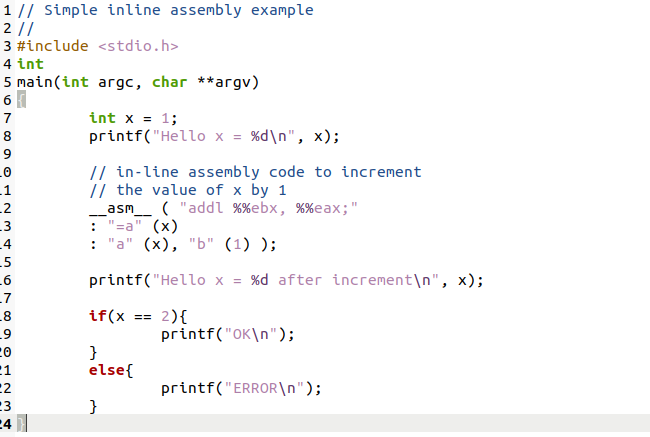
**NAME:** Dev Sandip Shah

**ROLL NO:** 200123074

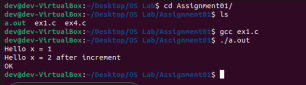
**Exercise 1**. **Become familiar with inline assembly by writing a simple program. Modify the program ex1.c (at end of this file) to include inline assembly that increments the value of x by 1.**

**Answer:**

Code:

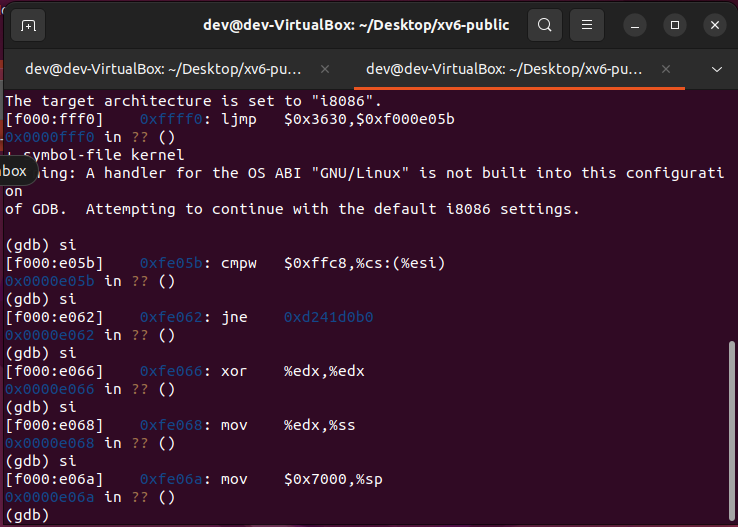


Output:



**Exercise 2**. **Use GDB's si (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing.**

**Answer:**



1st Instruction: [f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b

* Jump to CS = $0xffff0 and IP = 0xe05b
* 0x3630 is jump to this CS (earlier in the BIOS)
* 0xf000e05b is the IP which is different from the lab because it is 32 bits rather than 16 bits and that is all the way into the top of the extended memory location but before the memory mapped PCI device location reserved by the BIOS

2nd Instruction: [f000: e05b] 0xfe05b: cmpw $0xffc8, %cs:(%esi)

* Compare content at 0xffc8 and with content at code segment offset with value at esi.
* Esi: - 32-bit source index register

3rd Instruction: [f000: e062] 0xfe062: jne 0xd241d0b0

* Jump to 0xd241d0b0 if the above comparison does not set ZF

4th Instruction: [f000: e066] 0xfe066: xor %edx, %edx

* ZF was set thus jump of previous instruction doesn’t occur
* It set edx to zero, edx is 32-bit general-purpose register.

5th Instruction: [f000: e068] 0xfe068: mov %edx, %ss

* Move content of stack segment register(ss) to edx

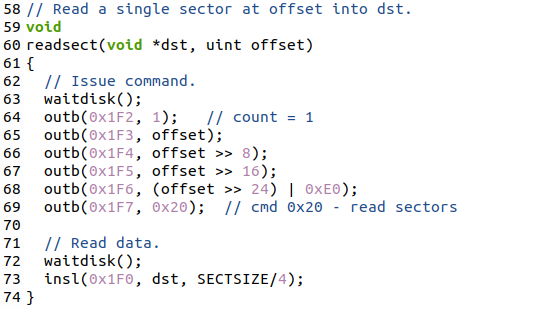
6th Instruction: [f000: e06a] 0xfe06a: mov $0x7000, %sp

