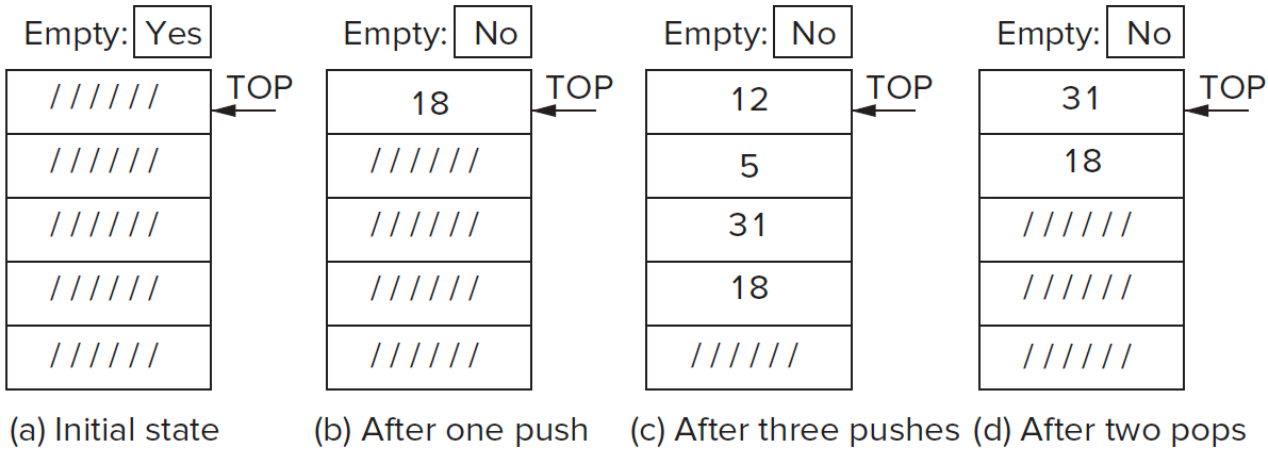
Lec 03 Stack Data Structure and Stack Operations

栈：一种抽象数据类型(ADT)

- **定义**：栈是一种LIFO(Last-In-First-Out,后进先出)的数据结构。
  - 最后压入(push)的元素最先弹出(pop)。
  - 最早压入的元素最后弹出。
- **特点**：
  - 栈的定义由其操作(push&pop)决定，而不是具体的实现方式。
  - 抽象数据类型(ADT)通过操作存储机制。
- **其他ADT示例**：队列、链表、树、字典。

