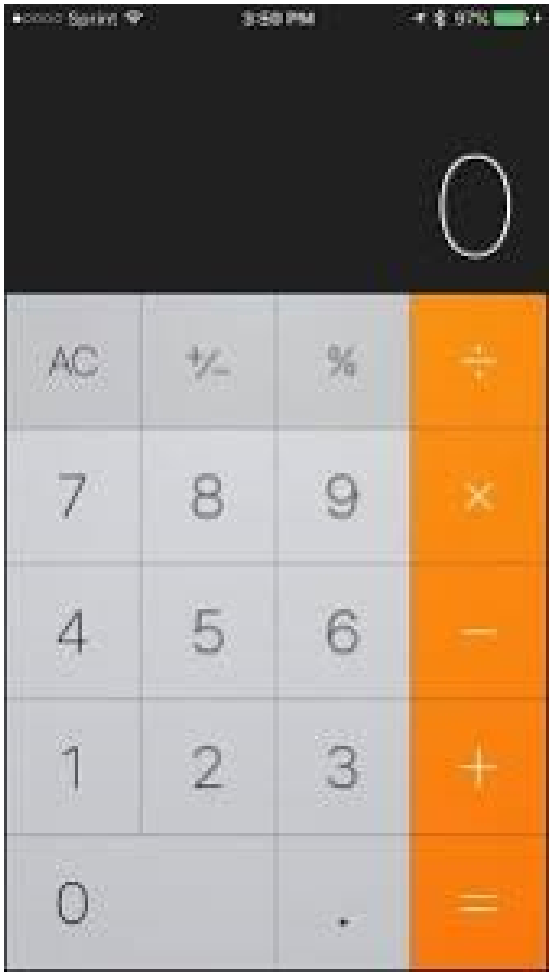
# ECE 220 Computer Systems & Programming

## Lecture 5 – Programming with Stack

## January 29, 2019

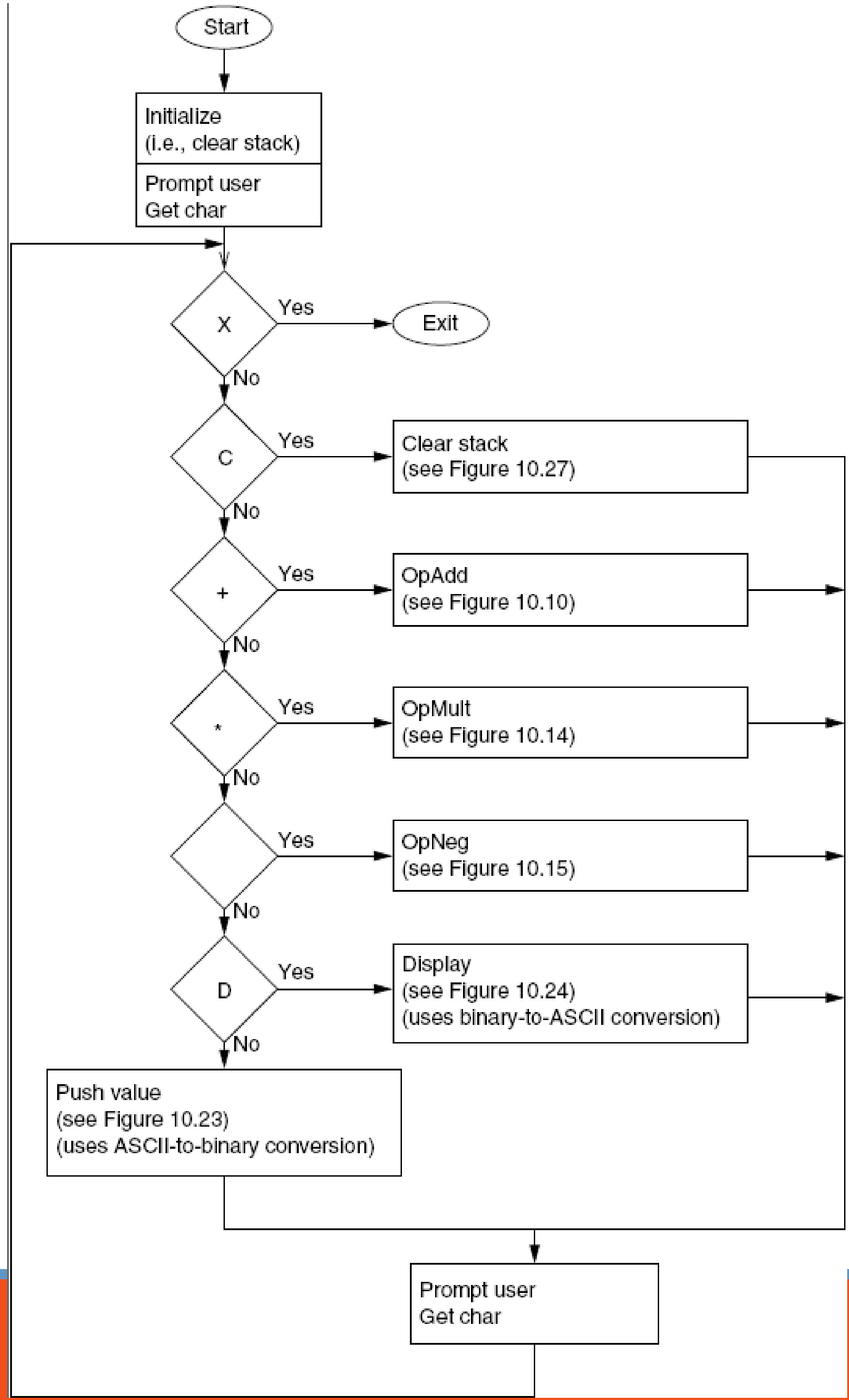
ECE ILLINOIS

ILLINOIS




**X**- Exit the Simulation
**D**- Display the Result from Stack Top
**C**-Clear Stack
**+** Perform Addition
**-** Negate the top element on the stack
***** Perform Multiplication
**Enter** – Push the typed data onto the stack