* Move content at the location pointed 16-bit stack pointer(sp) to $0x7000

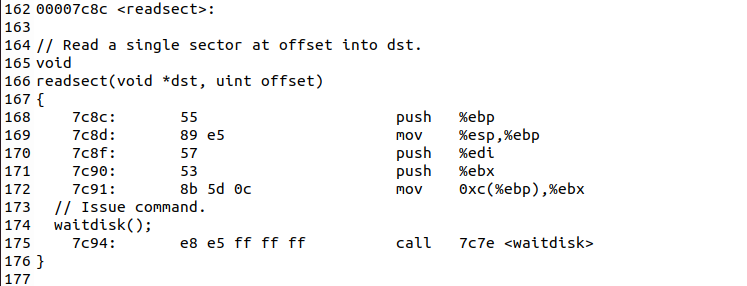
**Exercise 3**.

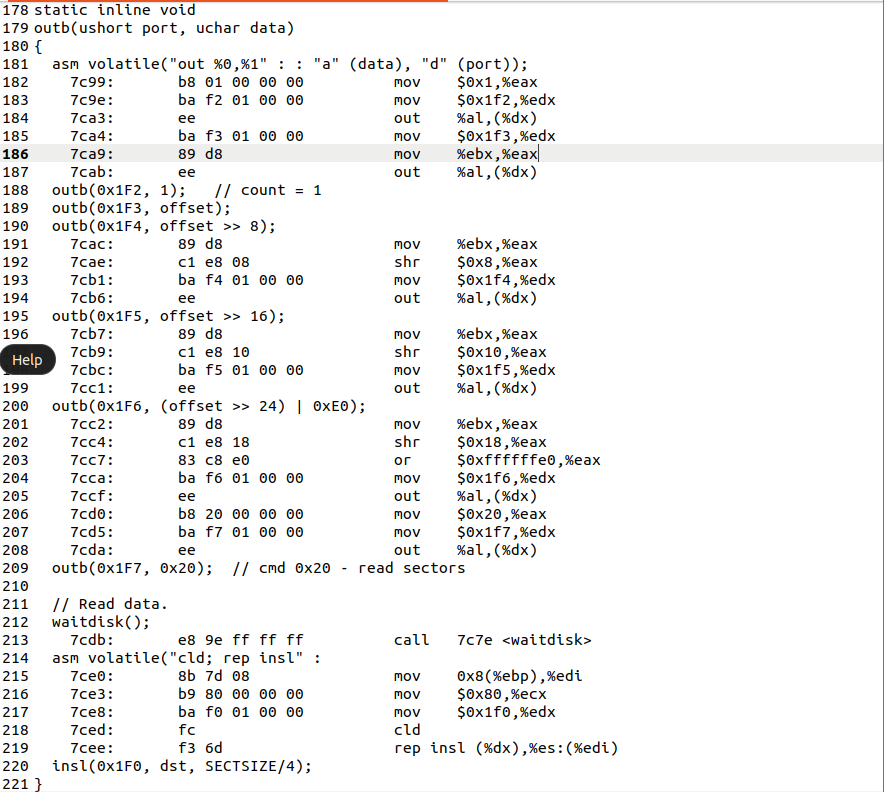
**Answer:**

The code for readsect() is given below-

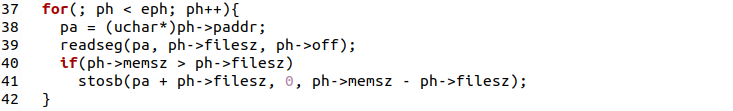


The assembly code for readsect() is given below-





The for loop that reads the sectors of kernel from the disk is given below:



The first instruction of this for loop is:



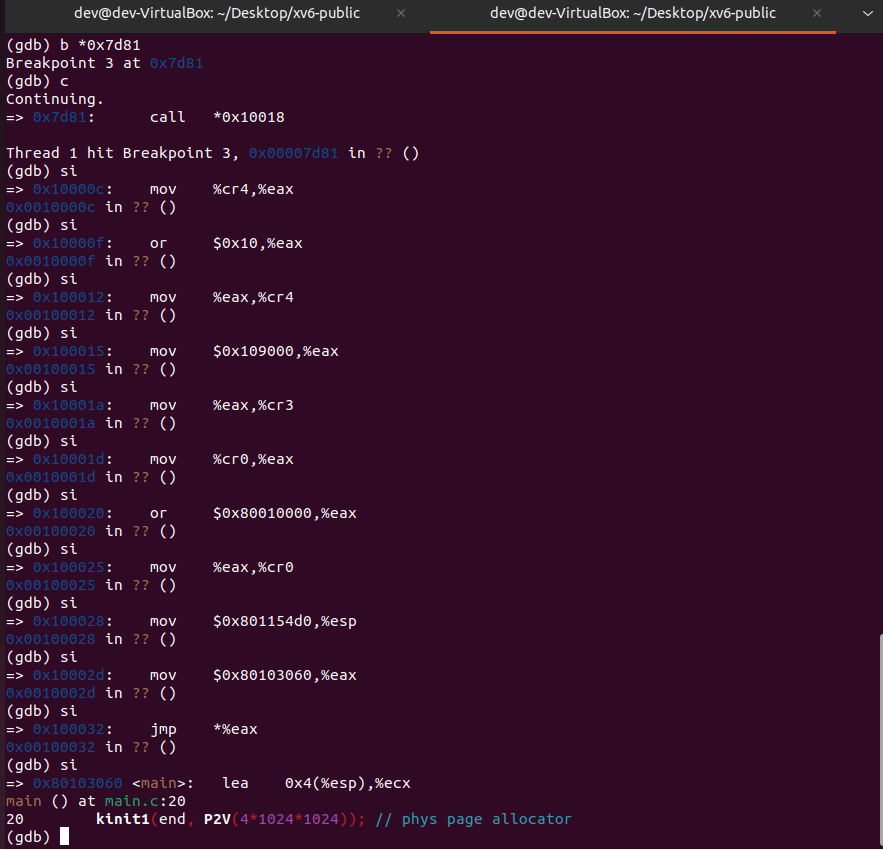
The Last Instruction of this for loop is:



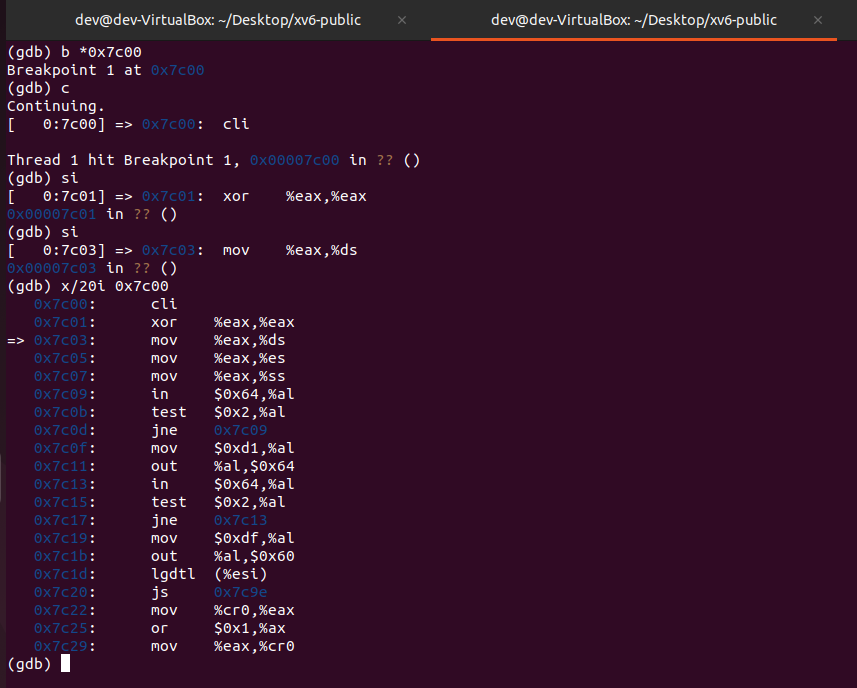
The explanation for the first instruction is that the first operation on entering the for loop will be comparison between the values of ph and eph because the loop will run only when ph < eph. The explanation of last instruction is that the loop ends when the values of ph and eph become equal and hence the loop jumps to the next instruction at 0x7d91. Hence the jump instruction will be the last instruction of the for loop. The next instruction after the for loop is

