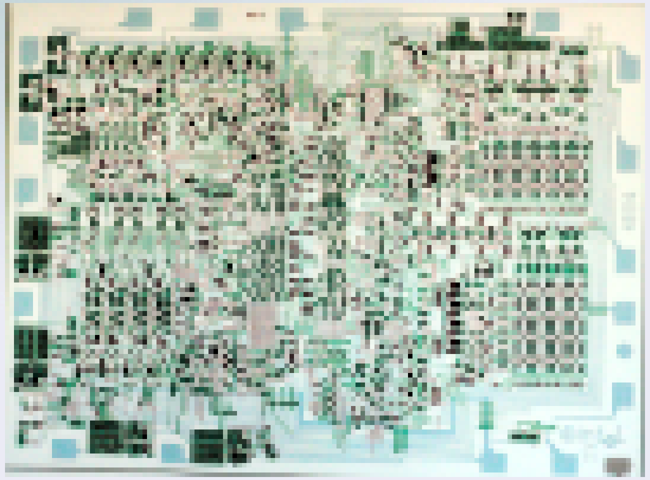


Lecture 7b: Subroutines and Stacks

CSIS11: **Computer Architectures and Organization**

Readings

- *Chapter 8.1-8.2* Patt and Patel: Introduction to Computing Systems...
- *Guide to Using LC-3 Tools* - *handout*
- *Appendix A* Patt and Patel: Introduction to Computing Systems... - *handout*
- C to LC-3 Excellent source for comparing *C* to *LC-3* syntax

Subroutines

- A subroutine is a program fragment that:
  - lives in user space
  - performs a well-defined task
  - is invoked (called) by another user program
  - returns control to the calling program when finished
- Reasons for subroutines:
  - reuse useful (and debugged!) code without having to
  - keep typing it in
  - divide task among multiple programmers
  - use vendor-supplied library of useful routines

JSR/JSRR Instruction

- Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7
  - saving the return address is called “linking”
  - target address is PC-relative (PC + Sext(IR[10:0]))
  - bit 11 specifies addressing mode
    - if =1, PC-relative: target address = PC + Sext(IR[10:0])
    - if =0, register: target address = contents of register

RET Instruction

- RET (JMP R7) gets us back to the calling routine.

DO NOT MESS WITH R7

Passing Information

Arguments

- A value passed in to a subroutine is called an argument.
- This is a value needed by the subroutine to do its job.
  Examples:
  - In OUT service routine, R0 is the character to be printed.
  - In PUTS routine, R0 is address of string to be printed.

Return Values

- A value passed out of a subroutine is called a return value.
- This is the value that you called the subroutine to compute.
  Example:
  - In IN service routine, character read from the keyboard

Stack: An Abstract Data Type

- Key abstraction with multiple uses:
    i. Interrupt-Driven I/O (*more, later*)
    ii. Evaluating arithmetic expressions (using stacks instead of registers)
    iii. Data type conversion (e.g., 2's complement binary to ASCII strings)
    iv. Recursion (a function which calls itself)

