Lab3 实验报告

王嵘晟

PB17111614

1. 程序区别

1.1 函数

Lab2 中提交的我自己写的 LC3 程序是通过提前把寄存器 R6 和 R5 分别存储栈指针和帧指针，main 函数中除了实现 C 语言中 main 函数的本来功能：读入 n 的值并传参调用 func 函数以外，还将 n 的值以及 a,b,c,d,e,f 的值给压入栈中，并移动栈指针。但压栈过程是压入元素结束后一起移动栈指针，栈指针直接减去压入元素个数（这么写逻辑有点乱）而 func 函数通过栈的存储模式实现递归调用

```
        .ORIG x3000
Start   LD R6,Addr          ;Stack Pointer
        LD R4,MINUS_ASCII
        JSR main
        LDR R0,R6,#0
        HALT
```

```
main    ADD R6,R6,#-1
        STR R7,R6,#0        ;Store the address to return
        ADD R6,R6,#-1
        STR R5,R6,#0
        ADD R5,R6,#-1
        ADD R6,R6,#-2
        TRAP x23
        ADD R0,R0,R4
        STR R0,R5,#0       ;Push n into the stack

        AND R2,R2,#0
        STR R2,R6,#0
        STR R2,R6,#-1
        STR R2,R6,#-2
        STR R2,R6,#-3
        STR R2,R6,#-4
        STR R2,R6,#-5     ;Push a,b,c,d,e,f into the stack (They are all zero in main)
        STR R0,R6,#-6
        ADD R6,R6,#-6

        JSR Func
        LDR R0,R6,#-1
        STR R0,R5,#3
        ADD R6,R5,#3
        LDR R5,R5,#1       ;Frame pointer
        LDR R7,R6,#-1      ;Return address
        RET
```

而 compiler 出来的 LC3 程序同样把 R6 作为栈指针，但没有用到帧指针。Main 函数初始化了栈的存储，每压入一个元素，栈指针-1，逻辑清晰。Func 函数的实现逻辑和手写函数基本一致。

```
1    .Orig x3000
2    INIT_CODE
3    LEA R6, #-1
4    ADD R5, R6, #0
5    ADD R6, R6, R6
6    ADD R6, R6, R6
7    ADD R6, R6, R5
8    ADD R6, R6, #-1
9    ADD R5, R5, R5
10   ADD R5, R6, #0
11   LD R4, GLOBAL_DATA_POINTER
12   LD R7, GLOBAL_MAIN_POINTER
13   LD R0, GLOBAL_MAIN_POINTER
14   jsrr R0
15   HALT
```

```
187  main
188  ADD R6, R6, #-2
189  STR R7, R6, #0
190  ADD R6, R6, #-1
191  STR R5, R6, #0
192  ADD R5, R6, #-1
193
194  ADD R6, R6, #-2
195  ADD R0, R4, #4
196  LDR R0, R0, #0
197  jsrr R0
198  LDR R7, R6, #0
199  ADD R6, R6, #1
200  ADD R3, R4, #8
201  ldr R3, R3, #0
202  ADD R6, R6, #-1
203  STR R0, R6, #0
204  ADD R6, R6, #-1
205  STR R3, R6, #0
206  NOT R3, R3
207  ADD R3, R3, #1
208  ADD R0, R7, R3
209  LDR R3, R6, #0
210  ADD R6, R6, #1
211  ADD R7, R0, #0
212  LDR R0, R6, #0
213  ADD R6, R6, #1
214  str R7, R5, #0
215  ADD R7, R4, #5
216  ldr R7, R7, #0
217  ADD R6, R6, #-1
218  STR R7, R6, #0
```

其他区别: 手写函数跳转使用的是 JSR 和 BR, 而编译器生成函数多用 JSRR 和 JMP。

1.2 prolog 和 epilog（“保存现场”和“恢复现场”）

两个程序都是通过把栈指针和帧指针的值存入栈里，以实现“保存”

而恢复过程就是把栈指针和帧指针重新从栈里读取出来并存入 R6 R5，完成“恢复”，

并通过对寄存器内容的逻辑运算将栈指针指向存入最终结果的位置，把它存入 R0

1.3 栈操作

Lab2 中的手写程序制定了栈底的位置在 xFFD0,直接 LD 得到

而 compiler 出来的 LC3 程序通过“巧妙”的逻辑运算实现栈底位置在 xEFFF。

Lab2 程序的栈的结构为最底层为栈指针 R6，紧接着是帧指针 R5，然后是 n 的值，然后是最终结果的值，然后是 a,b,c,d,e,f 的值。

而 compiler 出来的 LC3 程序最底层为最终结果，然后是栈指针 R6，然后是 n 的值然后是 a,b,c,d,e,f 的值。

2. Pros and cons（优势与劣势）

Lab2 中的手写程序代码行数更少，但是逻辑不够清晰。

编译器编译出来的程序代码代码行数多，但是逻辑清晰。编译器编译出来的程序对寄存器的使用个数更少。

3. 整数调用机制是怎么样的?

整数调用机制基于八个寄存器 R0~R7，前两个寄存器也多用于返回数值。最多 XLEN 位宽的标量在单个参数寄存器中传递，如果没有，则在值堆栈中传递。 当传入寄存器时，比 XLEN 位窄的标量根据其类型的符号加宽，最多为 32 位，然后符号扩展为 XLEN 位。宽度为 2×XLEN 的标量在一对参数寄存器中传递，如果没有，则在值堆栈中传递。如果只有一个寄存器可用，则低位 XLEN 位在寄存器中传递，高位 XLEN 位在堆栈上传递。

在基本整数调用约定中，可变参数以与命名参数相同的方式传递，但有一个例外。具有 2×XLEN 位对齐和大小最多 2×XLEN 位的变量参数在对齐的寄存器对中传递(即，该对中的第一个寄存器是偶数编号的)，或者如果没有可用则在值堆栈上传递。 在堆栈上传递了可变参数之后，所有将来的参数也将在堆栈上传递（即，由于对齐的寄存器对规则，最后一个参数寄存器可能未被使用)。