LAB2-Report

- 姓名：魏剑宇
- 学号：PB17111586

> Think as compiler

编译器名为c2lc42lc3

C Runtime

调用Main函数之前，需进行一些初始化过程

```
    LD R6, ebp
    JSR main
; ......
ebp .FILL xFFDF
```

这里，只需设置一下栈底的指针。在考虑程序栈的设计时，进行了一些思索，最后选择了下面的设计。

About Stack

考虑x86中，有bp和sp两个寄存器，分别存放当前的栈底和栈顶。一般而言使用一个栈顶指针已经足够，但增加栈顶指针能够方便程序的设计。

在这里，经过权衡，最终选择只有栈顶指针的设计方案。由于LC-3寄存器资源稀缺，且只是用栈顶指针在这里并不会带来较大不便。（实际上在c/c++的编译器生成的代码中常常也只是用(e/r)sp来获取栈中的变量）。

这里我使用R6来存储栈顶。

Calling Conventions

- 通过JSR调用函数
- function prologue

  ```
  PUSH R7
  ```

  ※：*这里，PUSH是LC-4（created by wjy）中的指令，后面附有翻译程序将LC-4语言转化为LC-3语言.*
- function epilogue

  ```
  POP R7
  RET
  ```

  ※：*同上*
- Caller clean-up。所以在调用完函数后可能需要进行栈的清理。

- 返回值（如果有）存在R0中
- Caller-saved registers：R0-R4
- Callee-saved registers：R5-R7
- parameters
  - **R**ight **T**o **L**eft
  - 前四个参数通过R0-R3，如果有更多，推入栈中
- 部分本地变量存在栈中

About LC-4

新增了两条指令，PUSH和POP。为了在LC-3这种陈旧的语言中也能使用，下面是翻译器，将其翻译为PUSH和POP在LC-3中的fallback。

lc42lc3 translator

```
# lc42lc3.py
import re
import sys

push_pattern = re.compile(r'^([a-z]*\s+)PUSH R([0-7])', re.MULTILINE)
push_pattern2 = re.compile(r'^([a-z]*\s+)PUSH ([0-9]+)', re.MULTILINE)
pop_pattern = re.compile(r'^([a-z]*\s+)POP R([0-7])', re.MULTILINE)

with open(sys.argv[1], 'r') as in_file:
    with open(sys.argv[2], 'w') as out_file:
        strz = in_file.read()
        strz = re.sub(push_pattern,
                r'\1ADD R6, R6, #-1\n        '
                r'STR R\2, R6, #0',
                strz)
        strz = re.sub(pop_pattern,
                r'\1LDR R\2, R6, #0\n        '
                r'ADD R6, R6, #1',
                strz)
        strz = re.sub(push_pattern2,
                r'\1ADD R6, R6, #-1\n        '
                r'AND R5, R5, #0\n        '
                r'ADD R5, R5, #\2\n        '
                r'STR R5, R6, #0',
                strz)
        out_file.write(strz)
```

只是简单地将PUSH和POP内联（:

不使用subroutine的原因是将PUSH和POP当作subroutine来使用**使我感到不适。**

实际上我的编译器也不会生成任何subroutine。

（在翻译PUSH字面值时比较尴尬，因为LC-3没有将一块内存存入字面值的指令）

Other Design

- c2lc42lc3将抛弃subroutine的概念，使用function（虽然在指令形式上是相同的）。所以c2lc42lc3不会生成任何subroutine。
- 使用PUSH和POP指令进行栈的控制，如上所述。
- trap routines不属于callee。除了能通过R0返回值外，还需保持Caller-saved registers的值不变，以及程序的状态，包括FLAGS。
- 由于程序的地址是随机加载的，而lc-3目前不存在VA这样的机制，故在实验中保证只使用PC相对寻址。
- callee-saved registers暂存在栈中。

Error Handling

在设计error handling时考虑了许多，主要考虑有哪些错误需要处理，以及能够如何处理。

能够处理且有必要处理的异常只有stack overflow和stack underflow。这里的overflow和underflow指的是超过了栈的总容量，而不是就某一个函数而言。书上对于此两种异常的处理给出了一个例子，但感觉与c语言不太契合，且对于编译器而言不太好实现一个自然的错误处理。主要有以下几点：

1. 观察x86中，（非软件上的）overflow和underflow是由指令抛出的，这是更加底层的，在LC-3中没有对应，且LC-3中实际上没有实现异常。
2. 此种overflow和underflow应由processor和os来处理。
3. **最重要的是**，编译器不知道怎么处理这样的异常。这种异常发生在**函数的调用过程中**，哪怕和cpp的SEH也是不符的。如果函数调用失败该怎么处理呢？书上给出的那种处理是，在失败时什么也不会做，使用R5来返回状态，但这个R5**对编译器并没有帮助。**因为即使知道了失败，编译器所唯一能做的也只是**终止程序**，而不可能再调用一次此函数。而且此异常无法在C中被捕获，用户也无法告诉编译器该如何处理这个异常。
4. 在msvc、gcc等生成的代码中，直接add esp /sub esp 也是很正常的，没有使用push和pop。

综合考虑以上的情况，有一个最基础的错误处理的思路。即在发生溢出时将R0设为1标志失败，并且立刻中止程序。（其实这样也不太对，因为程序正确的情况下R0也可能会被置为1）。最终选择不进行栈相关的错误处理。

Optimizations

只考虑编译器能做的优化。

1. 部分传入的参数和函数内的变量可能不存在栈中，而直接保留在寄存器中（如果编译器能确保之后不会被使用）。
2. 在PUSH进行内联以后发现有优化空间，即只需要在一个寄存器中存0，就可用于所有的PUSH。
3. 对内联后的PUSH进行简单地优化，即合并ADD。

Test

同样，这次测试的代码也使用到liblc3。注意，代码中通过`state.input = &in`来设定输出，保证GETC时没有一直polling。结果如下（下面测试中每一次输入的char值相同，且指令数未经过加权）：

char	intrs
48	305
49	304
50	385
51	466
52	629
53	874
54	1283
55	1938
56	3003
57	4724
58	7511
.....	.....
69	1452504
70	2350075