栈的硬件实现

此处数据元素通过两种操作进行移动。

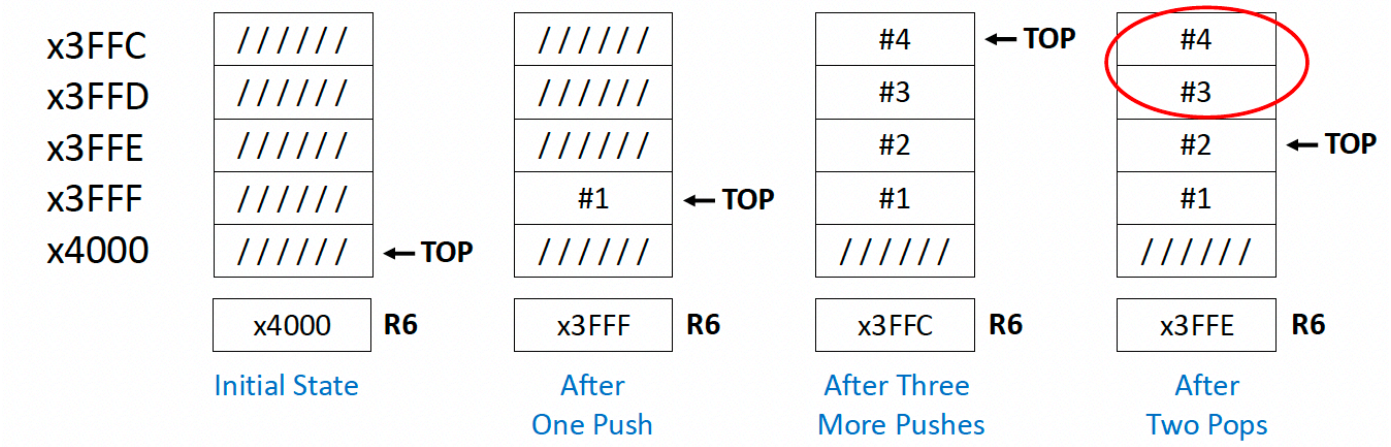


问题：

- 硬件实现中，数据项在操作间移动，效率可能较低，因此我们需要**内存实现**。

栈的内存实现

此处数据元素在内存中不移动，相反，移动的是栈的顶端。

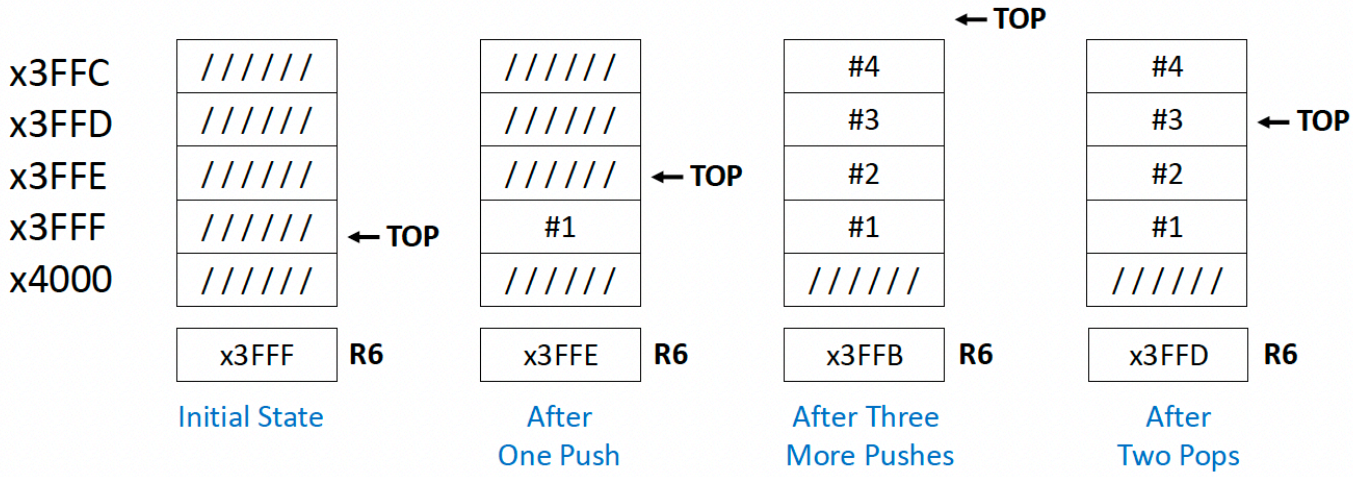


最后一幅图的#3和#4虽然仍然存储在内存中，但是已经无法通过栈来继续访问了。

一般用**寄存器R6**来存储栈顶的地址，当栈中存入一个数据后，栈顶的位置向x0000移动一格

- 操作：
  - **Push**：将数据存入TOS(Top Of Stack)指向的位置，TOS减1。
  - **Pop**：TOS加1，读取该位置的数据。

另一种栈



- 特点：
  - TOS指向“下一个可用位置”。

栈的基本操作

1. **Push**：将数据压入栈顶。

```
; R0:需要被压入栈中的数据
ADD R6, R6, #-1   ; 减小TOS
STR R0, R6, #0    ; 将R0存入TOS指向的位置
```

1. **Pop**：从栈顶弹出数据。

```
; R0:输出的数据
LDR R0, R6, #0   ; 从TOS读取数据到R0
ADD R6, R6, #1   ; 增大TOS
```

1. **溢出检测(Overflow)**：检查栈是否满。
2. **欠载检测(Underflow)**：检查栈是否为空。

栈的内存分配

- **定义**：
  - **STACK_START**: 栈的起始地址（如x4000）。
  - **STACK_END**: 栈的结束地址（如x3FF0）。
  - **STACK_TOP**: 当前栈顶指针。
- **状态**：
  - 栈空：STACK_TOP == STACK_START。
  - 栈满：STACK_TOP == STACK_END - 1。
- **示例**：

```
STACK_TOP .FILL x4000
STACK_START .FILL x4000
STACK_END .FILL x3FF0
; 这表示栈的范围从x4000到x3FF0(包含)
```

包含Overflow/Underflow的Push/Pop子程序实现

Push（此时不需要检查栈空，因为是往栈中加入数据）

- 参数：
  - 输入：R0（要压入的值）。
  - 输出：R5（0表示成功，1表示失败）。
- 代码：

```
PUSH
  ST R3, PUSH_SaveR3    ; 保存R3
  ST R6, PUSH_SaveR6    ; 保存R6
  LD R3, STACK_END      ; 加载STACK_END
  LD R6, STACK_TOP      ; 加载STACK_TOP
  ADD R3, R3, #-1       ; 计算STACK_END-1
  NOT R3, R3            ; 取反
  ADD R3, R3, #1        ; 变为负数
  ADD R3, R3, R6        ; 检查是否溢出
  BRz OVERFLOW          ; 若STACK_TOP == STACK_END-1，跳转
  ADD R6, R6, #-1       ; 减小TOS
  STR R0, R6, #0        ; 存数据
```

```
  ST R6, STACK_TOP      ; 更新STACK_TOP
  AND R5, R5, #0        ; R5 = 0（成功）
  BRnzp DONE_PUSH
OVERFLOW
  ADD R5, R5, #1        ; R5 = 1（失败）
DONE_PUSH
  LD R3, PUSH_SaveR3    ; 恢复R3
  LD R6, PUSH_SaveR6    ; 恢复R6
  RET
```

Pop（此时不需要检查栈满，因为是从栈中取出数据）

- 参数：
  - 输出：R0（弹出的值）、R5（0表示成功，1表示失败）。
- 代码：

```
POP
  ST R3, POP_SaveR3     ; 保存R3
  ST R6, POP_SaveR6     ; 保存R6
  LD R3, STACK_START    ; 加载STACK_START
  LD R6, STACK_TOP      ; 加载STACK_TOP
  NOT R3, R3            ; 取反
  ADD R3, R3, #1        ; 变为负数
  ADD R3, R3, R6        ; 检查是否欠载
  BRz UNDERFLOW         ; 若STACK_TOP == STACK_START，跳转
  LDR R0, R6, #0        ; 读取数据
  ADD R6, R6, #1        ; 增大TOS
  ST R6, STACK_TOP      ; 更新STACK_TOP
  AND R5, R5, #0        ; R5 = 0（成功）
  BRnzp DONE_POP
UNDERFLOW
  ADD R5, R5, #1        ; R5 = 1（失败）
DONE_POP
  LD R3, POP_SaveR3     ; 恢复R3
  LD R6, POP_SaveR6     ; 恢复R6
  RET
```

##