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

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

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 $c \equiv ab (\mathrm{mod}\ 2^{16})$, 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 $500 \times 433 = 216500 \equiv 19892 (\mathrm{mod}\ 2^{16})$

```
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

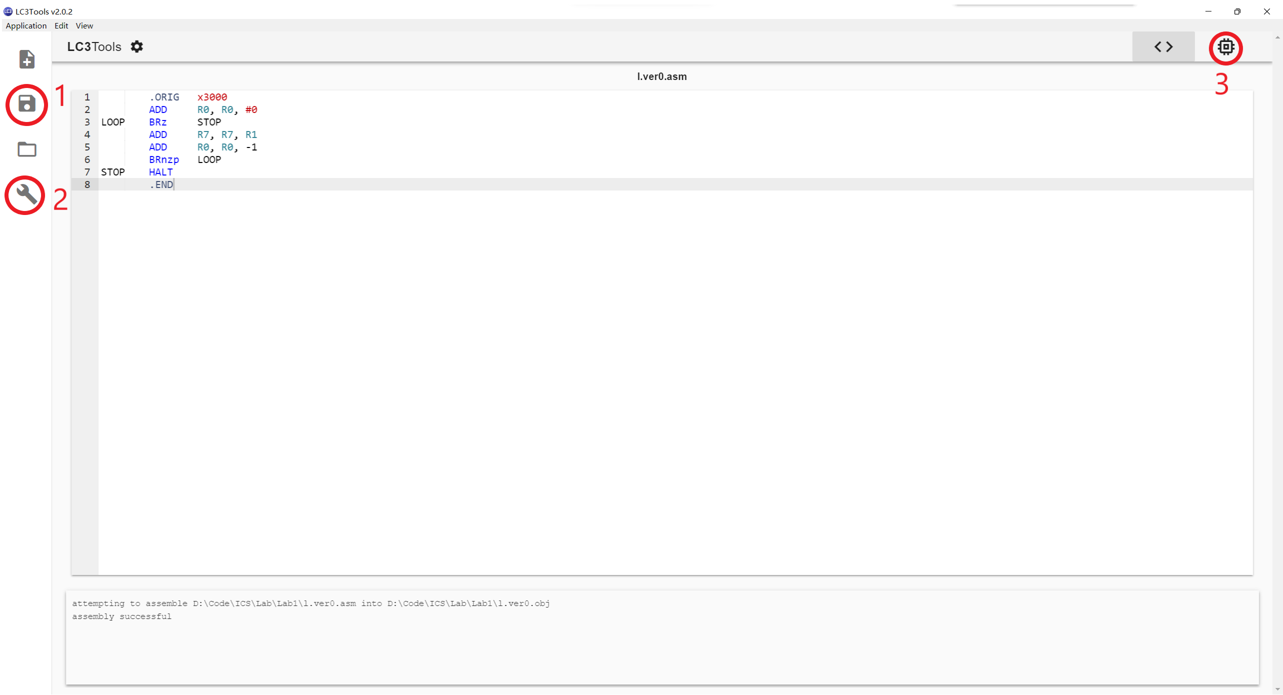
initial R2 to R7 $\leftarrow 0$
loop:
if $R0 = 0$ **then**
    HALT
else
    $R0 \leftarrow R0 - 1$
    $R7 \leftarrow R7 + R1$
    goto *loop*

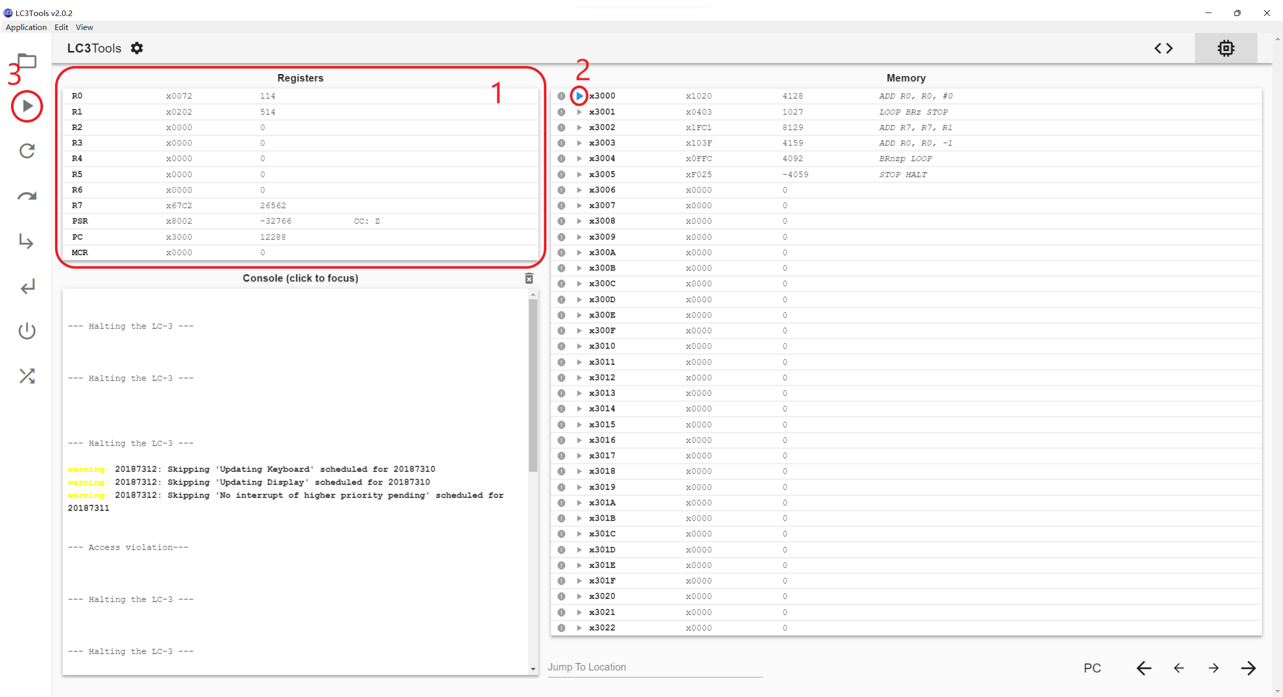
写成对应的汇编程序与机器码分别如下:

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

