# ECE 220: Computer Systems & Programming

## Lecture 5: Programming with Stack

Thomas Moon

January 30, 2024

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

- MP2 due next Thursday.

# Previous lecture

- Stack!

# Today’s lecture

- How/When/Why to use Stack?

# Previous Lecture

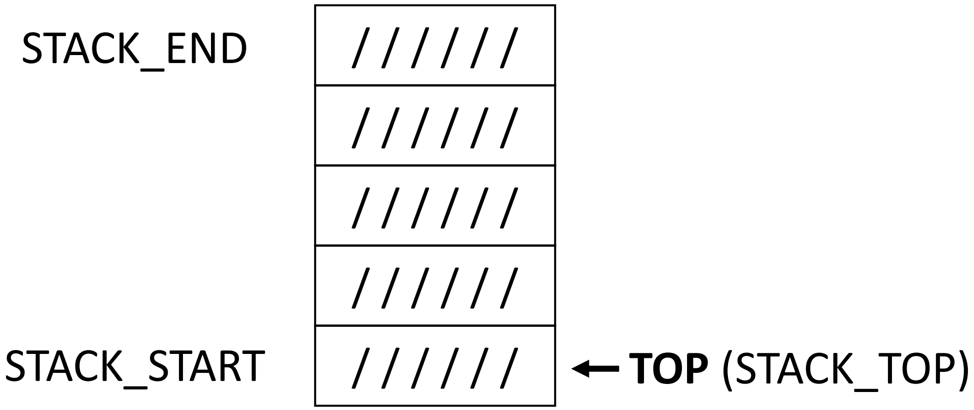
- Stack operation

  PUSH

  POP

  Overflow detection

  Underflow detection



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

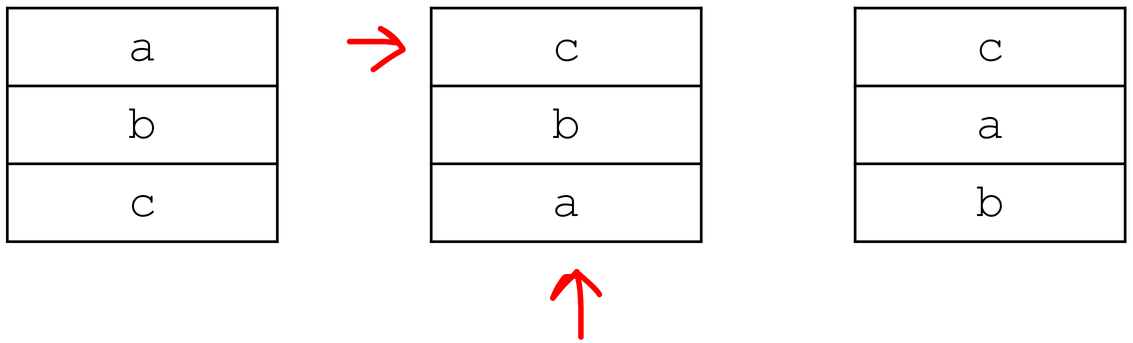
```
;POP subroutine
;IN: none
;OUT: R0 (value)
;OUT: R5 (0-success, 1-fail)
```

