中国科学技术大学计算机学院
《计算机系统概论实验报告》

实验题目：Learn from the past

学生姓名：叶子昂

学生学号：PB20020586

完成时间：2021年1月5日

实验目的：

> 用C程序重写LC3程序

实验环境：

- window11 和 wsl2
- VScode

实验过程：

实验一：

> 用LC3指令实现乘法，分别对程序指令数，和运行指令数优化。

L版本：通过对整数二进制补码性质的利用减少对乘数为正数，负数，零的讨论

```
int main(){
    short R7=0,R0=0,R1=0;
    cin>>R0>>R1;
    do{
        R7+=R0;
    }while(--R1!=0);
    cout<<R7<<endl;
    return 0;
}
```

P版本：通过掩码优化移位实现乘法，减少循环次数。

```
short R[8]={0};

int main() {
    R[8]={0};
    cin >> R[0] >> R[1];
    for(R[2] = R[7]-1,R[3] = R[7]+1,R[1]=R[1]&R[2];//initialization
        R[1];R[1]=R[1]&R[2]){//每次循环执行判断
        R[4]=R[1]&R[3];
        if(R[4]!=0)R[7]+=R[0];
        R[0]=R[0]<<1;
        R[2]=R[2]<<1;
        R[3]=R[3]<<1;
    }
    cout<<R[7]<<endl;
    return 0;
}
```

实验二：

> 求出类斐波那契数列的指定项。

减少指令数通过使用其递推公式迭代。

```
short R[8]={0};

int main() {
```

```
    cin>>R[0];
    R[1]=1;//枚举0，1，2
    R[2]=1;
    R[3]=2;
    R[5]=1023;
    R[7]=R[1];
    R[0]--;
    if(R[0]<0) {goto result;}
    R[7]=R[2];
    R[0]--;
    if(R[0]<0) {goto result;}
    R[7]=R[3];
    while((--R[0])>=0) {//迭代
        R[4]=R[1]<<1;
        R[7]=R[3]+R[4];
        R[7]=R[7]&R[5];//取模
        R[1]=R[2];
        R[2]=R[3];
        R[3]=R[7];
    }
result:
    cout<<R[7]<<endl;
    return 0;
}
```

实验三：

> 优化实验二的运行指令数。

通过对类斐波那契数列的结果进行归纳，总结规律“打表”建立对应关系大幅减少运行指令数。

```
int main(){
    short FILL[148]={
     1
    ,1
    ,2
    ,4
    ......//打表部分省略在源码中
    };
    short R[8]={0};
    cin >> R[0];
    R[2]=-19;
    R[3]=R[0]+R[2];
    if(R[3]<=0){//直到第十九个元素
        R[7]=FILL[R[0]];
    }
    else{//20以后的元素
        R[2]=127;
        R[3]=R[3]&R[2];
        R[7]=FILL[19+R[3]];
    }
    cout<<R[7]<<endl;
    return 0;
}
```

实验四：

> 修复缺失的代码。

rec为一个递归程序，进行反复调用到指定次数再根据保留的R7回退

```
const short addmain = 12291;//0x3003; address for R[7] to back to main program
const short addjsr = 12300;//子程序中调用自己时应返回的位置
const short address = 12303;//address of 0x300f
short num = 5;//存在程序结尾的数

int main(){
//begin program
    R[2]=address;
    R[7]=addmain;
    recursion();
//end
    for(int i=0; i<8;i++){//show general reg
        cout<<"R"<<i<<":"<<R[i]<<" ";
    }
    cout<<endl;
    for(int i=0;i<10;i++){//show blkw
        cout<<blkw[i]<<endl;
    }
    cout<<endl;
    return 0;
}

void recursion(){
    blkw[R[2]-address]=R[7];
    R[2]++;
    R[0]++;
    R[1]=num;
    R[1]--;
    num=R[1];
    R[7]=addjsr;
    if(num!=0)recursion();//JSR
    R[2]--;
    R[7]=blkw[R[2]-address];
    return ;//ret
}
```

mod为求7的余数的程序

```
short R[8]={0};
void getquotient();

int main(){
    cin>>R[1];
    do{
    getquotient();
    R[2]=R[1]%8;
    R[1]=R[2]+R[4];
    }while(R[1]>7);
    if(R[1]<7){
        cout<<R[1]<<endl;//print result
    }
```

```
    else {R[1]-=7;cout<<R[1]<<endl;}//print result
    return 0;
}

void getquotient(){
    R[2]=0;R[3]=0;R[4]=0;
    R[4]=R[1]/8;//得商
}
```

