第五次实验

姓名: 傅申 学号: PB20000051

1 设计思路

假设我们有平方的子程序和取模的子程序, 分别为 `SQUARE` 和 `MODULE`, 它们的功能分别为 `R3 * R3 → R4` (考虑到 `R0<=10000`, 不必处理溢出的情况), `R0 % R3 → R4`, 那么原来的 C++ 函数就可以被重写为以下的汇编代码

```
JUDGE   ST    R7, SaveR7
        AND   R1, R1, #0
        ADD   R1, R1, #1  ; R1 = 1
        NOT   R2, R0
        ADD   R2, R2, #1  ; R2 = -R0
        AND   R3, R3, #0
        ADD   R3, R3, #2  ; R3 = 2
AGAIN   JSR   SQUARE
        ADD   R4, R4, R2  ; R4 = R4 - R0
        BRp   DONE
        JSR   MODULE
        ADD   R4, R4, #0
        BRz   BAD
        ADD   R3, R3, #1
        BRnzp AGAIN
BAD     AND   R1, R1, #0
DONE    LD    R7, SaveR7
        RET
```

其中求平方的子程序可以由如下的汇编代码实现, 其思想与 Lab1 p 版本中的代码一致

```
SQUARE  ST    R1, SaveR1
        AND   R1, R1, #0
        ADD   R1, R1, #1  ; R1 = 1
        ST    R2, SaveR2
        ADD   R5, R3, #0
        ADD   R6, R3, #0
```

```
7          ST    R3, SaveR3
8  AGAIN   AND   R2, R1, R5
9          BRnp  BitOne
10         ADD   R6, R6, R6
11         ADD   R1, R1, R1
12         BRnzp LOOP
13 BitOne  ADD   R4, R4, R6
14         ADD   R6, R6, R6
15         ADD   R1, R1, R1
16         ADD   R3, R5, #-1
17         AND   R5, R5, R3
18         BRnp  LOOP
19         LD    R1, SaveR1
20         LD    R2, SaveR2
21         LD    R3, SaveR3
22         RET
```

为了简化程序并减少不必要的代码, 我们可以修改 `MODULE` 的功能并将其集成到 `JUDGE` 子程序中, 其中新的功能为 `(-R0) % R3 → R4`, 这样既没有改变即将执行的分支, 还能简化汇编代码 (因为我们已经将 `-R0 → R2`), 利用 $-x \equiv -x + n (\mathrm{mod}\ n)$, 给出以下汇编代码.

```
1  MODULE  ADD   R4, R2, #0
2  AGAIN   ADD   R4, R4, R3
3          BRn   AGAIN
```

这样, `JUDGE` 子程序中第 12 行的指令可以省去. 将上面的汇编代码汇总到一起, 并修改相应的标签, 得到汇编代码如下

```
1          .ORIG x3000
2          JSR   JUDGE
3          HALT
4
5  ; judge subroutine: R1 = R0.isPrime()
6  JUDGE   ST    R7, SaveR7
7          AND   R1, R1, #0
8          ADD   R1, R1, #1  ; R1 = 1
9          NOT   R2, R0
10         ADD   R2, R2, #1  ; R2 = -R0
11         AND   R3, R3, #0
           ADD   R3, R3, #2  ; R3 = 2
```

```
AGAIN   JSR   SQUARE
        ADD   R4, R4, R2  ; R4 = R4 - R0
        BRp   DONE
MODULE  ADD   R4, R2, #0
AGAINm  ADD   R4, R4, R3
        BRn   AGAINm
        BRz   BAD
        ADD   R3, R3, #1
        BRnzp AGAIN
BAD     AND   R1, R1, #0
DONE    LD    R7, SaveR7
        RET

; square subroutine: R4 = R3 * R3
SQUARE  ST    R1, SaveR1
        AND   R1, R1, #0
        ADD   R1, R1, #1  ; R1 = 1
        ST    R2, SaveR2
        ADD   R5, R3, #0
        ADD   R6, R3, #0
        ST    R3, SaveR3
        AND   R4, R4, #0
AGAINs  AND   R2, R1, R5
        BRnp  BitOne
        ADD   R6, R6, R6
        ADD   R1, R1, R1
        BRnzp AGAINs
BitOne  ADD   R4, R4, R6
        ADD   R6, R6, R6
        ADD   R1, R1, R1
        ADD   R3, R5, #-1
        AND   R5, R5, R3
        BRnp  AGAINs
        LD    R1, SaveR1
        LD    R2, SaveR2
        LD    R3, SaveR3
        RET
SaveR1  .BLKW #1
SaveR2  .BLKW #1
SaveR3  .BLKW #1