# Flow Chart of the Calculator




**X**- Exit the Simulation
**D**- Display the Result from Stack Top
**C**-Clear Stack
**+** Perform Addition
**-** Negate the top element on the stack
***** Perform Multiplication
**Enter** – Push the typed data onto the stack

# What are the different subroutines we need?

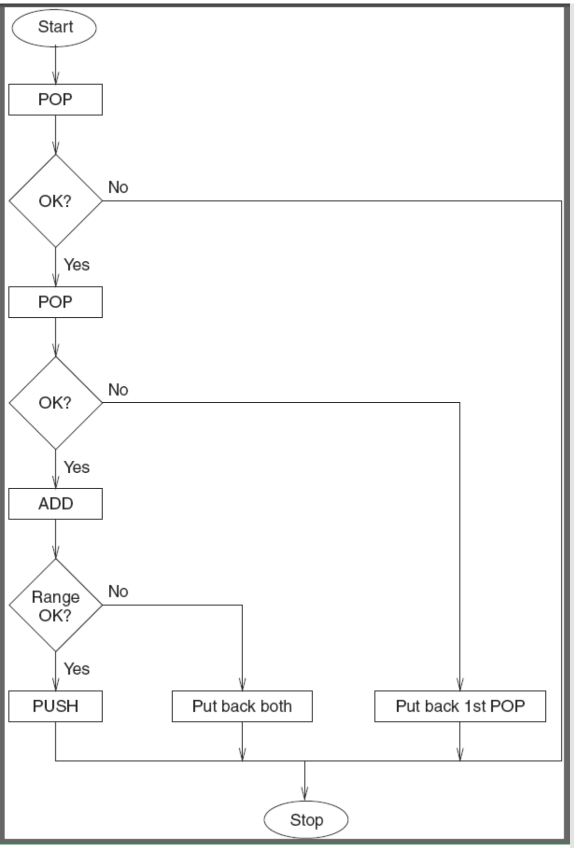
- OpAdd (Adder subroutine)
- OpMult (Multiplication subroutine)
- OpNeg (Negate subroutine)
- Range Check (integers in the range  -999 and +999)
- AsciitoBinary (input data type conversion)
- BinarytoAscii (output data type conversion)
- PushValue subroutine (which push input onto the Stack)
- OpDisplay (Which pop the result from stack and display Ascii result on the screen)
- OpClear (which clear the stack)

# Main Function:

```
;   The Calculator, Main Algorithm
;
.ORIG   x3000
                LEA     R6,StackBase    ; Initialize the
                ADD     R6,R6,#-1       ; R6 is stack poi
                LEA     R0,PromptMsg
                PUTS
                GETC
                OUT
;
; Check the command
;
Test            LD      R1,NegX         ; Check for X
                ADD     R1,R1,R0
                BRz     Exit
;
                LD      R1,NegC         ; Check for C
                ADD     R1,R1,R0
                BRz     OpClearC
;
                LD      R1,NegPlus      ; Ch
                ADD     R1,R1,R0
                BRz     OpAddC
;
                LD      R1,NegMult      ; Ch
                ADD     R1,R1,R0
                BRz     OpMultC
;
                LD      R1,NegMinus     ; Ch
                ADD     R1,R1,R0
                BRz     OpNegC
;
                LD      R1,NegD         ; Ch
                ADD     R1,R1,R0
                BRz     OpDisplayC
;
; Then we must be entering an integer
;
                JSR     PushValue       ; See
```

```
NewCommand      LEA     R0,PromptMsg
                PUTS
                GETC
                OUT
                BRnzp   Test

OpClearC    JSR OpClear
            BRnzp NewCommand
OpAddC      JSR OpAdd
            BRnzp NewCommand
OpMultC     JSR OpMult
            BRnzp NewCommand
OpNegC      JSR OpNeg
            BRnzp NewCommand
OpDisplayC  JSR OpDisplay
            BRnzp NewCommand

Exit            HALT
PromptMsg       .FILL       x000A
                .STRINGZ "Enter a command:"
NegX            .FILL       xFFA8
NegC            .FILL       xFFBD
NegPlus         .FILL       xFFD5
NegMinus        .FILL       xFFD3
NegMult         .FILL       xFFD6
NegD            .FILL       xFFBC
```

