1. Basics
   a. What are the four basic tasks for a computer?
      Modify data, store data, move data, control previous three tasks.
   b. Which components in computer system take care of those four tasks?
      data modification: cpu
      data storage: memory
      data transfer: cpu, bus
      control: cpu, memory, bus (fetch instructions)
   c. How are those four basic tasks implemented in the instruction fetch-execute cycle (Fig 3.12 [Sta13] [Sta10])?
      data modification: instruction execution phase
      data storage: instruction execution phase
      data transfer: get operand, store results
      control: instruction fetch and execution phases, recognizing interrupts, transferring control to interrupt handler

2. Program location in memory. Assume that ttk-91 program P is stored in main memory area 50200-50711.
   a. What are the values for base and limit registers? BASE = 50200, LIMIT = 512
   b. Assume that you are executing machine instruction "JUMP 200" (jump to location 200) in memory location 156 (program address space). What happens in the system hardware (in the instruction fetch-execute cycle)?
      At instruction fetch phase MMU maps the address 156 (program address in program address space) to main memory address 50356 and fetches the instruction stored there. The instruction is placed in IR and program counter PC is set to 157. At instruction execution phase the PC is changed to 200. Next instruction is fetched from (program) address 200.
   c. Assume that you are executing machine instruction "LOAD R1, 200" (load contents of memory location 200 to register R1)" in memory location 176. What happens in the system?
      In the instruction execution phase (in its fetch operand phase) the MMU is asked to load the value of (program) address 200. MMU will map it into main memory address 50400 and asks from main memory the value in that memory location, and it is stored as register R1 value. Next instruction is fetched from (program) address 177.
   d. Assume that you are executing machine instruction "JUMP 20000" in memory location 186. What happens in the system?
      At instruction execution phase the PC is set to 20000. Next instruction is supposed to be in address 20000. However, when the MMU tries to map 20000 to main memory address it finds that 20000≥512 and so the address in invalid. This causes an invalid memory address exception, and finally the exception handler for that type of exceptions will terminate the execution of this program.
   e. Assume now that the operating system decides to give that memory area to some other use, and relocates program P to memory area 134000-134512. What would the base and limit register values be now? What should be changed in program P (code) so that it would still work properly (i.e., exactly the same as in the original location)?
      BASE = 13400, LIMIT 512. You do not need to change anything in the program code, because all program addresses (in range 0-511) will be mapped to main memory address later on in the fetch-execute cycle.

3. Interrupts
   a. How does an interrupt differ from a subroutine (function, method)?
Interrupts are implemented in a very similar fashion, but they are not invoked (called) with their own machine instruction (CALL), but instead with an interrupt happening almost in random times. All possible interrupt types are known and there is an interrupt handler ready for all of them. Parameter passing and return value passing may happen in various ways depending on the interrupt. Interrupt handlers usually run in privileged state (all machine instruction available and all main memory addressable), but also many other operating systems routines execute in privileged state.

b. How are the interrupts implemented at hardware level (in fetch-execute cycle)?
   At the end of each fetch-execute cycle, the hardware (HW) checks whether any interrupts have occurred. If so, the HW stores the values of old PC, SR, and maybe BASE/LIMIT to the stack, and then finally the PC is set to the start of appropriate interrupt handler.

c. How are the interrupts implemented at software level (in the operating system)?
   There is an interrupt handler for each type of interrupts, and it takes care of all actions needed by that interrupt. After that the control returns to the interrupted program, or some other program (if the interrupted program can not continue).

d. Why would you want to access all of the memory and execute in privileged mode during interrupt handling?
   Interrupt processing is an essential part of operating system (OS), and you must be able to modify internal protected OS data structures during the interrupt handling. Also the return-from-interrupt instruction (e.g., IRET) does something special, that is not allowed by normal programs.

e. How else (in addition interrupt) could the operating system get to execute?
   The only other possibility is a special machine instruction that transfers control explicitly to the OS. E.g., it could be instruction "SVC <nr>", where <nr> the id for the OS-service requested.

f. What does the operating system do while the processor is executing user program?
   Nothing. When it is not in execution, it can not do anything. OS can do something only after some OS routine gets to execute. This happens explicitly with SVC instruction or via interrupts (exceptions).

