ECE 220 Computer Systems & Programming

Lecture 4: Programming with Stack

September 04, 2025

ECE ILLINOIS

ILLINOIS

- MP1 due Tonight (09/04) by 10pm

- CBTF mock quiz next week (09/08 - 09/10)
    - reserve your slot with prairieTest

- MP2 will be released tonight

Previous Lecture

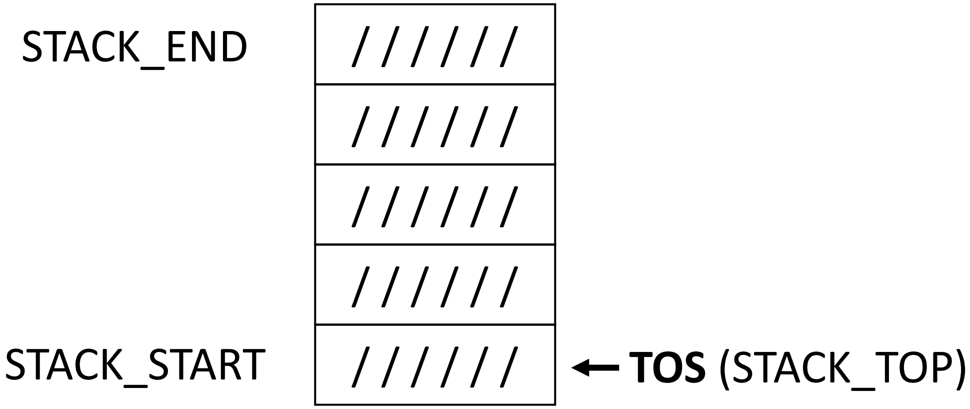
- Stack operation

  PUSH

  POP

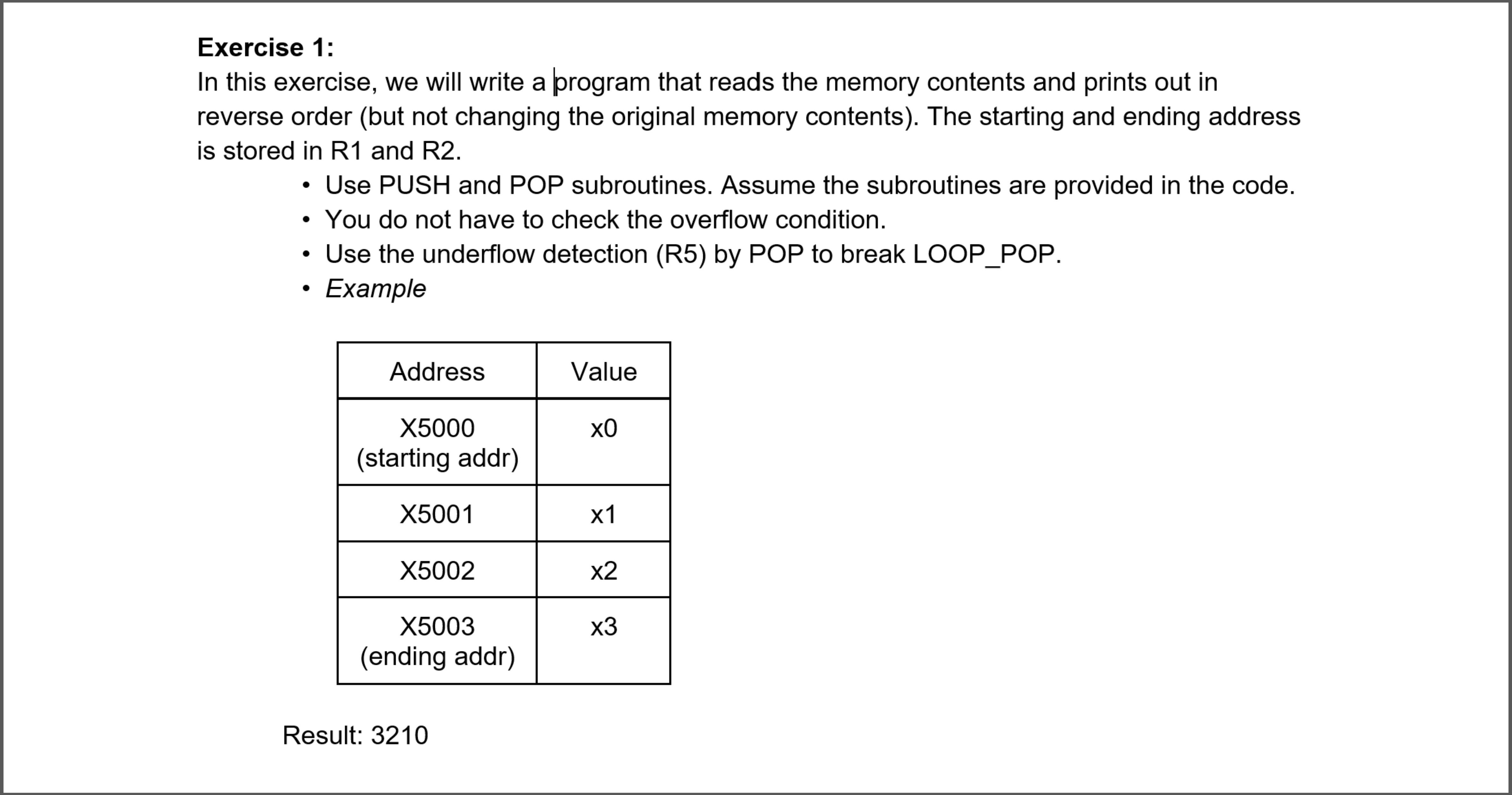
  Overflow detection

  Underflow detection



```
;PUSH subroutine
;IN: R0 (value)
;OUT: R5 (0-success, 1-fail)
```

```
;POP subroutine
;IN: none
;OUT: R0 (value)
;OUT: R5 (0-success, 1-fail)
```



Exercise 1:

In this exercise, we will write a program that reads the memory contents and prints out in reverse order (but not changing the original memory contents). The starting and ending address is stored in R1 and R2.

- Use PUSH and POP subroutines. Assume the subroutines are provided in the code.
- You do not have to check the overflow condition.
- Use the underflow detection (R5) by POP to break LOOP_POP.
- *Example*

Address	Value
X5000 (starting addr)	x0
X5001	x1
X5002	x2
X5003 (ending addr)	x3

Result: 3210

Caller-save vs Callee-save

```
.ORIG   x3000
; R0, R5, R7 have some important values that will be needed later
; …….

JSR     POP ; R7 saves PC

; want to keep original R0, R5, R7 after POP
```

Q. How will you save R0, R5, R7?

```
;POP subroutine
;IN: none
;OUT: R0 (value)
;OUT: R5 (0-success, 1-fail)

; save R0 and R5 here

R0 <- stack data
R5 <- flag

; restore R0 and R5

RET
```

Caller-save vs Callee-save

```
ORIG   x3000
; R0, R5, R7 have some important values that will be needed later
; …….
ST      R0, Save_R0
ST      R5, Save_R5
ST      R7, Save_R7

JSR     POP
; process R0 and R5, then restore


LD      R0, Save_R0
LD      R5, Save_R5
LD      R7, Save_R7
```

```
;POP subroutine
;IN: none
;OUT: R0 (value)
;OUT: R5 (0-success, 1-fail)
```

