ECE220: Stack Data Structure

Lecture 4

Sayan Mitra

Outline

- Review of Stack
  - Interface: Push, Pop, Isfull, IsEmpty
  - Implementation choices
- Applications of Stack
  - Example 1
  - Example 2

Stack Data Structure

- Basic operations
  - Push a: puts “a” at the top of the stack
  - Pop: removes the element at the top of the stack and returns it
  - Any other useful operations?
- What if memory is full?

Refresh: Design choices for implementing Stack

- **Definition.** Top of stack: Memory location which is the **first location available** above (less than) the last element pushed onto the stack
- The top of stack is **stored at a memory location** labeled as STACK_TOP and loaded to a register (R4 in this case) when needed
- PUSH a: First place a in memory at location stored in STACK_TOP and then **decrement** the value stored in STACK_TOP
- POP: **First increment** the value stored in STACK_TOP and **then read** from the address stored at the incremented STACK_TOP
- STACK_START: first memory location available for our stack
  - IsEmpty: val in STACK_TOP = val in STACK_START
- STACK_END: last memory location available for our stack
  - IsFull: val in STACK_TOP = val in STACK_END - 1

Label/address		
x44FF		
x4500		
…		
x45FD		
x45FE	C	
x45FF	B	
x4600	A	
STACK_TOP	x45FD	
STACK_START	x4600	
STACK_END	X4500	

Problems

- What are some potential problems with implementing a stack in memory?
- What could happen if we didn't check underflow when trying to POP?
- What could happen if we didn't check overflow when trying to PUSH?

PUSH subroutine implementation

- Argument
  - Value to be pushed onto the stack
  - Passed to the subroutine in R0
- Result
  - To indicate if push was successful
  - Will be returned in R5 (0 – success, 1 – fail)
- Internally uses R3 and R4

```
; IN: R0, OUT: R5
; R3: STACK_END
; R4: STACK_TOP
;
PUSH
;
; prepare registers
;
    ST R3, PUSH_SaveR3  ; save R3
    ST R4, PUSH_SaveR4  ; save R4
    AND R5, R5, #0 ; clear R5, indicates success
    LD R3, STACK_END
    LD R4, STACK_TOP
;
; check for overflow (when stack is full)
;
    ADD R3, R3, #-1
    NOT R3, R3
    ADD R3, R3, #1
    ADD R3, R3, R4; R3 = STACK_TOP – (STACK_END-1)
    BRz OVERFLOW ; stack is full
;
; store value in the stack
;
    STR R0, R4, #0   ; push onto the stack
    ADD R4, R4, #-1  ; move top of the stack
    ST R4, STACK_TOP ; store top of stack pointer
    BRnzp DONE_PUSH
;
; indicate the overflow condition on return
;
OVERFLOW ADD R5, R5, #1
;
; restore modified registers and return
;
DONE_PUSH
    LD R3, PUSH_SaveR3
    LD R4, PUSH_SaveR4
    RET
;
PUSH_SaveR3 .BLKW #1
PUSH_SaveR4 .BLKW #1
STACK_TOP .FILL x4000
STACK_START .FILL x4000
STACK_END .FILL x3FF0
.END
```

Label/address		
x3FEF		
x3FF0		
…		
x3FFD		
x3FFE		
x3FFF		
x4000		
STACK_TOP	x4000	
STACK_START	x4000	
STACK_END	x3FF0	

