Answer 6

A1

前两行程序先将 `TEXT` 和 `TEST` 处内存值存到 `R2` 和 `R3` 中，`TEXT` 处值显然为 `A` 的 ASCII 码 `x41`，但 `TEST` 处值需要考虑 `BR` 跳转值，而这又需要考虑到字符串 "An LC-3 program" 的长度为 16，因此 `TEST` 处指令为 `BRnp #-20`，对应十六进制 `x0BEC`.

之后的循环依据 `R2` 值是否为 0 决定终止，每次循环将 `R2` 值减小 1，因而循环执行共 65 次。每次循环中将 `R3` 加上本次的 `R2` 值，因此最终值为 $BEC_H + (65_D + 1_D) \times 65_D/2 = 144D_H$，也就是 `x144D` 或十进制 5197.

A2

1. 因为是中断服务，所以 `R7` 中并不保存返回地址，而 `RET` 会跳转到 `R7` 保存的地址，因此在中断服务中直接使用 `RET` 会跳转到错误的位置
2. 直接调用 `RET` 并没有恢复在中断服务中被保存下来的 Processor Status Register，因而并没有从特权模式恢复到用户模式，之前的状态码也没有被恢复
3. `RET` 也不涉及对栈的操作，但从中断服务恢复，应该要恢复栈指针

A3

1. 想打印的是字符 `I`，因为如果不考虑实际跳转，逻辑上在 `OUT` 前 `R0` 中保存的是 x0049，对应 ASCII 码表中的字符 `I`
2. 但实际上什么也不会输出，因为调用了一次 `JSR` 后再次调用 `JSR` 时没有保存原本的返回地址 `R7`，因此返回地址已经被覆盖，因而从 `B` 中回到 `A` 中再次执行 `RET` 会在 `A` 中出现死循环，无法真正执行到 `OUT`

A4

根据逻辑写出表达式，以下简记 XY 表示 X AND Y，$X + Y$ 表示 X OR Y，$\overline{X}$ 表示 NOT X，$X[N]$ 表示 X 的第 N 位

$X = PSR[15]\ (\overline{MAR[15]}\ \overline{MAR[14]}\ \overline{MAR[13]} +$
$\overline{MAR[15]}\ \overline{MAR[14]}\ \overline{MAR[12]} +$
$MAR[15]MAR[14]MAR[13]MAR[12]MAR[11]MAR[10]MAR[9])$

判断当 `PSR` 的最高位为 1 时（此时位于用户模式），且当 `MAR` 的值处于 [x0000,x3000)⋃[xFE00,xFFFF] 时（此时 `MAR` 位于特权内存区），`X` 为 1，否则为 0. 由 Figure C.2 中所述，`X` 为信号 `ACV`.

A5

1. `KBSR` 最高位为1
2. 读取 `KBDR`，并清除 `KBSR` 最高位
3. 当 `DSR` 最高位为 1 的时候，写入 `DDR`
4.

```
        .ORIG x3000
GET     LDI R1, KBSR
        BRzp GET
        LDI R0, KBDR
WAIT    LDI R1, DSR
        BRzp WAIT
        STI R0, DDR
        TRAP x25
KBSR    .FILL xFE00
KBDR    .FILL xFE02
DSR     .FILL xFE04
DDR     .FILL xFE06
        .END
```

A6

1. `WAIT`, `LETTER`, `CONTINUE`, `GETCHAR`, `-65`, `17`
2. R0 << 4, so that we can add the next value to the result.
3. It is not necessary. Dirty data will get shifted out.

Full code:

```
HEX_INPUT
        ST R1, SAVE_R1  ; R1 = Constant 1
        ST R2, SAVE_R2  ; R2 = Constant 2
        ST R3, SAVE_R3  ; R3 = Chars left (counter)
        ST R4, SAVE_R4  ; R4 = Keyboard status / Current char /
                        ;  Value represented by char
        LD R1, C1
        LD R2, C2
        AND R3, R3, #0
        ADD R3, R3, #4
        AND R0, R0, #0  ; R0 stores our result
GETCHAR
        ; R0 << 4
        ADD R0, R0, R0
        ADD R0, R0, R0
        ADD R0, R0, R0
        ADD R0, R0, R0
WAIT
        LDI R4, KBSR    ; Check keyboard status
        BRzp WAIT       ; KBSR[15] = 0, no char available, wait
        LDI R4, KBDR    ; Get KBDR
        ADD R4, R4, R1  ; Check if it is a letter
        BRzp LETTER     ; Got a capital letter
        ADD R4, R4, R2  ; Not a letter -> digit
        BR CONTINUE
LETTER
        ADD R4, R4, #10 ; Add 10 so that A -> 10
CONTINUE
        ADD R0, R0, R4  ; Add to result
        ADD R3, R3, #-1 ; Decr counter
        BRp GETCHAR     ; Wait for another char
        ; Restore regs
        LD R1, SAVE_R1
        LD R2, SAVE_R2
        LD R3, SAVE_R3
        LD R4, SAVE_R4
        RET
        ; Data
C1      .FILL #-65  ; -ord("A")
C2      .FILL #17   ; ord("A") - ord("0")
```

```
KBSR    .FILL xFE00
KBDR    .FILL xFE02
SAVE_R1 .BLKW 1
SAVE_R2 .BLKW 1
SAVE_R3 .BLKW 1
SAVE_R4 .BLKW 1
```

A7

1. `H3ll0_W0r1d!`
2. $18 \times 2 = 36$ bytes. (Each instruction takes 2 bytes; don't forget the `\0` at the end of the string.)

