ECE 220

Lecture x0004 - 01/29

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 3×4
  - 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. $5\ 3\ +$ or $3\ 4\ \times$
  - Implemented properly, does not require parenthesis/brackets

Postfix expressions

- The syntax for a postfix operation is:

```
<operand1> <operand2> <operator>
```

  - $(2 + 5) = 7 \implies 2\ 5\ +$

- Operands may be postfix subexpressions

  - $2 - (5 + 4) \implies 2\ 5\ 4\ +\ -$

Operand2

Postfix expressions

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

$2\ 5 + 4 \times$ $\qquad$ $2\ 5\ 4 \times +$

Practice RPN - MP2 material

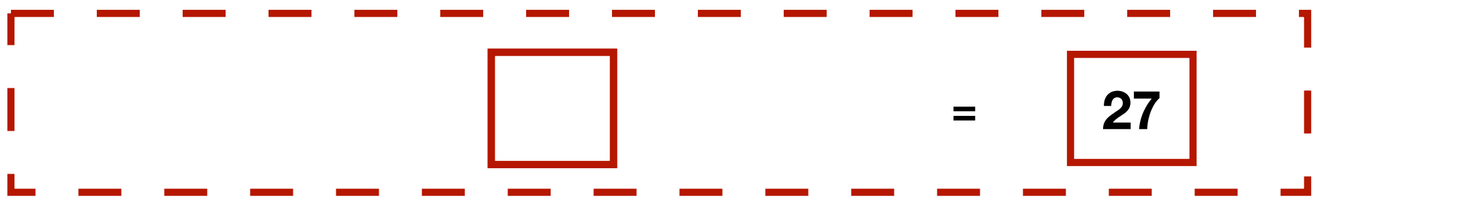
- Note: $53- \mapsto 5-3$
- Consider: $3\ 4\ *\ 7\ 2\ -\ 3\ *\ +$

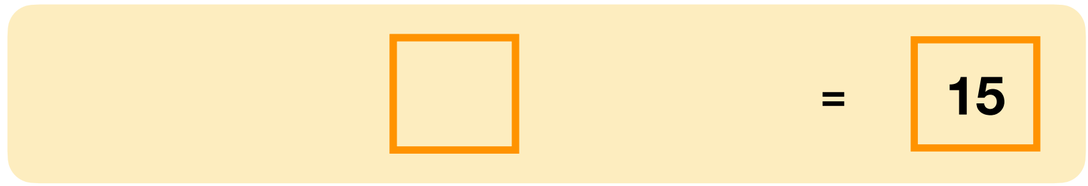
- What does it evaluate to?
- What is the *infix* version of the above?
- Can we evaluate it using a stack?

3 4 × 7 2 − 3 × +

Example

+
x
3
-
2
7
x
4
3





POP FROM STACK

Postfix expressions

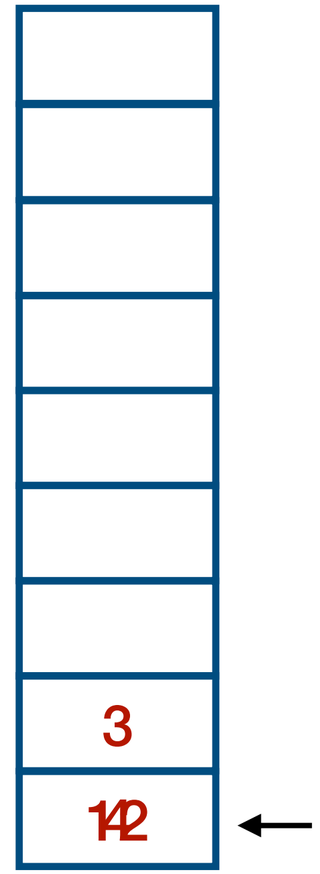
- Rewrite the following infix expressions in RPN:
  - $(8+4)^2$
  - $7+\left(9-6\right)/3$
  - $(5+(1+2)\times 4)-3$