POP subroutine implementation

- Argument - none
- Result
  - R0: Value popped of the stack
  - R5: Indicator if pop was successful (0 – success, 1 – fail)
- Internally uses R3 and R4

```
; OUT: R0 OUT: R5
; R3: STACK_START
; R4: STACK_TOP
;
POP
;
; prepare registers
;
    ST R3, POP_SaveR3  ; save R3
    ST R4, POP_SaveR4  ; save R4
    AND R5, R5, #0        ; clear R5 (assume success)
    LD R3, STACK_START
    LD R4, STACK_TOP
;
; check for underflow (when stack is empty)
; STACK_START = STACK_TOP, R4 – R3 = 0
    NOT R3, R3
    ADD R3, R3, #1
    ADD R3, R3, R4
    BRz UNDERFLOW ; stack is empty, no pop
;
; remove value from the stack
;
    ADD R4, R4, #1  ; move top of the stack
    LDR R0, R4, #0   ; read value form the stack
    ST R4, STACK_TOP ; store top of stack pointer
    BRnzp DONE_POP
;
; indicate the underflow condition on return
;
UNDERFLOW ADD R5, R5, #1
;
; restore modified registers and return
;
DONE_POP
    LD R3, POP_SaveR3
    LD R4, POP_SaveR4
    RET
;
POP_SaveR3 .BLKW #1
POP_SaveR4 .BLKW #1
```

Exercises

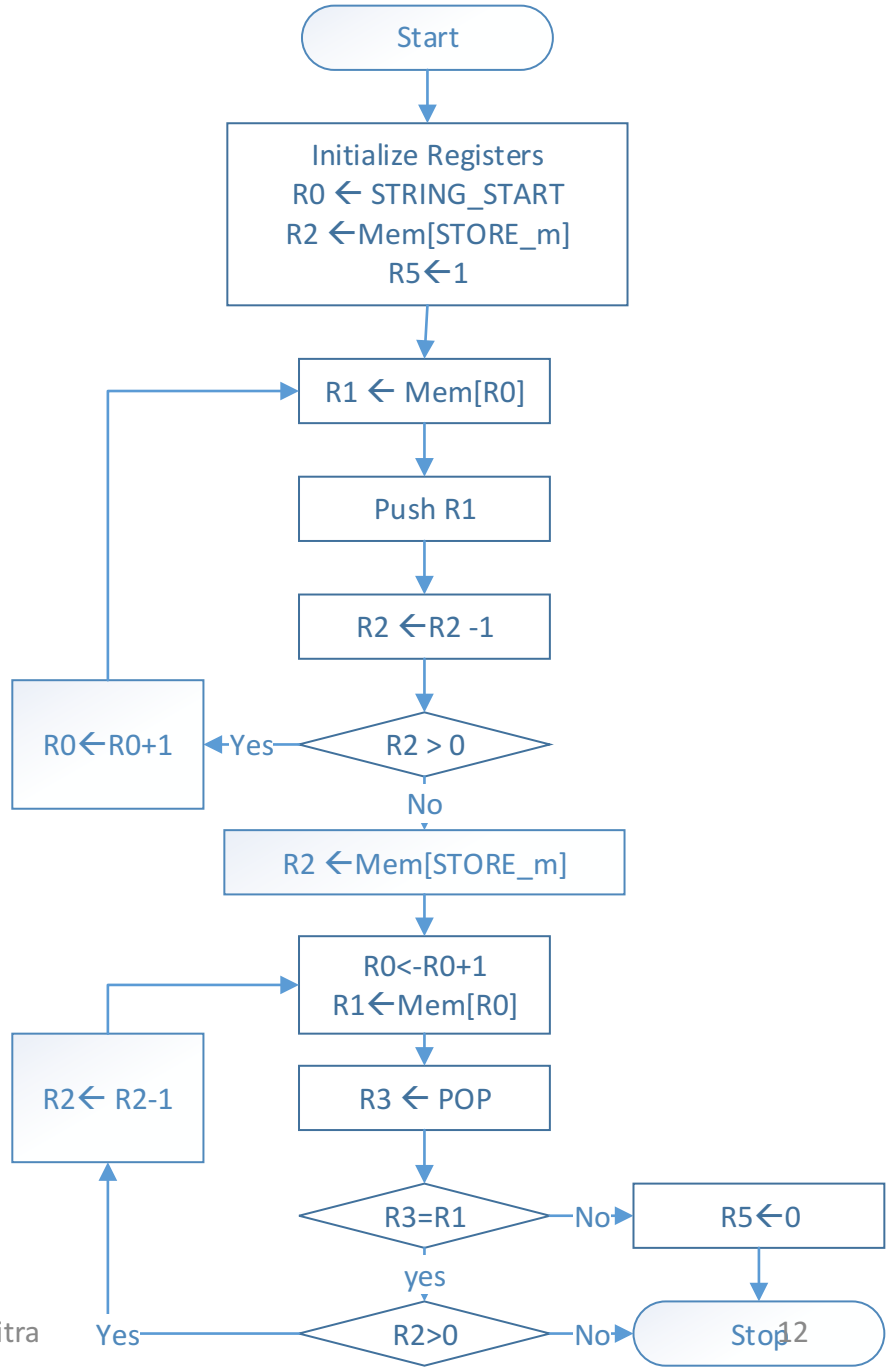
- Implementation of stack growing downward in memory
- Implement a stack of characters
- How to make it work for any data type?

