LAB3-Report

姓名：魏剑宇

学号：PB17111586

※：导出的时候在h4的大纲显示上有一些bug

Compile

- **编译器**：lcc-lc3
- **环境**：WSL Ubuntu18.04
- **过程**：build后，输入`lcc lab2.c -o lab2`即可完成编译。

Registers

description

经过简单地分析，可以发现各寄存器发挥的功能有以下特点

- R6：栈顶（sp）
- R5：栈底（bp）
- R4：指向data segment（`GLOBAL_DATA_START`），在程序的末尾，该区域存有一些地址和常量数据。主要用于远距离的跳转和常量访问，用法一般为

  ```
  ADD R0/3/7 R4 DS:OFFSET address
  LDR R0/3/7 R0/3/7 #0
  ```

- R0, R3, R7：general purpose registers。被广泛用于地址访问和计算的中间值存储。其中R7比较特殊，因为它也被用于subroutine（function）的跳转。这是没问题的，因为在function的开始R7会被储存在栈中，结束时R7会被恢复。
- R1,R2：很少被使用，不会在（至少在这个程序中）**编译器生成的代码**中出现。只能在一些**手写的代码**中看到。例如scanf和printf函数的实现中。
- Callee-saved registers：R5，R7

pros and cons

pros	cons
这种设计是通用的，通过R4避免了远距离下无法相对寻址	增加了代码量，降低了效率（但这是完成此种general而又简单易实现的寻址不可避免的）。*虽然可以通过将每个函数中使用的数据放在附近，能相对寻址时就相对寻址来优化，但此种设计复杂较难实现，且将数据和代码混在一起不是一种好的设计模式。*
既设置栈顶也设置栈底简化了程序的设计，进行栈的操作时十分方便	增加了代码量（在大部分的程序中可忽略不计）
只有R0，R3，R7充当GPR，方便实现。	在一些较简单的函数中，部分寄存器未使用会降低效率

C Runtime

description

```
INIT_CODE
LEA R6, #-1
ADD R5, R6, #0
ADD R6, R6, R6
ADD R6, R6, R6
ADD R6, R6, R5
ADD R6, R6, #-1
ADD R5, R5, R5
ADD R5, R6, #0
LD R4, GLOBAL_DATA_POINTER
LD R7, GLOBAL_MAIN_POINTER
jsrr R7
```

按顺序完成以下工作

- 设置栈顶指针R6. R6 <- x3000 * 5 - 1 = xEFFF（其中通过*2实现乘法是一个简单的优化）
- 设置栈底指针R5
- 加载GLOBAL_DATA_START
- 加载main函数入口

发现赘余的代码`ADD R5, R5, R5`.

pros and cons

None

Calling Convention

description

和我一样没有进行多此一举的栈的错误处理（准确来说是没法做，分析在上次报告中）。

由于我和lcc-lc3设计的不同，calling convention在一些地方就有了区别，以下是对比

function prologue/ epilogue

lcc的开场白和收场白和gcc/ msvc等生成的更为相似，为使描述更加清晰，这里使用`push`和`pop`和`mov`

```
; prologue
push R7 ; push 返回地址
push R5 ; push ebp
mov R5 R6 ; mov ebp esp

; epilogue
pop R5 ; pop ebp
pop R7
ret
```

一般情况下，栈是长这样的

address	content
ebp + x	参数
...	参数
ebp + 3	参数
ebp + 2	返回地址
ebp + 1	原来的ebp
ebp	local variables
...	local variables

而我的c2lc42lc3由于只使用了一个栈顶指针

```
; prologue
push R7

; epilogue
pop R7
ret
```

arguments

- lcc的传参是RTL，和我的一样，也比较符合C的特点（即变量有可能给多）。
- lcc所有的参数通过栈传递，而我的部分参数（前4个）通过寄存器传递。

clean-up

和我的一样都是caller clean-up

Call

进行函数调用时，lcc所有都是通过`JSRR R0`.在跳转前，通过R4，在R0内存入函数的地址。

pros and cons

pros	cons
既设置栈顶也设置栈底简化了程序的设计，进行栈的操作时十分方便	增加了代码量（在大部分的程序中可忽略不计）
所有参数通过栈传递减少了寄存器的使用	所有参数通过栈传递效率较低
RTL的参数传递方式有利于灵活地传递参数和可变长参数的实现	——

