LC-3 上栈的简单实现——求解 Hanoi 塔问题

林承宇 5100309007

January 17, 2011

Abstract

本文将粗略介绍 Hanoi 塔问题的一种递归解法，主要讲述这种解法在 LC-3 上如何用栈实现，以及在写汇编程序时的一些小细节。

1 什么是 Hanoi 塔？

传说中最早发明这个问题的人是法国数学家爱德华 · 卢卡斯。有一个传说是这样的：印度某间寺院有三根柱子，上串 64 个金盘。寺院里的僧侣依照一个古老的预言，以上述规则移动这些盘子；预言说当这些盘子移动完毕，世界就会灭亡。这个传说叫做梵天寺之塔问题。但不知道是卢卡斯自创的这个传说，还是他受其启发。

Hanoi 塔问题就由此而来，抽象一下，现在有 A、B、C 三根柱子，初始情况下 A 柱子上堆了 n 个大小各异的圆盘，从大到小依次叠放。每次可以将一根柱子最顶端的一个圆盘取下堆到另外一根柱子的最上层。大的圆盘不能置与小的圆盘上方。然后要求把 A 柱子上的所有圆盘移动到另外一根柱子上。

2 如何解决 Hanoi 塔问题

2.1 一个简单的想法

从缩小问题规模的想法来，假设现在有 N 个圆盘（编号为 1…N）要从 A 柱子移动到 C 柱子上，我们可以把前 N-1 个圆盘移动到 B 柱子上，然后把第 N 个圆盘移动到 C 柱子上，再把 B 柱子上的 N-1 个圆盘移动到 C 柱子上。由于在底部的盘子是最大的，所以在移动前 N-1 个盘子的时候可以忽略它，不会出现大的圆盘放置在小的圆盘上面。规模为 N 的问题就简化为规模为 N-1 的问题了。

2.2 C 语言的递归实现

就依照上面的想法，我们可以很轻松的利用 C 语言来递归的实现一个 Hanoi 塔问题求解程序：

```
void solve(int n, Stack *A, Stack *B, Stack *C) {
    //表示 n 个盘子从柱子 A 移动到柱子 C，利用中间柱子 B
```

```
    if (n == 1) {
        //只有一个盘子的情况，就直接处理掉了
        int x = A.pop();
        C.push(x);
        return;
    }
    solve(n - 1, A, C, B); //n-1 个盘子从 A 移动到 B
    int x = A.pop(); //第 n 个盘子移动到 C
    C.push(x);
    solve(n - 1, B, A, C); //n-1 个盘子从 B 移动到 C
}
```

2.3 在 LC-3 上用汇编语言实现

将 C 语言翻译成为 LC-3 上的汇编或许并不是难事，但是要实现一个递归调用的过程并不是那么直白的，因为在 LC-3 中首先并没有提供给我们一个完善的函数调用的环境。但是我们可以先写出一个简单的模型来：

```
; 约定 R3 寄存器代表 n
;     R4 寄存器代表柱子 A 的编号
;     R5 寄存器代表柱子 B 的编号
;     R6 寄存器代表柱子 C 的标号
SOLVE   ADD R3, R3, #0
        BRnz RETURN         ; 如果 n==0 就直接返回
        ; 调用 SOLVE R3-1, R4, R6, R5
        ; 移动过程（冗长了就不给出）
        ; 调用 SOLVE R3-1, R5, R4, R6
RETURN  ; 返回过程
```

现在的问题在于我们如何实现函数递归调用过程（移动过程还是很好实现的）。我们不可能对于每个 {R3,R4,R5,R6} 输入组合都写一个过程来，这个工作量太大了。我们想就用上面那个 SOLVE 程序块来实现整个功能。

一般程序语言中关于函数递归调用的地方，都是用栈这个工具来解决的。栈支持两种操作：从栈顶取出一个元素，往栈顶上压入一个元素。就像往一个口径和球一样大的试管里面塞球和取球一样，取球只能取最顶上的一个，塞球只能塞到所有管中球的上方。我们可以这么来解决上面调用的问题，在调用之前，先把现在程序块所运算出来的数据压入栈中，然后再给每个寄存器赋值为调用之后要用的数据，在调用完那个过程之后，再将原先压入栈中的数据弹出恢复原本的运算环境。由于栈的特性这并不破坏函数调用中数据的有序性。

这个可以这么来实现：

```
PUSH    STI R0, POINTER     ; 将 R0 中数据压入栈中
        LD R1, POINTER      ; 将栈顶指针往后移动一位
        ADD R1, R1, #1
        ST R1, POINTER
        RET
POP     LDI R0, POINTER     ; 将栈顶数据取出放在 R0 中
```

```
        LD R1, POINTER      ; 将栈顶指针后移一位
        ADD R1, R1, #-1
        ST R1, POINTER
        RET
POINTER .FILL STACK    ; 栈顶指针
STACK   .BLKW 400      ; 栈空间
```

模型中的; 调用 SOLVE R3-1, R4, R6, R5 这个部分就可以这么来实现：

```
LEA R0, (下一块地址)     ; 将调用完要返回的地址压入栈中
JSR PUSH
ADD R0, R3, #0
JSR PUSH
ADD R0, R4, #0
JSR PUSH
ADD R0, R5, #0
JSR PUSH
ADD R0, R6, #0
JSR PUSH
; 对参数重新赋值
BRnzp SOLVE
; 返回后执行语句
```

以及 RETURN 就可以这么写：

```
JSR POP
ADD R6, R0, #0         ; 顺序要注意
JSR POP
ADD R5, R0, #0
JSR POP
ADD R4, R0, #0
JSR POP
ADD R3, R0, #0
JSR POP
JMP R0                 ; 跳到返回地址
```