1*10 = 10

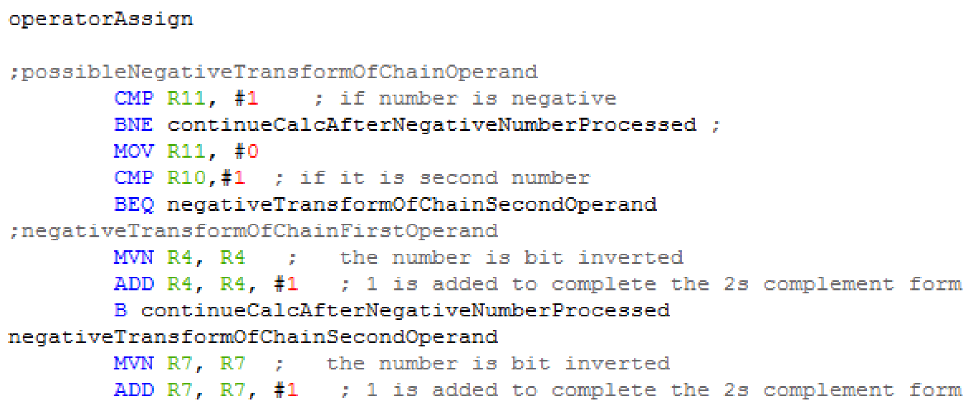
0x31 – 0x30 = ‘1’ 0x32 – 0x30 = ‘2’

Result = 0+1 now the result result is 1 nextLoop ➔ Result = 10+2, now the result is 12

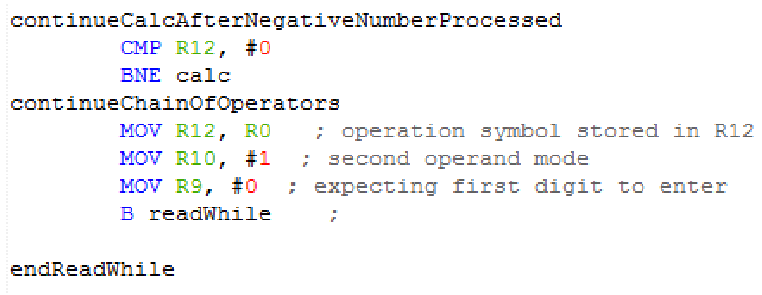
nextLoop ➔

12 * 10 = 120 => 0x33 – 0x30 = ‘3’ => Result = 120+3 = 123

In the code at the start if the “-“ sign has been pressed the R11 boolean is set 1 and the program is send back to the start of readWhile.



```
operatorAssign

;possibleNegativeTransformOfChainOperand
        CMP R11, #1    ; if number is negative
        BNE continueCalcAfterNegativeNumberProcessed ;
        MOV R11, #0
        CMP R10,#1   ; if it is second number
        BEQ negativeTransformOfChainSecondOperand
;negativeTransformOfChainFirstOperand
        MVN R4, R4    ;   the number is bit inverted
        ADD R4, R4, #1   ; 1 is added to complete the 2s complement form
        B continueCalcAfterNegativeNumberProcessed
negativeTransformOfChainSecondOperand
        MVN R7, R7   ;   the number is bit inverted
        ADD R7, R7, #1   ; 1 is added to complete the 2s complement form
```



```
continueCalcAfterNegativeNumberProcessed
        CMP R12, #0
        BNE calc
continueChainOfOperators
        MOV R12, R0    ; operation symbol stored in R12
        MOV R10, #1   ; second operand mode
        MOV R9, #0   ; expecting first digit to enter
        B readWhile    ;

endReadWhile
```