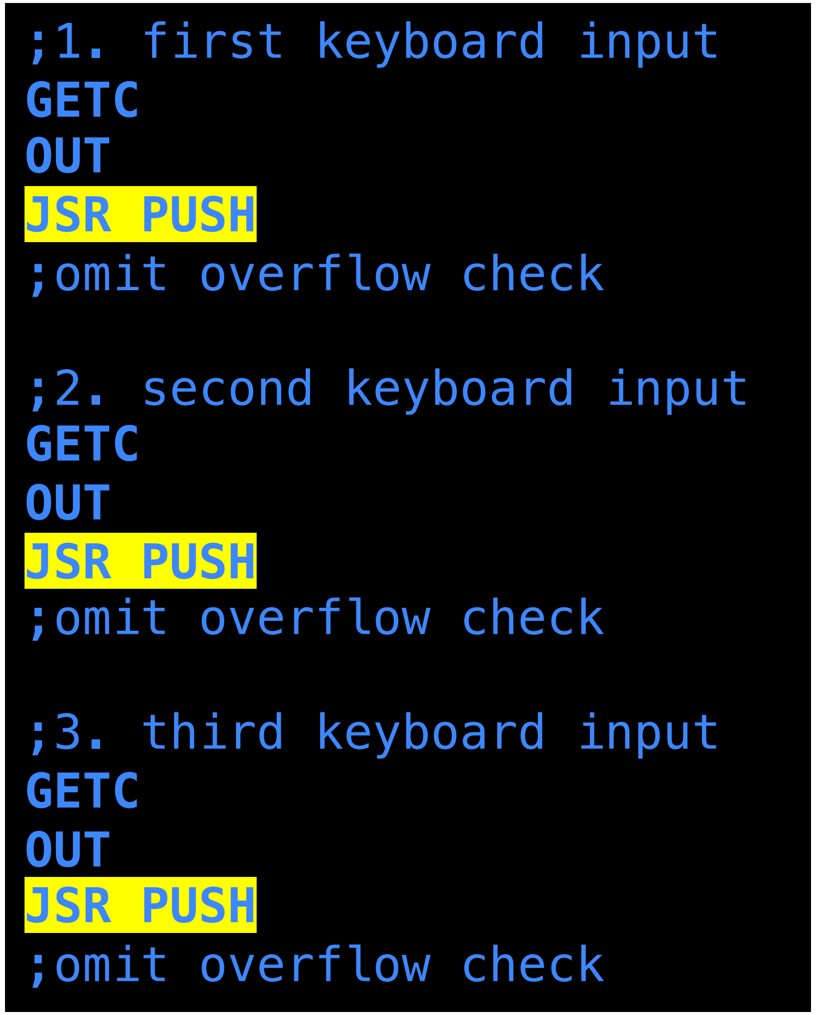
```
;1. first keyboard input
GETC
OUT
JSR PUSH
;omit overflow check

;2. second keyboard input
GETC
OUT
JSR PUSH
;omit overflow check

;3. third keyboard input
GETC
OUT
JSR PUSH
;omit overflow check
```

Q. If we type

abc

, how does the stack look like?





```
;1. first keyboard input
GETC
OUT
JSR PUSH
;omit overflow check

;2. second keyboard input
GETC
OUT
JSR PUSH
;omit overflow check

;3. third keyboard input
GETC
OUT
JSR PUSH
;omit overflow check
```

Q. After three PUSH, how to read the top data?

1. Use one of LD family to access the memory location.

[✓ 2.] JSR POP

**In STACK,**
**PUSH to write, POP to read.**

```
;1. first keyboard input
GETC
OUT
JSR PUSH
;omit overflow check

;2. second keyboard input
GETC
OUT
JSR PUSH
;omit overflow check

;3. third keyboard input
GETC
OUT
JSR PUSH
;omit overflow check
```

```
JSR POP
;omit underflow check
OUT


JSR POP
;omit underflow check
OUT


JSR POP
;omit underflow check
OUT
```

```
abccba
```

# Recap : Caller-save vs Callee-save

```
.ORIG   x3000
; do something important for R0, R5, R7

JSR     POP ; R7 saves PC

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

```
;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
```

Q. Which is the correct way to save R0, R5, R7?

A. Caller-save R7, and Callee-save R0 and R5

B. Caller-save R0 and R5, and Callee-save R7

[✓ C. Caller-save R0, R5, and R7]

D. Callee-save R0, R5, and R7

E. Either Caller-save or Callee-save works

# Recap : Caller-save vs Callee-save

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

```
.ORIG   x3000
; do something important for R0, R5, R7
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
```

Caller-save

# Recap : Caller-save vs Callee-save

```
;POP subroutine
;IN: none
;OUT: R0 (value)
;OUT: R5 (0: success, 1: fail)
;R3: STACK_START
;R6: STACK_TOP
POP
```

+ R7

Q. How many registers will be updated by calling POP?

A. 0

B. 1

C. 2

D. 3

E. 4

[✓ F. 5]

Q. How many registers should be saved/restored in POP?

A. 0

B. 1

[✓ C. 2] R3, 6

D. 3

E. 4

# Recap : Caller-save vs Callee-save

