ICS-lab1-report

- 姓名：魏剑宇
- 学号：PB17111586

算法

算法选择

考虑以下几种计算最大公因数的算法

1. 辗转相除法(Euclidean algorithm)
2. 质因数分解
3. **Binary gcd algorithm**

质因数分解效率太低。由于lc-3缺少除法指令，辗转相除法实现起来较为复杂，故考虑采用Binary gcd algorithm，基于位操作来实现算法。基于位操作的代码更加自然。

算法的一个递归描述如下（摘自Wikipedia[1]）

```
unsigned int gcd(unsigned int u, unsigned int v)
{
    // simple cases (termination)
    if (u == v)
        return u;

    if (u == 0)
        return v;

    if (v == 0)
        return u;

    // look for factors of 2
    if (~u & 1) // u is even
    {
        if (v & 1) // v is odd
            return gcd(u >> 1, v);
        else // both u and v are even
            return gcd(u >> 1, v >> 1) << 1;
    }

    if (~v & 1) // u is odd, v is even
        return gcd(u, v >> 1);

    // reduce larger argument
    if (u > v)
        return gcd((u - v) >> 1, v);

    return gcd((v - u) >> 1, u);
```

```
}
```

递归的实现执行的指令数显然会比非递归实现使用的指令数多，且需要我首先实现LC-3的栈，并且不便于我后续的优化，故我考虑通过非递归的方式实现。

最终选择算法的C语言描述为，**为了与我的代码契合，经过了细微修改**

```
unsigned int gcd(unsigned int u, unsigned int v)
{
  int shift;

  /* GCD(0,v) == v; GCD(u,0) == u, GCD(0,0) == 0 */
  if (u == 0) return v;
  if (v == 0) return u;

  /* Let shift := lg K, where K is the greatest power of 2
        dividing both u and v. */
  for (shift = 0; ((u | v) & 1) == 0; ++shift) {
         u >>= 1;
         v >>= 1;
  }

  while ((u & 1) == 0)
    u >>= 1;
  while ((v & 1) == 0)
    v >>= 1;
  /* From here on, u is always odd. */
  do {
       if (u > v) {
         unsigned int t = v; v = u; u = t; // Swap u and v.
       }

       v = v - u; // Here v >= u.
       while ((v & 1) == 0)  /* Loop X */
           v >>= 1;
     } while (v != 0);

  /* restore common factors of 2 */
  return u << shift;
}
```

算法效率

一般而言Binary gcd的时间复杂度为$O(n^2)$, n = max{number of bits of{u, v}}，空间复杂度为$O(1)$。然而，在LC-3中实现时，由于缺少右移指令，而在软件层面实现右移的时间复杂度为$O(n)$，故时间复杂度为 $O(n^3)$。**但经过了下面的LUT优化后，算法时间复杂度为**$O(n^2)$。

优化

由于算法操作的过程中，最耗时的部分是右移操作，故下面的优化大多针对右移操作进行。

Optimization #1

通过使用lut，来减少执行的代码数，考虑在LUT中存入所有数右移一位的数。这样，右移操作只需2条指令即可完成。这显然是足够的，因为int16最大为0x8000，而lc-3的内存有0x10000.

Optimization #2

然而在实现之前，突然想到，**LUT中存储右移一位的数是没有必要的。**因为算法中我实际不需要右移一位的数，只需要**经过右移后得到的奇数**，故不如在LUT中存储所有数经过右移后最终得到的奇数。

Optimization #3

然而上述优化有点问题，就是我在第一步找到`shift`时，仍然需要进行右移。此时又发现，**LUT中存在大量空间的浪费**，奇数右移得到的奇数显然就是它自己，所以没必要存下来。那么这些多余的空间，就可以用来**存储每个偶数需要右移多少位来得到奇数。**这样确定了不再需要进行右移了。

这样，LUT中的内存布局如下：

- x为偶数：x经过右移得到的奇数
- x为奇数：x-1需要右移的位数

如下进行填充LUT

```
import sys

with open(sys.argv[1], "a") as file:
    for x in range(0, 0x8000):
        if x & 1 == 0:
            while x != 0 and x & 1 == 0:
                x >>= 1
        else:
            x = x - 1
            shift = 0
            while x != 0 and x & 1 == 0:
                x >>= 1
                shift += 1
            x = shift
        file.write("        .FILL #" + str(x) + "\n")
```

Optimization #4

循环结尾有需要判断uv大小进行交换并相减的步骤，通过适当变换，可以优化为以下代码

```
loop    NOT R2,R0
        ADD R2,R1,R2
        BRzp swapuv ;that is, u-v-1>=0, u>v
        NOT R0,R2 ;R0 = NOT(u-v-1) = -(u-v-1)-1 = v-u
        BRnzp vodd
swapuv  ADD R1,R0,#0
        ADD R0,R2,#1
```

Optimization #5

LUT的结构过于复杂，导致**查询效率较低**，查询时还需要额外的奇偶性判断。故使用更简单的LUT，即每一位皆存储该位经过右移后得到的奇数。这样，**查询时只需要两条指令**。

而在`findshf`过程中，可以使用最朴素的过程找需要右移的位数。

经过测试，这个tradeoff是值得的。

Optimization #6

将部分循环结构由`while`改为`do while`，能减少几条指令数。

非法输入

若遇到非法输入，R0的值会直接被置为0，此值**不会**在正常输入的情况出现。

程序在开始时有如下非法输入的处理

```
ADD R0,R0,#0
BRnz exit1
ADD R1,R1,#0
BRnz exit1
//...
exit AND R0,R0,#0
end HALT
```

在处理后，R0的值会被置为0。

测试

测试中使用到liblc3.测试1到0x7fff间的10000组数据，最后得到结果

```
avg: 98.9273
min: 24
max: 144
```

---

1. https://en.wikipedia.org/wiki/Binary_GCD_algorithm↩