Assembly instruction for program 1

```
set #2          //set flag for mov
lrm r1, #7      //r1 = 7
set #0          //set flag for left shift
lrm r1, #3      //r1 = r1 << 3;          7* 8 = 56
set #2          //set flag for mov
lrm r2, #5      //r2 = 5
lsa r1, r2      //r1 = r1 + r2;          r1 = 61
lrm r0, #0      //r0 = 0
lsa r0, r1      //r0 = r0 + r1           r0 = 61

set #0          //set flag for load
lsa r4, #0      //r4 = mem[61]           r4 = N
lsa r2, #1      //r2 = mem[62]           r2 = tap
lsa r3, #2      //r3 = mem[63]           r3 = initial state
set #1          //set flag for setTap
bor r2          // setTap
set #3          //set flag for lfsr
lrm r3          // setLFSR
set #2          //set flag for mov and add
lrm r1, #3      //r1 = 3
lsa r0, r1      //r0 = r0 + r1           r0 = #64

lrm r3, #0      //r3 = 0                 for(int i = 0; i < N; i++)
lrm r6, #0      //r6 = 0
FOR1:
set #2          //set flag for add

bor END1        //if r3 >= r4, go to DONE1
lrm r1, #2      //r1 = 2
set #0          //set flag for left shift
lrm r1, #4      //r1 = r1 << 4           r1 = 32 = 0x20
```

```
snxa r1, r1          //r1 = space ^ lfsr
rxor r2, r1          //r2 = reduction xor on r1
lrm r2, #7           //r2 = r2 << 7              r2 = parity bit
bor r1, r2           //r1 = r1 | r2
set #1               //set flag for store
lsa r1, #0           //mem[64 + i] = r1;
snxa r0, r0          //update lfsr
set #2               //set flag for mov
lrm r2, #1           //r2 = 1
lsa r0, r2           //r0 = r0 + r2              r0 += 1
lsa r3, r2           //r3++                      i++
lrm r6, #0           //R6 = 0
lsa r6, r3           //R6 = r3
lrm r3, #0           //Mov r3, #0
set #3               //set flag for bne
bor FOR1             //check condition if r3!=r4 continue loop
END1:                //end of FOR1
set #2               //set flag for mov
lrm r3, #0           //r3 = 0                    for(int i = 0; i < #size; i++)
lrm r6, #0           //r6 = 0
lrm r4, #3           //r4 = 3
lrm r2, #1           //r2 = 1
set #0               //set flag for left shift
lrm r4, #4           //r4 = #3 << 4              r4 = 48
set #2               //set flag for add
lsa r4, r2           //r4 += 1                   r4 = 49, r3 = i
FOR2:                // .
set #2
lsa r3, r6           //r3 = r6
bor END2             //if i > 49, done
lrm r5, #0           //r5 = 0                    store r0 in r5
lsa r5, r0           //r5 = r0
lrm r0, #0           //r0 = 0
lsa r0, r3           //r0 = i
```

```
set #0          //set flag for load
lsa r1, #0      //r1 = mem[i]               r1 = data
snxa r1, r1     //r1 = r1 ^ lfsr            r1 = msb
rxor r2, r1     //r2 = reduction xor on r1
lrm r2, #7      //r2 = r2 << 7
bor r1, r2      //r1 = r1 | r2              r1 = msb with parity
set #2
lrm r0, #0      //r0 = 0
lsa r0, r5      //r0 = r5

set #1          //set flag for store
lsa r1, #0      //mem[64 + i] = r1;
snxa r0, r0     //update lfsr
set #2          //set flag for mov
lrm r2, #1      //r2 = 1
lsa r0, r2      //r0 = r0 + r2              r0 += 1
lsa r3, r2      //r3++                      i++
lrm r6, #0      //R6 = 0
lsa r6, r3      //R6 = r3
lrm r3, #0      //r3 = 0
set #3
bor FOR2        //branch back to for2
END2:           //end of FOR2

DONE            //end of the program