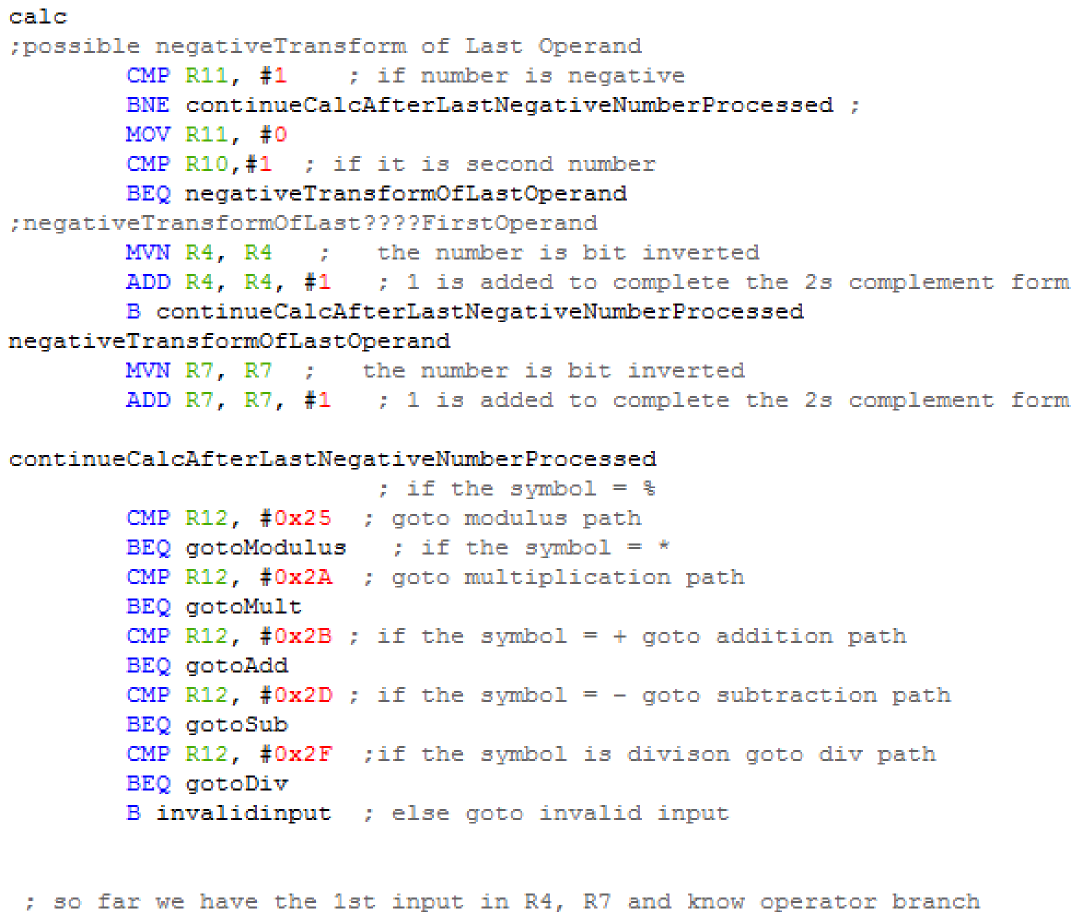
The operatorCheck block skips to this block of code, it holds an operator ascii sign in it’s R0 register.

Firstly the negative flag is checked if the number is negative it jumps to the continueCalcAfterNegativeNumberProcessed tag, after this is done for the value the negative flag is set to false, to prepare for another run of the loop.

Whether the number is the 2nd value is checked, if it is the program jumps to the negativeTransform section for the second operand assuming the first operand is one value / or a chain of previous values

The 2 negative number transform blocks work by inverting the number using the MVN operation and then adding 1 to convert it to 2s complement form.

These blocks of transformations are only applicable to the chain values eg everything except the last value, the continueCalcAfterNegativeNumberProcessed tag checks if the R12 register is not equal to 0 ie there is a symbol in there, then we can procede to calc, else we stay in the continueChainOperators branch, this places the symbol in R12, changes the readWhile to second operand mode as the operator has been inputted and sets R9 to 0 expecting the first digit now in the next run of the loop



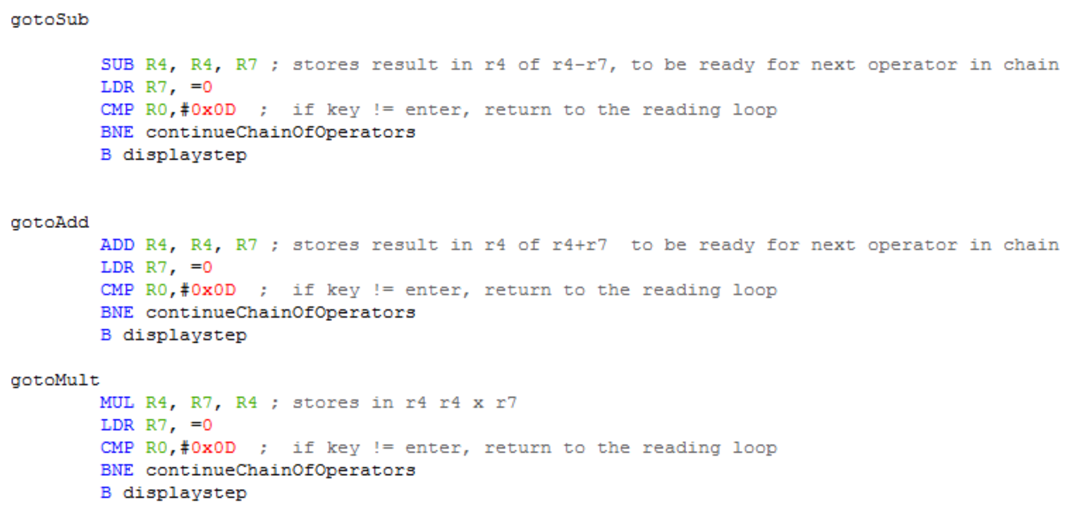
```
calc
;possible negativeTransform of Last Operand
        CMP R11, #1     ; if number is negative
        BNE continueCalcAfterLastNegativeNumberProcessed ;
        MOV R11, #0
        CMP R10,#1  ; if it is second number
        BEQ negativeTransformOfLastOperand
;negativeTransformOfLast????FirstOperand
        MVN R4, R4    ;    the number is bit inverted
        ADD R4, R4, #1    ; 1 is added to complete the 2s complement form
        B continueCalcAfterLastNegativeNumberProcessed
negativeTransformOfLastOperand
        MVN R7, R7  ;    the number is bit inverted
        ADD R7, R7, #1    ; 1 is added to complete the 2s complement form

continueCalcAfterLastNegativeNumberProcessed
                         ; if the symbol = %
        CMP R12, #0x25  ; goto modulus path
        BEQ gotoModulus   ; if the symbol = *
        CMP R12, #0x2A  ; goto multiplication path
        BEQ gotoMult
        CMP R12, #0x2B ; if the symbol = + goto addition path
        BEQ gotoAdd
        CMP R12, #0x2D ; if the symbol = - goto subtraction path
        BEQ gotoSub
        CMP R12, #0x2F  ;if the symbol is divison goto div path
        BEQ gotoDiv
        B invalidinput  ; else goto invalid input


 ; so far we have the 1st input in R4, R7 and know operator branch
```

