

# 第一次实验

姓名: 傅申 学号: PB20000051

## 1 L 版本程序设计

### 1.1 理论

### 1.2 L 版本程序

#### 1.2.1 最初版本

#### 1.2.2 优化与最终版本

## 2 P 版本程序设计

### 2.1 理论

### 2.2 P 版本程序

#### 2.2.1 最初版本

#### 2.2.2 优化与最终版本

### 2.3 附录

# 1 L 版本程序设计

## 1.1 理论

记  $a$  和  $b$  的乘积为  $MUL(a, b)$ , 则当  $a \geq 0$  时, 有

$$MUL(a, b) = \begin{cases} 0 & a = 0 \\ b + MUL(a - 1, b) & a \neq 0 \end{cases} \quad (1)$$

当  $a < 0$  时, 考虑到 LC3 的寄存器是 16 位的, 所以  $a$  的补码为  $2^{16} + a$ , 则当  $a$  减去了  $2^{16} + a$  次 1 后,  $a$  在寄存器中就变为了 `x0000`, 也就是 0. 按照上式计算, 有

$$MUL(a, b) \equiv ab \equiv 2^{16} + ab \pmod{2^{16}}$$

不考虑溢出 ( $-2^{15} \leq ab \leq 2^{15} - 1$ ), 有:

- $a \geq 0$  时  $MUL(a, b) = ab$ ;
- $a < 0$  时:
  - 若  $b \leq 0 \Rightarrow ab \geq 0$ , 则  $MUL(a, b)$  在寄存器中就是  $ab$ ;
  - 若  $b > 0 \Rightarrow ab < 0$ , 则  $2^{16} + ab$  就是  $ab$  的补码,  $MUL(a, b)$  在寄存器中就是  $ab$ .

即公式 (1) 对于所有不溢出的情况是正确的.

若考虑溢出, 记最后的计算结果在寄存器中为  $c$ , 则只能保证  $c \equiv ab \pmod{2^{16}}$ , 参考下面的 C++ 程序运行结果, 可以认为是正确的.

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int16_t num_pos_overflow = 500 * 433;
6 |     int16_t num_neg_overflow = 500 * -433;
7 |     std::cout << num_pos_overflow << std::endl;
8 |     std::cout << num_neg_overflow << std::endl;
9 |     return 0;
10 | }
```

输出为, 其中  $500 \times 433 = 216500 \equiv 19892 \pmod{2^{16}}$

```
1 | 19892
2 | -19892
```

## 1.2 L 版本程序

### 1.2.1 最初版本

根据公式 (1) 可以得到下面的算法:

---

**Algorithm 1:** Multiply

---

