第一次实验

姓名: 傅申 学号: PB20000051

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

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

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.

```cpp
#include <iostream>

int main()
{
    int16_t num_pos_overflow = 500 * 433;
    int16_t num_neg_overflow = 500 * -433;
    std::cout << num_pos_overflow << std::endl;
    std::cout << num_neg_overflow << std::endl;
    return 0;
}
```

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

```
19892
-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

initial R2 to R7 $\leftarrow 0$
loop:
if R0 $= 0$ **then**
  HALT
else
  R0 $\leftarrow$ R0 $- 1$
  R7 $\leftarrow$ R7 $+$ R1
  goto *loop*

写成对应的机器码与汇编程序分别如下:

```
0001 000 000 1 00000 ; setCC
0000 010 000000011   ; BRz 3
0001 111 111 000 001 ; R7 = R7 + R1
0001 000 000 1 11111 ; R0 = R0 - 1
0000 111 111111100   ; BRnzp -4

; halt
1111 0000 00100101
```

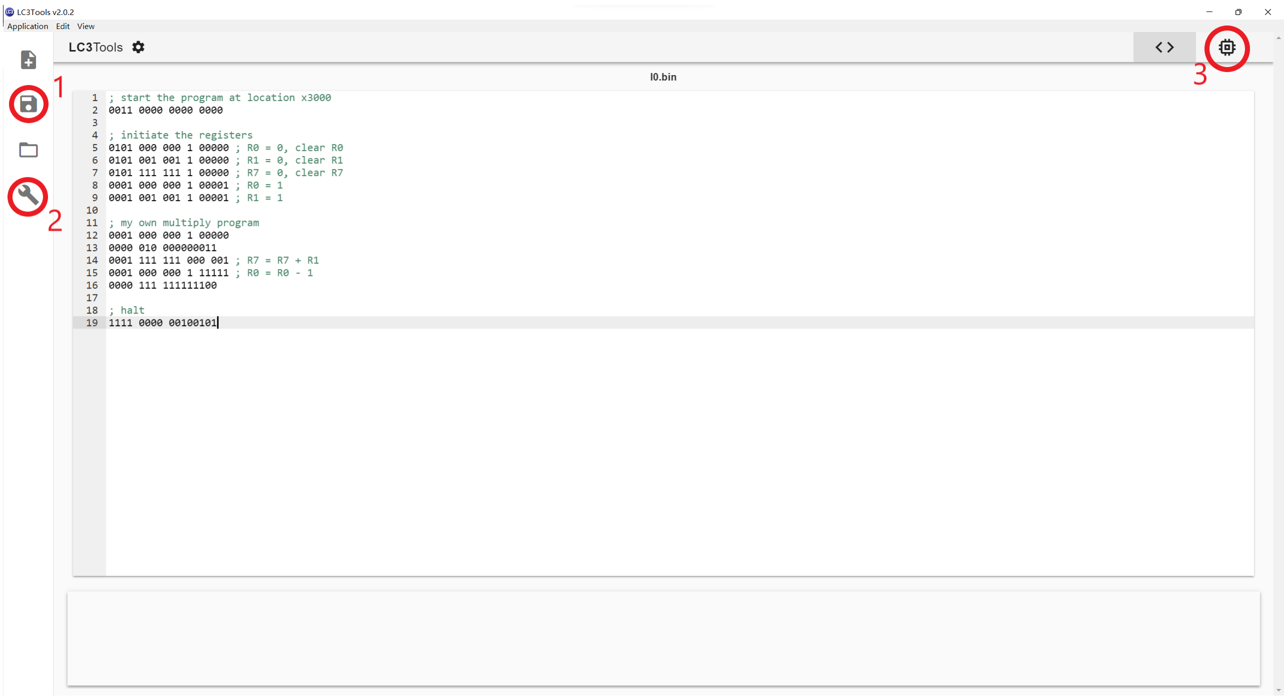
```
        ADD     R0, R0, #0
LOOP    BRz     STOP
        ADD     R7, R7, R1
        ADD     R0, R0, -1
        BRnzp   LOOP
STOP    HALT
```

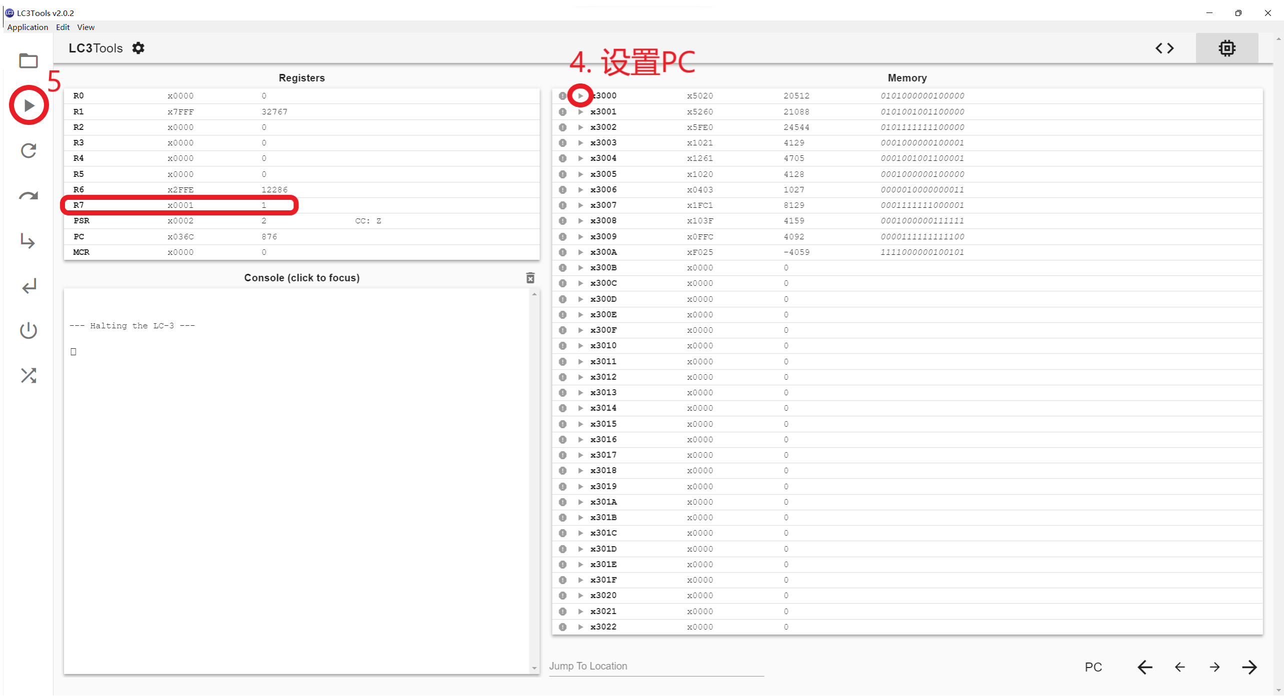
程序总共 5 行, 将程序的主体写入如下的 `bin` 文件中, 在 LC3Tools v2.0.2 中进行测试. 将 `bin` 文件编译后在模拟器中运行, 运行结果将显示在 Register 栏的 `R7` 中.

