第一次实验：在LC3中实现乘法

2021.11 赖永凡　PB20061210

实验内容

本次实验是实现乘法对应程序的机器码。两个运算数分别放置于R0和R1，结果存储于R7。初始状态：R0和R1存放待计算数，其余寄存器全部为0。

实验代码分为两个版本，**L版本尽量编写更少的代码行数，P版本尽量让程序执行更少的指令。**

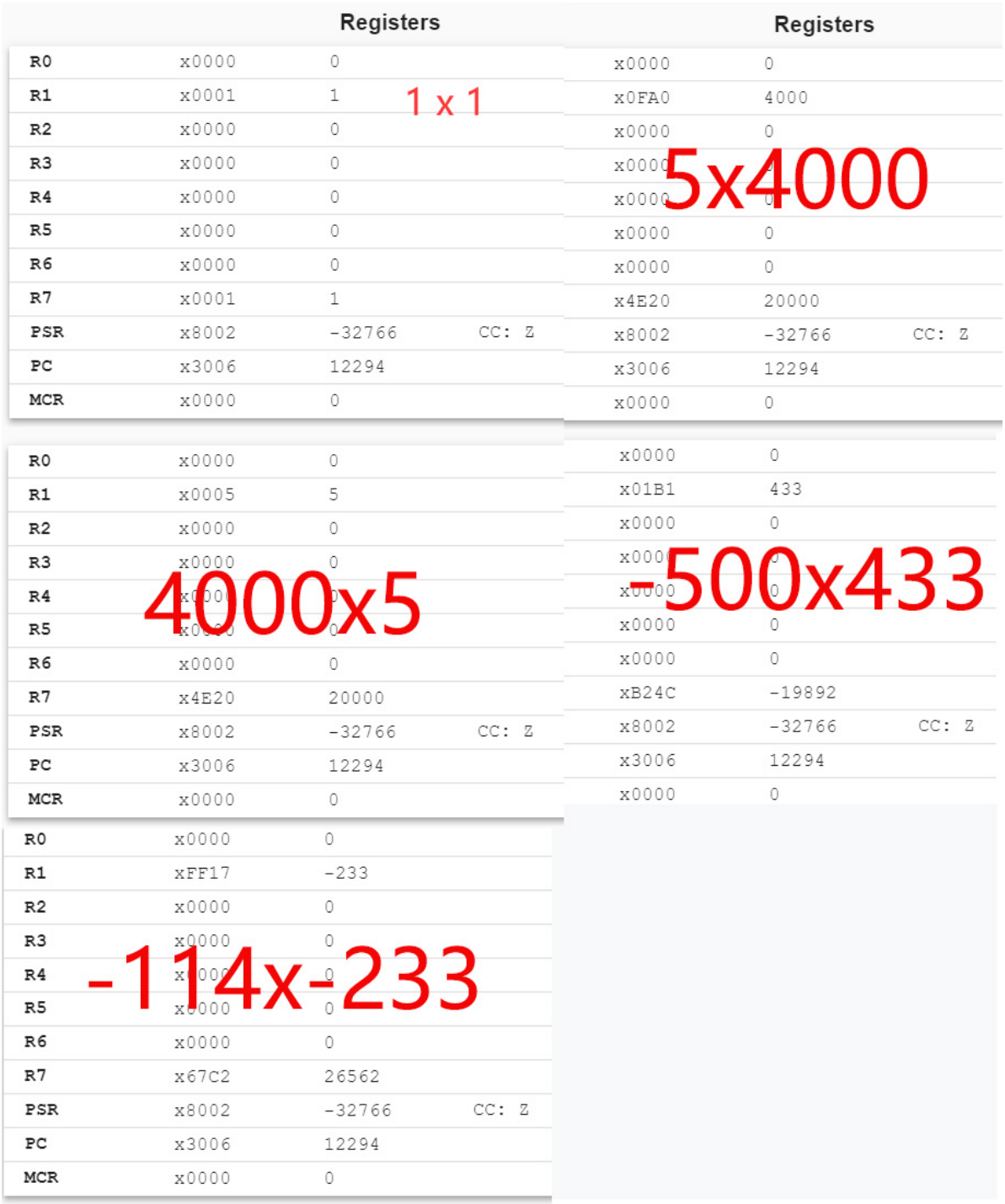
P.S. 为了便于阅读，本实验报告中均使用汇编语言来描述算法

L版本

测试样例

计算 1 * 1 = **1**
计算 5 * 4000 = **20000**
计算 4000 * 5 = **20000**
计算 -500 * 433 （刻意溢出）= **-19892**
计算 -114 * -233 = **26562**

