Report for CS344 Assignment-0

By:- Akshat Arun
190101007

Q1.

The modified code which includes inline assembly code to increment the value of x by 1

```
#include <stdio.h>
int main(int argc, char **argv)
{
        int x = 1;
        printf("Hello x = %d\n", x);
        //
        // Put in-line assembly here to increment
        // the value of x by 1 using in-line assembly
        //
        asm ("add $1, %0"
                 :"=r"(x)
                 :"0"(x)
                );
        printf("Hello x = %d after increment\n",x);
        if(x == 2){
                printf("OK\n");
        }
        else{
                printf("ERROR\n");
        }
}
```

Q2.

When the BIOS runs, it sets up an interrupt descriptor table and initializes various devices such as the VGA display. The explanation of the instructions is as follows:-

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

a) f000 is the code segment, fff0 is the IP, 0xffff0 is the physical address
b) ljmp is the instruction.
c) The instruction tells to jump to CS: 0xf000 and Ip: 0xe05b

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

a) Compare the value at address 0xffc8 and value at the physical address formed by the code segment CS with offset (equal to the value of esi register).

Instruction 3: [f000:e062] 0xfe062: jne 0xd241d0b2

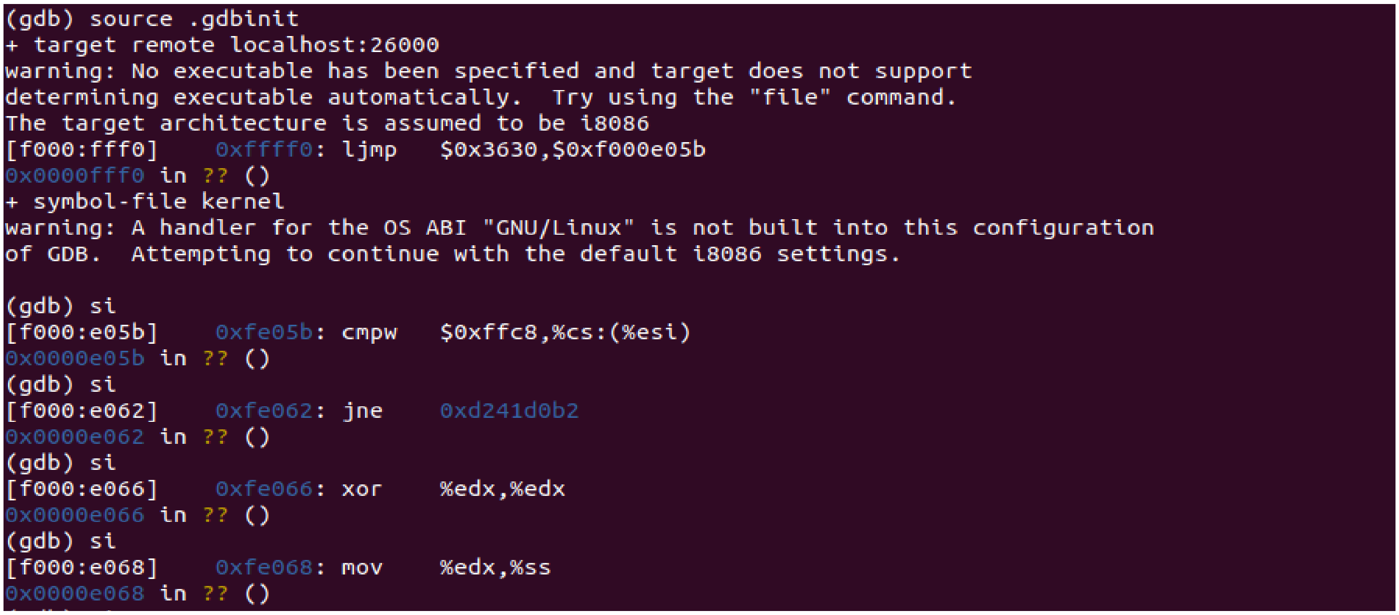
a) This instruction tells you to jump to the physical address 0xd241d0b2 if the above comparison is not true.
b) jne stands for jump if not equals to.

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

a) Set the value of edx register to 0.
b) Because the xor of a number with itself is always 0.

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

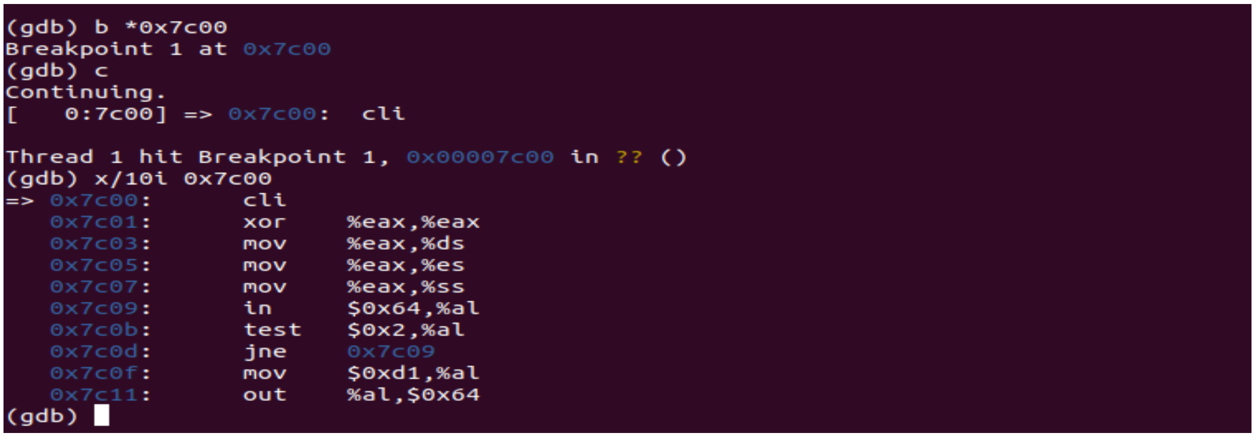
a) Move the contents of the edx register to the stack segment register.



```
(gdb) source .gdbinit
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
```