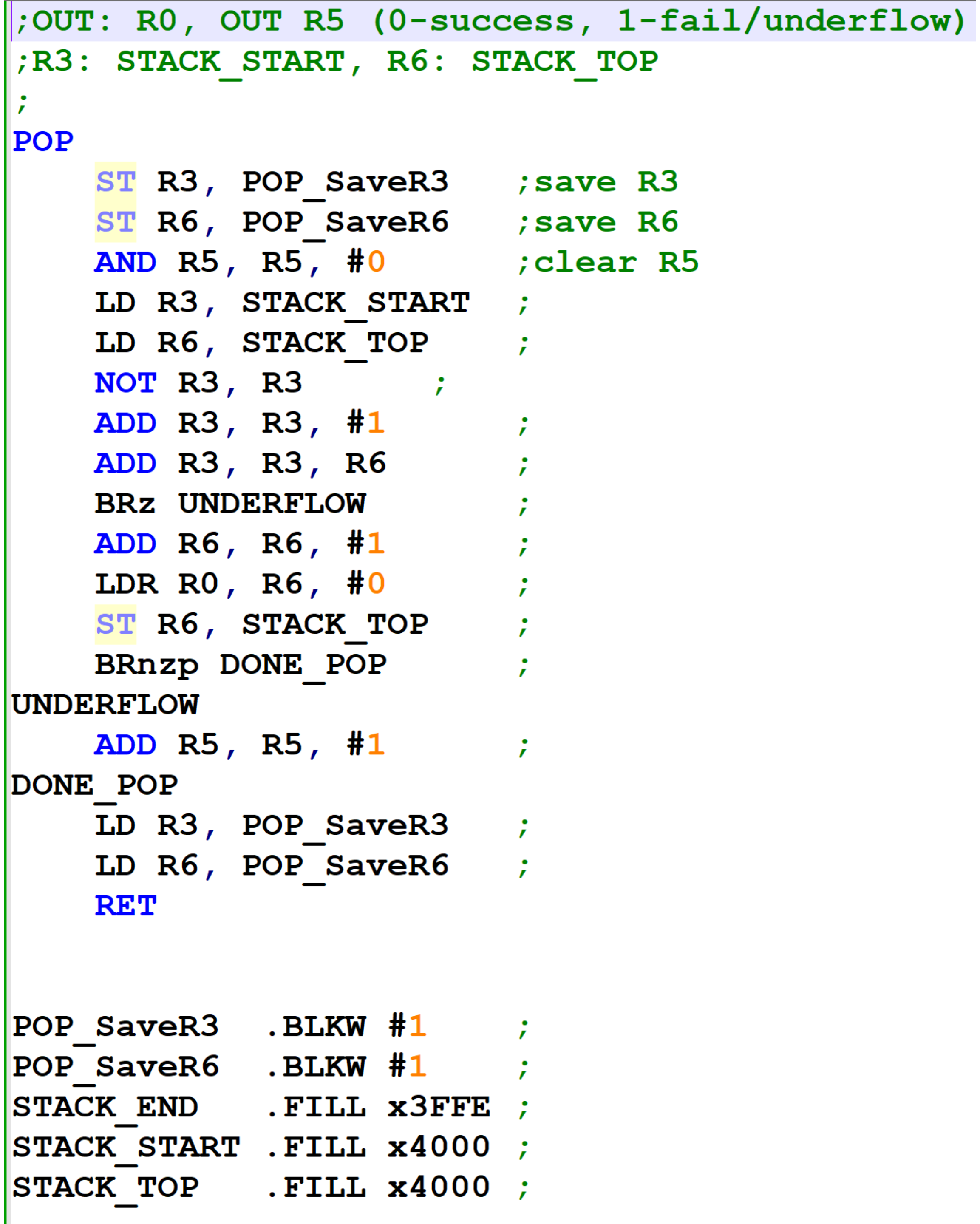
Caller-save

Caller-save vs Callee-save

R3 and R6 are saved and restored.

Is it callee-save or caller save?

Caller may not know the implementation details of the implementation of stack.
It only knows the input/output arguments



```
;OUT: R0, OUT R5 (0-success, 1-fail/underflow)
;R3: STACK_START, R6: STACK_TOP
;
POP
    ST R3, POP_SaveR3     ;save R3
    ST R6, POP_SaveR6     ;save R6
    AND R5, R5, #0        ;clear R5
    LD R3, STACK_START    ;
    LD R6, STACK_TOP      ;
    NOT R3, R3        ;
    ADD R3, R3, #1        ;
    ADD R3, R3, R6        ;
    BRz UNDERFLOW         ;
    ADD R6, R6, #1        ;
    LDR R0, R6, #0        ;
    ST R6, STACK_TOP      ;
    BRnzp DONE_POP        ;
UNDERFLOW
    ADD R5, R5, #1        ;
DONE_POP
    LD R3, POP_SaveR3     ;
    LD R6, POP_SaveR6     ;
    RET


POP_SaveR3   .BLKW #1     ;
POP_SaveR6   .BLKW #1     ;
STACK_END    .FILL x3FFE  ;
STACK_START  .FILL x4000  ;
STACK_TOP    .FILL x4000  ;
```

Using Stack convention in calling suboutine

Saving program state when serving interrupt-driven IO

PC and PSR saved in supervisor stack (discussed later)