# OpAdd

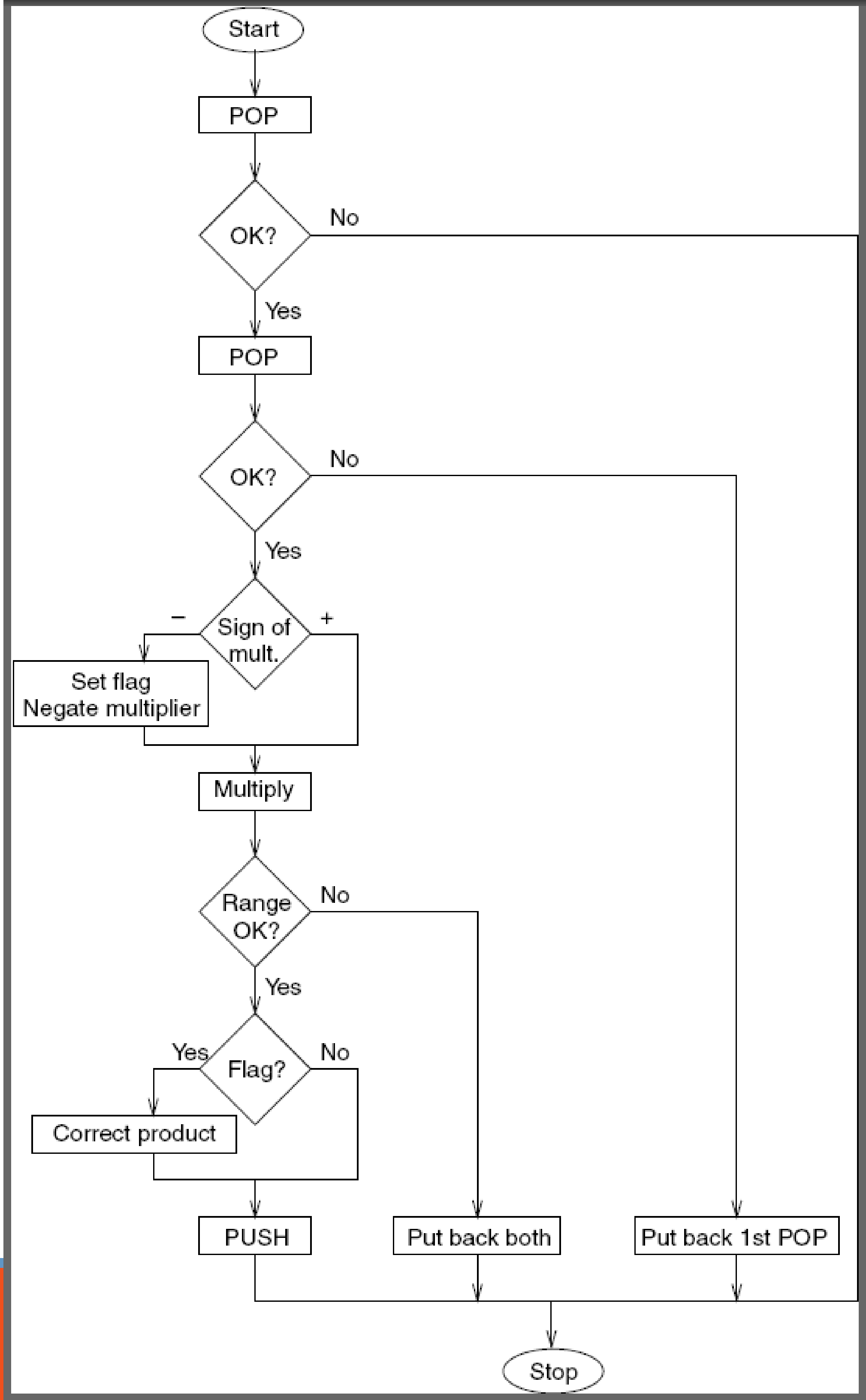



```
;          Routine to pop the top two elements from the stack,
;          add them, and push the sum onto the stack.  R6 is
;          the stack pointer.
;
OpAdd          ST       R7, Save_OpAdd
               JSR      POP                  ; Get first source operand.
               ADD      R5,R5,#0             ; Test if POP was successful.
               BRp      Exit_A                 ; Branch if not successful.
               ADD      R1,R0,#0             ; Make room for second operand
               JSR      POP                  ; Get second source operand.
               ADD      R5,R5,#0             ; Test if POP was successful.
               BRp      Restore1_A             ; Not successful, put back first.
               ADD      R0,R0,R1             ; THE Add.
               JSR      RangeCheck           ; Check size of result.
               BRp      Restore2_A             ; Out of range, restore both.
               JSR      PUSH                 ; Push sum on the stack.
               LD       R7, Save_OpAdd
               RET                           ; Return to the Main Program
Restore2_A        ADD      R6,R6,#-1            ; Decrement stack pointer.
Restore1_A        ADD      R6,R6,#-1            ; Decrement stack pointer.
Exit_A            LD       R7, Save_OpAdd
                  RET          ; Return to the Main Program

Save_OpAdd        .BLKW #1
```

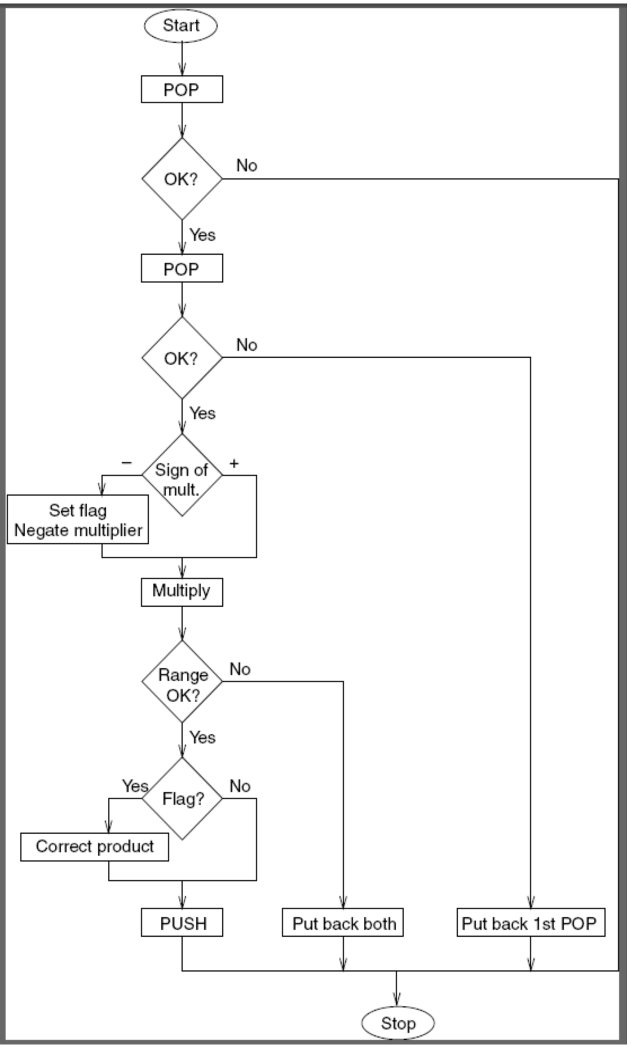
# OpNeg Subroutine

```
;     Algorithm to pop the top of the stack, form its negative,
;     and push the result on the stack.
;
OpNeg           ST R7, Save_OpNeg
                JSR     POP             ; Get the source operand
                ADD     R5,R5,#0        ; test for successful pop
                BRp     Exit_N            ; Branch if failure
                NOT     R0,R0
                ADD     R0,R0,#1        ; Form the negative of the source.
                JSR     PUSH            ; Push the result on the stack.
Exit_N          LD R7, Save_OpNeg
                RET                     ;Return to the Main Program

Save_OpNeg     .BLKW #1
```

