Report_lab1　实现 LC-3 机器乘法

PB20151793 宋玮

part 1: L 版本程序

1.设计思路

L 版本程序要求尽量编写更少的代码行数。

最终程序行数：11 行（如下图）

***大致思路*：**利用其中一个乘数（R0）作为“计数器”（count），对另外一个乘数(R1)进行加操作。

首先判断 R0 正负：

如果是正数（或 0），则每次对 R7 进行加 R1 操作后，R0 减 1，重复操作，直至 R0 减为 0。由于该过程对 R1 加了 R0 次。因此 R7 中的结果为 R0×R1。

如果是负数，则每次对 R7 进行加 R1 操作后，R0 加 1，重复操作，直至 R0 变为 0。最后对 R7 中的结果取反加一，可得到 R0×R1。

```
0001 010 000 1 00000 ; R2 = R0
0000 100 000000100 ;判断R2是否为负，如果为负，跳转至第二段
0000 010 000001000 ;判断R2是否为0，如果为0，跳转至halt
0001 111 111 000 001 ;R7=R7+R1
0001 010 010 1 11111 ;R2减1
0000 011 111111100 ;判断R2是否为0或正数

0001 111 111 000 001 ;R7=R7+R1
0001 010 010 1 00001 ;R2加1
0000 100 111111101 ;判断R2是否为负数
1001 111 111 111111 ;R7取反
0001 111 111 1 00001 ;R7加1
```

***测试数据*：**（此处仅列举两组数）

（1）-500 * 433 （刻意溢出）

初始状态

Registers

R0	xFE0C	-500
R1	x01B1	433
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0000	0
PSR	x8002	-3276 CC: Z
PC	x3000	12288
MCR	x8000	-3276

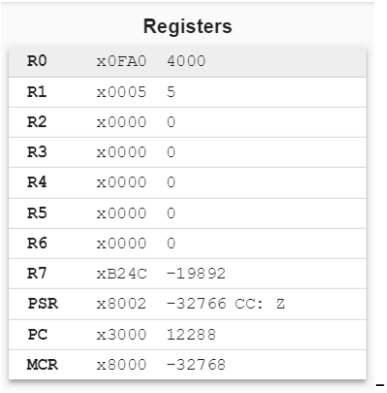
运行后结果

Registers

R0	x0000	0	
R1	x7FFF	32767	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x2FFE	12286	
R7	xB24C	-19892	
PSR	x0002	2	CC: Z
PC	x0263	611	
MCR	x0000	0	

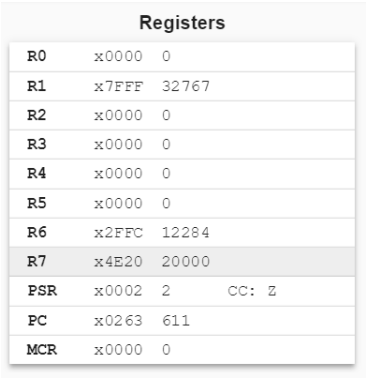
（2）4000 * 5

初始状态



Registers		
R0	x0FA0	4000
R1	x0005	5
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	xB24C	-19892
PSR	x8002	-32766 CC: Z
PC	x3000	12288
MCR	x8000	-32768

运行后结果



Registers		
R0	x0000	0
R1	x7FFF	32767
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x2FFC	12284
R7	x4E20	20000
PSR	x0002	2 CC: Z
PC	x0263	611
MCR	x0000	0

2.改良过程

起初我想采用移位方式进行乘法的计算。即对其中一个乘数进行左移操作（左移 15 位，14 位，13 位……）,另一个乘数每次左移一位，判断正负，以决定是否进行加操作，该过程即列竖式乘法计算，**详见 p 版本程序**。

