**实验报告——lab6 Learn from the past**

PB20151793 宋玮

***part1：****lab0l (lab1 L version)*

该实验是用尽量短的程序完成r0和r1的乘法，结果存储在r7中。

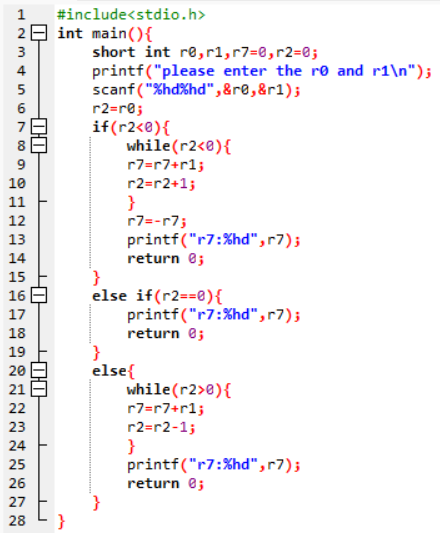
**算法思想：**

首先判断 R0 正负： 如果是正数（或 0），则每次对 R7 进行加 R1 操作后，R0 减 1，重复操作，直至 R0 减为 0。由于该过程对 R1 加了 R0 次。因此 R7 中的结果为 R0×R1。 如果是负数，则每次对 R7 进行加 R1 操作后，R0 加 1，重复操作，直至 R0 变为 0。 最后对 R7 中的结果取反加一，可得到 R0×R1。

用c语言撰写的版本如下，每一条语句基本遵循对应的lc-3汇编指令。

其中，变量名与lc-3中寄存器名相同。

不同的是，BR指令用循环（while语句）代替。

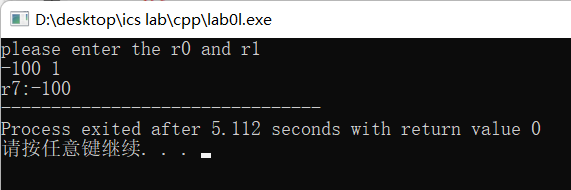


**运行结果：**

需要输入你想测试的r0和r1的值，

之后程序将输出r7的值，也即乘法结果。

如下图：**-100\*1**



***part2：****lab0p (lab1 P version)*

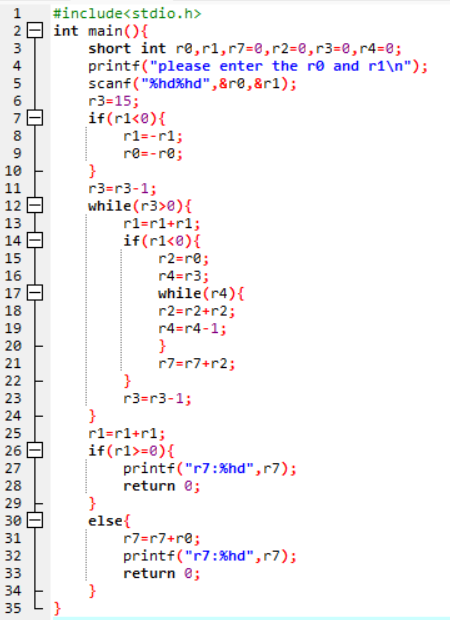
该实验是用尽量少的指令数完成r0和r1的乘法，结果存储在r7中。

**算法思想：**

由于 R0,R1 均为 01 串。因此在进行列竖式乘法计算时，每一行结果要么是 R0，要么 是 0，取决于 R1 对应位上是 1 还是 0。再通过对 R0 的移位后相加，则可以得到结果。 目前，仅能实现的移位方式是左移，即通过 R0=R0+R0，可以实现对 R1 的左移一位。 因此，本算法也采取自加左移方式。 而判断 R1 对应位上是 0 还是 1，也需要对 R1 进行左移，然后通过判断正负，判断该 位上是 0 还是 1。

