Reading:

PP, Chp 8.1-8.3.3 (device registers, memory-mapped I/O, keyboard and display I/O),

PP, Chp 8.5 (interrupt overview).

PP, Chp 9.1-9.2.2 (TRAP/JSR subroutine calls, register saving).

PP, Chp 10.1-10.2 (stacks, push/pop, stack under/overflow, interrupt I/O, saving/restoring program state).

PP LC3 Reference (Also in docs/):
ASCII table: ..... inside back cover
Instruction Formats: ..... inside back cover
Instruction Descriptions: ..... App. A.3
Notation for Descriptions: ..... App. A.2
Memory Map: ..... App. A.1
TRAP routines: ..... App. A.3, Table A.2
I/O Device Registers: ..... App. A.3, Table A.3
Interrupt and Exception execution: ..... App. A.4 and App. C.6
Control FSM state diagram: ..... App. C, Fig. C.2 and Fig. C.7
Complete Datapath: ..... App. C, Fig. C.8
Memory-IO Bus: ..... lib/system.jelib:top

Problems:

PP, Chp 8:
8.5 (what is KBSR[15]?)
8.11 (polling vs. intr. efficiency)
8.14 (I/O addr. decode)
8.15 (KBSR[14] and intr. handling)

PP, Chp 9:
9.2 (TRAP execution)
9.13 (debugging JSR and RET)
9.19 (complete the intr. priority service call)

PP, Chp 10:
10.10 (cc pushed in intr.)
10.11 (device registers and IVT)
10.15 (keyboard interrupt handler w/ circular buffer)
10.24 (intr. vector and pointer handler)

Question.
Explain how the TRAP x21 (aka "OUT") service routine works. To find that code, start PennSim.jar and look at the TVT slot for x21, find the routine's address there, then look at that memory location and see the code.

P&P --- Chp. 8

8.5
KBSR[15] is the keyboard ready bit. It is 1 if new keyboard data is currently in the KDBR.

8.11
Since I/O is very slow compared with cpu instruction execution, polling uses cpu cycles that might be useful for another purpose. If there are multiple jobs currently available to be run, the cpu could do interrupt-driven I/O and switch from a job that was doing I/O to another that wasn't, thereby using those polling cpu cycles for useful work. So, provided there are other jobs that can be run, interrupt-driven I/O is more efficient.

8.14
Memory address decoding will prevent the memory from responding to the I/O request. The address bit pattern xFE02 does not correspond to the range of memory addresses for which the memory's address decoder will produce a logical 1. The address decoder for the KBDR will produce a 1 for this address.

8.15(a)

```
ld r3, A        ;---- put x4000 = b0100000000000000 into kbsr. This will enable keyboard interrupts,
sti r3, kbsr    ;---- since kbsr[14] = 1 after execution of these instructions.

again:
ld r0, B         ;---- put ascii for '2' into r0[7:0], and
trap x21         ;---- jump to "OUT" trap routine, which will print the content of r0[7:0] on the display.
brnzp again      ;--- and do this forever.

A .fill x4000
B .fill x0032
kbsr .fill xfe00
```

8.15(b)

```
ldi r0, kbdr     ;---- R0 gets new keyboard ascii data for a keypress event.
trap x21         ;---- jump to "OUT" routine, which prints whatever ascii is transmitted for the keypress,
trap x21         ;---- and do it twice.

rti              ;--- go back to whatever program was interrupted.

; ------------ This code will echo any keypress twice to the display.
```

8.15(c)
You would see the entire screen filled w/ '2' characters. This might appear to scroll or roll, depending on the size of the display. Pressing a key would cause the keypress to be echoed twice, but you would not see this as the '2' characters would overwrite the echoed characters faster than you could see them.

PP Chp. 9

9.2
(a) Trap vectors are the last eight bits of the TRAP instruction. So, the trap numbers range from x0000 to x00FF, which is 2^8 or 256 different numbers, and that is how many trap routines there can be in the LC3.
(b) RET must be used because the Trap routine only knows where to jump back to by the address stored into R7 when the TRAP instruction was executed. An unconditional BR instruction would have to have a static return address (PC+offset), but a trap routine can be called from anywhere and ought to jump back when done.
(c) First, the TRAP instruction is fetched. Next, the trap vector is sent from the IR to the MAR and the corresponding memory content is fetched to the MDR and then passed to the PC. So, 2, if you count the instruction fetch.

9.13
"JSR A" loads R7 and "JSR B" overwrites its content. The RET in routine B jumps into routine A just after the "JSR B" instruction. That next instruction in A is "RET". Since R7 has not changed, A's RET jumps right back to itself, endlessly.

9.19
The service routine first gets the content of the INTCTL register, which has a 1 in position k if IRQ_k is 1. The code for each device masks off its own interrupt bit, then tests to see if it is on. If so, it JSRs to the device's subroutine code, and upon returning from the subroutine, the service routine exits. In case the bit was off, the next device is checked. The code for each device is nearly identical (a-b-c) = (d-e-f) = (g-h-i), except for CDROM (j), where no checking is needed since some device caused the interrupt and it wasn't the others.

(a) LD R2, MASK8
(b) JSR HARDDISK
(c) BR END

(d) LD R2, MASK4
(e) JSR ETHERNET
(f) BR END

(g) LD R2, MASK2
(h) JSR PRINTER
(i) BR END

(j) JSR CDROM

(k) RTI

PP Chp. 10

10.10

```
;---------- The code being interrupted:
add r0, r0, #0    ;---- test r0
brz done          ;---- for 0
...

;---------- interrupt handler code w/o PSR saving:
and r0, r0, #0
...
```

The interrupt handler, if the interrupt occured just before fetch of the "brz" instruction, would clobber the CC values produced by the add instruction and cause the brz instruction to branch regardless of what had been in R0

.

10.11

Memory address x01F1 contains the address of the B interrupt (xF2) handler, x6200. x01F2 contains the C interrupt (xF2) handler's address, x6300. The x01F1 and x01F2 memory locations are part of the Interrupt Vector Table (IVT), which covers memory addresses x0180 - x01FF.

10.24
Since the keyboard interrupt vector is x34, then the address in the IVT used for this interrupt is x0134. The content of that memory location is the address of the keyboard interrupt handler, x1000.