CH8

String

书上8.5节，只需要知道若干$Ascii$码+末尾$/0$即可。

Structure

personnel record:

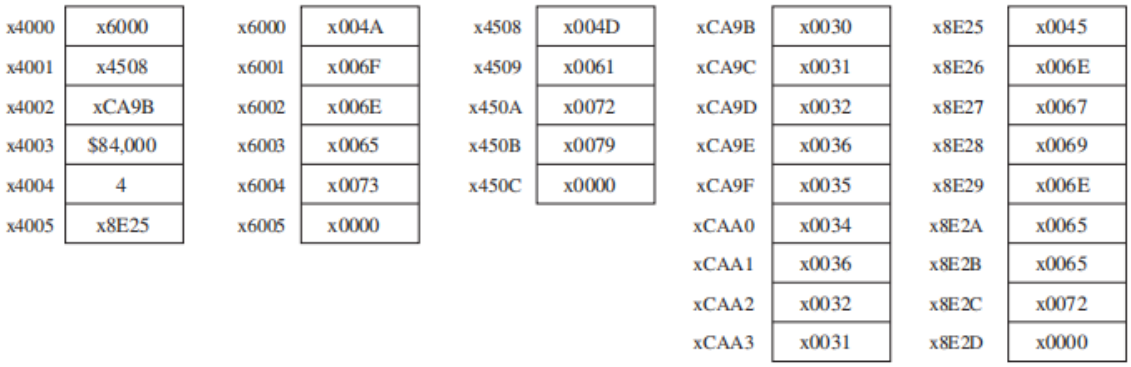


Figure 8.29 Mary Jones' personnel record.

Linked List

- **Access**

  > 数组：$O(1)$
  > 链表：$O(n)$

- **Update（广义上包括插入和删除）**

  > 数组：$O(n)$ for $insert$ and $delete$
  > 链表：$O(1)$

- **数组和$Linked\ List$的插入、删除、更新和查找的时间复杂度和具体过程**
- **即使维护一个有序的链表，时间复杂度也如上，想想为什么?**
- **Sequential Storage V.S. List Storage（CH19）**