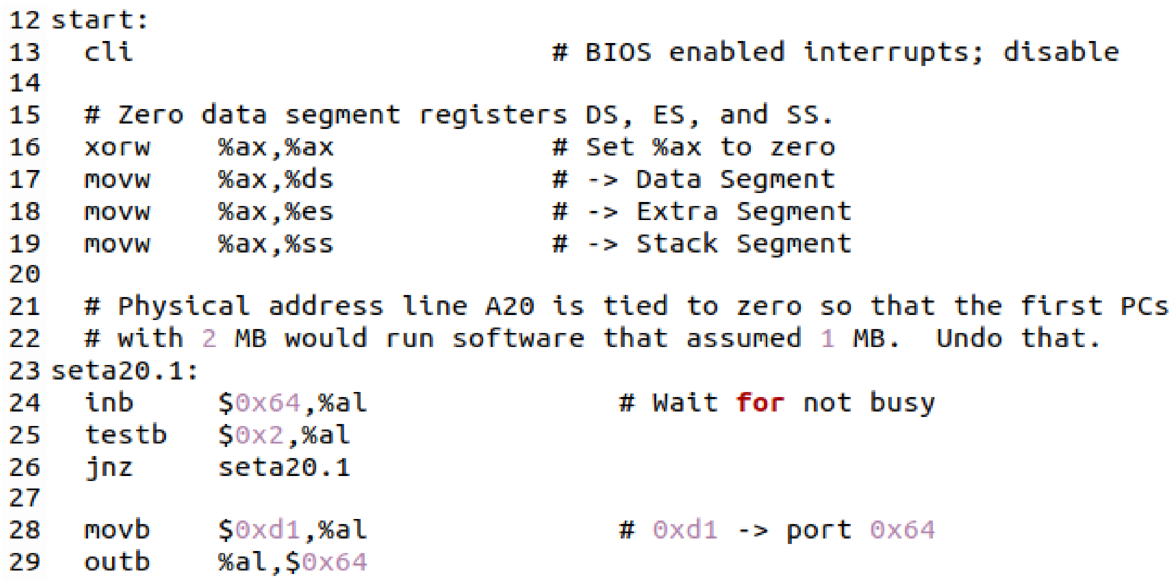
Q3.

Comparison of the original boot loader source code, the disassembly block in bootblock.asm and gdb:-



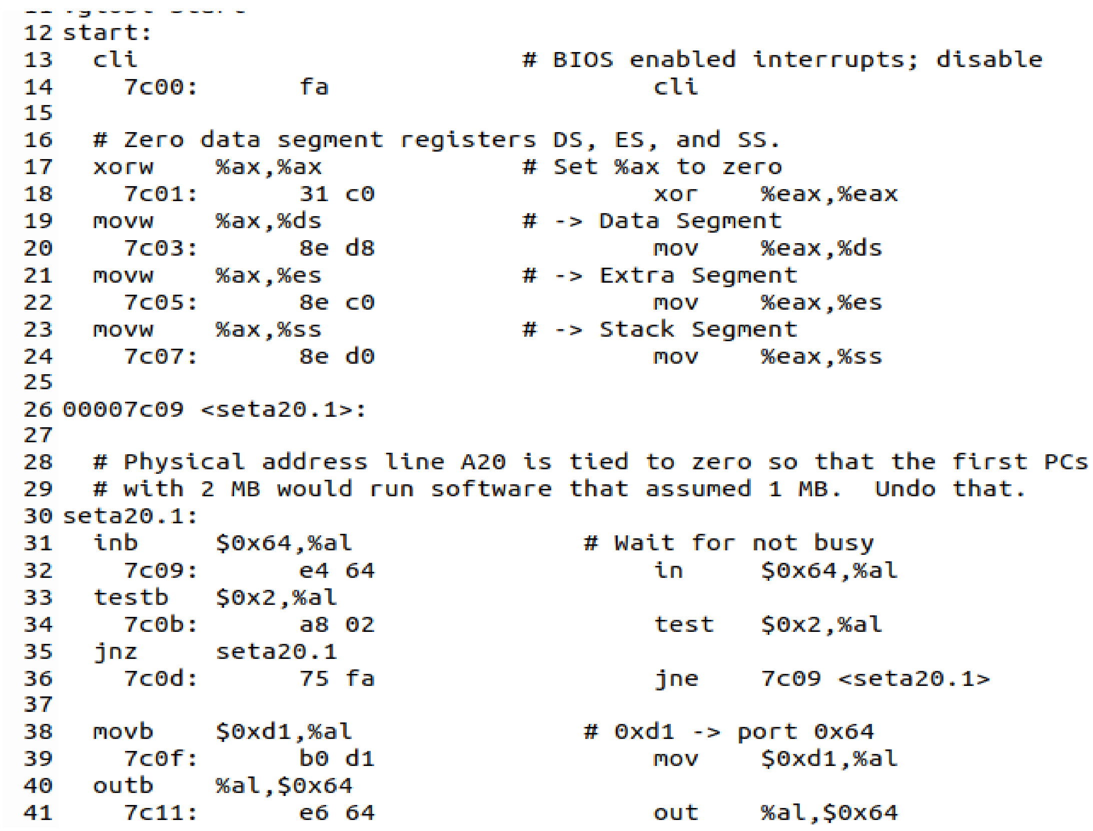
```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/10i 0x7c00
=> 0x7c00:      cli
   0x7c01:      xor    %eax,%eax
   0x7c03:      mov    %eax,%ds
   0x7c05:      mov    %eax,%es
   0x7c07:      mov    %eax,%ss
   0x7c09:      in     $0x64,%al
   0x7c0b:      test   $0x2,%al
   0x7c0d:      jne    0x7c09
   0x7c0f:      mov    $0xd1,%al
   0x7c11:      out    %al,$0x64
(gdb) 
```

Fig 3.1: Showing the first 10 instructions' disassembly from 0x7c00 in Gdb



```
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64
```

Fig 3.2: Showing The original boot loader source code from bootasm.s



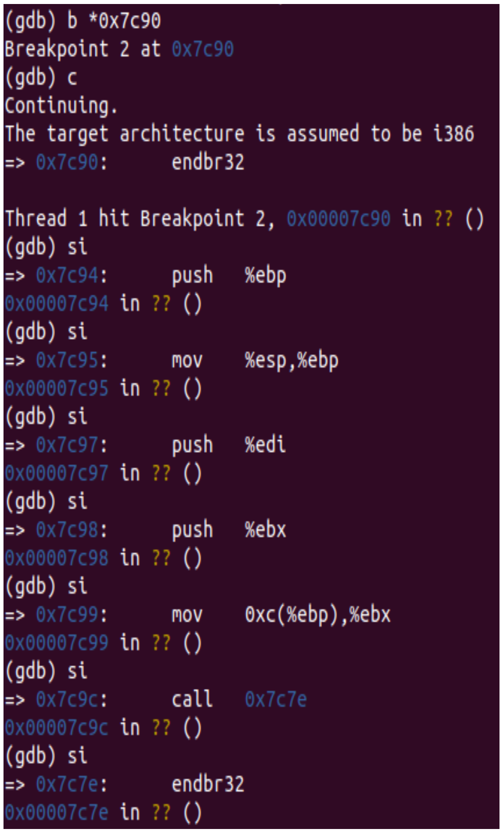
```
start:
  cli                         # BIOS enabled interrupts; disable
    7c00:       fa                      cli

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
    7c01:       31 c0                   xor    %eax,%eax
  movw    %ax,%ds             # -> Data Segment
    7c03:       8e d8                   mov    %eax,%ds
  movw    %ax,%es             # -> Extra Segment
    7c05:       8e c0                   mov    %eax,%es
  movw    %ax,%ss             # -> Stack Segment
    7c07:       8e d0                   mov    %eax,%ss

00007c09 <seta20.1>:

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
    7c09:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c0b:       a8 02                   test   $0x2,%al
  jnz     seta20.1
    7c0d:       75 fa                   jne    7c09 <seta20.1>

  movb    $0xd1,%al               # 0xd1 -> port 0x64
    7c0f:       b0 d1                   mov    $0xd1,%al
  outb    %al,$0x64
    7c11:       e6 64                   out    %al,$0x64
```

Fig 3.3: Showing the disassembly in bootblock.asm

After comparing these 3 images for the first 10 instructions we see that there is no difference among the instructions except for the way they are written.