The calc block, first checks if the last chain value is negative. If not it jumps to the step after the negative calculations, it sets the negative flag to 0

The last number of the chain has the inversion operation preformed on it.

Once all the calculations for each member of the chain and all the negative signs are set, the value that is held in R12, the operator variable, is checked and sent off to their relative branches eg gotoMult or gotoAdd, these are the section 2 of the program.

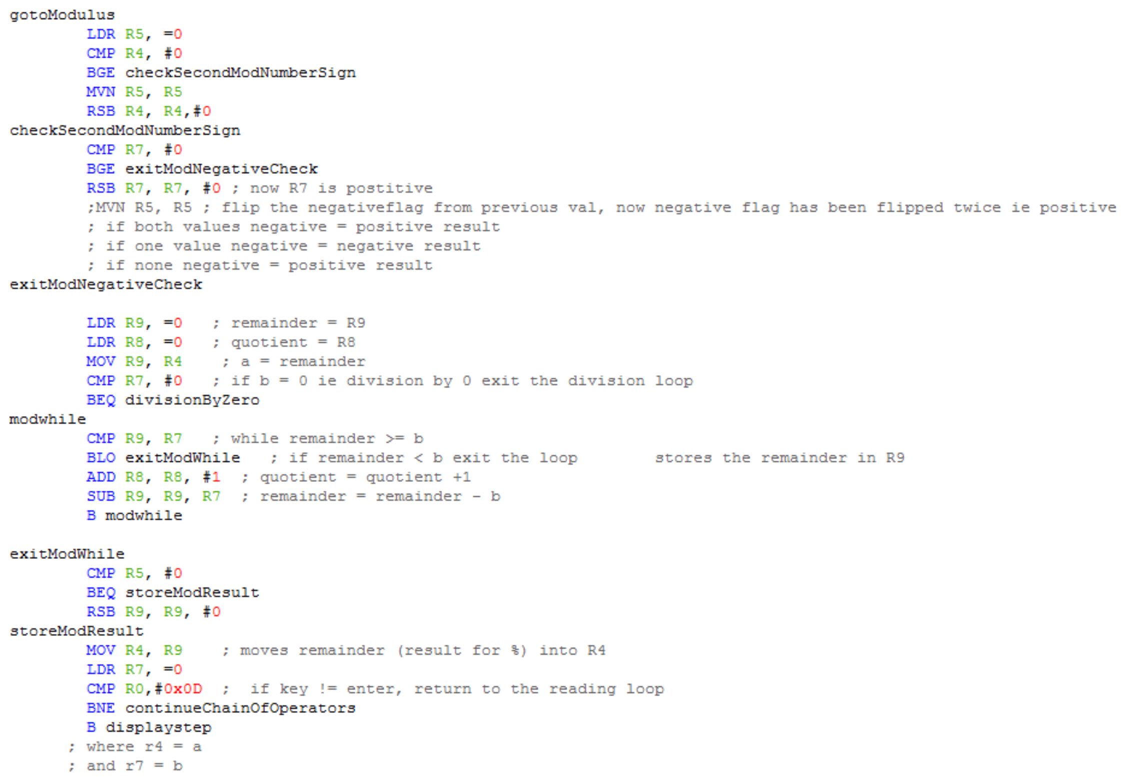
Calculations : Step #2



```
gotoSub

        SUB R4, R4, R7 ; stores result in r4 of r4-r7, to be ready for next operator in chain
        LDR R7, =0
        CMP R0,#0x0D  ;  if key != enter, return to the reading loop
        BNE continueChainOfOperators
        B displaystep


gotoAdd
        ADD R4, R4, R7 ; stores result in r4 of r4+r7  to be ready for next operator in chain
        LDR R7, =0
        CMP R0,#0x0D  ;  if key != enter, return to the reading loop
        BNE continueChainOfOperators
        B displaystep

gotoMult
        MUL R4, R7, R4 ; stores in r4 r4 x r7
        LDR R7, =0
        CMP R0,#0x0D  ;  if key != enter, return to the reading loop
        BNE continueChainOfOperators
        B displaystep
```

