

# lab1:实现LC3乘法

郑子涵 PB20000248

## 实验要求

- 两个运算数分别储存在R1和R0,其结果储存在R7，其他寄存器的结束状态不做要求。
- 初始状态：除R1和R0外的所有寄存器都为0。
- I部分：写出实现乘法的同时，尽可能少的机械码指令行数。
- p部分：写出实现乘法的同时，执行指令条数的次数最少。
- 要求实现与C语言的short型变量乘法结果相同。

## 设计实现

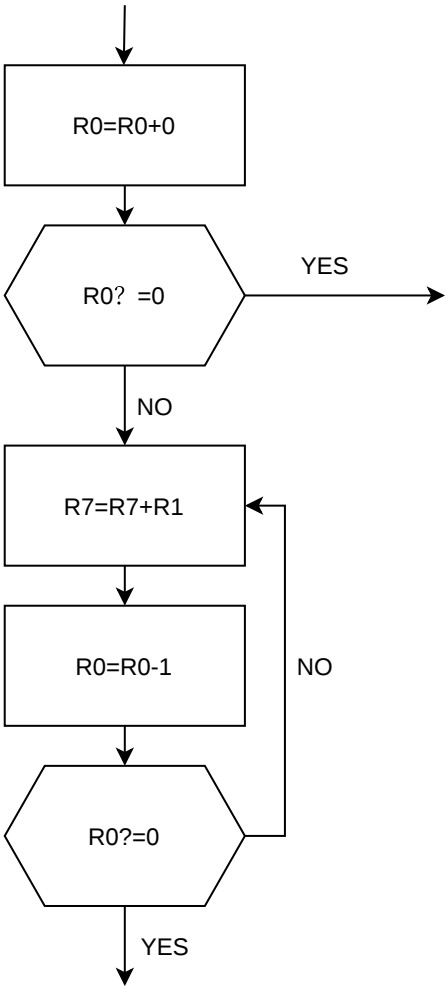
### I部分

#### 设计思路

##### 初始

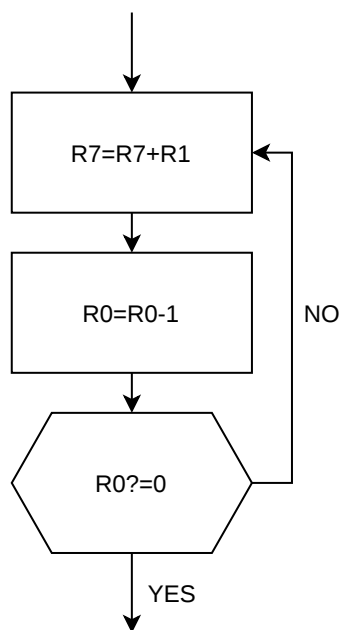
因为LC3机械可以实现加法，所以实现乘法的最直接的想法就是把乘法转化为多次加法来实现

根据这个思想可以画出流程图如下：



## 最终

根据观察，不需要最开始判断R0是否为0，即上述流程图中两个判断 `R0? =0` 是一样的，所以最终简化流程图如下



## 具体机械码实现

```
0011 0000 0000 0000;start at x3000

;my programe
0101 111 111 1 00000;清零R7

0001 111 111 0 00 001;R7=R7+R1
0001 000 000 1 11111;R0--
0000 101 111111101;如果R0不等于0，跳到R7=R7+R1
1111 0000 00100101;halt
```

## 总结

- 最开始是考虑把正负数分开来考虑，但发现按照补码的定义(负数的补码是数值位取反后加一)，最后都会因为进位而不会被保存下来，所以**有符号数和无符号数的乘法是一样的**，但初始版本还是考虑了有一位数是0的情况，在最终版本中发现0的情况也会最终因为最高位进位溢出而没影响。
- 最初的核心代码为5行。
- 最终得到的机械码如上，用了**3**行核心代码。

