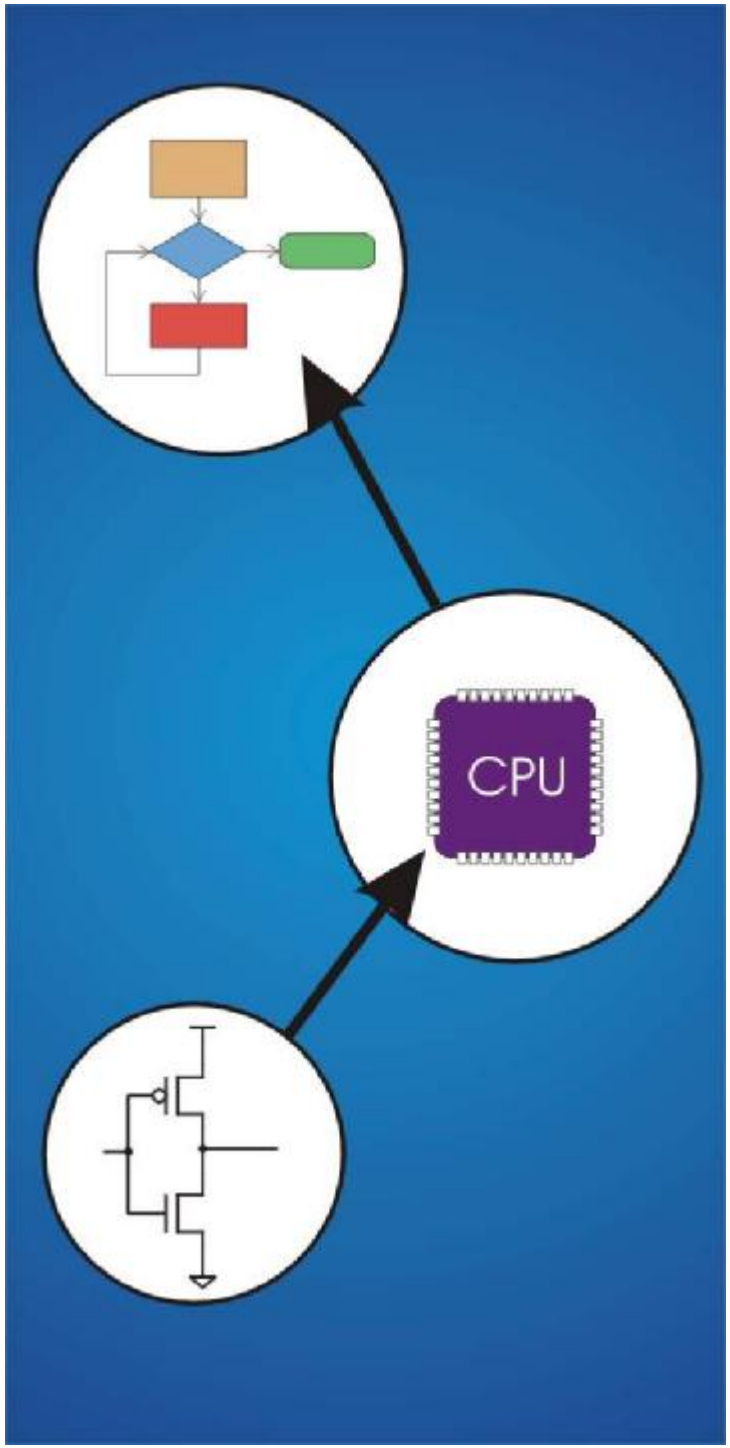


Chapter 10
栈

计算机系统的抽象层次

Problem Specification

compute the fibonacci sequence

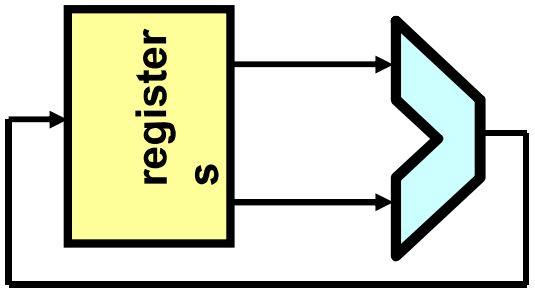
Algorithm Program

```
for(i=2; i<100; i++) {
  a[i] = a[i-1]+a[i-2];}
```

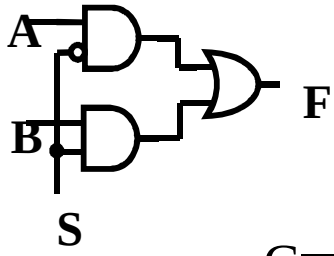
ISA (Instruction Set Architecture)

```
load r1, a[i];
add  r2, r2, r1;
```

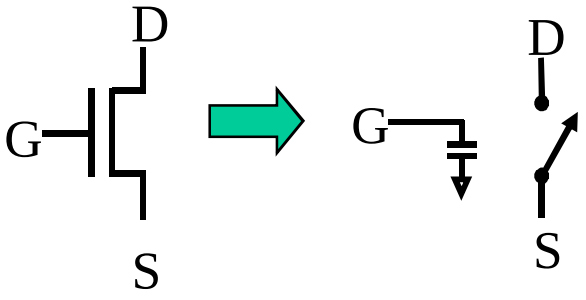
microArchitecture



Logic



Transistors



Physics/Chemistry

高级语言C/C++的内存管理

C/C++编译的程序中两个重要的内存区域：

1、栈区（stack）— 由编译器自动分配释放，存放函数的参数名，局部变量的名等。

2、堆区（heap）— 由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收。

Stack(栈):

由系统自动分配： int b; //系统自动在栈中为b开辟空间

Heap（堆）:需要程序员自己申请，并指明大小

```
p1 = (char *)malloc(10);  //C
p2 = new char[10];        //C++
```

p1,p2本身在栈中，分配的空间在堆中。

问题：计算机启动是执行的第一段代码必须用什么语言？

栈: 一种抽象数据类型

栈是一种存储机制，具有特有的访问规则

栈的重要作用：

中断驱动I/O

- 10.2节

算数运算机制：基于栈的算术运算

- 用栈来存储中间结果，取代寄存器

数据类型转换

- 二进制补码与ASCII字符串之间的转换算法

后续数据结构和操作系统课程还会深入学习

栈的基本结构

定义：栈是一种具有**LIFO (last-in first-out:** 后进先出)访问特性的存储结构

- 第一个放进去的，最后一个取出来
- 最后一个放进去的，第一个取出来

因而栈特殊的地方在于它的访问方式，而不是它的实现。

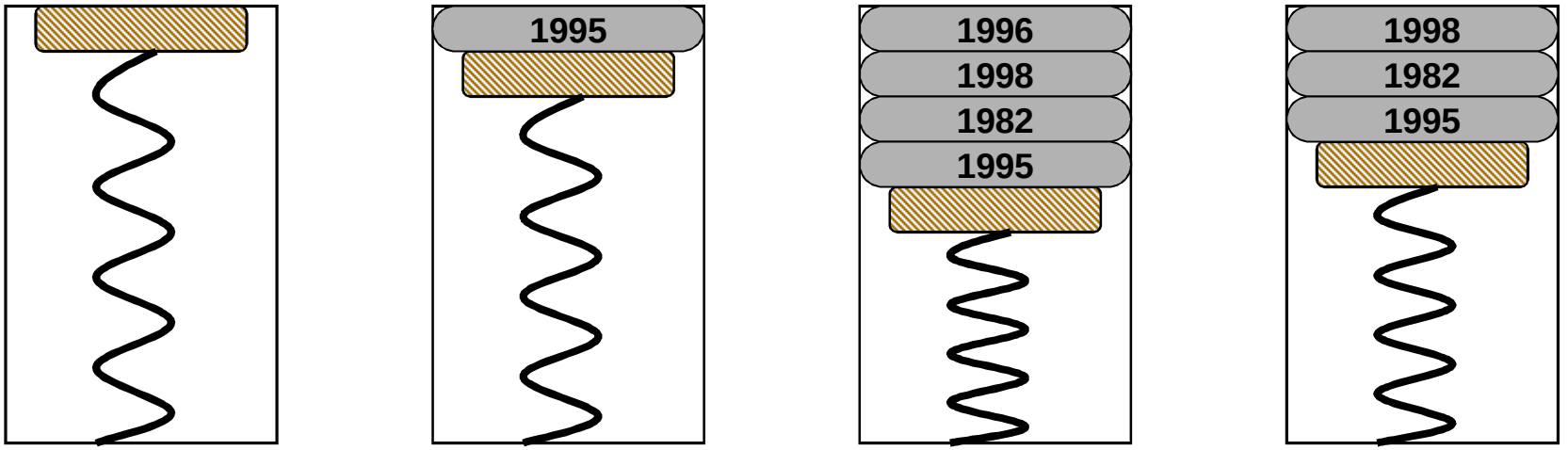
栈的两个主要操作：

PUSH（压入）: 在栈中插入一个元素

POP（弹出）: 在栈中删除一个元素

栈的实例-1

汽车上的硬币盒

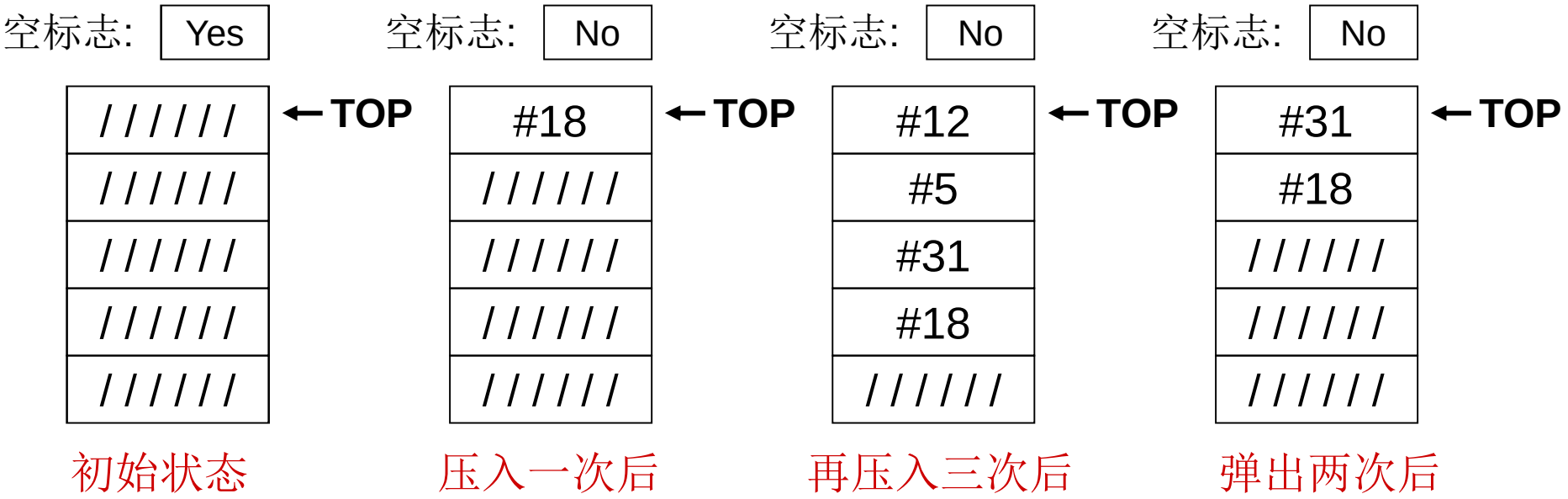


初始状态　　压入一个硬币　　再压入三个硬币　　弹出一个硬币

弹出来的第一个硬币是最后进去的。

栈的实例-2

硬件栈：在寄存器间移动数据



每压入或一个栈元素，栈中所有元素都将一起‘移动’