```
; start the program at location x3000
0011 0000 0000 0000

; initiate the registers
0101 000 000 1 00000 ; R0 = 0, clear R0
0101 001 001 1 00000 ; R1 = 0, clear R1
0101 111 111 1 00000 ; R7 = 0, clear R7
0001 000 000 1 00001 ; R0 = 1
0001 001 001 1 00001 ; R1 = 1

; my own multiply program
0001 000 000 1 00000 ; setCC(R0)
```

```
0000 010 000000011   ; BRz 3
0001 111 111 000 001 ; R7 = R7 + R1
0001 000 000 1 11111 ; R0 = R0 - 1
0000 111 111111100   ; BRnzp -4

; halt
1111 0000 00100101
```

测试过程如下两图，对于 `R0`，`R1` 初始值较大的情况，可以设置 `PC` 为初始化寄存器后一条指令，然后再手动调整各个寄存器的值，最后运行并检查。





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 $2^{16}b \equiv 0 (\bmod 2^{16})$, 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, 那么可以将机器码和汇编程序修改如下:

```
0001 111 111 000 001 ; R7 = R7 + R1
0001 000 000 1 11111 ; R0 = R0 - 1
0000 101 111111101   ; BRnp -3

; halt
1111 0000 00100101
```

```
LOOP    ADD     R7, R7, R1
        ADD     R0, R0, -1
        BRnp    LOOP
STOP    HALT
```

可以看到程序从 5 行缩短到了 3 行, 按照上面的测试方法, 测试机器码与测试结果如下