# OpMult:

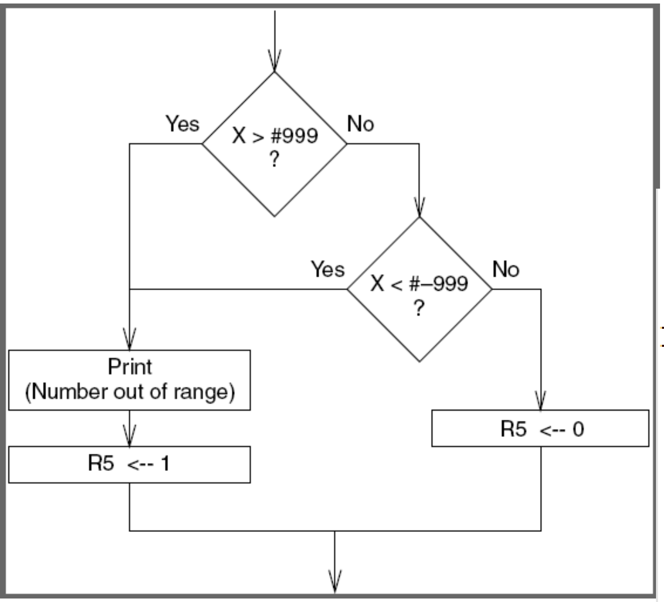

Start
POP
OK?
No
Yes
POP
OK?
No
Yes
Sign of mult.
−
+
Set flag
Negate multiplier
Multiply
Range OK?
No
Yes
Flag?
Yes
No
Correct product
PUSH
Put back both
Put back 1st POP
Stop

# OpMult (code)



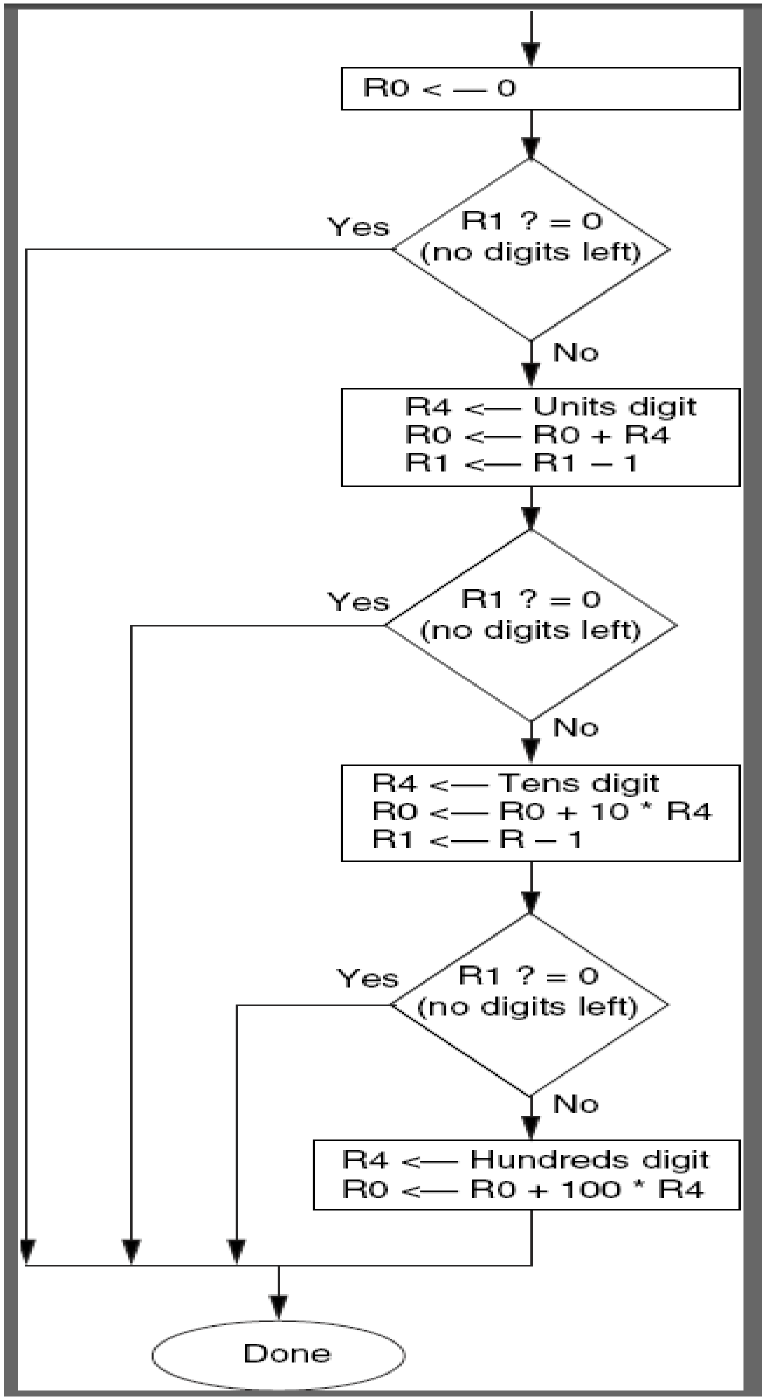

```
;     Algorithm to pop two values from the stack, multiply them
;     and if their product is within the acceptable range, push
;     the result on the stack.  R6 is stack pointer.
;
OpMult          ST R7, Save_OpMult
                AND    R3,R3,#0        ; R3 holds sign of multiplier.
                JSR    POP             ; Get first source from stack.
                ADD    R5,R5,#0        ; Test for successful POP
                BRp    Exit_M             ; Failure
                ADD    R1,R0,#0        ; Make room for next POP
                JSR    POP             ; Get second source operand
                ADD    R5,R5,#0        ; Test for successful POP
                BRp    Restore1_M         ; Failure; restore first POP
                ADD    R2,R0,#0        ; Moves multiplier, tests sign
                BRzp   PosMultiplier
                ADD    R3,R3,#1        ; Sets FLAG: Multiplier is neg
                NOT    R2,R2
                ADD    R2,R2,#1        ; R2 contains -(multiplier)
PosMultiplier   AND    R0,R0,#0        ; Clear product register
                ADD    R2,R2,#0
                BRz    PushMult        ; Multiplier = 0, Done.
;
MultLoop        ADD    R0,R0,R1        ; THE actual "multiply"
                ADD    R2,R2,#-1       ; Iteration Control
                BRp    MultLoop
;
                JSR    RangeCheck
                ADD    R5,R5,#0        ; R5 contains success/failure
                BRp    Restore2_M
;
                ADD    R3,R3,#0        ; Test for negative multiplier
                BRz    PushMult
                NOT    R0,R0           ; Adjust for
                ADD    R0,R0,#1        ; sign of result
PushMult        JSR    PUSH            ; Push product on the stack.
                LD     R7, Save_OpMult
                RET    ; Return to the Main Program
Restore2_M         ADD    R6,R6,#-1       ; Adjust stack pointer.
Restore1_M         ADD    R6,R6,#-1       ; Adjust stack pointer.
Exit_M             LD     R7, Save_OpMult
                   RET    ; Return to the Main Program
Save_OpMult       .BLKW #1
```