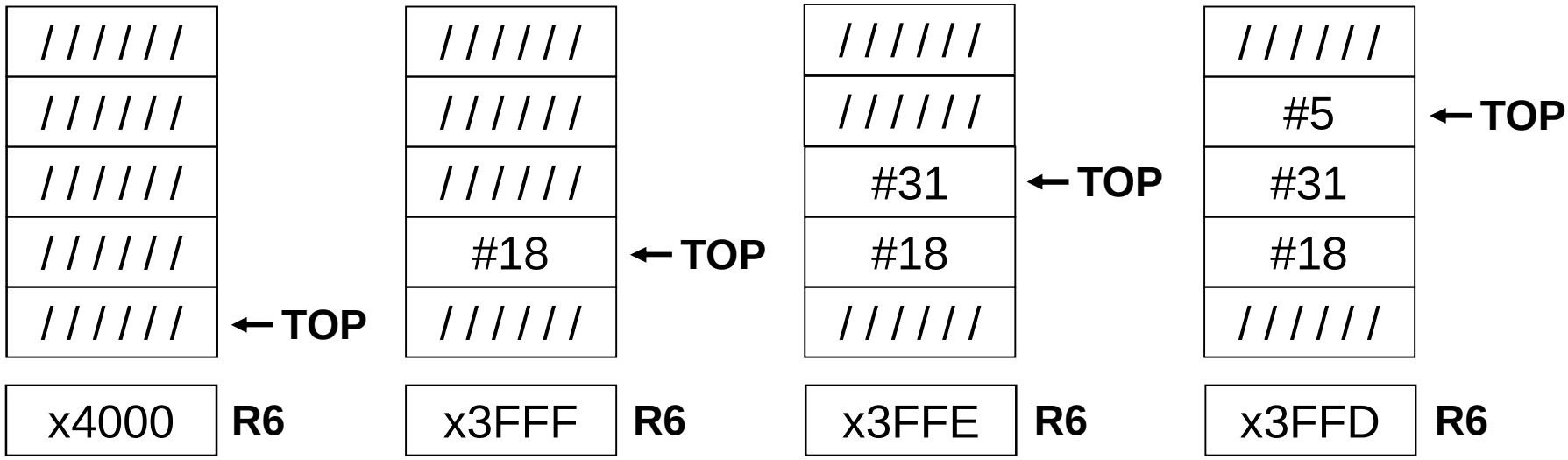
栈的访问总是针对第一个元素，该寄存器标识为‘TOP’

在内存中的实现：软件机制

数据在内存单元之间不需要移动

通过修改栈指针（**TOP**）总是指向最近压入的数据

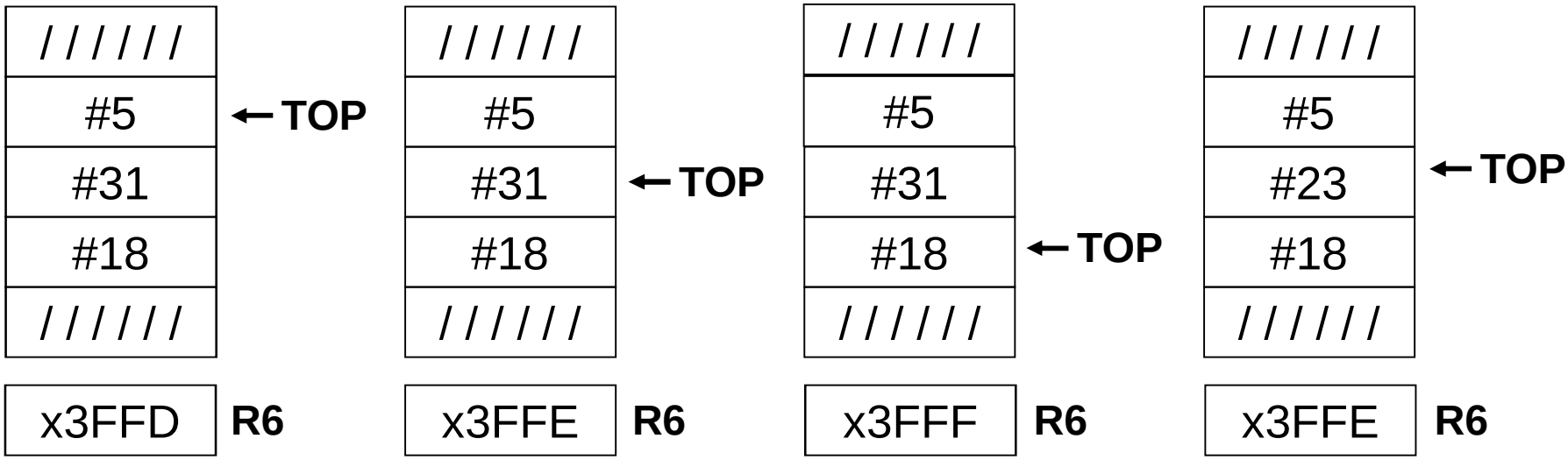
栈向下生长



假定用**R6**寄存器保存**TOP**指针

在内存中的实现：软件机制

弹出过程



假定用**R6**寄存器保存**TOP**指针，**R0**保存读出数据

LC-3:基本 Push 和 Pop 操作的实现指令

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

Push

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

Pop

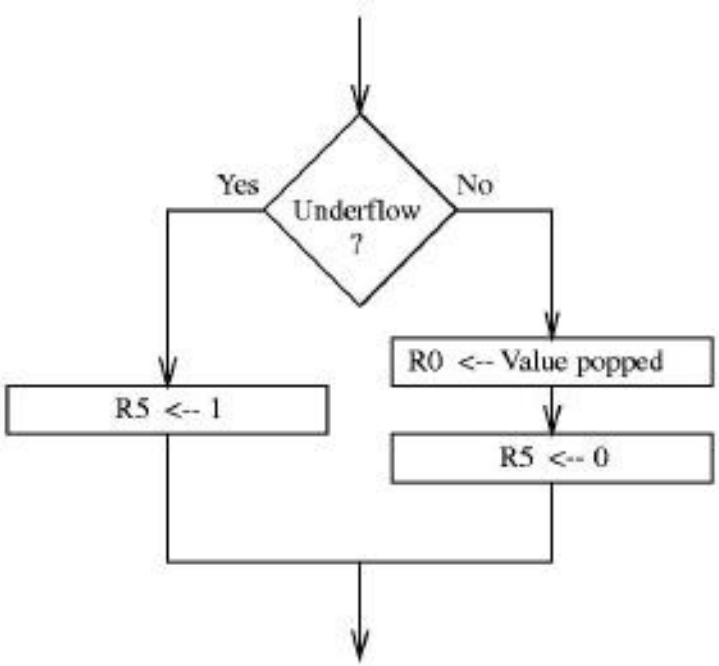
```
LDR   R0, R6, #0  ; load data from TOS
ADD   R6, R6, #1  ; decrement stack ptr
```

加入空栈和溢出检测的设计见**PP. 168-169**

完善Push 和 Pop操作

当栈已经满了或者空了的考虑

- 当执行**PUSH**操作 ，应该先检查栈是否满了
- 当执行**POP**操作，应该先检查栈是否为空
- 使用**R5**返回状态信息， **(0-**成功**, 1-**溢出**)**



支持下溢出检测的POP操作（入口 R6 返回 R0,R5）

如果弹出过多的元素（超过包含元素的个数），会出现下溢出

- 在执行弹出操作前检查是否下溢出
- 将状态记录在寄存器R5 (0-成功, 1-溢出)

```
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
```

Stack TOP: x3FFF

支持上溢出检测的Push操作（入口 R6 R0 返回R5）

如果压入过多的元素（超过栈的空间），会出现上溢出

- 在执行压入操作前检查是否上溢出
- 将状态记录在寄存器R5 (0-成功, 1-溢出)

```
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
```

Stack size: x3FFF to x3FFB

小结

1 调整**empty**和**MAX**的值来调整栈的大小

2 **2**个寄存器**R1**和**R2**在使用前保存其内容，使用结束后恢复原值

3 **R5**由调用程序负责保存

4 参数 **R0** **R6 R5** 需要明确作用

```
01  ;
02  ; Subroutines for carrying out the PUSH and POP functions.  This
03  ; program works with a stack consisting of memory locations x3FFF
04  ; (BASE) through x3FFB (MAX).  R6 is the stack pointer.
05  ;
06  POP             ST      R2,Save2        ; are needed by POP.
07                  ST      R1,Save1
08                  LD      R1,BASE         ; BASE contains -x3FFF.
09                  ADD     R1,R1,#-1       ; R1 contains -x4000.
0A                  ADD     R2,R6,R1        ; Compare stack pointer to x4000.
0B                  BRz     fail_exit       ; Branch if stack is empty.
0C                  LDR     R0,R6,#0        ; The actual "pop"
0D                  ADD     R6,R6,#1        ; Adjust stack pointer.
0E                  BRnzp   success_exit
0F  PUSH            ST      R2,Save2        ; Save registers that
10                  ST      R1,Save1        ; are needed by PUSH.
11                  LD      R1,MAX          ; MAX contains -x3FFB
12                  ADD     R2,R6,R1        ; Compare stack pointer to -x3FFB.
13                  BRz     fail_exit       ; Branch if stack is full.
14                  ADD     R6,R6,#-1       ; Adjust stack pointer.
15                  STR     R0,R6,#0        ; The actual "push"
16  success_exit    LD      R1,Save1        ; Restore original
17                  LD      R2,Save2        ; register values.
18                  AND     R5,R5,#0        ; R5 <-- success.
19                  RET
1A  fail_exit       LD      R1,Save1        ; Restore original
1B                  LD      R2,Save2        ; register values.
1C                  AND     R5,R5,#0
1D                  ADD     R5,R5,#1        ; R5 <-- failure.
1E                  RET
1F  BASE            .FILL   xC001           ; BASE contains -x3FFF.
20  MAX             .FILL   xC005
21  Save1           .FILL   x0000
22  Save2           .FILL   x0000
```

图10-5 栈协议

中断驱动I/O(第二部分)

回顾：中断在第8章介绍过

1. 中断服务程序的启动:有来自设备的中断信号服务请求
2. 中断服务程序的执行:处理器保存相关状态信息，启动中断处理程序
3. 中断服务程序的返回:中断处理程序结束，处理器恢复相关状态信息并重启暂停的程序

第8章没有解释（2）和（3），由于这两步涉及栈，现在继续介绍（2）和（3）

程序状态

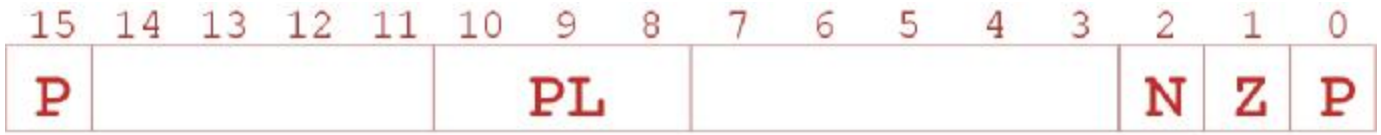
中断 VS TRAP调用　：中断具有随机性

```
add r0,r0,#-1        add r0,r0,#-1
BRz next             trap x21
                     BRz next
```

程序运行所涉及资源的快照，包括：

处理器状态寄存器：Processor Status Register

- 运行模式 **[15], 1** 代表特权模式（超级用户），**0**代表非特权模式（用户）
- 优先级别**[10:8],** 状态码 **[2:0]**



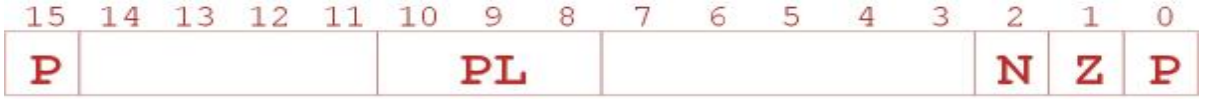
程序计数器：Program Counter（PC）

- 指向下一条执行指令的地址

寄存器：Registers

- 寄存器中保存了未来得及存回内存的处理器运行的临时状态。

处理器状态寄存器



运行模式

- **PSR[15]** 指示当前程序是否运行在超级用户 **Supervisor (0)**状态还是普通用户 **User (1)**模式
- 确保某些资源只允许被操作系统访问。

linux: 内核空间和用户空间

windows: 保护空间和非保护空间

优先级别

- **PSR[10:8]** 保存当前程序的优先级别
  - **Ø 8**个等级, 从 **PL0 (**最低**)**到 **PL7 (**最高**).**

状态码

PSR[2:0] 保存当前的 **NZP condition codes**

中断机制需要保存那些状态

我们只需要保存 **PC** 和 **PSR**

- 寄存器状态由 **ISRs**（中断服务程序）负责保存和恢复 **(“callee save”)**
- 进入中断服务程序前保存**PC** 和 **PSR**，确保程序执行完中断服务程序后能正确返回断点的现场，继续执行。

在哪里保存程序状态？

能使用寄存器保存PC 和 PSR吗？

- 程序员不知道什么时候会发生中断，不可能事先进行保存

在中断服务程序中分配内存?

