lab_report

PB24081571 刘沁昕

实验内容及其目标

利用LC-3汇编代码，实现三个函数 `s(i,j)`、`r(i,j)` 和 `p(i,j)` 的递归调用：

- `s(i,j)` 返回 `i+j`
- `r(i,j)` 递归计算：若 `i==0` 或 `j==0` 则返回1，否则返回 `r(i-1,j)+r(i,j-1)`
- `p(i,j)` 返回 `5*r(i,j)-s(i,j)`

要求使用递归栈的形式，即用 R6 作为栈顶指针，从内存地址 x3100 和 x3101 读取 x 和 y 的值，计算 `p(x,y)` 并将结果存储到 x3200。

完整代码

```
.ORIG x3000

; --- Main Program ---
; Initialize Stack Pointer (R6)
LD R6, STACK_START
; Initialize Frame Pointer (R5) to 0 to avoid simulator error
AND R5, R5, #0

; Load x and y
LDI R0, ADDR_X      ; Load x from x3100
LDI R1, ADDR_Y      ; Load y from x3101

; Call p(x, y)
; Push Space for Return Value
ADD R6, R6, #-1

; Push y (2nd argument)
ADD R6, R6, #-1
STR R1, R6, #0

; Push x (1st argument)
ADD R6, R6, #-1
STR R0, R6, #0

; Call p
JSR P_FUNC

; Pop Return Value and Arguments
LDR R0, R6, #2      ; Load return value (at R6+2)
ADD R6, R6, #3      ; Pop Arg1, Arg2, RV

; Store Result
```

```
STI R0, ADDR_RES    ; Store result to x3200

HALT

; --- Data for Main ---
STACK_START .FILL x4000
ADDR_X      .FILL x3100
ADDR_Y      .FILL x3101
ADDR_RES    .FILL x3200

; --- Function s(i, j) ---
; Returns i + j
; Stack Frame:
; R5+0: Saved R5
; R5+1: Saved R7
; R5+2: Arg1 (i)
; R5+3: Arg2 (j)
; R5+4: RV
S_FUNC
    ; Prologue
    ADD R6, R6, #-1
    STR R7, R6, #0      ; Push R7
    ADD R6, R6, #-1
    STR R5, R6, #0      ; Push R5
    ADD R5, R6, #0      ; Set Frame Pointer

    ; Body
    LDR R0, R5, #2      ; Load i (Arg 1)
    LDR R1, R5, #3      ; Load j (Arg 2)
    ADD R0, R0, R1      ; R0 = i + j

    ; Store Return Value
    STR R0, R5, #4      ; Store at RV slot (R5 + 4)

    ; Epilogue
    LDR R5, R6, #0      ; Pop R5
    ADD R6, R6, #1
    LDR R7, R6, #0      ; Pop R7
    ADD R6, R6, #1
    RET

; --- Function r(i, j) ---
; Returns 1 if i==0 or j==0, else r(i-1, j) + r(i, j-1)
R_FUNC
    ; Prologue
    ADD R6, R6, #-1
    STR R7, R6, #0
    ADD R6, R6, #-1
    STR R5, R6, #0
    ADD R5, R6, #0

    ; Check Base Cases
```

```
    LDR R0, R5, #2      ; Load i
    BRz BASE_CASE       ; if i == 0, return 1
    LDR R1, R5, #3      ; Load j
    BRz BASE_CASE       ; if j == 0, return 1

    ; Recursive Step
    ; Call r(i-1, j)
    ADD R6, R6, #-1     ; Push Space for RV
    LDR R1, R5, #3      ; Load j
    ADD R6, R6, #-1
    STR R1, R6, #0      ; Push j
    LDR R0, R5, #2      ; Load i
    ADD R0, R0, #-1     ; i - 1
    ADD R6, R6, #-1
    STR R0, R6, #0      ; Push i-1
    JSR R_FUNC
    LDR R2, R6, #2      ; Load result of r(i-1, j) into R2
    ADD R6, R6, #3      ; Pop RV, Arg1, Arg2

    ; Save R2 (result of first call) on stack
    ADD R6, R6, #-1
    STR R2, R6, #0

    ; Call r(i, j-1)
    ADD R6, R6, #-1     ; Push Space for RV
    LDR R1, R5, #3      ; Load j
    ADD R1, R1, #-1     ; j - 1
    ADD R6, R6, #-1
    STR R1, R6, #0      ; Push j-1
    LDR R0, R5, #2      ; Load i
    ADD R6, R6, #-1
    STR R0, R6, #0      ; Push i
    JSR R_FUNC
    LDR R3, R6, #2      ; Load result of r(i, j-1) into R3
    ADD R6, R6, #3      ; Pop RV, Arg1, Arg2

    ; Restore R2
    LDR R2, R6, #0
    ADD R6, R6, #1

    ; Add results
    ADD R0, R2, R3      ; R0 = r(i-1, j) + r(i, j-1)
    STR R0, R5, #4      ; Store Return Value
    BR TEARDOWN_R

BASE_CASE
    AND R0, R0, #0
    ADD R0, R0, #1      ; R0 = 1
    STR R0, R5, #4      ; Store Return Value

TEARDOWN_R
    ; Epilogue
```

```
    LDR R5, R6, #0
    ADD R6, R6, #1
    LDR R7, R6, #0
    ADD R6, R6, #1
    RET

; --- Function p(i, j) ---
; Returns 5*r(i, j) - s(i, j)
P_FUNC
    ; Prologue
    ADD R6, R6, #-1
    STR R7, R6, #0
    ADD R6, R6, #-1
    STR R5, R6, #0
    ADD R5, R6, #0

    ; Call r(i, j)
    ADD R6, R6, #-1     ; Push Space for RV
    LDR R1, R5, #3      ; Load j
    ADD R6, R6, #-1
    STR R1, R6, #0      ; Push j
    LDR R0, R5, #2      ; Load i
    ADD R6, R6, #-1
    STR R0, R6, #0      ; Push i
    JSR R_FUNC
    LDR R0, R6, #2      ; Load result r(i, j)
    ADD R6, R6, #3      ; Pop RV, args

    ; Calculate 5 * r(i, j)
    ADD R1, R0, R0      ; 2*r
    ADD R1, R1, R1      ; 4*r
    ADD R0, R1, R0      ; 5*r

    ; Save 5*r on stack
    ADD R6, R6, #-1
    STR R0, R6, #0

    ; Call s(i, j)
    ADD R6, R6, #-1     ; Push Space for RV
    LDR R1, R5, #3      ; Load j
    ADD R6, R6, #-1
    STR R1, R6, #0      ; Push j
    LDR R0, R5, #2      ; Load i
    ADD R6, R6, #-1
    STR R0, R6, #0      ; Push i
    JSR S_FUNC
    LDR R1, R6, #2      ; Load result s(i, j)
    ADD R6, R6, #3      ; Pop RV, args

    ; Restore 5*r
    LDR R0, R6, #0
    ADD R6, R6, #1
```