```
1 initial R2 to R7  $\leftarrow$  0
2 loop:
3 if R0 = 0 then
4   | HALT
5 else
6   | R0  $\leftarrow$  R0 - 1
7   | R7  $\leftarrow$  R7 + R1
8   | goto loop
```

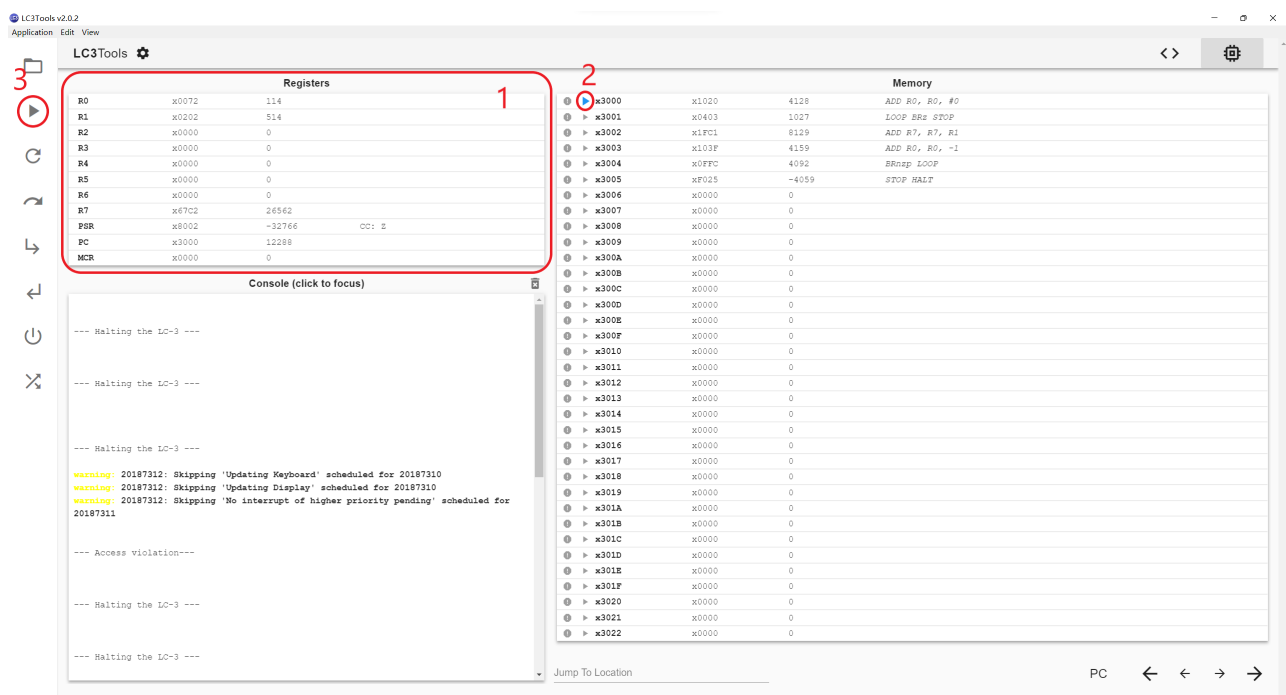
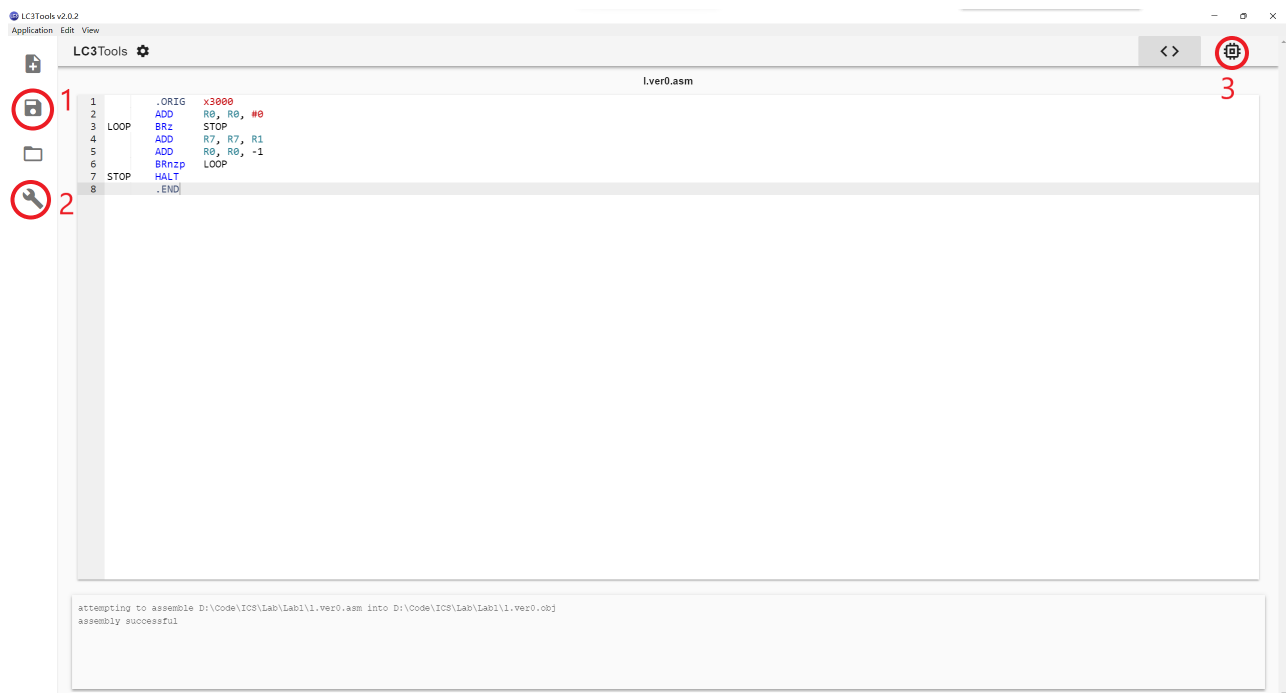
---

写成对应的汇编程序与机器码分别如下:

```
1 |          ADD      R0, R0, #0
2 | LOOP     BRz      STOP
3 |          ADD      R7, R7, R1
4 |          ADD      R0, R0, -1
5 |          BRnzp    LOOP
6 | STOP     HALT
```

```
1 | 0001000000100000
2 | 0000010000000011
3 | 0001111111100001
4 | 0001000000111111
5 | 0000111111111100
6 | 1111000000100101
```

程序总共 6 行, 使用 LC3Tools v2.0.2 进行测试, 在汇编程序的开头和结尾分别加上 `.ORIG x3000` 和 `.END`, 将 `.asm` 文件保存后, 点击左侧的 Assemble, 然后切换到模拟器页面, 在 Register 栏将寄存器设置为初始值后, 设置好 PC, 点击左侧的 Run 即可运行程序. 如下图



对测试样例进行测试，结果如下

R0	R1	R7	R0×R1	理论值
1	1	1	1	1
5	4000	20000	20000	20000
4000	5	20000	20000	20000
-500	433	-19892	-216500	-19892
-114	-233	26562	26562	26562

可以看到，程序运行没有问题。

1.2.2 优化与最终版本

考虑  $R1 = 0$  的情况, 这时  $R1$  需要减去  $2^{16}$  次 1 才能重新为 0, 而  $2^{16}b \equiv 0(\text{mod } 2^{16})$ , 则如果没有第二行的 `BR` 指令, 并将第五行的 `BRnzp` 改为 `BRnp`, 程序仍然是正确的, 那么可以将汇编程序和机器码修改如下:

```
1 | LOOP    ADD      R7, R7, R1
2 |         ADD      R0, R0, -1
3 |         BRnp    LOOP
4 | STOP    HALT
```

```
1 | 00011111111000001
2 | 0001000000111111
3 | 0000101111111100
4 | 1111000000100101
```

可以看到程序从 6 行缩短到了 4 行, 按照上面的测试方法, 测试结果如下

R0	R1	R7	R0×R1	理论值
0	114	0	0	0
1	1	1	1	1
5	4000	20000	20000	20000
4000	5	20000	20000	20000
-500	433	-19892	-216500	-19892
-114	-233	26562	26562	26562

可以看到, 程序运行没有问题.

## 2 P 版本程序设计

### 2.1 理论

根据前面的理论, 可以看出两数相乘其实就是补码相乘, 下面就只针对补码进行分析.

显然有公式:

$$MUL(a, b) = \begin{cases} 0 & a = 0 \\ MUL\left(\left\lfloor \frac{a}{2} \right\rfloor, b\right) + MUL\left(\left\lfloor \frac{a}{2} \right\rfloor, b\right) & a \text{ is even} \\ MUL\left(\left\lfloor \frac{a}{2} \right\rfloor, b\right) + MUL\left(\left\lfloor \frac{a}{2} \right\rfloor, b\right) + b & a \text{ is odd} \end{cases}$$

其中, 若  $a$  是偶数/奇数, 则 `a & x0001 = 0` / `1`, 且  $\left\lfloor \frac{a}{2} \right\rfloor = a >> 1$ .

上面给出的是递归公式, 若要改成递推公式, 则需要从  $a$  的最高位开始, 依次判断, 遇到 `1` 则乘 2 加  $b$ , 否则只乘 2, 直到遍历所有位. 算法如下:

Algorithm 2: Multiply	
1	initial R2 to R7 $\leftarrow$ 0
2	if R0 = 0    R1 = 0 then
3	HALT
4	else
5	foreach $i$ in R0[15 $\rightarrow$ 0] do
6	R7 $\leftarrow$ R7 + R7
7	if $i = 1$ then
8	R7 $\leftarrow$ R7 + R1

### 2.2 P 版本程序

#### 2.2.1 最初版本

写出上面算法的汇编程序如下

1		ADD	R0, R0, #0
2		BRz	STOP
3		ADD	R1, R1, #0
4		BRz	STOP
5		ADD	R2, R2, #15
6	LOOP	ADD	R7, R7, R7
7		ADD	R0, R0, #0
8		BRzp	ODD
9		ADD	R7, R7, R1
10	ODD	ADD	R0, R0, R0
11		ADD	R2, R2, #-1
12		BRzp	LOOP
13	STOP	HALT	

对应的机器码如下:

