CSAPP 第三次家庭作业

——1170300825 李大鑫

3.59

问题分析

假设用来相乘的两个数分别是 X 和 Y，用 Xh,Xl 来表示 X 的高 8 Byte 和低 8 Byte，同样用 Yh 和 Yl 来表示 Y 的高位和低位。

首先对应参数，按照参数的对应顺序，%rdx 存放 dest 地址，%rsi 存放 Xl，%rdx 存放 Yl。

其中首先需要明确的指令含义是:

1）cqto ：符号拓展，将%rax 的值进行符号拓展到%rdx 中，这时候两个 64 位寄存器共同构成 128 位的大数%rdx:%rax

2）mul 和 imul：mul 将操作数视为无符号数，imul 将操作数视为有符号数。其中 mulq 和 imulq 都支持两个 64 位的数进行相乘。

需要明确的以下多精度运算的必要性，整个大数存放在%rdx:%rax 中，代表的是这个数为%rdx*2^64+%rax，而不是将%rdx 和%rax 的二进制串相连表示这个 128 位数。

公式推导(用 W 表示 2^64):

P = Ph*2^64+Pl

= X*Y = (Xh*W+Xl)*(Yh*W+Yl)

=(Xh*Yh*W*W)+(Xh*Yl*W+Xl*Yh*W) +(Xl+Yl)

=(Xh*Yl*W+Xl*Yh*W) +(Xl+Yl) //(Xh*Yh*W*W)结果超出 128 位，在 128 位之内的部分始终为 0

代码解释

Store_prod:

Movq %rdx,%rax # %rax = Yl

Cqto # 符号拓展，将%rax 的值进行符号拓展到%rdx 中，这时候两个 64 位寄存器共同构成 128 位的大数%rdx:%rax，此时%rdx 存放着 Yh

Movq %rsi,%rcx # %rcx = Xl

Sarq $63,%rcx #将%rcx 进行算数右移 63 位，此时%rcx 存放的相当于是 Xl 的符号，其值要么是-1，要么是 0。

Imulq %rax,%rcx #%rcx = Yl * Xh，这里 Xh 代表 X 的符号高 8 位

Imulq %rsi,%rdx　#%rdx = Xl * Yh，这里 Yh 代表 Y 的符号高 8 位

Addq %rdx,%rcx　#%rcx = %rcx+%rdx=Yl*Xh + Xl *Yh,

Mulq %rsi　#无符号计算 Xl*Yl,并将 Xl*Yl 的 128 位结果的高位放在%rdx,低位放在%rax,这里%rdx = Zh, %rax = Zl.

