lab 6

Purpose

本次实验的目的在于使用高级语言重写之前用机器码或是汇编语言的实验，更加深入地理解这些实验，并了解高级语言与底层语音的不同，知道高级语言中的具体功能在底层是如何实现的。

Principles

lab 1

分析：求二进制码中0的个数可以模拟短除法来设计实验。 n为奇数时求原码中0的个数，但是由于原码长度为16位，而短除法求出的位数未知，所以如果直接求0的个数还需要一个量来表示短除法求得的位数长度。 n为偶数时
求-n的补码的0的个数，如果先求原码再转补码就会很麻烦。而-n的补码的数值位与n-1的反码的数值位相同，
其符号位又一定是1，这样-n的补码的0的个数就等于n-1的原码的1的个数。 于是可以求n或n-1的二进制码的1
的位数count，对于奇数，16-count即为二进制码中0的个数；而对于偶数count即为所求。最后再加上学号的最
后一位即可。

由于实验中不允许直接使用`/`和`%`这两种算符，所以这里需要先实现两个函数用于实现相应的功能。其逻辑也较为简单：

- 除法：设所求为`n/x`，定义一个变量`res`，初始化为0。使用一个`while`循环，当n大于等于0时，则将n减去x，res加上1。最后返回`res-1`。

  为什么不将循环条件设为`n >= x`而是设为`n >= 0`？

  因为在LC-3中要判断n与x的大小关系也需要将n先减去x，再根据结果判断，如果结果为负还需要恢复n的值，或者是使用另一个寄存器来保存结果，如果结果为非负则将结果赋给存储n的寄存器。虽然在C语言中不需要考虑这些问题，但为了保证与之前写的算法一致，这里我还是选择了以`n >= 0`为循环条件。

- 取余：设所求为`n%x`，使用一个`while`循环，只要`n >= 0`，就将n减去x，最后返回`n + x`。和上面除法的实现一样，这里我的循环条件没有设为`n >= x`，而是设为了`n >= 0`，最后返回`n + x`。

在实现了以上两个函数后，就可以根据上面的分析写出代码：

```
// 实现求余数
int Mod(int n,int x)
{
    while(n >= 0)
        n = n - x;
    return (n + x);
}
```

```
// 实现除法
int Div(int n,int x)
{
    int res = 0;
    while(n >= 0)
    {
        n = n - x;
        res++;
    }
    return (res - 1);
}

int16_t lab1(int16_t n) {
    int16_t count = 0,isEven = 0;
    if(Mod(n,2) == 0) {
        n--;//如果n为偶数，计算n-1的原码的1的数目，即为n的补码的0的树目
        isEven=1;
    }
    while(n){
        if(Mod(n,2)) count++;//也可写作count+=n%2;
        n = Div(n,2);
    }
    if(!isEven){
        count=16 - count;
    }
    //count = count + STUDENT_ID_LAST_DIGIT;
    return (count + STUDENT_ID_LAST_DIGIT);
}
```

以上的步骤为：首先判断n是否为偶数，若是偶数则将n减去1，并将标记n是否为偶数的变量 `isEven` 赋值为1。然后求n的二进制源码中1的位数（用 `count` 表示）：使用一个 `while` 循环，只要n不为0，则首先判断n是否为奇数即 `Mod(n,2)` 不为0，若为奇数则将 `count` 加1。然后将n赋值为 `Div(n,2)`。最后根据 `isEven` 的值改变 `count`：若 `isEven = 0`，则说明n是偶数，需要将16减去 `count` 的值赋给 `count`。最后返回 `count + STUDENT_ID_LAST_DIGIT`，即 `count` 加上学号的最后一位。

lab 2

分析：要计算f(N)，共需要循环n-1次，并且初始化结果变量f = 3。在循环体中需要做以下处理：

- 首先需要根据 `direction` 来更新 `f` 的值。这里用0表示 `direction` 是向上的，1表示向下的。若 `direction = 0`，则 `f = f + f + 1`(因为不允许使用 `*`，所以这里用f + f而不是f*2)。
- 然后判断 `f` 是否为8的倍数，这里调用 `lab1` 中的 `Mod` 函数即可，若 `Mod(f,8) == 0`，则将 `direction` 取反，然后 `break`，不再进行循环体中剩下的操作。
- 若 `Mod(f,8) != 0`，再判断 `f` 的十进制表示中是否含有8。只需要 `f` 对10取余数即可（调用 `lab1` 中的 `Mod` 函数即可），若余数为8，则将 `direction` 取反并 `break`；否则将 `f` 赋值为 `Div(f,10)`，这样一直判断至 `f == 0` 结束。

