Lab 6 实验报告

- 姓名：谭旭
- 学号：PB20030775

实验内容

使用高级语言实现 lab1 ~ lab5 。

实验过程

lab0l

lab01 的实现原理为对 R1 进行累减，直到 R1 为 0，每次令结果加上一个 R0 。

其 C++ 代码如下

```
#include <cstdint>
#include <iostream>

using std::cin;
using std::cout;

int main(int argc, char* argv[])
{
    uint16_t r0, r1, r7 = 0;
    cin >> r0 >> r1;
    while (r1)
    {
        r7 = r7 + r0;
        r1 = r1 - 1;
    }
    cout << r7;
    return 0;
}
```

lab0p

lab0p 在追求运算效率的情况下采用了类似快速幂的位运算算法，用 R0 与 R2 的结果判断当前位进行乘法运算后结果为 R1 还是 0 。

每次与运算后令 R1 和 R2 都左移一位。此处左移位操作通过一行加法实现。

C 语言代码如下

```
```

```c
#include <stdint.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
    uint16_t r0, r1, r2 = 1, r7 = 0;
    scanf("%hu %hu", &r0, &r1);
    while (r2)
    {
        if (r0 & r2)
        {
            r7 = r7 + r1;
        }
        r2 = r2 + r2;
        r1 = r1 + r1;
    }
    printf("%hu", r7);
    return 0;
}
```

fib

初版 fib 程序使用了朴素的递推的方法求出 $F(n + 2)$ ，并在此基础上可以通过查询 $F(n + 2 - 2)$ 得到 $F(n)$ 。

在使用高级语言改写后，仍然保持了这一点，省去了对特殊情况的判断。

C# 代码如下

```c#
using System;

namespace Fib
{
    class Fib
    {
        static void Main(string[] argv)
        {
            UInt16 r0, r1 = 1023, r4 = 2, r5 = 0, r6 = 1, r7 = 1;
            r0 = UInt16.Parse(Console.ReadLine());
            do
            {
                r5 = r4;
                r4 = (UInt16)(r4 + r7);
                r4 = (UInt16)(r4 + r7);
                r7 = r6;
                r6 = r5;
                r0 = (UInt16)(r0 - 1);
            }while (r0 > 0);
            r7 = (UInt16)(r7 & r1);
            Console.WriteLine(r7);
```

```
        }
    }
}
```

可以看出上述程序中 r4 记录的值为 $F(n + 2)$ ， r7 记录的是 $F(n)$ 。

fib-opt

该程序为 fib 程序的优化版，通过预先计算出要求范围内所有的 $F(n)$ ，并将其保存在内存中，在查询时直接跳转的方式，使得程序运行指令数变为一个固定值，省去了在程序运行过程中的计算耗时。

使用高级语言重写该程序时，仍采用预先计算的方法。

首先使用一程序，通过朴素的递推得到 $F(0) ~ F(14000)$ ，并将结果输出到一个文件中。

JavaScript 代码如下

```javascript
var n = 14000;
var f = [1, 1, 2];
for (var i = 3; i <= n; i++)
{
    f[i] = (f[i - 1] + 2 * f[i - 3]) % 1024;
}
for (var i = 0; i <= n; i++)
{
    document.write(f[i] + ", ");
}
```

将网页输出的结果复制到另一程序的数组中，对于每个询问 R0 ，直接访问数组中对应下标即可。

Go 代码如下

```go
package main

import "fmt"

func main() {
    var ansList = [14001]uint16{/* Here puts all F(n) */}
    var r0 uint16
    fmt.Scan(&r0)
    fmt.Print(ansList[r0])
    return
}
```

rec

rec 程序实现了一个函数的递归，每次递归 R0 自增， R1 自减。

使用 lc3 实现递归时需要手动实现栈，较为困难。

而使用高级语言时则不需要考虑这些，直接在函数内部调用自身即可。

Python 代码如下

```
global r0, r1
r0 = 0
r1 = 5

def Foo():
    global r0, r1
    r0 = r0 + 1
    r1 = r1 - 1
    if r1 != 0:
        Foo()

Foo()
```