实验五：

> 用LC3程序实现C++程序的功能。

判断是否为质数

```
short R[8]={0};

int main(){
    cin>>R[0];
//begin of the program
    R[2]=2;//i
    R[1]=1;
    while(R[2]*R[2]<=R[0]){//在汇编代码中采用第一次实验的p版本求i*i不再重复实现
        R[5]=0;
        R[6]=-R[0];
        while((R[5]+R[2])>0){
            R[5]=R[5]+R[2];//得到不同倍数的R[2]与待判断数比较
            R[3]=R[5]+R[6];
            if(R[3]==0) break;//跳出除断，此时R[3]=0
        }
        if((R[5]+R[2])<=0){R[3]=1;}//不能除断，R[3]=1
        if(R[3]==0){
            R[1]=0;
            break;
        }
        R[2]++;
    }
//end of program
    cout<<R[1]<<endl;
    return 0;
}
```

对几个问题的思考

- 我在对LC3小型计算机和LC3机器码和汇编语言学习之前，编程主要通过C/C++完成，由于现代的电脑普遍运行速度很快而大一所需要解决的问题的规模通常很小，任何方法乃至于枚举都能快速得到答案，除了老师教学时特别提到的算法通常只关心问题能不能解决并不关心怎么实现和效果如何。在学习LC3机器码和汇编语言学习后做Lab1时，第一次在“底层”切身体会到同样是乘法，在数据并不大的情况下运行指令数能有如此大的差异，综合前面的实验我总结了以下优化方法。

  优化程序长度的方法：

  - 利用性质减少分类讨论
  - 发现利用规律使问题分解为能循环或递归解决的子问题

  优化运行效率的方法：

  - 移位操作对部分计算和逻辑判断问题有很好的效果

    - 发现利用规律将大问题通过取模等收缩为小问题优化掉不必要的循环（重复任务）
    - 将能在运行前处理的问题尽量运行前处理，例如通过“打表”事先计算好部分结果和对应关系，运行时只需通过对应关系找到结果即可

  （不过很多时候优化程序长度和优化运行效率是冲突的）

- 在使用LC3机器码和和汇编语言编程时经常发现很多在高级语言中拿起来就能用的库函数甚至于基本运算在LC3中都无法直接实现，LC3提供的操作和功能很少。需要重新造“轮子”甚至于造“轮子”的过程中得造“轮子的轮子”，需要关注很多细节问题，有些过于麻烦而不得不更改思路。
  LC3上实现逻辑控制跳转多涉及到定位，机器码的地址计算相当烦人且由于代码全为01组合易读性很差，汇编语言好不少但LABEL的大量使用随意跳转也会提高设计时的出错率和降低易读性。
  高级语言针对LC3汇编语言的问题则有以下优点
    - 隐藏大部分涉及硬件以及具体存储的操作和实现，只需请求操作系统会自动完成。
    - 逻辑控制丰富直观，程序跳转调用只需call即可，在清晰变量的作用域前提下，可以不过多关心变量保存。
    - 类人类自然语言在规范编程的前提下，代码结构清晰易读性维护性强。
    - 很多基本操作和常用功能被高级语言内置，或存在于标准库和其他库文件中，它们通常效率高安全性强泛用性好，在需要时只需直接到用即可有很好的效果。
    - 高级语言编程通常易于配置良好的编程环境，在IDE和插件的帮助下能够很大提高效率。
- 1. 取余指令：由于正数二进制补码的特性，对2的正数次方取余比较简单，而对其他数取余则比较困难需要不少其他指令组合和发现利用特性（有时候特性也并不通用）
  2. 右移指令（且根据原数最后一位设置条件码）：左移通过ADD易于实现，而右移指令实现较为复杂但移位操作对底层语言很常用，同时根据原数最后一位设置条件码来得知舍弃的位。
- C/C++是高级语言中相对接近底层的语言提供指针对内存进行操作，或许可以提供用户打指针标记功能（扩充原有头尾和偏移定位）实现快速跳转。C/C++在编译后也会最终转为汇编码直至机器码，在通常环境下这些汇编语言通常包含LC3的全部功能并提供包括如乘法在内的更多功能。过于接近底层容易涉及很多硬件问题，个人认为不必再为C/C++添加更多的底层特性。

实验总结与思考：

- 本次实验与lab5相反是将前面的实验用高级语言重新实现，高级语言对很多细节进行了包装抽象，具体按原汇编程序向C++程序翻译时部分操作会很别扭，尤其是涉及到定位和跳转。不过总体来说，高级语言操作和控制逻辑更丰富，对前面很多汇编程序需要很多条指令才能完成的任务只需一行代码就能完成，同时也有更丰富的手段不必拘泥于LC3稀少的指令。
- 通过本次实验的编程和对几个问题的思考，我体会到了低级语言解决复杂问题的困难，和高级语言控制底层的不适，对底层功能实现的了解有助于我对高级语言的部分问题的重新认识。综合前几次实验提高了个人优化程序的意识（而非能跑就行），了解了很多特殊问题的快速处理方法，也提高了代码的阅读能力收获颇丰。