注：当 R1 为负数时，采取 R1 和 R0 同时取反加一的操作，即同时取相反数，可以缩 减指令执行条数。

用c语言撰写的版本如下，变量名与lc-3中寄存器名相同。

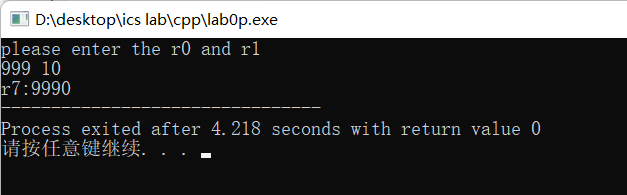


**运行结果：**

需要输入你想测试的r0和r1的值，

之后程序将输出r7的值，也即乘法结果。

如下图：**999\*10**



***part3：****fib (lab2 fibonacci)*

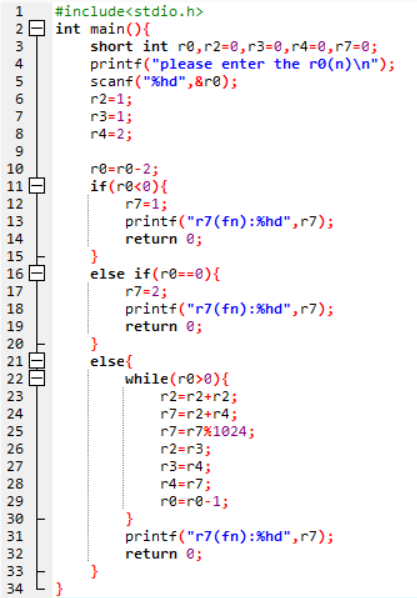
该实验是求f(n)，其中n存放在r0中。

**算法思想：**

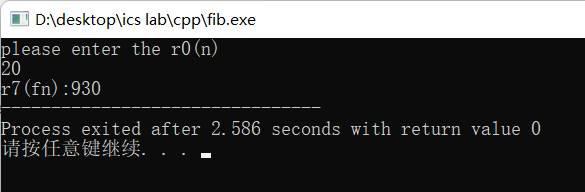
R2 存放 f(k-3)，R3 存放 f(k-2)，R4 存放 f(k-1). 首先对于特殊情况，n=1，n=2，做特殊情况判断，直接处理得到对于结果。 对于一般数据，每一次循环，进行 R7=R4+2\*R2 操作，并且对 R7 进行模 1024 操作。接着用此时的 R3 替换 R2，R4 替换 R3，R7 替换 R4，得到新的 f(k-3)，f(k-2)，f(k-1)。并且对计数减一。重复循环，直至计数为 0，跳转至结束（halt）。R7 中存放的即为f（n）。

用c语言撰写的版本如下，变量名与lc-3中寄存器名相同。

需要初始输入r0，程序输出为r7，即f(n)。



**运行结果：输入n=20**



***part4：****fib-opt (lab3 fibonacci)*

**算法思想：**

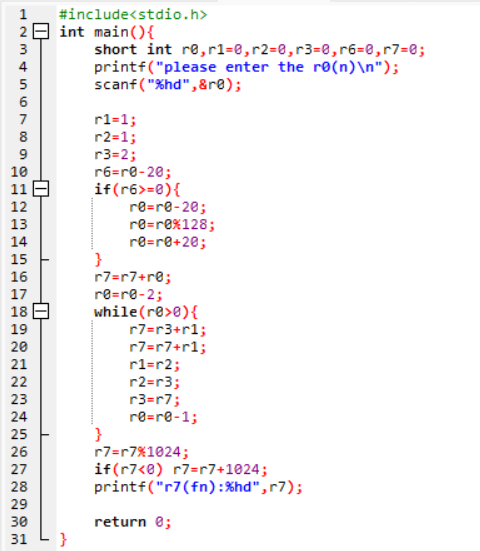
主要是在lab2 fibonacci做了一些改进，并且发现f(n)存在周期，因此可以利用这个性质，对于 n>=20 的数，先对其进行减 20，mode 128，再加 20 的操作。对于n<20的数，先不做操作，再利用改进的lab2求解。

用c语言撰写的版本如下，变量名与lc-3中寄存器名相同。

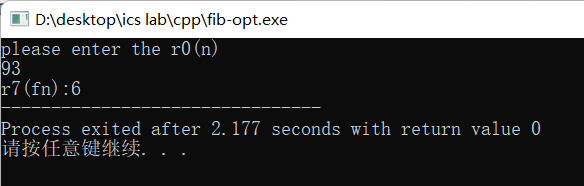
要注意的一点是：由于在c语言中%（取模运算），对负数取模的值可能还是负数，与lc-3汇编程序中and操作不同。因此在最后一个部分，我加上一个if语句。



需要初始输入r0，程序输出为r7，即f(n)。



**运行结果：输入n=93**



***part5：****rec (lab4 task1 rec)*

这个实验是一个填空实验。整体实现思想与c语言有一些不同。

首先，lc-3汇编程序中，不管是实现主程序，还是实现子程序，寄存器的值都在时刻改变，因此在利用c语言撰写时，我们应将寄存器变量设置为全局变量。



其次，lc-3汇编程序中，JSR指令跳到子程序前，r7会存储JSR下一条指令地址，方便返回时，直接return r7，即返回该指令地址。而在c语言实现时，不涉及到指令地址，因此我们需要模仿lc-3汇编程序，设置该“虚拟地址”。



