Lab6 Learn from the past

Lab1

程序设计

L程序

```
#include <iostream>
int main()
{
    short n, m, result;
    result = 0;
    std :: cin >> n >> m;
    do
    {
        m -= 1;
        result += n;
    }
    while(m != 0);
    std :: cout << result;
    return 0;
}
```

P程序

```
#include <iostream>
int main()
{
    short n, m, result;
    result = 0;
    short pt = 1;
    std::cin >> n >> m;
    for(int i = 0; i < 16; i++)
    {
        if(pt & m) result += n;
        n += n;
        pt += pt;
    }
    std::cout << result;
    return 0;
}
```

实验思考

在C++中一个 **'*'** 符号就可以完成的运算，事实上它的底层实现可能并不会非常简单。在数字电路课程上，我了解到了硬件层面实现两个4位2进制数的乘法的方法，但其实那并不非常容易。相比较加法而言，如果希望用一条指令完成乘法，那么ALU势必要花费更多时钟周期。即便是移位乘法，也至少用16次循环。

事实上，乘法运算只是一个引子，我们在高级语言中调用某些函数或者运算的时候，也要考虑到它底层实现的复杂度，这样有助于我们提升程序的运行效率。

Lab2 && Lab3

程序设计

Lab2

```
#include <iostream>
int main()
{
    int r0 = 0;
    int r1 = 1;
    int r2 = 1;
    int r3 = 0;
    int r4 = 1023;
    int num;
    std::cin >> r0;
    int r5 = r0;
    int r7 = 2;
    r0 -= 2;
    if(r0 <= 0) r7 = r5;
    else
    {
        do
        {
            r3 = r1 + r1;
            r1 = r2;
            r2 = r7;
            r7 = r7 + r3;
            r0 = r0 - 1;
        } while (r0 > 0);
    }
    r7 = r7 & r4;
    std::cout << r7 << std::endl;
    return 0;
}
```

Lab3

```
#include <iostream>
int main()
{
    int fib[150] = {
        1, 1, 2, 4, 6, 10, 18, 30, 50,
        86, 146, 246, 418, 710, 178,
        1014, 386, 742, 722, 470, 930,
        326, 242, 54, 706, 166, 274,
        662, 994, 518, 818, 758, 770,
        358, 850, 342, 34, 710, 370,
        438, 834, 550, 402, 22, 98,
        902, 946, 118, 898, 742, 978,
        726, 162, 70, 498, 822, 962,
```

```
        934, 530, 406, 226, 262, 50,
        502, 2, 102, 82, 86, 290,
        454, 626, 182, 66, 294, 658,
        790, 354, 646, 178, 886, 130,
        486, 210, 470, 418, 838, 754,
        566, 194, 678, 786, 150, 482,
        6, 306, 246, 258, 870, 338,
        854, 546, 198, 882, 950, 322,
        38, 914, 534, 610, 390, 434,
        630, 386, 230, 466, 214, 674,
        582, 1010, 310, 450, 422, 18,
        918, 738, 774, 562, 1014, 514,
        614, 594, 598, 802, 966, 114,
        694, 578, 806, 146, 278, 866,
        134, 690, 374, 642, 998, 722,
        982
        };
    int r0;
    std::cin >> r0;
    int r1 = 20;
    int r6 = 127;
    int r2 = r0 - 20;
    if(r2 >= 0) r2 &= r6;
    r1 += r2;
    int r7 = fib[r1];
    std :: cout << r7;
    return 0;
}
```

实验思考

Lab2和Lab3都是数列递推的计算，在C++实现数列递推的时候我们也常用相同的移位寄存器法进行计算。但不论是汇编语言还是高级语言，对于存在对某个数取模的运算的情况下，查表是可行且最为有效的方法。

这两次实验是关于算法的实验，不论是在汇编层面，还是在C++层面，一个算法的优劣事实上都是可以直观的评判的，而由于高级语言可能对某些复杂运算有很好的封装，因此在汇编层面评判算法是更有效的。

Lab4 Task1

程序设计

```
#include <iostream>
void fun1()
{
    r0 = r0 + 1;
    r1 = memory[10];
    r1 -= 1;
    memory[10] = r1;
    if(r1 != 0) fun1();
    return;
}
```

```
int r0, r1;
short memory[11] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5};
int main()
{
    r0 = 0;
    fun1();
    return 0;
}
```

实验思考

这次的二进制代码事实上是一个最基本的递归应用，在C++中的实现就是自己调用自己的函数，递归终止条件是memory某位置值是否减为0。我在这里使用一个数组进行栈的模拟，而r2就是栈顶指针。由于这个栈只存储了PC的值，但C++中是不能直接访问PC的值的，因此这个栈中并没有存储实际的数据，这里我只用r2对栈顶指针进行了一个模拟，具体的递归操作还是依靠了C++的自带递归机制。

这次实验让我理解了递归的底层实现，但递归是需要对PC值进行访问的，这里就可以看出高级语言对于PC的保护措施——不能直接访问PC的值，从而很好地保护了PC不能轻易被非法地篡改。