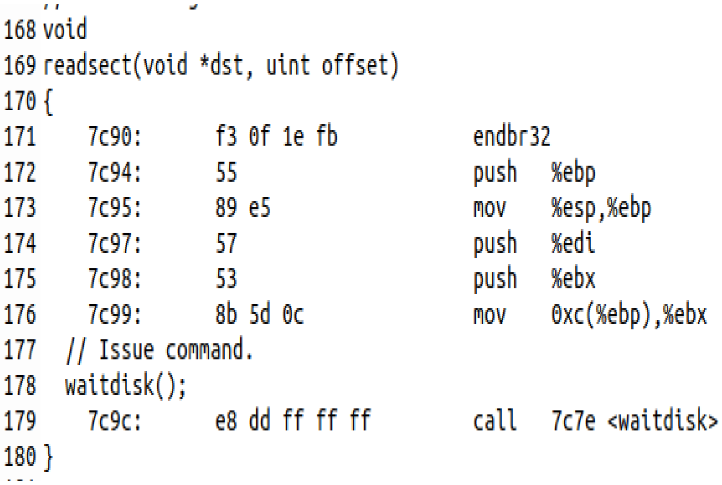
The readsect starts from 0x7c90(mentioned in bootblock.asm), so we can start executing the gdb from that instruction and can find the assembly code for readsect using gdb. I am also attaching the assembly code from bootblock.asm and we can clearly see it's the same assembly code which we got from gdb:-

The below 3 figures show the exact assembly code of readsect() in gdb and bootblock.asm



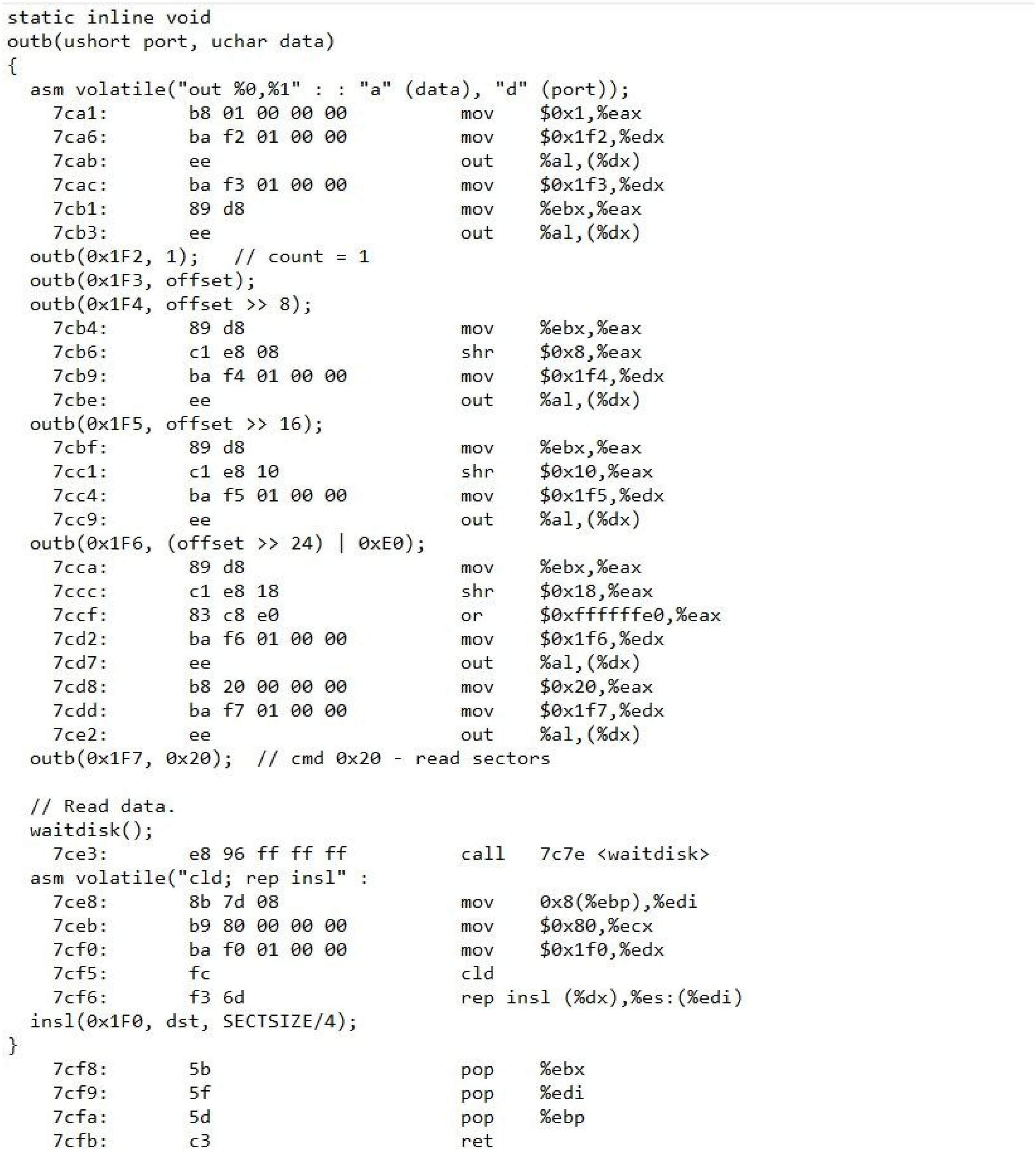
```
(gdb) b *0x7c90
Breakpoint 2 at 0x7c90
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7c90:      endbr32

Thread 1 hit Breakpoint 2, 0x00007c90 in ?? ()
(gdb) si
=> 0x7c94:      push   %ebp
0x00007c94 in ?? ()
(gdb) si
=> 0x7c95:      mov    %esp,%ebp
0x00007c95 in ?? ()
(gdb) si
=> 0x7c97:      push   %edi
0x00007c97 in ?? ()
(gdb) si
=> 0x7c98:      push   %ebx
0x00007c98 in ?? ()
(gdb) si
=> 0x7c99:      mov    0xc(%ebp),%ebx
0x00007c99 in ?? ()
(gdb) si
=> 0x7c9c:      call   0x7c7e
0x00007c9c in ?? ()
(gdb) si
=> 0x7c7e:      endbr32
0x00007c7e in ?? ()
```

Fig 3.4: first few instructions of the assembly code of readsect() using gdb



```
void
readsect(void *dst, uint offset)
{
    7c90:      f3 0f 1e fb           endbr32
    7c94:      55                    push   %ebp
    7c95:      89 e5                 mov    %esp,%ebp
    7c97:      57                    push   %edi
    7c98:      53                    push   %ebx
    7c99:      8b 5d 0c              mov    0xc(%ebp),%ebx
  // Issue command.
  waitdisk();
    7c9c:      e8 dd ff ff ff        call   7c7e <waitdisk>
}
```