```

1 0001000000100000 // x1020
2 0000010000001010 // x040A
3 0001001001100000 // x1260
4 0000010000001000 // x0408
5 0001010010101111 // x14AF
6 0001111111000111 // x1FC7
7 0001000000100000 // x1020
8 0000011000000001 // x0601
9 0001000000000000 // x1000
10 0001010010111111 // x14BF
11 0000011111111001 // x07F9
12 1111000000100101 // xF025

```

对测试样例的运行结果如下:

R0	R1	R7	执行指令数	R0×R1	理论值
-1	1	-1	118	-1	-1
1	1	1	103	1	1
5	4000	20000	104	20000	20000
4000	5	20000	108	20000	20000
-500	433	-19892	111	-216500	-19892
-114	-233	26562	114	26562	26562

其中执行指令数的统计是通过 `R3` 计算出的, 每执行一段指令, `R3` 便加上执行的指令数, 汇编代码如下:

```

1      .ORIG    x3000
2
3      ADD     R3, R3, #2
4      ADD     R0, R0, #0
5      BRz     STOP
6
7      ADD     R3, R3, #2
8      ADD     R1, R1, #0
9      BRz     STOP
10
11     ADD     R3, R3, #1
12     ADD     R2, R2, #15
13
14     LOOP    ADD     R3, R3, #3
15           ADD     R7, R7, R7
16           ADD     R0, R0, #0
17           BRz     ODD      ; R1[15] = 0
18
19           ADD     R3, R3, #1
20           ADD     R7, R7, R1
21

```

```

22  ODD      ADD      R3, R3, #3
23          ADD      R0, R0, R0
24          ADD      R2, R2, #-1
25          BRzp     LOOP          ; Loop 16 times
26
27  STOP     ADD      R3, R3, #1
28          HALT
29          .END

```

不难发现, 当  $R1 \times R2 \neq 0$  时, 程序需要执行的指令数为  $102 \sim 118$  ( $5 + 16 \times (6 \sim 7) + 1$ ). 所以程序最多需要 118 条指令. 对于测试样例, 执行指令数的平均为 108.

### 2.2.2 优化与最终版本

从理论出发, 根据乘法列竖式的计算方法, 有如下公式:

$$\begin{aligned}
 MUL(R0, R1) = \sum_{i=0}^{15} R0[i] \times 2^i \times R1
 \end{aligned}$$

要想得到 `R0` 的各位, 只需要让  $2^i$  和 `R0` 求与即可, 有如下算法:

Algorithm 3: Multiply	
1	initial R2 to R7 $\leftarrow$ 0
2	R2 $\leftarrow$ x0001
3	loop: if R0 & R2 $\neq$ 0 then
4	R7 $\leftarrow$ R7 + R1
5	R1 $\leftarrow$ R1 + R1
6	R2 $\leftarrow$ R2 + R2
7	if R2 $\neq$ 0 then
8	goto loop
9	
10	HALT

对应的汇编代码为:

```

1          ADD      R2, R2, #1
2  Loop     AND      R3, R0, R2
3          BRz      BitZero
4          ADD      R7, R7, R1
5  BitZero  ADD      R1, R1, R1
6          ADD      R2, R2, R2
7          BRnp     Loop
8          HALT

```

机器码为:



```
1 0001010010100001 // x14A1
2 0101011000000010 // x5602
3 0000010000000001 // x0401
4 0001111111000001 // x1FC1
5 0001001001000001 // x1241
6 0001010010000010 // x1482
7 0000101111111010 // x0BFA
8 1111000000100101 // xF025
```

对测试样例的运行结果如下, 执行指令数同样通过分块求和得出:

R0	R1	R7	执行指令数	R0×R1	理论值
0	1	0	82	0	0
-1	1	-1	98	-1	-1
1	1	1	83	1	1
5	4000	20000	84	20000	20000
4000	5	20000	88	20000	20000
-500	433	-19892	91	-216500	-19892
-114	-233	26562	94	26562	26562

可以看到程序需要执行的指令数为  $82 \sim 98(1 + 16 \times (5 \sim 6) + 1)$ , 所以程序最多需要 98 条指令. 对于测试样例, 执行指令数的平均为 88.6.

但是, 上面的算法必须执行 16 次循环, 不论 R0 剩余的 bits 是否全是 0, 所以可以针对这一点进行优化. 注意到  $R0 \& (R0 - 1)$  是 R0 去掉最低的 1 的结果, 因此, 可以加上这一操作, 以减少循环次数. 同时, 负数的最高位为 1, 对于负数, 可以对其取相反数, 再对最后的结过取相反数, 这样可以减少循环次数. 对应的汇编代码如下:

```
1      ADD      R0, R0, #0
2      BRz      Stop
3
4      BRp      Pos
5      NOT      R0, R0      ; Negate R0
6      ADD      R0, R0, #1
7      ADD      R5, R5, #1
8
9 Pos   ADD      R2, R2, #1
10
11 Loop AND      R3, R0, R2
12      BRz      BitZero
13
14      ADD      R7, R7, R1
15      ADD      R1, R1, R1
16      ADD      R2, R2, R2
17      ADD      R4, R0, #-1 ; Remove the lowest 1
18      AND      R0, R0, R4  ; in R0
19      BRnp     Loop
```

```

20
21      ADD      R5, R5, #0
22      BRz      Stop
23      NOT      R7, R7      ; Negate R7
24      ADD      R7, R7, #1
25      HALT
26
27 BitZero ADD      R1, R1, R1
28      ADD      R2, R2, R2
29      BRnp     Loop
30
31      ADD      R5, R5, #0
32      BRz      Stop
33      NOT      R7, R7      ; Negate R7
34      ADD      R7, R7, #1
35 Stop  HALT

```

机器码如下:

```

1  0001000000100000 // x1020
2  0000010000011001 // x0419
3  0000001000000011 // x0203
4  1001000000111111 // x903F
5  0001000000100001 // x1021
6  0001101101100001 // x1B61
7  0001010010100001 // x14A1
8  0101011000000010 // x5602
9  0000010000001011 // x040B
10 0001111111000001 // x1FC1
11 0001001001000001 // x1241
12 0001010010000010 // x1482
13 0001100000111111 // x183F
14 010100000000100 // x5004
15 0000101111111000 // x0BF8
16 0001101101100000 // x1B60
17 0000010000001010 // x040A
18 1001111111111111 // x9FFF
19 0001111111100001 // x1FE1
20 1111000000100101 // xF025
21 0001001001000001 // x1241
22 0001010010000010 // x1482
23 0000101111111000 // x0BF0
24 0001101101100000 // x1B60
25 0000010000000010 // x0402
26 1001111111111111 // x9FFF
27 0001111111100001 // x1FE1
28 1111000000100101 // xF025

```

对测试样例的运行结果如下, 执行指令数同样通过分块求和得出:

R0	R1	R7	执行指令数	R0×R1	理论值
0	1	0	3	0	0
-32767	1	-32767	132	-32767	-32767
1	1	1	15	1	1
5	4000	20000	28	20000	20000
4000	5	20000	85	20000	20000
-500	433	-19892	75	-216500	-19892
-114	-233	26562	59	26562	26562

可以看到程序最多需要执行 132 条指令, 对于测试样例, 执行指令数的平均值为 56.7 条. 对于所有可能的情况, 经计算, 执行指令数的平均值为: 102.0 条, 其中各个范围内的平均值如下:

R0范围	执行指令数平均值
$-2^4 \sim 2^4 - 1$	31.7
$-2^5 \sim 2^5 - 1$	37.6
$-2^6 \sim 2^6 - 1$	43.8
$-2^7 \sim 2^7 - 1$	50.2
$-2^8 \sim 2^8 - 1$	56.6
$-2^9 \sim 2^9 - 1$	63.1
$-2^{10} \sim 2^{10} - 1$	69.5
$-2^{11} \sim 2^{11} - 1$	76.0
$-2^{12} \sim 2^{12} - 1$	82.5
$-2^{13} \sim 2^{13} - 1$	89.0
$-2^{14} \sim 2^{14} - 1$	95.5
$-2^{15} \sim 2^{15} - 1$	102.0

可以看到, 对于规模不大的情况, 该方案比更高效, 而当规模较大时, 原方案效率更高. 因此, 可以对 `R0` 的范围进行选择, 汇编代码如下:

```
1      .ORIG    x3000
2
3      ADD     R0, R0, #0
4      BRz     Stop1
5
6      BRp     Pos
7      NOT     R0, R0      ; Negate R0
8      ADD     R0, R0, #1
9      ADD     R5, R5, #1
10
11 Pos    ADD     R6, R0, R0 ;
12        BRn    Large      ; R0 is too Large
13        ADD     R6, R0, R0
14        BRn    Large
```