```
0001000000100000
0000010000000011
0001111111000001
0001000000111111
0000111111111100
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 $2^{16}b \equiv 0 (\bmod 2^{16})$，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，那么可以将汇编程序和机器码修改如下：

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

```
0001111111000001
0001000000111111
0000101111111100
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

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

```
        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
```

对应的机器码如下:

```
0001000000100000 // x1020
0000010000001010 // x040A
0001001001100000 // x1260
0000010000001000 // x0408
0001010010101111 // x14AF
0001111111000111 // x1FC7
0001000000100000 // x1020
0000011000000001 // x0601
0001000000000000 // x1000
0001010010111111 // x14BF
0000011111111001 // x07F9
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:

```
        .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 ← 0
R2 ← x0001
loop: if R0 & R2 ≠ 0 then
    R7 ← R7 + R1
R1 ← R1 + R1
R2 ← R2 + R2
if R2 ≠ 0 then
    goto loop

HALT
```

对应的汇编代码为:

```
        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
```

机器码为:

```
0001010010100001 // x14A1
0101011000000010 // x5602
0000010000000001 // x0401
0001111111000001 // x1FC1
0001001001000001 // x1241
0001010010000010 // x1482
0000101111111010 // x0BFA
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:

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

        BRp     Pos
        NOT     R0, R0      ; Negate R0
        ADD     R0, R0, #1
        ADD     R5, R5, #1

Pos     ADD     R2, R2, #1

Loop    AND     R3, R0, R2
        BRz     BitZero

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