Fig 3.5.1: First half of assembly code of readsect inside bootblock.asm



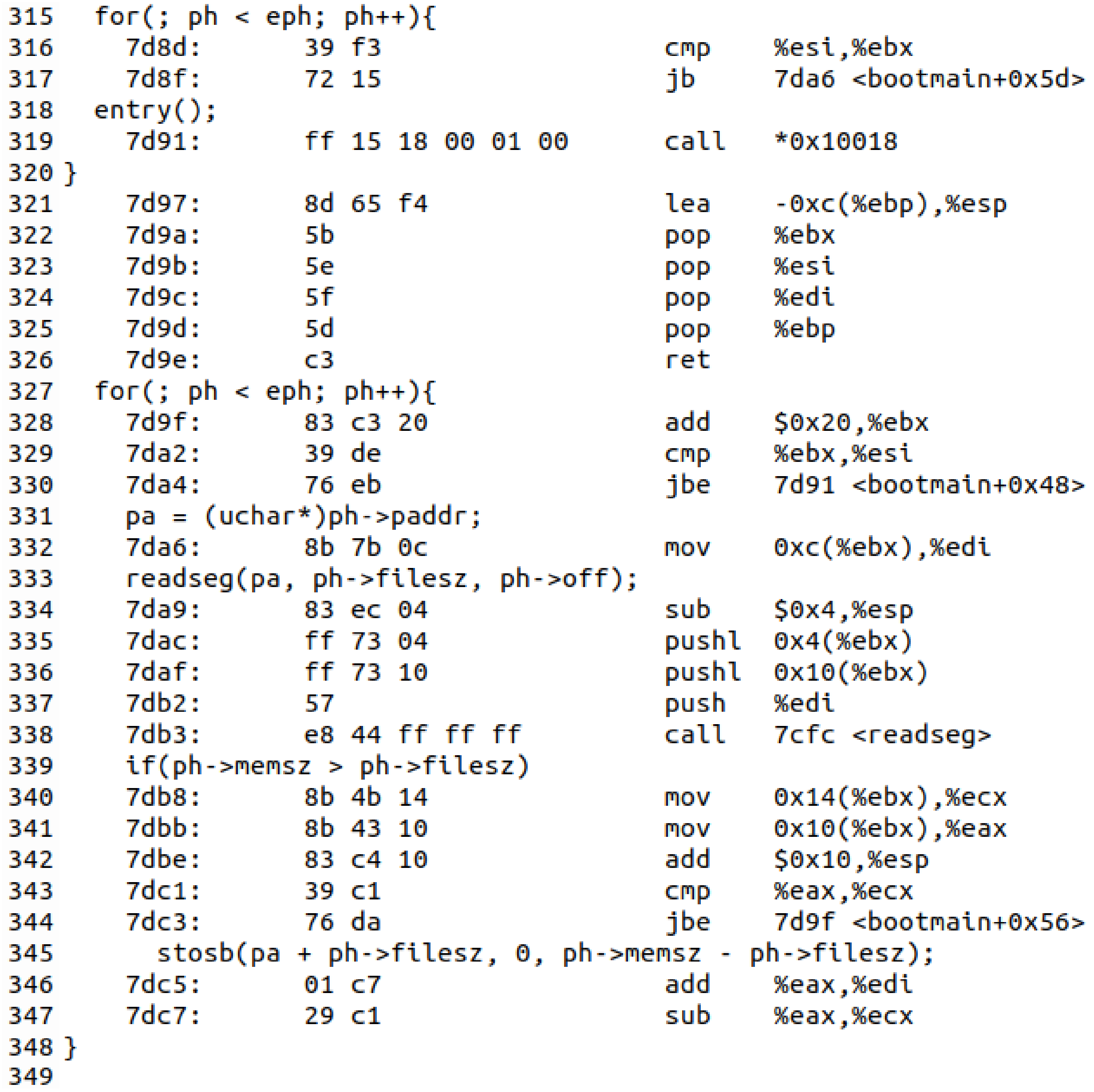
```
static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
    7ca1:       b8 01 00 00 00          mov    $0x1,%eax
    7ca6:       ba f2 01 00 00          mov    $0x1f2,%edx
    7cab:       ee                      out    %al,(%dx)
    7cac:       ba f3 01 00 00          mov    $0x1f3,%edx
    7cb1:       89 d8                   mov    %ebx,%eax
    7cb3:       ee                      out    %al,(%dx)
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
    7cb4:       89 d8                   mov    %ebx,%eax
    7cb6:       c1 e8 08                shr    $0x8,%eax
    7cb9:       ba f4 01 00 00          mov    $0x1f4,%edx
    7cbe:       ee                      out    %al,(%dx)
  outb(0x1F5, offset >> 16);
    7cbf:       89 d8                   mov    %ebx,%eax
    7cc1:       c1 e8 10                shr    $0x10,%eax
    7cc4:       ba f5 01 00 00          mov    $0x1f5,%edx
    7cc9:       ee                      out    %al,(%dx)
  outb(0x1F6, (offset >> 24) | 0xE0);
    7cca:       89 d8                   mov    %ebx,%eax
    7ccc:       c1 e8 18                shr    $0x18,%eax
    7ccf:       83 c8 e0                or     $0xffffffe0,%eax
    7cd2:       ba f6 01 00 00          mov    $0x1f6,%edx
    7cd7:       ee                      out    %al,(%dx)
    7cd8:       b8 20 00 00 00          mov    $0x20,%eax
    7cdd:       ba f7 01 00 00          mov    $0x1f7,%edx
    7ce2:       ee                      out    %al,(%dx)
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
    7ce3:       e8 96 ff ff ff          call   7c7e <waitdisk>
  asm volatile("cld; rep insl" :
    7ce8:       8b 7d 08                mov    0x8(%ebp),%edi
    7ceb:       b9 80 00 00 00          mov    $0x80,%ecx
    7cf0:       ba f0 01 00 00          mov    $0x1f0,%edx
    7cf5:       fc                      cld
    7cf6:       f3 6d                   rep insl (%dx),%es:(%edi)
  insl(0x1F0, dst, SECTSIZE/4);
}
    7cf8:       5b                      pop    %ebx
    7cf9:       5f                      pop    %edi
    7cfa:       5d                      pop    %ebp
    7cfb:       c3                      ret
```

Fig 3.5.2: Showing the next half of assembly code of readsect in bootblock.asm

In the below figure 3.6 of bootblock.asm, line number 316 is the instruction where the for loop begins and line number 330 is the end of the for loop.
Reason:- The for loop starts at 316 and compares ph and eph to check that ph is less than eph and then at line 317 it jumps to line 332 to start the execution and after completion of one cycle of the for loop it jumps to line 328 and increments ph by one

and in line 329 it compares ph and eph and then if equal then it jumps out of the for loop in line 330, and jumps to line 319. So line 319 would be executed when the for loop ends.