x86-64 ABI

ABI是程序二进制接口，保证在兼容的设备上二进制库不用重新编译也能使用。

大部分的ABI的思路基本一样，下面只列出它们的部分值得关注的特点。

cdecl

和lcc-lc3的相似，也是C的标准做法（和用的最多的做法）

stdcall

和上面的一种主要的不同是callee clean-up。在funciton的结束一般用

```
ret n  ; note: not retn
```

来清理栈空间。

fastcall

如名字所说是一种高效的调用方式（只在老式的没怎么优化的cpu上有优势），主要的特点是callee clean-up，同样使用`ret n`。以及部分参数会通过寄存器传递。

thiscall

C++调用method时使用。this是指向该类的数据的指针。在MSVC中此指针通过ecx传递。

Windows x64

x64下只有一种调用模式。前4个参数中的整型值通过RCX，RDX，R8，R9传递。整型值（包括指针）通过RAX返回，而超过RAX width（64 bits）的值通过RDX:RAX返回。同时caller需要为callee准备4 * （32/64）bits的shadow space，这些空间用于存储通过寄存器传递的参数。（无论这些空间究竟有没有被使用）。

这样，就给程序留下了优化的空间。比较大的函数可能需要将这些参数存起来，而比较小的简单的函数可以直接通过寄存器获取这些参数。

在浮点数上面，前**四个参数中的**浮点数通过XMM0-3的对应位置上（例如，第二个参数是浮点数，则存在XMM1，无论XMM0是否已经被使用）传递，其它的通过栈传递。浮点数通过XMM0返回，超过64bits的值通过XMM1:XMM0返回。

（注意返回结构实际返回的是指向结构（或类）的指针。

System V AMD64

大部分与Windows x64相似，前六个参数通过RDI, RSI, RDX, RCX, R8, R9传递，浮点数通过XMM0-7传递，其它的参数通过栈传递。返回值与Windows x64一样。与Windows x64主要的不同是不会提供shadow space。

Q & A

Q：Why shadow space?

为什么win x64下要有shadow space呢？事实上shadow space不是必须的，gcc就没有这样的东西。无论是否使用shadow space都给它分配空间，无端地增加了栈的使用，降低了栈的利用率。既然这样，为什么要有shadow space呢?

A

微软是这样说的.

> This convention simplifies support for unprototyped C-language functions and vararg C/C++ functions.

非常简略，没有详细的解释。初接触时我一直没有弄清楚原因，后来看了许多汇编后渐渐有了一些理解。这里就不解释unprototyped和varadic function是什么了。

1. 保证了所有参数在栈中是连续的。

   首先要清楚，这些空间要么由caller来分配，要么由callee来分配。必须有人来分配。由callee分配则相对比较灵活，它可以根据自己是否要使用、要使用来多少来分配进行调整，但会导致参数在栈中不连续。中间会有栈的信息。这对可变参数的实现是不利的。

2. 实际并不会浪费栈空间。

   对shadow space的质疑主要在内存的浪费。认为它在带来一点点的方便时，浪费了栈空间。但实际上呢?

   首先，栈空间毕竟是动态的，在函数结束后就会释放，所以只要总的内存足够，这对空间的消耗实际上没有影响。而在x64环境下，内存已经很多了，4个字节对4GB以上的内存不会造成什么影响。而且我们也要清楚，栈这种东西，分配之后不用和不分配是一样的。更何况分配之后有可能会用到，不分配就一定会浪费了。

   另一个重要的原因是，实际上这部分栈空间没有被浪费，它们被利用了。姑且不谈Non-optimizing code[1]，optimizing code中，这些栈空间实际可能会被用来做各种事。包括存本地变量、以及保存callee-saved registers等。可以将它当成空余的空间来使用。

3. 不会降低效率。

   这是很明显的，因为caller唯一需要多做的事就是多分配32个字节的空间。这不过是将`sub esp N`的N变成了N + 32 * num_func罢了。而且一般也能提高效率，因为它保证了参数的连续、减少了callee的工作。

4. 你不知道实际传了几个参数。

   既然是由caller来分配这个空间，为什么不根据传入的参数的个数来调整分配的shadow space的空间量呢?

因为callee不知道传入的参数个数。特别是unprototyped函数中，传入的参数可多**可少**，callee不知道哪些寄存器里有需要用到的参数。如果caller分配的空间恰好是传入的参数个数，例如传入了3个参数，但callee一开始[2]以为传入了4个参数，这时候它将R9也存入栈中，这样**栈就被破坏了。**

这样callee永远无法确保shadow space是安全的，shadow space就失去了意义（然后就回到了第一条）。

References

1. Intel® 64 and IA-32 Architectures Software Developer's Manual
2. RE4B
3. win x64-calling-convention

---

1. 对于non-optimizing code这点浪费就更不算什么了。而且shadow space在未经过优化的代码中对debug是十分有利的。↩
2. 即刚进入函数进行spilling，还未进行判断和分支。↩