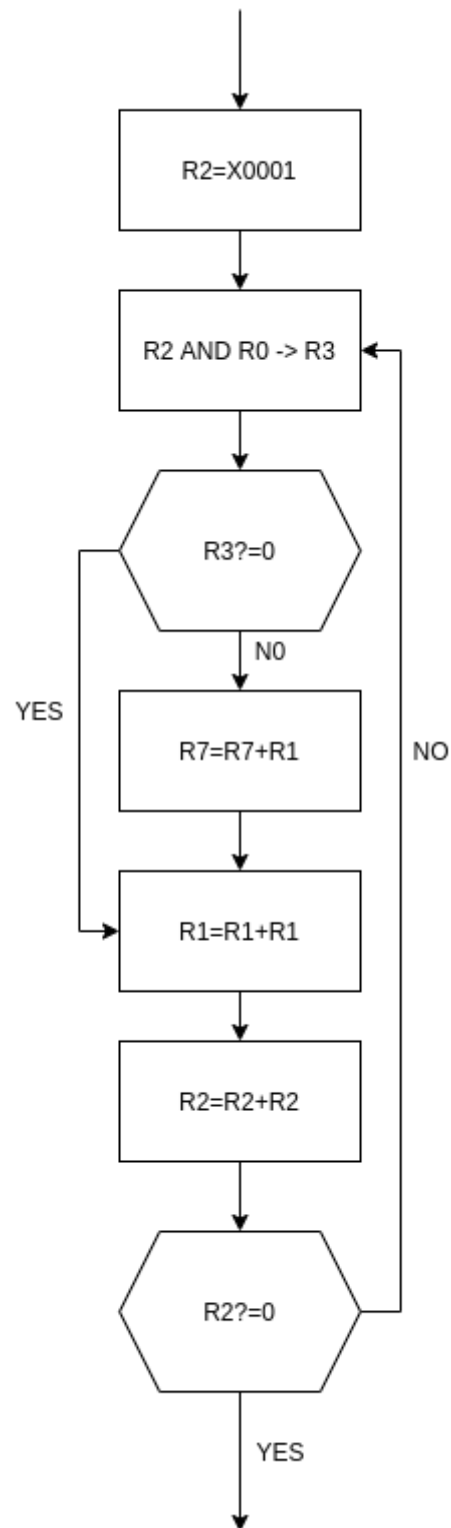
## P部分

### 设计思路

## 初始

乘法除了可以改为加法来实现外，还可以通过结合移位[左移可以通过自己加自己来实现]和加法来实现，就像平时10进制列的式子一样。我们可以从低位到高位依次取R0的值，如果是1的话，就加上对应的移位后的R1，如果是0则不加（要求每次R0取下一位前都要对R1进行移位到对应的权值），一直到取到最高位为止。

流程图如下



## 最终

可以发现最后的一个BR语句可以不用，循环可以分开来写，这样可以省去BR语句，但命令的条数会跟多。

## 具体机械码实现

[illegible]

```

0001 111 111 0 00 001;R7=R7+R1
0001 001 001 0 00 001;R1=R1+R1(左移一位)
0001 010 010 0 00 010;R2=R2+R2(左移一位)
0101 011 000 0 00 010;R2 AND R0-> R3
0000 010 000000001;如果R3=0则跳到 R1=R1+R1
0001 111 111 0 00 001;R7=R7+R1
0001 001 001 0 00 001;R1=R1+R1(左移一位)
0001 010 010 0 00 010;R2=R2+R2(左移一位)
0101 011 000 0 00 010;R2 AND R0-> R3
0000 010 000000001;如果R3=0则跳到 R1=R1+R1
0001 111 111 0 00 001;R7=R7+R1
0001 001 001 0 00 001;R1=R1+R1(左移一位)
0001 010 010 0 00 010;R2=R2+R2(左移一位)
0101 011 000 0 00 010;R2 AND R0-> R3
0000 010 000000001;如果R3=0则跳到 R1=R1+R1
0001 111 111 0 00 001;R7=R7+R1
0001 001 001 0 00 001;R1=R1+R1(左移一位)
0001 010 010 0 00 010;R2=R2+R2(左移一位)
0101 011 000 0 00 010;R2 AND R0-> R3
0000 010 000000001;如果R3=0则跳到 R1=R1+R1
0001 111 111 0 00 001;R7=R7+R1
0001 001 001 0 00 001;R1=R1+R1(左移一位)
0001 010 010 0 00 010;R2=R2+R2(左移一位)
0101 011 000 0 00 010;R2 AND R0-> R3
0000 010 000000001;如果R3=0则跳到 R1=R1+R1
0001 111 111 0 00 001;R7=R7+R1
0001 001 001 0 00 001;R1=R1+R1(左移一位)
0001 010 010 0 00 010;R2=R2+R2(左移一位)
0101 011 000 0 00 010;R2 AND R0-> R3
0000 010 000000001;如果R3=0则跳到 halt
0001 111 111 0 00 001;R7=R7+R1
1111 0000 00100101;halt