Region of Memory

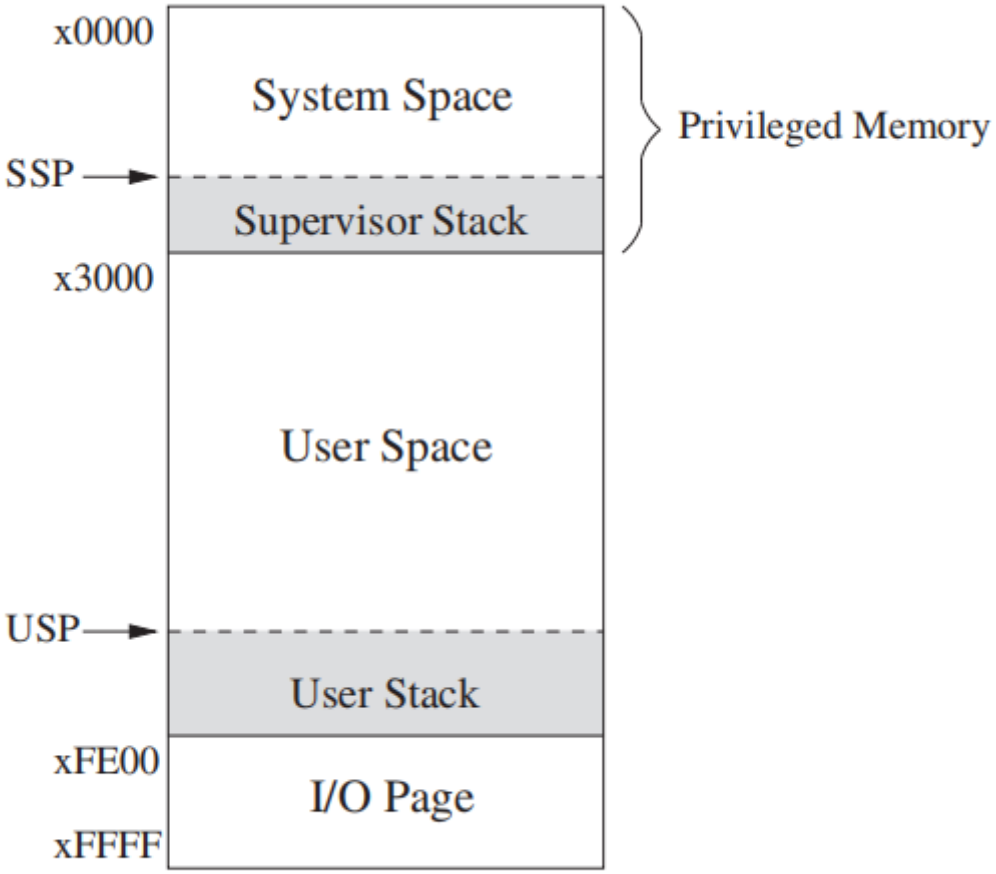


Figure 9.2 Regions of memory.

$Stack$从高地址开始，能够让$Stack$的容量受到的限制更小，上面的空间可能会被代码和数据所占据。

Stack

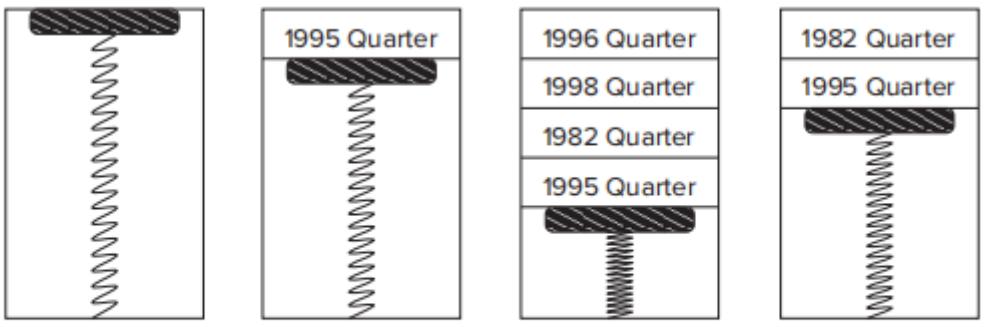


Figure 8.7 A coin holder in an automobile armrest–example of a stack.

We need to save a stack pointer（SP，Usually R_6）.

In $LC-3$, the stack grows to zero, which means that the stack will grows to lower location, for example, from $x3FFF$ to $x3FFFE$, $x3FFD$ and so on.

- **FILO（The defining notion of a stack）**

  LIFO, that is the last thing you stored in the stack is the first thing you remove from it.

- **Push（One of two typical operations on stack）**

```
PUSH ADD R6,R6,#-1
STR R0,R6,#0
```

- **Pop（One of two typical operations on stack）**

```
POP LDR R0,R6,#0
ADD R6,R6,#1
```

- **Overflow**

  超出容量，无法再进行$push$

```
PUSH AND R5,R5,#0
     LD R1,MAX
     ADD R2,R6,R1
     BRz Failure
     ADD R6,R6,#-1
     STR R0,R6,#0
     RET
     Failure ADD R5,R5,#1
     RET
     MAX .FILL xC005 ; MAX <-- -x3000
```

- **Underflow**

  容量为0，无法再进行pop

  假设$3FFB$~$3FFF$是合法的空间

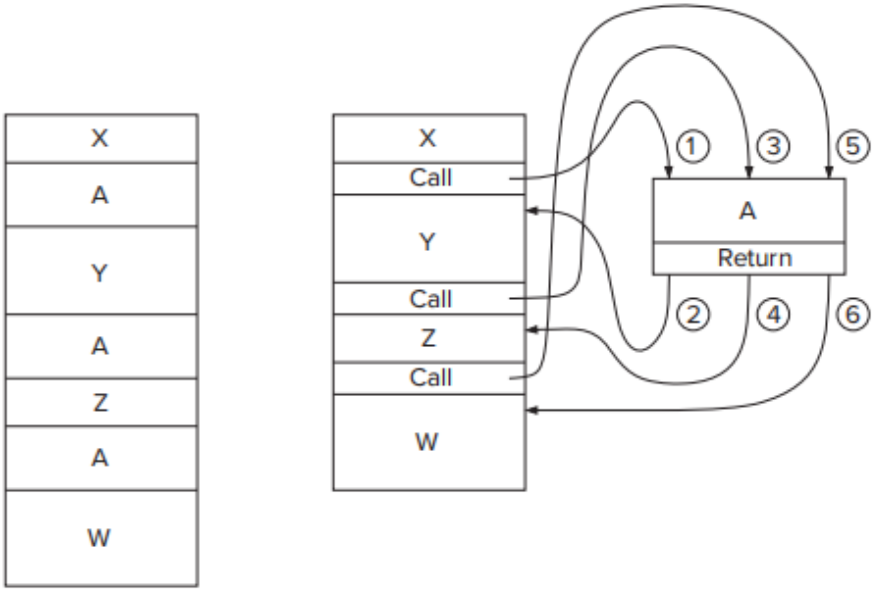
```
POP AND R5,R5,#0
    LD R1,EMPTY
    ADD R2,R6,R1
    BRz Failure
    LDR R0,R6,#0
    ADD R6,R6,#1
    RET
    Failure ADD R5,R5,#1
    RET
    EMPTY .FILL x0200 ; EMPTY <-- -xFE00
```