Addq %rcx,%rdx　# %rdx = Yl*Xh+Xl*Yh+Zh（高 64 位加上 Xl*Yl 的进位）

Movq %rax,(%rdi) #将 rax 的值存放到 dest 指向地址的低 8 位

Movq %rdx,8(%rdi) #将 rdx 的值存放到 dest 指向地址的高 8 位。

Ret

3.61

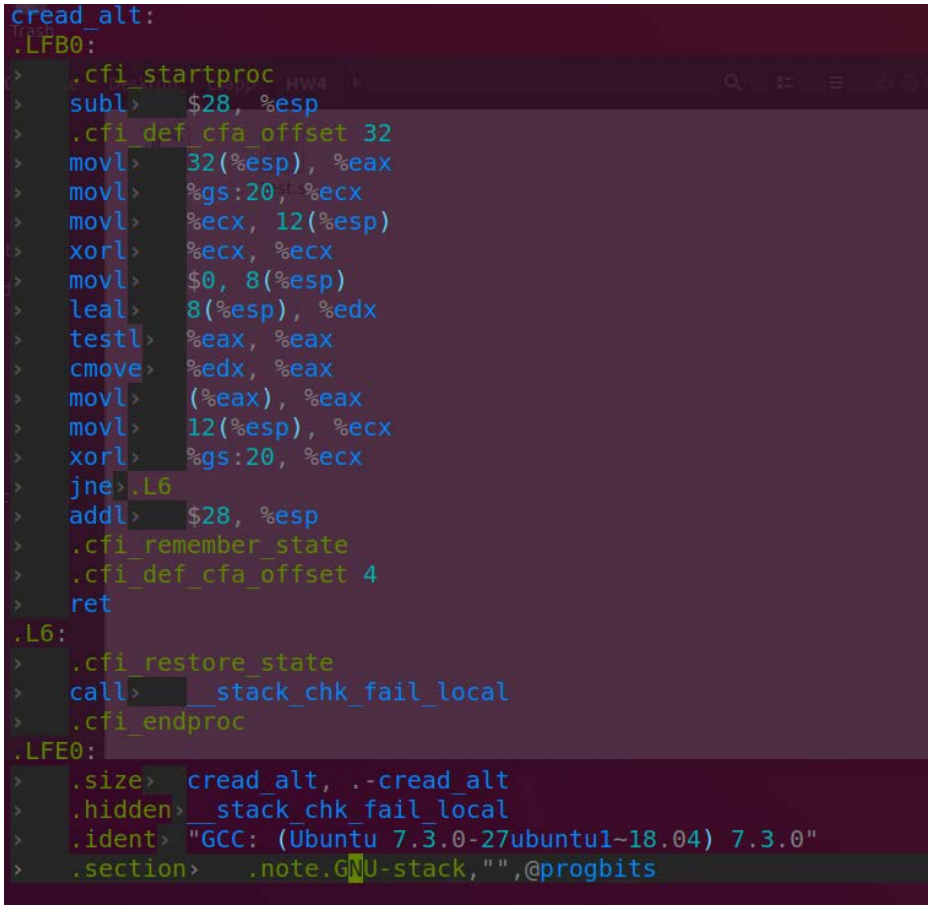
问题分析

本题要求 cread_alt 能够使用 cmov 语句进行条件传送而不是跳转语句进行赋值，首先会发现原来的代码即使使用-O1 优化依然是利用跳转语句进行赋值，这是因为如果想要使用 cmov 语句*xp 和 0 两个数必须是提前求出来的，而这里如果要先求出*p 的话因为 p 是一个指针它有可能指向 null，所以 gcc 编译的时候不会使用 cmov 形式，因此这里需要三元运算符中所有的数字都是已知的，因为 xp 指针是已知的（一个地址），因此采用以下求出指针的方法而不是直接求值的方法就可以避免这个问题。

答案

```
1 long cread_alt(long *xp)
2 {
3   long t=0;
4   long *p = xp ? xp : &t;
5   return *p;
6 }
```

Gcc 使用-O1 优化之后生成的汇编代码为：



3.63

问题分析

根据汇编程序逆向得出 c 语言逻辑。Switch 语句使用跳转指令实现，需要注意 break 对应 retq 的情况，判断好 switch 的跳出。

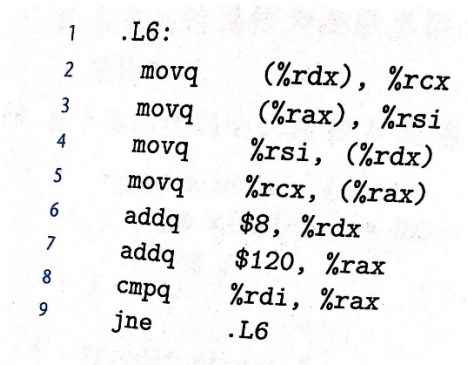
答案

```
long switch_prob(long x,long n) {
    long result = x;
    switch(n) {
        /*Code filled in here*/
        case 60:
        case 62:
            result = 8*x;
            break;
        case 63:
```

```
            result = x>>3;
            break;
        case 64:
            result = (x<<4)-x ;
            x = result;
        case 65:
            x*=x;
        default:
            result = x+0x4b;
    }
}
```

