第二章　指令：计算机的语言

2.3 [5]<2.2,2.3> 对于以下 C 语句，请编写相应的 RISC-V 汇编代码。假设变量 f、g、h、i 和 j 分别分配给寄存器 x5、x6、x7、x28 和 x29。假设数组 A 和 B 的基地址分别在寄存器 x10 和 x11 中。

```
B[8] = A[i-j];
```

这里假设数组 A 和 B 中的元素都是 64 位的，即“双字”，即 8 个字节。

```
sub     x30, x28, x29   // i-j，从 A 开始的双字偏移
slli    x30, x30, 3     // (i-j)*8，从 A 开始的字节偏移
add     x30, x30, x10   // A[i-j] 的地址
ld      x31, 0(x30)     // 加载 A[i-j]
sd      x31, 64(x11)    // 放到 B[8]，8 是双字偏移，转化成字节偏移就是 64
```

验证一下

使用命令 `riscv64-unknown-linux-gnu-gcc -S main.c -o main.s` 编译，左边是 C 代码，右边是汇编代码。

```
...
long long *A, *B, i,j;
B[8] = A[i - j];
...
```

```
...
ld      a4,-24(s0)  // 加载 i
ld      a5,-32(s0)  // 加载 j
sub     a5,a4,a5
slli    a5,a5,3
ld      a4,-40(s0)  // 加载 A
add     a4,a4,a5
ld      a5,-48(s0)  // 加载 B
addi    a5,a5,64
ld      a4,0(a4)
sd      a4,0(a5)
...
```

比较后可以发现基本差不多，但是最后一步赋值时这里先用了 addi 偏移 64 位，再存储，为什么不是直接 64(a5) 一步存储到从 a5 偏移 64 个字节的位置呢？

2.4 [10]<2.2,2.3> 对于以下 RISC-V 汇编指令，相应的 C 语句是什么？假设变量 f、g、h、i 和 j 分别分配给寄存器 x5、x6、x7、x28 和 x29。假设数组 A 和 B 的基地址分别在寄存器 x10 和 x11 中。

```
slli    x30, x5, 3      // x30 = f*8
add     x30, x10, x30   // x30 = &A[f]
slli    x31, x6, 3      // x31 = g*8
add     x31, x11, x31   // x31 = &B[g]
ld      x5, 0(x30)      // f = A[f]

addi    x12, x30, 8     // x12 = (&f + 1) = (原始的 f)&A[f + 1]
ld      x30, 0(x12)     // x30 = A[f + 1]
add     x30, x30, x5    // x30 = f + A[f + 1] = (原始的 f)A[f] + A[f + 1]
sd      x30, 0(x31)     // B[g] = A[f] + A[f + 1]
```

注释已写在上方，因此此汇编相应的 C 语句如下：

```
B[g] = A[f] + A[f + 1];
```

2.10 假设寄存器 x5 和 x6 分别保存值 0x8000000000000000 和 0xD000000000000000。

2.10.1 [5]<2.4> 以下汇编代码中 x30 的值是多少?

```
add x30, x5, x6
```

由于后面全是 0，只需要考虑最高 4 位二进制。x5 为 8_{16} 即 1000_2，x6 为 D_{16} 即 1101_2，要注意最高位为 1 代表负数，由于负数已按照补码方式表示，所以可以直接相加。这里两个数都是负数，相加后为 $1\ 0101_2$，进位忽略，即 $0101_2 = 5_{16}$，此时 x30 的值为

0x5000000000000000

2.10.2 [5]<2.4> x30 中的结果是否为预期结果，或者是否溢出?

两个负数相加，结果的最高位为 0，表示正数，不是预期结果，即产生了溢出。

2.10.3 [5]<2.4> 对于上面指定的寄存器 x5 和 x6 的内容，以下汇编代码中 x30 的值是多少?

```
sub x30, x5, x6
```

还是只考虑最高 4 位二进制，最高位为 1 代表负数，直接相减不方便，可以先按照补码方式转换为相反数再相加，按照补码方式 1101_2 的相反数为 0011，因此 $1000_2 - 1101_2 = 1000_2 + 0011_2 = 1011_2$，即 B_{16}。因此 x30 的值为

0xB000000000000000

2.10.4 [5]<2.4> x30 中的结果是否为预期结果，或者是否溢出?

两个负数相减，结果仍为负数，最高位是 1，为预期结果，无溢出。

2.10.5 [5]<2.4> 对于上面指定的寄存器 x5 和 x6 的内容，以下汇编代码中 x30 的值是多少?

```
add x30, x5, x6
add x30, x30, x5
```

第一行就是 2.10.1 的指令，之后执行第二行，按照最高位来看就是 $0101_2+1000_2=1101_2$，即 D_{16}，所以 x30 的值为

$$0xD000000000000000$$

2.10.6 [5]<2.4> x30 中的结果是否为预期结果，或者是否溢出?

