Lab1 LC3中的乘法

————By 马子睿

一、L程序的设计

> L程序以**代码行数最短**为宗旨，因此可以采取以下算法：
>
> - 将某一个寄存器（这里选择的是R1）视作计数器，令其不断递减至0;
> - 每次递减时令R7中的值加上R0中的值，最终便可以得到乘法的结果。
>
> LC3代码仅需3行，代码如下：

```
0001 111 111 0 00000 ; R7 += R0
0001 001 001 1 11111 ; R1 -= 1
0000 101 111111101 ; if R1 == 0 break
```

代码分析

- **可行性分析**：

> 当R1中值为正数时，算法是易于理解的；当R1中值为负数或0时，我们可以先将其看做无符号整数，运算完毕后，根据代数结构中数论部分同余类的性质，此时的数字恰好**与所求数模2^{16}同余**，故截断后恰好为所求结果。

- **效率分析**：

> 当R1中值为正数，代码运行时间与R1中值得大小成正比。当R1中值为0或负数时，该代码相当于**把负数看作极大的正数来运算**，因此代码运行时间与R1中值大小也成正比。可以看出，当R1中值为极大正数或极小负数时，该代码所需执行的命令数都很多，执行效率并不高。

- **代码开发过程**：

> 最初的代码由于要对R1中值的正负进行判断,故命令数多达10行。后来经过对代数结构有关知识的温习，我省去了对正负性的判断，将代码缩减至3行。

- **正确性验证**：

> L型程序基于LC3的加法指令设计，故在LC3加法指令正确执行的基础上，只需要对R1中值的正负性进行验证即可。

1. R1中值为正数
   （运行前）

Registers

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

（运行后）

Registers

R0	x0005	5	
R1	x0000	0	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x4E20	20000	
PSR	x8002	-32766	CC: Z
PC	x3003	12291	
MCR	x0000	0	

2. R1中值为零

（运行前）

Registers

R0	x0005	5	
R1	x0000	0	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x0000	0	
PSR	x8002	-32766	CC: Z
PC	x3000	12288	
MCR	x0000	0	

（运行后）

Registers

R0	x0005	5	
R1	x0000	0	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x0000	0	
PSR	x8002	-32766	CC: Z
PC	x3003	12291	
MCR	x0000	0	

3. R1中值为负数

（运行前）

Registers

R0	xFF8E	-114	
R1	xFF17	-233	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x0000	0	
PSR	x8002	-32766	CC: Z
PC	x3000	12288	
MCR	x0000	0	

（运行后）

Registers

R0	xFF8E	-114	
R1	x0000	0	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x67C2	26562	
PSR	x8002	-32766	CC: Z
PC	x3003	12291	
MCR	x0000	0	

二、P程序的设计

> P程序以**代码运行时间最短**为宗旨，因此可以采取以下**快速幂**算法：
>
> - 构造一个试探数num，令num = 16'b1，令i = 0。再设R0中值为a，R1中值为b；
> - 令num与b作逻辑与运算，相当于检查b的二进制表示中第i位是否为1。若运算结果不为0，则证明最终运算结果中有$a * 2^i$这一项，只需加入当前和即可；
> - 将num左移1位（相当于自加），i += 1。若num变为0，证明已经溢出，算法终止，否则转上一步。
> - **注：对于$a * 2^i$，我们可以采取每一次循环都令R0中值自加的方式进行构造。**
>
> LC3代码如下：

```
0001 010 010 1 00001 ; R2 = 1
;1
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;2
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;3
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;4
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
```

```
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;5
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;6
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;7
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;8
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;9
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;10
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;11
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;12
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;13
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;14
0101 011 010 0 00001 ; R3 = R2 & R1
```

```
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;15
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
;16
0101 011 010 0 00001 ; R3 = R2 & R1
0000 010 000000001; if R3 == 0 ignore add
0001 111 111 0 00000 ; R7 = R7 + R0
0001 000 000 0 00000 ; R0 = R0 * 2
0001 010 010 0 00010 ; R2 = R2 << 1
```

代码分析

- **可行性分析**:

> 该算法的本质是将R1中的值拆分为二进制幂形式进行表示。设R0中值为a，R1中值二进制表示为b，则乘法结果自然可以表示成如下形式：
>
> $$a * 2^{15} * b[15] + \ldots + a * 2^{0} * b[0]$$
>
> 我们在这里使用试探数去取b的每一位，而$a * 2^{i} (0 <= i <= 15)$可以采用令a不断左移（即自加）来实现。
>
> 对于b为负数的情况，请参见L程序可行性分析。

- **效率分析**:

> 无论R1中数字为何值，其必然有且仅有16个bit位，因此R2中试探数左移为零（即溢出）标志着运算结束，故只需进行16次左移即可，我们可以将其视作一个循环。每次循环中，都会执行4—5条指令（这和R2&R1的运算结果有关），故整个程序执行的指令条数区间为[16 * 4 + 1, 16 * 5 + 1],，即[65, 81]，且不随R1中值大小而产生较大变动。

- **代码开发过程**

> 最初的代码依旧采取了R1递减的方法，只是先去比较R0和R1的无符号补码，令较小的值递减。这种做法执行的指令条数依然会随着数值大小的变化而产生的极大的变化，具体来讲是与较小值成正比。之后改用快速幂算法，使得执行指令条数大大缩减，且不随数值大小而产生剧烈变化。最终利用将循环展开，省去了对循环跳出条件的判断，使得执行指令数再缩短16条。

- **正确性验证**

> 有了“可行性分析”中的理论保证，对于P程序我们只需要验证R1中值为正数或负数时的正确性即可：
>
> 1. R1中值为正数：
>
>   （运行前）

Registers

R0	x007E	126	
R1	x007D	125	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x0000	0	
PSR	x8002	-32766	CC: Z
PC	x3000	12288	
MCR	x0000	0	

(运行后)

Registers

R0	x0000	0	
R1	x007D	125	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x3D86	15750	
PSR	x8002	-32766	CC: Z
PC	x3007	12295	
MCR	x0000	0	

2. R1中值为负数:

(运行前)

Registers

R0	xFF82	-126	
R1	xFF81	-127	
R2	x0000	0	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x0000	0	
PSR	x8002	-32766	CC: Z
PC	x3000	12288	
MCR	x0000	0	

(运行后)

Registers

R0	x0000	0	
R1	xFF81	-127	
R2	x0000	0	
R3	x8000	-32768	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x3E82	16002	
PSR	x8002	-32766	CC: Z
PC	x3007	12295	
MCR	x0000	0	

三、实验总结

本次实验进行了LC3乘法指令的实现，在代码优化过程当中，我深刻体会到算法优劣会使得最终执行的效率产生极大的影响，因为一个不优的算法会导致计算机进行无谓的重复运算。因此，在后续代码开发中，我们更要通过对计算机执行指令的方法的分析，构造出最适合计算机执行的程序，以此获得更优的程序运行效率。