*已在.halt前加入断点



设计思路

本着越简单的算法写出来的行数越少的思想，L版本代码核心思想均是将n*m转换成“n”次m自身相加。
于是不可避免地遇到了符号的问题，在最初，我先判断R0中存的数是否为负数，若为负数则将其取相反数，并将R2（标志）置1，经过这个预处理后，R0中存储的总是一个正数。
之后是累加的循环，让R0作为哨兵，实现|n|*m的操作，循环结束后如果R2（标志）为1，则再将所的结果取相反数存入R7中。

```
;Version L(v.1)
      .ORIG x3000
      LD    R0,LEFT
      LD    R1,RIGHT
      AND   R7,R7,#0
      AND   R2,R2,#0  ;Initialization

      ADD R0,R0,#0    ;1
```

```
       BRzp LOOP       ;2
       NOT R0,R0       ;3
       ADD R0,R0,#1    ;4
       ADD R2,R2,#-1   ;5:Whether R0 is negative
                       ;  If it is, flip it and set R2 as a flag
LOOP   BRz REVISE      ;6
       ADD R7,R7,R1    ;7
       ADD R0,R0,#-1   ;8
       BRnzp LOOP      ;9:using R0 as the sentinel

REVISE ADD R2,R2,#0    ;10
       BRzp FIN        ;11
       NOT R7,R7       ;12
       ADD R7,R7,#1    ;13:If the flag was set, flip the result back

FIN    HALT
```

可以看出，此版本的代码花了大量的行数用于处理符号的问题，便自然而然地引人深思：能否更为简洁而优雅地处理呢?

不过，改进后的代码可以说是简洁但不优雅——它直接回避了符号的问题

```
;Version L(v.2)
       .ORIG x3000
       LD    R0,LEFT
       LD    R1,RIGHT
       AND   R7,R7,#0  ;Initialization

       ADD R0,R0,#0    ;1
LOOP   BRz FIN         ;2
       ADD R7,R7,R1    ;3
       ADD R0,R0,#-1   ;4
       BRnzp LOOP      ;5:using R0 as the sentinel


FIN    HALT
```

在某种机缘巧合之下，我发现可以利用溢出机制来回避符号的问题，换言之，即无论R0正负与否，均将其作为一个哨兵:

这种情况下，若R0大于零，则与通常思维无异，皆大欢喜；倘若R0小于零，虽然看似算法出现了逻辑错误，但最终溢出后的数值惊人地准确。

以“-1*1”为例，“-1”在寄存器中以65535形式储存，于是程序实现了65535个1相加，R7中储存的值便是65535，而这个值正好对应着-1。

然而，虽说这个版本的程序简洁得有点难以置信，却是在极大程度上牺牲了效率实现的。直观上而言，每次涉及R0中为负数的计算时，程序的响应时间达到了几秒钟，相比上一个版本的瞬间完成，效率之低可见一斑。

不过，既然L版本不看效率，那么这应该也算是一个满意的答案了。

> 如无必要，勿增实体——奥卡姆剃刀原则

P版本

测试样例

计算 1 * 1 = **1**
计算 5 * 4000 = **20000**
计算 4000 * 5 = **20000**
计算 -500 * 433 （刻意溢出）= **-19892**
计算 -114 * -233 = **26562**

*已在.halt前加入断点

Registers				Registers		
R0	x0001	1		x0005	5	
R1	x0001	1		x0FA0	4000	
R2	x8000	-32768		x8000	-32768	
R3	x0000	0		x0000	0	
R4	x8000	-32768		x0000	0	
R5	x0000	0		x0000	0	
R6	x0000	0		x0000	0	
R7	x0001	1		x4E20	20000	
PSR	x8002	-32766	CC: Z	x8002	-32766	CC: Z
PC	x3053	12371		x3053	12371	
MCR	x0000	0		x0000	0	

1x1

5x4000

R0	x0FA0	4000		xFE0C	-500	
R1	x0005	5		x01B1	433	
R2	x8000	-32768		x8000	-32768	
R3	x0000	0		x8000	-32768	
R4	x8000	-32768		x8000	-32768	
R5	x0000	0		x0000	0	
R6	x0000	0		x0000	0	
R7	x4E20	20000		xB24C	-19892	
PSR	x8002	-32766	CC: Z	x8004	-32764	CC: N
PC	x3053	12371		x3053	12371	
MCR	x0000	0		x0000	0	

4000x5

-500x433

R0	xFF8E	-114	
R1	xFF17	-233	
R2	x8000	-32768	
R3	x8000	-32768	
R4	x8000	-32768	
R5	x0000	0	
R6	x0000	0	
R7	x67C2	26562	
PSR	x8001	-32767	CC: P
PC	x3053	12371	
MCR	x0000	0	

