第三次实验

姓名: 傅申 学号: PB20000051

1. Task 1: read and understand
2. Task 2: guess
3. Task 3: Optimize
    1. 减少不必要的指令
    2. 对 n 取模后递推
    3. 打表
        1. 暴力打表
        2. 对 n 取模 + 打表

1 Task 1: read and understand

收到的汇编代码主体部分如下:

```
        AND R7, R7, #0
        ADD R1, R1, #1
        ADD R2, R2, #1
        ADD R3, R3, #2
        LD  R5, NUMBER0  ; load #1023
        ADD R0, R0, #0   ; refresh nzp
        BRz OUTPUT
AGAIN   AND R4, R4, #0   ; clear R4
        ADD R4, R1, R1
        ADD R4, R4, R3
        AND R4, R4, R5   ; calculate f(n+3)
        ADD R1, R2, #0
        ADD R2, R3, #0
        ADD R3, R4, #0
        ADD R0, R0, #-1  ; now f(R0) is stored in R1
        BRp AGAIN        ; if R0=0,output R1
OUTPUT  ADD R7, R7, R1
        HALT
NUMBER0 .FILL x03FF
```

程序的大致思想为: `R1` `R2` `R3` 中分别存入当前计算到的 $f(m)$ $f(m+1)$ $f(m+2)$ 的值, 然后将 `R4` 用作临时寄存器进行递推运算, 如下

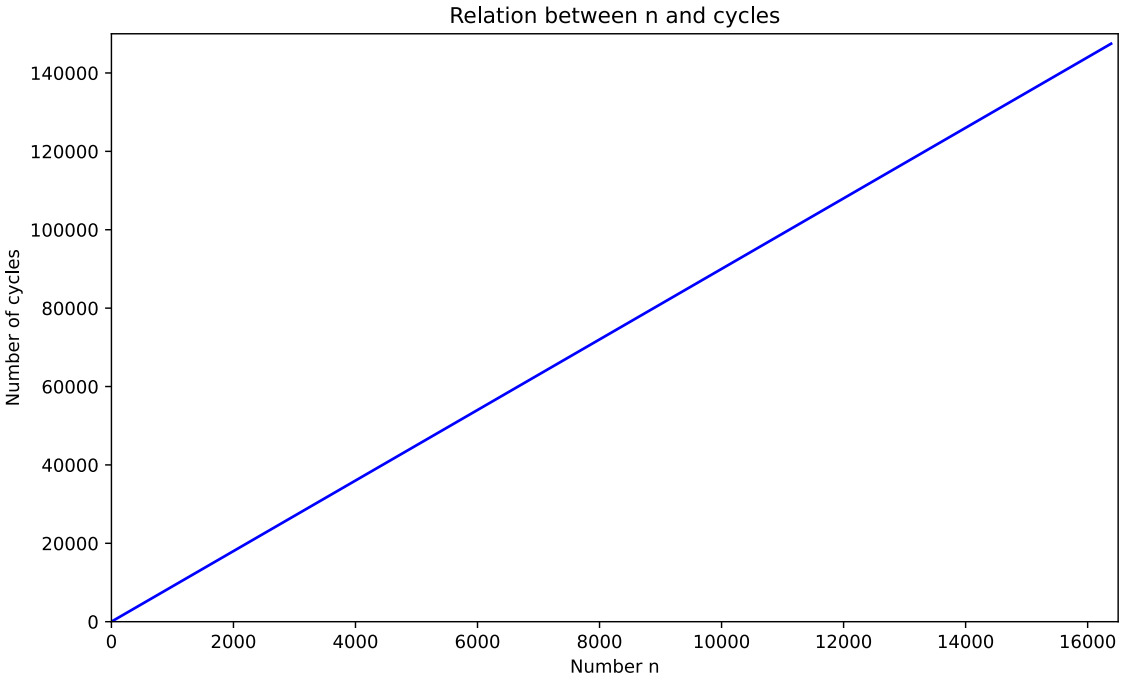
Algorithm 1: fib-raw

Data: n stored in R0
Result: $F(n)$ stored in R7
initialization: $\{\mathrm{R1}, \mathrm{R2}, \mathrm{R3}, \mathrm{R5}, \mathrm{R7}\} \leftarrow \{1, 1, 2, 1023, 0\}$
/* R1, R2, R3 stores $f(m)$, $f(m+1)$, $f(m+2)$ */
while R0 > 0 **do**
    $\mathrm{R4} \leftarrow 2 \times \mathrm{R1} + \mathrm{R3}$
    R4 ← R4&R5 // R4 = R4 mod 1024
    $\mathrm{R1} \leftarrow \mathrm{R2}$
    $\mathrm{R2} \leftarrow \mathrm{R3}$
    $\mathrm{R3} \leftarrow \mathrm{R4}$
    $\mathrm{R0} \leftarrow \mathrm{R0} - 1$
end
$\mathrm{R7} \leftarrow \mathrm{R1}$
HALT

对程序进行性能测试, 结果如下

N	24	144	576	1088	1092	2096	4200	8192	12000	14000	Average
cycles	225	1305	5193	9801	9837	18873	37809	73737	108009	126009	39079.8

对所有的 $n(0 \leq n \leq 16384)$, 执行指令数如下图所示



可以看出, 程序的时间复杂度为 $O(n)$, 但是常数很大.