最后得到的 `f` 即为所求，以下为代码实现：

```
int16_t lab2(int16_t n) {
    int16_t f = 3, direction = 0,temp,temp_f;
```

```
    for(n--;n > 0;n--)
    {
        f = direction ? (f + f - 2) : (f + f + 2);
        if(f > 4096)
            f = f - 4096;
        if(Mod(f,8) == 0)
        {
            direction = !direction;
        }
        else
        {
            temp_f = f;
            do
            {
                temp = Div(temp_f,10);
                temp_f = Mod(temp_f,10);
                if(temp_f == 8)
                {
                    direction = !direction;
                    break;
                }
                else
                {
                    temp_f = temp;
                }
            } while(temp);
        }
    }
    return f;
}
```

lab 3

在用汇编语言来完成这个实验时，关键在于子程序的使用，通过调用子程序的方式将一个复杂的问题简单化，拆分成几个更为容易的小实验，而现在使用高级语言就十分简单，只需要添加函数即可。在汇编语言实现中我也为读取输入的字符这一功能写了一个子程序，但是在`C++`中只需要使用`>>`来读取即可，所以这个实验只需要写个用于比较两个字符串是否相等的函数`compare`即可。其逻辑也较为简单：

设输入的字符串为`s2`，要与已知的字符串`s1`比较是否相等。只需要用一个while循环，依次比较`s1`和`s2`的每一位，这里需要注意的是`s1`的串尾标识`'\0'`也需要与`s2`进行比较，以防`s1`，`s2`的长度不一致，所以循环条件可以设为常量`1`，当`*s1 == '\0'`时，若`*s2 == 0`，则返回1，否则返回0；而如果`*s1 != '\0'`并且`*s1 != *s2`则返回0。代码实现如下：

```
// 比较两个字符串是否相同
int compare(char *s1,char *s2)
{
    while(1)
    {
        if(*s1 == '\0') {
            return (*s2 == '\0') ? 1 : 0;
        }
        if(*(s1++) != *(s2++))
            return 0;
    }
}
```

在汇编语言中如果想要实现返回0或1的功能，可以用一个寄存器初始化为0，需要返回1时将这个寄存器加上1即可。

实现`compare`函数后，就可以按部就班的一步步比较了：

- 首先判断输入的`input_cnt`是否合法，这里最少需要输入一个字符串，最多输入三个字符串，若`input_cnt < 1`或是`input_cnt > 3`，则打印错误信息，返回0。
- 然后先比较输入的第一个字符串是否与密码相同，若相同，则打印`righ\n`，返回1，否则打印`wron\n`，然后打印验证码，执行下一步。
- 首先判断`input_cnt`是否为1，若为1则需要打印提示信息提醒用户输入验证码，若不为1则执行下一步。
- 比较输入的第二个字符串是否与验证码相同，若相同，则执行下一步，否则打印`wron`，返回0。
- 首先判断`input_cnt`是否为2，若为2则需要打印提示信息提醒用户再次输入密码，若不为1则执行下一步。
- 比较输入的第三个字符串是否与密码相同，若相同，则打印`righ\n`，返回1，否则打印`wron\n`，返回0。

以下为代码实现：

```
int16_t lab3(char s1[], char s2[], int input_cnt, char my_input[10][MAXLEN]) {
    if(input_cnt == 0) {
        printf("no input!");
        return 0;
    }
    else if(input_cnt > 3) {
        printf("wrong input number!\n");
        return 0;
    }
    if(compare(s1,my_input[0])) {
        printf("righ\n");
        return 1;
    }
    else {
        printf("wron\n%s\n",s2);
        if(input_cnt == 1) {
            printf("please input verify:\n");
            return 0;
        }
        else if(compare(s2,my_input[1])) {
            if(input_cnt == 2) {
                printf("please input password again:\n");
```

```
            }
            else if(compare(s1,my_input[2])) {
                printf("righ\n");
            }
            else {
                printf("wron\n");
            }
        }
        else {
            printf("worn\n");
        }
    }
    return 1;
}
```

lab 4

分析：本次实验的关键在于实现递归函数`REMOVE`和`PUT`。其中`REMOVE(i)`用于将前i个环从板上移除，即将state的前n位由1变为0；而`PUT(i)`用于将前i个环放到板上，即将state的前n位由0变为1。二者是相反的操作。