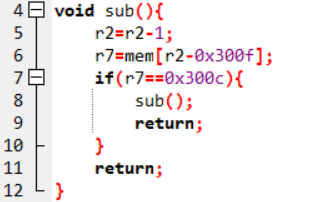
再者，整个程序中，只涉及到两条JSR指令，则只涉及两个子程序。因此我们可以根据判断当前r7的值，然后再选择调用哪个子程序。

最后还有一点，由于在整个程序中，我们频繁用到r7=mem(r2)，mem(r2)=r7，因此我用了一个数组mem[ ]来实现该操作。并且利用memx3019变量来表示lc-3汇编程序中的mem(x3019)。

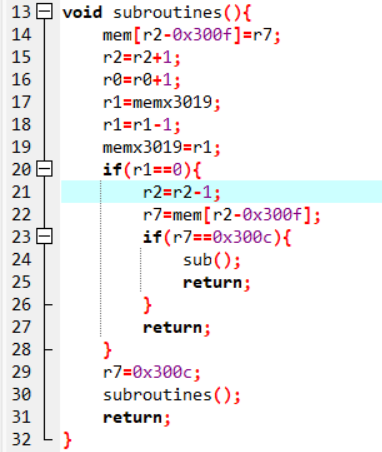


用c语言撰写的版本如下：

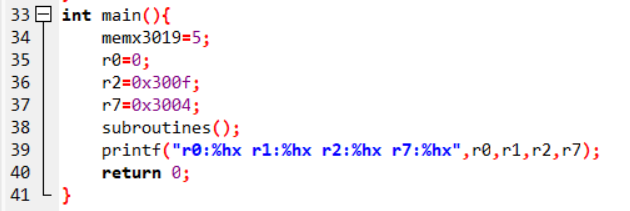
子程序1：



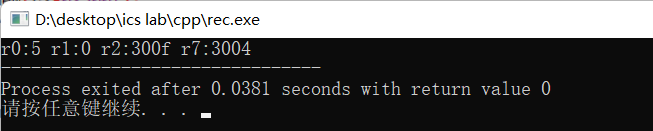
子程序2：



主程序：



**运行结果：输出为主要的四个变量r0，r1，r2，r7**



***part6：****mod (lab4 task2 mod)*

**算法思想：**

用c语言撰写的版本，每一条语句基本遵循对应的lc-3汇编指令。

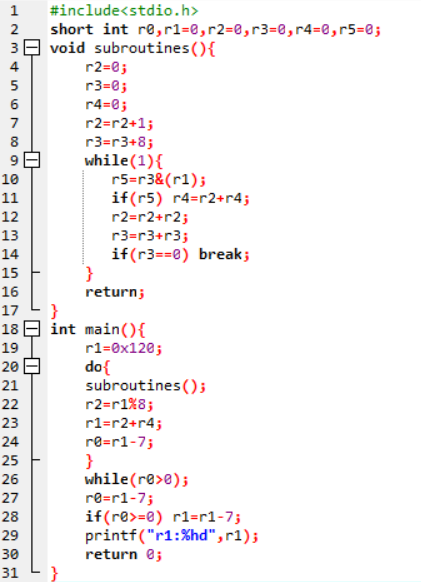
即子程序实现r1/8；

主程序求r1 mode 7的值。

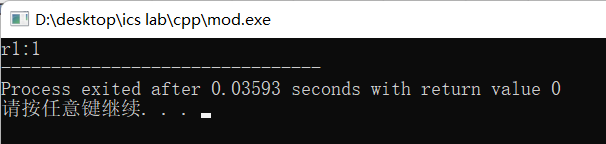
与part5同样的道理，由于寄存器的值时刻在变，我们需要在c语言程序中，把变量都定义为全局变量。