结果的最高位为 1，虽然仍是负数但显然不是 x5 + x6 + x5 的结果，所以不是预期结果，产生了溢出。

2.27 [5]<2.7> 将以下循环转换为 C 代码。假设 C 语言级的整数 i 保存在寄存器 x5 中, x6 保存名为 result 的 C 语言级的整数，x10 保存整数 MemArray 的基址。

```
        addi    x6, x0, 0       // result = 0
        addi    x29, x0, 100    // x29 = 100
LOOP:   ld      x7, 0(x10)      // x7 = *MemArray
        add     x5, x5, x7      // i = i + *MemArray
        addi    x10, x10, 8     // MemArray++
        addi    x6, x6, 1       // result++
        blt     x6, x29, LOOP   // (result < 100)
```

注释已写在上方。根据题意和命名来看，MemArray 应该是整数数组而不是整数吧，不然总不能 (&MemArray)++ 吧。这里假设整数都是 64 位的，所以 MemArray++ 对应的汇编代码是增加 8 个字节，即 addi x10, x10, 8 。

根据注释分析出的结果，可以得到 C 代码:

```
for (int result = 0; result < 100; result++) {
    i += *(MemArray++);
}
```

为什么这里是 result 作为循环变量而 i 作为结果啊 ??? 但按照题意分析出来就是这样。

2.31 [20]<2.8> 将函数 f 转换为 RISC-V 汇编语言。假设 g 的函数声明是 `int g(int a,int b)`。函数 f 的代码如下:

```
int f(int a, int b, int c, int d) {
    return g(g(a,b), c+d);
}
```

注意:

(1) 调用函数时参数应该是从右往左计算的，但是需要保存的局部变量是从左到右保存的;

(2) 栈指针应该是 16 字节（四字）对齐的;

(3) 返回地址应该是调用者保存的。

```
f:                          // 保存自己要用到的保存寄存器，好像没有
                            // 此时 a,b,c,d 分别保存在 x10, x11, x12, x13 中
    addw    x5, x12, x13    // 计算 c+d, 4 字节整数的要加 w
    addi    sp, sp, -16     // 移动栈指针，栈指针应该是 16 字节对齐的？
    sd      x1, 8(sp)       // 保存 x1，返回地址
    sd      x5, 0(sp)       // 保存 x5，这里 int 类型 4 个字节，由于对齐产生了空位
                            // a 和 b 在调用 g(a,b) 后不会用到，所以不用保存
                            // *** 开始计算 g(a,b)
                            // a 和 b 已经在 x10 和 x11 中所以不需要移动
    jal     x1, g           // 跳转到 g
                            // g 的返回值在 x10 中，正好是下一次调用的第一个参数
    ld      x11, 0(sp)      // 恢复 x5，直接恢复到 x11 上避免再移动
                            // x1 不需要着急恢复，马上就是下一次调用了
                            // 只恢复了 x5，栈指针要 16 字节对齐不能移动
                            // *** 开始计算 g(g(a,b), c+d)
                            // 第二个参数已从 x5 恢复
                            // 第一个参数已经在 x10 中
    jal     x1, g           // 跳转到 g
                            // g 的返回值在 x10 中，不需要移动
    ld      x1, 0(sp)       // 恢复 x1
    addi    sp, sp, 16      // 恢复栈指针
                            // 恢复保存的保存寄存器，好像没有
    jalr    x0, x1          // 返回
```

验证一下

使用命令 `riscv64-unknown-linux-gnu-gcc -S main.c -o main.s` 编译。

```
f:
.LFB0:
```

```
        .cfi_startproc
        addi    sp,sp,-32
        .cfi_def_cfa_offset 32
        sd      ra,24(sp)
        sd      s0,16(sp)
        .cfi_offset 1, -8
        .cfi_offset 8, -16
        addi    s0,sp,32
        .cfi_def_cfa 8, 0
        mv      a5,a0
        mv      a4,a3
        sw      a5,-20(s0)
        mv      a5,a1
        sw      a5,-24(s0)
        mv      a5,a2
        sw      a5,-28(s0)
        mv      a5,a4
        sw      a5,-32(s0)
        lw      a4,-24(s0)
        lw      a5,-20(s0)
        mv      a1,a4
        mv      a0,a5
        call    g
        mv      a5,a0
        mv      a3,a5
        lw      a5,-28(s0)
        mv      a4,a5
        lw      a5,-32(s0)
        addw    a5,a4,a5
        sext.w  a5,a5
        mv      a1,a5
        mv      a0,a3
        call    g
        mv      a5,a0
        mv      a0,a5
        ld      ra,24(sp)
        .cfi_restore 1
        ld      s0,16(sp)
        .cfi_restore 8
        .cfi_def_cfa 2, 32
        addi    sp,sp,32
        .cfi_def_cfa_offset 0
        jr      ra
        .cfi_endproc
```