由于原题目中并没有明确输出，上述改写后的程序也没有输出。

mod

将原本的 mod 二进制代码文件翻译为汇编语言后如下

```
LD R1 x3016
JSR x300A
AND R2, R1, #7
ADD R1, R2, R4
ADD R0, R1, #-7
BRp x3001
ADD R0, R1, #-7
BRn x3009
ADD R1, R1, #-7
HALT
AND R2, R2, #0
AND R3, R3, #0
AND R4, R4, #0
ADD R2, R2, #1
ADD R3, R3, #8
AND R5, R3, R1
BRz x3012
ADD R4, R2, R4
ADD R2, R2, R2
ADD R3, R3, R3
BRp x300F
RET
x0120
```

可以看出， `x300A` 到 `x3015` 为一个函数，并且在这个函数内，经过对寄存器的初始化后，存在一个循环 `x300F` 到 `x3014` 。

除此之外，程序中并没有其他函数。

考虑将该部分封装为高级语言中的一个函数，结果如下

```
public static short r0, r1, r2, r3, r4;

public static void Foo()
{
    r2 = 0;
    r3 = 0;
    r4 = 0;
    r2 = (short)(r2 + 1);
    r3 = (short)(r3 + 8);
    do
    {
        if ((r3 & r1) != 0)
        {
            r4 = (short)(r2 + r4);
        }
        r2 = (short)(r2 + r2);
        r3 = (short)(r3 + r3);
    }while (r3 > 0);
}
```

通过汇编代码了解到，主函数应当一开始就进入一个循环，并在循环内部调用上述函数，直到 r1 不大于 7 。

并且在循环结束后，判断 r1 是否等于 7 ，若等于 7 则 r1 减去 7 。

主函数如下

```
public static short r0, r1, r2, r3, r4;

public static void main(String[] args)
{
    do
    {
        Foo();
        r2 = (short)(r1 & 7);
        r1 = (short)(r2 + r4);
    }while (r1 - 7 > 0);
    if (r1 == 7)
    {
        r1 = (short)(r1 - 7);
    }
}
```

总的 Java 代码如下

```java
import java.util.Scanner;

public class mod
{
    public static short r0, r1, r2, r3, r4;

    public static void Foo()
    {
        r2 = 0;
        r3 = 0;
        r4 = 0;
        r2 = (short)(r2 + 1);
        r3 = (short)(r3 + 8);
        do
        {
            if ((r3 & r1) != 0)
            {
                r4 = (short)(r2 + r4);
            }
            r2 = (short)(r2 + r2);
            r3 = (short)(r3 + r3);
        }while (r3 > 0);
    }

    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        r1 = input.nextShort();
        do
        {
            Foo();
            r2 = (short)(r1 & 7);
            r1 = (short)(r2 + r4);
        }while (r1 - 7 > 0);
        if (r1 == 7)
        {
            r1 = (short)(r1 - 7);
        }
        System.out.print(r1);
    }
}
```

prime

在使用 lc3 实现该程序时，并没有涉及到对 R0 的开方操作，而是把所有不大于 R0 的数据都计算了一遍。

在使用高级语言改写时，仍没有进行开方。

另外，该程序中的取模运算是由朴素的减法实现的。

Shell 代码如下

```
#!/bin/bash

read r0
r1=true
for ((r2 = $r0-2; r2 > 1; r2 -= 2)); do
    r3=$r0
    while [[ r3 -gt 0 ]]; do
        r3=$(($r3-$r2))
    done
    if [[ r3 -eq 0 ]]; then
        r1=false
    fi
done
echo $r1
read r7
```

实验思考

使用高级语言实现 lab1 ~ lab5 的过程中，可以感受到高级语言相较于 lc3 的汇编语言主要有以下优点：

- 能够自由定义变量
- 能够定义函数，并且可以实现函数递归
- 具有多样的控制结构

因此，使用高级语言编写递归、循环、判断的部分时，比 lc3 汇编语言更简便。

考虑到 lc3 的每一条指令实现过程都不会过于复杂，因此，为 lc3 添加乘法、除法、取模指令或许是不现实的。但是，可以考虑为 lc3 添加左移位、右移位指令，这样能够在求积、求幂的过程中提供很多便利。