Begin of for loop: **7d8d: 39 f3   cmp %esi,%ebx**
End of for loop: **7da4:   76 eb   jbe   7d91 <bootmain+0x48>**
First inst. After the end of for loop: **7d91:   ff 15 18 00 01 00   call   *0x10018**



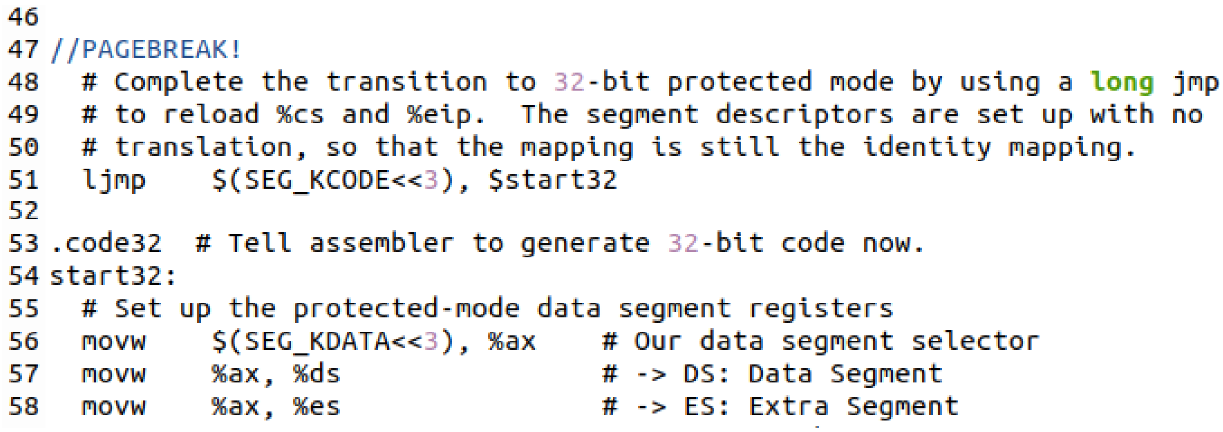
```
315   for(; ph < eph; ph++){
316     7d8d:       39 f3                   cmp    %esi,%ebx
317     7d8f:       72 15                   jb     7da6 <bootmain+0x5d>
318   entry();
319     7d91:       ff 15 18 00 01 00       call   *0x10018
320 }
321     7d97:       8d 65 f4                lea    -0xc(%ebp),%esp
322     7d9a:       5b                      pop    %ebx
323     7d9b:       5e                      pop    %esi
324     7d9c:       5f                      pop    %edi
325     7d9d:       5d                      pop    %ebp
326     7d9e:       c3                      ret
327   for(; ph < eph; ph++){
328     7d9f:       83 c3 20                add    $0x20,%ebx
329     7da2:       39 de                   cmp    %ebx,%esi
330     7da4:       76 eb                   jbe    7d91 <bootmain+0x48>
331     pa = (uchar*)ph->paddr;
332     7da6:       8b 7b 0c                mov    0xc(%ebx),%edi
333     readseg(pa, ph->filesz, ph->off);
334     7da9:       83 ec 04                sub    $0x4,%esp
335     7dac:       ff 73 04                pushl  0x4(%ebx)
336     7daf:       ff 73 10                pushl  0x10(%ebx)
337     7db2:       57                      push   %edi
338     7db3:       e8 44 ff ff ff          call   7cfc <readseg>
339     if(ph->memsz > ph->filesz)
340     7db8:       8b 4b 14                mov    0x14(%ebx),%ecx
341     7dbb:       8b 43 10                mov    0x10(%ebx),%eax
342     7dbe:       83 c4 10                add    $0x10,%esp
343     7dc1:       39 c1                   cmp    %eax,%ecx
344     7dc3:       76 da                   jbe    7d9f <bootmain+0x56>
345       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
346     7dc5:       01 c7                   add    %eax,%edi
347     7dc7:       29 c1                   sub    %eax,%ecx
348 }
349
```

Fig 3.6: Showing the bootblock.asm code for the for loop which reads the sectors of kernel from the disk.

a) Line 56 in fig 3.7 is the first instruction which is executed in 32 bits: **movw $(SEG_KDATA<<3), %ax**.

And line 51 is the instruction which caused this change from 16 to 32 bits:

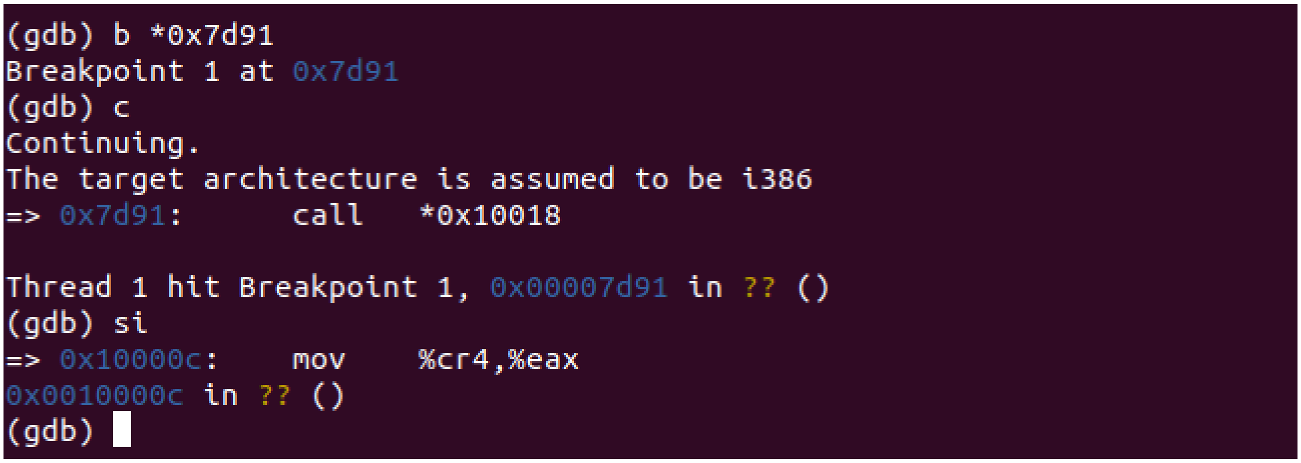
ljmp $(SEG_KCODE<<3), $start32



```
46
47 //PAGEBREAK!
48   # Complete the transition to 32-bit protected mode by using a long jmp
49   # to reload %cs and %eip.  The segment descriptors are set up with no
50   # translation, so that the mapping is still the identity mapping.
51   ljmp    $(SEG_KCODE<<3), $start32
52
53 .code32  # Tell assembler to generate 32-bit code now.
54 start32:
55   # Set up the protected-mode data segment registers
56   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
57   movw    %ax, %ds                # -> DS: Data Segment
58   movw    %ax, %es                # -> ES: Extra Segment
```

Fig 3.7: Showing the point at which the processor started executing 32 bits code and the instruction which caused this to happen