```
;POP subroutine
;IN: none
;OUT: R0 (value)
;OUT: R5 (0: success, 1: fail)
;R3: STACK_START
;R6: STACK_TOP
POP
```

Q. Which is the correct way to save R3 and R6?

A. Caller-save R3, and Callee-save R6
B. Caller-save R6, and Callee-save R3
C. Caller-save R3 and R6
[✓ D.] Callee-save R3 and R6
E. Either Caller-save or Callee-save works

# Recap : Caller-save vs Callee-save

```
;POP subroutine
;IN: none
;OUT: R0 (value)
;OUT: R5 (0: success, 1: fail)
;R3: STACK_START
;R6: STACK_TOP
POP
;callee-save & initialize registers
        ST      R3,SaveR3
        ST      R6,SaveR6



;     code omitted



        LD      R3,SaveR3
        LD      R6,SaveR6
        RET
```

Callee-save

# Using Stack…

**Saving program state** when serving interrupt-driven IO (Lecture 26?)

- PC and PSR saved in supervisor stack

**Saving and restoring registers** when calling a subroutine (Lecture 10,14)

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

Many other applications such as calculator, checking balanced parentheses, etc.

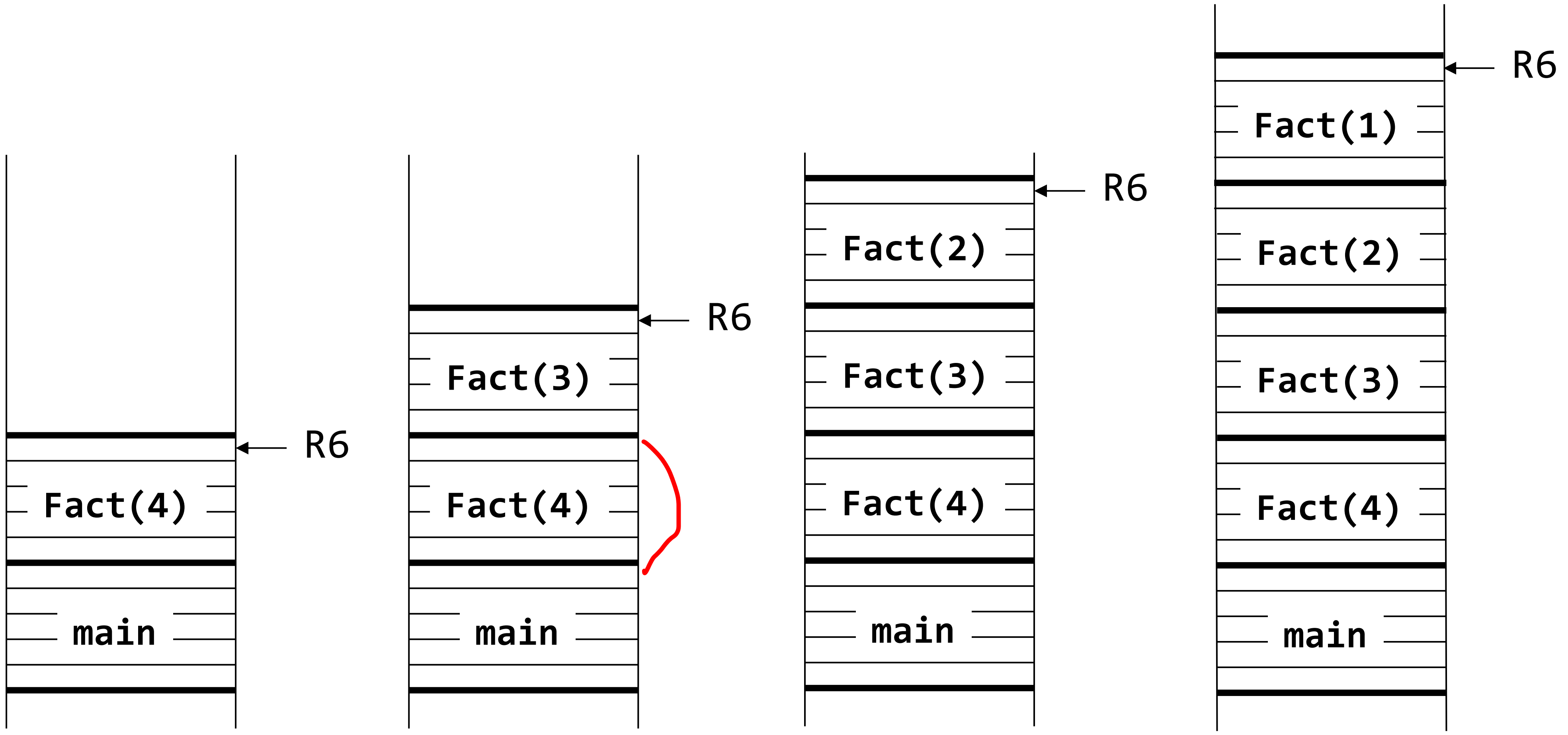
# Spoiler Alert – Lec14
# Run-time "Stack"

main calls Factorial(4)

Factorial(4) calls Factorial(3)

Factorial(3) calls Factorial(2)

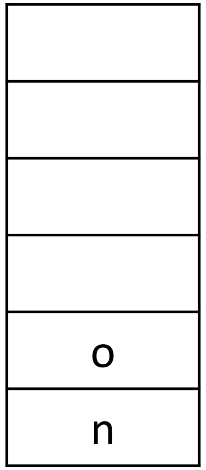
Factorial(2) calls Factorial(1)



# Palindrome Check Using a Stack

- A word, phrase, number or other sequence of characters which reads the same forward or backward.
- Examples
  - madam
  - noon
  - racecar

noon



o

n

- How can we test for palindromes using a stack?

## Lab2 Review

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

Q. Which one is “unbalanced parenthesis”?

1. (()()()())
[✓ 2.] )))(((
[✓ 3.] ((((((())
4. (((())))

# Lab2 Review: Check Balanced Parentheses Using a Stack

Examples of balanced parentheses:

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

Examples of unbalanced parentheses:

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

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

For this problem,
we only care **the status of the stack**, not the data.

# How to Detect Unbalanced Expression

Case 1. ())(

More CLOSE than OPEN

More POP than PUSH

→Stack UNDERFLOW detected while inputting expression

Case 2. ((((((((((((()))

# CLOSE < #OPEN

# POP < #PUSH

→Stack is NOT EMPTY at the end

but, how do we know a stack is EMPTY?

→One more dummy POP will tell

# Example: Arithmetic Calculator Using a Stack

R1

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