2 Task 2: guess

收到的代码后四行为

```
F(a) .FILL #930
F(b) .FILL #1
F(c) .FILL #2
F(d) .FILL #1
```

其中有

$$930 = f(20) \quad 1 = f(0) = f(1) \quad 2 = f(2)$$

因此代码的作者学号应该为 20000200/20000201/20010200/20010201 中的一个, 查询学生名单, 发现只有 20000201 存在, 所以代码的作者应该为 PB20000201 曾川铭.

3 Task 3: Optimize

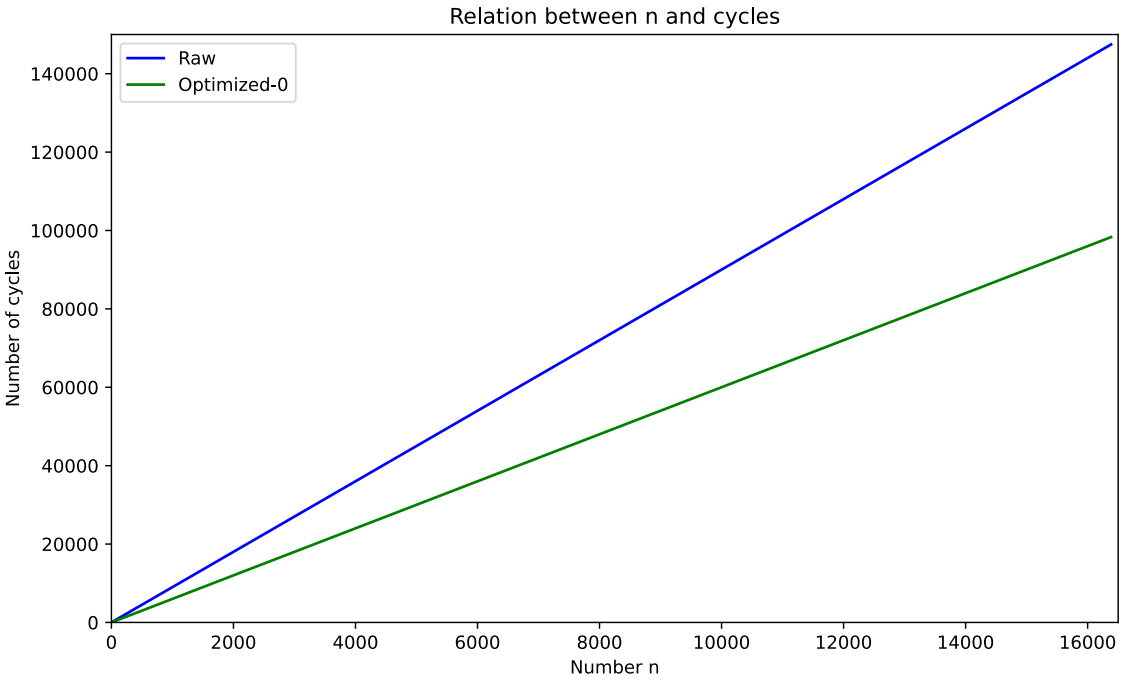
3.1 减少不必要的指令

观察代码不难发现, 在其他寄存器默认置 0 的情况下, 第 1 行的 `AND R7, R7, #0` 并没有必要, 而且第 8 行的 `AND R4, R4, #0` 同样没有必要, 因为下一行的 `ADD` 指令会覆盖掉原来 `R4` 的值.
同时, 第 10 行的 `ADD` 可以和第 14 行合并, 第 11 行的取模操作可以放在最后进行, 减少这些不必要的指令, 如下

```
1         .ORIG x300
2         ADD R1, R1, #1
3         ADD R2, R2, #1
4         ADD R3, R3, #2
5         LD  R5, NUMBER0  ; load #1023
6         ADD R0, R0, #0   ; refresh nzp
7         BRz OUTPUT
8  AGAIN  ADD R4, R1, R1
9         ADD R1, R2, #0
10        ADD R2, R3, #0
11        ADD R3, R4, R3
12        ADD R0, R0, #-1  ; now f(R0) is stored in R1
13        BRp AGAIN        ; if R0=0,output R1
14 OUTPUT AND R7, R5, R1
15        HALT
16 NUMBER0 .FILL x03FF
17        .END
```

对程序进行性能测试, 结果如下

N	24	144	576	1088	1092	2096	4200	8192	12000	14000	Average
cycles	152	872	3464	6536	6560	12584	25208	49160	72008	84008	26055.2



可以看出, 程序的时间复杂度仍为 $O(n)$, 但是常数减小大约 $1/3$.

