lab4

实验目的

1. 使用LC-3实现九连环游戏的解法。
2. 体会递归程序的编写技巧。
3. 巩固函数调用的知识。

实验方法

递归模型

文档中给出了`REMOVE`函数的递归式，而`PUT`函数的递归式为

$$P(0) = do\ nothing,\ P(1) = put\ the\ 1^{st}\ ring$$
$$P(i) = P(i-1) + R(i-2) + put\ the\ i^{th}\ ring + P(i-2)$$

其中前两个操作是为$put\ the\ i^{th}\ ring$做准备，当前三个操作做完后，第i个和第$i-1$个都在板上，因此只需要递归地放置前$i-2$个。

主函数

用`R1`存储环的状态，`R0`作为调用`REMOVE`和`PUT`函数的参数`i`，`R2`记录结果保存的地址（每保存一次结果都要递增它），`R6`作为栈顶指针。主过程就是调用`REMOVE(n)`，代码如下：

```
LD R6, TOP_OF_STACK     ; R6 <- top of stack
LD R2, ADDR_N           ; R2 <- x3100, the address to store results
AND R1, R1, #0          ; R1 <- state, init to 0
LDI R0, ADDR_N          ; R0 <- M[3100] = n
JSR REMOVE              ; call REMOVE(R0: n)
HALT
```

结果保存

每次保存结果时，先递增`R2`，再用`STR`指令将`R1`存到`R2`所指的位置。代码如下：

```
ADD R2, R2, #1          ; increment result address
STR R1, R2, #0          ; store result
```

REMOVE函数

先判断两个特殊情况：$i = 0$时什么也不做，直接返回；$i = 1$时直接将`R0`最低位由0变成1（也就是直接将`R0`加1），保存结果，然后返回。代码如下：

```
BRz REMOVE_END          ; i = 0, do nothing
ADD R3, R0, #-1         ; R3 <- R0 - 1
BRp REMOVE_START        ; i > 1
ADD R1, R1, #1          ; i = 1, remove the 1st ring (flip R1[1] from 0 to 1)
ADD R2, R2, #1          ; increment result address
STR R1, R2, #0          ; store result
BR REMOVE_END           ; RET
```

> *由于每次调用REMOVE的前一步操作基本都是修改R0的值，因此这里第一行可以直接调用`BRz`命令.*

然后将`R7`入栈保存，因为后续递归调用时会修改`R7`。代码如下：

```
REMOVE_START    ADD R6, R6, #-1
                STR R7, R6, #0          ; save R7
```

接下来根据递归式，调用`REMOVE(i-2)`，即先将`R0`减2，再调用`REMOVE`函数。代码如下：

```
ADD R0, R0, #-2         ; R0 <- i - 2
JSR REMOVE              ; call REMOVE(R0: i - 2)
```

接下来根据递归式，要将`R1`的第i位由0变成1，方法是首先通过左移设置一个掩码`R3 = 1 << (i - 1)`，将`R1`与这个掩码相加即可，完成后要保存结果。代码如下：

```
                ADD R4, R0, #2          ; R4 <- i
                AND R3, R3, #0
                ADD R3, R3, #1          ; R3 <- 1
SET_REMOVE_MASK ADD R4, R4, #-1         ; R4 <- R4 - 1
                BRz REMOVE_i            ; now R3 = (1 << i)
                ADD R3, R3, R3          ; R3 <- R3 << 1
                BR SET_REMOVE_MASK
REMOVE_i        ADD R1, R1, R3          ; remove the i-th ring (flip R1[i] from 0 to
1)
                ADD R2, R2, #1          ; increment result address
                STR R1, R2, #0          ; store result
```

接下来根据递归式，调用`PUT(i-2)`。注意此时`R0`的值已为$i-2$，故先给它加0。代码如下：

```
ADD R0, R0, #0          ; R0 <- i - 2
JSR PUT                 ; call PUT(R0: i - 2)
```

接下来根据递归式，调用`REMOVE(i-1)`，要先给`R0`加1变成$i-1$。代码如下：

```
ADD R0, R0, #1          ; R0 <- i - 1
JSR REMOVE              ; call REMOVE(R0: i - 1)
```

最后将`R0`的值恢复为i，同时从栈中恢复先前保存的`R7`的值，返回。代码如下：

```
                ADD R0, R0, #1          ; R0 <- i
                LDR R7, R6, #0          ; restore R7
                ADD R6, R6, #1
REMOVE_END      RET
```

PUT函数

代码框架与`REMOVE`函数几乎相同，不再赘述。唯一的区别是在`remove`第i位时，要把`R1`的第i位由1变成0，因此同上设置好掩码后，将掩码反转，就得到一个除第i位是0外其余位都是1的掩码。将这个掩码与`R1`按位与，由于一个1位的数与1相与仍是它自身，因此`R1`除第i位外其余位都不变，只有第i位和0相与变成了0。代码如下：

```
NOT R3, R3
AND R1, R1, R3          ; put the i-th ring (flip R1[i] from 1 to 0)
```