```

Assembly instruction for program 2

```
set #2
lrm r1, #0      //currentTap = r1 = 0
lrm r0, #4      // r0 = #4
lrm r6, #4      // r6 = #4
lrm r2, #3      // r2 = #3
set #0
lrm r0, #4      //r0 << 4
lrm r6, #3      //r6 << 3, r6 = 0x20
set #2
lsa r0, r2      // r0 += 3
lsa r0, r2      // r0 += 3
lsa r0, r2      // r0 += 3
set #0
lsa r4, #0      // r4 = mem[73]

set #2
snxa r4, r6     // r4 = mem[73] ^ 0x20

ISRIGHTTAP:
set #2
rxor r2, r1     // getTap check if r1 is right tap, store result into r2
lrm r3, #0      // mov r3, #0
lsa r3, r2      // add r3, r2
lrm r5, #1      // mov r5, #1
lsa r1, r5      // r1++
set #3
bor ISRIGHTTAP  // if r3 != mem[73], then “incorrect tap”, loop again

set #2
lrm r5, #2      //r5 = 2
lrm r0, #4      // r0 = 4
lrm r4, #1      // mov r4, #1
set #3
lsa r1, r4      // r1 --
// incremented 1 extra time, so sub 1
// final tap value in r2
// final tap number in r1

set #0                  //set flag for left shift
lrm r5, #4              //r5 = r5 << 4              r5 = 32 = 0x20
lrm r0, #4              //r0 = r0 << 4              r0 = 64
lsa r4, #0      // r4= mem[64]
```

```
set #2
snxa r5, r4 // xor r5 = r5^r4
set #3
lrm r5 // currentLFSR = r5, the initial LFSR

set #1
bor r1 // setTap r1

set #2
lrm r2, #0 // set counter = r2 = 0
lrm r4, #4 // r4 = 4
lrm r7, #1 // r7 = 1
set #0
lrm r4, #4 // r4 << 4 = 64

WHILECOUNTER:
set #2          //set flag for mov
lrm r3, #0 // r3 = 0
lsa r3, r2 // r3 = r2

bor ENDP2 // if r3 >= 6, branch to end
set #2
lrm r0, #0 // r0 = 0
lsa r0, r4 // r0 = r4 = 64
lsa r0, r2 // r0 = r4 + r2 (counter + 64)
set #0
lsa r3, #0 // r3 = mem[counter+64], getting the encoded message
snxa r3, r3 // decodedMessage = r3 = memloc ^ currentLFSR, decode the message
set #2
lrm r0, #0 // r0 = 0
lsa r0, r2 // r0 = r2 = counter
set #1
lsa r3, #0 // store r3, mem[counter], store the decoded message to 0 - 63
snxa r5, r5 // get next lfsr, store in r5
set #2
lsa r2, r7 // r2++
lrm r3, #0 // mov r3, #0
Set #3          //set flag for BNE
bor WHILECOUNTER // branch always

ENDP2:
DONE
```

Assembly instruction for program 3

```
set #2
lrm r1, #0      //currentTap = r1 = 0
lrm r0, #4      // r0 = #4
lrm r6, #4      // r6 = #4
lrm r2, #3      // r2 = #3
set #0
lrm r0, #4      //r0 << 4
lrm r6, #3      //r6 << 3, r6 = 0x20
set #2
lsa r0, r2      // r0 += 3
lsa r0, r2      // r0 += 3
lsa r0, r2      // r0 += 3
set #0
lsa r4, #0      // r4 = mem[73]

set #2
snxa r4, r6     // r4 = mem[73] ^ 0x20

ISRIGHTTAP:
set #2
rxor r2, r1     // getTap check if r1 is right tap, store result into r2
lrm r3, #0      // mov r3, #0
lsa r3, r2      // add r3, r2
lrm r5, #1      // mov r5, #1
lsa r1, r5      // r1++
set #3
bor ISRIGHTTAP  // if r3 != mem[73], then "incorrect tap", loop again

set #2
lrm r5, #2      //r5 = 2
lrm r0, #4      // r0 = 4
lrm r4, #1      // mov r4, #1
set #3
lsa r1, r4      // r1 --
// incremented 1 extra time, so sub 1
// final tap value in r2
// final tap number in r1

set #0                  //set flag for left shift
lrm r5, #4              //r5 = r5 << 4              r5 = 32 = 0x20
lrm r0, #4              //r0 = r0 << 4                  r0 = 64
lsa r4, #0      // r4= mem[64]
set #2
snxa r5, r4 // xor r5 = r5^r4
set #3
lrm r5 // currentLFSR = r5, the initial LFSR

set #1
bor r1 // setTap r1