```
;LC-3 implementation
;(three-address machine)
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
;(zero-address machine)
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
```

# MP2 Preview: Postfix Expression

- Postfix expression: a sequence of numbers and operators where every operator comes after its pair of operands
  - **Infix:** **<operand1> <operator> <operand2>**
  - **Postfix:** **<operand1> <operand2> <operator>**

  ex) 3 + 2 → 3 2 +
- ‘=‘ (equal sign) character ends the expression

**Example**

| Infix | Postfix |
|---|---|
| (3+4)*5 = | 3 4 + 5 * = |
| 7 + (9-6)/3= | 7 9 6 – 3 / + = |
| 2 - (1/2) = | |
| invalid | 4 6 * - = |
| invalid | 1 3 + 5 7 = |

# MP2 - Part1: Postfix Expression & Stack

Number = 1 PUSH

Operator = 2 POPs → Calculate → 1 PUSH

( = 1 POP)

Unbalanced-case1

(Underflow while inputting)

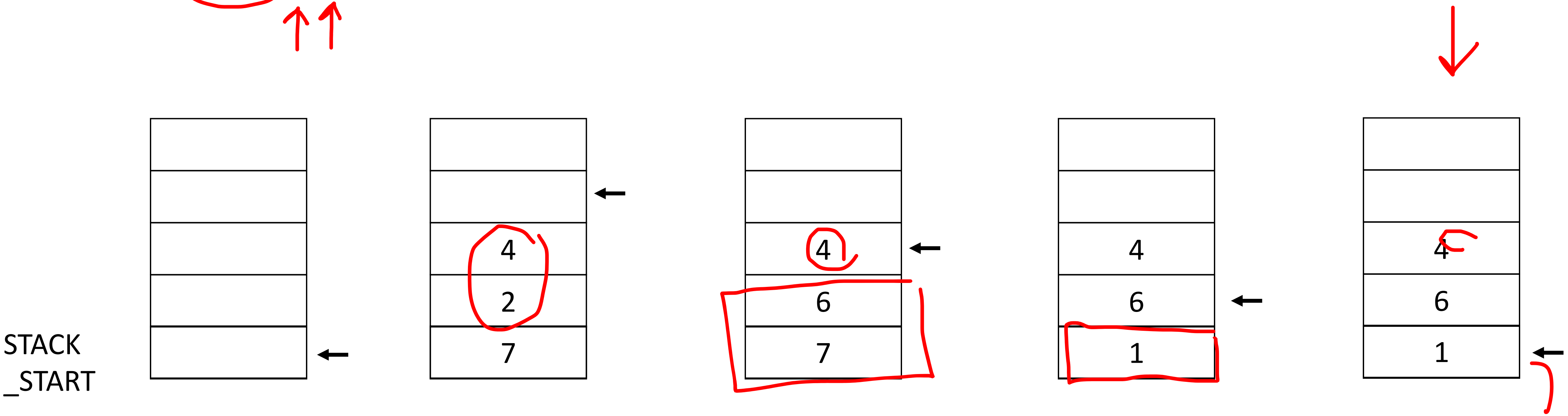
Unbalanced-case2

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

print
“invalid expression”

# Valid Post Expression & Stack

7 2 4 + - =

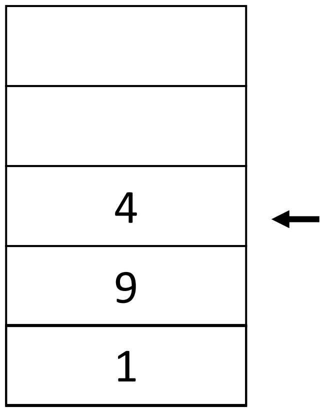


| Empty | After 3 numbers | After + | After - | After = |
