Assignment – 1

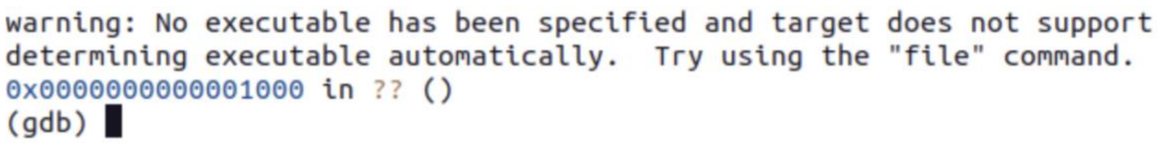
Course: CSE536 Advanced Operating Systems

Name: Sri Rupin Potula

Question -1:

a. **At what address is the Boot ROM loaded by QEMU?**

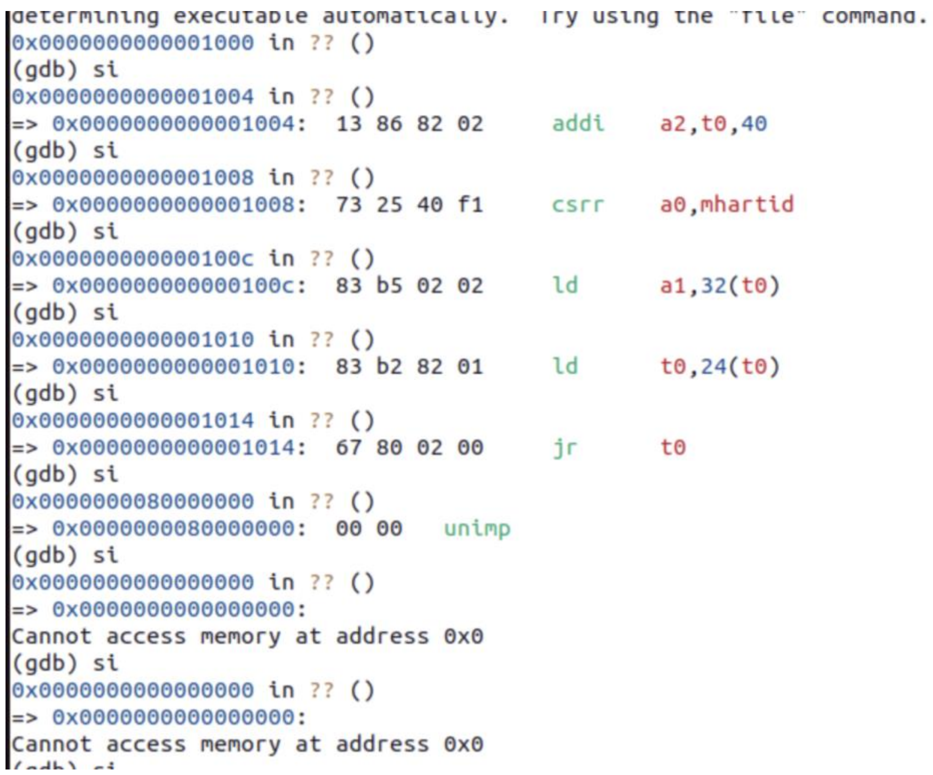
Answer: The BOOT ROM was loaded by QEMU at 0x0000000000001000.



```
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000001000 in ?? ()
(gdb) 
```