3.2 对 n 取模后递推

尝试使用高级语言输出 $0 \leq n \leq 200$ 的结果, 发现

$$f(19) \neq f(147) \quad f(20) = f(148) \quad f(21) = f(149) \quad f(22) = f(150) \cdots$$

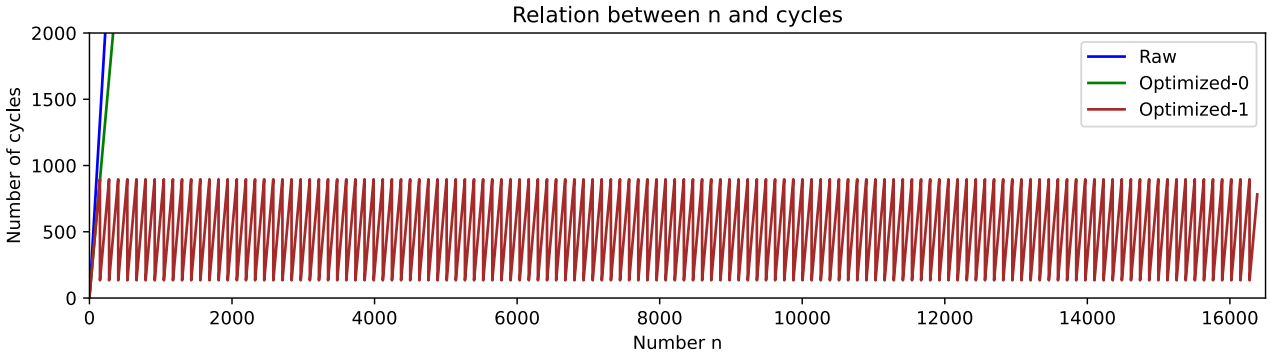
说明从第 20 项开始, $f(n)$ 出现了长度为 128 的循环, 即 $f(n) = f(((n-20) \bmod 128) + 20)$
根据这一思想, 对 n 取模后递推, 能减少指令执行次数, 汇编代码如下:

```
        .ORIG x3000
        ADD R1, R1, #1
        ADD R2, R2, #1
        ADD R3, R3, #2
        LD  R5, NUMBER0  ; load #1023
        ADD R0, R0, #-16
        ADD R0, R0, #-4
        BRn NEG
        LD  R6, NUMBER1  ; load #127
        AND R0, R0, R6
NEG     ADD R0, R0, #15
```

```
        ADD R0, R0, #5
        BRz OUTPUT
AGAIN   ADD R4, R1, R1
        ADD R1, R2, #0
        ADD R2, R3, #0
        ADD R3, R4, R3
        ADD R0, R0, #-1  ; now f(R0) is stored in R1
        BRp AGAIN        ; if R0=0,output R1
OUTPUT  AND R7, R5, R1
        HALT
NUMBER0 .FILL x03FF
NUMBER1 .FILL x007F
        .END
```

对程序进行性能测试, 结果如下

N	24	144	576	1088	1092	2096	4200	8192	12000	14000	Average
cycles	158	878	398	398	422	302	638	782	590	302	485.8



可以看到程序执行次数呈周期性变化, 测试得最大值为 895, 最小值为 8, 平均值为 514.

到这里, 常规优化结束, 下面进行两种打表.

3.3 打表

3.3.1 暴力打表

暴力打表, 即将所有的执行结果按顺序存入内存中, 然后再通过 `LD` 指令储存到 `R7` 中, 使用如下的 Python 程序编写汇编代码:

```python
fib = [1, 1, 2]
for i in range(3, 16385):
```

```
  fib.append((fib[i - 1] + 2 * fib[i - 3]) % 1024)
with open("path to asm", "w+") as f:
  f.write(".ORIG x3000\n")
  f.write("LEA R1, #3\n")
  f.write("ADD R1, R0, R1\n")
  f.write("LDR R7, R1, #0\n")
  f.write("HALT\n")
  for i in range(16385):
    f.write(".FILL #%d\n" % fib[i])
  f.write(".END\n")
```

得到的汇编代码部分如下

```
.ORIG x3000
LEA R1, #3
ADD R1, R0, R1
LDR R7, R1, #0
HALT
.FILL #1
.FILL #1
.FILL #2
; ...
.END
```

总共 16391 行, 对任意的 $0 \leq n \leq 16384$, 执行指令数均为 4 条. 但是代码过于臃肿, 占用了很大的内存空间.