- 必须在进入中断服务程序前保存
- 迭代和中断的嵌套：中断服务程序可能被高优先级的程序中断
- **PL0　(PL7->PL0)**

保存时机

用户程序被中断处的指令执行完毕（存储阶段之后），在中断服务程序第一条指令的取 指令之前，必须由额外处理器硬件实现。**TRAP**在指令执行时候通过硬件保存**PC**到**R7**

解决方法：**Use a stack!**

- 栈事先实现（硬件或操作系统）
- **Push state to save, pop to restore.**
- 先进后出特性方便支持中断嵌套

Supervisor Stack

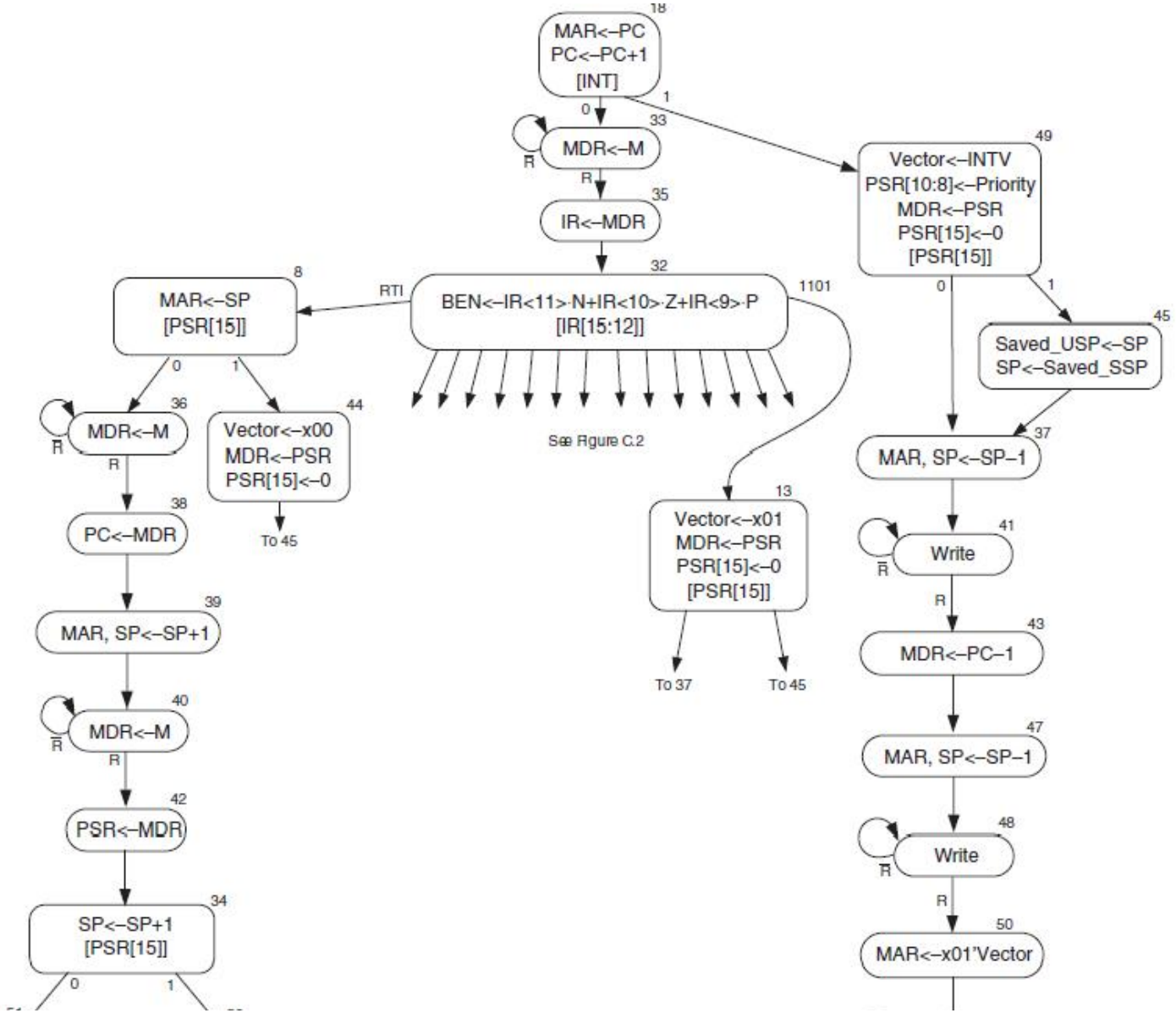
超级用户栈（**Supervisor Stack**）：内存中开辟的一个特殊的栈空间，仅供特权模式下的程序使用，与用户程序栈空间隔离。

- 指向超级用户栈的指针保存在内部寄存器**Saved.SSP.**
- 指向用户栈的指针保存在内部寄存器**Saved.USP.**
- **Saved.SSP， Saved.USP**类似**MDR,MAR**寄存器

所有程序都通过**R6**访问栈空间。

发生中断后，特权模式将从用户模式切换到超级用户模式，并将**R6**的内容保存到**Saved.USP.**

中断的启动　状态18
（在PC增量后，检测int信号是否有效）



中断的执行：

状态49：

1) PSR->MDR，准备入栈

2) PSR[15]<-0切换到特权模式

3)PSR[10:8]<-Priority

记录中断设备的 优先级（3bit）

4)Vector<-INTV

记录中断设备的 中断号(8bit)

5)测试原来的PSR[15]位，以判断堆栈指针指向的是哪一个栈

状态45（原PSR=1）

保存用户堆栈指针，堆栈指针切换到系统超级栈

状态37,41

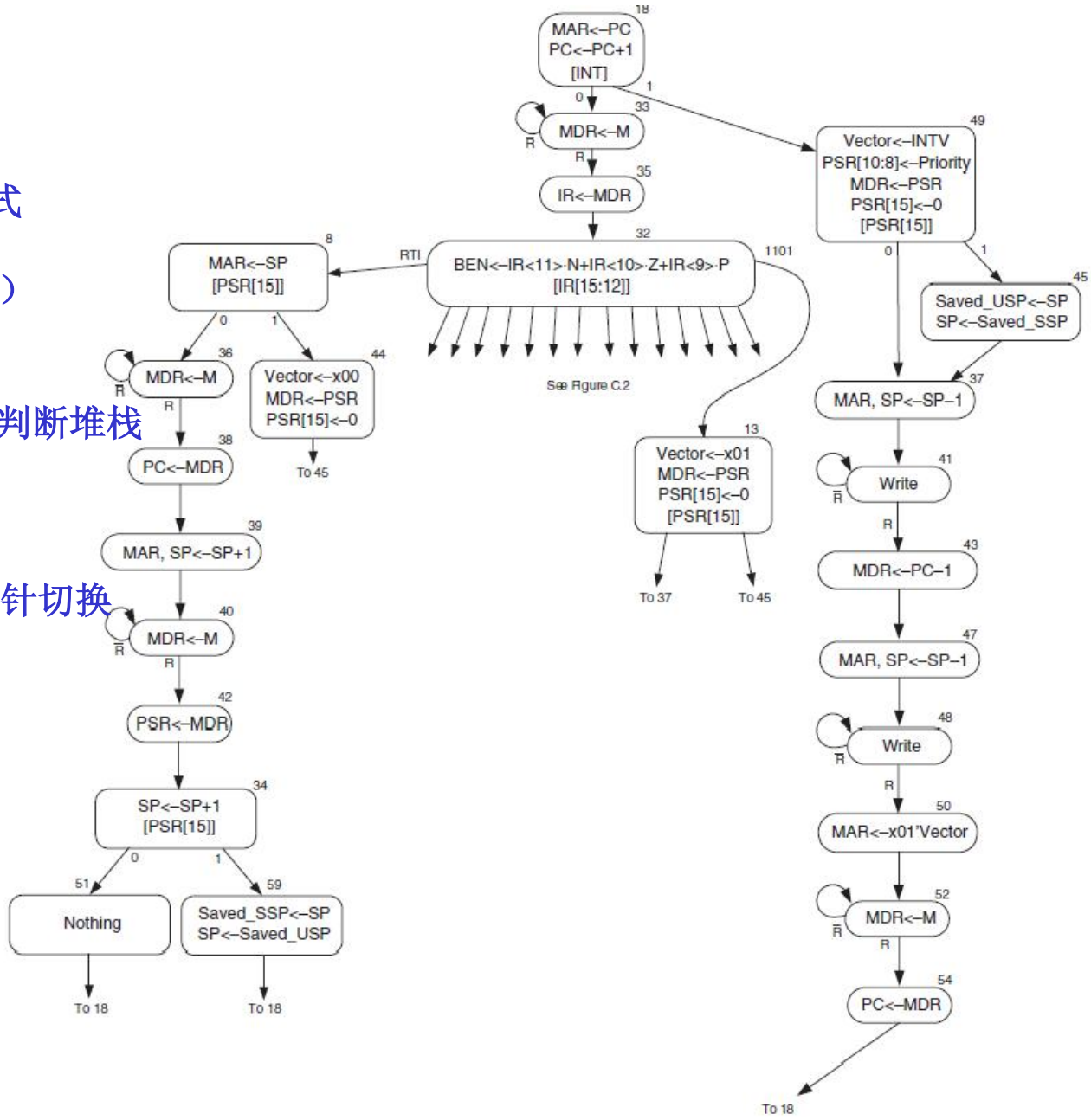
保存MDR中存储的PSR

状态43,47,48

保存PC

状态50,52,54

定位中断服务程序并运行



中断的执行

1 保存现场

2 定位中断服务程序,通过中断矢量表（**x0100-x01ff**，支持**256**个外部设备中断 **x00-xff**）

1. 中断矢量高位扩展，设置 **MAR = x01vv**, 这里 **vv** 为对应于终端设备的**8-bit**的中断矢量 （**INTV**）**(**例如，键盘 **= x80).**
2. 将中断矢量表对应的位置 **(M[x01vv])** 写入 **MDR.**
3. 设置 **PC = MDR**，指向中断服务程序的起始地址。