Postfix expressions

- Now evaluate them
  - $8\ 4 + 2 \wedge$
  - $7\ 9\ 6\ -\ 3\ \div +$
  - $5\ 1\ 2 + 4 \times + 3\ -$

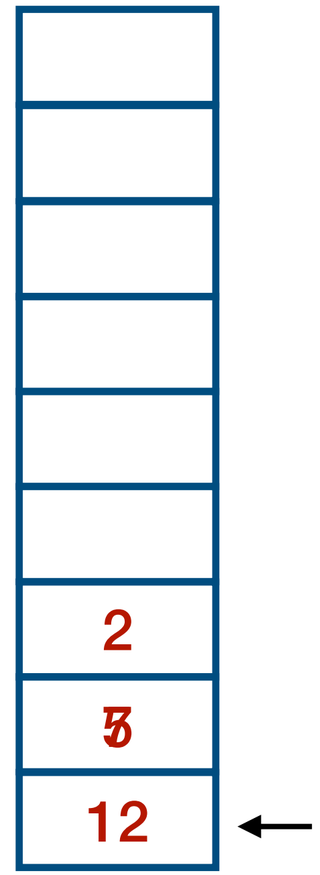
Redo example - single pass?

- Expression: $\downarrow$ $4\ 3\ \times 7\ 2\ -3\times+$
- Strategy:
  - Numbers → Push
  - Operator:
    - Pop two elements
    - Perform operation
    - Push on stack



Redo example - single pass?

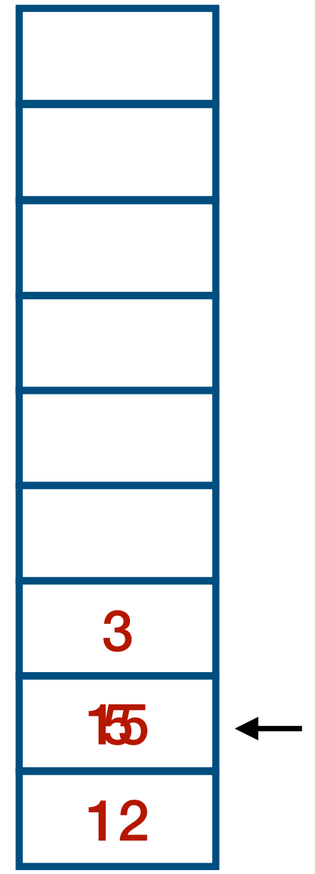
- Expression: $4\ 3\ \times 7\ 2\ -3\times+$
  ↓
- Strategy:
  - Numbers → Push
  - Operator:
    - Pop two elements
    - Perform operation
    - Push on stack



Redo example - single pass?

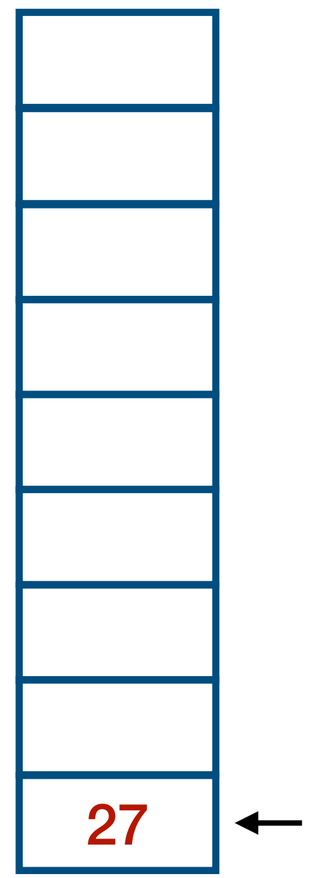
$$\downarrow$$

- Expression: $4\ 3\ \times 7\ 2\ -3\times +$
- Strategy:
  - Numbers → Push
  - Operator:
    - Pop two elements
    - Perform operation
    - Push on stack