```

## 总结

- 因为移位所需要操作数与数值的大小无关，所以选择移位操作使得操作的次数最小，I部分的时间复杂度为 $O(n)$  [ $n$ 为 $R0$ 的大小]。
- 在核心语句中（从 $R2=X0001$ 开始），最长的执行条数（即从头执行到halt前一条）为79条，最短为63条。所以平均执行的命令条数约为**71**条。
- 优化思路：可以把一些情况分开来讨论，以减少特殊情况命令执行数，如：把其中有个数为0的情况分开来描述。

## 实验结果

- L部分最终用了**3**行核心代码。
- P部分最终平均执行了**71**条核心语句。

## 样例测试

- $4 * 450$ (L部分)

Registers			Memory		
R0	x0000	0	1	x3000	x5FE0 24544 0101111111100000
R1	x01C2	450	1	x3001	x1FC1 8129 0001111111000001
R2	x0000	0	1	x3002	x103F 4159 0001000000111111
R3	x0000	0	1	x3003	x0BFD 3069 000010111111101
R4	x0000	0	1	x3004	xF025 61477 1111000000100101
R5	x0000	0	1	x3005	x0000 0
R6	x0000	0	1	x3006	x0000 0
R7	x0708	1800	1	x3007	x0000 0
PSR	x8002	32770 CC: Z	1	x3008	x0000 0
PC	x3004	12292	1	x3009	x0000 0
MCR	x0000	0	1	x300A	x0000 0
Console (click to focus)			1	x300B	x0000 0
			1	x300C	x0000 0
			1	x300D	x0000 0
			1	x300E	x0000 0
			1	x300F	x0000 0
			1	x3010	x0000 0
			1	x3011	x0000 0
			1	x3012	x0000 0
			1	x3013	x0000 0
			1	x3014	x0000 0
			1	x3015	x0000 0
			1	x3016	x0000 0

结果正确，且按照算法R0清为0停止，R1不会改变，过程也正确

- 6 \* -50 (P部分)

R0	x0006	6	1	x3052	xF025 61477 1111000000100101
R1	x0000	0	1	x3053	x0000 0
R2	x8000	32768	1	x3054	x0000 0
R3	x0000	0	1	x3055	x0000 0
R4	x0000	0	1	x3056	x0000 0
R5	x0000	0	1	x3057	x0000 0
R6	x0000	0	1	x3058	x0000 0
R7	xFED4	65236	1	x3059	x0000 0
PSR	x8002	32770 CC: Z	1	x305A	x0000 0
PC	x3052	12370	1	x305B	x0000 0
MCR	x0000	0	1	x305C	x0000 0
Console (click to focus)			1	x305D	x0000 0
			1	x305E	x0000 0
			1	x305F	x0000 0
			1	x3060	x0000 0
			1	x3061	x0000 0
			1	x3062	x0000 0
			1	x3063	x0000 0
			1	x3064	x0000 0
			1	x3065	x0000 0
			1	x3066	x0000 0
			1	x3067	x0000 0
			1	x3068	x0000 0

结果正确，xFED4对应的数值为-300，R0不变，R1会因为移位而变成0，符合过程