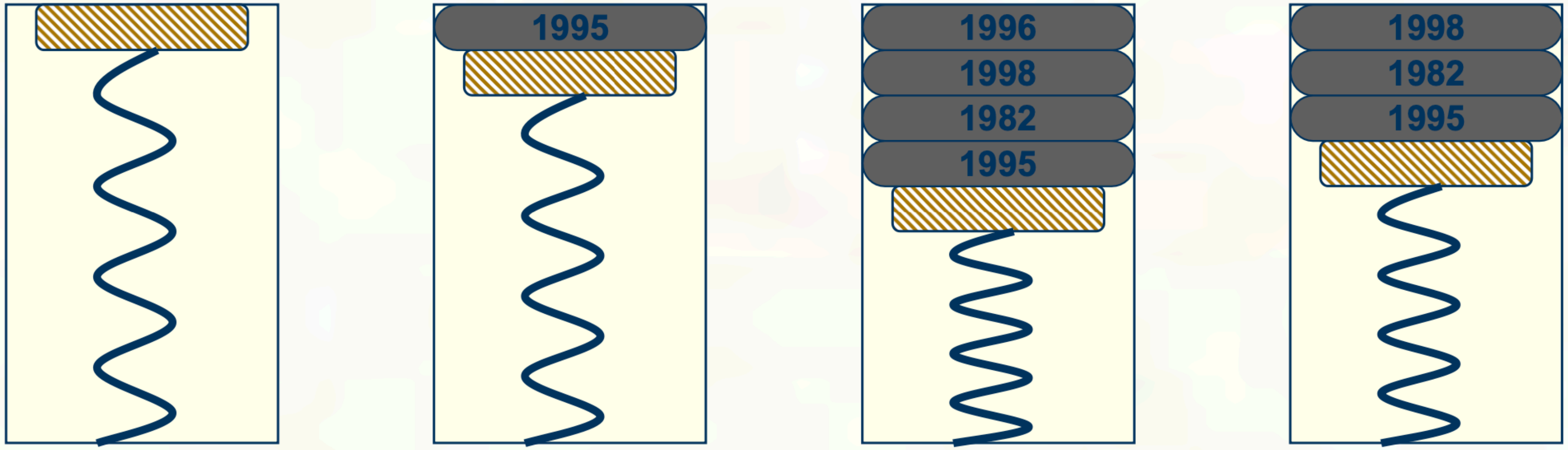
Stacks

- LIFO (Last-In First-Out) structure.
- Operations:
  - **PUSH**: Add an item to the stack.
  - **POP**: Remove an item from the stack.

A Physical Stack

- Analogy: Coin holder in a car armrest.
- Demonstrates LIFO behavior with quarters (last quarter in = first out).