A8

1. It might reads an input character more than once.
2. It might overwrite an input character before it is processed.
3. The first scenario is more likely to happen, because CPU is much faster than human input.

A9

输出为 `F !` (注意中间有个空格)

A10

因为最后存到 `A` 处的值为 x1800，因此循环需要被执行，题干又提到一共执行了 9 条指令，因此循环只被执行了一次。同时，可以确定在 `BRnzp AGAIN` 之后，就紧跟着一条 `BR` 指令，且正确跳转到了 `DONE` 位置。

假设每次访存需要经过 x 个时钟周期，在状态机中数一遍，可以发现 `LD` 指令需要 $7 + 2x$ 个时钟周期，未进行跳转的 `BR` 指令需要 $5 + x$ 个时钟周期，进行跳转的 `BR` 指令需要 $6 + x$ 个时钟周期，`ST` 指令需要 $6 + 2x$ 个时钟周期，`HALT` 也就是 `TRAP` 指令需要 $5 + x$ 个时钟周期，这样已经有了 9 条指令中的 7 条经过的时钟周期数。

从表中可以得知，第 77 个周期下，处于状态 22，查表看到此时位于 `BR` 指令执行过程中最后一个状态，因为一共就 100 个周期，所以可以猜测此时为最后一个

`BR` 指令，这样就得到方程 $6+2x+5+x=100-77$，解得 $x=4$. (其实也可以尝试猜测为倒数第二个 `BR` 指令，这时有方程 $6+x+6+2x+5+x=100-77$，得到 $x=1.5$ 不可能；而再往前就会出现 $x<0$ 那就更不能成立了) 因此可以确定每次访存需要 4 个时钟周期。

从第 77 个周期往回推，可以得到第 57 个周期为循环中的最后一条指令，此处位于状态 1，也即是 `ADD`，再往前一行的 `DR` 为 000，也就是此处在更新 `R0`，因此均不可能为跳转，因而 `AGAIN` 只能位于 `BRz DONE` 这里. 表格再往前推，是状态 0，对应的是 `BR` 指令，因此这只能为第一次执行到 `BRz DONE`，由之前推理，对应的周期数应为 $2(7+2x)+5+x=19+5x=39$.

考虑到执行完挖空的两行后，执行到 `BRz DONE` 时会跳转到 `DONE`，也就是 57 个周期处的指令执行完后会得到结果为 0，而且是 `ADD`，而因为跳转后 `R0` 存的是 x1800，因而此处更新的是 `R1`，而 `R1` 之前为 x0001，因此 (b) 处空的指令为 `ADD R1, R1, #-1`.

对于 (a) 处的空，因为整个程序只涉及 `R0` 和 `R1`，而只能使用 `ADD`, `AND`, `NOT`，从 x0____ 变为 x1800 不能通过 `AND` 或是 `NOT`，所以 (a) 处只能是 `ADD R0, R0, R0`，且 (c) 处为 `c00`.

```
        .ORIG x3000
        LD R0, A
        LD R1, B
AGAIN   BRz DONE
        ADD R0, R0, R0
        ADD R1, R1, #-1
        BRnzp AGAIN
DONE    ST R0, A
        HALT
A       .FILL x0c00
B       .FILL x0001
        .END
```

完整代码如上所示，接着只需按照数据通路填写如下表格即可，注意 LD.XX 实际上是对 XX 的写使能信号。该程序功能为计算 $A\cdot 2^B$ 并存到 `A` 处。

Cycle Number	State Number	Control Signals
1	18	LD.MAR: LOAD LD.REG: NO GateMDR: NO LD.PC: LOAD PCMUX: PC+1 GatePC: YES
39	0	LD.MAR: NO LD.REG: NO BEN: 0/FALSE LD.PC: NO LD.CC: NO
48	1	LD.REG: LOAD DR: 000 GateMDR: NO GateALU: YES GateMARMUX: NO
57	1	LD.MAR: NO ALUK: ADD GateALU: YES LD.REG: LOAD DR: 001 GatePC: NO
77	22	ADDR1MUX: PC ADDR2MUX: PCoffset9 LD.PC: LOAD LD.MAR: NO PCMUX: ADDER
100	15	