



00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
1C3015C0 01010100 30011100 00002020 20202E4F 52494720 20207833 3030300A E0001300 00002020 20204C45 41202052  
302C206D 794C696E 6509E200 13000000 20202020 4C454120 2052312C 206D794C 696E6540 60001600 00004C4F 4F502020  
20204C44 52205230 2C205231 2C202330 21F00010 00000020 20202020 20202054 52415020 78323105 24001400 00002020  
20202020 20204C44 20205232 2C207465 726D8014 00160000 00202020 20202020 20414444 2052322C 2052322C 20523002  
04001000 00002020 20202020 20204252 7A20354 F50612 00150000 00202020 20202020 20414444 2052312C 2052312C  
2031F90F 00120000 00202020 20202020 2042365 7A702046 4F502020 F000C00 00005354 4F502020 20204841 4C54D0FF  
00150000 00746572 6D202020 202E4649 4C4C2020 20784646 44306900 00010000 00697400 00010000 00746100 00010000  
00616200 00010000 00627200 00010000 00726100 00010000 00010000 00683200 00010000 00324000 00010000  
00406600 00010000 00666100 00010000 00613200 00010000 00323300 00010000 00332D00 00010000 002D6500 00010000  
00656300 00010000 00636500 00010000 00653200 00010000 00323200 00010000 00323000 00010000 00300000 002A0000  
006D794C 696E6520 202E5354 52494E47 5A202020 20226974 61627261 68324066 6132332D 65636532 32302200 00000000  
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

# ECE 220

## Lecture x0004

Slides based on material by: Yuting Chen, Yih-Chun Hu & Ujjal Bhowmik

# Recap

- Last time we discussed:
  - Stacks
    - Quarters vs. pancakes
    - Implementing PUSH/POP
    - Examples of use cases for stacks

Implementation differences in TOS convention

- Current top-most element
- Next available spot

A. Balanced parentheses

B. Palindrome check

C. Stack arithmetic

# Lesson objectives

- Understand and explain concepts of infix and postfix notation for arithmetic.
- Understand and explain advantage of postfix notation over infix notation. Be able to evaluate postfix expressions.
- Understand implementation of arithmetic using postfix notation and stack ADT
- Understand steps necessary to implement an RPN calculator in LC3
  - Know how to deal with overflow and underflow on the stack

# RPN or postfix arithmetic

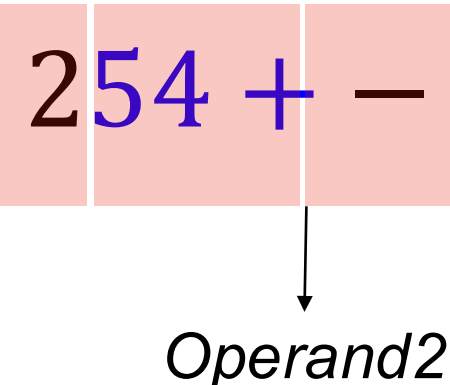
- Traditional arithmetic notation is called *infix* notation. Operations are inserted between operands. E.g.  $5 + 3$  or  $3 \times 4$ 
  - Requires use of parenthesis to indicate order of operations
- An alternative notation is called postfix notation a.k.a Reverse Polish notation (RPN). E.g.  $53 +$  or  $34 \times$ 
  - Implemented properly, does not require parenthesis/brackets

# Postfix expressions

- The syntax for a postfix operation is:

`<operand1> <operand2> <operator>`

- $(2 + 5) = 7 \Rightarrow 25 +$
- Operands may be postfix subexpressions

- $2 - (5 + 4) \Rightarrow 254 + -$   


The diagram shows the postfix expression `254+-` where the characters are contained within a light red rectangular box. The characters are arranged as follows: '2' in a light red box, '5' in a light blue box, '4' in a light blue box, '+' in a light blue box, and '-' in a light red box. A vertical line separates the '54+' part from the '-'. An arrow points from the '+' character down to the text 'Operand2'.

