LAB.1

程序是如何设计的（how to design）：

时间线索

L版本：

首先，产生最朴素的想法，试图完成一个乘法的机器码程序，然后在进行后期的程序优化；

进行R0 R1的潜在的数据分析：

R0	R1	result
2	3	6
-2	3	-6
2	-3	-6
-2	-3	6

对于LC-3编译机器码最大的困难在于正负符号的问题；

如果两个待操作数都是正数，那么只需要进行一个简单的循环即可：

```
//机器码来源参考 LC3 TOOLS
0001 010 010 0 00 100 ; add R4 to R2, put result in R2
0001 101 101 1 00001 ; subtract 1 from R5, put result in R5
0000 100 111111101 ; branch to location x3201 if zero or positive
```

即以其中的一个操作数作为不变量，另一个操作数不断的**subtract 1**从而实现将不变量**累加**成**乘法效果**

注意：此时对不变量操作时，不需要考虑他的正负值，那么就相当于可以不对不变量考虑

所以：最终result的结果只需要调控两个量：（若此时R0为不变量）

判断R1的符号
若R1为负，取反作正数进行循环，否则直接循环

因此，增加一个存储符号的寄存器，进行步骤（**Algorithm**）：

1. 输入：输入操作数A,硬性要求对其中一个需要机器码输入，或者采用add 0 的方式
2. BR指令进行跳转到**步骤5**，如果输入的值A为0或者正数
3. 对A取反 且 +1
4. 设置标记寄存器，将其设置为-1
5. 将B存入到目标寄存器里
6. 并对A减1
7. 检验A是否为0或者正数，若是，将其跳转到**步骤5**
8. 将R3+R3放到R3,用来更改cc，若为cc=1或者0，程序结束，否则继续执行
9. 将R7取反+1并存放在R7中

机器码实现:

版本1.0**（12行）**

0011 0000 0000 0000 ; start the program at x3000
0001 010 010 1 00000 ; add 0 to R2, put result in R2
0000 011 000000011; branch to location x3005 if zero or positive
1001 010 010 1 11111;NOT R2,R2;
0001 010 010 1 00001;ADD R2,R2,#1
0001 011 011 1 11111;ADD R3,R3,#-1
0001 111 001 0 00 111;ADD,R7,R1,R7;
0001 010 010 1 11111 ; subtract 1 from R2, put result in R2
0000 001 111111101 ; branch to location x3005 if zero or positive
0001 011 011 0 00 011;ADD R3,R3,R3
0000 011 000000010; branch to location x30 if zero or positive
1001 111 111 1 11111;NOT R7,R7;
0001 111 111 1 00001;ADD R7,R7,#1
1111 0000 00100101 ; halt

版本2.0 （11行）

0011 0000 0000 0000 ; start the program at x3000
0101 000 000 1 00000; clear R0
0101 001 001 1 00000; clear R1
0101 111 111 1 00000; clear R7
0101 011 011 1 00000; clear R3
0001 000 000 1 00011; ADD R0 R0 #3
0001 001 001 1 11011; ADD R1 R1 #-5
0000 011 000000011; branch to location x3005 if zero or positive
1001 001 001 1 11111; NOT R1
0001 001 001 1 00001;ADD R1,R1,#1
0001 011 011 1 11111;ADD R3,R3,#-1
0001 111 001 0 00 111;ADD,R7,R1,R7;
0001 000 000 1 11111 ; subtract 1 from R2, put result in R2
0000 001 111111101 ; branch to location x30 if zero or positive
0001 011 011 0 00 011;ADD R3,R3,R3
0000 011 000000010; branch to location x30 if zero or positive
1001 111 111 1 11111;NOT R7,R7;
0001 111 111 1 00001;ADD R7,R7,#1
1111 0000 00100101 ; halt

L版本未完待续.....