以`REMOVE`为例，在函数中需要做如下操作：

1. 若$n=0$，则直接返回state
2. 若$n=1$, 则将state的最后一位由0变为1，直接对state加1即可，然后存储操作后的state于特定的存储空间，再返回state。
3. 若以上两种情况均不成立，则先将`state`的前n-2位全部变为1，即调用`REMOVE(n-2,state,memory)`，然后再将`state`的第n位由0变为1，这里可以将一个二进制表示的只有第n位为1的掩码加到`state`上，要得到这样的一个掩码可以定义一个变量`mask`初始化为1，然后将使用一个循环，循环n-1次，每次都将`mask`赋值为`mask + mask`。更新完`state`后，将其存储于`memory`中，因为`memory`是一个数组，所以这里需要一个全局变量`times`来记录`state`变化的次数，也即`state`要存储的位置，每次存储完后将`times`加1即可，即`memory[++times] = state`；然后再调用`PUT(n-2,state,memory)`将state的前n-2位复原为0，最后再调用`REMOVE(n-1,state)`将state的前n-1位由0变为1。

`PUT`的操作与`REMOVE`相似，不同的是`PUT`是要将第n位由1变成0，更新`state`将其减去`mask`即可：只有第n位0，其余全为1，只要按照`REMOVE`中的方法构造出掩码再取反即可。

实现了以上两个函数之后，主函数中只需要将初始的n存入`memory[0]`并调用`REMOVE`即可。

代码如下：

```
int times;//times表示操作的次数

int REMOVE(int n,int state,int16_t *memory){
    if(n==0) return state;
    if(n==1){
        state++;//改变state
        memory[++times] = state;//存储当前操作后的state
```

```
        return state;
    }
    state = REMOVE(n-2, state,memory);
    int16_t mark = 1;
    for(int i = 0;i < n-1;i++)
        mark = mark + mark;
    state = state + mark;//将state的第n位变成1
    memory[++times] = state;
    state = PUT(n-2, state,memory);
    state = REMOVE(n-1, state,memory);
    return state;
}

int PUT(int n,int state,int16_t *memory){
    if(n == 0)  return state;
    if(n == 1)
    {
        state--;//改变state
        memory[++times] = state;//存储当前操作后的state
        return state;
    }
    state = PUT(n-2, state,memory);
    int16_t mark = 1;
    for(int i = 0;i < n-1;i++)
        mark = mark + mark;
    state = state - mark;//将state的第n位变成0
    memory[++times] = state;
    state = REMOVE(n-2,state,memory);
    state = PUT(n-1, state,memory);
    return state;
}


int16_t lab4(int16_t *memory, int16_t n) {
    //memory[0] = n;
    times = -1;
    REMOVE(n,0,memory);
    return times+1;
}
```

Procedure

1. 实验过程中我遇到的问题有lab4中我打印出的信息起初始终多了一行，后来发现是我在将`n`存在了`memory[0]`，因为lab4中要求要将n存在`x3100`中，所以我这里就将n存在了`memory`中，但是根据框架中的循环范围来看，这里并不需要存储n，所以我就将相应的代码注释掉了，并且将`times`的起始值从0改成了-1(因为我存储`state`时用的是`memory[times++] = state;`)，解决了问题。
2. 我是使用vscode来编写程序的，在运行时遇到了文件无法打开的情况，换成绝对路径后成功打开了文件。在网上查找了使用相对路径的解决方法但都没有成功，因此我运行时还是使用的是绝对路径。为了防止检查的时候出现问题，在提交的文件中，我还是使用了相对路径。

Results

以下为运行结果：

以下为提供的测试文件的运行结果：

```
===== lab1 =====
15
===== lab2 =====
786
===== lab3 =====
wron
world
wron
===== lab4 =====
0000000000000001
0000000000000101
0000000000000100
0000000000000110
0000000000000111
```

其中lab 1的结果与教程中不一致，因为我的学号最后一位是1，和提供的框架中设置的不一样。其他结果均与教程的一致。

我自己又对测试文件做了一些修改如下：

```
100
11
hello world 3 abcde world fghij
5
```

运行结果如下：

```
===== lab1 =====
5
===== lab2 =====
3150
===== lab3 =====
wron
world
wron
===== lab4 =====
0000000000000001
0000000000000101
0000000000000100
0000000000000110
0000000000000111
0000000000010111
0000000000010110
0000000000010010
0000000000010011
0000000000010001
0000000000010000
0000000000010010
0000000000010011
0000000000011011
0000000000011001
0000000000011000
0000000000011001
0000000000011101
0000000000011100
0000000000011110
0000000000011111
```

结果也都正确。