```
; start the program at location x3000
0011 0000 0000 0000

; initiate the registers
0101 000 000 1 00000 ; R0 = 0, clear R0
0101 001 001 1 00000 ; R1 = 0, clear R1
0101 111 111 1 00000 ; R7 = 0, clear R7
0001 000 000 1 00001 ; R0 = 1
0001 001 001 1 00001 ; R1 = 1

; my own multiply program
0001 111 111 000 001 ; R7 = R7 + R1
0001 000 000 1 11111 ; R0 = R0 - 1
0000 101 111111101   ; BRnp -3

; halt
1111 0000 00100101
```

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

initial R2 to R7 $\leftarrow 0$
if R0 $=0$ || R1 $=0$ **then**
  HALT
else
  foreach i **in** `R0[15→0]` **do**
    R7 $\leftarrow$ R7 + R7
    if $i=1$ **then**
      R7 $\leftarrow$ R7 + R1

2.2 P 版本程序

2.2.1 最初版本

写出上面算法的机器码如下

```
0001 000 000 1 00000 ; setCC(R0)
0000 010 000001010   ; BRz 10
0001 001 001 1 00000 ; setCC(R1)
0000 010 000001000   ; BRz 8
0001 010 010 1 01111 ; R2 = 15
0001 111 111 000 111 ; R7 = R7 + R7
0001 000 000 1 00000 ; setCC(R0)
0000 011 000000001   ; BRzp 1 // R0[15] == 0
0001 111 111 000 001 ; R7 = R7 + R1
0001 000 000 000 000 ; R0 = R0 << 1
0001 010 010 1 11111 ; R2 = R2 - 1
0000 011 111111001   ; BRzp -7

; halt
1111 0000 00100101
```

对应的汇编程序如下:

```
        ADD     R0, R0, #0
        BRz     STOP
        ADD     R1, R1, #0
        BRz     STOP
        ADD     R2, R2, #15
LOOP    ADD     R7, R7, R7
        ADD     R0, R0, #0
        BRzp    ODD
        ADD     R7, R7, R1
ODD     ADD     R0, R0, R0
        ADD     R2, R2, #-1
        BRzp    LOOP
STOP    HALT
```

测试机器码与对测试样例的运行结果如下:

```
; start the program at location x3000
0011 0000 0000 0000

; initiate the registers
0101 000 000 1 00000 ; R0 = 0, clear R0
0101 001 001 1 00000 ; R1 = 0, clear R1
0101 010 010 1 00000 ; R2 = 0, clear R2
0101 111 111 1 00000 ; R7 = 0, clear R7
0001 000 000 1 00001 ; R0 = 1
0001 001 001 1 00001 ; R1 = 1

; my own multiply program
0001 000 000 1 00000 ; setCC(R0)
0000 010 000001010   ; BRz 10
0001 001 001 1 00000 ; setCC(R1)
0000 010 000001000   ; BRz 8
0001 010 010 1 01111 ; R2 = 15
0001 111 111 000 111 ; R7 = R7 + R7
0001 000 000 1 00000 ; setCC(R0)
0000 011 000000001   ; BRzp 1 // R0[15] == 0
0001 111 111 000 001 ; R7 = R7 + R1
0001 000 000 000 000 ; R0 = R0 << 1
0001 010 010 1 11111 ; R2 = R2 - 1
0000 011 111111001   ; BRzp -7

; halt
1111 0000 00100101
```

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，对应的汇编代码如下:

```
        .ORIG   x3000

        ADD     R3, R3, #2
        ADD     R0, R0, #0
        BRz     STOP

        ADD     R3, R3, #2
        ADD     R1, R1, #0
        BRz     STOP

        ADD     R3, R3, #1
        ADD     R2, R2, #15

LOOP    ADD     R3, R3, #3
        ADD     R7, R7, R7
        ADD     R0, R0, #0
        BRzp    ODD         ; R1[15] = 0

        ADD     R3, R3, #1
        ADD     R7, R7, R1

ODD     ADD     R3, R3, #3
        ADD     R0, R0, R0
        ADD     R2, R2, #-1
        BRzp    LOOP        ; Loop 16 times

STOP    ADD     R3, R3, #1
        HALT
        .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：