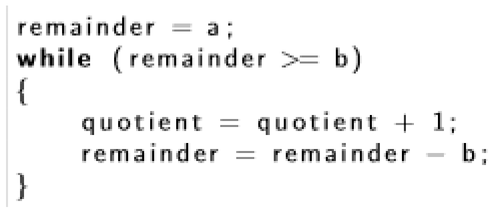
The calculations step for the 3 first operations are straight-forward, subtraction, addition and multiplication have their own inbuilt operators, for example a = b+c where a is R8, b is R4, and c is R7

If the enter key isn’t pressed the calculations don’t head to the displaying step, they continue on the readWhile loop and go to continueChainOperators, which sets R10 the second operand mode to 1 and sets the first digit Boolean to 1 aswell.



```
gotoModulus
        LDR R5, =0
        CMP R4, #0
        BGE checkSecondModNumberSign
        MVN R5, R5
        RSB R4, R4,#0
checkSecondModNumberSign
        CMP R7, #0
        BGE exitModNegativeCheck
        RSB R7, R7, #0 ; now R7 is postitive
        ;MVN R5, R5 ; flip the negativeflag from previous val, now negative flag has been flipped twice ie positive
        ; if both values negative = positive result
        ; if one value negative = negative result
        ; if none negative = positive result
exitModNegativeCheck

        LDR R9, =0    ; remainder = R9
        LDR R8, =0    ; quotient = R8
        MOV R9, R4     ; a = remainder
        CMP R7, #0    ; if b = 0 ie division by 0 exit the division loop
        BEQ divisionByZero
modwhile
        CMP R9, R7    ; while remainder >= b
        BLO exitModWhile    ; if remainder < b exit the loop        stores the remainder in R9
        ADD R8, R8, #1  ; quotient = quotient +1
        SUB R9, R9, R7  ; remainder = remainder - b
        B modwhile

exitModWhile
        CMP R5, #0
        BEQ storeModResult
        RSB R9, R9, #0
storeModResult
        MOV R4, R9     ; moves remainder (result for %) into R4
        LDR R7, =0
        CMP R0,#0x0D  ;  if key != enter, return to the reading loop
        BNE continueChainOfOperators
        B displaystep
      ; where r4 = a
      ; and r7 = b
```

The next block of code is responsible for the calculations carried out in the modulus block. For the modulus operator, the basic operation is very similar to the division algorithm except giving out a result as the remainder of the process, It operates like this:



```
remainder = a;
while (remainder >= b)
{
    quotient = quotient + 1;
    remainder = remainder - b;
}
```

This method is called repeated subtraction, we use it quite a lot throughout steps 2 and 3 of the program, as ARM assembly doesn't have an inbuilt division operator.

The real problems with the modulus block arise when dealing with negative numbers as first and second operands, to solve this I first set a flag to indicate whether the answer will be negative, first setting it to 0 ie +ve, then I check whether the first operand is less than 0, if so I flip the flag, then I repeat this operation for the second operand, if they are both negative, the flag turns out positive, but if only 1 is negative, I make all the values positive and carry the operation out as normal, then I check the flag when I have my result and apply the sign to it.

And if the enter key isn't pressed after the modulus operation is inputted, I go back to the main loop that reads in another operator and sets the needed Booleans for another expression to be inputted.

```
gotoDiv
;       check first number sign
        LDR R5, =0; ; load 0 into negativeflag for division as initial val
        CMP R4, #0
        BGE checkSecondNumberSign
        RSB R4, R4, #0 ; now R4 is positive
        MVN R5, R5 ; flip the negativeflag from initial val
checkSecondNumberSign
        CMP R7, #0
        BGE continueDivision
        RSB R7, R7, #0 ; now R7 is postitive
        MVN R5, R5 ; flip the negativeflag from previous val, now negative flag has been flipped twice ie positive
        ; if both values negative = positive result
        ; if one value negative = negative result
        ; if none negative = positive result
continueDivision

        LDR R9, =0
        LDR R8, =0   ; quotient = R8
        MOV R9, R4    ; a = remainder
        CMP R7, #0   ; if b = 0 ie division by 0 exit the division loop
        BEQ divisionByZero


divwhile
        CMP R9, R7   ; while remainder >= b
        BLO exitDivWhile   ; if remainder < b exit the loop        stores the remainder in R9  (modulus operator action can be transfered here and given the remainder as
        ADD R8, R8, #1  ; quotient = quotient +1
        SUB R9, R9, R7  ; remainder = remainder - b
        B divwhile

exitDivWhile
        CMP R5, #0  ; if negative flag is set flip result
        BEQ storedivisionresult ; else go as normal
        ;flip the R8 value
        RSB R8, R8, #0 ; r8 = 0-r8

storedivisionresult
        MOV R4, R8    ; moves quotient (result for /) into R4
        LDR R7, =0
        CMP R0,#0x0D  ;  if key != enter, return to the reading loop
        BNE continueChainOfOperators
        B displaystep
```

The division operation is the last that is calculated, an operation very similar to the modulus check for negative results is preformed eg -12/-3 = 4, only this time the operation is the division which gives out only a quotient, this is identical to the modulus structure only changing the quotient at the end, The operation of the division is also the repeated subtraction algorithm described above.

