19. The Stack
Chapter 10

November 5, 2018

- Data Structures
  - Linked Lists ←
  - Queues ←
- Hardware stack — LIFO
  - Software implementation
  - PUSH and POP
- Arithmetic using a stack

QUEUE: FIFO
FIRST IN FIRST OUT

Stack: An Abstract Data Type

An important abstraction that you will encounter in many applications

We will describe three uses:

Interrupt-Driven I/O
- The rest of the story…

Evaluating arithmetic expressions
- Store intermediate results on stack instead of in registers

Data type conversion
- 2's comp binary to ASCII strings

SUBROUTINE CALLS — RECURSION

Stacks

A LIFO (last-in first-out) storage structure.
- The **first** thing you put in is the **last** thing you take out.
- The **last** thing you put in is the **first** thing you take out.

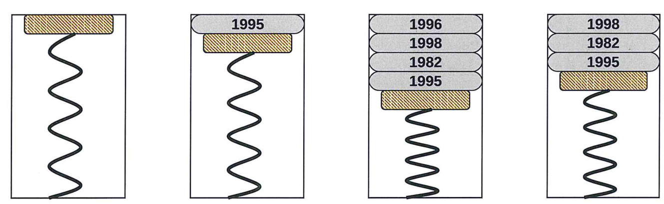
This means of access is what defines a stack, not the specific implementation.

Two main operations:
- **PUSH:** add an item to the stack
- **POP:** remove an item from the stack

A Physical Stack

(HARDWARE)

Coin rest in the arm of an automobile

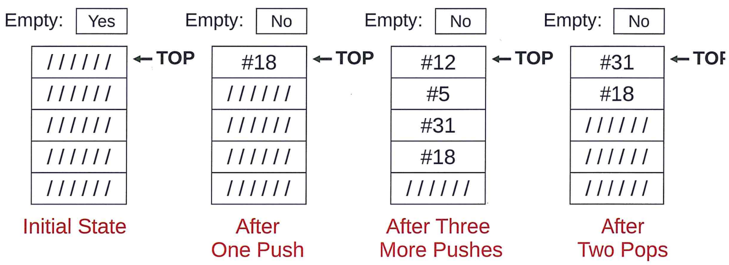


Initial State | After One Push | After Three More Pushes | After One Pop

First quarter out is the last quarter in.

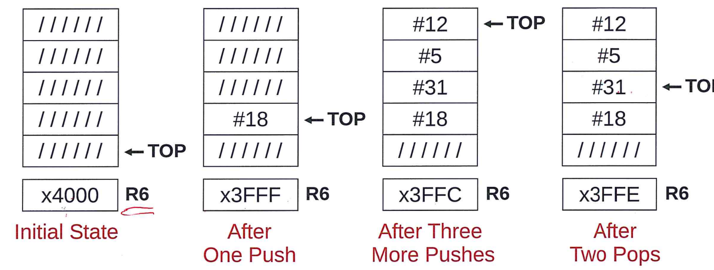
A Hardware Implementation

Data items move between registers



A Software Implementation

Data items don't move in memory, just our idea about there the TOP of the stack is.



By convention, R6 holds the Top of Stack (TOS) pointer.

Basic Push and Pop Code

For our implementation, stack grows downward (when item added, TOS moves closer to 0) DATA IN R0

Push

```
ADD  R6, R6, #-1 ; decrement stack ptr
STR  R0, R6, #0  ; store data (R0)
```

Pop

```
LDR  R0, R6, #0  ; load data from TOS
ADD  R6, R6, #1  ; decrement stack ptr
```

increment

Pop with Underflow Detection

If we try to pop too many items off the stack, an underflow condition occurs.

- Check for underflow by checking TOS before removing data.
- Return status code in R5 (0 for success, 1 for underflow)

```
POP    LD  R1, EMPTY  ; EMPTY = -x4000
       ADD R2, R6, R1 ; Compare stack pointer
       BRz FAIL       ; with x3FFF
       LDR R0, R6, #0
       ADD R6, R6, #1
       AND R5, R5, #0 ; SUCCESS: R5 = 0
       RET
FAIL   AND R5, R5, #0 ; FAIL: R5 = 1
       ADD R5, R5, #1
       RET
EMPTY .FILL xC000
```

Push with Overflow Detection

If we try to push too many items onto the stack, an overflow condition occurs.