经过一系列改动，该思路写出的程序最短 17 行（如下图）。不符合要求。

```
0001 011 011 1 01111 ; R3 = 15
0001 010 000 1 00000 ; R2=R0
0001 100 011 1 00000 ; R4=R3
0000 010 000000011 ;
0001 010 010 000 010 ; R2 =R2 + R2, MOVE ONE BIT LEFT
0001 100 100 1 11111 ;R4 =  R4-1
0000 001 111111101 ; IF R4 IS POSITIVE, JUMP TO R2= R2+R2;

0001 010 010 1 00000 ; R2 = R2
0000 100 000000100 ;
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0001 011 011 1 11111 ; R3 = R3-1
0000 011 111110101 ; IF R3 IS POSITIVE OR ZERO, JUMP TO R2= R0
0000 111 000000100 ; JUMP TO HALT;

0001 111 111 000 001; R7 = R7 + R1
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0001 011 011 1 11111 ; R3 = R3-1
0000 011 111110000 ; IF R3 IS POSITIVE OR ZERO, JUMP TO R2= R0
```

因此，我改变思路，即最终方案为上述 **1.设计思路**中的方案，**程序最短为 11 行。**

```
0001 010 000 1 00000 ; R2 = R0
0000 100 000000100 ;判断R2是否为负，如果为负，跳转至第二段
0000 010 000001000 ;判断R2是否为0，如果为0，跳转至halt
0001 111 111 000 001 ;R7=R7+R1
0001 010 010 1 11111 ;R2减1
0000 011 111111100 ;判断R2是否为0或正数

0001 111 111 000 001 ;R7=R7+R1
0001 010 010 1 00001 ;R2加1
0000 100 111111101 ;判断R2是否为负数
1001 111 111 111111 ;R7取反
0001 111 111 1 00001 ;R7加1
```

part 2: P 版本程序

1.设计思路

P 版本尽量让程序执行更少的指令。
最终程序平均执行指令条数约为：132 行

大致思路：通过移位和加法模拟列竖式计算。如下图：

```
          A16 A15 A14 ..... A3 A2 A1
×         B16 B15 B14 ..... B3 B2 B1
-------------------------------------
          A16 A15 A14 ..... A3 A2 A1  (当B1为1)
   0      0     0  ..... 0    0  0    (当B2为0)
.....
-------------------------------------
          C16 C15 C14 ..... C3 C2 C1
```

由于 R0,R1 均为 01 串。因此在进行列竖式乘法计算时，每一行结果要么是 R0，要么是 0，取决于 R1 对应位上是 1 还是 0。再通过对 R0 的移位后相加，则可以得到结果。

目前，仅能实现的移位方式是左移，即通过 R0=R0+R0，可以实现对 R1 的左移一位。因此，本算法也采取自加左移方式。

而判断 R1 对应位上是 0 还是 1，也需要对 R1 进行左移，然后通过判断正负，判断该位上是 0 还是 1。

注：当 R1 为负数时，采取 R1 和 R0 同时取反加一的操作，即同时取相反数，可以缩减指令执行条数。

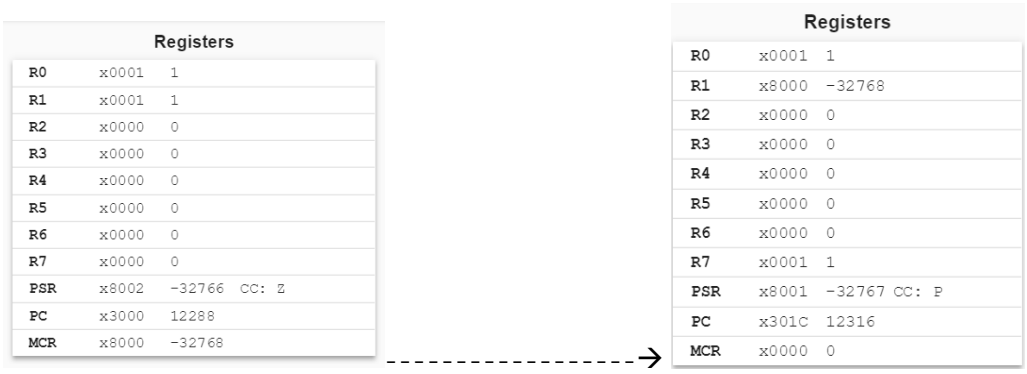
代码如下：

```
0001 011 011 1 01111 ; R3 = 15
0001 001 001 1 00000 ; R1 = R1
0000 011 000000100 ; 判断R1是否为负，如果为负，则执行下面的R0取反加一，R1取反加一
1001 001 001 111111;
1001 000 000 111111;
0001 001 001 1 00001;
0001 000 000 1 00001;

0001 011 011 1 11111 ; R3 = R3-1
0000 110 000001001 ; IF R3 IS NEGATIVE OR ZERO, JUMP TO JUDGE R3
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0000 011 111111100 ; JUMP TO  R3 = R3-1

0001 010 000 1 00000 ; R2=R0
0001 100 011 1 00000 ; R4 = R3
0001 010 010 000 010 ; R2 = R2 + R2, MOVE ONE BIT LEFT
0001 100 100 1 11111 ;R4 =  R4-1
0000 001 111111101 ; IF R4 IS POSITIVE, JUMP TO R2 = R2+R2;
0001 111 111 000 010; R7 = R7 + R2
0000 111 111110101;JUMP TO  R3 = R3-1

0000 100 000000011; JUDGE R3. IF R3 IS NEGATIVE,JUMP TO HALT
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0000 011 000000001 ; JUMP TO  HALT
0001 111 111 000 000; R7 = R7 + R0
```

