Lab6 Learn From the Past

赖永凡 2022.1

实验要求

使用高级语言来重新实现之前写过的代码，注意使用到的算法需与之前一致。

程序列表：

- lab0l (lab1 L version)
- lab0p (lab1 P version)
- fib (lab2 fibonacci)
- fib-opt (lab3 fibonacci)
- rec (lab4 task1 rec)
- mod (lab4 task2 mod)
- prime (lab5 prime)

同时，思考以下问题：

1. How to evaluate the performance of your own high-level language programs
2. Why is a high-level language easier to write than LC3 assembly
3. What instructions do you think need to be added to LC3? (You can think about the previous experiments and what instructions could be added to greatly simplify the previous programming)
4. Is there anything you need to learn from LC3 for the high-level language you use?

实验内容

```c
//lab0l
#include <stdio.h>

int main(){
    short R0=-1000;
    short R1=-233;
    short R7;
    if(R0 == 0) R7 = 0;
    else{
        while(R0 != 0){
            R7+= R1;
            R0--;
        }
    }
    printf("%d\n",R7);
    return 0;
}
```

lab0l使用的是无符号数的方法，因此在C中实现起来也同样可行，当时我还多判断了一下R0是否为零，其实这个也可以省略。

```c
//lab0p
#include <stdio.h>
```

```
int main(){
    short R0=0;
    short R1=1;
    short R7;
    short R2;
    short R3;
    short R4;

    R2 = 1;
    R3 = R2 & R0;
    if(R3 != 0) R7 += R1;//bit[0]

    R4 = R1 + R1;
    R2 += R2;
    R3 = R2 & R0;
    if(R3 != 0) R7 += R4;//bit[1]

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
```

```
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    R4 += R4;
    R2 += R2;
    R3 = R0 & R2;
    if(R3 != 0) R7 += R4;

    printf("%d\n",R7);
}
```

lab0p使用的是循环展开的方法，在高级语言里实现的话，同样会出现大量重复的循环节

```
//fib
#include <stdio.h>

int main(){
    short R0 = 2;
    short R1;
    short R2;
    short R3;
    short R4;
    short R5;
    short R6;
    short R7 = 0;

    R1 = 1;
    R2 = 1;
    R3 = 2;
    R5 = 0x03ff;

    R0 = R0 - 2;
    if(R0 <= 0){
        if(R0 == 0) R7 +=1;
        R7 += 1;
        printf("%d\n",R7);
```

```
        return 0;
    }

    while(R0)
    {
        R4 = R1 + R1;
        R7 = R4 + R3;
        R7 = R7 & R5;
        R1 = R2;
        R2 = R3;
        R3 = R7;
        R0--;
    }
    printf("%d\n",R7);
}
```

fib使用的是迭代的算法，由于是从LC3汇编语言直接翻译过来的，所以在初始条件的判别处显得十分笨拙。

```
//fib-opt
#include <stdio.h>

int main(){
    short R0 = 18;
    short R1;
    short R2;
    short R3;
    short R4;
    short R5;
    short R7;

    R5 = 0x3ff;
    R1 = 1;
    R2 = 1;
    R3 = 2;
    R7 = 1;

    R0 -= 2;
    if(R0 == 0){
        R7 = R3;
        printf("%d\n",R7);
        return 0;
    }
    else if(R0 < 0) {
        printf("%d\n",R7);
        return 0;
    }

    while(R0)
    {
        R7 = R3 + R1;
        R7 = R7 + R1;
        R1 = R2;
        R2 = R3;
        R3 = R7 & R5;
        R0--;
    }
```

```
    R7 = R3;
    printf("%d\n",R7);
}
```

我拿到的代码在核心算法上同样使用的是迭代的方式，不过在处理MOD1024上复杂一些，未使用AND 0x3FF 的方法，因此我只对这一处进行了优化。当然，我知道无论拿到什么算法，都可以使用打表来优化，不过这里既然已经有了可优化的点就没必要再折腾了:-)

```
//rec
#include<stdio.h>

void recursion(short *R1,short &R0)
{
    R0++;
    *R1 = *R1 - 1;
    if(*R1 != 0) recursion(R1,R0);
}

int main(){
    short R0 = 0;
    short R1 = 5;
    recursion(&R1,R0);
    printf("R0:%d\tR1:%d\n",R0,R1);
}
```