set #2
lrm r2, #0 // set counter = r2 = 0
lrm r4, #4 // r4 = 4
lrm r7, #1 // r7 = 1
set #0
lrm r4, #4 // r4 << 4 = 64
set #2
lrm r6, #0      //r6 = 0
lsa r6, r4      //r6 += r4;     r6 = r4
lrm r5, #4      //r5 = 4
set #0          //set flag for lsl
lrm r5, #5      //r5 = r5 << 5          r5 = 0x80


WHILECOUNTER:
set #2          //set flag for mov
lrm r3, #0      // r3 = 0
lsa r3, r2      // r3 = r2

bor ENDP2 // if r3 >= 64, branch to end
set #2
lrm r0, #0 // r0 = 0
lsa r0, r4 // r0 = r4 = 64
lsa r0, r2 // r0 = r4 + r2 (counter + 64)
set #0
lsa r3, #0 // r3 = mem[counter+64], getting the encoded message
rxor r4, r3     //r4 = ^r3, get the reduction xor of the encode message
set #1          //set flag for right shift
lrm r3, #7      //r3 = r3 >> 7, get encode[7]
rxor GOOD       //if r3 == r4, branch to good
set #2          //set flag for mov
lrm r3, #0      //r3 = 0
lsa r3, r5      //r3 += r5              r3 = r5 = 0x80
```

```
STORE:
set #2
lrm r0, #0      // r0 = 0
lsa r0, r2      // r0 = r2 = counter
set #1
lsa r3, #0      // store r3, mem[counter], store the decoded message to 0 - 63
snxa r5, r5     // get next lfsr, store in r5
set #2
lsa r2, r7      // r2++
lrm r3, #0      // mov r3, #0
lrm r4, #0      //r4 = 0
lsa r4, r6      //r4 = r6
set #3          //set flag for BNE
bor WHILECOUNTER // branch always
ENDP2:
set #2
lrm r3, #0      // r3 = 0
lrm r4, #0      // r4 = 0
bor AFTERALEXIS // branch to AFTERALEXIS (BGE, always branch)
GOOD:
set #0          //set flag for load
lsa r3, #0      //r3 = mem[counter+64], getting the encoded message
snxa r3, r3     //decodedMessage = r3 = memloc ^ currentLFSR, decode the message
set #2
lrm r4, #0      // r4 = #0
lsa r4, r3      // r4 = r3
set #1
rxor STORE      // if r4 == r3, branch STORE (always branch)

AFTERALEXIS:

set #2
lrm r5, #0 // numspaces = r5 = 0
lrm r4, #4 // r4 = #4
set #0
lrm r4, #3 // r4 << 3 = 0x20

COUNTSPACES:
set #2
lrm r0, #0 // set #2, lrm r0 = #0
lsa r0, r5 // r0 = r5 = numspaces
set #0
lsa r3, #0 // r3 = mem[numspaces]
set #2
```

```
lrm r7, #1 // r7 == #1
lsa r5, r7 // r5++
set #1
rxor COUNTSPACES // BEQ

set #2
lrm r2, #0 // counter = r2 = 0
set #3
lsa r5, r7 // sub r5--

SHIFTOVERSPACES:
set #2
lrm r3, #0 // r3 = 0
lsa r3, r2 // r3 = r2 = counter
lrm r0, #0 // r0 = 0
lsa r0, r5 // r0 = r5 = numspaces
lrm r4, #4 // r4 = 4
set #0
lrm r4, #4 // r4 << 4 = 64
set #3
lsa r4, r0 // r4 = 64 - r0 (numspaces)
set #2
bor SHIFTNEXT // if counter >= numspaces, branch
lrm r0, #0 // r0 = 0
lsa r0, r2 // r0 = counter = r2
lsa r0, r5 // r0 = counter + numspaces
set #0
lsa r1, #0 // r1 = mem[counter+spaces]
set #2
lrm r0, #0 // r0 = 0
lsa r0, r2 // r0 = counter
set #1
lsa r1, #0 // mem[counter] = r1

set #2
lsa r2, r7 // counter = r2++

lrm r3, #0 // r3 = 0
lrm r4, #0 // r4 = 0
bor SHIFTOVERSPACES // always branch (set #2, BGE)

SHIFTNEXT:
set #2
lrm r3, #0 // r3 = 0
```

```
lsa r3, r2 // r3 = r2 = counter
lrm r4, #4 // r4 = 4
set #0
lrm r4, #4 // r4 << 4 = 64
set #2
bor FINALLYY // branch (set #2, BGE)
lrm r0, #0 // r0 = 0
lsa r0, r2 // r0 = r2 = counter
lrm r5, #4 // r5 = #4
set #0
lrm r5, #3 // r5 << 3 = 0x20
set #1
lsa r5, #0 // mem[count] = r5
set #2
lsa r2, r7 // counter = r2++
lrm r3, #0 // r3 = 0
lrm r4, #0 // r4 = 0
bor SHIFTNEXT // always branch (set #2, bor)

FINALLYY:
DONE
```