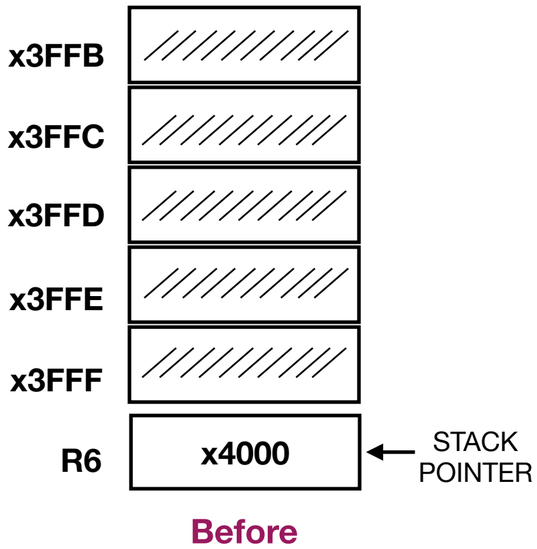
Redo example - single pass?

- Expression: $4\ 3\ \times 7\ 2\ -3\times+$
- Strategy:
  - Numbers → Push
  - Operator:
    - Pop two elements
    - Perform operation
    - Push on stack

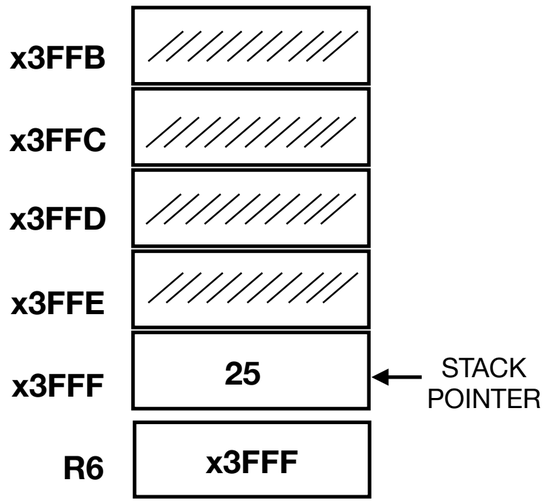


Given that below is an evaluation of an RPN expression: what expression is being evaluated?

