Question 1: At what address is the Boot ROM loaded by QEMU?
0x1000

Question 2: At a high-level, what are the steps taken by the loaded Boot ROM?

1. Setting up registers like a2 and a1 with necessary values, pointers or offsets to other memory locations.
2. Using `mhartid` to determine which CPU core is active.
3. Transfer control to the bootloader, where the system startup process continues.

Question 3: What address does our Boot ROM jump to at the end of its execution?

Boot ROM jumps to the address 0x80000000

Question 4: What is the specified entry function of the bootloader in the linker descriptor?

start ()

Question 5: Once your linker descriptor is correctly specified, how can you check that your bootloader's entry function starts to execute after the Boot ROM code?

Using gdb to Step through the instructions and checking that the execution moves from the Boot ROM to the bootloaders entry function

Question 6: What happens if you jump to C code without setting up the stack and why?

Jumping to the C code without properly stacking up the stack would lead to unpredictable situations like the program crashing which I did encounter while working on the project. I didn't use the thread id and moved the pointer to the base address of the stack + STSIZE * NCPU which caused the program to crash.

Question 7: How would you setup the stack if your system had 2 CPU cores instead of 1?

I would say the process would be the same as we did for one CPU but we would just allocate two distinct regions of memory based on the hartid to determine which CPU is executing the code and also a stack pointer pointing to the top of the stack.

Question 8: By looking at the start function, explain how privilege is being switched from M-mode to S-mode in the mstatus register. Explain what fields in the register are being updated and why.

First the current status of the machine is read and stored in the variable x. Then the MPP field is cleared which is a part of the mstatus register which stores the privilege level to the which the machine return to after executing the mret. Since it is set to M mode at the start we clear it and then set it to S mode. After this modification of the mstatus register the updated value is written back using w_mstatus(x).

Implementation:

The first coding task of the assignment was to set up the linker script to ensure the right entry point for the bootloader after Boot ROM execution which was achieved by setting the correct start address and the code sections (.text, .data, .rodata, .bss) and also the end address for the bootloader. Next task was to setup the stack before jumping to the C code. The bl_stack was initialised with the base address and using the CPU hartid I moved the stack pointer to the top of the stack. This was also the first place I made a mistake of setting the sp to the address (bl_stack + (STSIZE)*NCPU) which led to my program crashing. The next task was to load the use selected OS after the program jumps to the start function. For this I just had to point the struct elf_hdr to the address at RAMDISK which tells the CPU to treat the memory address as the starting point of the struct and the fields are present at the memory location at an offset and can be accessed using the pointer. This initialises the pointer with the kernel binary elf headers. Using the fields of the struct at the offset I found out the kernel size and the starting address of the .text section. These values are then used to copy the kernel binary to the start address we found using the kernel_copy function. For this the first 4 KB or four blocks each of size 1024 were skipped to skip copying the elf header and then rest of the blocks found by the formula (kernel_size/BSIZE). Here the struct buf was also used to copy the data from the kernel binary to the buffer and then to the start address. Then the kernel entry address is found by using the method find_kernel_entry_addr. The w_mepc function writes this address to the mepc register. The mret instruction is executed to jump to the address in the mepc register switching from M-mode to S-mode. And then to pass the system information to the kernel we use the structure sys_info which is written to a specific memory address (SYSINFOADDR: 0X84500000) and the bootloader start and end address and the DRAM start and end address is stored. The next task was to set up the PMP configuration. In which we used the TOR configuration and NAPOT to set up the privilege bits for the specified regions. The TOR configuration requires the base address and the Top of the range address. The highest accessible address to the OS is added to the base address which is then right shifted by 2 bits and we also use the pmpcfg register to setup the permission bits for this address range. Furthermore NAPOT configuration is done by using the base address and the size using the formula (base address >> 2) + (size >> 3)-1 . The final task was to enable secure boot. I used the sha256 hash functions provided to compute the hash of the kernel binary and compare it with the expected measurement and if the verification is complete the kernel boots up in NORMAL mode otherwise it boots up in RECOVERY mode for which I set up the recovery kernel by again finding the kernel load address by passing the struct elf_hdr to the memory location RECOVERYDISK this time and setting it up in a similar way as I did for NORMAL mode. There were many errors that I just couldn't figure out at the start. Also not having coded in C in a long time had me beat. But I think this was a great project to understand and refresh the C language specific challenges.