Answer 5

A1

`.END` 不是一条指令，并不会显式地存放在内存中，只是在汇编语言中标识在哪里停止汇编的一个标识符，而相比起来，`HALT` 是一条指令，它会存放在内存中，当 LC-3 执行到这条命令时，会停机.

A2

队列的定义特征主要包括以下几点：

1. 先进先出 (FIFO) 原则：队列中的元素从队尾 (rear) 进入，从队头 (front) 出来，最先加入队列的元素将是最先被移除的。
2. 线性结构：队列是一种线性结构，元素之间是一对一的关系，每个元素有且仅有一个前驱和一个后继（除了队头和队尾的特殊元素）。
3. 操作限定：队列的操作主要限定在队头和队尾进行，包括入队 (enqueue) 操作在队尾添加元素，出队 (dequeue) 操作在队头移除元素。

A3

程序中 A 和 B 处存放的值会被当做命令执行，而 `A` 处的 OPCODE 为 1101，是保留的指令，因此会出错；要修正，只需将 `A` 和 `B` 移到 `HALT` 之后即可.

> 如果只将 `A` 移到后面，其实 `B` 处的指令也会报错，因为它是 `STI R7, xEF`，初始化时 x30F0 处的值为 x0000，位于 privilege 区域，不可写，会报错 Access Violation.

A4

没有问题，因为每个模块会有自己的符号表，没有 `.EXTERNAL` 声明时，只会在自己模块的符号表内查找 `AGAIN` 的地址，不会出现冲突.

A5

1. x0FFF.
2. Yes.
3. Infinite loop. x0FFF interpreted as "BRnzp #-1".

Root cause: LC3 stores data and instructions together, so data might be interpreted as instructions.

Solution: Store data and instructions in separate. (Instruction memory + Data memory)

Comment: Your program will be very tricky if you utilize this behavior of LC3. If you use it well, you might create something amazing; However, it is most likely to cause more trouble than it's worth. However, do note that this may give rise to security issues. In this case, by controlling input data at DATA, one can execute a line of ANY assembly code!

A6

- `.FILL` 用于存放一个 16 位的数据在它的地址。例如 `.FILL x1234` 将 `x1234` 存放到内存中
- `.BLKW` 用于预留多个字。例如 `.BLKW #3` 将在这个位置留下 3 个字，以便后续存放数据，被预留的字在内存中默认为 `x0000`
- `.STRINGZ` 用于在连续的内存中存放字符串。例如 `.STRINGZ "Hello"` 将字符串 中的字符 ASCII 码存放到连续内存中，最后以 `x0000` 结尾

Pseudo-op	可自定义预留值	可占用多个字
`.FILL`	是	否
`.BLKW`	否	是
`.STRINGZ`	是	是

A7

因为 (d) 之后为无条件跳转，因此此时应该是已经判断当前还不相等，且已经将 `A` 加一，将 `B` 减一后的了，而最初就应该判断相等，且 (b) 为将 `R2` 和 `R1` 相加，猜测 (a) 和 (b) 为取 `R0` 负数补码存到 `R2` 中，(c) 为如果相等则直接跳转到存储的指令，否则 (d) 要将 `R0` 加一.

(a) `NOT R2, R1`
(b) `ADD R2, R2, #1`
(c) `BRz DONE`
(d) `ADD R0, R0, #1`

A8

(a) 2
(b) z
(c) R3, R3
(d) R2, R2, #-1

```
    .ORIG x3000
    LD R0, INP      ; Target
    ; Initialize
    AND R1, R1, #0  ; Result
    ADD R2, R1, #15 ; Loop var i
    ADD R3, R1, #2  ; 2^(16 - i), for AND (Source mask)
    ADD R4, R1, #1  ; 2^(15 - i), for ADD (Dest mask)
    AND R5, R5, #0  ; Temp result
    ; Main Loop
L   AND R5, R3, R0  ; Test bit
    BRz N           ; The bit is 0, skip add
    ADD R1, R1, R4  ; Add dest mask to result
N   ADD R3, R3, R3  ; L-shift source mask
    ADD R4, R4, R4  ; L-shift dest mask
    ADD R2, R2, #-1 ; Decr loop var
    BRp L
    ; End
    HALT
INP .FILL x1234
    .END
```

A9

1

```
PUSH A  -> A
PUSH B  -> B, A
POP     -> A            (B is removed)
PUSH C  -> C, A
POP     -> A            (C is removed)
PUSH D  -> D, A
PUSH E  -> E, D, A
PUSH F  -> F, E, D, A
POP     -> E, D, A      (F is removed)
PUSH G  -> G, E, D, A
POP     -> E, D, A      (G is removed)
POP     -> D, A         (E is removed)
POP     -> A            (D is removed)
PUSH H  -> H, A
```

2

最多的时候是执行 `PUSH F` 后或者 `PUSH G` 后，总共有 4 个元素

3

```
ENQUEUE I -> H, A, I
DEQUEUE   -> A, I       (H is removed)
ENQUEUE J -> A, I, J
ENQUEUE K -> A, I, J, K
DEQUEUE   -> I, J, K    (A is removed)
ENQUEUE L -> I, J, K, L
DEQUEUE   -> J, K, L    (I is removed)
DEQUEUE   -> K, L       (J is removed)
DEQUEUE   -> L          (K is removed)
DEQUEUE   -> (Empty)    (L is removed)
ENQUEUE M -> M
DEQUEUE   -> (Empty)    (M is removed)
```

A10

```
; PEEK函数实现
; 假设栈顶指针存储在R6中
PEEK    ST R1, SAVE_R1  ; 保存寄存器
        ST R2, SAVE_R2
        LD R1, EMPTY    ; R1 = -x4000
        ADD R2, R6, R1  ; 把栈顶指针和 x4000 比较
        BRz UNDERFLOW   ; 如果相等，说明栈空，下溢
        LDR R0, R6, #0  ; 否则，取栈顶元素
        LD R2, SAVE_R2  ; 恢复寄存器
        LD R1, SAVE_R1
        RET

; 栈下溢错误处理
UNDERFLOW LEA R0, UNDERFLOW_MSG
          PUTS
          HALT

; 数据定义
EMPTY   .FILL xC000 ; EMPTY 保存 -x4000
SAVE_R1 .BLKW #1
SAVE_R2 .BLKW #1
UNDERFLOW_MSG .STRINGZ "Stack underflow error"
```