```

        ADD     R5, R5, #0
        BRz     Stop
        NOT     R7, R7       ; Negate R7
        ADD     R7, R7, #1
        HALT

BitZero ADD     R1, R1, R1
        ADD     R2, R2, R2
        BRnp    Loop

        ADD     R5, R5, #0
        BRz     Stop
        NOT     R7, R7       ; Negate R7
        ADD     R7, R7, #1
Stop    HALT
```

机器码如下:

```
0001000000100000 // x1020
0000010000011001 // x0419
0000001000000011 // x0203
1001000000111111 // x903F
0001000000100001 // x1021
0001101101100001 // x1B61
0001010010100001 // x14A1
0101011000000010 // x5602
0000010000001011 // x040B
0001111111000001 // x1FC1
0001001001000001 // x1241
0001010010000010 // x1482
0001100000111111 // x183F
0101000000000100 // x5004
0000101111111000 // x0BF8
0001101101100000 // x1B60
0000010000001010 // x040A
1001111111111111 // x9FFF
0001111111100001 // x1FE1
1111000000100101 // xF025
0001001001000001 // x1241
0001010010000010 // x1482
0000101111110000 // x0BF0
0001101101100000 // x1B60
0000010000000010 // x0402
1001111111111111 // x9FFF
0001111111100001 // x1FE1
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:

```
            .ORIG   x3000

            ADD     R0, R0, #0
            BRz     Stop1

            BRp     Pos
            NOT     R0, R0      ; Negate R0
            ADD     R0, R0, #1
            ADD     R5, R5, #1

Pos         ADD     R6, R0, R0  ;
            BRn     Large       ; R0 is too Large
            ADD     R6, R0, R0
            BRn     Large
```

```
            ADD     R2, R2, #1

Loop1       AND     R3, R0, R2
            BRz     BitZero1

            ADD     R7, R7, R1
            ADD     R1, R1, R1
            ADD     R2, R2, R2
            ADD     R4, R0, #-1 ; Remove the lowest 1
            AND     R0, R0, R4  ; in R0
            BRnp    Loop1

            ADD     R5, R5, #0
            BRz     Stop1
            NOT     R7, R7      ; Negate R7
            ADD     R7, R7, #1
            HALT

BitZero1    ADD     R1, R1, R1
            ADD     R2, R2, R2
            BRnp    Loop1

            ADD     R5, R5, #0
            BRz     Stop1
            NOT     R7, R7      ; Negate R7
            ADD     R7, R7, #1
Stop1       HALT

Large       ADD     R2, R2, #1
Loop2       AND     R3, R0, R2
            BRz     BitZero2

            ADD     R7, R7, R1
BitZero2    ADD     R1, R1, R1
            ADD     R2, R2, R2
            BRp     Loop2

            ADD     R5, R5, #0
            BRz     Stop2
            NOT     R7, R7      ; Negate R7
            ADD     R7, R7, #1
Stop2       HALT

            .END
```

对应的机器码如下:

```
0001000000100000 // x1020
```

```
0000010000011101 // x041D
0000001000000011 // x0203
1001000000111111 // x903F
0001000000100001 // x1021
0001101101100001 // x1B61
0001010010100001 // x14A1
0001110000000000 // x1C00
0000100000010111 // x0817
0001110110000110 // x1D86
0000100000010101 // x0815
0101011000000010 // x5602
0000010000001011 // x040B
0001111111000001 // x1FC1
0001001001000001 // x1241
0001010010000010 // x1482
0001100000111111 // x183F
0101000000000100 // x5004
0000101111111000 // x0BF8
0001101101100000 // x1B60
0000010000001010 // x040A
1001111111111111 // x9FFF
0001111111100001 // x1FE1
1111000000100101 // xF025
0001001001000001 // x1241
0001010010000010 // x1482
0000101111110000 // x0BF0
0001101101100000 // x1B60
0000010000000010 // x0402
1001111111111111 // x9FFF
0001111111100001 // x1FE1
1111000000100101 // xF025
0101011000000010 // x5602
0000010000000001 // x0401
0001111111000001 // x1FC1
0001001001000001 // x1241
0001010010000010 // x1482
0000001111111010 // x03FA
0001101101100000 // x1B60
0000010000000010 // x0402
1001111111111111 // x9FFF
0001111111100001 // x1FE1
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)：