4. [2 htp] Interrupt handler. Assume that the interrupt handler can access all of the memory and that it executes in privileged mode. Assume also, that at the end of fetch-execute cycle one has observed
   i. integer overflow interrupt (interrupt nr 28, state register bit O in ttk-91), or
   ii. I/O interrupt (interrupt nr 24, state register bit I)
   We want to implement an interrupt handler (e.g., to ttk-91 architecture).

a. How do you implement the transfer of control to the interrupt handler?
   What do you need to save before transferring control to the interrupt handler?
   In the last phase of fetch-execute cycle the HW checks if any interrupt has occurred. If so, then (i) it saves registers PC, SR, BASE and LIMIT somewhere (e.g., stack), sets BASE=0, LIMIT=memory size, state register bit P=1 (privileged mode), and finally set PC=interrupt handler address for interrupt 28 or 24. This may be found from (absolute) main memory address 28 or 24.

b. What does the IRET-instruction (interrupt return) actually do?
   Original values for registers BASE, LIMIT and SR are restored. Bit P in SR is set to 0. Finally PC value is restored and control transferred to the interrupted program (if that is allowed to continue the execution).

c. Describe the general structure of the interrupt handler in cases i and ii. What does it do?
   Overflow (fatal error, can not continue)
   - give an error description to user
   - save all register (and memory) contents to disk for further analysis
- SVC <scheduler> to select another program to run now

I/O interrupt (std OS management)
- save those register values that you need to use yourself
- do the admin that is needed (and tell waiting program that it can proceed)
- recover register values that were saved
- IRET to recover all saved register and to return control to the interrupted program

d. What happens if during the execution of interrupt handler nr 4 an integer overflow interrupt occurs?
   Can not happen. System will crash. There must not be errors in interrupts handlers (or other important parts of the OS). This is relating to larger question on how an OS could recover from its own mistakes. Usually it does not.

5. Assume that you are coding the ttk-91 simulator with some high level programming language, like Javalla or C. Simulator reads machine language ttk-91 code (stored in simulated ttk-91 memory) and emulates instruction execution with simulated registers one instruction at a time.
   a. How would you represent the simulated structures (ttk-91 registers, memory) in your program?
      The referable registers (those that can be named) in machine instructions are a small integer array r[8], and the memory is larger integer array mem[512] or mem[MESMSIZE]. All special registers (PC, SR, TR, etc) are integer variables.
   b. How would you code the instruction fetch phase?
      IR = mem[PC];  // IR is simulated instruction register, PC simulated program counter, and mem the simulated memory.
   c. How you you code disassembling instruction parts (opcode, Rj, M, Ri, ADDR) from the instruction?
      The contents of simulated instruction register IR must be split to fields Opcode, Ri, M, Rj and Addr (which are all implemented as integer variables). You can do this easily with arithmetic operations:
      Opcode = IR / 0x1000000;  // remove 24 zeroes from end
      Rj = IR / 0x200000 % 8;  // remove 24 zeroes from end, take last 3 bits
      M = IR / 0x40000 % 4;  // remove 18 zeroes from end, take last 2 bits
      Ri = IR / 0x10000 % 8;  // remove 16 zeroes from end, take last 3 bits
      Addr = IR % 0x10000;  // take last 16 bits
      or bit manipulation
      Opcode = IR >> 24;  // move right 24 bits
      Rj = (IR >> 21) & 0x7;  // move right 21 bits and take last 3 bits
      M = (IR >> 19) & 0x3;
      Ri = (IR >> 18) & 0x7;
      Addr = IR & 0xFFFF;  // take last 16 bits
      Note: if your programming language does not allow hexadecimal numbers, you need to use corresponding decimal numbers.
      Using mathematical operations works this way only because the leftmost bit is always zero and all instructions are represented by positive integers.
   d. How would you code fetching the operand to temporary register TR?
      if (Rj == 0)
         TR = Addr;
      else
         TR = r[Ri] + Addr;
      if (M == 3) or (Opcode == 1 && M == 2)) HandleError (BadMode);
if (M > 0) TR = Mem[TR];     // memory read, or indirect memory write
if (M > 1) TR = Mem[TR];     // indirect memory read

e. How would you code the execution phase of instruction "add r2, r3"?
   You may assume that the value of the latter operand (r3) is already in simulated register TR.
   if (Opcode == 17)     // unless a case-statement in large switch structure
      r[Rj] += TR;

f. How would you program the execution phase of instruction "jump loop"?
   You may assume that the value of the latter operand (memory address "loop") is already in
   simulated register TR.
   if (Opcode == 32)     // unless a case-statement in large switch structure
      PC = TR;