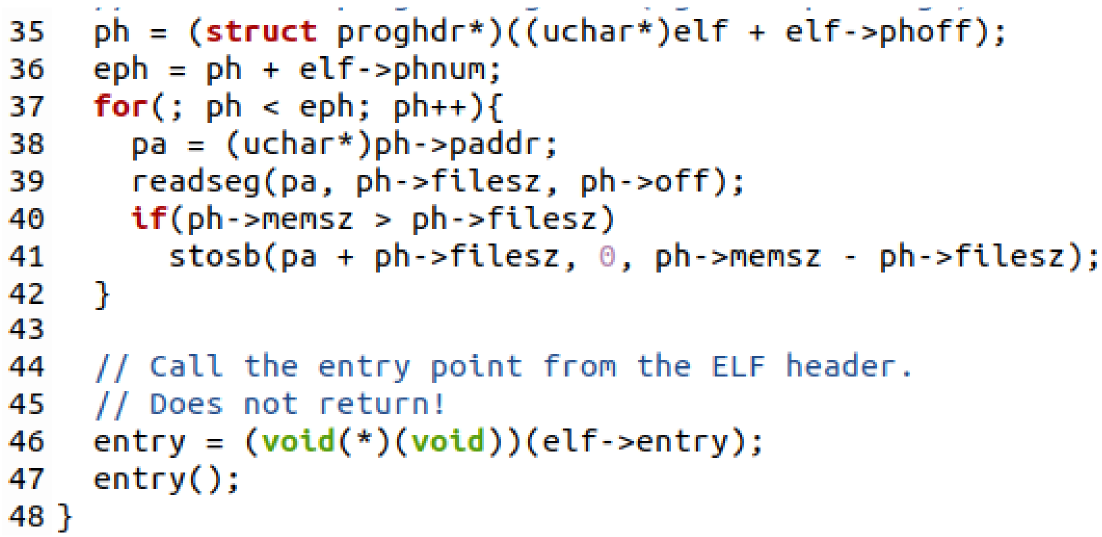
b) **entry()** is the last instruction executed by the boot loader and the bootblock.asm instruction corresponding to entry() is: **0x7d91: call *0x10018**.
And the first instruction of the kernel is: **0x10000c: mov %cr4,%eax**



```
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) 
```

Fig 3.8: Showing the last instruction of the boot loader and the first instruction of the kernel

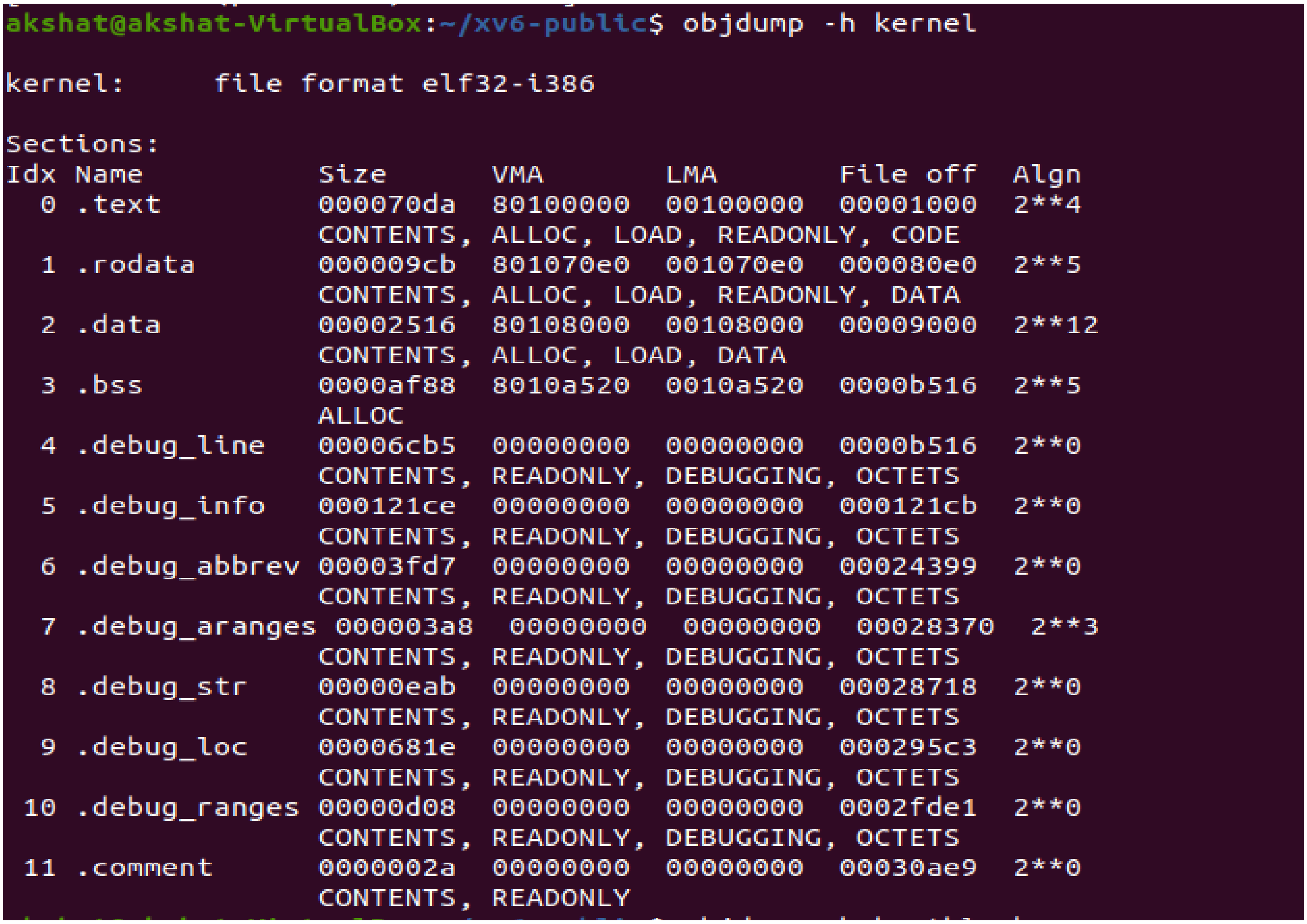
c) The boot loader decides how many sectors it must read by using the information available in the ELF header file, which was loaded previously. elf is a pointer pointing to this header file. Initially the first sector(512 bytes) of the kernel is loaded into the main memory which contains this header file. A variable ph points to the first program header row and another variable is maintained which points to the last program header row. And finally we iterate through the program header rows and just simply read the segment corresponding to each program header table's row. Figure 3.9 shows the code of bootmain.c which corresponds to all of this:-



```c
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

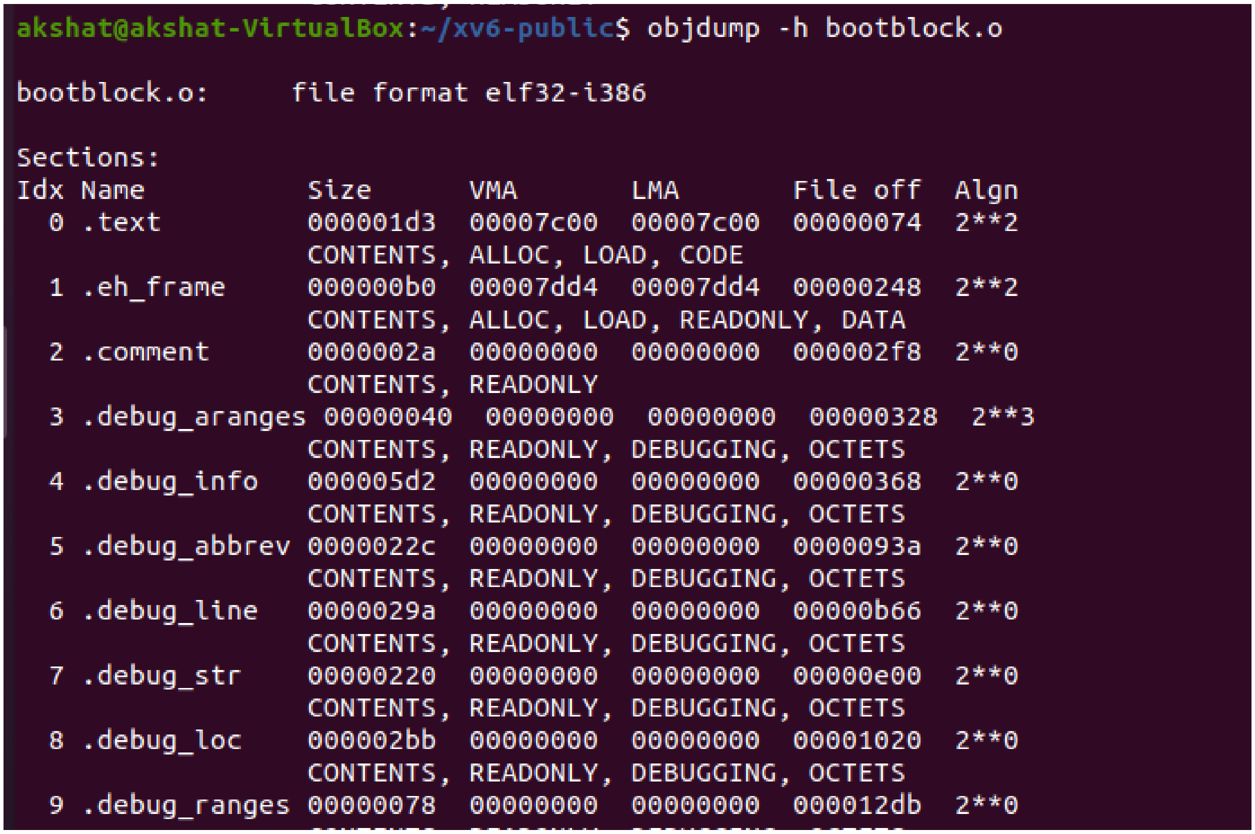
Fig 3.9: Showing the section of code from bootmain.c which is responsible for loading the kernel

Q4.



