Lab4 Reveal Yourself

2021.12 赖永凡　PB20061210

实验要求

修复task1.txt和task2.txt中的代码

Task1

修复要求

代码执行前除PC外所有寄存器储存值为零，执行后寄存器状态如下：

```
R0 = 5, R1 = 0, R2 = 300f, R3 = 0
R4 = 0, R5 = 0, R6 = 0, R7 = 3003
```

修复后的代码

```
0011 0000 0000 0000

1110 010 000001110      ;LEA R2,x300F   x3000
0101 000 000 1 00000    ;AND R0,R0,0    x3001
0100 1 00000000001      ;JSR x3004      x3002
1111 000000100101       ;HALT           x3003
0111 111 010 000000     ;STR R7,R2,#0   x3004
0001 010 010 1 00001    ;ADD R2,R2,#1   x3005
0001 000 000 1 00001    ;ADD R0,R0,#1   x3006
0010 001 000010001      ;LD R1,x3019    x3007
0001 001 001 1 11111    ;ADD R1,R1,#-1  x3008
0011 001 000001111      ;ST R1,x3019    x3009
0000 010 000000001      ;BRz x300C      x300A
0100 1 11111111000      ;JSR x3004      x300B
0001 010 010 1 11111    ;ADD R2,R2,#-1  x300C
0110 111 010 000000     ;LDR R7,R2+#0   x300D
1100 000 111 000000     ;RET            x300E
0000000000000000        ;#0             x300F
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000101        ;#5             x3019
```

这里为叙述方便，假设第一条指令储存在x3000地址

下图是在**HALT**指令前加上断点后，程序执行到断点时的寄存器状态：

Registers

R0	x0005	5	
R1	x0000	0	
R2	x300F	12303	
R3	x0000	0	
R4	x0000	0	
R5	x0000	0	
R6	x0000	0	
R7	x3003	12291	
PSR	x8001	-32767	CC: P
PC	x3003	12291	
MCR	x0000	0	

与所要求的寄存器状态一致，代表着修复工作完成。

简要分析

此代码x3004到x300E段构成了一个递归结构，当R1为零时是递归出口。由于R1同x3019处的值紧密地关联在了一起，而本代码里已对x3019处赋初始值#5，所以R1初始值为5。

在知晓代码使用了递归这一数据结构后，就不难看出x3004,x3005和x300C,x300D前后各两行代码的作用便是维护递归栈的使用，具体而言，R2作为栈顶指针，储存R7这个**return linkage**。而当程序结束，递归栈不再使用时，R7储存的便是第一次调用子函数时PC+1的值，即x3003，而R2就指向栈底，即x300F。

进一步分析，不难看出子函数的作用即是让R1递减，让R0来计数，因此在执行结束后，R1递减至零，而R5的值为5。

Task2

修复要求

第二部分的代码用于计算$N \mod 7, (N \in \mathbb{Z}^+)$。

提示：程序中用到了“除以8”这种操作。

修复前的代码

```
0010 001 000010101      ;x3000 LD R1,x3016
0100 1 00000001000      ;x3001 JSR x300A
0101 010 001 1 00111    ;x3002 AND R2,R1,#7
0001 001 010 0 00 100   ;x3003 ADD R1,R2,R4
0001 000 0xx x 11001    ;x3004 ADD R0,Rx,#-7
0000 001 1xxx11011      ;x3005 BRp x3001
0001 000 0xx x 11001    ;x3006 ADD R0,Rx,#-7
0000 100 000000001      ;x3007 BRn x3009
0001 001 001 1 11001    ;x3008 ADD R1,R1,#-7
1111 000000100101       ;x3009 HALT
0101 010 010 1 00000    ;x300A AND R2,R2,#0
0101 011 011 1 00000    ;x300B AND R3,R3,#0
0101 100 100 1 00000    ;x300C AND R4,R4,#0
0001 010 010 1 00001    ;x300D ADD R2,R2,#1
```

```
0001 011 011 1 01000    ;x300E ADD R3,R3,#8
0101 101 011 0 00 001   ;x300F AND R5,R3,R1
0000 010 000000001      ;x3010 BRz x3012
0001 100 010 0 00 100   ;x3011 ADD R4,R2,R4
0001 010 010 0 00 010   ;x3012 ADD R2,R2,R2
0001 xxx 011 0 00 011   ;x3013 ADD Rx,R3,R3
0000 xxx 111111010      ;x3014 BR? x300F
1100 000 111 000000     ;x3015 RET
0000000100100000        ;x3016 x0120 #288
```