3.65

问题分析



```
1   .L6:
2     movq    (%rdx), %rcx
3     movq    (%rax), %rsi
4     movq    %rsi, (%rdx)
5     movq    %rcx, (%rax)
6     addq    $8, %rdx
7     addq    $120, %rax
8     cmpq    %rdi, %rax
9     jne     .L6
```

2-5 行目的交换(%rdx)和(%rax)

结合数组在内存中的存放方式以及 rax 和 rdx 的增加值，可以判断出%rdx 保存着指向 Aij 的指针，%rax 保存着指向 Aji 的指针。
一行有 120bit，即 120/8=15 个 int

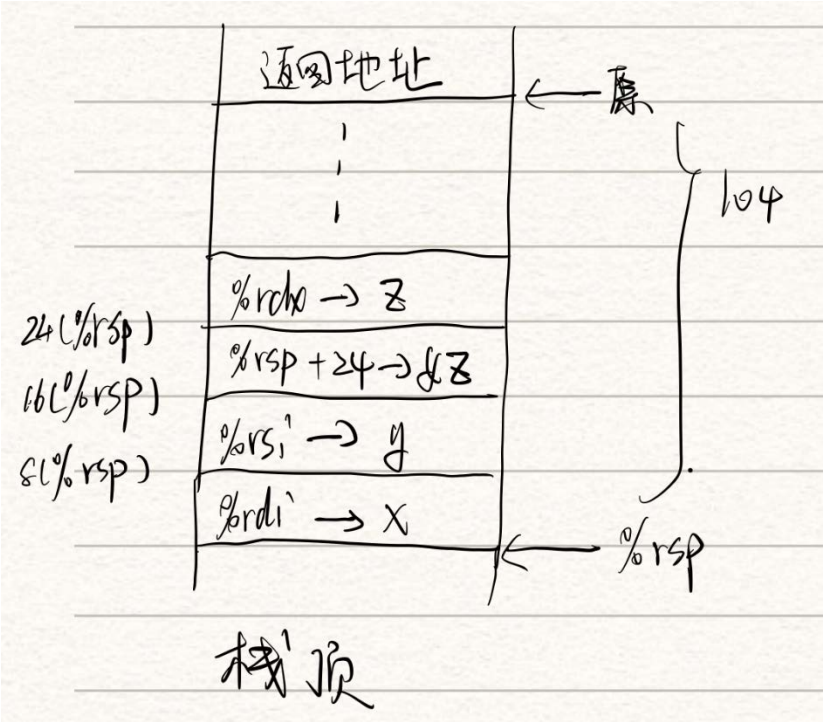
答案

A：%rdx 保存着指向 Aij 的指针
B：%rax 保存着指向 Aji 的指针
C：15

3.67

答案

A：在调用 process 之前存储在栈上的值



B：%rdi，传入的%rdi=64(%rsp)，代表的是 strB r 在 eval 函数中存储的栈地址。之后在 process 中的赋值操作就是用%rdi 定位地址的。