```

15      ADD      R2, R2, #1
16
17  Loop1      AND      R3, R0, R2
18            BRz      BitZero1
19
20            ADD      R7, R7, R1
21            ADD      R1, R1, R1
22            ADD      R2, R2, R2
23            ADD      R4, R0, #-1 ; Remove the lowest 1
24            AND      R0, R0, R4 ; in R0
25            BRnp     Loop1
26
27            ADD      R5, R5, #0
28            BRz      Stop1
29            NOT      R7, R7 ; Negate R7
30            ADD      R7, R7, #1
31            HALT
32
33  BitZero1    ADD      R1, R1, R1
34            ADD      R2, R2, R2
35            BRnp     Loop1
36
37            ADD      R5, R5, #0
38            BRz      Stop1
39            NOT      R7, R7 ; Negate R7
40            ADD      R7, R7, #1
41  Stop1      HALT
42
43  Large      ADD      R2, R2, #1
44  Loop2      AND      R3, R0, R2
45            BRz      BitZero2
46
47            ADD      R7, R7, R1
48  BitZero2    ADD      R1, R1, R1
49            ADD      R2, R2, R2
50            BRp      Loop2
51
52            ADD      R5, R5, #0
53            BRz      Stop2
54            NOT      R7, R7 ; Negate R7
55            ADD      R7, R7, #1
56  Stop2      HALT
57
58            .END

```

对应的机器码如下:

```
1 | 0001000000100000 // x1020
```

```

2  00000100000011101 // x041D
3  00000010000000011 // x0203
4  1001000000111111 // x903F
5  0001000000100001 // x1021
6  0001101101100001 // x1B61
7  0001010010100001 // x14A1
8  0001110000000000 // x1C00
9  0000100000010111 // x0817
10 0001110110000110 // x1D86
11 0000100000010101 // x0815
12 0101011000000010 // x5602
13 0000010000001011 // x040B
14 0001111111000001 // x1FC1
15 0001001001000001 // x1241
16 0001010010000010 // x1482
17 0001100000111111 // x183F
18 0101000000000100 // x5004
19 0000101111111000 // x0BF8
20 0001101101100000 // x1B60
21 0000010000001010 // x040A
22 1001111111111111 // x9FFF
23 0001111111100001 // x1FE1
24 1111000000100101 // xF025
25 0001001001000001 // x1241
26 0001010010000010 // x1482
27 0000101111110000 // x0BF0
28 0001101101100000 // x1B60
29 0000010000000010 // x0402
30 1001111111111111 // x9FFF
31 0001111111100001 // x1FE1
32 1111000000100101 // xF025
33 0101011000000010 // x5602
34 0000010000000001 // x0401
35 0001111111100001 // x1FC1
36 0001001001000001 // x1241
37 0001010010000010 // x1482
38 0000001111111010 // x03FA
39 0001101101100000 // x1B60
40 0000010000000010 // x0402
41 1001111111111111 // x9FFF
42 0001111111100001 // x1FE1
43 1111000000100101 // xF025

```

对测试样例的运行结果如下, 由于没有多余的寄存器, 对执行指令数的统计由附录的 C++ 程序给出:

R0	R1	R7	执行指令数	R0×R1	理论值
0	1	0	3	0	0
-32767	1	-32767	108	-32767	-32767
1	1	1	18	1	1
5	4000	20000	31	20000	20000
4000	5	20000	88	20000	20000
-500	433	-19892	79	-216500	-19892
-114	-233	26562	63	26562	26562

对于测试样例, 执行指令数的平均值为 55.7 条. 对于所有可能的情况, 经统计, 执行指令数的最大值与平均值分别为: 120 条和 97.4 条. 其中各个范围内的平均值如下:

R0范围	执行指令数平均值
$-2^4 \sim 2^4 - 1$	35.0
$-2^5 \sim 2^5 - 1$	41.1
$-2^6 \sim 2^6 - 1$	47.3
$-2^7 \sim 2^7 - 1$	53.7
$-2^8 \sim 2^8 - 1$	60.1
$-2^9 \sim 2^9 - 1$	66.6
$-2^{10} \sim 2^{10} - 1$	73.0
$-2^{11} \sim 2^{11} - 1$	79.5
$-2^{12} \sim 2^{12} - 1$	86.0
$-2^{13} \sim 2^{13} - 1$	92.5
$-2^{14} \sim 2^{14} - 1$	96.3
$-2^{15} \sim 2^{15} - 1$	97.4

至此, 优化结束, 最终版本就如上.

## 2.3 附录

统计 P 最终版本执行指令数的 C++ 程序 (部分) :