```
akshat@akshat-VirtualBox:~/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000070da  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801070e0  001070e0  000080e0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000af88  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006cb5  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   000121ce  00000000  00000000  000121cb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00003fd7  00000000  00000000  00024399  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003a8  00000000  00000000  00028370  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000eab  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295c3  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fde1  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030ae9  2**0
                  CONTENTS, READONLY
```

Fig 4.1: Showing the different sections of the kernel along with their fields



```
akshat@akshat-VirtualBox:~/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001d3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dd4  00007dd4  00000248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000002a  00000000  00000000  000002f8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000328  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   000005d2  00000000  00000000  00000368  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000022c  00000000  00000000  0000093a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   0000029a  00000000  00000000  00000b66  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000220  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  00001020  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012db  2**0
```

Fig 4.2: Showing the different sections of the bootblock.o file along with their fields

Explanation of the fields of the sections:

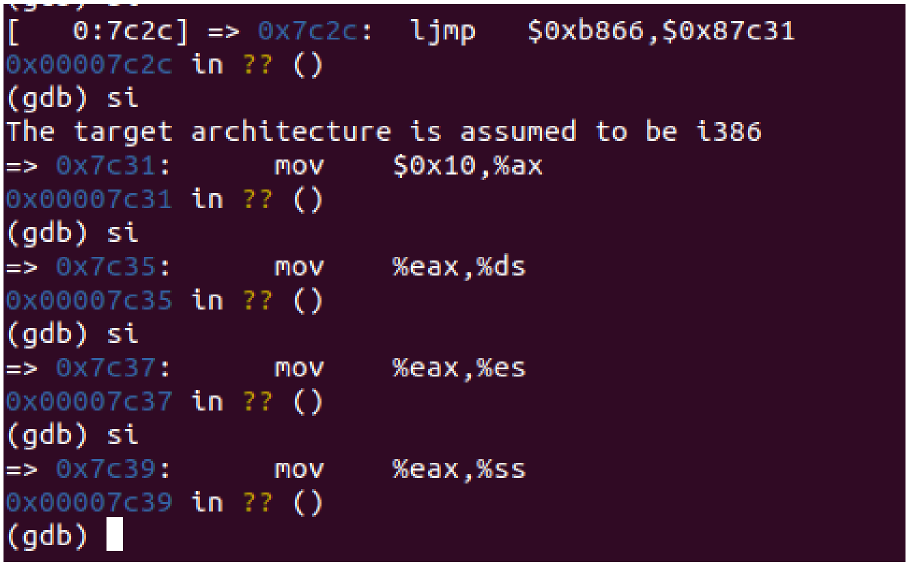
a) Name: It is the name of the section(program header row)
b) Size: It is the size of the section
c) VMA: It stands for virtual memory address which basically consists of the link address of the corresponding section.
d) LMA: It stands for load memory address which basically consists of the load address of the corresponding section.
e) File off: It is the offset of this section from the very start of the file.
f) Algn: It is the alignment of the section.

Q5.

The first instruction which would break if link address is wrongly provided would be:

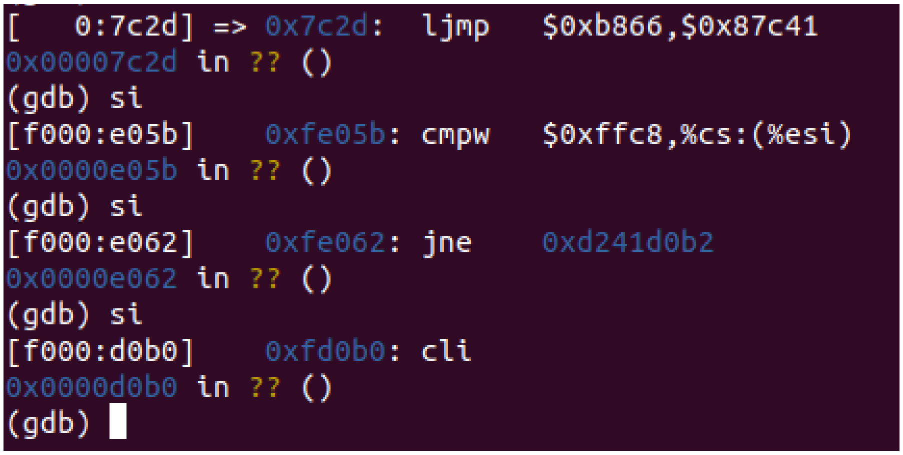
ljmp $(SEG_KCODE<<3), $start32

I have changed the link address to 0x7c0f in the MakeFile. Before the instruction ljmp $0xb866,$0x87c31, all instructions were in the correct order but after this instruction wrong instructions started executing.



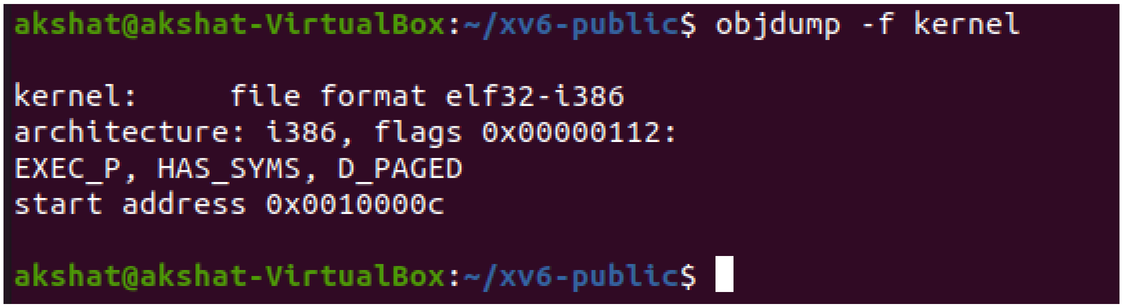
```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
(gdb) si
=> 0x7c37:      mov    %eax,%es
0x00007c37 in ?? ()
(gdb) si
=> 0x7c39:      mov    %eax,%ss
0x00007c39 in ?? ()
(gdb) 
```

Fig 5.1: Showing the correct sequence of instructions after 7c2c ljmp instruction before changing the link address



