ECE 220 Computer Systems & Programming

Lecture 4: Programming with Stack

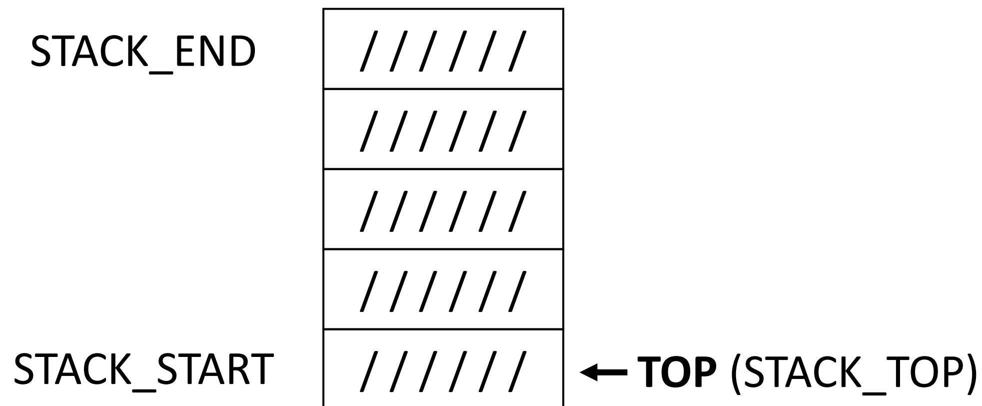
January 25, 2024

ECE ILLINOIS

ILLINOIS

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)
```

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
    - "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)

E = (A + B) * (C + D)

; LC-3 implementation	; stack-based calculator

LD R0, A
LD R1, B
ADD R0, R0, R1
LD R2, C
LD R3, D
ADD R2, R2, R3
MUL R0, R0, R2 ; assuming MULT exists
ST R0, E
	PUSH A
	PUSH B
	ADD
	PUSH C
	PUSH D
	ADD
	MULT
	POP E

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
```

Another Protocol for Saving and Restoring Registers

The protocol for saving registers onto the stack and restoring them might look as follows:

- Once entered the subroutine
  - Push values from all registers that are to be used/modified in the subroutine onto the stack
- Before exiting the subroutine
  - Pop all values from the stack and store them back in the registers

- Example: Implement the multiplication function using this protocol

```
.ORIG x3000
; R1 <- a
; R2 <- b
; prepare arguments
   AND R0, R0, #0
   ADD R1, R0, #5 ; R1 <- 5
   ADD R2, R0, #7 ; R2 <- 7
 ;….. R3 has some important value
; save R3
   ADD R0, R3, #0
   JSR PUSH

; call subroutine
   JSR MULT ; R3 <- R1 * R2

; consume result
   ADD R5, R3, #0

; restore R3
   JSR POP
   ADD R3, R0, #0
; continue
HALT
```

```
; MULT multiplies two positive numbers
; IN: R1, R2
; OUT: R3 <- R1 * R2
```

```
MULT
  ; save R7
  ADD R0, R7, #0
  JSR PUSH

; save R2
   ADD R0, R2, #0 ; R0 <- R2
   JSR PUSH

; compute product
   AND R3, R3, #0
   LOOP ADD R3, R3, R1 ;
   ADD R2, R2, #-1
   BRp LOOP

  ; Restore R2
     JSR POP
     ADD R2, R0, #0 ; R2 <- R0

  ; Restore R7
     JSR POP
     ADD R7, R0, #0
RET
```

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 <u>balanced</u> parentheses:

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

Examples of <u>unbalanced</u> 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 =