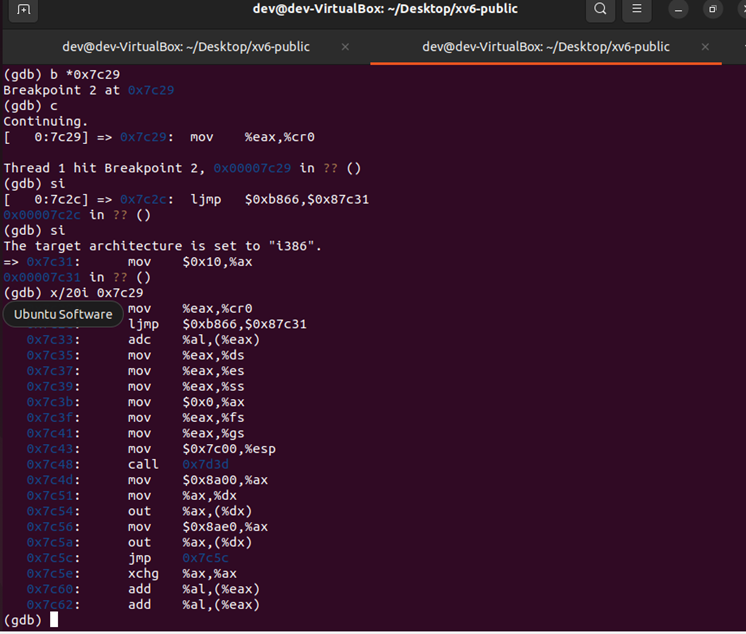


Making a breakpoint at that address and then stepping into further instructions gives the following output.



1. The command ljmp $(SEG\_KCODE<<3), $start32 causes the switch from 16 to 32-bit mode in bootasm.S





1. By analysing the contents of bootasm.S, bootmain.c and bootblock.asm, we conclude that bootasm.S switches the OS into 32-bit mode and then calls bootmain.c which first loads the kernel using ELF header and the enters the kernel using entry(). Hence the last instruction of bootloader is entry(). Looking for the same in bootblock.asm, we find out the instruction to be

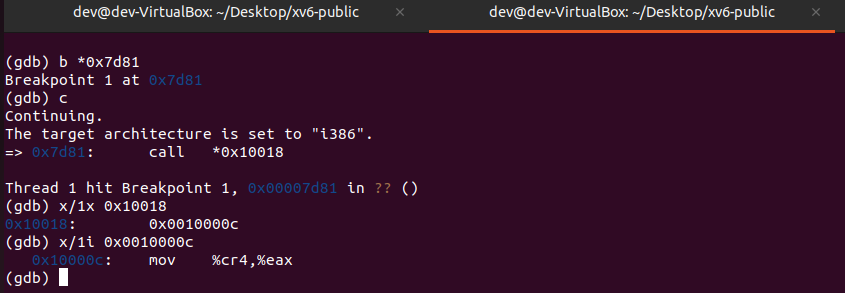


which is a call instruction which shifts control to the address stored at 0x10018 since dereferencing operator (\*) has been used. Now we need to know the starting address of the kernel. We can find this by two methods:

1. By looking at the first word of memory stored at 0x10018 (by using the command “x/1x 0x10018”)
2. (ii) By looking at the contents of “objdump -f kernel”

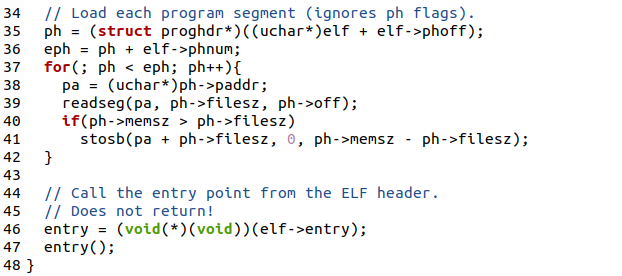
After getting the starting address of kernel, we need to see what is the instruction stored at that address to get the first instruction of kernel. We can do this by two methods:

1. By using “x/1i 0x0010000c”
2. By looking into kernel.asm



Hence, the first instruction of kernel is:





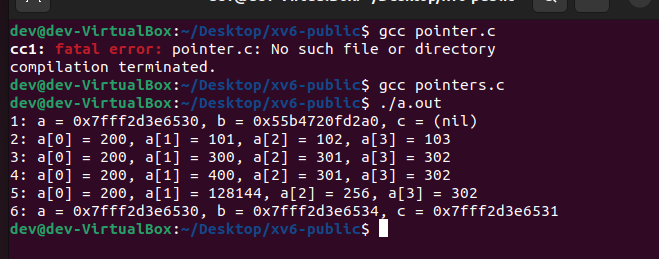
The above lines of code are present in bootmain.c. This is the code that is used by xv6 to load the kernel. xv6 first loads ELF headers of kernel into a memory location pointed to by “elf”. Then it stores the starting address of the first segment of the kernel to be loaded in “ph” by adding an offset (“elf->phoff”) to the starting address (elf). It also maintains an end pointer eph which points to the memory location after the end of the last segment. It then iterates over all the segments. For every segment, pa points to the address at which this segment has to be loaded. Then it loads the current segment at that location by passing pa, ph->filesz and ph->off parameters to readseg. It then checks the memory assigned to this sector is greater than the data copied. If this is true, it initializes the extra memory with zeros.