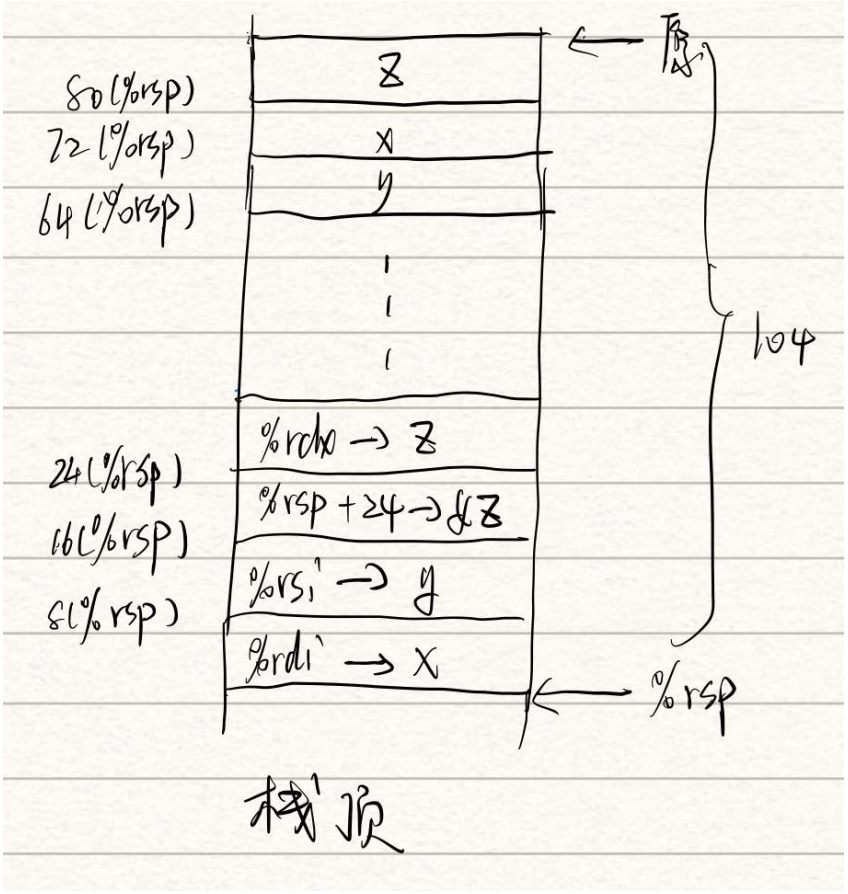
C：通过%rsp 在栈上进行寻址取值，有以下对应：（这里的 rsp 是调用 eval 之前的 rsp）

s.p	**16(%rsp）**
s.a[0]	8(%rsp)
s.a[1]	(%rsp)

D：直接传入结果结构的地址，通过使用 64(%rsp)、72(%rsp)、80(%rsp)来进行 mov 赋值，其中 64(%rsp)就指向了 r 的位置。

r.q	**80(%rsp）**
r.u[0]	72(%rsp)
r.u[1]	64(%rsp)

E：完成 eval 的栈帧图



F:
作为参数和作为函数结果返回的结构都是通过传递参数来进行传递，然后对应的赋值直接在栈上通过 mov 直接赋值完成。

3.69

问题分析：

```
.Test
    mov 0x120(%rsi),%ecx          # 取值b_struct.last，这里可以推出a[CNT]共占0x120-0x8=280bit
    add (%rsi),%ecx               # (%rsi)取值b_struct.first这一步进行两者之间求和
    lea (%rdi,%rdi,4),%rax
    lea (%rsi,%rax,8),%rax        # %rax=40*i + bp，这里可以推出一个a_struct共占40bit
    mov 0x8(%rax),%rdx            # 取值ap->idx，
    movslq %ecx,%rcx              #拓展为64位数字，a_struct.x[idx]中存储的是8Byte的Long类型。
    mov %rcx,0x10(%rax,%rdx,8)    #%rcx=n，所以0x10(%rax,%rdx,8)对应ap->x[ap->idx]，地址为8*%rdx+%rax+16。说明x数组为8Byte的Long数组，得出x有
```