This block of code also doesn't stop after getting and result but checks if the user has pressed the enter key after it if so it goes to the display step, or if there isn't a enter key pressed it goes back to looking for a new operator setting the required flags.

Display the result : Step #3

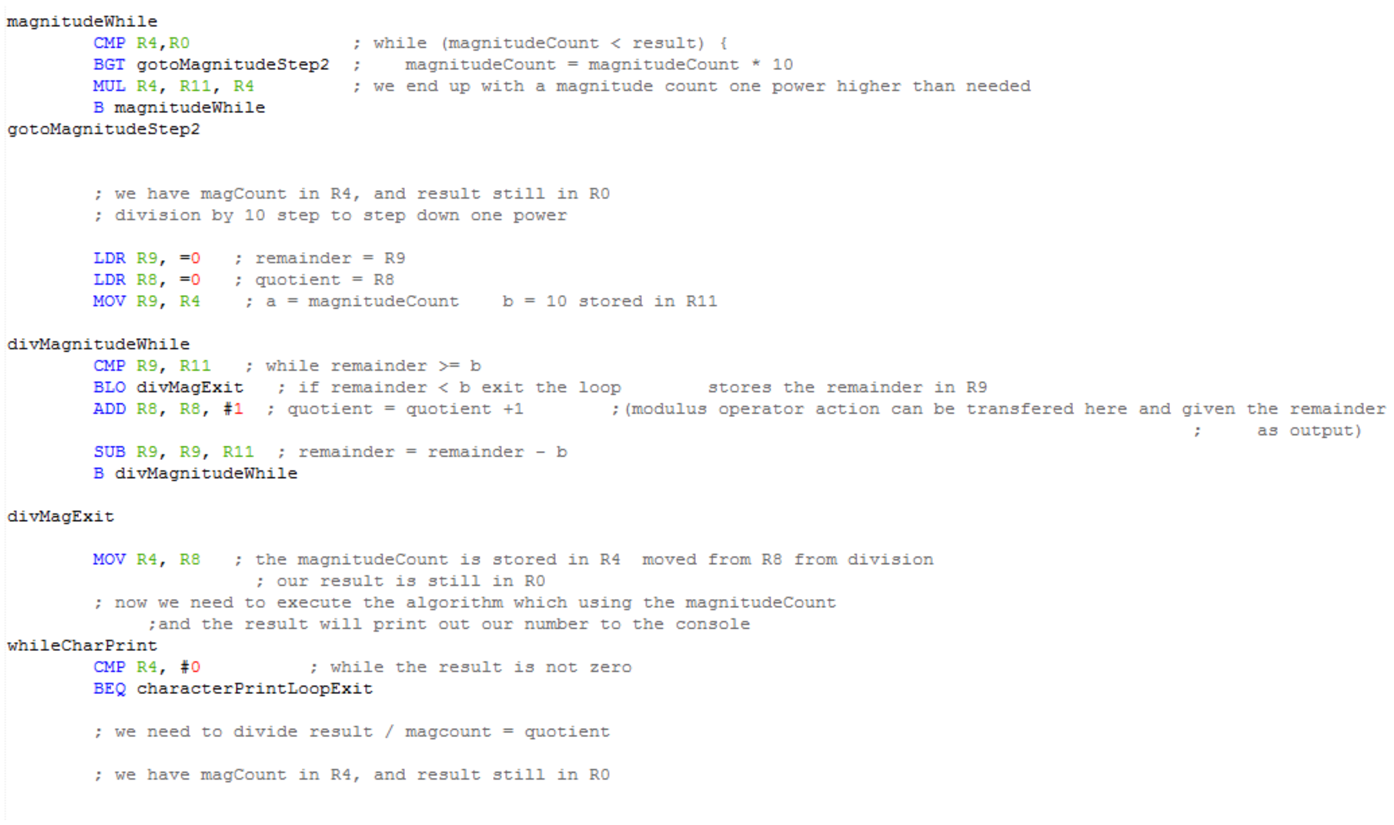
The display step only sets in when we have a value that is in the register and needs to be converted step by step to characters and printed out to the console with the appropriate sign to represent its value well.

```
displaystep
        MOV R8, R4  ; R4 contains operation result move it to R8 for compatibility with old code


        MOV R0,#0x3D    ; equals sign is moved into R0 and printed onto the screen
        BL sendchar

        CMP R8, #0
        BNE printNonZeroResult
        MOV R0,#0x30
        BL sendchar
        B stop
printNonZeroResult

        CMP R8, #0
        BGT printPositiveResult
        RSB R8, R8, #0   ; r8 = 0-r8
        MOV R0,#0x2D    ; prints a "-" sign to the console
        BL sendchar



printPositiveResult

        MOV R0, R8 ; now we have the result (in hex stored in R0)
        LDR R4, =1 ; R4 is the magnitudeCount
        LDR R11, =0xA; negative register not used anymore can be used to store value 10
```

