Lab 4 Reveal Yourself

实验内容

分别根据寄存器运行前后的数值变化和对7取余的功能，填补指令中缺失的若干二进制位。

代码分析

task 1

在该任务中，我们可以先假设起始指令地址为x3000，并对逐条指令的功能分析如下：

```
1110 010 000001110;   R2 = x300F
0101 000 000 1 00000; R0 = x0000;
0100 1 0000000000x;   R7 = x3003 PC = x3003 + ?
1111 0000 00100101;   TRAP x25 Halt
0111 111 010 000000;  Mem[R2] = R7
0001 010 010 1 0x001; R2 = R2 + ?
0001 000 000 1 00001; R0 = R0 + 1
0010 001 000010001;   R1 = Mem[x3019] (initial)= x0005
0001 001 x01 1 11111; R1 = R? - 1
0011 001 000001111;   Mem[x3019] = R1
0000 010 000000001;   if R1 == 0, PC = x300C
0100 1 11111111000;   R7 = x300C; PC = x3004
0001 010 010 1 11111; R2 = R2 - 1
01x0 111 010 000000;  ?
1100 000111000000;    PC = R7
0000000000000000;     x3003;
0000000000000000;     x300C;
0000000000000000;     x300C;
0000000000000000;     x300C;
0000000000000000;     x300C;
0000000000000000;     x300C;
0000000000000000;
0000000000000000;
0000000000000000;
0000000000000000;
0000000000000101;     x3019
```

- 第3行中PCoffset值只有可能是0或1，而若其为0则程序直接终止，这显然不可，因此填入1；
- 填好第3行后可以观察到，第5行——第12行是一个循环，这个循环的退出是由R1是否为0控制的，而R1初始值为5，如果第9行中未知数为1，则由R5 == 0，R1会直接变为 -1，从而陷入死循环，显然不可，因此填入0，使得R1能在每次循环递减；
- 再看这个循环体，第五行中需要用LDR访问内存，因此R2值不能太大。若第6行填入1，则3次循环后内存地址便出现非法情况，故填入0；
- 最后来分析第14行的未知数，这个位置只有可能是JSR或LDR指令。若为JSR指令，则PCoffset的范围已经远远超出了可操作的地址空间，因此只能为LDR指令，填入1；
- 填入后验证整个程序，每个内存位置的数据在注释中已经标出，执行一次程序后各寄存器值均正确，故可以证明未知数填入是正确的。

修改后的程序如下：

```
0011 000 000000000
1110 010 000001110;   R2 = x300F
0101 000 000 1 00000; R0 = x0000;
0100 1 00000000001 ;  R7 = x3003 PC = x3006 goto line 5
1111 0000 00100101;   TRAP x25 Halt
0111 111 010 000000;  Mem[R2] = R7
0001 010 010 1 00001; R2 = R2 + 1
0001 000 000 1 00001; R0 = R0 + 1
0010 001 000010001;   R1 = Mem[x3019] = initial x0005
0001 001 001 1 11111; R1 = R1 - 1
0011 001 000001111;   Mem[x3019] = R1
0000 010 000000001;   if R1 == 0, goto line 13
0100 1 11111111000;   R7 = x300C; PC = x3004 goto line 5
0001 010 010 1 11111; R2 = R2 - 1
0110 111 010 000000;  R7 = Mem[R2]
1100 000111000000;    PC = R7
0000000000000000;     x300F; x3003
0000000000000000;     x300A; x300C
0000000000000000;     x300B; x300C
0000000000000000;     x300C; x300C
0000000000000000;     x300D; x300C
0000000000000000;     x300E; x300C
0000000000000000;
0000000000000000;
0000000000000000;
0000000000000000;
0000000000000101;     x3019
```

task 2

在该任务中，我们也假设起始地址为x3000，分析程序如下：

```
0010 001 000010101;   R1 = Mem[x3001 + x0015 = x3016] = x0120
0100 1 00000001000;   R7 = x3002; PC = x3002 + x0008 = x300A goto line 11
0101 010 001 1 00111; R2 = R1 mod8 aim: get the low 3bits
0001 001 010 0 00100; R1 = R2 + R4
0001 000 0xx x 11001; R0 = R? - 7;
0000 001 1xxx11011;   if(R0 > 0) goto ?
0001 000 0xx x 11001; R0 = R? - 7;
0000 100 000000001;   if(R0 < 0) goto line 10;
0001 001 001 1 11001; R1 = R1 - 7;
1111 000000100101;    TRAP x25
0101 010 010 1 00000; R2 = 0;
0101 011 011 1 00000; R3 = 0;
0101 100 100 1 00000; R4 = 0;
0001 010 010 1 00001; R2 = 1;
0001 011 011 1 01000; R3 = 8;
0101 101 011 0 00001; R5 = R3 & R1
0000 010 000000001;   if(R5 == 0) goto line 19
0001 100 010 0 00100; R4 += R2;
0001 010 010 0 00010; R2 << 1;
0001 xxx 011 0 00011; R? = R3 + R3; aim: divide 8
0000 xxx 111111010;   if(?) goto line 16;
1100 000 111 000000;  PC = R7
0000 0001 0010 0000;  x3016
```