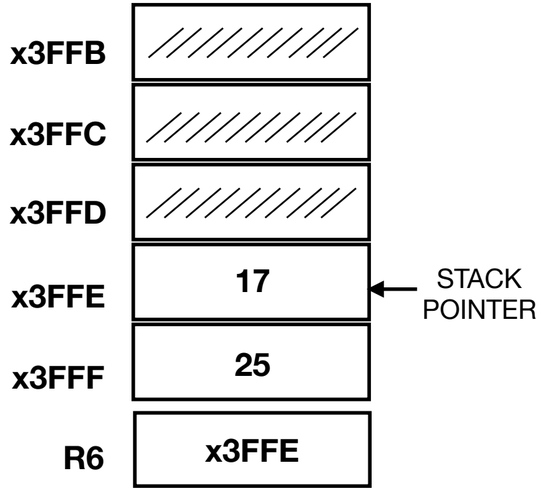
Stack usage - memory



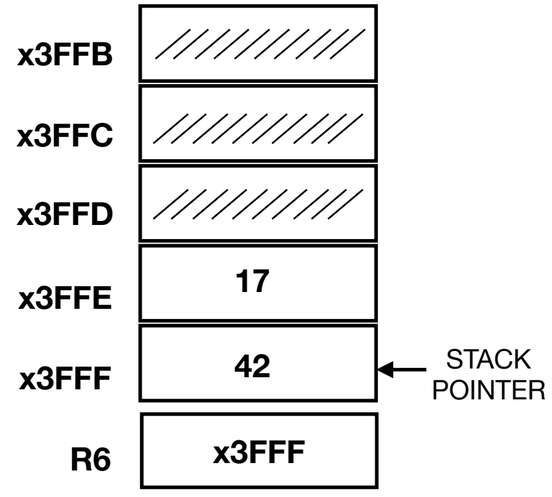
Before



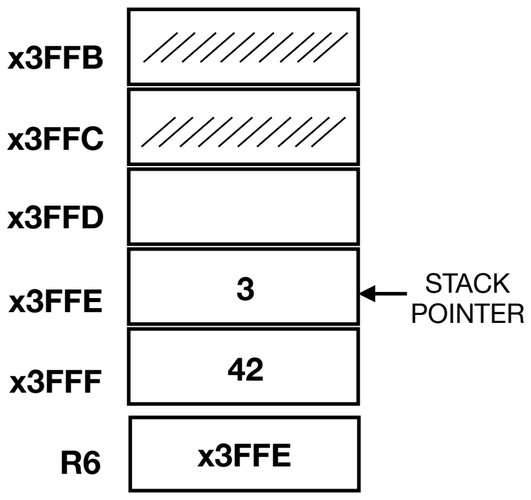
After first push



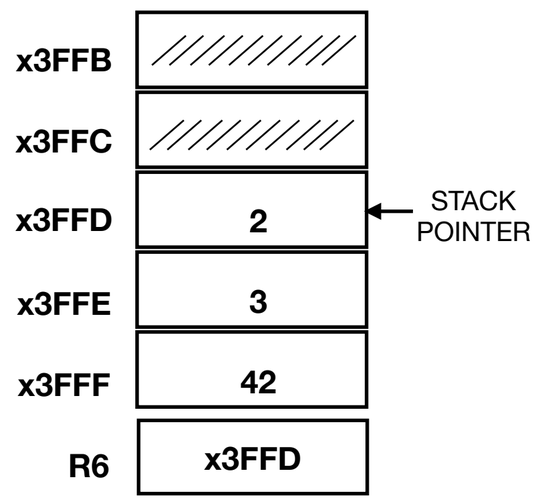
After second push



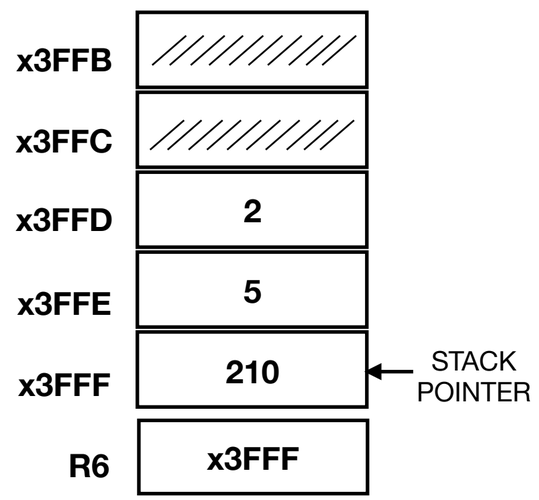
After first pop



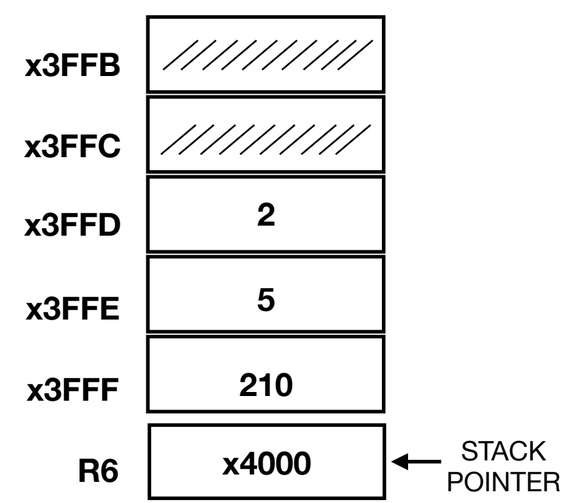
After third push



After fourth push



After 2nd & 3rd pops



Pop

Arithmetic using stack - LC3

- Compute $(A + B) \times (C + D)$ 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