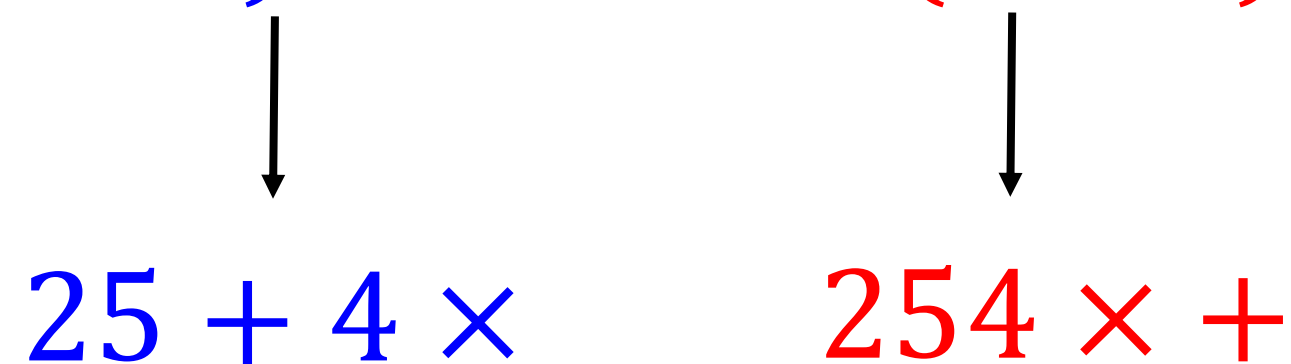
# Postfix expressions advantages

- The syntax for a postfix operation is:

`<operand1> <operand2> <operator>`

- Advantage: no need for parentheses

- $2 + 5 \times 4 \implies (2 + 5) \times 4$  or  $2 + (5 \times 4)$ ?


$$25 + 4 \times \qquad 254 \times +$$

# Practice RPN - MP2 material

- Note:  $53 \text{ --} \mapsto 5 \text{ --} 3$
- Consider:  $34 * 72 - 3 * +$ 
  - What does it evaluate to?
  - What is the *infix* version of the above?
  - Can we evaluate it using a stack?

# Example

+
x
3
-
2
7
x
4
3

$$\square = 27$$

$$\square = 15$$

$$\square = 5$$

$$\square = 12$$

POP FROM STACK

# Convert to Postfix

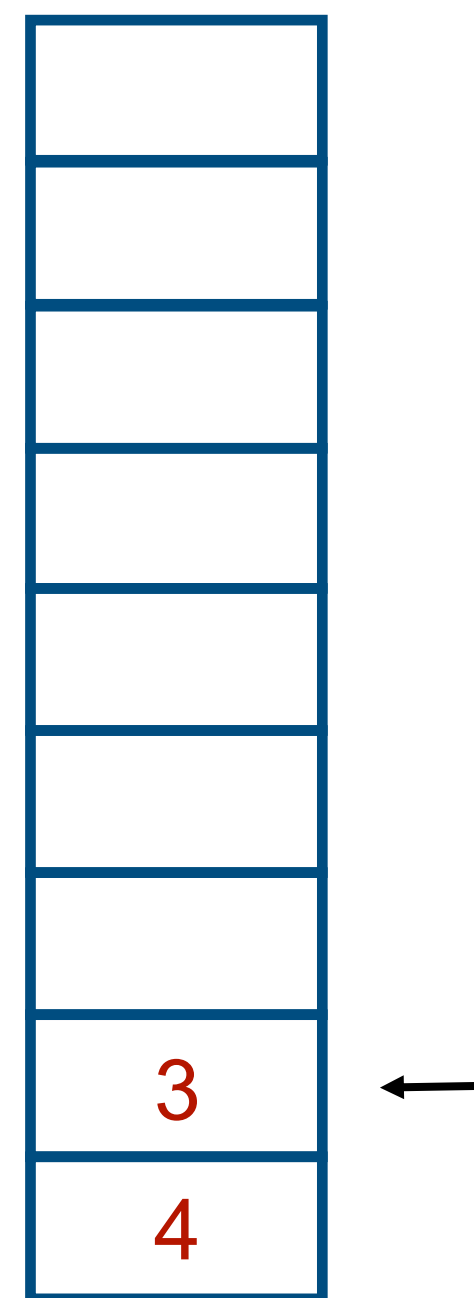
- Rewrite the following infix expressions in RPN:
  - $(8 + 4)^2$
  - $7 + (9 - 6)/3$
  - $(5 + (1 + 2) \times 4) - 3$

# Postfix expressions - evaluation

- Now evaluate them
  - $84 + 2 \wedge$
  - $796 - 3 \div +$
  - $512 + 4 \times +3 -$