Saving and restoring registers when calling a subroutine

- Stack enables subroutines to be re-entrant
    - It can be interrupted and then safely resume its operation.
    - It can call other subroutines including itself (recursive)
    - Part of the foundation for multi-threading

Some applications: calculator, checking balanced parentheses, etc. (related to MP2)

Programming with Stack

- Most calculators use a stack to store operands and results of the calculation
  - Recall from LC-3’s ISA that ADD instruction requires 3 operands
    - “ADD DR, SR1, SR2”
    - All 3 locations of the operands are explicitly identified
- Many calculators are implemented in a way that none of the operands need to be explicitly identified
    - Operands are pushed into the stack
    - “ADD” is sufficient
    - To perform it, two values are popped off the stack, added, and the result is pushed back onto the stack
  - Example: E = (A + B) * (C + D)

Example: Arithmetic Calculator Using a Stack

- Example: E = (A+B)*(C+D)

```
;LC-3 implementation
LD     R0, A
LD     R1, B
ADD    R1, R0, R1
LD     R2, C
LD     R3, D
ADD    R3, R2, R3
JSR    MULT

;MULT subroutine
;IN: R1,R3
;OUT: R0
```

```
;Stack-based implementation
PUSH  ;A
PUSH  ;B
ADD
PUSH  ;C
PUSH  ;D
ADD
MULT
POP   ;E

;ADD- POP 2 numbers, compute and then
;PUSH result back
;MULT- POP 2 numbers, compute and then
;PUSH result back
```

Arithmetic Using Stack

Implement a multiplication subroutine (MUL) that pops two numbers from a stack and perform the multiplication operation and put the result back into the stack.

Recall:

```
; multiply R0 = R1*R2
  AND R0, R0, #0
  LOOP ADD R0, R0, R1 ;
  ADD R2, R2, #-1
  BRp LOOP
```

```
.ORIG x3000
; R1 <- a
; R2 <- b

; prepare arguments
   AND R0, R0, #0
   ADD R1, R0, #5 ; R1 <- 5
   ADD R2, R0, #7 ; R2 <- 7

; save R0
   ST R0, MAIN_SaveR0 ;

; push arguments
   ADD R0, R1, #0
   JSR PUSH
   ADD R0, R2, #0
   JSR PUSH
```

```
; call subroutine
   JSR MULT ; stack <- result

; consume result
   JSR POP
   ADD R5, R0, #0

; restore R0
   LD R0, MAIN_SaveR0 ;

; continue
HALT

; main's data
MAIN_SaveR0 .BLKW #1
```

```
; MULT multiplies two positive numbers
; IN: stack
; OUT: val in stack <- (val1 from stack*
                        val2 from stack)
; R1, R2: val1, val2

MULT
  ST R2, MULT_SaveR2
  ST R7, MULT_SaveR7

; get operands from the stack
  JSR POP
  ADD R2, R0, #0
  JSR POP
  ADD R1, R0, #0

; multiply
  AND R0, R0, #0
  LOOP ADD R0, R0, R1 ;
  ADD R2, R2, #-1
  BRp LOOP

; put result onto the stack
  JSR PUSH


   LD R2, MULT_SaveR2
   LD R7, MULT_SaveR7

RET

; data
MULT_SaveR2 .BLKW #1
MULT_SaveR7 .BLKW #1
```

Lab2 Review

- Balanced parentheses: each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested.

Which are “balanced parenthesis”?

1. (()()()())
2.)))(((
3. ((((((()))
4. (((())))

How do you check Balanced Parentheses?

Examples of balanced parentheses:

- (()()()())   (((())))   (()((())()))

Examples of unbalanced parentheses:

- ((((((())   ()))   )))(((

Use Stack

- Open parenthesis ‘(‘ – PUSH to the stack
- Close parenthesis ‘)‘ – POP from the stack

Assuming the expression would fit into the stack, unbalanced expression can be found under two situations:

1. At the end of the expression – Stack is not **EMPTY**
2. While entering expression – Stack detects **UNDERFLOW**

MP2 Preview: Postfix Expression

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

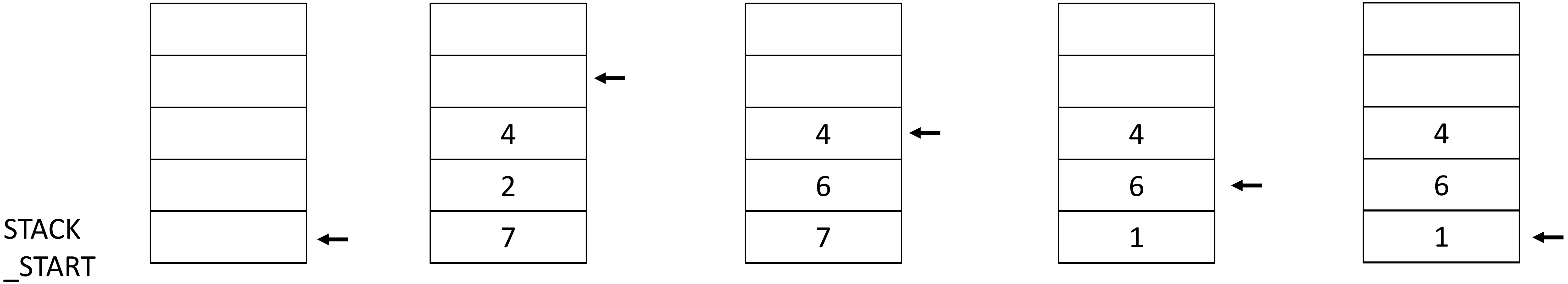
Examples

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 +

How about: 3 1 / + 3 =

Valid Post Expression & Stack

7 2 4 + - =



Empty

After 3 numbers

push 7
push 2
push 4

After +

pop 4
pop 2
push 2+4=6

After -

pop 6
pop 7
push 7-6=1

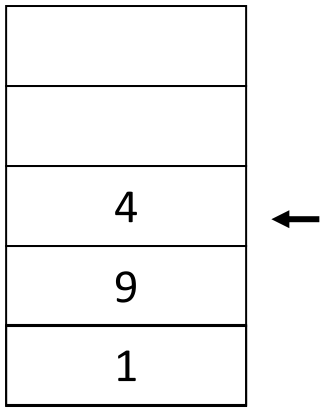
After =

pop 1
Result : 1

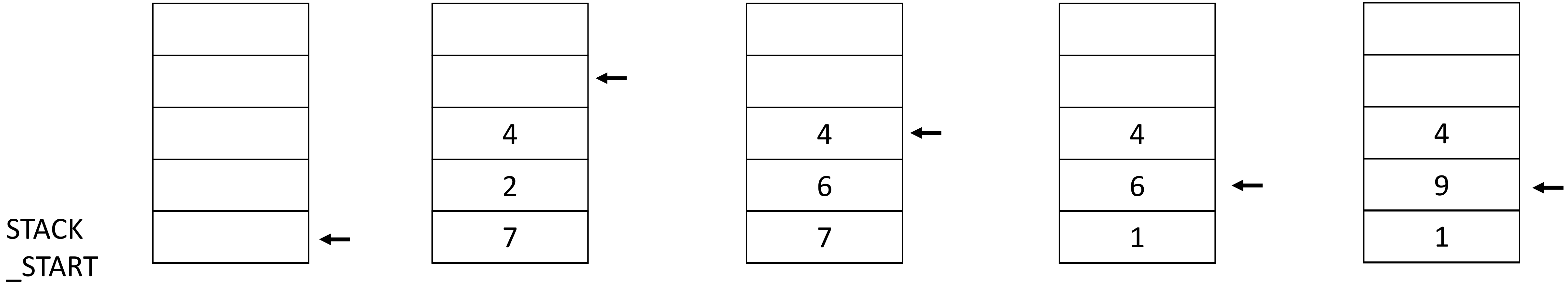
Invalid Post Expression & Stack

~~7 2 4 + - =~~

What if
7 2 4 + - 9 =



After 9



Empty	After 3 numbers	After +	After -	After =
	push 7 push 2 push 4	pop 4 pop 2 push 2+4=6	pop 6 pop 7 push 7-6=1	pop 9 Result: 9

MP2 - Part1: Postfix Expression & Stack

Unbalanced-case1

(Underflow while taking actions for an operator)

Unbalanced-case2

How do we know?→ (Stack has more than one number before ‘=‘)