P版本：

最初想法：试图进行一个在L的版本基础上，对循环+1过程的改进

例如 5*400=2000

正常循环应该是400+400+400+400+400

我先用判断大小的程序将5作为循环位，400变成不变量如果我采取

400+800+400的话，那会很好的优化行数

400+800+1600+3200+6400的想法

即每循环一次，都让R7+R7 ->R7中，

但这个过程中，需要考虑大量的问题，于是在多番尝试后**放弃。**

第二个想法：

移位：

类似于对十位数乘法的操作

甚至不需要考虑两个操作数的正负！

```
    0 0 1 0      2
    1 1 0 1     -3
    0 0 1 0
  0 0 0 0
0 0 1 0
0 0 1 0
  1 0 1 0       -6
```

代码也更简洁方便：

P版本　（7行）

指令行 对 R0 R1 输入 A B 版本

```
0011 0000 0000 0000 ; start the program at x3000
0001 000 000 1 11101  ;ADD,R0 R0 A  11101
0001 001 001 1 00010 ;ADD R1 R1 B  00010
0001 010 010 1 00001 ;ADD R2 R2 #1
0101 011 001 0 00 010 ;AND R3 R1 R2
0000 010 000 0 00 001 ;BRz jump 1 line
0001 111 000 0 00 111 ;ADD R7 R0 #0
0001 010 010 0 00 010 ;ADD R2 R2 R2
0001 000 000 0 00 000 ;ADD R0 R0 R0
0000 101 111 1 11010 ; BRnp jump 5 up lines
1111 0000 00100101 ; halt
```

手动清零输入 （7行）

```
0011 0000 0000 0000 ; start the program at x3000
0001 010 010 1 00001 ;ADD R2 R2 #1
0101 011 001 0 00 010 ;AND R3 R1 R2
0000 010 000 0 00 001 ;BRz jump 1 line
0001 111 000 0 00 111 ;ADD R7 R0 #0
0001 010 010 0 00 010 ;ADD R2 R2 R2
0001 000 000 0 00 000 ;ADD R0 R0 R0
0000 101 111 1 11010 ; BRnp jump 5 up lines
1111 0000 00100101 ; halt
```

P版本结束

最初：7行

最终：7行

续L版本：直接使用P版本代码，同一代码既满足**L**也符合**P**

L版本结束

最初：11行

最终：3行

测试&指令条数

测试：样例通过

指令条数：

1. **0001 010 010 1 00001 ;ADD R2 R2 #1**
2. **0101 011 001 0 00 010 ;AND R3 R1 R2**
3. **0000 010 000 0 00 001 ;BRz jump 1 line**
4. **0001 111 000 0 00 111 ;ADD R7 R0 #0**
5. **0001 010 010 0 00 010 ;ADD R2 R2 R2**
6. **0001 000 000 0 00 000 ;ADD R0 R0 R0**
7. **0000 101 111 1 11010 ; BRnp jump 5 up lines**

指令1.每个程序只运行一次

指令2-7 考虑最坏的情况，即16位的操作数一直移位，整个2-7循环16次

则指令条数为 **1+6*16=97**

N天之后的L版本！

事实证明**三行**就可以完成L版本代码

Chapter 4: Debugging programs in the simulator

Now that you've experienced the ideal situation of seeing a program work perfectly the first time, you're ready for a more realistic challenge – realizing that a program has a problem and trying to track down that problem and fix it.

The following are a series of steps that are common while debugging any program, but we will work with a single example so it is easier to follow along.

Example 1: Debugging a program to multiply without a multiply instruction

The Problem Statement

Our goal is to multiply the values in R4 and R5 and store the result in R2.

Entering your program in machine language

Enter the following program into the editor.

```
0011 0010 0000 0000    ; start the program at x3200
0101 010 010 1 00000   ; clear R2
0001 010 010 0 00 100  ; add R4 to R2, put result in R2
0001 101 101 1 11111   ; subtract 1 from R5, put result in R5
0000 011 111111101     ; branch to location x3201 if zero or positive
1111 0000 00100101     ; halt
```

在教程里debug部分的第一个代码块的三行代码就可以解决L版本所遇到的问题，这种方法是利用了二进制机器码的溢出特性，但是缺点也是显著的，因为当进行负数运算的时候，要进行大量的指令计算！

代码：

```
0011 0000 0000 0000 ; start the program at x3000
0001111111000000
0001001001111111
0000101111111101
1111 0000 00100101 ; halt
```