# lab3

PB17030889 吉志远

## 1.编译

```
lcc -o obj.asm lab2.c
```

obj.asm

```
.Orig x3000
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
HALT

GLOBAL_DATA_POINTER .FILL GLOBAL_DATA_START
GLOBAL_MAIN_POINTER .FILL main
;;;;;;;;;;;;;;;;;;;;;;;;;;;;func;;;;;;;;;;;;;;;;;;;;;;;;;;;;
lc3_func
ADD R6, R6, #-2
STR R7, R6, #0
ADD R6, R6, #-1
STR R5, R6, #0
ADD R5, R6, #-1

ADD R6, R6, #-3
ADD R0, R4, #7
LDR R0, R0, #0
jsrr R0
LDR R7, R6, #0
ADD R6, R6, #1
ADD R3, R4, #7
ldr R3, R3, #0
ADD R6, R6, #-1
STR R0, R6, #0
ADD R6, R6, #-1
STR R3, R6, #0
NOT R3, R3
ADD R3, R3, #1
```

```
ADD R0, R7, R3
LDR R3, R6, #0
ADD R6, R6, #1
ADD R7, R0, #0
LDR R0, R6, #0
ADD R6, R6, #1
ldr R3, R5, #5
add R7, R7, R3
ldr R3, R5, #6
add R7, R7, R3
ldr R3, R5, #7
add R7, R7, R3
ldr R3, R5, #8
add R7, R7, R3
ldr R3, R5, #9
add R7, R7, R3
ldr R3, R5, #10
add R7, R7, R3
str R7, R5, #0
ldr R7, R5, #4
ADD R3, R4, #6
ldr R3, R3, #0
NOT R7, R7
ADD R7, R7, #1
ADD R7, R7, R3
BRn L7
ADD R7, R4, #1
LDR R7, R7, #0
jmp R7
L7
ldr R7, R5, #10
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #9
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #8
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #7
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #6
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #5
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #4
ADD R3, R4, #6
ldr R3, R3, #0
ADD R6, R6, #-1
STR R0, R6, #0
```

```
ADD R6, R6, #-1
STR R3, R6, #0
NOT R3, R3
ADD R3, R3, #1
ADD R0, R7, R3
LDR R3, R6, #0
ADD R6, R6, #1
ADD R7, R0, #0
LDR R0, R6, #0
ADD R6, R6, #1
ADD R6, R6, #-1
STR R7, R6, #0
ADD R0, R4, #0
LDR R0, R0, #0
jsrr R0
LDR R7, R6, #0
ADD R6, R6, #1
str R7, R5, #-1
ldr R7, R5, #10
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #9
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #8
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #7
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #6
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #5
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #4
ADD R3, R4, #5
ldr R3, R3, #0
ADD R6, R6, #-1
STR R0, R6, #0
ADD R6, R6, #-1
STR R3, R6, #0
NOT R3, R3
ADD R3, R3, #1
ADD R0, R7, R3
LDR R3, R6, #0
ADD R6, R6, #1
ADD R7, R0, #0
LDR R0, R6, #0
ADD R6, R6, #1
ADD R6, R6, #-1
STR R7, R6, #0
```

```
ADD R0, R4, #0
LDR R0, R0, #0
jsrr R0
LDR R7, R6, #0
ADD R6, R6, #1
str R7, R5, #-2
ldr R7, R5, #-1
ldr R3, R5, #-2
add R7, R7, R3
ldr R3, R5, #0
add R7, R7, R3
ADD R3, R4, #6
ldr R3, R3, #0
ADD R6, R6, #-1
STR R0, R6, #0
ADD R6, R6, #-1
STR R3, R6, #0
NOT R3, R3
ADD R3, R3, #1
ADD R0, R7, R3
LDR R3, R6, #0
ADD R6, R6, #1
ADD R7, R0, #0
LDR R0, R6, #0
ADD R6, R6, #1
ADD R0, R4, #2
LDR R0, R0, #0
JMP R0
lc3_L3_test
ldr R7, R5, #0
lc3_L1_test
STR R7, R5, #3
ADD R6, R5, #1
LDR R5, R6, #0
ADD R6, R6, #1
LDR R7, R6, #0
ADD R6, R6, #1
RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;main;;;;;;;;;;;;;;;;;;;;;;;;;;
main
ADD R6, R6, #-2
STR R7, R6, #0
ADD R6, R6, #-1
STR R5, R6, #0
ADD R5, R6, #-1

ADD R6, R6, #-1
ADD R0, R4, #7
LDR R0, R0, #0
jsrr R0
LDR R7, R6, #0
ADD R6, R6, #1
```

```
ADD R3, R4, #7
ldr R3, R3, #0
ADD R6, R6, #-1
STR R0, R6, #0
ADD R6, R6, #-1
STR R3, R6, #0
NOT R3, R3
ADD R3, R3, #1
ADD R0, R7, R3
LDR R3, R6, #0
ADD R6, R6, #1
ADD R7, R0, #0
LDR R0, R6, #0
ADD R6, R6, #1
str R7, R5, #0
ADD R7, R4, #4
ldr R7, R7, #0
ADD R6, R6, #-1
STR R7, R6, #0
ADD R6, R6, #-1
STR R7, R6, #0
ADD R6, R6, #-1
STR R7, R6, #0
ADD R6, R6, #-1
STR R7, R6, #0
ADD R6, R6, #-1
STR R7, R6, #0
ADD R6, R6, #-1
STR R7, R6, #0
ldr R7, R5, #0
ADD R6, R6, #-1
STR R7, R6, #0
ADD R0, R4, #0
LDR R0, R0, #0
jsrr R0
LDR R7, R6, #0
ADD R6, R6, #1
lc3_L8_test
STR R7, R5, #3
ADD R6, R5, #1
LDR R5, R6, #0
ADD R6, R6, #1
LDR R7, R6, #0
ADD R6, R6, #1
RET


GLOBAL_DATA_START
func .FILL lc3_func
L3_test .FILL lc3_L3_test
L1_test .FILL lc3_L1_test
L8_test .FILL lc3_L8_test
L9_test .FILL #0
```

```
L6_test .FILL #2
L5_test .FILL #1
L2_test .FILL #48
.END
```