If you meet ‘=‘ , do 2 POPs

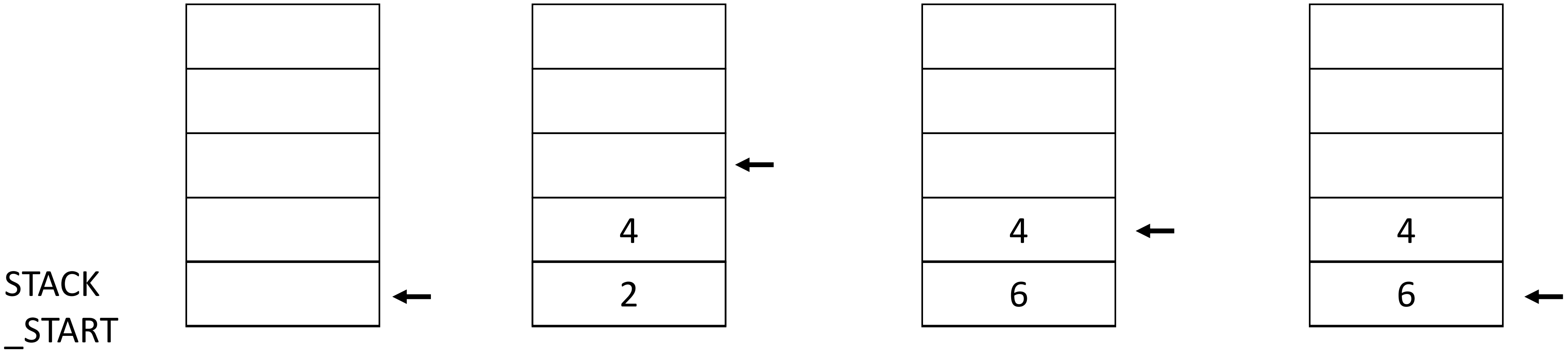
- first POP to grab the result
- second POP to check it’s empty

  → If underflow, valid

  → If not, invalid

Invalid Post Expression & Stack

2 4 + - =



Empty

After 2 numbers

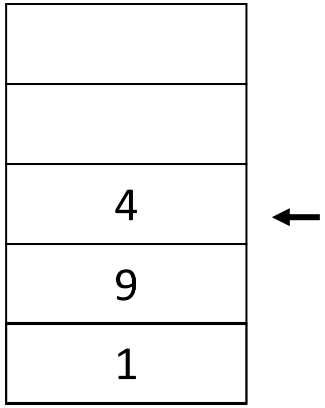
After +

Underflow
while -

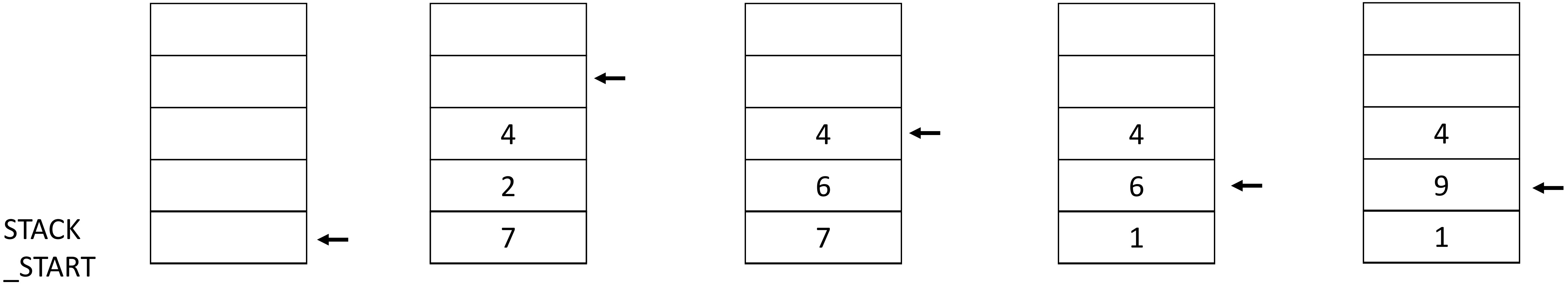
Invalid Post Expression & Stack

~~7 2 4 + - =~~

What if
7 2 4 + - 9 =



After 9



Empty

After 3 numbers

push 7
push 2
push 4

After +

pop 4
pop 2
push 2+4=6

After -

pop 6
pop 7
push 7-6=1

After =

pop 9
Result: 9