注: 以上过程发生在用户程序被中断处的指令的存储阶段之后，在中断服务程序第一条指令的取指令之前。

LC-3内存布局总结

x0000 – x00FF Trap vectors (Supports Software Interrupts)

x0020 [x0400] GETC (Read Char from Keyboard)
x0021 [x0430] OUT (Write Character to Console)
x0022 [x0450] PUTS (Write string to Console)
x0023 [x04A0] IN (Prompt, input character from Keyboard, echo character to Console)
x0024 [x04E0] PUTSP (Write “packed” string to Console)

x0025 [xFD70] HALT (Turn off run latch in MCR)

x0100 – x01FF Interrupt Vectors (Supports Hardware Interrupts)

x0200 – x2FFF System Programs & Data (“Operating System”)

x3000 – xFDFF User Programs Area

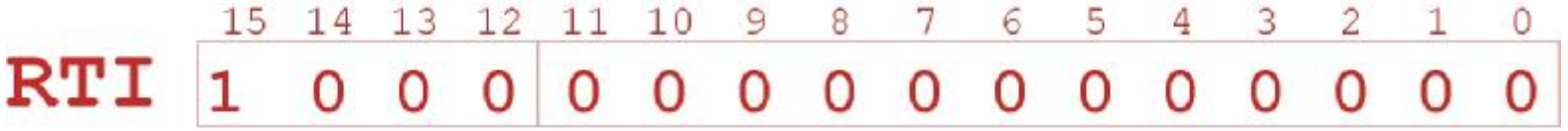
xFE00 – xFFFF I/O Programming “Registers” (Mapped I/O Registers)

xFE00 KBSR [15 {Ready}, 14 {Intr enable}] (Keyboard Status Register)
xFE02 KBDR [7:0{ascii data}] (Keyboard Data Register)
xFE04 DSR [15{Done}, 14{Intr enable}] (Display Status Register)
xFE06 DDR [7:0{ascii data}] (Display Data Register

xFFFE MCR [15{Run latch}] (Machine Control Register)

中断返回

中断返回指令 – RTI – 恢复中断前的用户状态



1. 将 **PC**从超级用户栈弹出 (PC = M[R6]; R6 = R6 + 1)
2. 将 **PSR**从超级用户栈弹出 (PSR = M[R6]; R6 = R6 + 1)
3. 如果 **PSR[15] = 1, R6 = Saved.USP.**
   (如果返回用户模式，需要重新加载用户栈指针；否则不改变栈指针内容)

中断的返回

状态8：

栈顶地址送**MAR,**测试**PSR[15]=0,**则执行返回，否则“权限异常”

状态36，38：

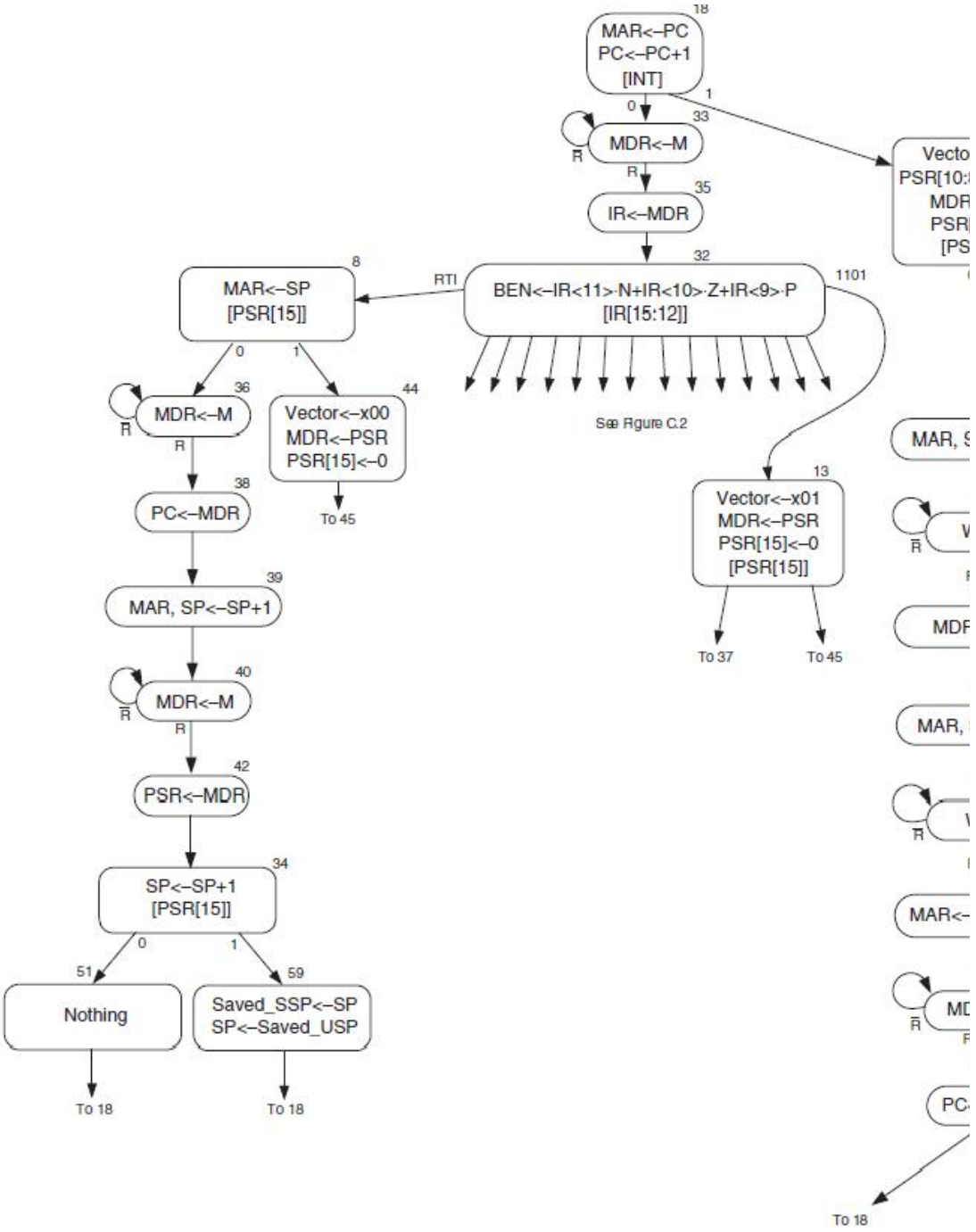
恢复**PC**

状态39，40，42

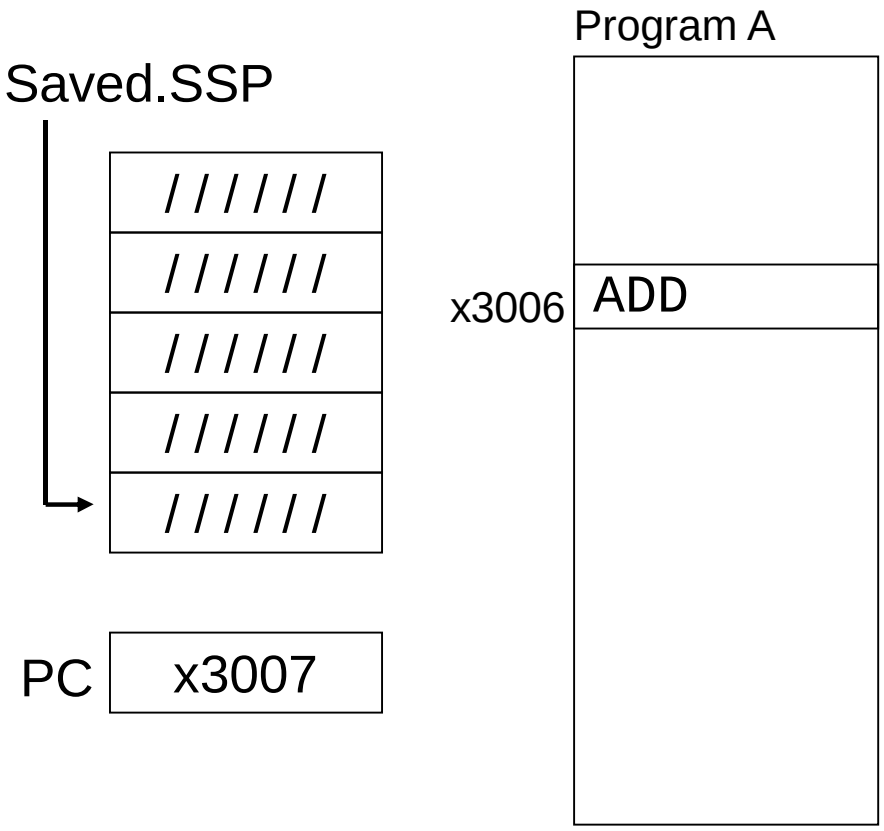
恢复**PSR**

状态34

如果返回用户模式则恢复用户模式下的**SP**指针

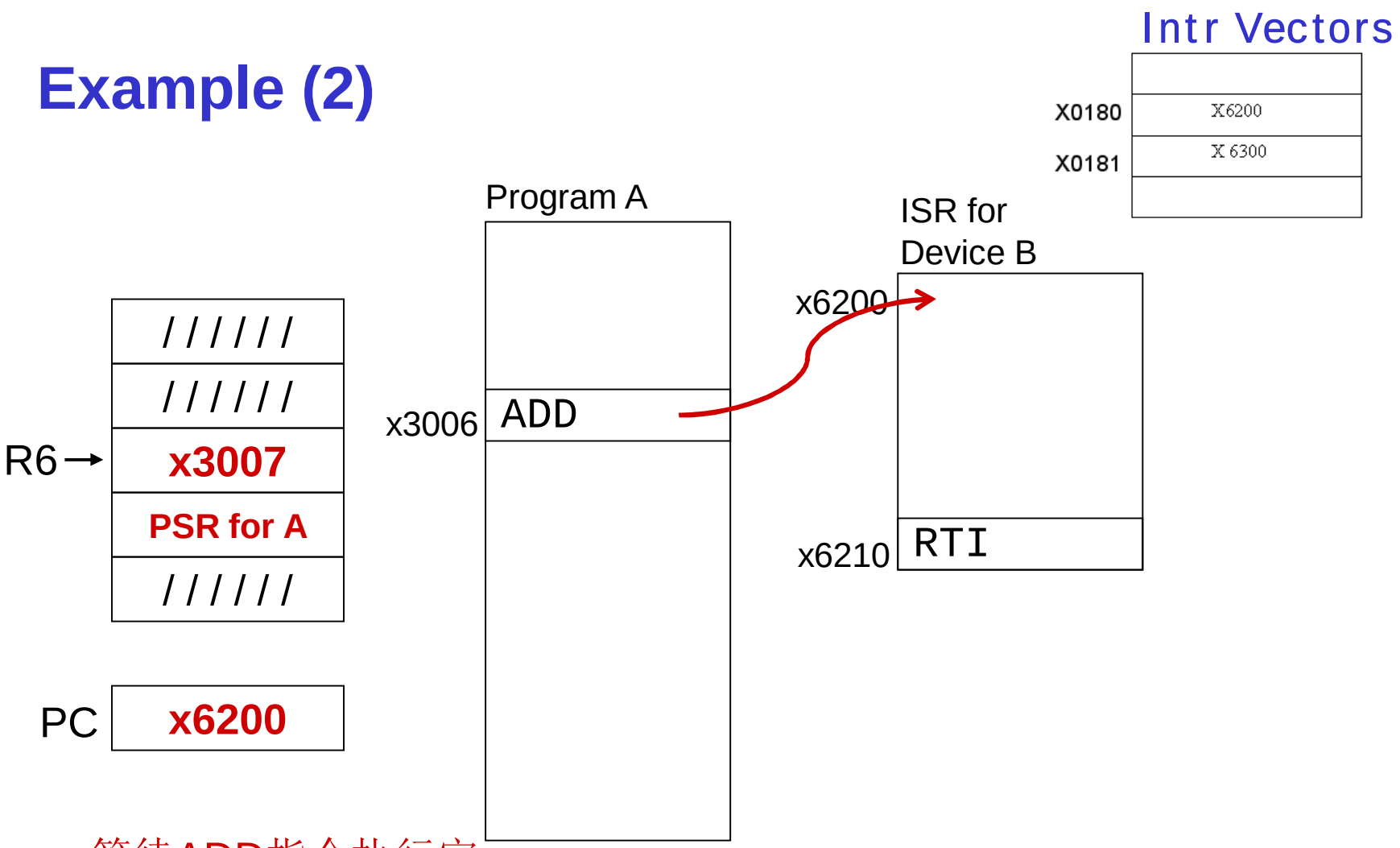


Example (1)