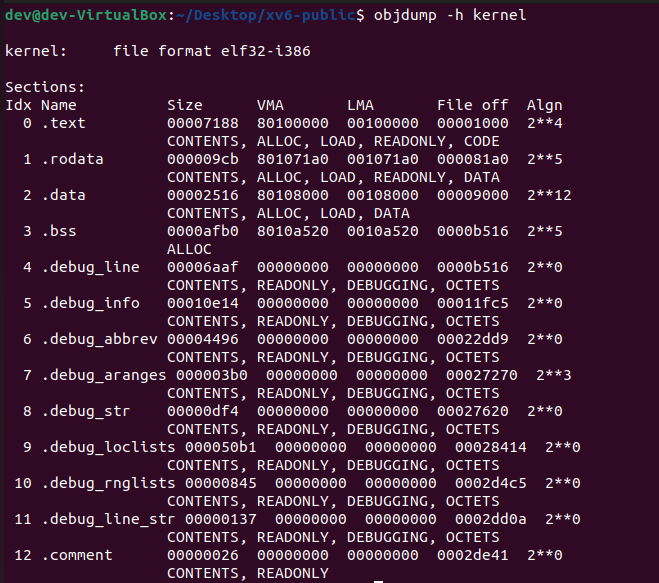
Coming back to the question, the boot loader keeps loading segments while the condition “ph < eph” is true. The values of ph and eph are determined using attributes phoff and phnum of the ELF header. So, the information stores in the ELF header helps the boot loader to decide how many sectors it has to read.

**Exercise 4**. **Read about programming with pointers in C. Then download the code for pointers.c, run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in lines 1 and 6 come from, how all the values in lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted. We also recommend reading the K-splice pointer challenge as a way to test that you understand how pointer arithmetic and arrays work in C.**

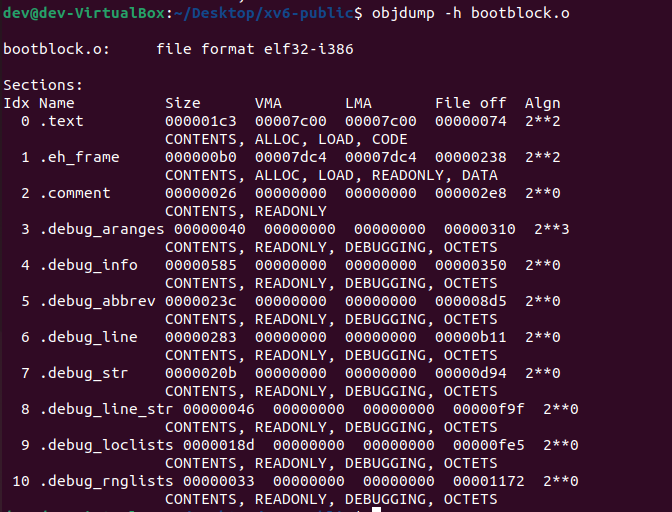
**Answer:**

Output of code in pointer.c





As we can see in the above screenshot, VMA and LMA of .text section is different indicating that it loads and executes from different addresses.



As we can see in the above screenshot, VMA and LMA of .text section is same indicating that it loads and executes from the same address.

**Exercise 5. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in Makefile to something wrong, run make clean, recompile the lab with make, and trace into the boot loader again to see what happens. Don't forget to change the link address back and make clean again afterwards!**

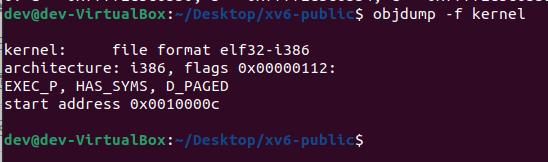
**Look back at the load and link addresses for the kernel. Unlike the boot loader, these two addresses aren't the same: the kernel is telling the boot loader to load it into memory at a low address (1 MB), but it expects to execute from a high address. We'll dig in to how we make this work in the next section.**

**Besides the section information, there is one more field in the ELF header that is important to us, named e\_entry. This field holds the link address of the entry point in the program: the memory address in the program's text section at which the program should begin executing. You can see the entry point:**