Once we enter the display step we have the result R8 from all the previous steps

To start off for presentation we print out an equals sign to the console after the initial equation inputted. The neck check is whether the result is 0 if it is simply a zero character gets printed to the console, otherwise the number is checked to be negative, if the value is negative a "-" sign is printed out to the console straight away before the main procedure, after these special cases are dealt with, we head over to the main case:

At first the result is moved into R0, R4 is set as the magnitudeCount variable which will be raised to a certain power of 10 to help complete the conversion print process, "10" in hex is loaded into the R11 register not being used anymore to act as a variable eg var =10;



```
magnitudeWhile
        CMP R4,R0               ; while (magnitudeCount < result) {
        BGT gotoMagnitudeStep2  ;    magnitudeCount = magnitudeCount * 10
        MUL R4, R11, R4         ; we end up with a magnitude count one power higher than needed
        B magnitudeWhile
gotoMagnitudeStep2


        ; we have magCount in R4, and result still in R0
        ; division by 10 step to step down one power

        LDR R9, =0    ; remainder = R9
        LDR R8, =0    ; quotient = R8
        MOV R9, R4     ; a = magnitudeCount     b = 10 stored in R11

divMagnitudeWhile
        CMP R9, R11    ; while remainder >= b
        BLO divMagExit    ; if remainder < b exit the loop        stores the remainder in R9
        ADD R8, R8, #1  ; quotient = quotient +1          ;(modulus operator action can be transfered here and given the remainder
                                                                                                              ;     as output)
        SUB R9, R9, R11  ; remainder = remainder - b
        B divMagnitudeWhile

divMagExit

        MOV R4, R8    ; the magnitudeCount is stored in R4  moved from R8 from division
                       ; our result is still in R0
        ; now we need to execute the algorithm which using the magnitudeCount
             ;and the result will print out our number to the console
whileCharPrint
        CMP R4, #0           ; while the result is not zero
        BEQ characterPrintLoopExit

        ; we need to divide result / magcount = quotient

        ; we have magCount in R4, and result still in R0
```

The magnitudeWhile loop starts here this loop calculates the (magnitude level needed *10)

It works by multiplying the magnitudeCount * 10 until it is larger than the result in R0,

For the next step we load in R9 = 0 R8 = 0 and set a in the division process at magnitudeCount and b is set as 10 which is stored in R11

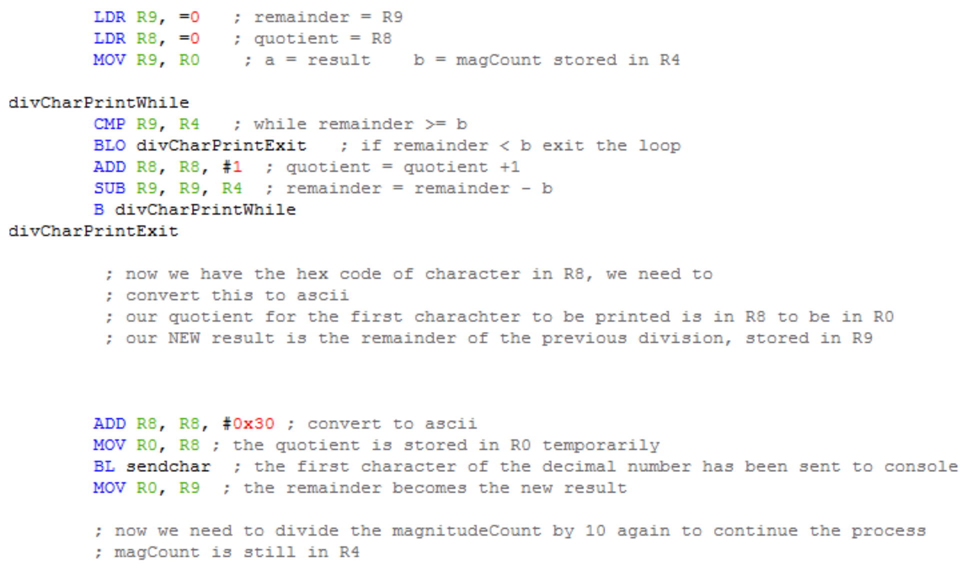
The division algorithm uses repeated subtraction, it works by this principle:

```
While ( remainder >= b )
{
        Quotient = quotient +1
        Remainder = remainder – b
}
```

Once we have divided the magnitudeCount by 10 to receive the desired number for the conversion algorithm eg to convert 4180 to characters you need magCount to be 1000.

We store the result of the division eg the magCount back into R4, the last part of the block shown above initiates the main conversion loop, the condition of the while loop is that while the result is not 0, keep going and trying to print characters, we have the magCount stored in

R4 and the result stored in R0, and need to find repeatedly a quotient by dividing the result by the magcount and incrementing the magCount every time by division by 10, the new remainder of said division is stored as the new result for the new run of the loop, While the quotient has 0x30 added onto it and printed out as a sendchar into the console.