测试数据：

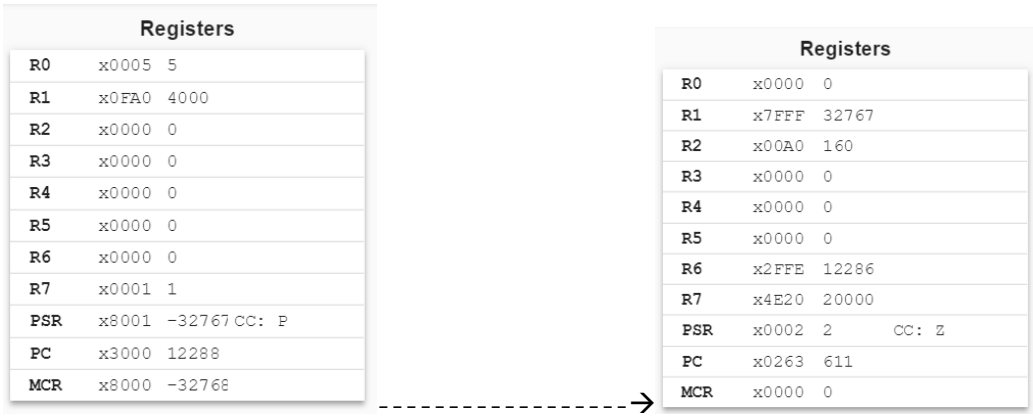
指令条数是通过模拟器的 step in 至结束的执行次数得出的。

（1）1*1



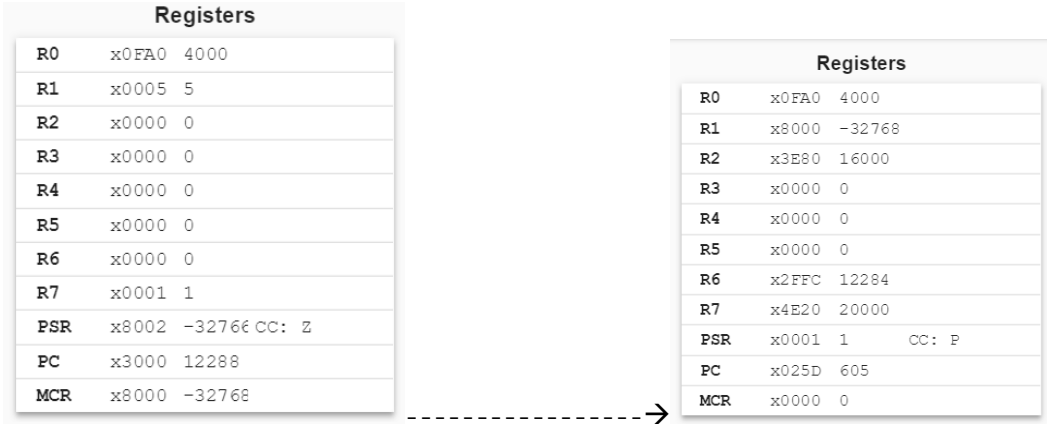
执行指令条数约为 66。

（2）5 * 4000



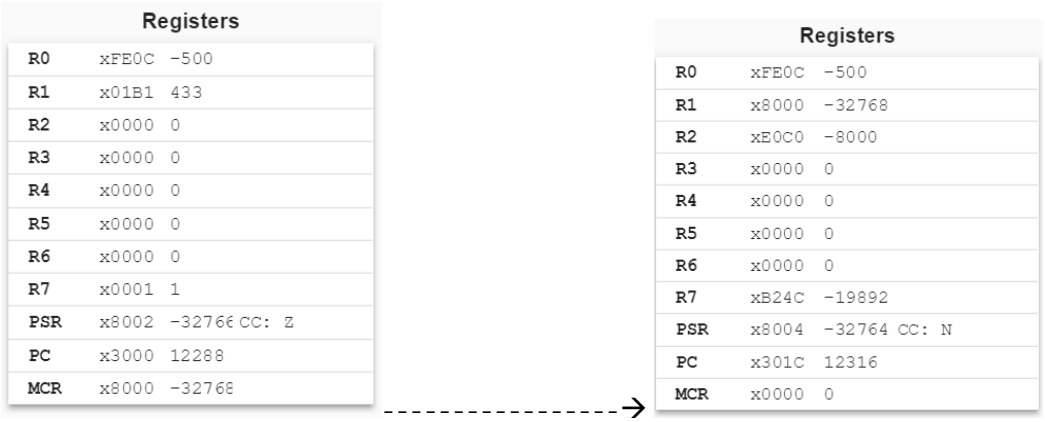
执行指令条数约为 237。

（3）计算 4000 * 5



执行指令条数约为 74。

（4）-500 * 433 （刻意溢出）

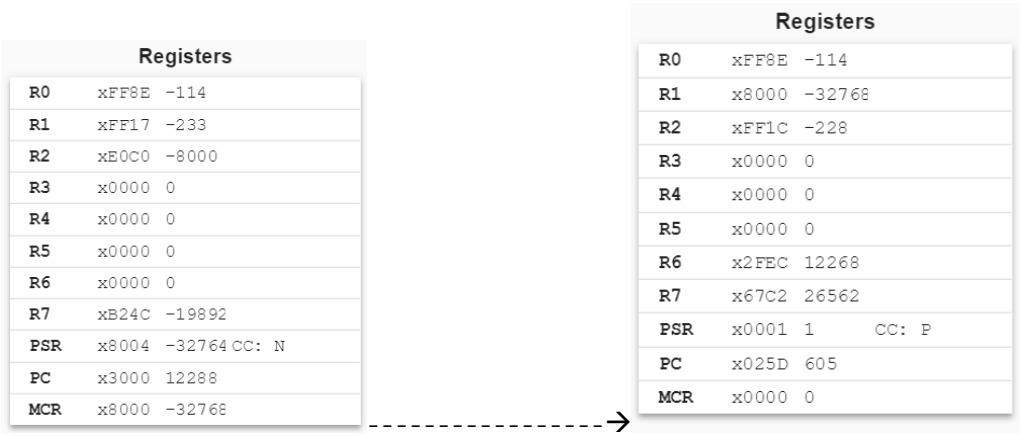


Registers		
R0	xFE0C	-500
R1	x01B1	433
R2	x0000	0
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	x0001	1
PSR	x8002	-32766 CC: Z
PC	x3000	12288
MCR	x8000	-32768

Registers		
R0	xFE0C	-500
R1	x8000	-32768
R2	xE0C0	-8000
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	xB24C	-19892
PSR	x8004	-32764 CC: N
PC	x301C	12316
MCR	x0000	0

执行指令条数约为 152。

（5）-114 * -233



Registers		
R0	xFF8E	-114
R1	xFF17	-233
R2	xE0C0	-8000
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x0000	0
R7	xB24C	-19892
PSR	x8004	-32764 CC: N
PC	x3000	12288
MCR	x8000	-32768

Registers		
R0	xFF8E	-114
R1	x8000	-32768
R2	xFF1C	-228
R3	x0000	0
R4	x0000	0
R5	x0000	0
R6	x2FEC	12268
R7	x67C2	26562
PSR	x0001	1 CC: P
PC	x025D	605
MCR	x0000	0