当执行x3006处的ADD指令时，设备B（中断号x80）引发中断

Example (2)



等待ADD指令执行完

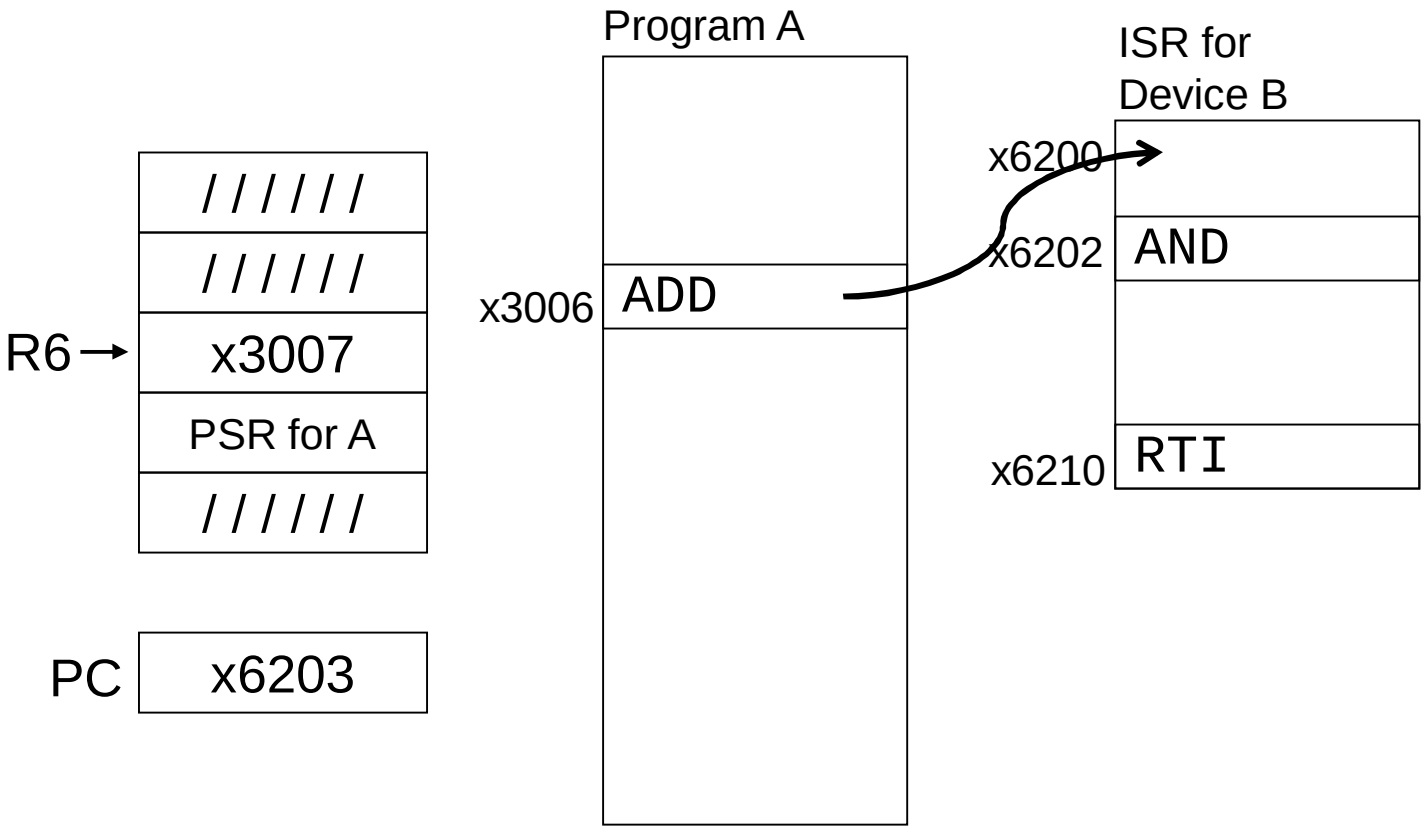
Saved.USP = R6. R6 = Saved.SSP.

将PSR和PC压入超级用户栈

通过中断向量表找到设备B的中断服务程序的入口地址，设置PC

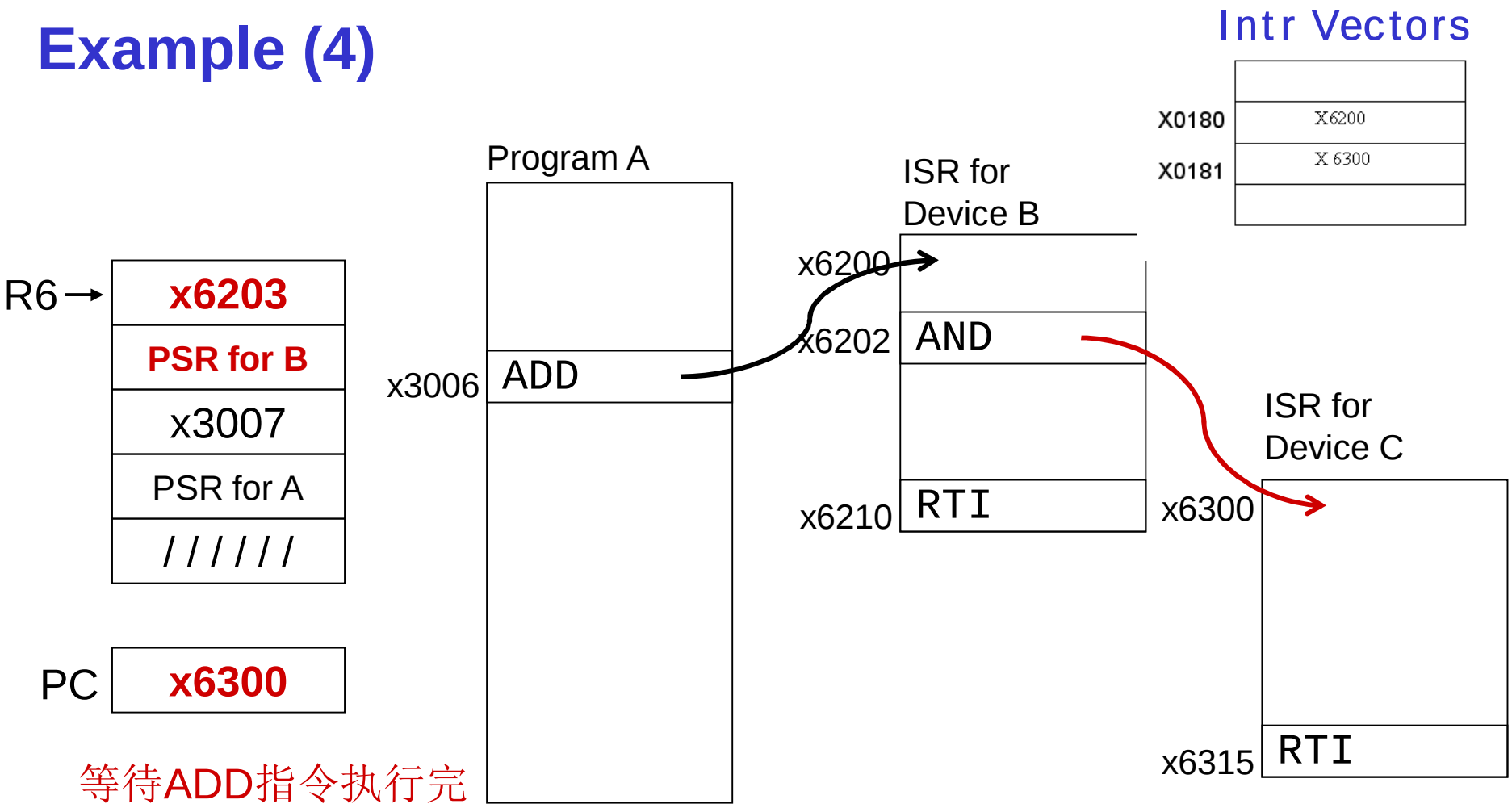
运行设备B的中断服务程序 (在 x6200).

Example (3)



当执行x6202处的ADD指令时，设备c(中短号x81)引发中断

Example (4)



等待ADD指令执行完

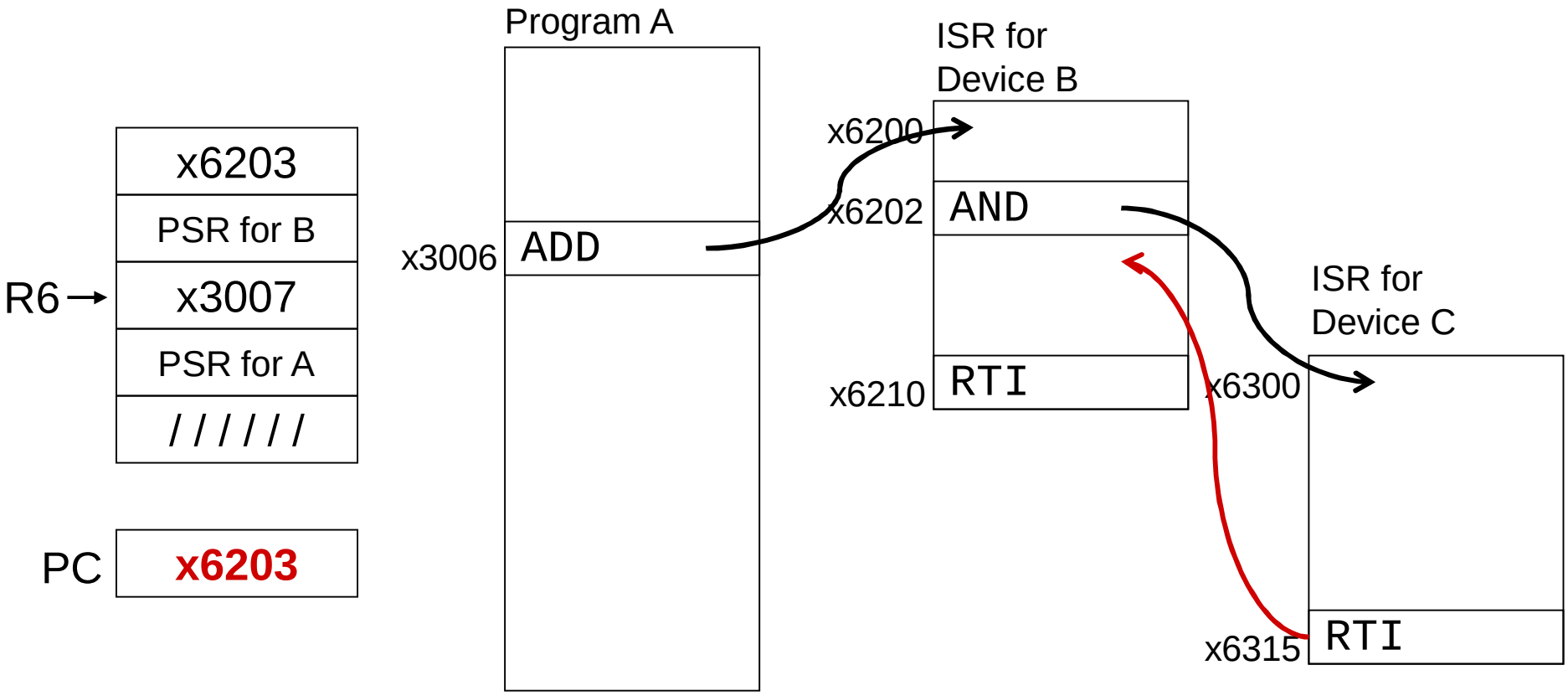
Saved.USP = R6. R6 = Saved.SSP.