并且在and 7的操作部分，我直接用了%8，在操作数是正整数的情况下，两种是等效的。

最后输出的r1，为初始r1 mode 7的值。



**运行结果：**



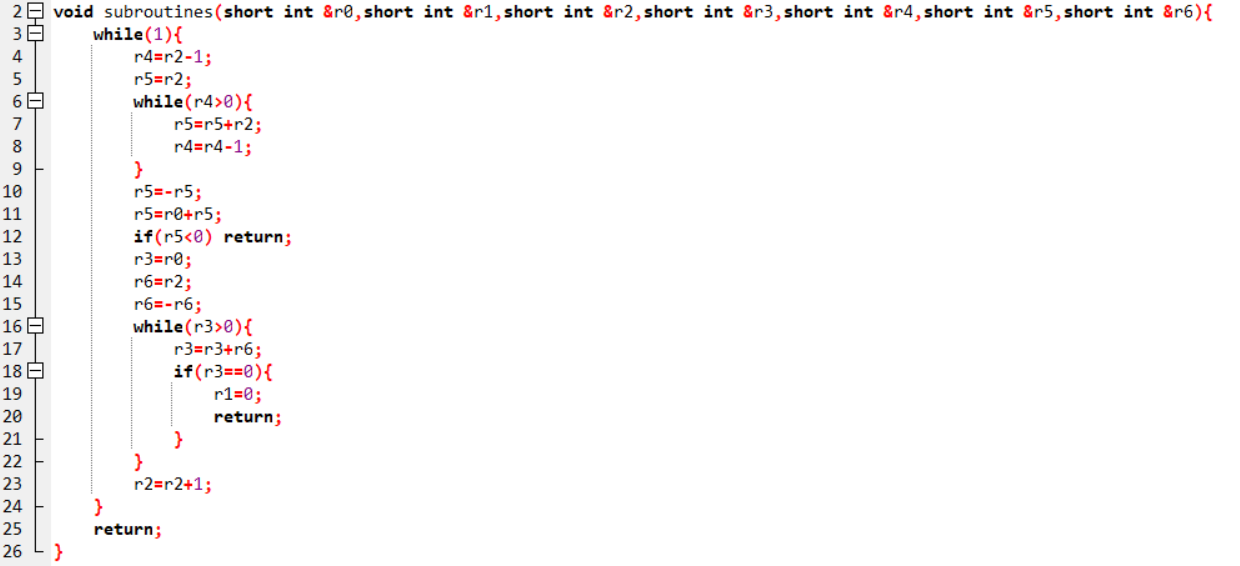
***part7：****prime (lab5 prime)*

**算法思想：**

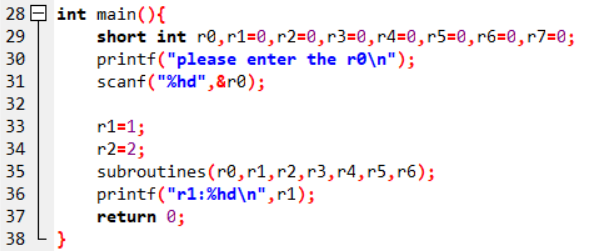
与上述两个part相同，由于寄存器的值时刻在变，我们需要在编写c语言程序中，注意到这一点。与前面两个part不同的是，由于该程序只有一个子程序，我们完全可以利用引用参数（&）的做法，来完成寄存器的值也随子程序变化。

其他部分基本相同。完全按照lc-3汇编程序的思路编写。

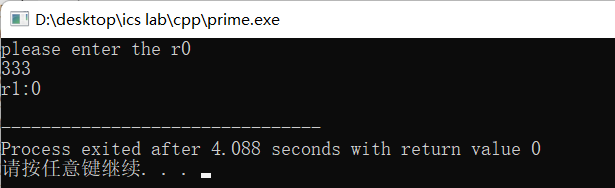
子程序：



主程序：



**运行结果：输入为r0，输出为r1**



***总结：***

通过用c语言编写之前的7个lc3程序，我有了如下一些体会。

高级语言程序（c语言）在编写上更加简单明了，并且操作更加灵活。

我觉得原因在于，高级语言给我们提供了更多的框架，如if判断语句，while循环语句，这更加符合我们的逻辑思维。而不用像在lc-3汇编程序中思考BR指令应该怎么判断什么，要跳转至哪里，以及BR指令要放到哪个位置。

并且lc-3汇编指令有限，而c语言定义的操作丰富且灵活。比如lc-3汇编指令甚至都没有乘法指令。当你需要用到乘法时，c语言一条语句可以搞定，而lc-3汇编程序需要很多的逻辑组合，指令组合才能实现这样一个简单的功能。

我认为乘法指令和除法指令需要被添加至lc3中，毕竟乘法和除法都是基础的运算，如果添加进去，将使得程序实现更加简单。

但是，我们也可以从lc3中学习到的一点是，返回（return）问题。在lc3中，返回时，是return r7，可以顺利跳转至r7所在的指令位置。这样的逻辑非常清楚。而在c语言中，在调用多个函数以后，甚至在多个函数互相调用以后，这个逻辑将变得十分复杂。甚至不知道return将跳转至哪个地方。我觉得，这是lc3中比较出彩的一点。