|---|---|---|---|---|
| | push 7<br>push 2<br>push 4 | pop 4<br>pop 2<br>push 2+4=6 | pop 6 → 2<br>pop 7 → 1<br>push 7-6=1 | pop 1<br>Result : 1 |

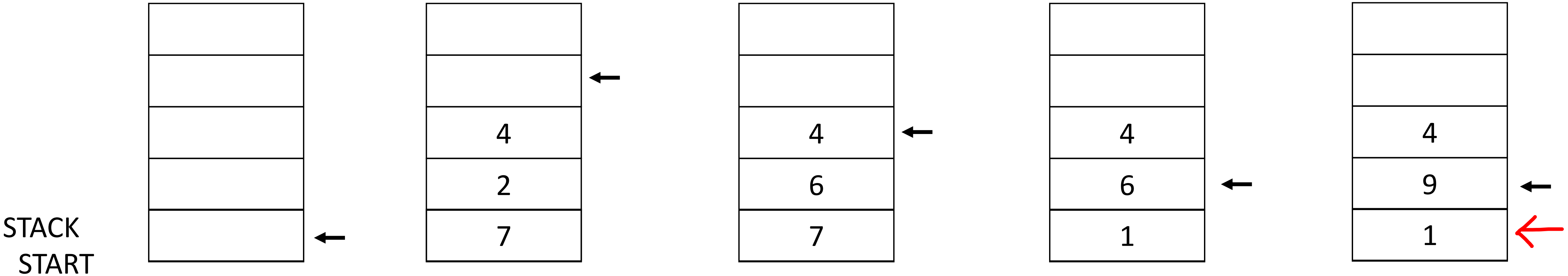
# Invalid Post Expression & Stack

~~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

# MP2 - Part1: Postfix Expression & Stack

## Unbalanced-case1

(Underflow while inputting)

## Unbalanced-case2

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

If you meet ‘=‘ , do 2 POPs

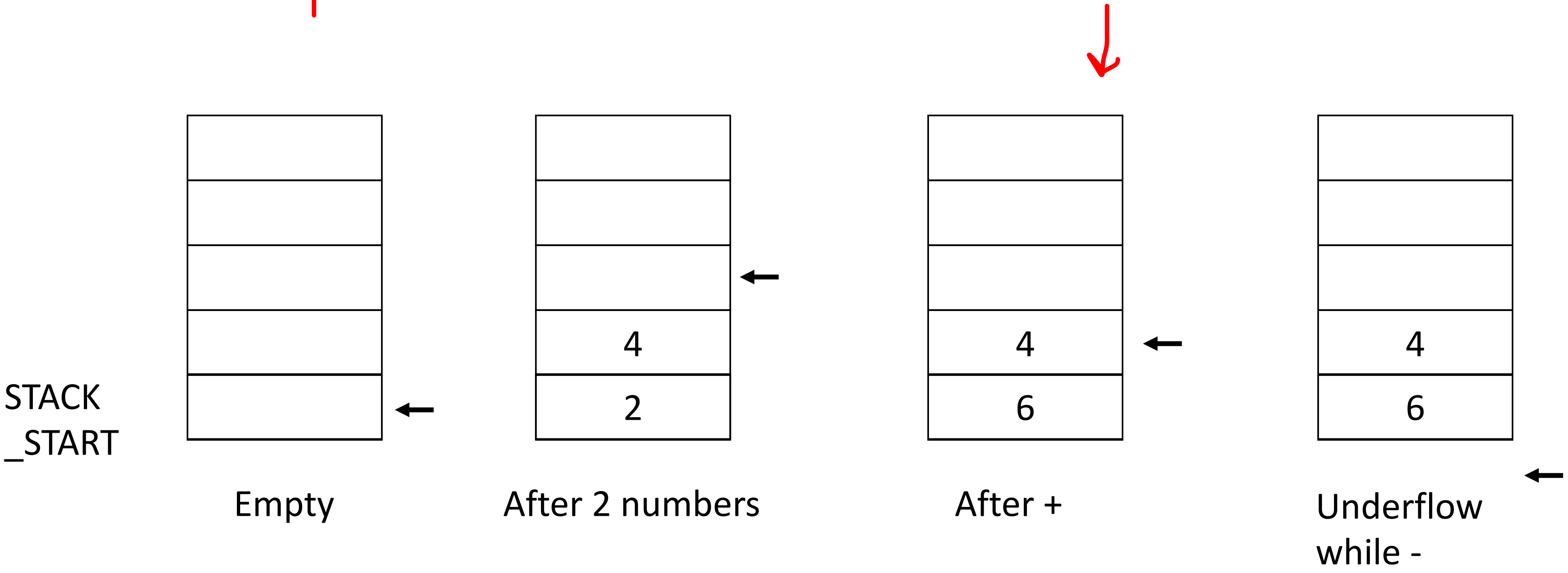
- first POP to grab the result
- second POP to check it’s empty

→ If underflow, valid

→ If not, invalid

# Invalid Post Expression & Stack

2 4 + - =



STACK
_START

| Empty | After 2 numbers | After + | Underflow while - |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | 4 | 4 | 4 |
| | 2 | 6 | 6 |

# MP2 - Part2: Operators

## Add (+)

## Subtract (-)

## Multiply (*)

- (2*3) → 2 + 2 + 2
- ((-2)*3) → (-2)+(-2)+(-2)
- → (2*(-3)) → - (2+2+2)
- sign?

## Divide (/)

- (7/2)=3 → 7 – 2 – 2 – 2
- input values are positive

## Power (^)

- 2^4 → ((2*2) * 2)*2
- input values are positive
- nested subroutine (multiply)