- 首先先填入第5、7行的条件码，因为后五位已经远远超出7，故只有可能是立即数加法，填入0；
- 再填入第6行的PCoffset，若有任意一个为0，则最终结果超出了可操作的地址范围，因此均为1；
- 再对整体算法进行分析。我们看到在算法中存在一步为对8取模，因此考虑可能算法逻辑如下：

> 设需要算出n除以7的余数，我们可以令n / 8 = k …… r，则可知 n = 8k + r， 则也有n = 7k + (k + r)。因此，只需要每次保存下R1中值除以8的商k和余数r，k + r就是和R1模7同余的，因此可以再将k + r赋值给R1进行迭代。可以证明，只有当R1中值小于8的时候，这种迭代才会陷入死循环，否则k + r永远小于当前R1的值，因此这种迭代是会逐渐逼近正确解的。
>
> 当k + r 中值小于8时，我们已经找到了一个最小的能够与R1模7同余的数，若这个数小于7，自然是正确的解，但事实上它也存在恰好为7的可能，这时显然R1可以被7整除，取模结果应为0。

- 有了上述的算法思想，我们就可以分析代码了。首先我们要找到代码中**除以8**的功能模块，注意到除以8相当于右移3位，代码第11——21行中存在利用单二进制位取与运算以取数的操作，注意R2初始值为1，因此R4中值会从第1位开始构建，而R3值为8，会从R1中第4位开始取数，显然应令R3不断左移取数，因此**第20行的3个未知数应是R3的标识011**；同时，构建除以8的商过程应以R3中的1移位至最高位停止，这时只需要检查R3为0就可以结束循环，因此**第19行的3个未知数是条件码101**；
- 再来分析第5,6,7行的未知数，根据算法逻辑，第5,6行是为了检查k + r是否小于8，如果小于8则终止除以8函数的调用，因此**第5行的3个未知数是代表R1的001，第6行的3个未知数填补了除以8函数的地址，因此是111**；如果结果k + r小于8，还需检查是否为7，因此**第5行的3个未知数仍是代表R1的001。**
- 填入完毕后再验证整个程序，输出结果正确，因此填补之后的代码如下：

```
0010 001 000010101;   R1 = Mem[x3001 + x0015 = x3016] = x0120
0100 1 00000001000;   R7 = x3002; PC = x3002 + x0008 = x300A goto line 11
0101 010 001 1 00111; R2 = R1 mod8 aim: get the low 3bits
0001 001 010 0 00100; R1 = R2 + R4
0001 000 001 1 11001; R0 = R1 - 7;
0000 001 111111011;   if(R0 > 0) goto line 2
0001 000 001 1 11001; R0 = R1 - 7;
0000 100 000000001;   if(R0 < 0) goto line 10;
0001 001 001 1 11001; R1 = R1 - 7;
1111 000000100101;    TRAP x25
0101 010 010 1 00000; R2 = 0;
0101 011 011 1 00000; R3 = 0;
0101 100 100 1 00000; R4 = 0;
0001 010 010 1 00001; R2 = 1;
0001 011 011 1 01000; R3 = 8;
0101 101 011 0 00001; R5 = R3 & R1
0000 010 000000001;   if(R5 == 0) goto line 19
0001 100 010 0 00100; R4 += R2;
0001 010 010 0 00010; R2 << 1;
0001 011 011 0 00011; R3 << 1; aim: divide 8
0000 101 111111010;   if(R3 != 0) goto line 16;
1100 000 111 000000;  PC = R7
0000 0001 0010 0000;  x3016
```

实验总结

本次实验是对代码空缺的填补，通过这两段代码，我加深了对算法的低层实现的理解，而在填补代码的过程中，我也总结出了以下几点经验：

- 对于他人的程序，我们首先要进行细节的理解，特别是对整段的代码进行一个良好的分割，从而帮助我们更好地读懂代码；
- 在浏览过程中，要通过完整的代码段判断可能的算法，再通过其中某些运算的实际意义（比如取位、左移）思考算法的实际实现途径；
- 填补之后，要先逐步在脑海中运行一次，可以不完整运行，但要保证每次循环的结果都能向自己希望的方向发展，最后再使用计算机进行一次运算，最终判断填补是否正确。