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

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

initial R2 to R7 $\leftarrow 0$
$\texttt{R2} \leftarrow \texttt{x0001}$
loop: **if** $\texttt{R0 \& R2} \neq 0$ **then**
    $\texttt{R7} \leftarrow \texttt{R7} + \texttt{R1}$
$\texttt{R1} \leftarrow \texttt{R1} + \texttt{R1}$
$\texttt{R2} \leftarrow \texttt{R2} + \texttt{R2}$
if $\texttt{R2} \neq 0$ **then**
    goto *loop*

HALT

对应的机器码为：

```
0001 010 010 1 00001 ; R2 = 1
0101 011 000 000 010 ; R3 = R0 & R2
0000 010 000000001   ; BRz 1
0001 111 111 000 001 ; R7 = R7 + R1
0001 001 001 000 001 ; R1 = R1 + R1
0001 010 010 000 010 ; R2 = R2 + R2
0000 101 111111010   ; BRnp -6

; halt
1111 0000 00100101
```

汇编代码为：

```
        ADD     R2, R2, #1
Loop    AND     R3, R0, R2
        BRz     BitZero
        ADD     R7, R7, R1
BitZero ADD     R1, R1, R1
        ADD     R2, R2, R2
        BRnp    Loop
        HALT
```

测试机器码与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：

```
; start the program at location x3000
0011 0000 0000 0000

```

```
; initiate the registers
0101 000 000 1 00000 ; R0 = 0, clear R0
0101 001 001 1 00000 ; R1 = 0, clear R1
0101 010 010 1 00000 ; R2 = 0, clear R2
0101 111 111 1 00000 ; R7 = 0, clear R7
0001 000 000 1 00001 ; R0 = 1
0001 001 001 1 00001 ; R1 = 1

; my own multiply program
0001 010 010 1 00001 ; R2 = 1
0101 011 000 000 010 ; R3 = R0 & R2
0000 010 000000001   ; BRz 1
0001 111 111 000 001 ; R7 = R7 + R1
0001 001 001 000 001 ; R1 = R1 + R1
0001 010 010 000 010 ; R2 = R2 + R2
0000 101 111111010   ; BRnp -6

; halt
1111 0000 00100101
```

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 $82 \sim 98(1+16\times(5\sim6)+1)$, 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 `R0` 和 `R1` 同时取相反数. 而且程序必然是在当前 bit 为 `1` 时停止. 根据以上分析, 修改对应的机器码如下:

```
0001 000 000 1 00000 ; setCC(R0)
0000 010 000010001   ; BRz 17
0000 001 000000100   ; BRp 4
1001 000 000 111111  ; Negate R0
0001 000 000 1 00001
1001 001 001 111111  ; Negate R1
0001 001 001 1 00001
0001 010 010 1 00001 ; R2 = 1
0101 011 000 000 010 ; R3 = R0 & R2
0000 101 000000011   ; BRnp 3
```

```
0001 001 001 000 001 ; R1 = R1 + R1
0001 010 010 000 010 ; R2 = R2 + R2
0000 111 111111011   ; BRnzp -5
0001 111 111 000 001 ; R7 = R7 + R1
0001 001 001 000 001 ; R1 = R1 + R1
0001 010 010 000 010 ; R2 = R2 + R2
0001 100 000 1 11111 ; R4 = R0 - 1  // Remove the lowest
0101 000 100 000 000 ; R0 = R4 & R0 // 1 in R0
0000 101 111110101   ; BRnp -11

; halt
1111 0000 0010 0101
```

汇编代码如下:

```
        ADD     R0, R0, #0
        BRz     Stop

        BRp     Pos
        NOT     R0, R0      ; Negate R0
        ADD     R0, R0, #1
        NOT     R1, R1      ; Nagate R1
        ADD     R1, R1, #1

Pos     ADD     R2, R2, #1
Loop    AND     R3, R0, R2
        BRnp    BitOne

        ADD     R1, R1, R1
        ADD     R2, R2, R2
        BRnzp   Loop

BitOne  ADD     R7, R7, R1
        ADD     R1, R1, R1
        ADD     R2, R2, R2
        ADD     R4, R0, #-1 ; Remove the lowest
        AND     R0, R4, R0  ; 1 in R0
        BRnp    Loop

Stop    HALT
```