- Coin rest in the arm of an automobile



Initial State

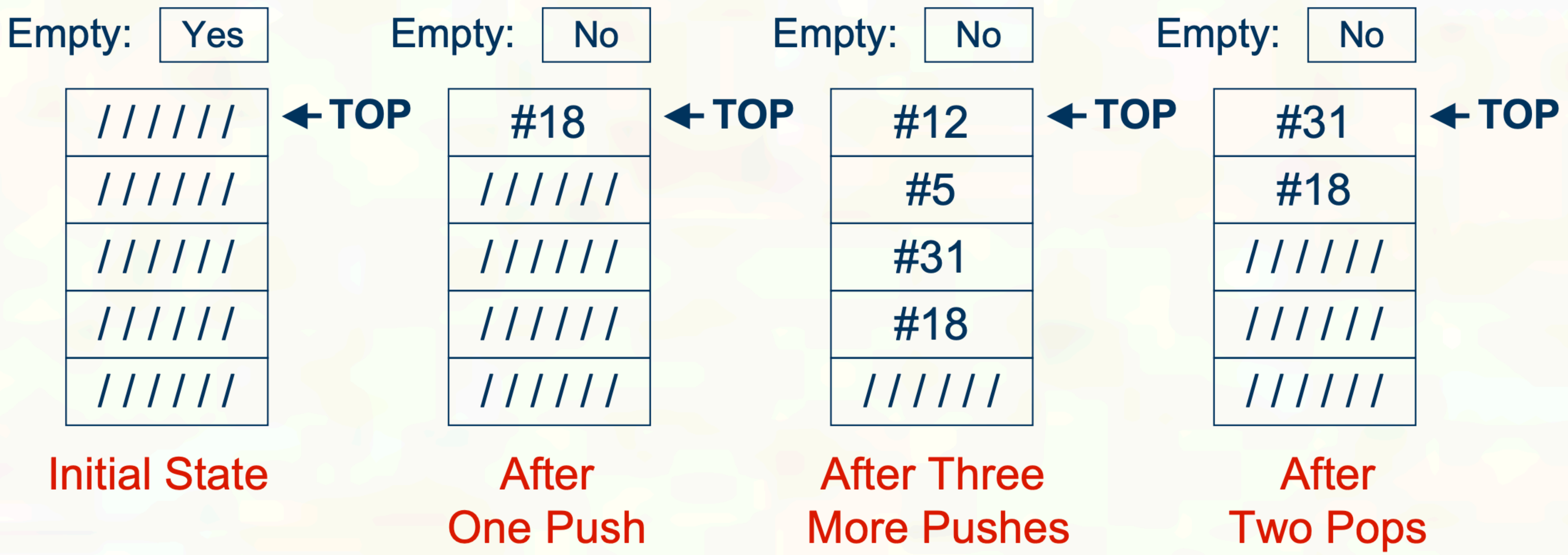
After One Push

After Three More Pushes

After One Pop

A Hardware Implementation

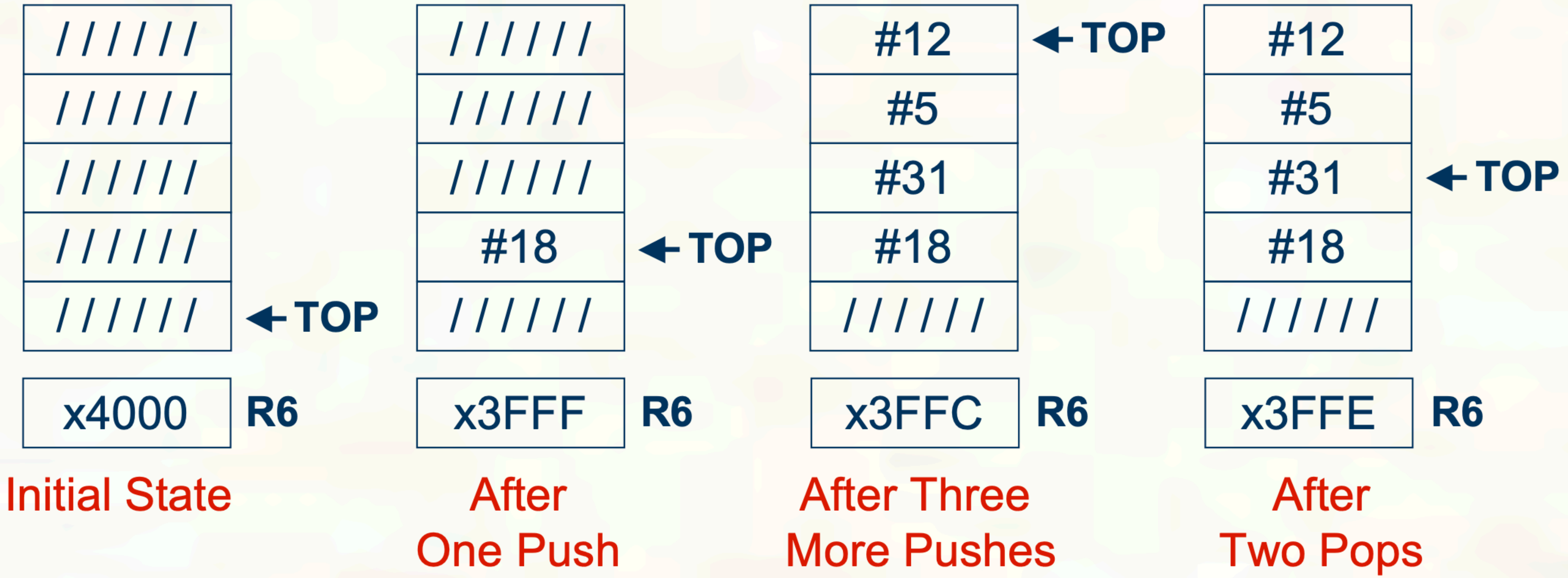
- Stack visualized with fixed memory locations and a TOP pointer.
- Example states: Initial, after pushes, and after pops.


Empty: Yes
//////
//////
//////
//////
//////
TOP
Initial State
Empty: No
#18
//////
//////
//////
//////
TOP
After One Push
Empty: No
#12
#5
#31
#18
//////
TOP
After Three More Pushes
Empty: No
#31
#18
//////
//////
//////
TOP
After Two Pops

A Software Implementation

- Uses register R6 as the Top of Stack (TOS) pointer.
- Stack grows downward (lower memory addresses).
- Example memory addresses: `x4000`, `x3FFF`, etc.

- 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

- **Push**:

```
ADD R6, R6, #-1  ; Decrement stack pointer
STR R0, R6, #0   ; Store data (R0)
```

- **Pop**:

```
LDR R0, R6, #0   ; Load data from TOS
ADD R6, R6, #1   ; Increment stack pointer
```

Pop with Underflow Detection (From Text)

- Checks for empty stack (TOS = `x4000`).
- Returns status code in R5: `0` = success, `1` = underflow.

```
POP     LD R1, EMPTY      ; EMPTY = −x4000
        ADD R2, R6, R1    ; Compare stack pointer
        BRz FAIL          ; with −x4000
        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
```

A Better Way to Define EMPTY

- Define EMPTY as first STACK Address + 1
- Remember that PUSH is defined as *Decrement/Store*
- Use that value to load STACK at initialization
- Then 2's Complement value and store at EMPTY
- This solves both initialization and definition
- Easily changed to new address with one change, less bugs

Push with Overflow Detection

- Checks for full stack (`x3FFB`), only 4 spots allowed
- Returns status code in R5: `0` = success, `1` = 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       ; or −x3FFB
```

See stack.asm