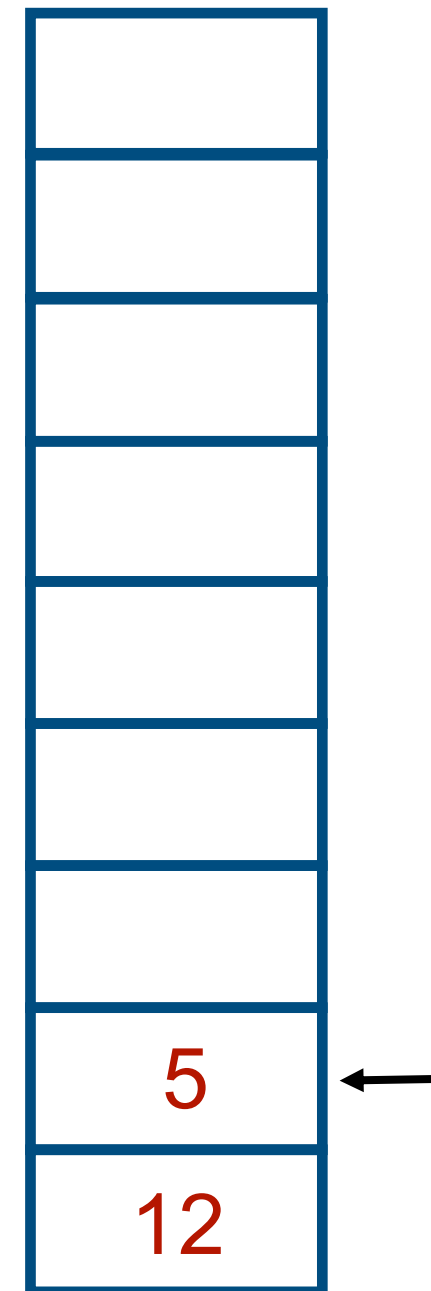
# Redo example - Initial data push

- Expression:  $43 \times 72 - 3 \times +$ 
  - Strategy:
    - Numbers  $\Rightarrow$  Push
    - Operator:
      - Pop two elements
      - Perform operation
      - Push on stack



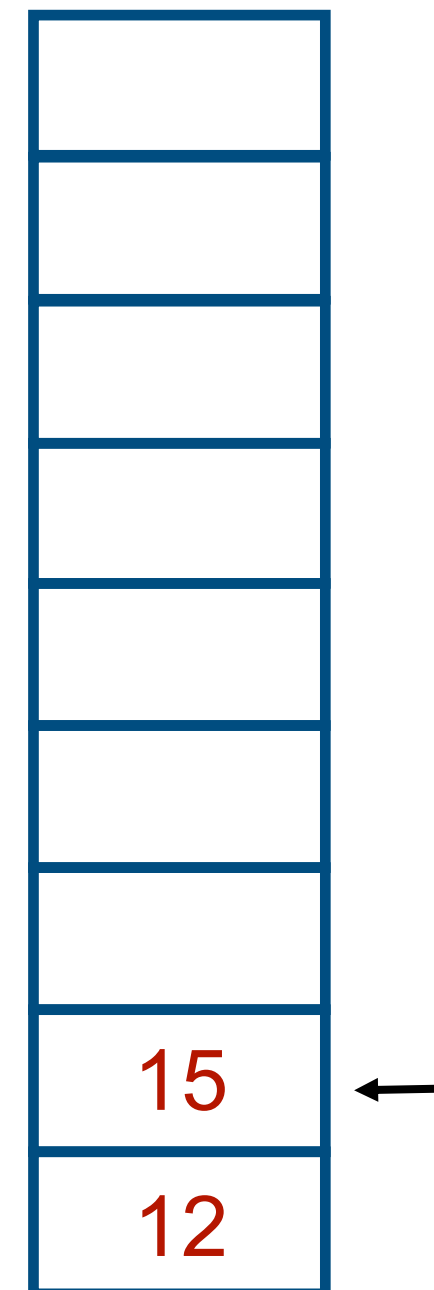
# 1<sup>st</sup> & 2<sup>nd</sup> operator evaluation

- Expression:  $43 \times 72 - 3 \times +$   
↓
- Strategy:
  - Numbers  $\Rightarrow$  Push
  - Operator:
    - Pop two elements
    - Perform operation
    - Push on stack