b. **At a high-level, what are the steps taken by the loaded Boot ROM?**

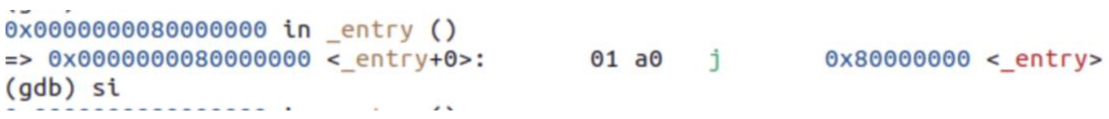
Answer: The BOOT ROM is loaded by QEMU at 0x0000000000001000. Post this, the addition operation was performed to store the value in a2. This step is actually storing the address in t0. Then, from the instruction “mhartid” was used to identify the thread that was being used. After this, the jump instruction was employed to jump to _entry.



```
determining executable automatically.  Try using the "file" command.
0x0000000000001000 in ?? ()
(gdb) si
0x0000000000001004 in ?? ()
=> 0x0000000000001004:  13 86 82 02     addi    a2,t0,40
(gdb) si
0x0000000000001008 in ?? ()
=> 0x0000000000001008:  73 25 40 f1     csrr    a0,mhartid
(gdb) si
0x000000000000100c in ?? ()
=> 0x000000000000100c:  83 b5 02 02     ld      a1,32(t0)
(gdb) si
0x0000000000001010 in ?? ()
=> 0x0000000000001010:  83 b2 82 01     ld      t0,24(t0)
(gdb) si
0x0000000000001014 in ?? ()
=> 0x0000000000001014:  67 80 02 00     jr      t0
(gdb) si
0x0000000080000000 in ?? ()
=> 0x0000000080000000:  00 00   unimp
(gdb) si
0x0000000000000000 in ?? ()
=> 0x0000000000000000:
Cannot access memory at address 0x0
(gdb) si
0x0000000000000000 in ?? ()
=> 0x0000000000000000:
Cannot access memory at address 0x0
```

c. **What address does our Boot ROM jump to at the end of its execution?**

Answer: At the end, the BOOT ROM jumped to 0x0000000080000000 in _entry().



```
0x0000000080000000 in _entry ()
=> 0x0000000080000000 <_entry+0>:       01 a0   j       0x80000000 <_entry>
(gdb) si
```

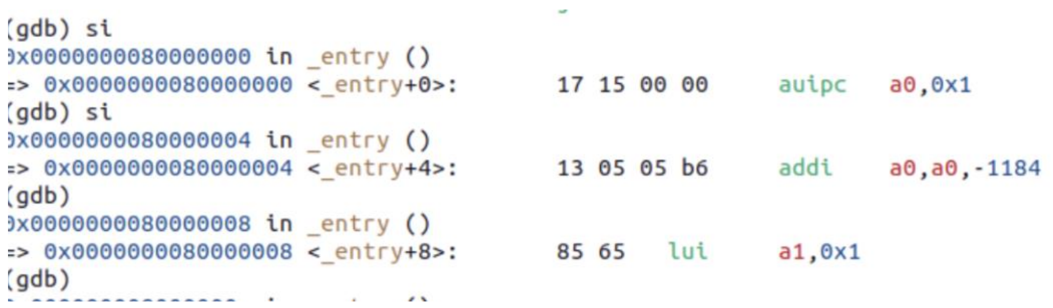
Question -2:

a. **What is the specified entry function of the bootloader in the linker descriptor?**

Answer: The entry function that was specified is "_entry".

b. **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?**

Answer: One way that can be confirmed that the entry function starts to execute after the BOOT ROM is in the linker script the entry value is defined which is ". = 0x80000000;", which maps the output as shown in the figure below, where the value was jumped to the entry address. If this wasn't mentioned, it would say that the address cannot be accessed at 0x0.