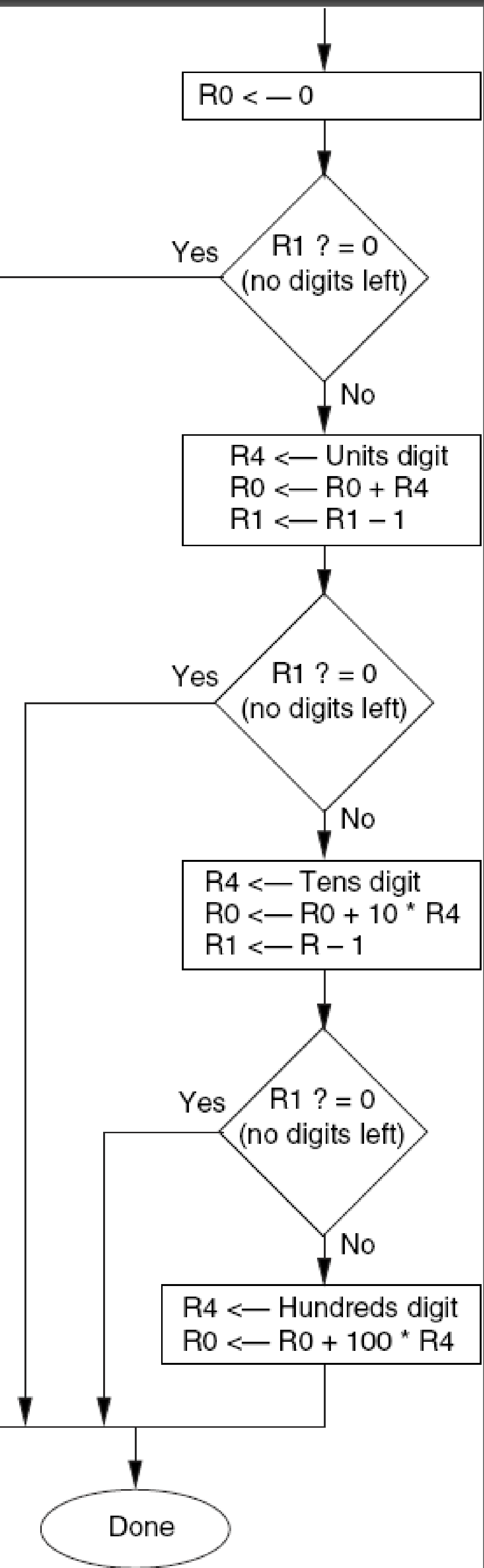
# RangeCheck




```
;       Routine to check that the magnitude of a value is
;       between -999 and +999.
;
RangeCheck    LD       R5,Neg999
              ADD      R4,R0,R5    ; Recall that R0 contains the
              BRp      BadRange    ; result being checked.
              LD       R5,Pos999
              ADD      R4,R0,R5
              BRn      BadRange
              AND      R5,R5,#0    ; R5 <-- success
              RET
BadRange      ST       R7,Save_R     ; R7 is needed by TRAP/RET
              LEA      R0,RangeErrorMsg
              TRAP     x22         ; Output character string
              LD       R7,Save_R
              AND      R5,R5,#0    ;
              ADD      R5,R5,#1    ; R5 <-- failure
              RET
Neg999        .FILL    #-999
Pos999        .FILL    #999
Save_R          .FILL    x0000
RangeErrorMsg .FILL    x000A
              .STRINGZ "Error: Number is out of range."
```

# AsciitoBinary






```
;  This algorithm takes an ASCII string of three decimal digits and
;  converts it into a binary number.  R0 is used to collect the result.
;  R1 keeps track of how many digits are left to process.  ASCIIBUFF
;  contains the most significant digit in the ASCII string.
;
ASCIItoBinary   AND     R0,R0,#0        ; R0 will be used for our result
                ADD     R1,R1,#0        ; Test number of digits.
                BRz     DoneAtoB        ; There are no digits
;
                LD      R3,NegASCIIOffset  ; R3 gets xFFD0, i.e., -x0030
                LEA     R2,ASCIIBUFF
                ADD     R2,R2,R1
                ADD     R2,R2,#-1       ; R2 now points to "ones" digit
;
                LDR     R4,R2,#0        ; R4 <-- "ones" digit
                ADD     R4,R4,R3        ; Strip off the ASCII template
                ADD     R0,R0,R4        ; Add ones contribution
;
                ADD     R1,R1,#-1
                BRz     DoneAtoB        ; The original number had one digit
                ADD     R2,R2,#-1       ; R2  now points to "tens" digit
;
                LDR     R4,R2,#0        ; R4 <-- "tens" digit
                ADD     R4,R4,R3        ; Strip off ASCII  template
                LEA     R5,LookUp10     ; LookUp10 is BASE of tens values
                ADD     R5,R5,R4        ; R5 points to the right tens value
                LDR     R4,R5,#0
                ADD     R0,R0,R4        ; Add tens contribution to total
;
                ADD     R1,R1,#-1
                BRz     DoneAtoB        ; The original number had two digits
                ADD     R2,R2,#-1       ; R2 now points to "hundreds" digit
;
                LDR     R4,R2,#0        ; R4 <-- "hundreds" digit
                ADD     R4,R4,R3        ; Strip off ASCII template
                LEA     R5,LookUp100    ; LookUp100 is hundreds BASE
                ADD     R5,R5,R4        ; R5 points to hundreds value
                LDR     R4,R5,#0
                ADD     R0,R0,R4        ; Add hundreds contribution to total
DoneAtoB        RET
```