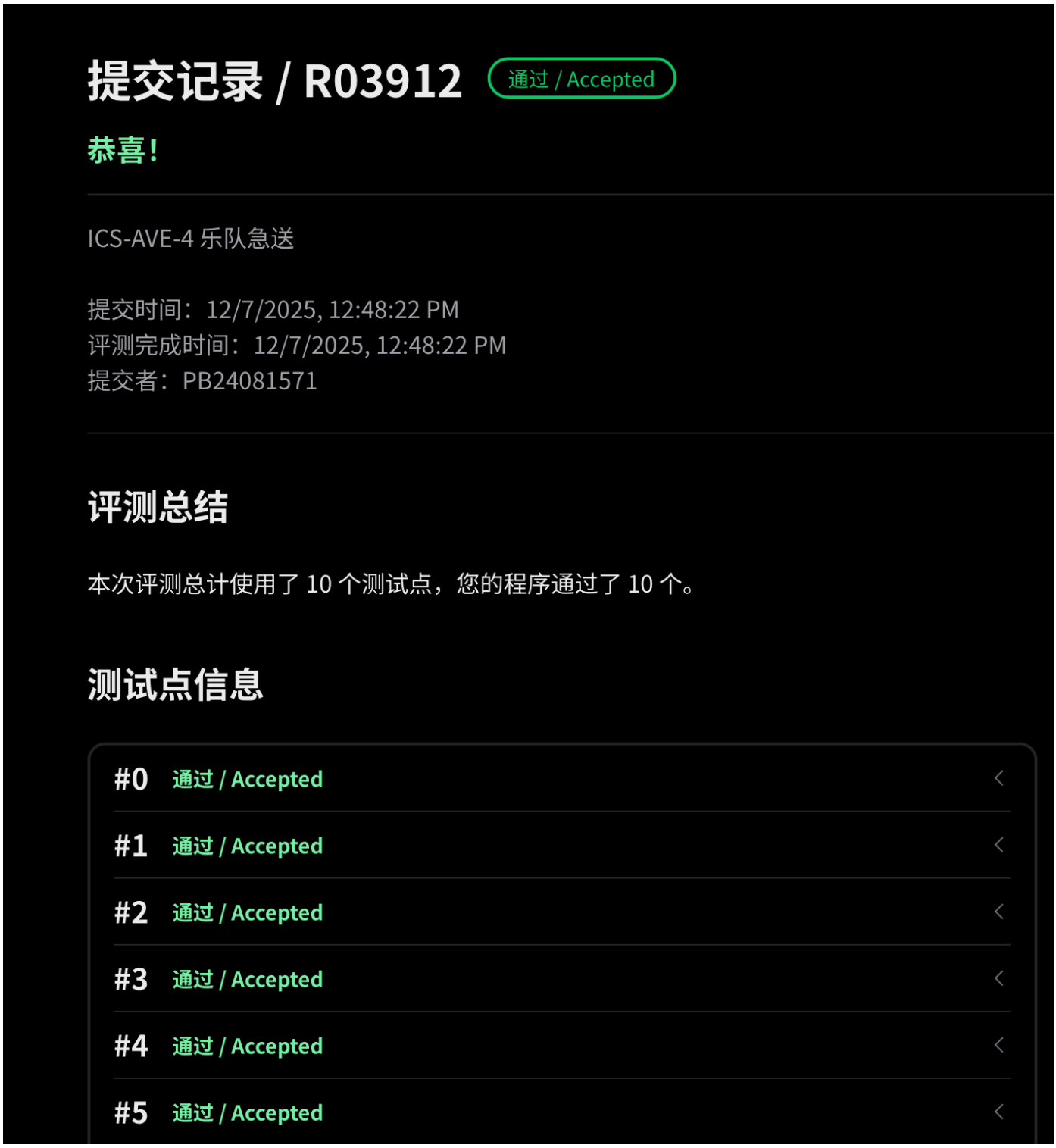
```
    ; Calculate 5*r - s
    NOT R1, R1
    ADD R1, R1, #1      ; -s
    ADD R0, R0, R1      ; 5*r - s

    ; Store Return Value
    STR R0, R5, #4

    ; Epilogue
    LDR R5, R6, #0
    ADD R6, R6, #1
    LDR R7, R6, #0
    ADD R6, R6, #1
    RET

.END
```

