Lab5 Riddle

2021.12 赖永凡 PB20061210

Task

For this experiment you need to functionally reproduce the c++ programs below.

```
int judge(int r0) {
    int i = 2;
    r1 = 1;
    while (i * i <= r0) {
        if (r0 % i == 0) {
            r1 = 0;
            break;
        }
        i++;
    }
    return r1;
}
```

P.S. r0 (an integer, $0 \leq r0 \leq 100$) is given before program executes (just like lab1), and store the final result in r1. (no need to print out with TRAP).

Translation

Since this task asks us to write a C++ program in LC3, I think the term $Translation$ is more appropriate.

Plus, it is easy to find out that this specific C++ program is born to judge whether a number is a prime number. If so r0 is 1, else r0 is 0.

Although solving the $Riddle$ would benefit your comprehension, translating it line by line without not knowing its meaning is also an implementable method.

Anyway, let's see the result.

```
        .ORIG x3000

        JSR JUDGE
        HALT

JUDGE   AND R2,R2,#0    ;R2 serves as i
        ADD R1,R2,#1    ;R1 -> 1
        ADD R2,R2,#2    ;R2 -> 2

LOOP    AND R3,R3,#0    ;R3 stores R2 * R2
        AND R5,R5,#0    ;R5 stores negative R0
        ADD R4,R2,#0    ;R4 serves as sentinel
MUL     ADD R3,R3,R2
        ADD R4,R4,#-1
        BRNP MUL        ;this block does i * i
```

```
        NOT R5,R0
        ADD R5,R5,#1
        ADD R3,R5,R3    ;R3 = R3 - R0
        BRP DONE        ;this block tests if i * i <= R0

        NOT R3,R2
        ADD R3,R3,#1    ;R3 = 0 - R2
        ADD R5,R0,#0    ;R5 = R0
MOD     ADD R5,R5,R3
        BRZ ALDONE
        BRP MOD         ;this block tests if R0 % i == 0

        ADD R2,R2,#1    ;if not, i++
        BRNZP LOOP
ALDONE  AND R1,R1,#0    ;if yes, set R1

DONE    RET

        .END
```

Although having known what it does, I still translated it literally.

Generally, a key word in C++ can be constructed by combination of LC3 opcodes, more specifically, we can use a branch of BR codes to develop a while loop in this case.

However, meeting with some operator that don't occur in LC3 can be a tricky problem. It means we need to establish little sub-function to fulfill the obligation.

First comes the i^2. Note that we have written two versions of $a \times b$ program in lab1, one uses the bit operation, the other just let a plus itself b times. Because i have already been restricted to positive number and ranges from 0 to $100(\sqrt{10000})$, the latter version is more efficient on average, and much more short obviously. Hence the three lines I used to calculate i^2.

The other problem is to test the remainder of r0 divided by i. Because i is an incrementing value, we can not use some miraculous fashion to quickly get the answer just like task 2 in lab4. Therefore, I turn to the simplest method or the original one. I keep r0 subtracting i until it reaches 0 or smaller. For we only need to test whether r0 could be finely divided by i, if during the decay r0 (stores in R5) turns negative without tapping zero, we can conclude that the remainder is not zero.

Till here, we roughly rewrite the whole program. Doing some check, I am satisfied to find that the output is correct. However, when r0 is 0 or 1, the output is also 1, which contradicts to the definition of prime number I have learned. Then I rewound to the initial C++ program, noticing that its output would also be 1. The rationale behind I think is when the program is initiated, we tend to consider r0 to be a prime number and wants to prove it false during the checking process. So, if we couldn't even begin the check, then no one would suspect its latent essence.