## 与自己写的lab2的主要区别

1. function calling
    1. 汇编出的代码先把R6，R5放到7FFF处，然后从高地址往低地址方向增长，~~（难道R5就是传说中的帧指针？？）~~比较模拟x86机器的内存实际状况，而我用的是 “STACK .BLKW #100 ”的方法来申请100个空间，然后从栈开始向高地址方向增长，显然汇编出来的代码不容易发生栈溢出的问题，更安全
    2. 汇编器使用jsrr R0的方式实现函数调用而我是JSR func的方法，使用寄存器显然能有更大的全球寻址范围
    3. func被用几个L段来实现，在gcc产生的汇编代码中看到过类似的方法，感觉是更模块化一点，同时在最后用L1_test .FILL lc3_L1_test 来实现寻址，很神奇
2. prolog and epilog

    汇编得到的代码分段和跳转较多，在不影响执行效果下顺序有较大变化（事实上有点看不明

    查维基百科，得prolog通常要做的操作

    > 将当前基指针推送到堆栈，以便以后可以恢复。 将栈指针（指向保存的基的指针）的值分配给基指针，以便在旧堆栈帧的顶部创建新的堆栈帧。 通过减少或增加其值来进一步移动堆栈指针，具体取决于堆栈是向下还是向上增长。 在x86上，堆栈指针被减少，以便为函数的局部变量腾出空间。

    也就是说大致形式为：

    ```
    ;intel as style:
    push ebp
    mov ebp,esp
    sub esp,N   ;此处N就是分给local var的空间了吧
    ```

    epilog

    > 将栈指针拖放到当前基指针，从而释放在本地变量的prolog中保留的空间。 将基指针弹出堆栈，因此在prolog之前将其恢复为其值。 返回到调用函数，方法是将前一帧的程序计数器从堆栈中弹出并跳转到它。 给定的结语将颠倒上述任何一个序言（完整的序言或使用录入的序言）的影响。 在某些调用约定下，被调用者有责任清除堆栈中的参数，因此结尾还可以包括向下或向上移动堆栈指针的步骤。