将PSR和PC压入超级用户栈

通过中断向量表找到设备C的中断服务程序的入口地址，设置PC

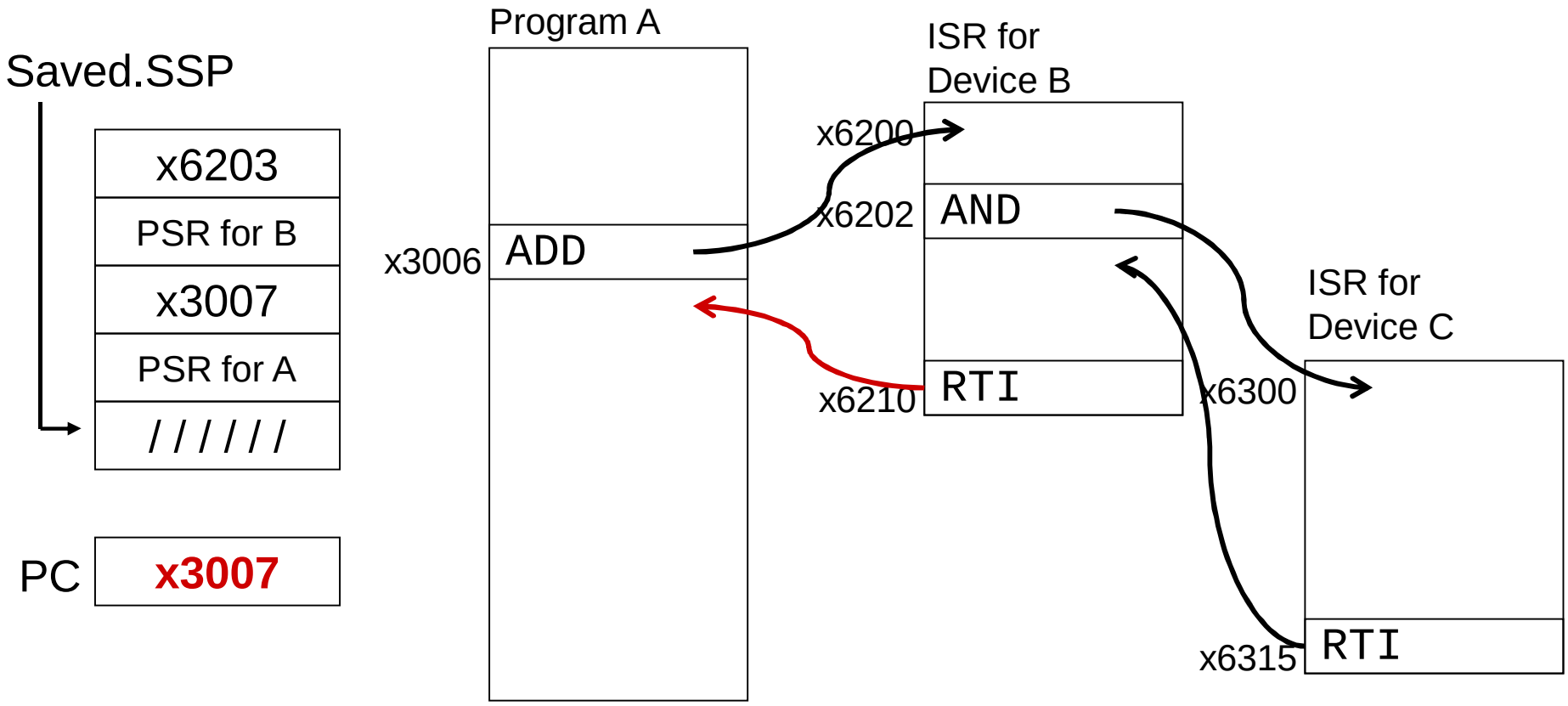
运行设备B的中断服务程序 (在 x6300).

Example (5)



执行RTI at x6315; 从栈pop PC and PSR

Example (6)



执行RTI at x6210; 从栈恢复 PSR and PC
恢复 R6. 继续执行程序A,好像什么也没有发生.

中断程序的设计-实验

本实验的目的是展示如何让输入输出通过执行中断处理程序的方式来暂停和恢复一个正在运行的程序，恢复后的程序就像中间什么都没有发生过，本实验使用键盘作为输入来中断正在运行的程序。程序包括以下三个部分：

一、用户程序：该程序持续间隔的输出两行不同的“ICS”，示例如下：

ICS ICS ICS ICS ICS ICS
ICS ICS ICS ICS ICS
ICS ICS ICS ICS ICS ICS
ICS ICS ICS ICS ICS
ICS ICS ICS ICS ICS ICS
ICS ICS ICS ICS ICS
ICS ICS ICS ICS ICS ICS
ICS ICS ICS ICS ICS

中断程序的设计-实验

二、键盘中断处理程序：该程序每次简单的把用户键入的回车(x0A)之前的字符打印10次。在中断处理程序中，TRAP指令是不能使用的，当需要显示字符时，必须通过读写DSR的方式，也不能用TRAP x21(OUT)和其他的TRAP指令。在中断处理程序中要对用到的寄存器的状态暂存和恢复。

三、操作系统使能代码：很不幸的是，LC-3上还不能安装Windows 和 Linux ，所以如下工作需要在用户程序中首先完成：

1、通常情况下，当遇到中断发生之前，操作系统已经开辟好栈空间，保存PC和PSR，当执行到RTI时，PC和PSR会被弹栈，因为没有操作系统，需要初始化R6为X3000，指示一个空栈。

2、同样，操作系统会建立一张中断向量表，用来包含中断处理程序对应的起始执行地址，因此，本实验的键盘中断处理程序需要你来做，中断向量表的起始地址为X0100，键盘中断处理程序的起始地址为X80，本实验只需要提供该中断处理程序的地址即可。

3、最后，操作系统会把KBSR的IE(Interrupt Enable) 位置1，所以你也需要这样做。

程序组成

用户程序

.ORIG x3000

;初始化

;1)R6<-x3000,初始化堆栈指针

;2)设置键盘中断矢量表条目,X0180对应键盘中断服务程序入口地址改为X2000

;3)启动键盘中断,KBSR【14】<-1

;用户程序

1）循环输出两行”ICS”

可以使用系统调用

Halt

.end

中断程序

.ORIG x2000

保存寄存器

….

恢复寄存器

RTI

Halt

.end

基于栈的算术运算

有些ISA使用栈来代替寄存器完成算术运算：零地址机

如：

ADD

该指令从栈弹出两个数并相加，把计算结果压入栈。

例：算术表达式

用栈来计算 (A+B)·(C+D) ：

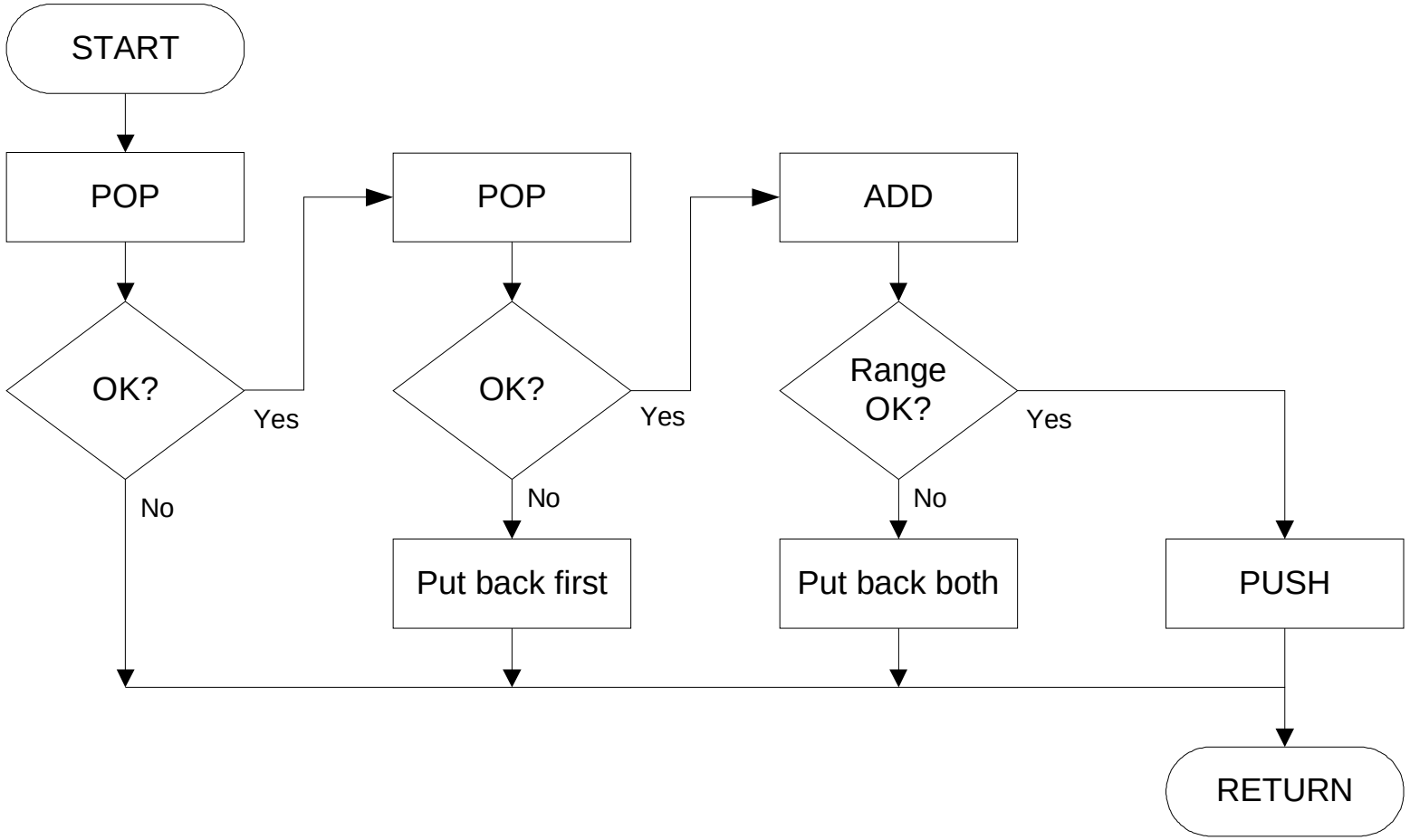
(1) push A
(2) push B
(3) ADD
(4) push C
(5) push D
(6) ADD
(7) MULTIPLY
(8) pop result