Lab4 Task2

程序设计

```
#include <iostream>
short Divide8(short num)
{
    short result = 0;
    short n = 1;
    short r = 8;
    while(r != 0)
    {
        if(r & num) result += n;
        n += n;
        r += r;
    }
    return result;
}
int main()
{
    short r1;
    std :: cin >> r1;
    short div, r2;
    do
    {
        div = Divide8(r1);
        r2 = r1 & 7;
        r1 = div + r2;
    }
    while(r1 > 7);
    if(r1 == 7) r1 -= 7;
    std :: cout << r1;
    return 0;
}
```

实验思考

一个在C++中一句就可以实现的模7运算，其低层实现却远没有那么简单。如果采用不断减7直至这个数在0-6之间的方法取模，势必会带来巨大的时间开销。借助一个较好实现的模8运算来完成模7运算，与Lab1中的快速乘法有一定的异曲同工之妙。

LC3中并没有直接定义“右移”运算，这可能给除以8带来一定的困难，但这里依然可以使用双位指针构造的方法进行左移3位（也就是除以8）的运算。但我认为，在底层实现右移操作并不会非常困难（移位寄存器就可以实现），所以这种算法对于定义了右移运算的ISA来说，可以节省大把的时间开销。

事实上，我了解到许多高级语言中，模n的运算是依靠除以n的运算完成的，但除以n依然是一个时间开销很大的运算。虽然我们可以借助牛顿迭代法快速求倒数并将除法转化为乘法进行运算，但我们在高级语言高速运行下的设计依然要尽量避免除法和取模运算。

Lab5

程序设计

```
#include <iostream>
short r0, r1, r2, r3, r4, r5, r6, r7;
void Pow()
{
    r6 = r2;
    r3 = 0;
    r4 = r3 + 1;
    do{

        r5 = r4 & r2;
        if(r5 != 0) r3 = r3 + r6;
        r6 += r6;
        r4 += r4;
    }while(r4 != 0);
    return;

}
void Mod()
{
    r3 = ~r2;
    r3 += 1;
    r4 = 0;
    do{
        r4 += r3;
        r5 = r4 + r0;
    }while(r5 > 0);
    return;

}
void Judge()
{
    do{
        Pow();
        r3 = ~r3;
        r3 += 1;
        r3 = r0 + r3;
        if(r3 < 0) goto OVER;
```

```
        Mod();
        if(r5 == 0) goto IsNotPrimer;
        r2 += 1;
    }while(1);
IsNotPrimer: r1 = 0;
OVER:   return;

}
int main()
{
    r1 = 0;
    r2 = r1 + 2;
    r1 += 1;
    std::cin >> r0;
    Judge();
    std::cout << r1;
    return 0;
}
```

实验思考

这是一个最基础的判断质数的算法，也是对于之前实验的一个综合运用，比如乘方运算和取模运算。这次实验给我最大的启示是：遇到较难实现的部分，可以将其独立出来单独进行设计，这样在思路上较为简单，程序结构也比较直观。

事实上，在这次实验的改写过程中，我也注意到了循环对任意数取模的这个运算。逻辑与运算只能简化对2^n取模的运算，对于任意数的运算，如果我们不能很简便地调用除法运算，那么累减也是一个可行的思路。这里也再次强化了Lab4的体会：如果十分追求程序的运行效率，我们还是要尽量避免取模运算。

实验总结

本次实验使我明晰了汇编/二进制代码和高级语言代码的联系，同时也对高级语言中“隐形”的时间开销有了一定的概念。同时，在了解汇编语言的同时，我也能更好的对一些基本的算法优劣进行更加准确的评定。

对于本实验的几个重要思考如下：

1. 如何评价使用高级语言编写的程序的优劣？

   A：程序运行都是依靠指令进行的，因此我们在大体上可以根据该程序的时间复杂度和空间复杂度来进行评测，但如果要更加详细，我们还需要了解某些运算在高级语言中的实现复杂度，因为某条语句执行的指令数可能会远远大于另一条语句。

2. 为什么高级语言会使得编写程序更加容易？

   A：因为通过一些良好的封装，高级语言会更加贴合“人的语言”，这会使得编写程序不需要特别考虑计算机的实现，这自然会使得编写程序更加容易。

3. 希望在LC3中加入什么指令？

   A：我希望加入乘法和除法指令。这两种运算作为四则运算的重要成员，在程序中的应用频率是很高的，而且加入除法后可以实现取模运算，这增强了编写LC3汇编程序的简便性。

4. 对于高级语言，从LC3中学到了什么？

A：我了解到了一些高级语言中的基本的运算的底层实现方法，同时也理解到了：程序执行效率并不一定由执行语句的多少决定，执行的指令条数以及指令周期需要占用的时钟周期数是更为重要的评判标准。

本次实验也是本课程的最后一个基础实验。在本学期的实验中，我逐渐了解了一些底层的实现方法，这几次实验也让我对计算系统有了更加深刻、更加准确的理解。很感谢这门课程带给我对计算机的深层理解，我也坚信在未来，我能够运用好这门课带给我的知识，去研究计算机这门学科。