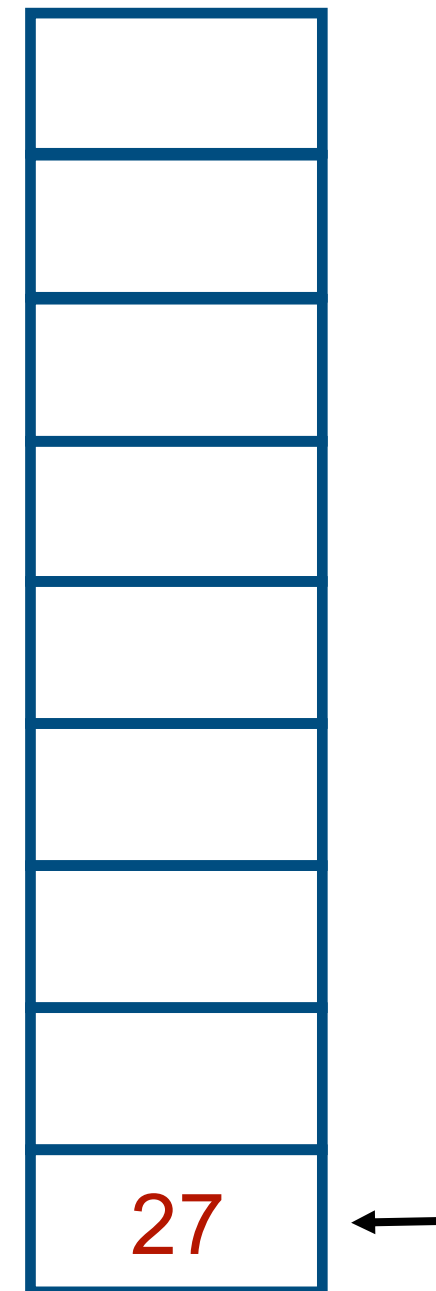
# Intermediate result storage

- Expression:  $43 \times 72 - 3 \times +$   
↓
- Strategy:
  - Numbers  $\Rightarrow$  Push
  - Operator:
    - Pop two elements
    - Perform operation
    - Push on stack