Problem 1: Palindromes

- Examples of palindromes
  - Madam
  - Kayak
  - Tennis set won now Tess in net
  - Was it a car or a cat I saw?
  - Aibophobia
- Problem:
  - 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.
  - An implementation of stack (Push/Pop) is given

Design choices & decomposition

- R0: address of character being read
- R1: current character being read
- R3: ‘mirror’ character
- R2: (m - #characters read)



Simple usage

```
.ORIG x3000
; R2 <-- some value
; R3 <-- another value
; store R3 and R2 in the stack
; push one value onto the stack

LD R2, ITEM1 ; prepare argument
LD R3, ITEM2 ; prepare argument
ADD R0, R2, #0 ; prepare argument for Push
JSR PUSH
; push another value onto the stack
ADD R0, R3, #0 ; prepare argument for Push
JSR PUSH
; pop from the stack
JSR POP
ST R0, RITEM1
JSR POP
ST R0, RITEM2
HALT

ITEM1 .FILL x10FF
ITEM2 .FILL x20FF
RITEM1 .BLKW 1
RITEM2 .BLKW 1
```

- Pushes two values into the stack then pops them and stores the results in 2 memory locations
- Does not check for success of Push or Pop but how would we check? Then, how would we respond once we knew?

Exercises

- 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?

Problem 2. Postfix evaluation

- **Definition.** A postfix expression is defined as
  - <operand1><operand2><operator>
  - Example: 3 4 + evaluates to (3 + 4) = 7
- The operands may be postfix subexpressions
  - Example: 3 4 5 + - evaluates to 3 - (4 + 5)
- The advantage of using postfix: no need for parentheses, i.e., unabmiguous
  - Infix 3 + 4 x 5 can have two values (3 + 4) x 5 = 35 OR
  - 3 + (4 x 5) = 23
  - The above two expressions in postfix will be 3 4 + 5 x and 3 4 5 x +, respectively.

Problem 2

- Problem statement. Given a valid postfix expression with numerals and ‘+’, ‘-’, ‘x’ in the form of a string, evaluate it and store the answer in R5.
- 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 stack.
- Example: 3 4 + 5 * and 3 4 5 + *
- How would you write 7-4*(6-2)?