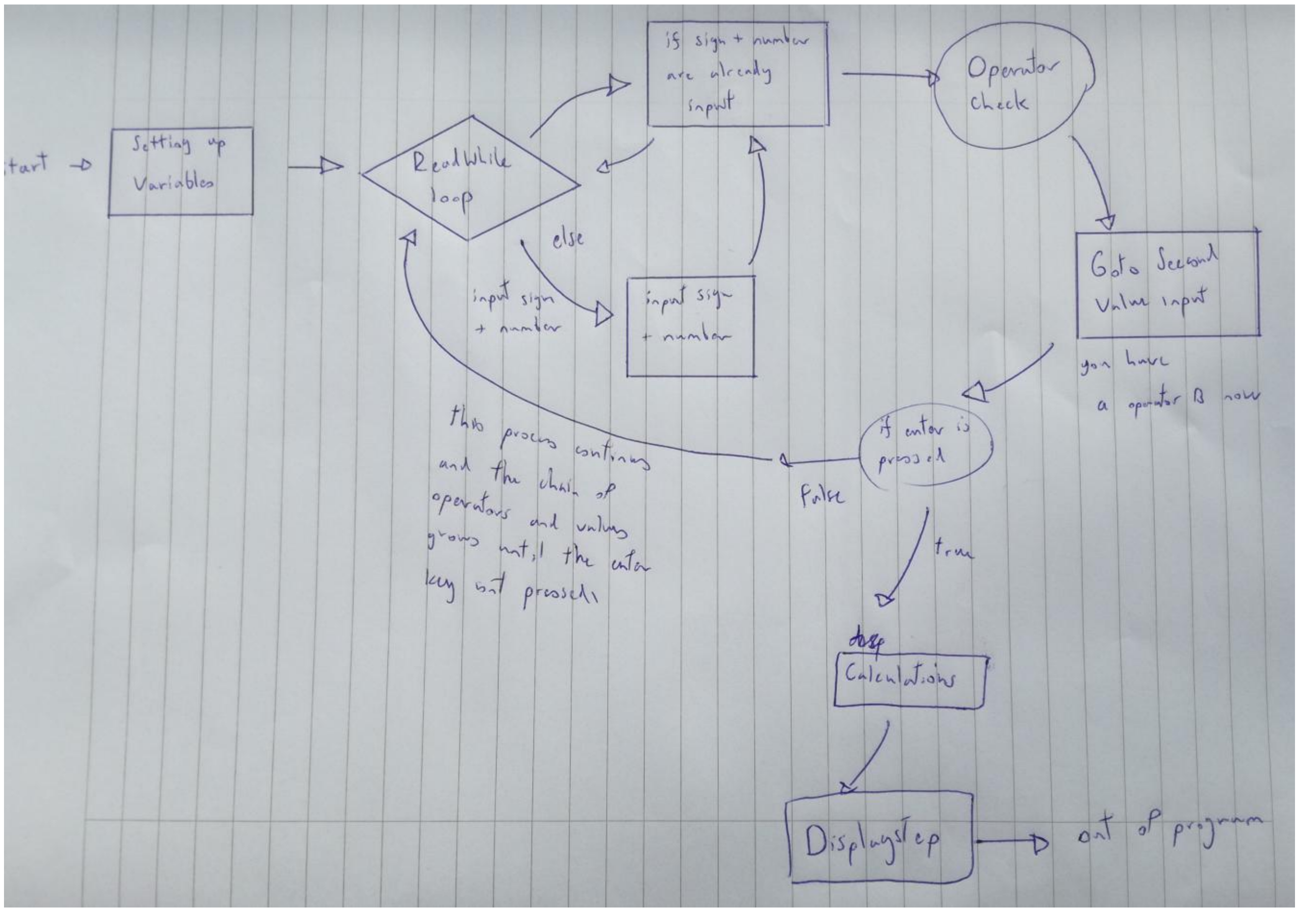
```
        LDR R9, =0   ; remainder = R9
        LDR R8, =0   ; quotient = R8
        MOV R9, R0    ; a = result    b = magCount stored in R4

divCharPrintWhile
        CMP R9, R4   ; while remainder >= b
        BLO divCharPrintExit   ; if remainder < b exit the loop
        ADD R8, R8, #1  ; quotient = quotient +1
        SUB R9, R9, R4  ; remainder = remainder - b
        B divCharPrintWhile
divCharPrintExit

         ; now we have the hex code of character in R8, we need to
         ; convert this to ascii
         ; our quotient for the first charachter to be printed is in R8 to be in R0
         ; our NEW result is the remainder of the previous division, stored in R9


        ADD R8, R8, #0x30 ; convert to ascii
        MOV R0, R8 ; the quotient is stored in R0 temporarily
        BL sendchar  ; the first character of the decimal number has been sent to console
        MOV R0, R9  ; the remainder becomes the new result

        ; now we need to divide the magnitudeCount by 10 again to continue the process
        ; magCount is still in R4
```