```
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c41
0x00007c2d in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:d0b0]    0xfd0b0: cli
0x0000d0b0 in ?? ()
(gdb) 
```

Fig 5.2: Showing the incorrect sequence of instructions after 7c2d ljmp instruction after changing the link address

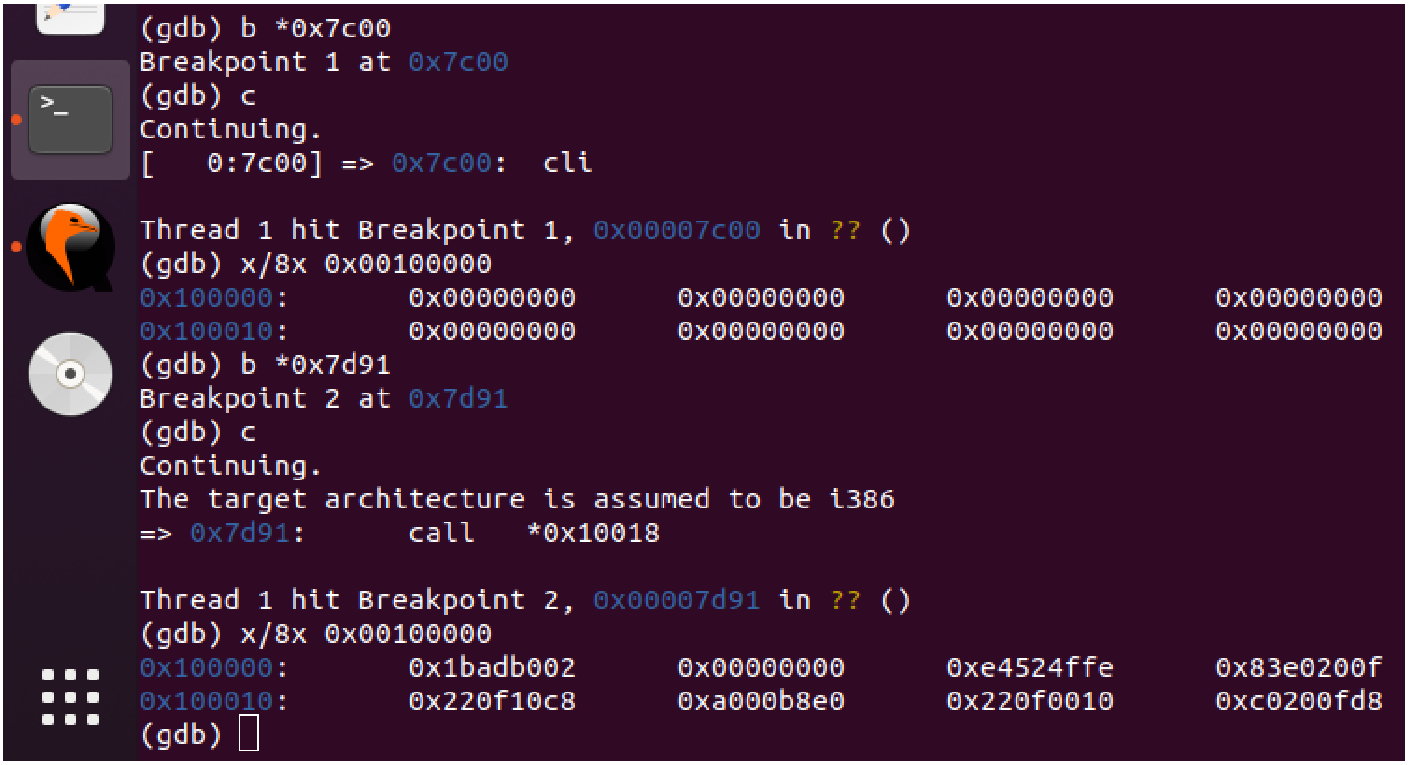


```
akshat@akshat-VirtualBox:~/xv6-public$ objdump -f kernel

kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

akshat@akshat-VirtualBox:~/xv6-public$ 
```

Fig 5.3: Showing the objdump -f kernel command with wrong link address with the entry point as 0x0010000c

Q6.

When we run x/8x 0x00100000 from the address where the boot loader is loaded i.e. 0x7c00 then we get all the 8 words as 0x00000000 because 0x00100000 is an address which belongs to the kernel's section and as the kernel is still not loaded hence all the words are returned as zero because in xv6 uninitialised values are set to 0 by default. But when we load the kernel completely and then run the same command from the address where the boot loader enters the kernel i.e. 0x7d91 then we get the correct data in the 8 consecutive words starting from the address 0x00100000. We can also confirm this fact by seeing the output of gdb which is provided in the below image:-



```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0xa000b8e0      0x220f0010      0xc0200fd8
(gdb) 
```

Fig 6.1: Showing the execution of the command x/8x 0x00100000 before and after loading the kernel into memory