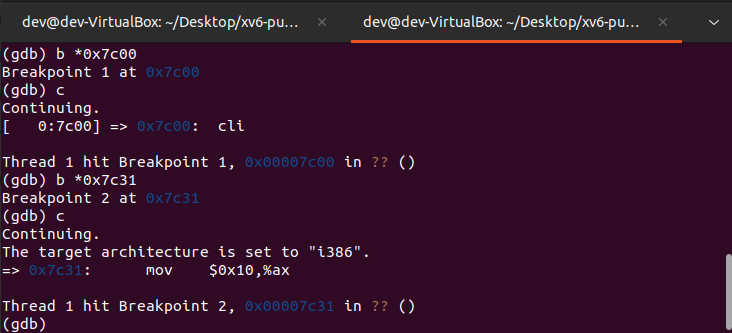
**$ objdump -f kernel**

**You should now be able to understand the minimal ELF loader in bootmain.c. It reads each section of the kernel from disk into memory at the section's load address and then jumps to the kernel's entry point.**

**Answer:**

****

When boot loader’s link address is **0x7C00** then commands are running properly and transition from 16 to 32 bit was occurring at **0x7C31** address location as seen below:



But when the boot loader’s link address is changed to any other address (I took **0x7C24** in this case), after running

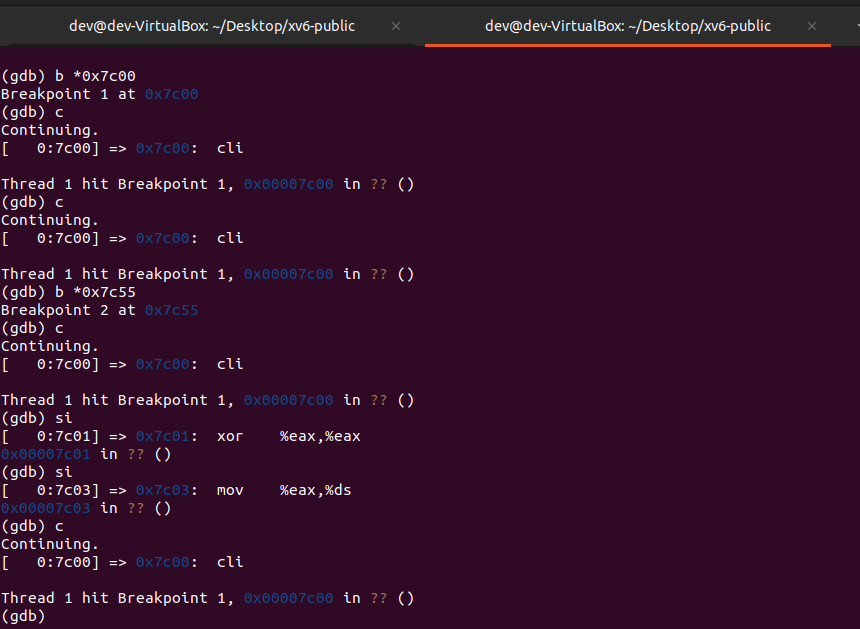
**make clean**

**make**

**and restarting gdb**

**and continuing from address location 0x7C00**,

then the boot loader is restarting again and again after running some instructions in the gdb.



As seen in the image above, we tried to run commands after continuing from breakpoint at **0x7C00** address location and we always end up hitting the same breakpoint at **0x7C00**. Also 16-to-32-bit architecture change didn’t occur as breakpoint b \***0x7C55** is not hit which should be responsible for architecture change in this case.

**ljmp $(SEG\_KCODE<<3), $start32** is the first instruction that breaks. Before changing the link address of the boot loader, from address **0x7C00**, after performing 2-3 si 10 instructions, architecture changed from 16 to 32 bit.

But after changing the link address to **0x7C24**, architecture didn’t change which means that the boot loader is not loaded properly at the changed link address.

**Exercise 6:**

**Answer:**

At the point when BIOS enters the boot loader (at first breakpoint):



At the point when the boot loader enters the kernel (at second breakpoint):



8 words of instruction at 0x00100000 at the point when BIOS enters the boot loader and 8 words of instruction at 0x00100000 at the point when the boot loader enters the kernel are different as when the BIOS enters and loads the boot loader, then it just loads it in memory location between 0x7C00 and 0x7DFF due to which all the 8 words of instructions are zero at 0x00100000. But before the boot loader enters the kernel, it already has performed the 16-to-32-bit transition and setting up of stack which leads to new instructions at address 0x00100000.