3.3.2 对 n 取模 + 打表

我们已经知道了数列中有一个循环, 利用这一特性, 只在内存中存入 $f(0 \to 147)$, 使用如下 Python 程序编写汇编代码:

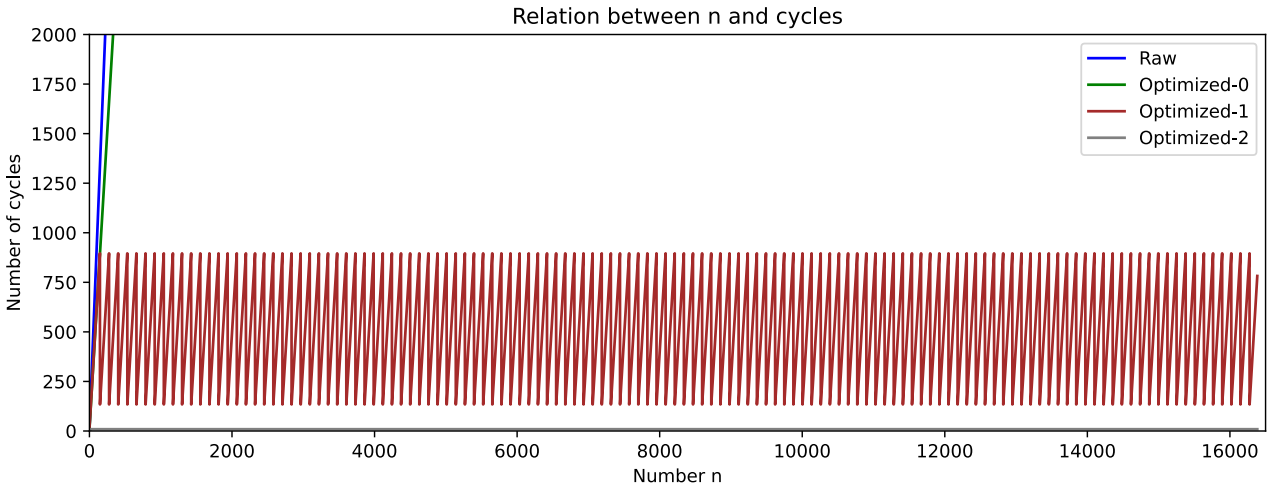
```
fib = [1, 1, 2]
for i in range(3, 148):
  fib.append((fib[i - 1] + 2 * fib[i - 3]) % 1024)
with open("path to asm", "w+") as f:
  f.write("        .ORIG x3000\n")
  f.write("        ADD R0, R0, #-16\n")
  f.write("        ADD R0, R0, #-4\n")
  f.write("        BRn NEG\n")
```

```
  f.write("        LD  R6, MOD\n")
  f.write("        AND R0, R0, R6\n")
  f.write("NEG     LD  R1, BASE\n")
  f.write("        ADD R1, R0, R1\n")
  f.write("        LDR R7, R1, #0\n")
  f.write("        HALT\n")
  f.write("MOD     .FILL #127\n")
  f.write("BASE    .FILL x301F\n")
  for i in range(148):
    f.write("        .FILL #%d\n" % fib[i])
  f.write("        .END\n")
```

得到的汇编代码部分如下

```
        .ORIG x3000
        ADD R0, R0, #-16
        ADD R0, R0, #-4
        BRn NEG
        LD  R6, MOD
        AND R0, R0, R6
NEG     LD  R1, BASE
        ADD R1, R0, R1
        LDR R7, R1, #0
        HALT
MOD     .FILL #127
BASE    .FILL x301F
        .FILL #1
        .FILL #1
        .FILL #2
        ; ...
        .END
```

代码共有 161 行, 执行指令数为 $number\ of\ cycles = \begin{cases} 7 & \text{if } n < 20 \\ 9 & \text{if } n \geq 20 \end{cases}$, 在保证了速度的同时减小了大小.
最后提交的代码即为 3.3.2 中的代码.
指令数随优化过程变化如下:


Relation between n and cycles
Number of cycles
Number n
Raw
Optimized-0
Optimized-1
Optimized-2
0
250
500
750
1000
1250
1500
1750
2000
0
2000
4000
6000
8000
10000
12000
14000
16000