```
1  int16_t R[8] = {0, 0, 0, 0, 0, 0, 0, 0};
2  int n = 0, z = 0, p = 0;
3
4  void setCC(int16_t rst)
5  {
6      n = z = p = 0;
7      if (rst == 0)
8      {
9          z = 1;
10     }
11     else if (rst < 0)
12     {
13         n = 1;
14     }
15     else
16     {
17         p = 1;
18     }
19 }
20
21 int simulator(int16_t R0, int16_t R1)
22 {
23     memset(R, 0, sizeof(R));
24     R[0] = R0;
25     R[1] = R1;
26     int16_t PC = 0;
27
28     int count = 0;
29     while (true)
30     {
31         switch (PC)
32         {
33             case 0:
34                 PC++;
35                 count += 2;
36                 setCC(R[0]);
37                 if (z)
38                 {
39                     count++;
40                     PC = 4;
41                     break;
42                 }
43
```

```

44         count++;
45         if (p)
46             break;
47
48         count += 3;
49         R[0] = -R[0];
50         R[5] += 1;
51
52     case 1:
53         PC++;
54         count += 3;
55         R[2] = 1;
56         R[6] = 2 * R[0];
57         setCC(R[6]);
58         if (n)
59         {
60             PC = 5;
61             break;
62         }
63         count += 2;
64         R[6] += R[6];
65         setCC(R[6]);
66         if (n)
67         {
68             PC = 5;
69             break;
70         }
71
72     case 2: // Loop1
73         PC++;
74         count += 2;
75         R[3] = R[0] & R[2];
76         setCC(R[3]);
77         if (z)
78         {
79             PC = 3;
80             break;
81         }
82         count += 6;
83         R[7] += R[1];
84         R[1] += R[1];
85         R[2] += R[2];
86         R[4] = R[0] - 1;
87         R[0] &= R[4];
88         setCC(R[0]);
89         if (n || p)
90         {
91             PC = 2;

```



```

92         break;
93     }
94     count += 2;
95     setCC(R[5]);
96     if (z)
97     {
98         PC = 4;
99         break;
100    }
101    count += 3;
102    R[7] = -R[7];
103    if (R[7] != (int16_t)(R1 * R0))
104        std::cout << "Error: " << R[7] << " != " << (int16_t)(R1 *
R0) << std::endl;
105    return count;
106
107    case 3: //BitZero1
108        PC++;
109        count += 3;
110        R[1] += R[1];
111        R[2] += R[2];
112        setCC(R[2]);
113        if (n || p)
114        {
115            PC = 2;
116            break;
117        }
118        count += 2;
119        setCC(R[5]);
120        if (z)
121        {
122            PC = 4;
123            break;
124        }
125        count += 2;
126        R[7] = -R[7];
127
128    case 4: // Stop1
129        if (R[7] != (int16_t)(R1 * R0))
130            std::cout << "Error: " << R[7] << " != " << (int16_t)(R1 *
R0) << std::endl;
131        return count;
132
133    case 5: // Loop2
134        PC++;
135        count += 2;
136        R[3] = R[0] & R[2];
137        setCC(R[3]);

```

```

138         if (z)
139         {
140             PC = 6;
141             break;
142         }
143         count++;
144         R[7] += R[1];
145
146     case 6: //BitZero2
147         PC++;
148         count += 3;
149         R[1] += R[1];
150         R[2] += R[2];
151         setCC(R[2]);
152         if (n || p)
153         {
154             PC = 5;
155             break;
156         }
157         count += 2;
158         setCC(R[5]);
159         if (z)
160         {
161             PC = 7;
162             break;
163         }
164         count += 2;
165         R[7] = -R[7];
166
167     case 7: // Stop2
168         if (R[7] != (int16_t)(R1 * R0))
169             std::cout << "Error: " << R[7] << " != " << (int16_t)(R1 *
R0) << std::endl;
170         return count;
171
172
173     default:
174         std::cout << "Error: PC = " << PC << std::endl;
175         break;
176     }
177 }
178 }

```