Lab2实验报告

181240035
刘春旭

> 先看例子：
>
> ```
> int inc(int x){
>   asm ("incl %[t];" //内联汇编一条incl指令，[t]指定一个名为t的操作数，能够被分配到任意寄存器
>         : [t] "+r"(x) //“+r”表示能够同时作为输入和输出，
>       );
>   return x;
> }
> ```

1. 实现函数 `int64_t asm_add(int64_t a, int64_t b)`

这个函数可以返回有符号64位整数`a+b`的数值，使用`leaq`可以一步得到我们需要的结果；但是一定要抢下这样的思想：asm对c编译器相当于一个黑箱子，你要向它说明输入是什么，输出是什么；

于是经过几次尝试，我们得到了如下的代码：

```
int64_t asm_add(int64_t a, int64_t b){
  int64_t result = 0;
    asm volatile(
    "leaq (%[a], %[b]), %[t];"
    : [t] "=r"(result) //output
    : [a] "r"(a), [b] "r"(b) //input
  );
  return result;
}
```

2. 实现函数`int asm_popcnt(uint64_t n)`

这个函数是用来返回某个数字二进制表达中1的个数的，讲义上面给出了最朴素的表达：

```
int asm_popcnt(uint64_t x) {
  int s = 0;
  for (int i = 0; i < 64; i++) {
    if ((x >> i) & 1) s++;
  }
  return s;
}
```

但是事实上可以使用更精简的方式来写出，算是一种比较基础的bit hack：

```
int asm_popcnt(uint64_t x) {
  unsigned int c =0 ;
    for (c =0; x; ++c)
        x &= (x -1) ; // 清除最低位的1
    return c ;
}
```

事实上反汇编出来的东西也比较少，非常美观；最终写出的汇编代码就两行，非常令人神清气爽：

```
int asm_popcnt(uint64_t n) {
  unsigned int c = 0;
  for (c = 0; x; c++)
  {
    asm volatile(
      "\t leaq -1(%[x]), %%rdx;"
      "\t andq %%rdx, %[x];"
      : [x]"+r"(x)
      :"[x]"(x)
      :"rdx"
    );
  }
  return c;
}
```

3. 实现函数`void *asm_memcpy(void *dest, const void *src, size_t n)`

这个函数其实有点小tricky，但是灵魂汇编码其实也就两行：

```
"movb (%%rsi), %%al;" //rsi存的是src地址
"movb %%al, (%%rdi);" //rdi存的是dest地址
```

写出的代码略；

4. `int asm_setjmp(asm_jmp_buf env)`和`void asm_longjmp(asm_jmp_buf env, int val)`

写出这两个函数的关键其实是看懂`env`是什么东西：

首先讲义上的这个例子很好的解释了这两个函数的功能，保存到博客上以免自己忘了：

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf env;

int f(int n) {
  if (n >= 8) longjmp(env, n); // 某个条件达成时，恢复快照
  printf("Call f(%d)\n", n);
  f(n + 1);
}

int main() {
  int r = setjmp(env); // 快照
  if (r == 0) {
    f(1);
  } else { // longjmp goes here
    printf("Recursion reaches %d\n", r);
  }
}
```

好的，所以经过一番查询，其实就是把所有寄存器的状态保存到env参数指的一个区域；类似于“上下文”只不过没有“下文”；

```
> man setjmp
 The setjmp() function saves various information about  the  calling
environment  (typically,  the  stack pointer, the instruction pointer,
possibly the values of other registers and the signal mask) in the buffer env
for later use by longjmp().  In this case, setjmp() returns 0.

The longjmp() function uses the information saved in env to transfer control
back  to  the  point  where setjmp() was called and to restore ("rewind") the
stack to its state at the time of the setjmp() call. In addition, and
depending on the implementation (see NOTES), the values of some other
registers and the process signal mask may be restored to their state at the
time of the setjmp() call.
```

但是其实这个manual也没说清楚，感觉帮到自己最多的是ics2019里面的`amd64.S`；里面更详细的描述了这两个函数：（后面放到博客上当备忘录，所以不要说我水报告字数orz）

```
#ifdef __ISA_AM_NATIVE__
#
```

```
# our buffer looks like:
#  rbx,rbp,r12,r13,r14,r15,rsp,rip
.globl  setjmp
setjmp:
  movq  %rbx,  0(%rdi)
  movq  %rbp,  8(%rdi)
  movq  %r12, 16(%rdi)
  movq  %r13, 24(%rdi)
  movq  %r14, 32(%rdi)
  movq  %r15, 40(%rdi)
  leaq  8(%rsp), %rax
  movq  %rax, 48(%rdi)
  movq  (%rsp), %rax
  movq  %rax, 56(%rdi)
  xorq  %rax, %rax
  ret

#
# our buffer looks like:
#  rbx,rbp,r12,r13,r14,r15,rsp,rip
#
# _longjmp is called with two parameters:  jmp_buf*,int
# jmp_buf* is at %rdi, int is at %rsi

.globl  longjmp
longjmp:
  movq   %rsi, %rax        /* Return value */

  movq     8 (%rdi), %rbp

  movq    48 (%rdi), %rsp
  pushq   56 (%rdi)
  movq     0 (%rdi), %rbx
  movq    16 (%rdi), %r12
  movq    24 (%rdi), %r13
  movq    32 (%rdi), %r14
  movq    40 (%rdi), %r15

  testq  %rax, %rax
  jne  bye
  incq  %rax    # rax hold 0 if we are here
bye:
  ret
#endif
```

理解后，仿照这个写出自己的代码就OK啦！