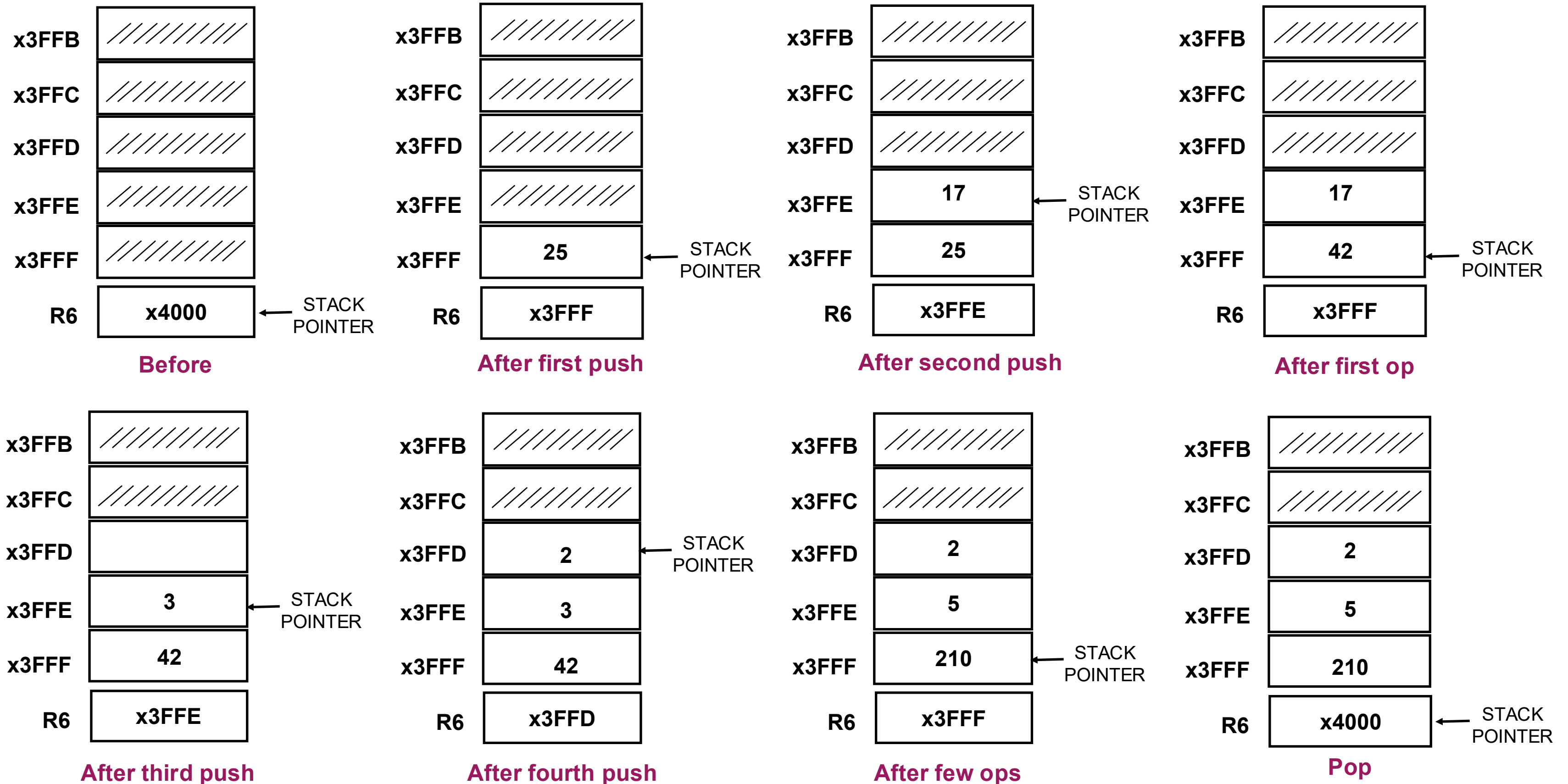
# Final operator and result

- Expression:  $43 \times 72 - 3 \times +$   
↓
- Strategy:
  - Numbers  $\Rightarrow$  Push
  - Operator:
    - Pop two elements
    - Perform operation
    - Push on stack



Given that below is an evaluation of an RPN expression: what expression is being evaluated?

# Stack usage - memory



# Arithmetic using stack - LC3

- Compute  $(A + B) \times (C + D)$  and store result in R0
- Compute  $AB + CD + X$  and store result in R0

;Implementation using registers

```
LD R0, A
LD R1, B
ADD R1, R0, R1
LD R2, C
LD R3, D
ADD R3, R2, R3
JSR MULT
HALT
```

**MULT:** subroutine such that  
Input: R1, R3 and Output: R0

**MULTIPLY:** POP two  
numbers, compute and  
then PUSH result back

;Implementation using stack

```
LD R0, A
PUSH
LD R0, B
PUSH
JSR ADD ;Assuming ADD exists
LD R0, C
PUSH
LD R0, D
PUSH
JSR ADD
JSR MULTIPLY ;Assuming MULTIPLY exists
POP ;RESULT in R0
```