    至于我的代码，就是用了两个栈，一个存放t，一个存放n，在Fibonachi递归方法的基础上修改得来..

    prolog主要完成的就是把n，t，R7入栈

    epilog是把n，t，R7恢复到caller的状态

3. stacks

    汇编出来的代码用栈帧的方法保存局部变量

    我的是用好多个栈，一个存R7，一个存t，一个存n

4. pros and cons

    我的代码不容易维护，而且其他人如果想要使用我的func()是极其不友好的，不仅会污染很多空间，而且还有栈溢出等潜在问题，而汇编出来的代码使用的是7FFF处的地址，方便管理，调用和恢复比较清晰

感觉认识到了汇编和计算机底层的魅力之处...

## question&solve

x86_64机是怎么实现多参数的调用的（如C语言中的printf函数）

1. 多参数的栈细节

google，以printf为例，可以找到如“printf是从右到左的压栈顺序来实现传入参数"，关于这个参数压栈的顺序，应该是function calling决定的

查MSDN文档，得到“x86平台下常用的有三种调用约定 cdecl、stdcall、fastcall“

cdecl规定了参数是从右到左入栈，然后由caller清理栈区

以如下c代码为例：

```
int TestCdecl(int a, int b, int c, int d, int e)
{
    return a + b + c + d + e;
}
int main(){
    return  TestCdecl(1,2,3,4,5);
}
```

> gcc -m32 testcdecl.c -o test #编译为32位代码
>
> gdb test

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000513 <+0>: push   %ebp
   0x00000514 <+1>: mov    %esp,%ebp
   0x00000516 <+3>: call   0x534 <__x86.get_pc_thunk.ax>
   0x0000051b <+8>: add    $0x1ac1,%eax
   0x00000520 <+13>:    push   $0x5
   0x00000522 <+15>:    push   $0x4
   0x00000524 <+17>:    push   $0x3
   0x00000526 <+19>:    push   $0x2
   0x00000528 <+21>:    push   $0x1
   0x0000052a <+23>:    call   0x4ed <TestCdecl>
   0x0000052f <+28>:    add    $0x14,%esp
   0x00000532 <+31>:    leave
   0x00000533 <+32>:    ret
End of assembler dump.
```

可以看到push顺序确实是5,4,3,2,1，然后通过call 0x4ed完成testcdel的调用之后

查intel x64汇编指令可以知道这个esp是”栈指针“，再加上内存里栈是高地址往低地址方向增长，增加栈指针也就相当于回退，这里可以看出是由caller完成了对子进程栈空间的清理

而查x64下，可以知道x64只有一种cdecl的约定，类似于x86下的fastcall约定，所谓fastcall，就是尽量多通过寄存器来实现传参，这个最大数量在x86是2，在x64是4个（大概

所以知道了压栈的顺序和栈增长方向，对于

```
void test(int a,int b,int c,int d,int e,int f,int g,int h)
{
    printf("%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n",&a,*(&a+1),*(&a+2),*(&a+3),*(&a+4));
}
```

就可以实现顺序打印a，b，c，d，e，f，g，h的值

2. 不定数量大小参数的实现

参考C标准库对于printf的实现用了以下这些宏

1. va_list（大概是char*
2. va_start （第一个参数地址
3. ap（可变参数列表地址
4. va_end

```
#include<stdio.h>
#include<stdarg.h>
int getSum(int num,...)//可变参数列表
{
    va_list ap;//定义参数列表变量
    int sum = 0;
    int loop = 0;
    va_start(ap,num);

    for(;loop < num ; loop++)
        sum += va_arg(ap,int);

    va_end(ap);
    return sum;
}
int main(int argc,char *argv[])
{
    int sum = 0;
    sum = getSum(5,1,2,3,4,5);
    printf("%d\n",sum);
    return 0;
}
```

总结：变长参数的第一个必须是确定类型，然后编译器通过压栈顺序和地址偏移来确定之后的参数