-114x-233

设计思路

P版本沿用了L版本的思路，即依然使用累加的算法。在此基础上要实现程序执行的高效，最简单的想法即使用绝对值更小的那个乘数作为哨兵。

```
Version P(v.1)
      .ORIG x3000
```

```
      LD R0,LEFT
      LD R1,RIGHT
      AND R2,R2,#0
      AND R3,R3,#0
      AND R7,R7,#0   ;initialization

      ADD R0,R0,#0   ;line 1
      BRz FIN
      BRp CHR1
      NOT R0,R0
      ADD R0,R0,#1   ;if negative, flip R0
      ADD R2,R2,#1   ;set R2 as a flag

CHR1  ADD R1,R1,#0
      BRz FIN
      BRp CMP
      NOT R1,R1
      ADD R1,R1,#1   ;line 11
      ADD R2,R2,#1

CMP   NOT R3,R1      ;want to use the smaller one as the sentinel
      ADD R3,R3,#1
      ADD R3,R3,R0   ;R0-R1
      BRzp LP1

LP0   ADD R7,R7,R1   ;set R0 as a sentinel
      ADD R0,R0,#-1
      BRz RVSE
      BRnzp LP0

LP1   ADD R7,R7,R0   ;line 21:set R1 as a sentinel
      ADD R1,R1,#-1
      BRz RVSE
      BRnzp LP1

RVSE  ADD R2,R2,#-1
      BRnp FIN
      NOT R7,R7
      ADD R7,R7,#1   ;line 28

FIN HALT
```

初代P程序执行时将先查看R0、R1两个寄存器的值，并将其均转为正数，并使用较小的那个作为哨兵。其中R2的值表示负数的个数，在最后修正的过程中，只有当R2为1时需要将所得结果取相反数。

此外，如果发现某一乘数为0，则直接结束程序。

由此可计算执行的指令数：

计算 1 * 1 ： 15条
计算 5 * 4000 ： 29条
计算 4000 * 5 ： 29条
计算 -500 * 433 ： 1746条
计算 -114 * -233 ： 475条

平均执行指令数：458.8条
可以看出，影响执行指令数量的主要是对溢出表达式的计算造成了无意义的消耗。

```
Version P(v.2)
     .ORIG x3000
     LD R0,LEFT
     LD R1,RIGHT
     AND R2,R2,#0   ;sign mark
     AND R3,R3,#0   ;temp
     AND R4,R4,#0   ;-128 or -256
     AND R7,R7,#0

     ADD R0,R0,#0
     BRz FIN
     BRp CHR1
     NOT R0,R0
     ADD R0,R0,#1   ;if negative, flip R0
     ADD R2,R2,#1   ;set R2 as a flag

CHR1 ADD R1,R1,#0
     BRz FIN
     BRp CMP
     NOT R1,R1
     ADD R1,R1,#1
     ADD R2,R2,#1


CMP  NOT R3,R1      ;want to use the smaller one as the sentinel
     ADD R3,R3,#1
     ADD R3,R3,R0   ;R0-R1
     BRnz LP0I
     ADD R3,R0,#0
     ADD R0,R1,#0
     ADD R1,R3,#0   ;swap R0 and R1, so that R0 always stores the smaller
number.

LP0I LD R4,M128
     ADD R3,R0,R4
     BRn LP0        ;if R0<128
     ADD R4,R4,R4   ;to form 256
     ADD R3,R1,R4
     BRn LP0        ;if R1<256, too
     LD  R7,OVFL    ;set the overflow mark
     BRnzp FIN

LP0  ADD R7,R7,R1   ;set R0 as a sentinel
     ADD R0,R0,#-1
     BRp LP0

RVSE ADD R2,R2,#-1
     BRnp FIN
     NOT R7,R7
     ADD R7,R7,#1

FIN HALT

M128  .FILL #-128
OVFL  .FILL #32768

      .END
```

此程序相较于上一代主要有两个方面的变化：

1、针对溢出的处理：如果更小的那个乘数的绝对值大于128，且更大的那个乘数绝对值大于256，则判断表达式溢出，自动给R7存储32768（-32768）。这里也是考虑到-128*256=-32768并不算是溢出。

2、将主循环从四条指令压缩至3条指令，因为之前已经对乘数含有0的情况做了特殊处理，使得这样压缩得以实现。

由此可计算执行的指令数：

计算 1 * 1 ： 18条
计算 5 * 4000 ： 30条
计算 4000 * 5 ： 33条
计算 -500 * 433 ： 24条
计算 -114 * -233 ： 362条