$Example$：用栈进行表达式计算$(6+4)*(3+2)$

Subroutine/Function/Procedure

对于重复多次使用的代码块，我们希望能够不用重复书写这些代码。To enable the programmer to write the code more efficiently.

- **Call/Return Mechanism**



(a) Without subroutines (b) With subroutines

Figure 8.2 Instruction execution flow with/without subroutines.

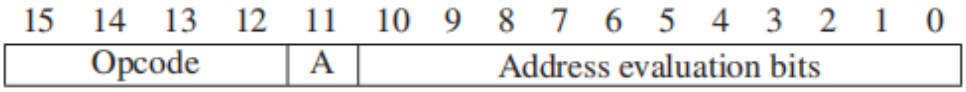
Call：跳转到函数F的开始位置

Return：回到正常执行的下一条指令，这个过程通过保存**R7**实现

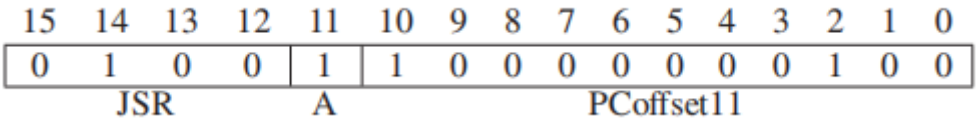
- **JSR & JSRR & RET**

Linkage：R_7**实现了Subroutine和主程序之间的连接**

- The JSR(R) instruction consists of three parts.



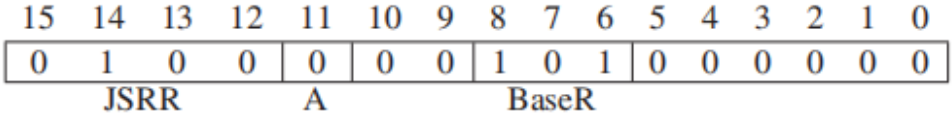
- JSR



$$R_7 \leftarrow PC$$

$$PC \leftarrow PC + PCoffset11$$

- JSRR



$$R_7 \leftarrow PC$$

$$PC \leftarrow R[BaseR]$$

P267$Question$：What important feature does the JSRR instruction provide that the

JSR instruction does not provide?

$JSRR$解除了距离限制，并且实现了类似函数指针的效果

- RET = JMP R7

$$PC \leftarrow R_7$$

- **Saving and Restoring Registers**

Why we need saving & restoring?

Every time an instruction loads a value into a register, the value that was previously in the register is **lost**

When we need saving & restoring?

The value will be destroyed by some subsequent instruction **and** we need it after that subsequent instruction.

$Callee\ Save$ OR $Callee\ Save$:

- Caller save：由调用者保存自己需要的值

```
Save
Save
JSR Function
Restore
Restore
```

- Callee save：由被调用者来保存自己需要改写的值

```
Function   Save
           Save
           ...
           ...
           Restore
           Restore
           RET
```

Use $Stack$ OR $Fixed\ Position$

- $Fixed\ Position$是不安全的
  - 多个函数调用（每个函数不同的$save$位置）
  - 递归（无法解决）
  - 因此通常用于$leaf\ procedure$
- $Stack$是最保险的
  - 非$leaf\ procedure$最好使用栈

Library Routines

Chapter 8.1.4 in the textbook is recommend reading.