- Check for underflow by checking TOS before adding data.
- Return status code in R5 (0 for success, 1 for overflow)

```
PUSH  LD  R1, MAX    ; MAX = -x3FFB
      ADD R2, R6, R1 ; Compare stack pointer
      BRz FAIL       ; with x3FFF
      ADD R6, R6, #-1
      STR R0, R6, #0
      AND R5, R5, #0 ; SUCCESS: R5 = 0
      RET
FAIL  AND R5, R5, #0 ; FAIL: R5 = 1
      ADD R5, R5, #1
      RET
MAX   .FILL xC005
```

Saving Registers when using Stack

Using R1, R2 and R5

Save R1 and R2 in PUSH and POP routines then restore before return

- Calling program does not have to know that these registers are being used
- "Callee-save"

R5 is needed to report success or failure

- Calling program needs to save R5 before the JSR routine is executed
- "Caller-save"

Arithmetic Using a Stack

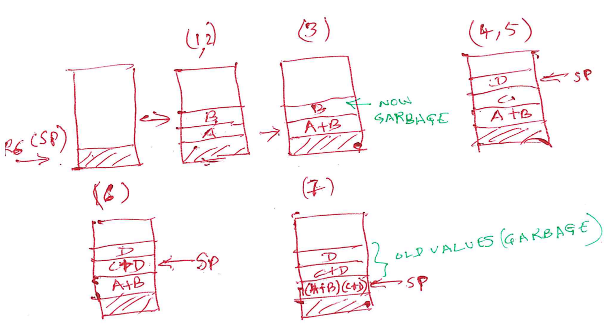
Instead of registers, some ISAs use a stack for source and destination operations: a zero-address machin

- Example:
  ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

Evaluating (A+B)·(C+D) using a stack:

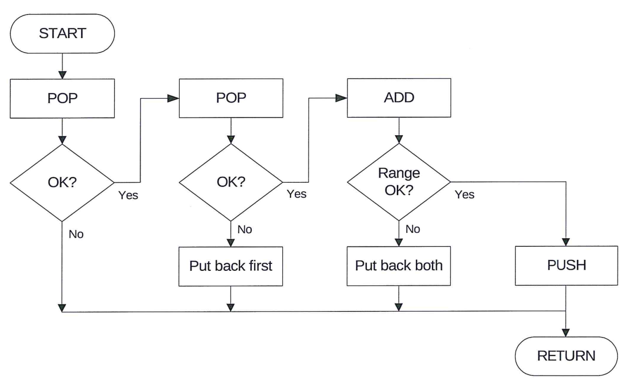
(1) push A
(2) push B
(3) ADD
(4) push C
(5) push D
(6) ADD
(7) MULTIPLY
(8) pop result

Why use a stack?
- Limited registers.
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using FIFO data structure.



Example: OpAdd

POP two values, ADD, then PUSH result.



Example: OpAdd

```
OpAdd     JSR POP        ; Get first operand.
          ADD R5,R5,#0   ; Check for POP success.
          BRp Exit       ; If error, bail.
          ADD R1,R0,#0   ; Make room for second.
          JSR POP        ; Get second operand.
          ADD R5,R5,#0   ; Check for POP success.
          BRp Restore1   ; If err, restore & bail.
          ADD R0,R0,R1   ; Compute sum.
          JSR RangeCheck ; Check size.
          BRp Restore2   ; If err, restore & bail.
          JSR PUSH       ; Push sum onto stack.
         XRET
Restore2  ADD R6,R6,#-1  ; Decrement stack pointer
                         ; (undo POP)
Restore1  ADD R6,R6,#-1  ; Decrement stack pointer
Exit     X RET
```

ISSUE WITH R7 (SUBROUTINE CALLING ANOTHER SUBROUTINE)

SAVE R7 IN MEMORY OR PUSH/POP ON/FROM STACK