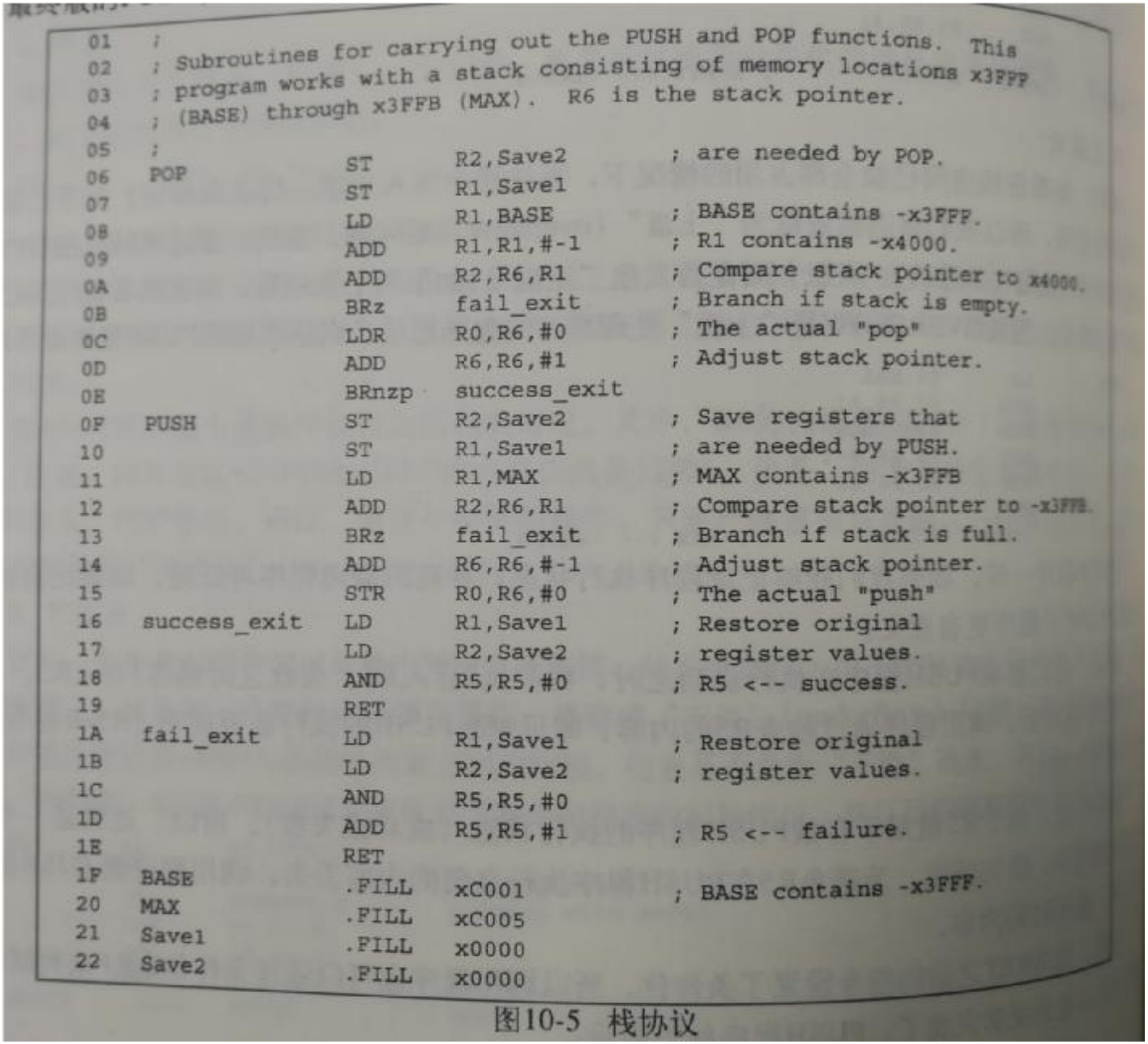
为什么使用栈?

- 寄存器个数有限
- 方便子程序调用
- 算法自然地用FIFO数据结构表达

基于栈的加法运算流程图

从栈里弹出两个值，相加，然后将计算结果压入栈。





```
01  ;
02  ; Subroutines for carrying out the PUSH and POP functions.  This
03  ; program works with a stack consisting of memory locations x3FFF
04  ; (BASE) through x3FFB (MAX).  R6 is the stack pointer.
05  ;
06  POP             ST      R2,Save2        ; are needed by POP.
07                  ST      R1,Save1
08                  LD      R1,BASE         ; BASE contains -x3FFF.
09                  ADD     R1,R1,#-1       ; R1 contains -x4000.
0A                  ADD     R2,R6,R1        ; Compare stack pointer to x4000.
0B                  BRz     fail_exit       ; Branch if stack is empty.
0C                  LDR     R0,R6,#0        ; The actual "pop"
0D                  ADD     R6,R6,#1        ; Adjust stack pointer.
0E                  BRnzp   success_exit
0F  PUSH            ST      R2,Save2        ; Save registers that
10                  ST      R1,Save1        ; are needed by PUSH.
11                  LD      R1,MAX          ; MAX contains -x3FFB
12                  ADD     R2,R6,R1        ; Compare stack pointer to -x3FFB
13                  BRz     fail_exit       ; Branch if stack is full.
14                  ADD     R6,R6,#-1       ; Adjust stack pointer.
15                  STR     R0,R6,#0        ; The actual "push"
16  success_exit    LD      R1,Save1        ; Restore original
17                  LD      R2,Save2        ; register values.
18                  AND     R5,R5,#0        ; R5 <-- success.
19                  RET
1A  fail_exit       LD      R1,Save1        ; Restore original
1B                  LD      R2,Save2        ; register values.
1C                  AND     R5,R5,#0
1D                  ADD     R5,R5,#1        ; R5 <-- failure.
1E                  RET
1F  BASE            .FILL   xC001           ; BASE contains -x3FFF.
20  MAX             .FILL   xC005
21  Save1           .FILL   x0000
22  Save2           .FILL   x0000
```

图10-5 栈协议

回顾栈程序

基于栈的加法运算程序

```
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.
            RET
Restore2    ADD R6,R6,#-1    ; Decr stack ptr (undo POP)
Restore1    ADD R6,R6,#-1    ; Decr stack ptr
Exit        RET
```

Range Check程序 （pp.178）

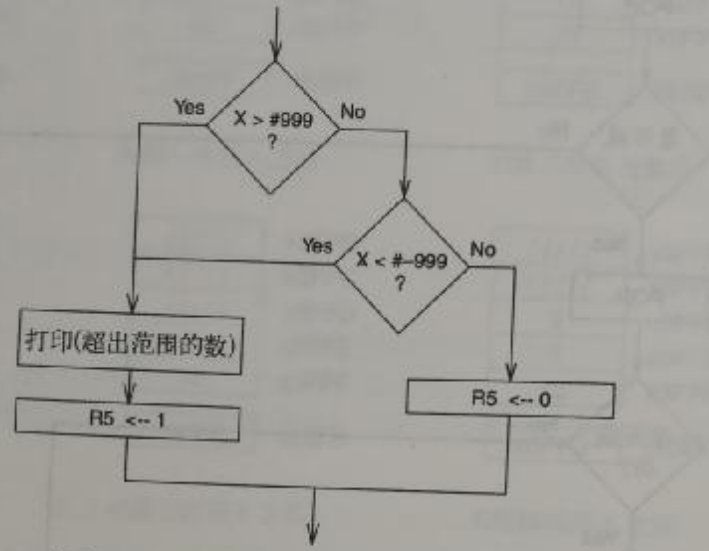


图10-11 范围检查函数的算法流程图

```
01   ;
02   ;     Routine to check that the magnitude of a value is
03   ;     between -999 and +999.
04   ;
05   RangeCheck    LD        R5,Neg999
06                 ADD       R4,R0,R5    ; Recall that R0 contains the
07                 BRp       BadRange    ; result being checked.
08                 LD        R5,Pos999
09                 ADD       R4,R0,R5
0A                 BRn       BadRange
0B                 AND       R5,R5,#0    ; R5 <-- success
0C                 RET
0D   BadRange      ST        R7,Save     ; R7 is needed by TRAP/RET.
0E                 LEA       R0,RangeErrorMsg
0F                 TRAP      x22         ; Output character string
10                 LD        R7,Save
11                 AND       R5,R5,#0    ;
12                 ADD       R5,R5,#1    ; R5 <-- failure
13                 RET
14   Neg999        .FILL     #-999
15   Pos999        .FILL     #999
16   Save          .FILL     x0000
17   RangeErrorMsg .FILL     x000A
18                 .STRINGZ  "Error: Number is out of range."
```

图10-12 范围检查程序

数据类型转换

键盘输入程序读取ASCII码（非二进制码）。

输出程序写ASCII码。

考虑如下程序：

```
TRAP  x23        ; input from keybd
ADD   R1, R0, #0  ; move to R1
TRAP  x23        ; input from keybd
ADD   R0, R1, R0  ; add two inputs
TRAP  x21        ; display result
TRAP  x25        ; HALT
```

输入 2 和 3 – 发生什么？

屏幕显示结果： e

为什么？ ASCII '2' (x32) + ASCII '3' (x33) = ASCII 'e' (x65)

ASCII 到 二进制

在处理多位数的数字时很有用

假设读了3位ASCII数到内存缓冲

x32	'2'
x35	'5'
x39	'9'

如何将其转换为ASCII码？

将第一位字符转换为数字并乘以100（200）；
将第二位字符转换为数字并乘以10（50）；
将第三位字符转换为数字（9）.
将三个数相加（259）

完整程序 （pp.183）

查表乘法

如何乘100?

1：自相加100次；

2：把100相加 <number> 次 (如果 number < 100更有效)

用单位数（ 0-9 ）作为查找表格的索引

Entry 0: 0 x 100 = 0
Entry 1: 1 x 100 = 100
Entry 2: 2 x 100 = 200
Entry 3: 3 x 100 = 300
etc.