```cpp
int16_t R[8] = {0, 0, 0, 0, 0, 0, 0, 0};
int n = 0, z = 0, p = 0;

void setCC(int16_t rst)
{
    n = z = p = 0;
    if (rst == 0)
    {
        z = 1;
    }
    else if (rst < 0)
    {
        n = 1;
    }
    else
    {
        p = 1;
    }
}

int simulator(int16_t R0, int16_t R1)
{
    memset(R, 0, sizeof(R));
    R[0] = R0;
    R[1] = R1;
    int16_t PC = 0;

    int count = 0;
    while (true)
    {
        switch (PC)
        {
        case 0:
            PC++;
            count += 2;
            setCC(R[0]);
            if (z)
            {
                count++;
                PC = 4;
                break;
            }

```

```
            count++;
            if (p)
                break;

            count += 3;
            R[0] = -R[0];
            R[5] += 1;

        case 1:
            PC++;
            count += 3;
            R[2] = 1;
            R[6] = 2 * R[0];
            setCC(R[6]);
            if (n)
            {
                PC = 5;
                break;
            }
            count += 2;
            R[6] += R[6];
            setCC(R[6]);
            if (n)
            {
                PC = 5;
                break;
            }

        case 2: // Loop1
            PC++;
            count += 2;
            R[3] = R[0] & R[2];
            setCC(R[3]);
            if (z)
            {
                PC = 3;
                break;
            }
            count += 6;
            R[7] += R[1];
            R[1] += R[1];
            R[2] += R[2];
            R[4] = R[0] - 1;
            R[0] &= R[4];
            setCC(R[0]);
            if (n || p)
            {
                PC = 2;
```

```
                break;
            }
            count += 2;
            setCC(R[5]);
            if (z)
            {
                PC = 4;
                break;
            }
            count += 3;
            R[7] = -R[7];
            if (R[7] != (int16_t)(R1 * R0))
                std::cout << "Error: " << R[7] << " != " << (int16_t)(R1 * R0) << std::endl;
            return count;

        case 3: //BitZero1
            PC++;
            count += 3;
            R[1] += R[1];
            R[2] += R[2];
            setCC(R[2]);
            if (n || p)
            {
                PC = 2;
                break;
            }
            count += 2;
            setCC(R[5]);
            if (z)
            {
                PC = 4;
                break;
            }
            count += 2;
            R[7] = -R[7];

        case 4: // Stop1
            if (R[7] != (int16_t)(R1 * R0))
                std::cout << "Error: " << R[7] << " != " << (int16_t)(R1 * R0) << std::endl;
            return count;

        case 5: // Loop2
            PC++;
            count += 2;
            R[3] = R[0] & R[2];
            setCC(R[3]);
```

```
            if (z)
            {
                PC = 6;
                break;
            }
            count++;
            R[7] += R[1];

        case 6: //BitZero2
            PC++;
            count += 3;
            R[1] += R[1];
            R[2] += R[2];
            setCC(R[2]);
            if (n || p)
            {
                PC = 5;
                break;
            }
            count += 2;
            setCC(R[5]);
            if (z)
            {
                PC = 7;
                break;
            }
            count += 2;
            R[7] = -R[7];

        case 7: // Stop2
            if (R[7] != (int16_t)(R1 * R0))
                std::cout << "Error: " << R[7] << " != " << (int16_t)(R1 * R0) << std::endl;
            return count;


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