Problem 2

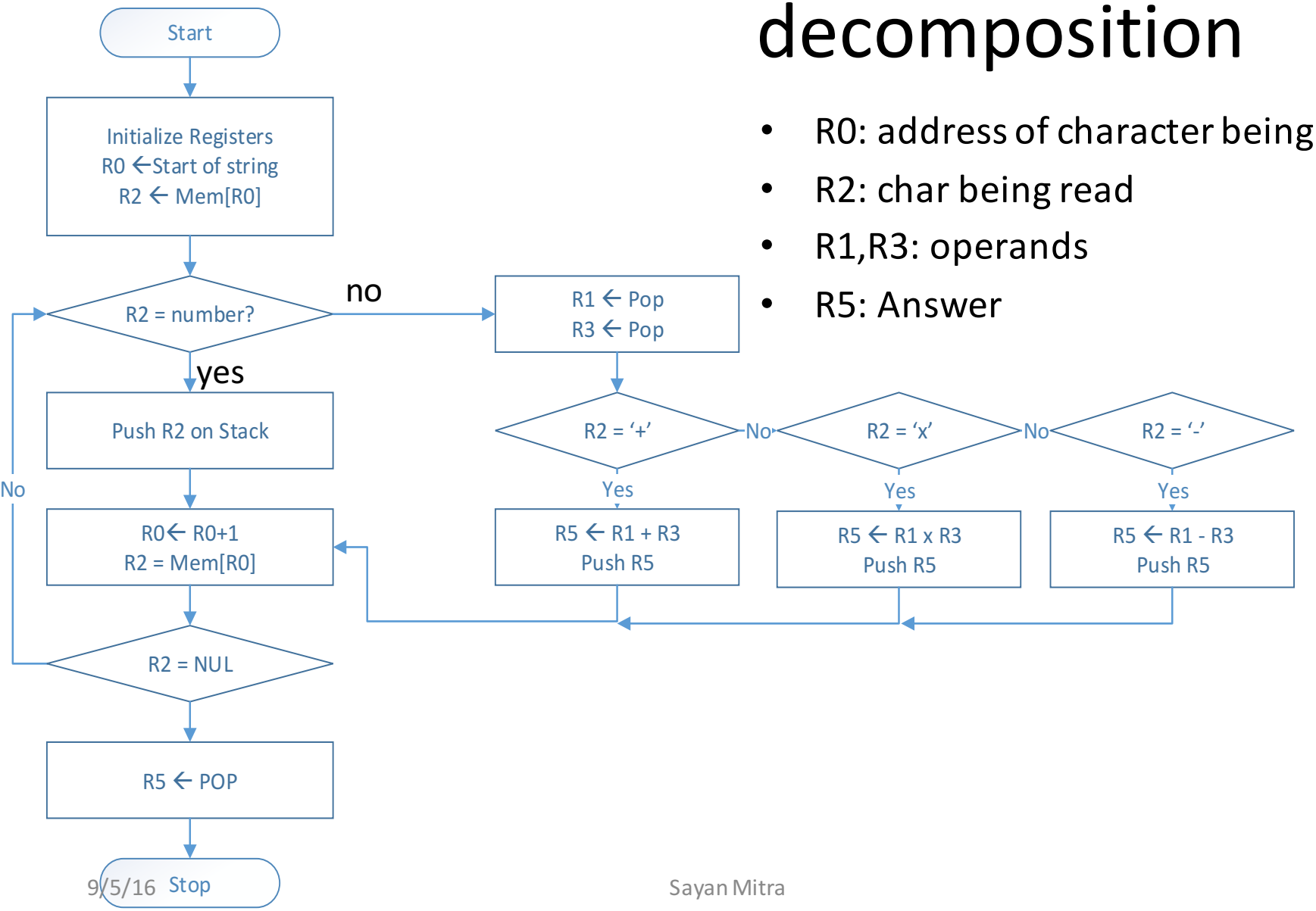
- Example: 3 4 + 5 *
  - input 3: push 3
  - input 4: push 4
  - input +: R1 <- pop (4), R2 <- pop (3), push (R1 + R2 = 7)
  - input 5: push 5
  - input *: R1 <- pop (5), R2 <- pop (7), push (R1 * R2 = 35) Ans
- Example: 3 4 5 + *
  - 3*(4+5)

How to Evaluate

- Input: String of numbers and operators “3 4 + 5 *”
- Output : 35
- Assume each **numerical** argument is a single char (Exercise: how to relax this?)
- Stack implementation is given
- Idea: 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 in stack
  - Done when input expression is completely read

Design choices and decomposition

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



Exercises

- When would this implementation fail?
  - What type of expressions are “bad” for this implementation
- Write the code with subroutine calls to Push, Pop, Mult, Sub