**ADD:** POP two numbers,  
compute and then PUSH  
result back

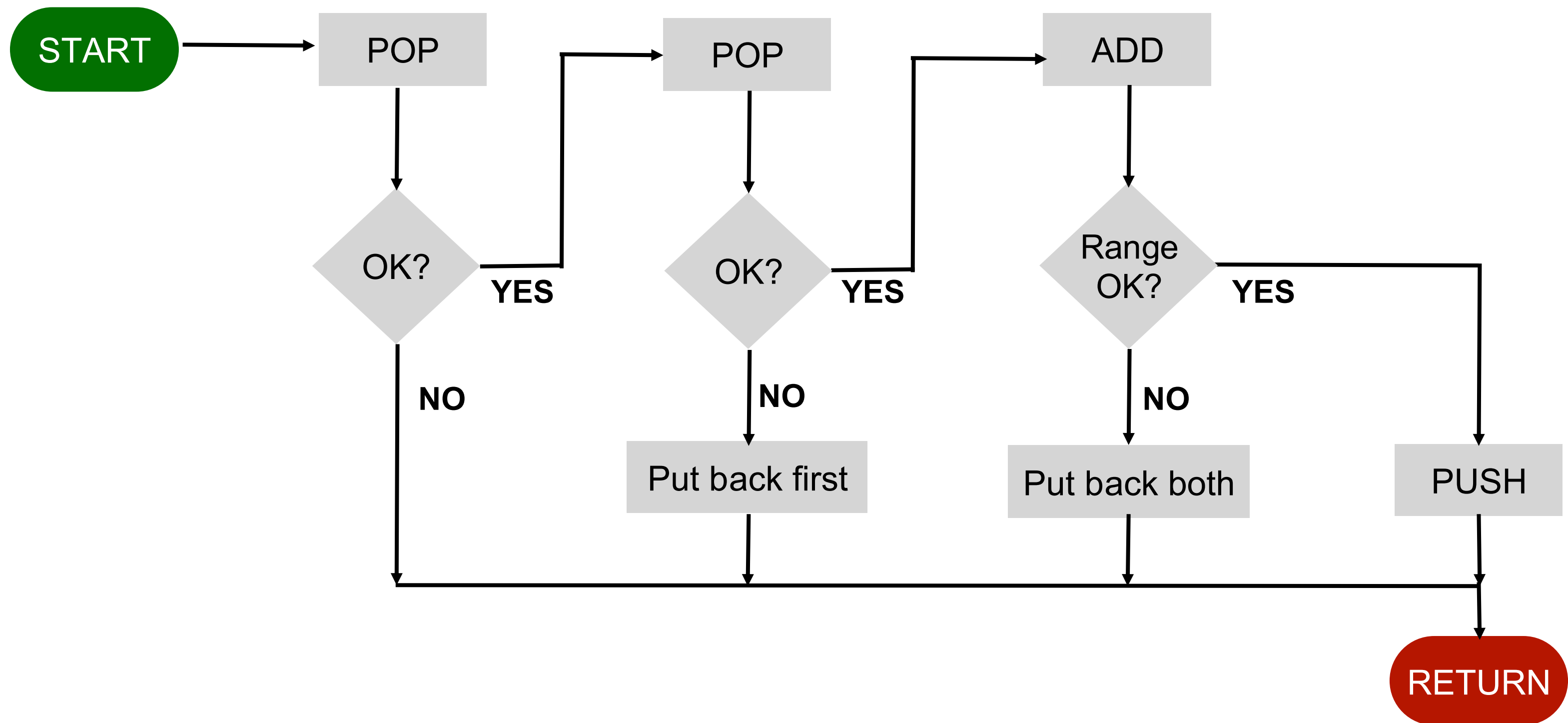
# Postfix evaluation

**Problem:** Given a postfix expression with numerals and '+', '-', '\*' in the form of a string, evaluate it and store the answer in  $\mathbb{R}^5$ . Each numeral is a single character.

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 answer and push it on the stack

# Flowchart - ADD subroutine



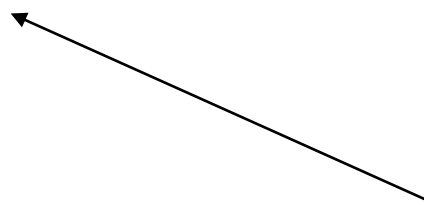
# Implement ADD subroutine

```
;PUSH
;Input: R0 (value to store on stack)
;Output: R5 (0-success, 1-fail)
```

```
;POP
;Output: R0 (value to load from stack)
;Output: R5 (0-success, 1-fail)
```

```
;CHECK_RANGE
;Input: R0 (value to be checked)
;Output: R5 (0-success, 1-fail)
```

- Save R7 before calling other subroutines.
- Save registers that will be altered in this subroutine
- R6 is stack pointer (points to the next available spot on the stack)
- Assume PUSH, POP and CHECK\_RANGE subroutines are provided to you



```
; ADD subroutine - pop two numbers from stack,  
; perform '+' operation and then push result back to  
the stack
```

```
ADD_OP  
; save registers
```

```
; initialize R5
```

```
; first pop
```

```
; check return value of first pop, go to EXIT if it  
failed (R5 = 1)
```

```
; save value in R1 before second pop
```

```
; second pop
```

```
; check result of second pop, go to RESTORE_1 if it  
failed
```

```
; add two numbers: R0 <- R0 + R1
```

```
; check range of sum, go to RESTORE_2 if it failed
```

# Code

```
; everything is good, push sum (already in R0) to  
stack  
;
```

```
RESTORE_1      ; put back first number
```

```
    ; Load STACK_TOP  
    ; Put back item  
    ; Update STACK_TOP  
    ; Go to exit
```