rec就可以看出高级语言的优越之处了，对于子程序的递归调用，高级语言无需维护递归栈，因此在代码上看上去直观得多。

```
//mod
#include<stdio.h>

short DIV8(short R1)
{
    short R2,R3,R4,R5;
    R4 = 0;
    R2 = 1;
    R3 = 8;
    while(R3){
        R5 = R1 & R3;
        if(R5 != 0) R4 += R2;
        R2 += R2;
        R3 += R3;
    }
    return R4;
}

int main(){
    short R1 = 288;
    short R0 = 0,R2 = 0,R4 = 0;
    do
    {
        R2 = R1 & 7;
        R4 = DIV8(R1);
        R1 = R2 + R4;
        R0 = R1 - 7;
    }while(R0>0);
```

```
    if(R0 == 0) R1 -= 7;
    printf("%d\n",R1);
    return 0;
}
```

mod程序主要完成两个步骤，除以8和mod8。这两个在高级语言中用一句指令就能够完成的操作在LC3中则需要更多的步骤，尤其是计算一个数除以8。这里则在高级语言中将LC3中的操作都还原了出来。

```
//prime
#include<stdio.h>

short Judge(short R0)
{//if R0 is a prime number, return 1, else return 0
    short R1,R2,R3,R4,R5;
    for(R1 = 1, R2 = 2;;R2++){
        R4 = R2;
        R3 = 0;
        while(R4){
            R3 += R2;
            R4--;
        }
        if(R3 > R0) return 1;
        R5 = R0;
        while(R5 > 0) R5 -= R2;
        if(R5 == 0) return 0;
    }
}

int main(){
    short R0 = 7;
    printf("R1 = %d\n",Judge(R0));
    return 0;
}
```

prime是一个经典的判断是否是质数的算法，同样，我没有使用乘法和求余数运算符，而是像LC3中那样全部拆开来完成。

实验思考

How to evaluate the performance of your own high-level language programs

就刚学完数据结构的我来说，第一反应就是看程序的时间复杂度，虽然这个说法十分笼统，但是我感觉在粗略估计时还是相当有用的。

计算时间复杂度的话，一言以蔽之，就是看程序中循环的次数以及其量级。像$lab0l$的算法就是一个典型的$O(n)$量级，$prime$中判断$i^2 \leq n$的外层循环是$O(\sqrt{n})$量级，而由于未使用$\%$运算符而是采用定义法来求余数，故内层循环实验复杂度可以近似看成$O(n)$级别，所以整个$prime$程序的时间复杂度就大致为$O(n^{1.5})$。

Why is a high-level language easier to write than LC3 assembly

最大的原因应该在于高级语言允许人们自己申明变量，使用起来也更加舒适，简洁，而无需像LC3汇编语言那样，仅仅局限于8个寄存器，虽然这一点或许没有让代码变得更加复杂，但确实让人在写的过程中不那么“$easier$”。

其次LC3的指令数比较少，且对于最简单的操作符还是用文字来呈现的，对于一个习惯于看表达式的人而言，用文字表述的直观性更是大大下降。例如：$\mathrm{R1} = \mathrm{R1} + \mathrm{R0}$显然要比$\mathrm{ADD\ R1\ R1\ R0}$让人易读。

其三就是在对内存的存取上。LC3里，需要我们自己来将某些值储存到内存中，这样就又涉及到一系列LD, ST指令使得程序更加复杂，而在高级语言中，几乎不需要我们来操心这件事。在写递归程序时我对这一点更是深有感触。

最后则在于指令体系，显然，拥有着丰富得多的指令的高级语言写起来要比只有15条指令的LC3方便。

What instructions do you think need to be added to LC3

$-, \times, \div, \%$ absolutely.

Is there anything you need to learn from LC3 for the high-level language you use

如果是在代码撰写方面，坦白而言几乎很少，毕竟高级语言写起来比LC3汇编语言方便得多，也难以在后者中学到一些$fancy\ tricks$，如果坚持要说的话，或许位运算这一被疏忽的点可以应用到今后的高级语言中。

但LC3毕竟是汇编语言，它教给我更多的不是如何去写代码，而是电脑是如何运行代码。今后在用高级语言来写程序的时候，或许我也会想想这种算法如果要用汇编码来写的话应该是怎样的，在编译运行的时候，编译器又会把我的这些代码转换成何种形式？

真正的影响或许就是，你还没意识到，它就已经深入到意识中了。