# AsciitoBinary (Continued)

```
NegASCIIOffset .FILL   xFFD0
ASCIIBUFF      .BLKW   #4
LookUp10       .FILL   #0
               .FILL   #10
               .FILL   #20
               .FILL   #30
               .FILL   #40
               .FILL   #50
               .FILL   #60
               .FILL   #70
               .FILL   #80
               .FILL   #90
;
LookUp100      .FILL   #0
               .FILL   #100
               .FILL   #200
               .FILL   #300
               .FILL   #400
               .FILL   #500
               .FILL   #600
               .FILL   #700
               .FILL   #800
               .FILL   #900
```

# BinarytoAscii

```
;  This algorithm takes the 2's complement representation of a signed
;  integer, within the range -999 to +999, and converts it into an ASCII
;  string consisting of a sign digit, followed by three decimal digits.
;  R0 contains the initial value being converted.
;
BinarytoASCII  LEA    R1,ASCIIBUFF  ; R1 points to string being generated
               ADD    R0,R0,#0      ; R0 contains the binary value
               BRn    NegSign       ;
               LD     R2,ASCIIplus  ; First store the ASCII plus sign
               STR    R2,R1,#0
               BRnzp  Begin100
NegSign        LD     R2,ASCIIminus ; First store ASCII minus sign
               STR    R2,R1,#0
               NOT    R0,R0         ; Convert the number to absolute
               ADD    R0,R0,#1      ; value; it is easier to work with.
Begin100       LD     R2,ASCIIoffset ; Prepare for "hundreds" digit
               LD     R3,Neg100     ; Determine the hundreds digit
Loop100        ADD    R0,R0,R3
               BRn    End100
               ADD    R2,R2,#1
               BRnzp  Loop100
End100         STR     R2,R1,#1    ; Store ASCII code for hundreds digit
               LD      R3,Pos100
               ADD     R0,R0,R3    ; Correct R0 for one-too-many subtracts
;
               LD      R2,ASCIIoffset ; Prepare for "tens" digit
;
Begin10        LD      R3,Neg10    ; Determine the tens digit
Loop10         ADD     R0,R0,R3
               BRn     End10
               ADD     R2,R2,#1
               BRnzp   Loop10
;
End10          STR     R2,R1,#2    ; Store ASCII code for tens digit
               ADD     R0,R0,#10   ; Correct R0 for one-too-many subtracts
Begin1         LD      R2,ASCIIoffset ; Prepare for "ones" digit
               ADD     R2,R2,R0
               STR     R2,R1,#3
               RET
```

```
ASCIIplus      .FILL   x002B
ASCIIminus     .FILL   x002D
ASCIIoffset    .FILL   x0030
Neg100         .FILL   xFF9C
Pos100         .FILL   x0064
Neg10          .FILL   xFFF6
```

I L L I N O I S

# PushValue

```
; This algorithm takes a sequence of ASCII digits typed by the user,
; converts it into a binary value by calling the ASCIItoBinary
; subroutine and pushes the binary value onto the stack.
;
PushValue       ST      R7, Save_PushValue
                LEA     R1,ASCIIBUFF  ; R1 points to string being
                LD      R2,MaxDigits  ; generated
;
ValueLoop       ADD     R3,R0,xFFF6   ; Test for carriage return
                BRz     GoodInput
                ADD     R2,R2,#0
                BRz     TooLargeInput
                ADD     R2,R2,#-1     ; Still room for more digits
                STR     R0,R1,#0      ; Store last character read
                ADD     R1,R1,#1
                GETC
                OUT                   ; Echo it
                BRnzp   ValueLoop
;
GoodInput       LEA     R2,ASCIIBUFF
                NOT     R2,R2
                ADD     R2,R2,#1
                ADD     R1,R1,R2      ; R1 now contains no. of char.
                JSR     ASCIItoBinary
                JSR     PUSH
                LD      R7, Save_PushValue
                RET
;
TooLargeInput   GETC                  ; Spin until carriage return
                OUT
                ADD     R3,R0,xFFF6
                BRnp    TooLargeInput
                LEA     R0,TooManyDigits
                PUTS
                LD      R7, Save_PushValue
                RET
TooManyDigits   .FILL   x000A
                .STRINGZ "Too many digits"
MaxDigits       .FILL   x0003
Save_PushValue  .BLKW   #1
```

# POP Subroutine

```
;  This algorithm POPs a value from the stack and puts it in
;  R0 before returning to the calling program.  R5 is used to
;  report success (R5=0) or failure (R5=1) of the POP operation.
POP             LEA     R0,StackBase
                NOT     R0,R0
                ADD     R0,R0,#1        ; R0 = -addr.ofStackBase
                ADD     R0,R0,R6        ; R6 = StackPointer
                BRz     Underflow
                LDR     R0,R6,#0        ; The actual POP
                ADD     R6,R6,#1       ; Adjust StackPointer
                AND     R5,R5,#0        ; R5 <-- success
                RET
Underflow       ST      R7,Save_P         ; TRAP/RET needs R7
                LEA     R0,UnderflowMsg
                PUTS                    ; Print error message.
                LD      R7,Save_P         ; Restore R7
                AND     R5,R5,#0
                ADD     R5,R5,#1        ; R5 <-- failure
                RET
Save_P            .FILL   x0000
StackMax        .BLKW   #9
StackBase       .FILL   x0000
UnderflowMsg    .FILL   x000A
                .STRINGZ "Error: Too Few Values on the Stack."
```