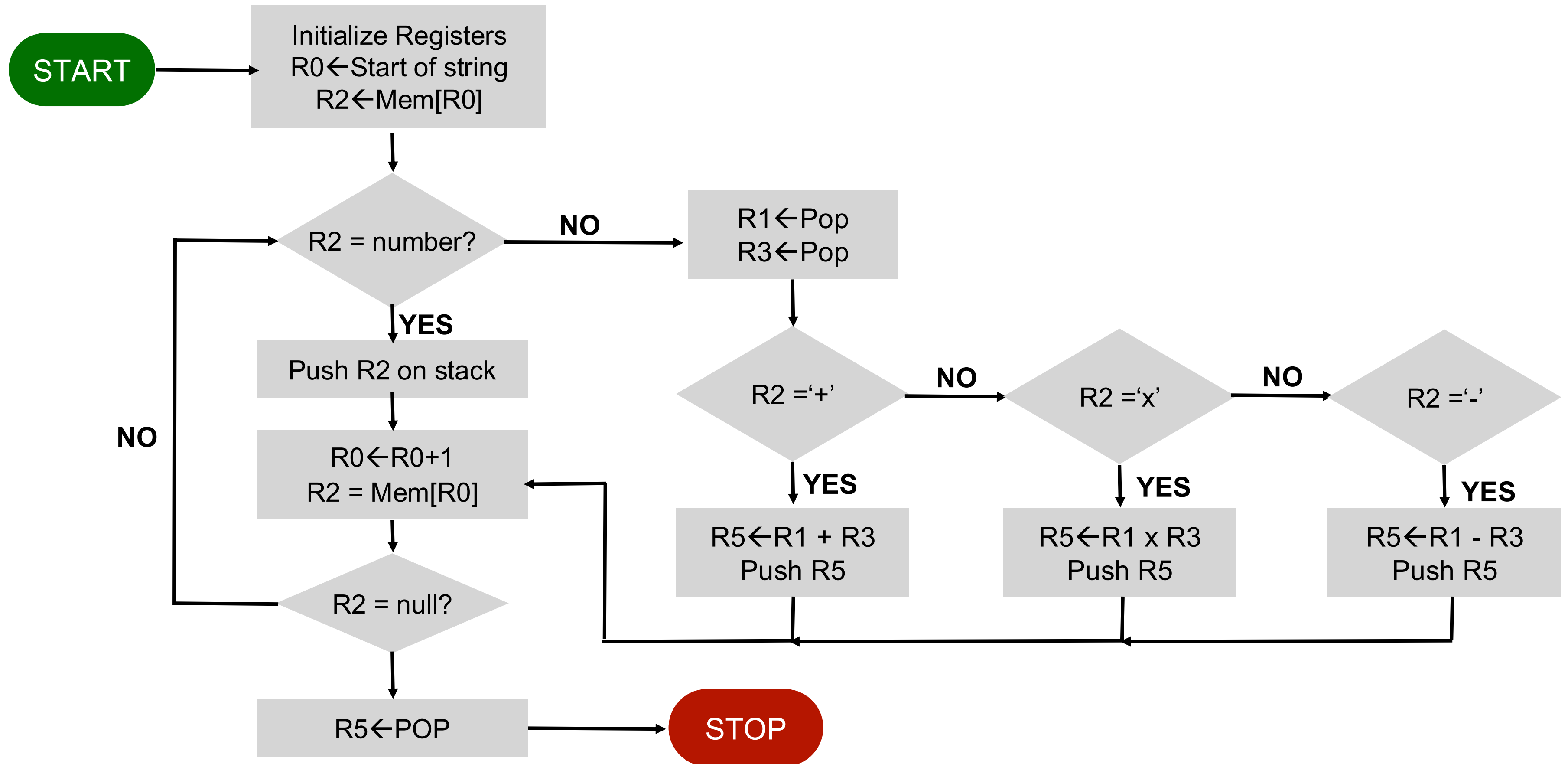
```
RESTORE_2      ; put back both numbers
```

```
    ; Load STACK_TOP  
    ; Put back item(s)  
    ; Update STACK_TOP
```

```
;  
EXIT  
; update stack top pointer  
; restore registers
```

```
RET
```

# Example decomposition



# Next time

- Introduction to C
  - Compiling a C program on EWS
  - Running the GNU debugger etc.
  - Bring your laptop!