测试机器码与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:

```
; start the program at location x3000
0011 0000 0000 0000

; initiate the registers
```

```
0101 000 000 1 00000 ; R0 = 0, clear R0
0101 001 001 1 00000 ; R1 = 0, clear R1
0101 010 010 1 00000 ; R2 = 0, clear R2
0101 111 111 1 00000 ; R7 = 0, clear R7
0001 000 000 1 00001 ; R0 = 1
0001 001 001 1 00001 ; R1 = 1

; my own multiply program
0001 000 000 1 00000 ; setCC(R0)
0000 010 000010001   ; BRz 17
0000 001 000000100   ; BRp 4
1001 000 000 111111  ; Negate R0
0001 000 000 1 00001
1001 001 001 111111  ; Negate R1
0001 001 001 1 00001
0001 010 010 1 00001 ; R2 = 1
0101 011 000 000 010 ; R3 = R0 & R2
0000 101 000000011   ; BRnp 3
0001 001 001 000 001 ; R1 = R1 + R1
0001 010 010 000 010 ; R2 = R2 + R2
0000 111 111111011   ; BRnzp -5
0001 111 111 000 001 ; R7 = R7 + R1
0001 001 001 000 001 ; R1 = R1 + R1
0001 010 010 000 010 ; R2 = R2 + R2
0001 100 000 1 11111 ; R4 = R0 - 1  // Remove the lowest
0101 000 100 000 000 ; R0 = R4 & R0 // 1 in R0
0000 101 111110101   ; BRnp -11

; halt
1111 0000 00100101
```

R0	**R1**	**R7**	执行指令数	**R0×R1**	理论值
0	1	0	3	0	0
-32767	1	-32767	129	-32767	-32767
1	1	1	13	1	1
5	4000	20000	26	20000	20000
4000	5	20000	83	20000	20000
-500	433	-19892	72	-216500	-19892
-114	-233	26562	56	26562	26562

可以看到程序最多需要执行 129 条指令, 对于测试样例, 执行指令数的平均值为 54.6 条. 对于所有可能的情况, 经统计, 执行指令数的平均值为: 99.5 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
$-2^4 \sim 2^4 - 1$	29.1
$-2^5 \sim 2^5 - 1$	35.1
$-2^6 \sim 2^6 - 1$	41.4
$-2^7 \sim 2^7 - 1$	47.7
$-2^8 \sim 2^8 - 1$	54.1
$-2^9 \sim 2^9 - 1$	60.6
$-2^{10} \sim 2^{10} - 1$	67.0
$-2^{11} \sim 2^{11} - 1$	73.5
$-2^{12} \sim 2^{12} - 1$	80.0
$-2^{13} \sim 2^{13} - 1$	86.5
$-2^{14} \sim 2^{14} - 1$	93.0
$-2^{15} \sim 2^{15} - 1$	99.5

其中进行统计的程序 (简化版模拟器) 见附录. 可以看出, 对于数据规模不太大的情况, 优化的效果非常好.

2.3 附录

统计 P 最终版本执行指令数的 C++ 程序 (部分)：

```
int n = 0, z = 0, p = 0;
inline void set(int rst)
{
    n = z = p = 0;
    if (rst == 0)
        z = 1;
    else if (rst > 0)
        p = 1;
    else
        n = 1;
}

int sim(int16_t R0, int16_t R1)
{
    int PC = 0;
    int16_t R[8] = {R0, R1, 0, 0, 0, 0, 0, 0};
    int count = 0;
    while (true)
    {
        switch (PC)
        {
        case 0:
            PC++;
            count += 2;
            set(R[0]);
            if (z)
            {
                PC = 4;
                break;
            }
            count ++;
            if (p)
            {
                PC = 1;
                break;
            }
            count += 4;
            R[0] = -R[0];
            R[1] = -R[1];

        case 1: // pos
            PC++;
            count ++;
```

```
            R[2] += 1;

        case 2: // loop
            PC++;
            count += 2;
            R[3] = R[0] & R[2];
            set(R[3]);
            if (n || p)
            {
                PC = 3;
                break;
            }
            count += 3;
            R[1] += R[1];
            R[2] += R[2];
            PC = 2;
            break;

        case 3: // BitOne
            PC++;
            count += 6;
            R[7] += R[1];
            R[1] += R[1];
            R[2] += R[2];
            R[4] = R[0] - 1;
            R[0] &= R[4];
            set(R[0]);
            if (n || p)
            {
                PC = 2;
                break;
            }

        case 4: // halt
            count ++;
            if (R[7] != (int16_t)(R1 * R0))
                cout << "Error: " << R[7] << " != " << (int16_t)(R1 * R0) << endl;
            return count;

        default:
            cout << "Error: PC = " << PC << endl;
            break;
        }
    }
}
```