# PUSH Subroutine

```
;   This algorithm PUSHes on the stack the value stored in R0.
;   R5 is used to report success (R5=0) or failure (R5=1) of
;   the PUSH operation.
;
PUSH            ST      R1,Save1_push       ; R1 is needed by this routine
                LEA     R1,StackMax
                NOT     R1,R1
                ADD     R1,R1,#1        ; R1 = - addr. of StackMax
                ADD     R1,R1,R6        ; R6 = StackPointer
                BRz     Overflow
                ADD     R6,R6,#-1       ; Adjust StackPointer for PUSH
                STR     R0,R6,#0        ; The actual PUSH
                BRnzp   Success_exit
Overflow        ST      R7,Save_push
                LEA     R0,OverflowMsg
                PUTS
                LD      R7,Save_push
                LD      R1, Save1_push      ; Restore R1
                AND     R5,R5,#0
                ADD     R5,R5,#1        ; R5 <-- failure
                RET
Success_exit    LD      R1,Save1_push       ; Restore R1
                AND     R5,R5,#0        ; R5 <-- success
                RET
Save_push           .FILL   x0000
Save1_push          .FILL   x0000
OverflowMsg     .STRINGZ "Error: Stack is Full."
```

# OpDisplay and OpClear

```
; This algorithm calls BinarytoASCII to convert the 2's complement
; number on the top of the stack into an ASCII character string, and
; then calls PUTS to display that number on the screen.
OpDisplay      ST          R7,Save_OpDisplay
               JSR         POP          ; R0 gets the value to be displayed
               ADD         R5,R5,#0
               BRp         NewCommandC  ; POP failed, nothing on the stack.
               JSR         BinarytoASCII
               LD          R0,NewlineChar
               OUT
               LEA         R0,ASCIIBUFF
               PUTS
               ADD         R6,R6,#-1    ; Push displayed number back on stack
               LD          R7, Save_OpDisplay
               RET

NewCommandC    LD          R7, Save_OpDisplay
               RET

NewlineChar      .FILL   x000A
Save_OpDisplay   .BLKW   #1
;

;
; This routine clears the stack by resetting the stack pointer (R6).
;
OpClear        ST          R7,Save_OpClear
               LEA         R6,StackBase   ; Initialize the Stack.
               ADD         R6,R6,#-1        ; R6 is stack pointer
               LD          R7, Save_OpClear
               RET
Save_OpClear .BLKW #1
```

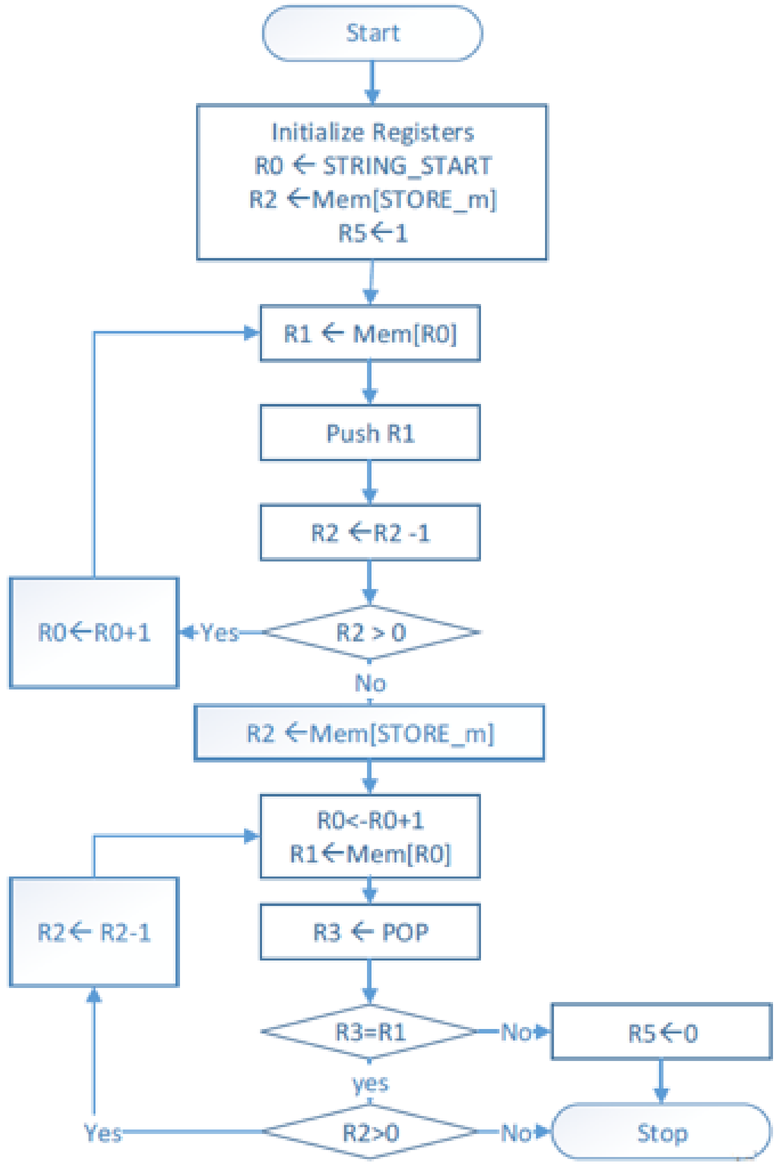
# Exercise:

- Palindrome Checker
- Arithmetic Postfix Evaluation

# Palindrome checker

- A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward or forward
- Examples of palindromes
  - Madam
  - Kayak
  - race car
  - 123456654321
  - Was it a car or a cat I saw?
- Problem statement: implement a program that checks if a given *word* is a palindrome
  - INPUTS: String starting from memory location STRING_START
    - Length of the string is **2m**, m is stored in memory location STORE_m
  - OUTPUT: R5=1 if palindrome and R5=0 indicates not a palindrome
  - Assume that the string is NUL terminated; no spaces and punctuations.
- Overall algorithm
  - Store first m characters of the input string in a stack
    - This will let use to read them backwards
  - Check the remaining m characters against the characters stored in the stack
    - If they are identical, the word is a palindrome

- Let's use registers as follows:
  - R0: address of character being read
  - R1: current character being read
  - R3: 'mirror' character
  - R2: (m - # characters read)
- Flowchart



- Some open questions about this implementation
    - How to handle strings of odd length (2m+1)?
    - What if the length of string is not known a priori?
    - How to handle punctuations and space?