执行指令条数约为 144。

除去赋初值和 trap 指令，平均执行指令约为：134.6-3≈132

2.改良过程

起初，我也是采用列竖式乘法计算，但是由于第一次尝试是 R0 一直在移动，因此每次执行条数都在 **500** 次左右。代码如图（1）。

然后，我将方案改成对 R1 第一位进行判断，只有为 1 时才对 R0 移位。平均执行条数在 **187** 条左右。代码如图二。

由于方案二执行条数仍然过大，并且我发现在计算-114*-233 时尤其大，达到了 404 次。于是我在方案二的基础上继续改进，改成先判断 R1 是否为负数，如果 R1 为负数，则对 R1 和 R0 都取相反数，即 R1 将变成正数，这样将减少 R1 中 1 的个数，大大减少执行条数，于是有了 **1.设计思路**中所述的最终方案，去头去尾，平均执行指令条数大概是 **132** 条。

```
0001 011 011 1 01111 ; R3 = 15
0001 010 000 1 00000 ; R2=R0
0001 100 011 1 00000 ; R4=R3
0000 010 000000011 ;
0001 010 010 000 010 ; R2 =R2 + R2, MOVE ONE BIT LEFT
0001 100 100 1 11111 ;R4 =  R4-1
0000 001 111111101 ; IF R4 IS POSITIVE, JUMP TO R2= R2+R2;

0001 010 010 1 00000 ; R2 = R2
0000 100 000000100 ;
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0001 011 011 1 11111 ; R3 = R3-1
0000 011 111110101 ; IF R3 IS POSITIVE OR ZERO, JUMP TO R2= R0
0000 111 000000100 ; JUMP TO HALT;

0001 111 111 000 001; R7 = R7 + R1
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0001 011 011 1 11111 ; R3 = R3-1
0000 011 111110000 ; IF R3 IS POSITIVE OR ZERO, JUMP TO R2= R0
```

图（1）

```
0001 011 011 1 01111 ; R3 = 15
0001 001 001 1 00000 ; R1 = R1
0000 100 000000100 ;

0001 011 011 1 11111 ; R3 = R3-1
0000 110 000001001 ; IF R3 IS NEGATIVE OR ZERO, JUMP TO JUDGE R3
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0000 011 111111100 ; JUMP TO  R3 = R3-1

0001 010 000 1 00000 ; R2=R0
0001 100 011 1 00000 ; R4 = R3
0001 010 010 000 010 ; R2 = R2 + R2, MOVE ONE BIT LEFT
0001 100 100 1 11111 ;R4 =  R4-1
0000 001 111111101 ; IF R4 IS POSITIVE, JUMP TO R2 = R2+R2;
0001 111 111 000 010; R7 = R7 + R2
0000 111 111110101;JUMP TO  R3 = R3-1

0000 100 000000011; JUDGE R3. IF R3 IS NEGATIVE,JUMP TO HALT
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0000 011 000000001 ; JUMP TO  HALT
0001 111 111 000 000; R7 = R7 + R0
```

图（2）

```
0001 011 011 1 01111 ; R3 = 15
0001 001 001 1 00000 ; R1 = R1
0000 011 000000100 ; 判断R1是否为负，如果为负，则执行下面的R0取反加一，R1取反加一
1001 001 001 111111;
1001 000 000 111111;
0001 001 001 1 00001;
0001 000 000 1 00001;

0001 011 011 1 11111 ; R3 = R3-1
0000 110 000001001 ; IF R3 IS NEGATIVE OR ZERO, JUMP TO JUDGE R3
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0000 011 111111100 ; JUMP TO  R3 = R3-1

0001 010 000 1 00000 ; R2=R0
0001 100 011 1 00000 ; R4 = R3
0001 010 010 000 010 ; R2 = R2 + R2, MOVE ONE BIT LEFT
0001 100 100 1 11111 ;R4 =  R4-1
0000 001 111111101 ; IF R4 IS POSITIVE, JUMP TO R2 = R2+R2;
0001 111 111 000 010; R7 = R7 + R2
0000 111 111110101;JUMP TO  R3 = R3-1

0000 100 000000011; JUDGE R3. IF R3 IS NEGATIVE,JUMP TO HALT
0001 001 001 000 001 ; R1 =R1 + R1, MOVE ONE BIT LEFT
0000 011 000000001 ; JUMP TO  HALT
0001 111 111 000 000; R7 = R7 + R0
```

图（3）最终方案