为了更好地说明修复工作以及阐述算法，这里先给出了尚未修复但经过初步翻译的代码。同样，假定程序从x3000开始。
初步浏览代码，可以发现需要修复的部分一共有5处，分别对应**x3004,x3005, x3006,x3013,x3014**处，以下将分析修复的思路。

x3005

此处的offset已经被限定为一个负数，如果前面的**x**中有一个不是0，那么程序将跳转到x3000之前的位置，显然不符合逻辑，故只能是**111**，且BR语句执行后，程序将跳转至x3001处。

x3004 & x3006

这两行程序具有高度的相似性，因此大概率是相同的。首先，通过对比机械码的标准格式可知，两处的bit[5]一定为**1**，即语句的结构为**ADD R0,Rx,#-7**。

在修复了x3005处的代码后，我们已经可以看出算法的大致逻辑，即：先执行依次位于x300A处的子程序，然后对R1进行一些操作后再对某个寄存器进行判断，判断的结果为正数：重新调用子程序；负数：直接结束；零：令R1-7后结束。

鉴于程序一开始将R1赋予了**#288**这个看上去十分奇怪的值，我们可以合理推测这是就是待求余数的值。结合程序的作用是求7的余数这一操作，以及上述判断历程，不难得出最终结果也储存在R1中这一结论，于是Rx也自然而然的表示R1。因此，这一部分的逻辑为：先对**R1减7**，存在中间寄存器R0中，然后判断正负：若为负，则说明此时R1中的值小于7，**就是所要求的余数的值**；若为零，则说明此时R1中存的值为7，于是将R1中的值减去7就得到了**所要的余数值——0**；若为正，则说明R1中的值大于7，**需要重新调用子函数。**

当然，目前的一切都基于一个假设——**两次判断之间，R1中的值对7同余。**

x3013 & x3014

不难看出，x3013作用是将R3中的值左移一位，存储结果的寄存器为R3不言而喻，否则，前面对R2和R4的操作将毫无意义。不难看出，R3的作用是检验R1对应位是否为1，若不为1则将R4与此时的R2相加，注意到这里R2同样也在左移，但是R2里为1的位始终在R3里为1的那位的左侧三位处，稍作思考不难发现，**当R3遍历比较完R1的所有16bits后，R4中的值实际为R1中的值右移3位，即**$R4 = [R1 \div 8]$。因此x3014处的条件码自然为**101。**

至此，我们已将程序全部修复完毕。

修复后的代码 & 简要分析

```
0011 0000 0000 0000     ;.ORIG x3000
0010 001 000010101      ;x3000 LD R1,x3016
0100 1 00000001000      ;x3001 JSR x300A
0101 010 001 1 00111    ;x3002 AND R2,R1,#7     R1 mod 8 -> R2
0001 001 010 0 00 100   ;x3003 ADD R1,R2,R4     R1 = (R1 mod 8) + R1 / 8
0001 000 001 1 11001    ;x3004 ADD R0,R1,#-7
```

```
0000 001 111111011      ;x3005 BRp x3001
0001 000 001 1 11001    ;x3006 ADD R0,R1,#-7
0000 100 000000001      ;x3007 BRn x3009
0001 001 001 1 11001    ;x3008 ADD R1,R1,#-7
1111 000000100101       ;x3009 HALT
0101 010 010 1 00000    ;x300A AND R2,R2,#0
0101 011 011 1 00000    ;x300B AND R3,R3,#0
0101 100 100 1 00000    ;x300C AND R4,R4,#0
0001 010 010 1 00001    ;x300D ADD R2,R2,#1
0001 011 011 1 01000    ;x300E ADD R3,R3,#8
0101 101 011 0 00 001   ;x300F AND R5,R3,R1
0000 010 000000001      ;x3010 BRz x3012
0001 100 010 0 00 100   ;x3011 ADD R4,R2,R4
0001 010 010 0 00 010   ;x3012 ADD R2,R2,R2
0001 011 011 0 00 011   ;x3013 ADD R3,R3,R3
0000 101 111111010      ;x3014 BRnp x300F
1100 000 111 000000     ;x3015 RET
0000 0001 0010 0000     ;x3016 x0120 #288
```

至此，程序的算法已经柳暗花明：

$$n\text{和}[\frac{n}{8}]+n \mod 8\text{对}7\text{同余}$$

因此，程序不断地计算$[R1/8]+R1 \mod 8$，将其代替原先的$R1$，直到其小于等于7为止。

如果一般的算法是通过不断地将原来的值减7，直到其介于0到7之间的话，那么本快速算法将时间复杂度从$O(n)$量级减少到了$O(logn)$量级。

最后附上程序执行结束后的寄存器状态图（已在HALT前加上断点）

Registers

R0	xFFFA	-6	
R1	x0001	1	
R2	x0000	0	
R3	x0000	0	
R4	x0001	1	
R5	x0000	0	
R6	x0000	0	
R7	x3002	12290	
PSR	x8004	-32764	CC: N
PC	x3009	12297	
MCR	x0000	0	

可见得出的储存于R1中的结果为1，这与 $288 = 7 \times 41 + 1$ 一致，进一步验证了修复后程序的正确性。