# Arithmetic using a stack

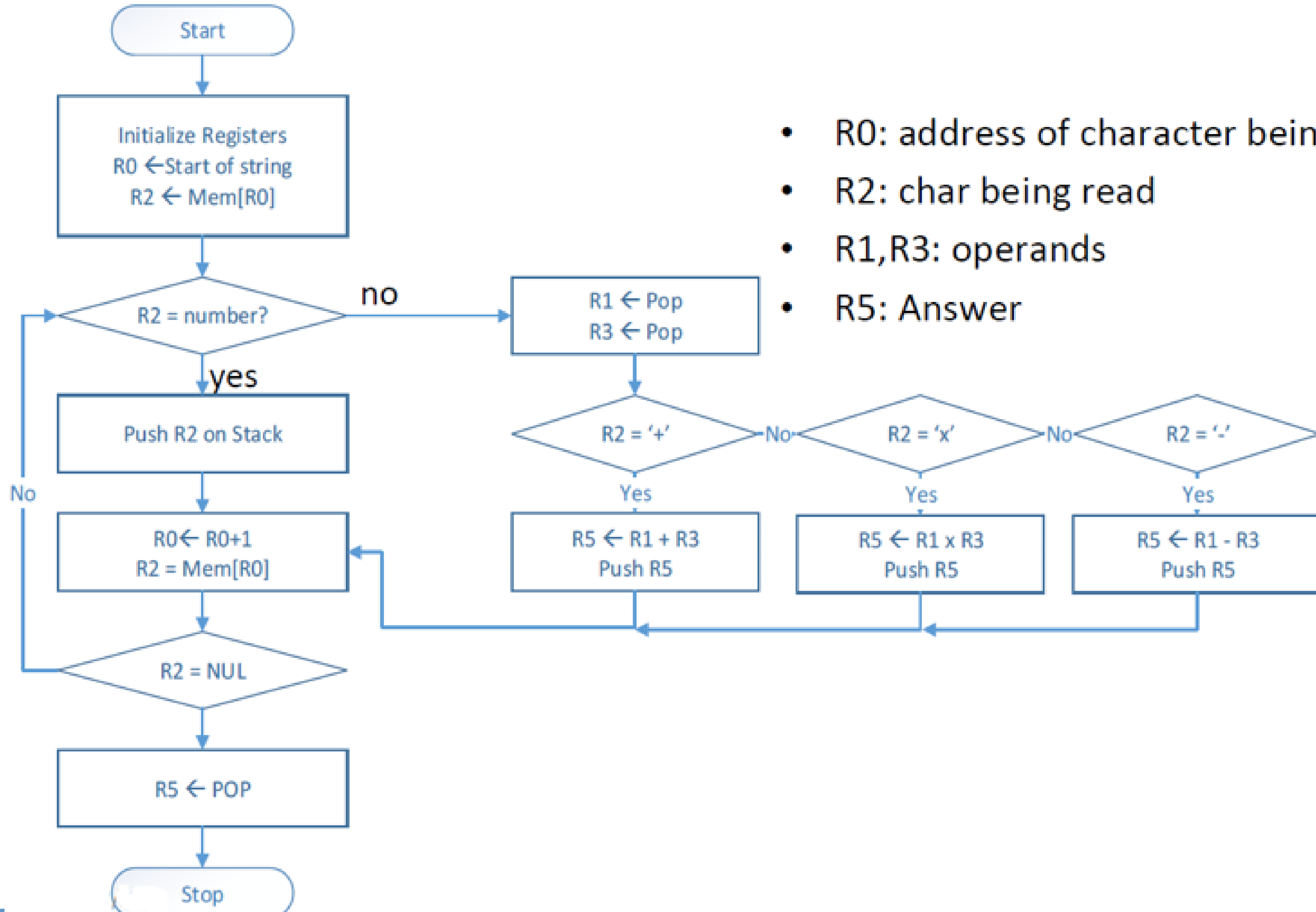
## Postfix expressions

- A postfix expression is a sequence of numbers ('1','5', etc.) and operators ('+', 'x', '-', etc.) where every operator comes after its pair of operands:
    - <operand1> <operand2> <operator>
    - For example "3 + 2" would be represented as "3 2 + " in postfix
    - The expression "(3 – 4) + 5" with 2 operators would be "3 4 – 5 +" in postfix
- Notice that a nice feature of postfix is that the parentheses are not necessary, which makes the expressions more compact, and unambiguous
- Examples
    - Infix: (3+4)x5 postfix: 3 4 + 5 x
    - Infix: 3+(4x5) postfix: 3 4 5 x +
    - Infix: 7+(4x(6-2)) postfix: 7 4 6 2 – x +

## Postfix evaluation

- Problem statement:
    - Given a valid postfix expression with numerals and '+, '-', 'x' operators in the form of a string, evaluate it and store the answer in R5. E.g.,
        - Input: String of numbers and operators "3 4 + 5 x"
        - Output: 35
- For simplicity, let's assume that each numerical argument is a single character
- Idea: use stack
    - Push one argument (char) in string at a time onto a stack
    - If the argument is a number, then do nothing
    - Else (operator) pop last two elements from stack, perform operation and push the result back onto the stack
    - Done when input expression is completely read

- Algorithm
  - Read the string (postfix expression) left to right;
  - Push the numbers in the expression on the stack;
  - For an operator, pop the top two elements, compute the operation and push the result on stack.




- R0: address of character being read
- R2: char being read
- R1,R3: operands
- R5: Answer

- When would this implementation fail?
  - What type of expressions are "bad" for this implementation?