Like normally when performing the division algorithm, the R8, R9 registers are used to store the quotient and remainder, this loop works as described in the last paragraph, to continue after this block we still need to increment the magCount by dividing it by 10, the same division algorithm is used to perform this.

```
        ;
        ; R11 stored value 10

        LDR R9, =0    ; remainder = R9
        LDR R8, =0    ; quotient = R8
        MOV R9, R4     ; a = magnitudeCount    b = 10 stored in R11

divMagnitudeDivision
        CMP R9, R11    ; while remainder >= b
        BLO divMagDivExit    ; if remainder < b exit the loop        stores the remainder in R9
        ADD R8, R8, #1   ; quotient = quotient +1
        SUB R9, R9, R11   ; remainder = remainder - b
        B divMagnitudeDivision

divMagDivExit
        ;  quotient from division eg new magCount stored in R8


        MOV R4, R8

; now we have the new magnitudeCount in R4
; we have our result in R0
; and we have the first character printed out to console
; now we can loop back to the mainfinal while loop to continue this process until the result ==0
        B whileCharPrint
characterPrintLoopExit


        B stop
invalidinput
        MOV R0, #0x3F    ; if the input is invalid, a question mark is sent into the console
        BL sendchar
        B stop

divisionByZero
        MOV R0, #0x21    ; if the division is by 0, a exclamation mark is sent into the console
        BL sendchar



stop    B   stop

    END
```

The division algorithm is preformed on the magCount dividing it by 10 for the next run of the loop, after this we jump back to the loop tag whileCharPrint, this loop keeps going until the whole result value is printed out the console with a suitable sign and correct formatting.

Simplified Flowchart of the operation of the program:



Testing each of the program steps : Step #4

Test cases for inputting different numbers into the Console Input step.

Positive Numbers: 123 , 74, 100, 1000000	0x7B, 0x4A, 0x64 , 0xF4240
Negative Numbers: - 12, -100 , -22	0xFFFFFFF4 , 0xFFFFFF9C, 0xFFFFFFEA

Testing for different types of calculations and their outputs.

#1 Addition Cases

Types of Addition Equations	Results in register // In console
-a+b // -100 + 250 = result	0x96 // 150
-a +(-b) // -50 + (-100) = result	0x FFFFFF6A // -150
a+b // 12+ 56 = result	0x44 // 68
a + (-b) // 625+(-626) = result	0xFFFFFFFF // -1
a + b + c + d .. // 12 + 12 + 6 + 6+2 + 1	0x27 //39
-a + -b + -c + -d // -12 +-5 +-6+-7	0xFFFFFFE2 //-30

#2 Subraction Cases

Types of Subraction Equations	Result in register // In console
a-b // 45 – 22 = result	0x17 // 23
-a –b // -500-300= result	0x FFFFFCE0 // -800
-a –(-b) // -12-(-12) = result	0x00 // 0
a – (-b) // 66- (-12) = result	0x4E // 78
a –b –c – d... // 6-2-5-12	0xFFFFFFF3 // -13

#3 Multiplication Cases

Types of Multiplication Equations	Results
a * b // 12*5	0x3C // 60
-a * -b // -12 * -5	0x3C // 60
b * b // 12 * 12	0x90 // 144
-a * b // -60 * 3	0xFFFFFF4C//-180
a*-b // 180 * -3	0xFFFFFDE4 // -540

a * b * c * d … and so on // 2* 3* 4 * 5	0x78//120
-a * -b * -c * -d … // -3*-2*-1*-4	0x18 // 24

#4 Division Cases

Types of Division Equations	Result
a / b // 12/4	0x3 // 3
A /a // 12/12	0x1 // 1
-a / b // -12/4	0xFFFFFFFD// -3
-a / -b // -70/-7	0xA// 10
a / -b // 10/-2	0xFFFFFFFB// -5
a / b / c / d/ e… // 40/2/4/2	0x2// 2
-a / -b / -c /-d … // -12/-2/-2/-2	0x1// 1

#4 Modulus Cases

Types of Modulus Equations	Result
a % b // 12 % 5	0x2 // 2
a % a // 12 % 12	0x0//0
-a % b // -12 % 5	0xFFFFFFFE // -2
-a % -b // -12%-5	0xFFFFFFFE // -2
a % -b // 12 % -5	0x2 // 2
a % b % c %d… 24 % 7 % 3	0x // 0
-a % -b % -c % -d… -20 % -6 %-2	0x // 0

#5 Other testing cases

Equations	Results
a * b * c * d + e + f // 12*4*3 +12+12	0xA8 // 168
a/ b / c + d –e // 40/2/2+2-6	0x6 // 6
a%b*c+d //12%5 * 2 +33	0x25 // 37
a /0 // 12/0	! = this sign indicates division by zero, not allowed
a * nothing or a / nothing	? printed out to console for invalid input.

The majority of testing that I did for this project was throughout the process , when a part of my project wasn’t working I would use the debugger and try to make blocks that print a character in a certain place to see whether my program reached this place, in the program, I would use these equation tests nearer to the end of my program to make final checks on whether everything was working correctly.

The order of operations isn’t taken into account, because this is a whole problem on its own separate from the assignment, and requires stack implementation.