```

```
53  SaveR7  .BLKW #1
54          .END
```

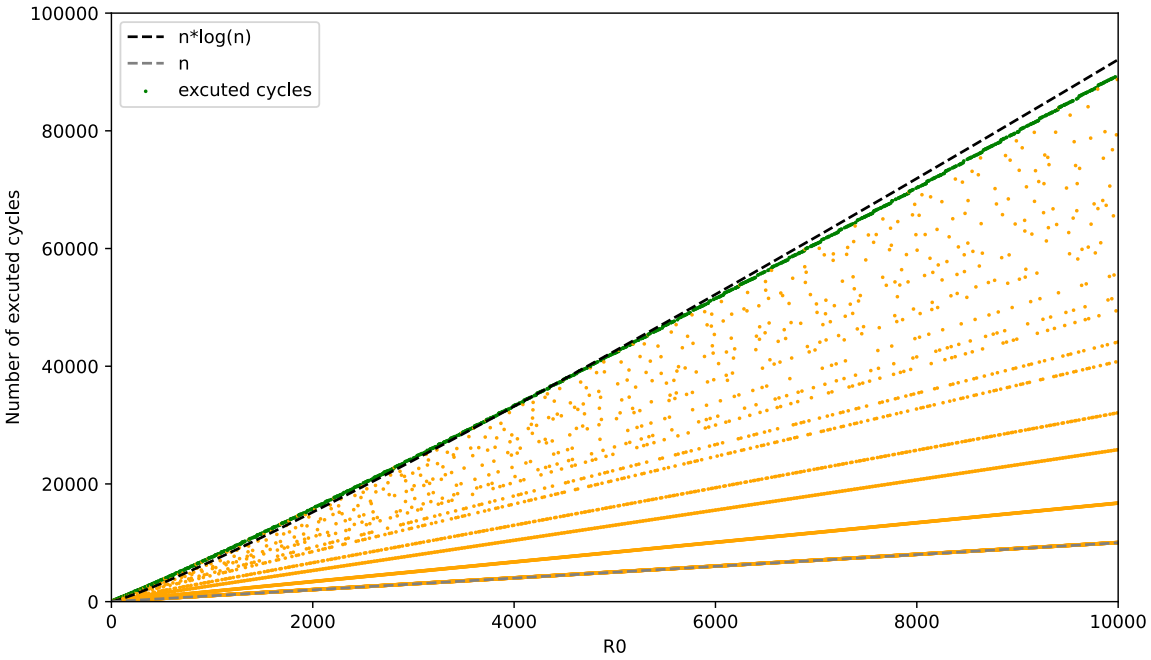
2 代码测试

分析代码可知 `SQUARE` 子程序的时间复杂度为 $O(\log \texttt{R3})$, `MODULE` 部分的时间复杂度为 $O(\texttt{R0}/\texttt{R3})$, 那么 `JUDGE` 子程序的时间复杂度为$O(n\log n)$, 其中利用到:

$$\sum_{i=1}^{\sqrt{n}}\log i=\log\left(\prod_{i=1}^{\sqrt{n}}i\right)=\log(\sqrt{n}!)=\log\left(\sqrt{2\pi\sqrt{n}}\left(\frac{\sqrt{n}}{\mathrm{e}}\right)^{\sqrt{n}}\exp\left(\frac{\theta}{12\sqrt{n}}\right)\right)\sim\sqrt{n}\log n$$

$$\sum_{i=1}^{\sqrt{n}}\frac{n}{i}=n\left(\ln\sqrt{n}+\gamma+\epsilon_{\sqrt{n}}\right)\sim n\log n$$

利用 LabA 和 LabS 对代码进行测试, 针对所有可能的情况 (`0<=R0<=10000`), 程序运行都是正确的, 且执行指令数与 `R0` 之间的关系为如下图所示



其中绿色的点代表素数, 橙色的点代表非素数. 程序的平均执行指令数为 13638.95 条.