完整代码

```
                .ORIG x3000
                LD R6, TOP_OF_STACK     ; R6 <- top of stack
                LD R2, ADDR_N           ; R2 <- x3100, the address to store results
                AND R1, R1, #0          ; R1 <- state, init to 0
                LDI R0, ADDR_N          ; R0 <- M[3100] = n
                JSR REMOVE              ; call REMOVE(R0: n)
                HALT


REMOVE          BRz REMOVE_END          ; i = 0, do nothing
                ADD R3, R0, #-1         ; R3 <- R0 - 1
                BRp REMOVE_START
                ADD R1, R1, #1          ; i = 1, remove the 1st ring (flip R1[1] from
0 to 1)
                ADD R2, R2, #1          ; increment result address
                STR R1, R2, #0          ; store result
                BR REMOVE_END           ; RET

REMOVE_START    ADD R6, R6, #-1
                STR R7, R6, #0          ; save R7

                ADD R0, R0, #-2         ; R0 <- i - 2
```

```
                JSR REMOVE              ; call REMOVE(R0: i - 2)

                ADD R4, R0, #2          ; R4 <- i
                AND R3, R3, #0
                ADD R3, R3, #1          ; R3 <- 1
SET_REMOVE_MASK ADD R4, R4, #-1         ; R4 <- R4 - 1
                BRz REMOVE_i            ; now R3 = (1 << i)
                ADD R3, R3, R3          ; R3 <- R3 << 1
                BR SET_REMOVE_MASK
REMOVE_i        ADD R1, R1, R3          ; remove the i-th ring (flip R1[i] from 0 to
1)
                ADD R2, R2, #1          ; increment result address
                STR R1, R2, #0          ; store result

                ADD R0, R0, #0          ; R0 <- i - 2
                JSR PUT                 ; call PUT(R0: i - 2)

                ADD R0, R0, #1          ; R0 <- i - 1
                JSR REMOVE              ; call REMOVE(R0: i - 1)

                ADD R0, R0, #1          ; R0 <- i
                LDR R7, R6, #0          ; restore R7
                ADD R6, R6, #1
REMOVE_END      RET


PUT             BRz PUT_END             ; i = 0, do nothing
                ADD R3, R0, #-1         ; R3 <- R0 - 1
                BRp PUT_START
                ADD R1, R1, #-1         ; i = 1, put the 1st ring (flip R1[1] from 1
to 0)
                ADD R2, R2, #1          ; increment result address
                STR R1, R2, #0          ; store result
                BR PUT_END

PUT_START       ADD R6, R6, #-1
                STR R7, R6, #0          ; save R7

                ADD R0, R0, #-1         ; R0 <- i - 1
                JSR PUT                 ; call PUT(R0: i - 1)

                ADD R0, R0, #-1         ; R0 <- i - 2
                JSR REMOVE              ; call REMOVE(R0: i - 2)

                ADD R4, R0, #2          ; R4 <- i
                AND R3, R3, #0
```

```
                ADD R3, R3, #1          ; R3 <- 1
SET_PUT_MASK    ADD R4, R4, #-1         ; R4 <- R4 - 1
                BRz PUT_i               ; now R3 = 1 << (i - 1)
                ADD R3, R3, R3          ; R3 <- R3 << 1
                BR SET_PUT_MASK
PUT_i           NOT R3, R3
                AND R1, R1, R3          ; put the i-th ring (flip R1[i] from 1 to 0)
                ADD R2, R2, #1          ; increment result address
                STR R1, R2, #0          ; store result

                ADD R0, R0, #0
                JSR PUT                 ; call PUT(R0: i - 2)

                ADD R0, R0, #2          ; R0 <- i
                LDR R7, R6, #0          ; restore R7
                ADD R6, R6, #1
PUT_END         RET


                TOP_OF_STACK .FILL xFDFF
                ADDR_N       .FILL x3100
                .END
```

实验中遇到的BUGS

1. 一开始在纠结本程序中递归调用的函数到底要保存哪些寄存器到栈上。后来发现`R1`可以作为全局变量；`R0`在函数内部无非是在i, $i-1$, $i-2$三种取值中变动，最后恢复为i即可。于是只需要保存`R7`即可。
2. `REMOVE`函数第三步调用`PUT(i-2)`时，`R0`的值已经是$i-2$了，原本就偷懒直接调用了，发现结果不对。后来想起自己对`REMOVE`函数和`PUT`函数的设置是第一行就要用到`R0`的条件码，因此要先给`R0`加0以设置条件码。
3. 原先选择的栈的起始位置为`x4000`，好奇实验文档为什么说n要限制为不超过12，于是试了$n=13$的情况，发现报错`Access violation`。检查内存发现原因在于从`x3100`开始记录的操作结果与向上增长的栈重叠了，如下图，这将导致`RET`到`x1A8C`的位置，而这里是Privileged Memory。

		Memory
x3FEC	x1AA2	6818
x3FED	x1AA3	6819
x3FEE	x1AAB	6827
x3FEF	x1AAA	6826
x3FF0	x1AA8	6824
x3FF1	x1AA9	6825
x3FF2	x1AAD	6829
x3FF3	x1AAC	6828
x3FF4	x1AAE	6830
x3FF5	x1AAF	6831
x3FF6	x1A8F	6799
x3FF7	x1A8E	6798
x3FF8	x1A8C	6796
x3FF9	x3034	12340
x3FFA	x3044	12356
x3FFB	x3021	12321
x3FFC	x3036	12342

后来尝试将栈的起始位置改为`xFDFF`，发现可以正常执行$n = 13 \sim 16$的情况。

实验结果

选取$n = 0 \sim 12$， 在自动评测机上的评测结果如下：

汇编评测

13 / 13 个通过测试用例

- 平均指令数: 8692
- 通过 12, 指令数: 56625, 输出: 2,3,11,10,8,9,13,12,14,15,47,46,44,45,41,40,42,43,35,34,32,33,37,36,38,39,55,54,52,53,49,48,50,51,59,58,56,57,61,60,62,63,191,190,188,189,185,184,186,187,179,178,176,177,181,180,182,183,167,166,164,165,161,160,162,163,171,170,168,169,173,172,174,175,143,142,140,141,137,136,138,139,131,130,128,129,133,132,134,135,151,150,148,149,145,144,146,147,155,154,152,153,157,156,158,159,223,222,220,221,217,216,218,219,211,210,208,209,213,212,214,215,199,198,196,197,193,192,194,195,203,202,2