(40-8）/8=4个。
    retq

答案

A:
CNT=7

B:
```
typedef struct {
    long idx;
    long x[4];
} a_struct;
```

3.71

问题分析

可行性：因为 gets 是系统给分配空间的，所以我们无法决定分配空间的大小，因此无法避免缓冲区溢出，而 fgets 是可以定制读入长度的，因此我们可以利用这一特性写出解决缓冲区溢出问题的读入方法。

fgets api 如下：

```
char *fgets(char *str, int n, FILE *stream)
```

答案

代码:
```
void good_echo()
 {
    const int bufsz = 0x8;          //一次读取 缓冲区的长度
    char str[bufsz];
    int i;
    while(fgets(str, bufsz, stdin)!=NULL)
    {
       for(i=0;str[i];i++) {           //str[i]如果为/0则为false结束
          putchar(str[i]);          //打印
       if(str[i]=='\n')          //如果读到换行符 则直接退出
           break;                //因为如果只有下面一个判断则长度为8的倍数的
```

```
时候不能停止
        }
         if(i<bufsz-1) break;
    }
    return;
}
```

运行结果:

```
fasdfjldsjafkldajjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjfadsfjkasjdflkjasdlkfjsldjflksjdaflkjadlksfjklsdjfiwefjs
dlkjfsdkjfd;askjfkajsdfkjsadkfjksad;jfksdajfkjsdfk;jasdkfjasdkfja;sdfjakldjfaksdjfkajsdfk
fasdfjldsjafkldajjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjfadsfjkasjdflkjasdlkfjsldjflksjdaflkjadlksfjklsdjfiwefjs
dlkjfsdkjfd;askjfkajsdfkjsadkfjksad;jfksdajfkjsdfk;jasdkfjasdkfja;sdfjakldjfaksdjfkajsdfk

--------------------------------
Process exited after 3.542 seconds with return value 0
请按任意键继续. . .
```

3.73

问题分析

函数 find_range 如下:

```
typedef enum {NEG, ZERO, POS, OTHER} range_t;

range_t find_range(float x)
{
    int result;
    if (x < 0)
        result = NEG;
    else if (x == 0)
        result = ZERO;
    else if (x > 0)
        result = POS;
    else
        result = OTHER;
    return result;
}
```

a) C代码

```
  range_t find_range(float x)
  x in %xmm0
find_range:
  vxorps  %xmm1, %xmm1, %xmm1          Set %xmm1 = 0
  vucomiss        %xmm0, %xmm1         Compare 0:x
  ja      .L5                          If >, goto neg
  vucomiss        %xmm1, %xmm0         Compare x:0
  jp      .L8                          If NaN, goto posornan
  movl    $1, %eax                     result = ZERO
  je      .L3                          If =, goto done
.L8:                                 posornan:
  vucomiss        .LC0(%rip), %xmm0    Compare x:0
  setbe   %al                          Set result = NaN ? 1 : 0
  movzbl  %al, %eax                    Zero-extend
  addl    $2, %eax                     result += 2 (POS for > 0, OTHER for NaN)
  ret                                  Return
.L5:                                 neg:
  movl    $0, %eax                     result = NEG
.L3:                                 done:
  rep; ret                             Return
```

b) 产生的汇编代码

图 3-51 浮点代码中的条件分支说明

要求代码只包含一个浮点比较指令和可使用一些条件传送指令。直接利用一个比较加上条件跳转指令实现类似于无 break 的 switch 效果的功能。

答案：

```
vxorps xmm0,xmm0,xmm0
movl $3,eax
vucomiss xmm0,xmm1    #cmp x:0
 ja .L1        #大于0,注意跳转之后顺序向下执行
 je .L2        #等于0
 jb .L3        #小于0
 jp .L4
 .L3:
    subl $1,eax
 .L2:
    subl $1,eax
 .L1:
    subl $1,eax
 .L4:
    rep;ret
```

3.75

答案

A：如何向函数传递复数参数？

参数的传递是调用函数直接将值放在指定 XMM 寄存器实现的。其中对于第 i 个参数，器 imag 部分被存储在%xmm(2i-1)中，real 部分被保存在%xmm(2i-2)中，i 取 1,2,3...

B：如何从函数返回复数值？

返回的结果是被共同保存到%xmm0 和%xmm1 寄存器的，其中%xmm0 保存 read 部分，%xmm1 保存 imag 部分。