查找表代码

```
; multiply R0 by 100, using lookup table
;
        LEA  R1, Lookup100  ; R1 = table base
        ADD  R1, R1, R0     ; add index (R0)
        LDR  R0, R1, #0     ; load from M[R1]
        ...
Lookup100  .FILL 0   ; entry 0
        .FILL 100 ; entry 1
        .FILL 200 ; entry 2
        .FILL 300 ; entry 3
        .FILL 400 ; entry 4
        .FILL 500 ; entry 5
        .FILL 600 ; entry 6
        .FILL 700 ; entry 7
        .FILL 800 ; entry 8
        .FILL 900 ; entry 9
```

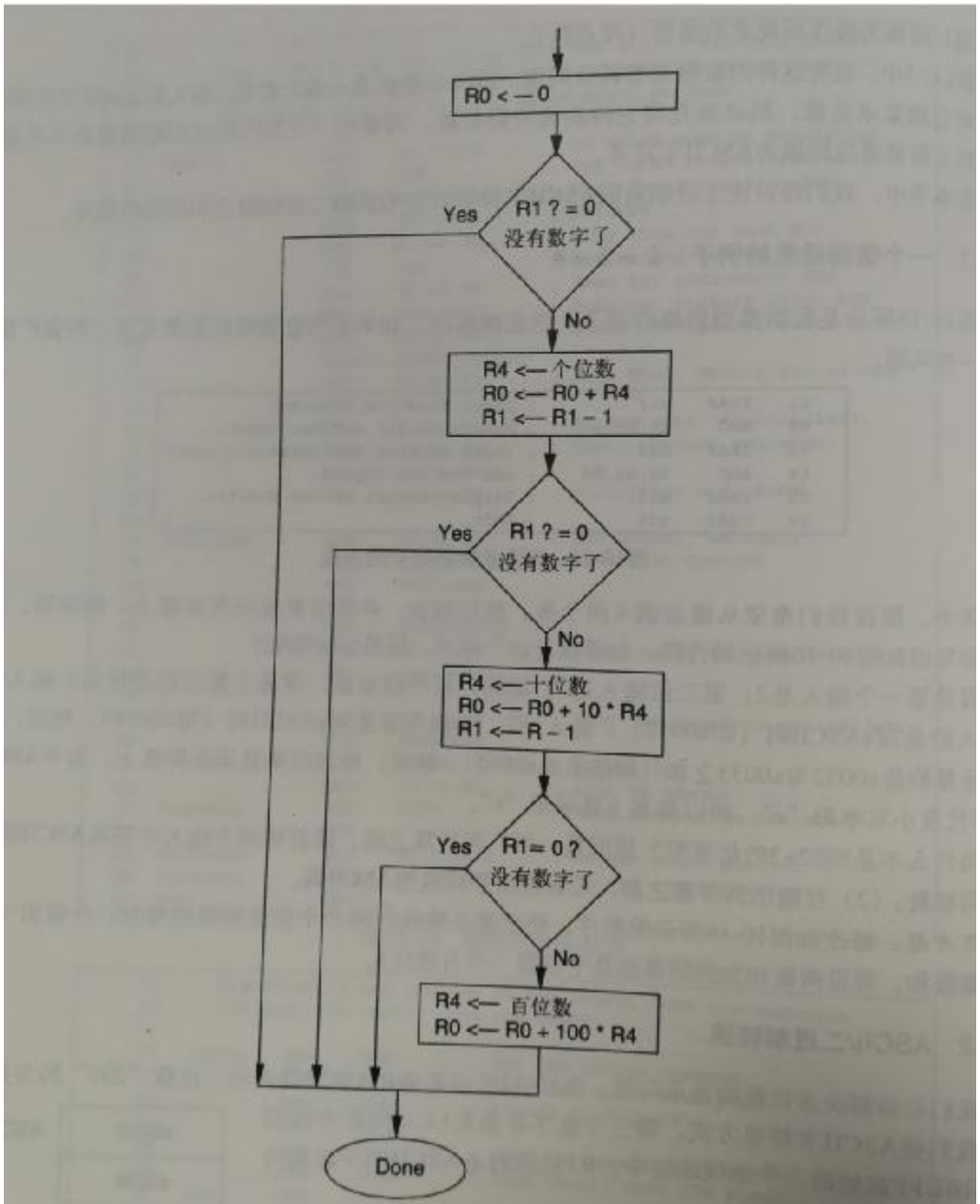


图10-18 将ASCII码转化为二进制的算法流程图

完整程序 （pp.184）

ASCII码到二进制的转换程序（1）

```
; Three-digit buffer at ASCIIBUF.
; R1 tells how many digits to convert.
; Put resulting decimal number in R0.
ASCIItoBinary  AND  R0, R0, #0  ; clear result
               ADD  R1, R1, #0  ; test # digits
               BRz  DoneAtoB    ; done if no digits
;
               LD   R3, NegZero ; R3 = -x30
               LEA  R2, ASCIIBUF
               ADD  R2, R2, R1
               ADD  R2, R2, #-1 ; points to ones digit
;
               LDR  R4, R2, #0  ; load digit
               ADD  R4, R4, R3  ; convert to number
               ADD  R0, R0, R4  ; add ones contrib
```

ASCII码到二进制的转换程序（2）

```
            ADD  R1, R1, #-1  ; one less digit
            BRz  DoneAtoB     ; done if zero
            ADD  R2, R2, #-1  ; points to tens digit
;
            LDR  R4, R2, #0   ; load digit
            ADD  R4, R4, R3   ; convert to number
            LEA  R5, Lookup10 ; multiply by 10
            ADD  R5, R5, R4
            LDR  R4, R5, #0
            ADD  R0, R0, R4   ; adds tens contrib
;
            ADD  R1, R1, #-1  ; one less digit
            BRz  DoneAtoB     ; done if zero
            ADD  R2, R2, #-1  ; points to hundreds
                              ; digit
```

ASCII码到二进制的转换程序（3）

```
            LDR  R4, R2, #0   ; load digit
            ADD  R4, R4, R3   ; convert to number
            LEA  R5, Lookup100 ; multiply by 100
            ADD  R5, R5, R4
            LDR  R4, R5, #0
            ADD  R0, R0, R4   ; adds 100's contrib
;
DoneAtoB      RET
NegZero      .FILL xFFD0  ; -x30
ASCIIBUF      .BLKW 4
Lookup10     .FILL 0
            .FILL 10
            .FILL 20
...
Lookup100    .FILL 0
            .FILL 100
...
```

二进制到ASCII码转换

将补码转换为包含3位数的ASCII码

这里需要除以100来得到百位数

这里为什么不能使用查找表?

重复地减100来做除法

转换开始前，需要判断该数的符号

将符号字符 (+ or -) 写入缓存，并将剩下的数变位正数

二进制到ASCII码的转换程序（1）

```
; R0 is between -999 and +999.
; Put sign character in ASCIIBUF, followed by three
; ASCII digit characters.

BinaryToASCII  LEA R1, ASCIIBUF  ; pt to result string
               ADD R0, R0, #0    ; test sign of value
               BRn NegSign
               LD  R2, ASCIIplus ; store '+'
               STR R2, R1, #0
               BRnzp  Begin100
NegSign        LD  R2, ASCIIneg  ; store '-'
               STR R2, R1, #0
               NOT R0, R0        ; convert value to pos
               ADD R0, R0, #1
```

二进制到ASCII码的转换程序（2）

```
Begin100      LD  R2, ASCIIoffset
              LD  R3, Neg100
Loop100       ADD R0, R0, R3
              BRn End100
              ADD R2, R2, #1  ; add one to digit
              BRnzp  Loop100
End100        STR R2, R1, #1  ; store ASCII 100's digit
              LD  R3, Pos100
              ADD R0, R0, R3  ; restore last subtract
;
              LD  R2, ASCIIoffset
              LD  R3, Neg10
Loop100       ADD R0, R0, R3
              BRn End10
              ADD R2, R2, #1  ; add one to digit
              BRnzp  Loop10
```

二进制到ASCII码的转换程序（3）

```
End10       STR R2, R1, #2  ; store ASCII 10's digit
            ADD R0, R0, #10 ; restore last subtract
;
            LD  R2, ASCIIoffset
            ADD R2, R2, R0  ; convert one's digit
            STR R2, R1, #3  ; store one's digit
            RET

;
ASCIIplus   .FILL x2B   ; plus sign
ASCIIneg    .FILL x2D   ; neg sign
ASCIIoffset .FILL x30   ; zero
Neg100      .FILL xFF9C ; -100
Pos100      .FILL #100
Neg10       .FILL xFFF6 ; -10
```

第十章简介

- 栈的基本结构
- 中断驱动I/O
- 基于栈的算术运算
- 数据类型转换
- 模拟计算器

模拟计算器允许的操作命令：

X 退出模拟器

D 显示栈顶元素

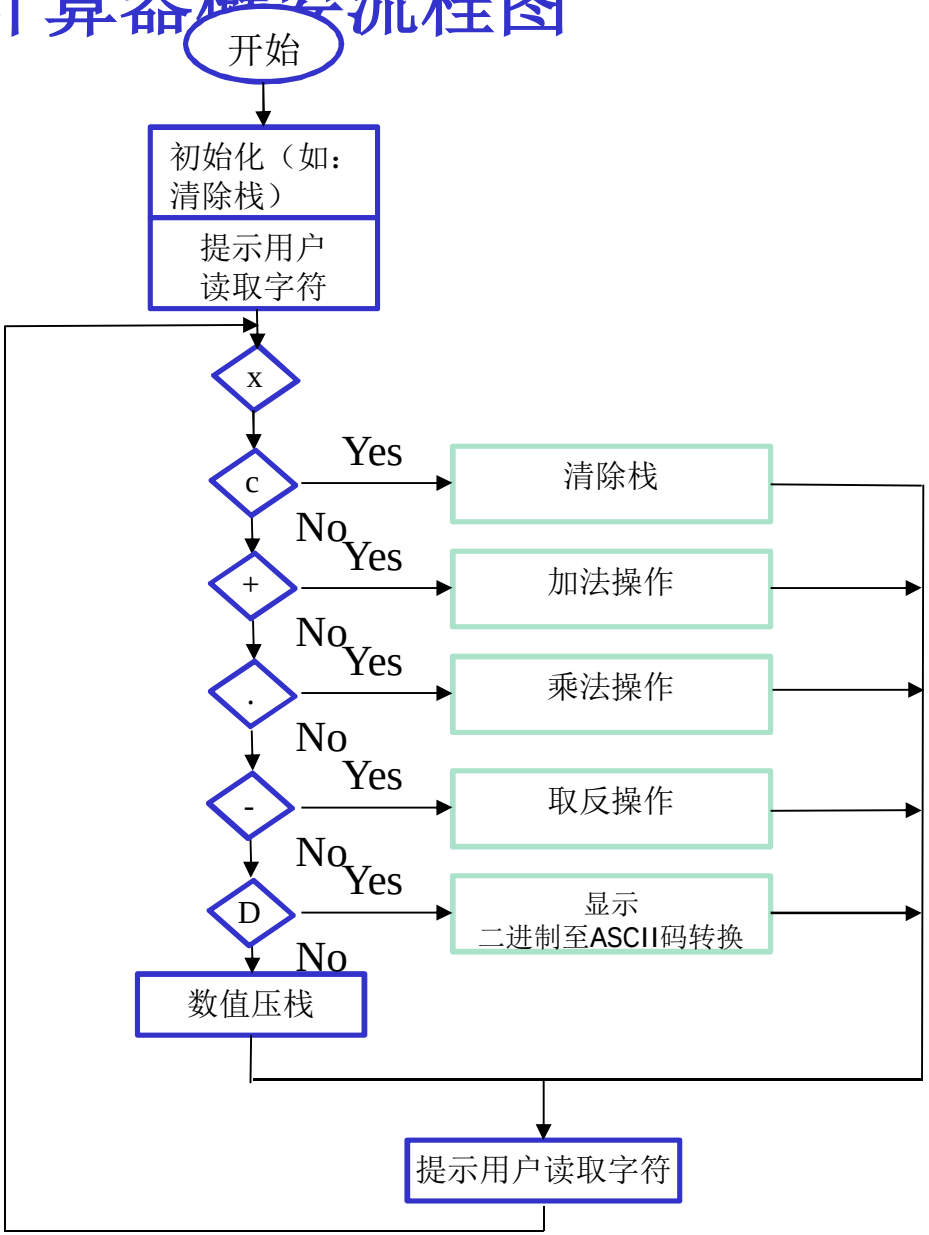
+ 将栈顶的两个元素求和，并替换栈顶

***** 将栈顶的两个元素求积，并替换栈顶

- 对栈顶元素取反

Enter 将键盘输入值压入栈

计算器概要流程图



模拟计算器允许的操作命令：

X 退出模拟器

D 显示栈顶元素

+ 将栈顶的两个元素求和，并替换栈顶

***** 将栈顶的两个元素求积，并替换栈顶

- 对栈顶元素取反

Enter 将键盘输入值压入栈

Ex 10.10

- **Ex 10.14 (advanced)**