运行结果



在LC-3评测姬运行，所有测试点均通过

设计思路

栈帧结构设计

采用标准的LC-3调用约定，栈帧结构如下：

```
高地址
+-------------------+
|   Return Value    | R5+4
+-------------------+
|   Arg2 (j)        | R5+3
+-------------------+
|   Arg1 (i)        | R5+2
+-------------------+
|   Saved R7        | R5+1
+-------------------+
|   Saved R5        | R5+0 (Frame Pointer)
+-------------------+
低地址（栈顶方向）
```

函数实现策略

1. **函数 s(i,j)**：简单加法运算
    - 从栈帧读取参数 i 和 j
    - 计算 i+j
    - 将结果存储到返回值位置
2. **函数 r(i,j)**：双递归实现
    - 基准情况：i0 或 j0 时返回1
    - 递归步骤：
        - 调用 r(i-1, j)，保存结果到栈
        - 调用 r(i, j-1)，获取结果
        - 将两个结果相加后返回
3. **函数 p(i,j)**：组合调用
    - 调用 r(i,j) 获取结果，计算 5*r (通过 2r+2r+r 实现)
    - 调用 s(i,j) 获取结果
    - 计算 5*r - s 并返回

关键点

- **栈指针**：使用 R6 作为栈顶指针，向下增长
- **帧指针**：使用 R5 保存当前栈帧基址
- **寄存器保存与恢复**：在函数序言和尾声中正确保存/恢复 R5 和 R7
- **参数传递**：按照从右到左的顺序将参数压栈
- **返回值传递**：通过栈帧中的返回值位置传递

调试过程中遇到的问题

1. **R5 未初始化错误**

- 问题：LC-3 模拟器报错"Register R5 not initialized"
- 解决：在主程序开始处添加 `AND R5, R5, #0` 初始化 R5

2. **栈帧偏移量计算错误**
   - 问题：初次尝试时参数和返回值的偏移量计算有误（off-by-one error）
   - 原因：混淆了栈的增长方向和偏移量的关系
   - 解决：重新梳理栈帧结构，修正所有偏移量
     - 参数 i 的偏移量：R5+2
     - 参数 j 的偏移量：R5+3
     - 返回值的偏移量：R5+4
3. **返回值读取位置错误**
   - 问题：调用者在函数返回后无法正确获取返回值
   - 解决：将 `LDR R0, R6, #0` 改为 `LDR R0, R6, #2`，因为返回值在栈顶上方两个位置
4. **递归调用中寄存器冲突**
   - 问题：r(i,j) 两次递归调用之间，第一次的结果被覆盖
   - 解决：在第二次递归调用前，将第一次的结果暂存到栈上

收获

1. 深入理解了 LC-3 的函数调用约定和栈帧结构
2. 掌握了递归函数在汇编层面的实现方法，包括：
   - 基准情况和递归步骤的处理
   - 多次递归调用时的现场保护
   - 栈空间的动态管理
3. 提升了汇编代码的调试能力，学会了：
   - 使用模拟器追踪栈的变化
   - 通过内存查看验证栈帧结构
   - 定位并修复偏移量相关的错误
4. 加深了对高级语言函数调用机制的理解，认识到编译器在栈管理、参数传递等方面做的工作