平均执行指令数：93.4条

虽然在较为简单的计算上花费了更多的指令，但是当涉及溢出亦或是循环次数较多时，效率的提升还是很可观的。与此相比，额外付出的指令也无足轻重了。

但是，以上的版本却隐藏着巨大的隐患如果两个乘数都比较大的话，其需要执行的指令数将达到几百数量级，虽然深知这一点，但我本来还是打算就此停笔。

结果后来实在是被身边的同学卷怕了，就又写了第三代的P版本代码。

不过想要在原来的思想上再提升效率已经没有多少空间了，毕竟算法的局限性摆在那里。如果想要彻底压缩执行的指令数，还是得将原来一切推倒重做，改弦更张。

> 不破不立

```
;verp(v.3)

         .ORIG x3000
          LD R0,LEFT
          LD R1,RIGHT
          AND R2,R2,#0
          AND R3,R3,#0
          AND R7,R7,#0

          ADD R2,R2,#1 ;0000 0000 0000 0001
          AND R3,R0,R2
          BRz B1
          ADD R7,R7,R1

ONE       ADD R4,R1,R1 ;R1 begins shifts left, stores in R4
          ADD R2,R2,R2 ;0000 0000 0000 0010
          AND R3,R0,R2 ;distinguish whether bit[1] is 0
          BRz TWO
          ADD R7,R7,R4 ;if not, augment R7 with R4.

TWO       ADD R4,R4,R4
          ADD R2,R2,R2 ;0000 0000 0000 0100
          AND R3,R0,R2
          BRz THREE
          ADD R7,R7,R4

THREE     ADD R4,R4,R4
          ADD R2,R2,R2 ;0000 0000 0000 1000
```

```
         AND R3,R0,R2
         BRz FOUR
         ADD R7,R7,R4

FOUR     ADD R4,R4,R4
         ADD R2,R2,R2 ;0000 0000 0001 0000
         AND R3,R0,R2
         BRz FIVE
         ADD R7,R7,R4

FIVE     ADD R4,R4,R4
         ADD R2,R2,R2 ;0000 0000 0010 0000
         AND R3,R0,R2
         BRz SIX
         ADD R7,R7,R4

SIX      ADD R4,R4,R4
         ADD R2,R2,R2 ;0000 0000 0100 0000
         AND R3,R0,R2
         BRz SEVEN
         ADD R7,R7,R4

SEVEN    ADD R4,R4,R4
         ADD R2,R2,R2 ;0000 0000 1000 0000
         AND R3,R0,R2
         BRz EIGHT
         ADD R7,R7,R4

EIGHT    ADD R4,R4,R4
         ADD R2,R2,R2 ;0000 0001 0000 0000
         AND R3,R0,R2
         BRz NINE
         ADD R7,R7,R4

NINE     ADD R4,R4,R4
         ADD R2,R2,R2 ;0000 0010 0000 0000
         AND R3,R0,R2
         BRz TEN
         ADD R7,R7,R4

TEN      ADD R4,R4,R4
         ADD R2,R2,R2 ;0000 0100 0000 0000
         AND R3,R0,R2
         BRz ELE
         ADD R7,R7,R4

ELE      ADD R4,R4,R4
         ADD R2,R2,R2 ;0000 1000 0000 0000
         AND R3,R0,R2
         BRz TWE
         ADD R7,R7,R4

TWE      ADD R4,R4,R4
         ADD R2,R2,R2 ;0001 0000 0000 0000
         AND R3,R0,R2
         BRz THR
         ADD R7,R7,R4
```

```
THR      ADD R4,R4,R4
         ADD R2,R2,R2 ;0010 0000 0000 0000
         AND R3,R0,R2
         BRz FOUT
         ADD R7,R7,R4

FOUT     ADD R4,R4,R4
         ADD R2,R2,R2 ;0100 0000 0000 0000
         AND R3,R0,R2
         BRz FIFT
         ADD R7,R7,R4

FIFT     ADD R4,R4,R4
         ADD R2,R2,R2 ;1000 0000 0000 0000
         AND R3,R0,R2
         BRz FIN
         ADD R7,R7,R4

FIN      HALT

LEFT    .FILL   #tbd
RIGHT   .FILL   #tbd

        .END
```

这次采用了基于位运算的算法，换言之，就是人类计算乘法的方式。

大部分描述已经在注释中给出，就不再赘述。值得一提的是，这里为了节省指令数，我放弃了使用循环结构，转而使用朴实无华的顺序结构，所以代码具有高度的重复性，代码的编写也几乎由复制粘贴组成，但是这也恰恰说明了执行的指令数不会超过代码的总条数（79条）

最后以讨论代码的执行平均条数作为结尾：

不难发现，除了bit[0]处是4行代码，其余位均为5行，这里为了讨论方便，均记为使用了5行代码，到计算最后结果时仅需减一即可。
而对于R0的bit[n]而言，若为1，需执行全部的5条指令；若为0，则需执行4条指令。又简单假设R0中值含零的个数的期望为8，因此平均执行条数为：5 x 8 + 4 x 8 - 1 = 71 条。