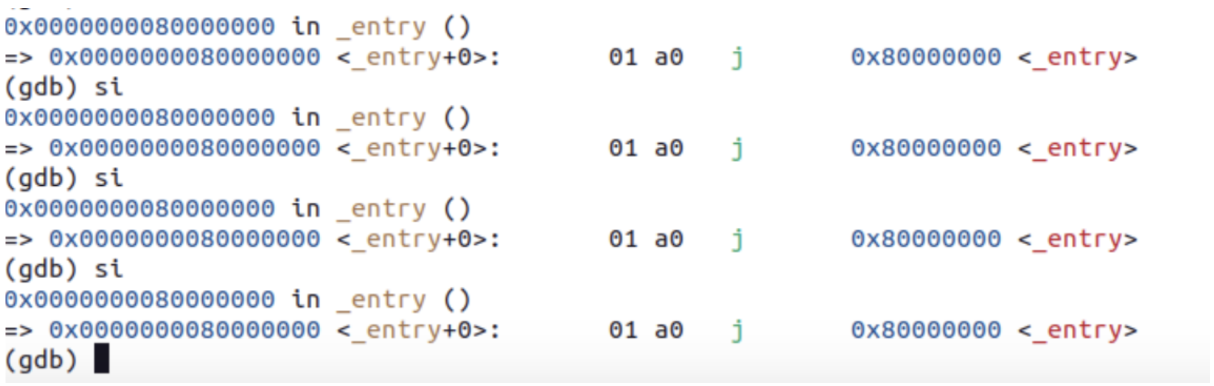


```
(gdb) si
0x0000000080000000 in _entry ()
=> 0x0000000080000000 <_entry+0>:       17 15 00 00     auipc   a0,0x1
(gdb) si
0x0000000080000004 in _entry ()
=> 0x0000000080000004 <_entry+4>:       13 05 05 b6     addi    a0,a0,-1184
(gdb)
0x0000000080000008 in _entry ()
=> 0x0000000080000008 <_entry+8>:       85 65   lui     a1,0x1
(gdb)
```

Question -3:

a. **What happens if you jump to C code without setting up the stack and why?**

Answer: Without setting up stack, the bootloader cannot proceed further in the execution. For instance, when stack was not initialized the code couldn't be jumped to the start function and it falls under infinite loop execution starting from 0x0000000080000000. Because of which the bootloader initialization will not happen. The main reason this would be happening is the stack the place where the variables stored and the right values mayn't be read. Incorrect mapping of the values would happen as the stack pointer would not be initialized as this would be dumped with random values.



```
0x0000000080000000 in _entry ()
=> 0x0000000080000000 <_entry+0>:       01 a0   j       0x80000000 <_entry>
(gdb) si
0x0000000080000000 in _entry ()
=> 0x0000000080000000 <_entry+0>:       01 a0   j       0x80000000 <_entry>
(gdb) si
0x0000000080000000 in _entry ()
=> 0x0000000080000000 <_entry+0>:       01 a0   j       0x80000000 <_entry>
(gdb) si
0x0000000080000000 in _entry ()
=> 0x0000000080000000 <_entry+0>:       01 a0   j       0x80000000 <_entry>
(gdb) █
```

b. **How would you setup the stack if your system had 2 CPU cores instead of 1?**

Answer: When the system has two CPU cores, two CPU stacks can be setup with the required sizes corresponding to each core. The stacks must be specified in the linker script for each core.

Question -4:

a. **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.**

Answer:

1. We to fetch the value of x from r_mstatus().
2. To the value in STATUS_MPP_MASK a negation is applied. Then an AND operation is made. Then the bits of x and STATUS_MPP_MASK values are compared which typically does for all the 1 bits in x to that correspondence bit in STATUS_MPP_MASK are made 1, whereas when there is a mismatch between them 0 would be obtained. The output would be like: The STATUS_MPP_MASK (3L<<11) then the negation value would be 1111 1111 1111 1111 1111 **00**11 1111 1111.
3. The equation for STATUS_MPP_MASK the OR operation was applied to add the one bits. Using STATUS_MPP_MASK(1L<<11) then the OR operation makes transforms the 0 to 1 => 0000000000000000000**1**00000000000
4. The value was updated using m_status.

Question – 5:

a. **How does kernel_copy work? Please document its steps.**

Answer: Function kernel_copy is used to load the kernel namely RECOVERY and NORMAL kernel. Firstly, the block-number from the buffer would be compared with the full size if the block-number exceeds the system enters into panic mode. Else, the offset is calculated using the ‘b->blockno’ which is the block number then multiplied by BSIZE(1024). Then the addr is calculated. Initially, the addr is initiated to zero. If the kernel is NORMAL, the addr would be the RAMDISK + disaddr else, addr would be RECOVERY + diskaddr. Post this, memmove operation is made that copies the addr (the address that is calculated) into the buffer data that is of size BSIZE. The valid bit of the buffer is set to 1.