- Compute $A\ B + C\ D + \times$ and store result in R0

```
;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

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

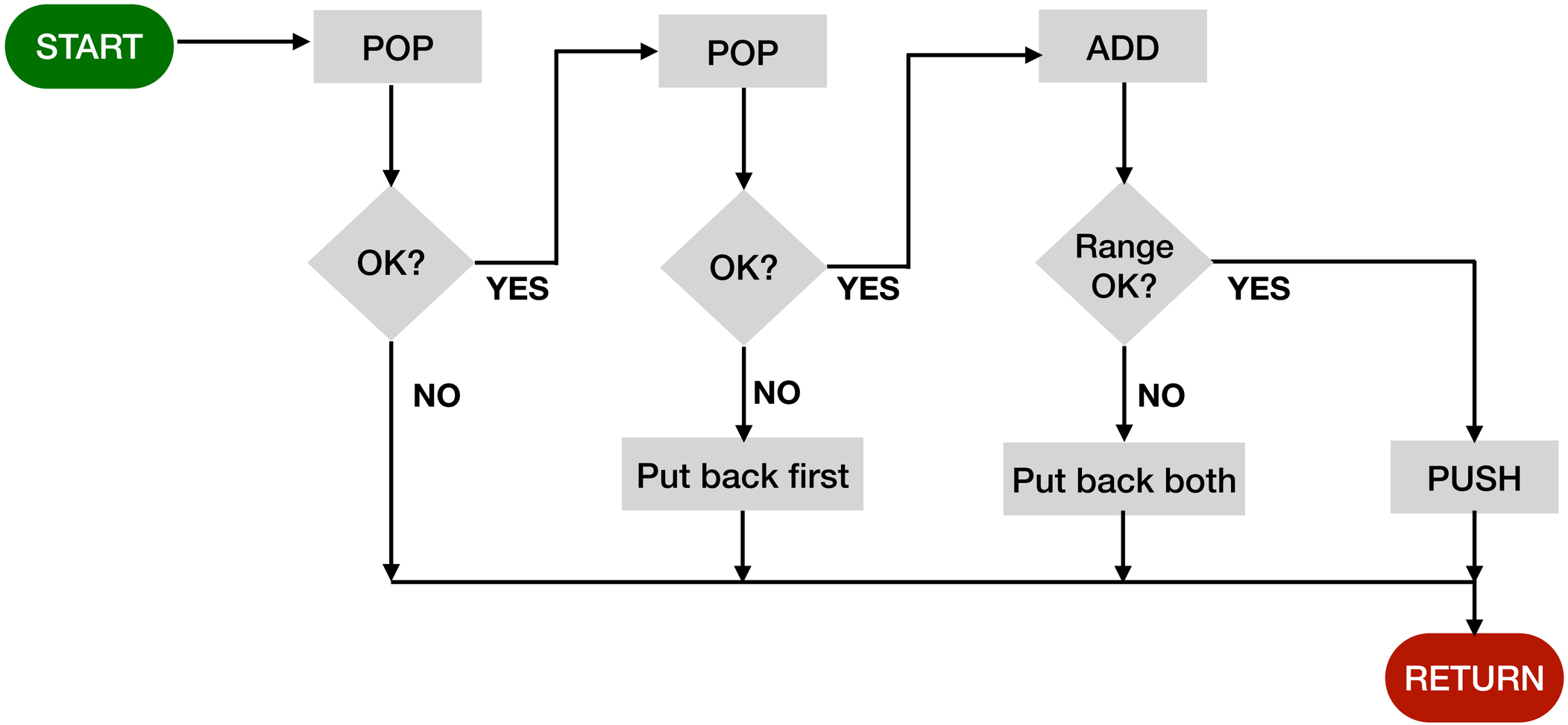
Postfix evaluation

Problem: Given a postfix expression with numerals and ‘+’, ‘-’, ‘*’ in the form of a string, evaluate it and store the answer in `R5`. 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



Required reading: Section 10.2 and 10. 3 of Patt & Patel

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




; 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
```

Next time

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