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:

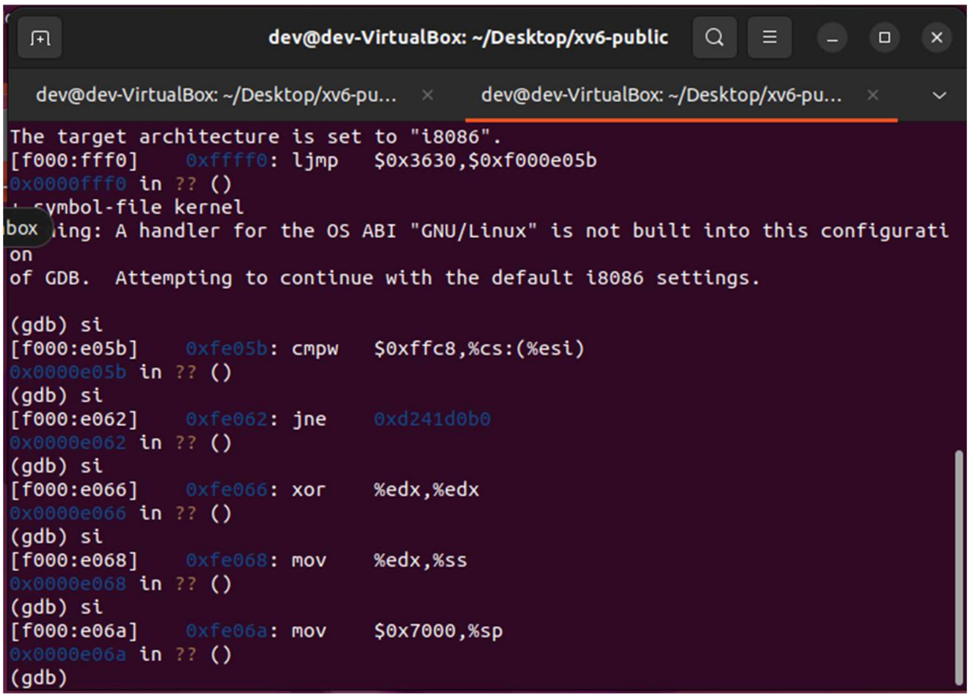
```c
// Simple inline assembly example
//
#include <stdio.h>
int
main(int argc, char **argv)
{
        int x = 1;
        printf("Hello x = %d\n", x);

        // in-line assembly code to increment
        // the value of x by 1
        __asm__ ( "addl %%ebx, %%eax;"
        : "=a" (x)
        : "a" (x), "b" (1) );

        printf("Hello x = %d after increment\n", x);

        if(x == 2){
                printf("OK\n");
        }
        else{
                printf("ERROR\n");
        }
}
```

Output:

```
dev@dev-VirtualBox:~/Desktop/OS Lab$ cd Assignment01/
dev@dev-VirtualBox:~/Desktop/OS Lab/Assignment01$ ls
a.out  ex1.c  ex4.c
dev@dev-VirtualBox:~/Desktop/OS Lab/Assignment01$ gcc ex1.c
dev@dev-VirtualBox:~/Desktop/OS Lab/Assignment01$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
dev@dev-VirtualBox:~/Desktop/OS Lab/Assignment01$
```

<u>Exercise 2</u>. 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

<u>Exercise 3.</u>

Answer:

The code for readsect() is given below-

```
// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}
```

The assembly code for readsect() is given below-

```
00007c8c <readsect>:

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
    7c8c:       55                      push   %ebp
    7c8d:       89 e5                   mov    %esp,%ebp
    7c8f:       57                      push   %edi
    7c90:       53                      push   %ebx
    7c91:       8b 5d 0c                mov    0xc(%ebp),%ebx
  // Issue command.
  waitdisk();
    7c94:       e8 e5 ff ff ff          call   7c7e <waitdisk>
}

```

```
static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
    7c99:       b8 01 00 00 00          mov    $0x1,%eax
    7c9e:       ba f2 01 00 00          mov    $0x1f2,%edx
    7ca3:       ee                      out    %al,(%dx)
    7ca4:       ba f3 01 00 00          mov    $0x1f3,%edx
    7ca9:       89 d8                   mov    %ebx,%eax
    7cab:       ee                      out    %al,(%dx)
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
    7cac:       89 d8                   mov    %ebx,%eax
    7cae:       c1 e8 08                shr    $0x8,%eax
    7cb1:       ba f4 01 00 00          mov    $0x1f4,%edx
    7cb6:       ee                      out    %al,(%dx)
  outb(0x1F5, offset >> 16);
    7cb7:       89 d8                   mov    %ebx,%eax
    7cb9:       c1 e8 10                shr    $0x10,%eax
    7cbc:       ba f5 01 00 00          mov    $0x1f5,%edx
    7cc1:       ee                      out    %al,(%dx)
  outb(0x1F6, (offset >> 24) | 0xE0);
    7cc2:       89 d8                   mov    %ebx,%eax
    7cc4:       c1 e8 18                shr    $0x18,%eax
    7cc7:       83 c8 e0                or     $0xffffffe0,%eax
    7cca:       ba f6 01 00 00          mov    $0x1f6,%edx
    7ccf:       ee                      out    %al,(%dx)
    7cd0:       b8 20 00 00 00          mov    $0x20,%eax
    7cd5:       ba f7 01 00 00          mov    $0x1f7,%edx
    7cda:       ee                      out    %al,(%dx)
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
    7cdb:       e8 9e ff ff ff          call   7c7e <waitdisk>
  asm volatile("cld; rep insl" :
    7ce0:       8b 7d 08                mov    0x8(%ebp),%edi
    7ce3:       b9 80 00 00 00          mov    $0x80,%ecx
    7ce8:       ba f0 01 00 00          mov    $0x1f0,%edx
    7ced:       fc                      cld
    7cee:       f3 6d                   rep insl (%dx),%es:(%edi)
  insl(0x1F0, dst, SECTSIZE/4);
}
```

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

```
  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);
  }
```

The first instruction of this for loop is:

```
    7d7d:       39 f3                   cmp    %esi,%ebx
```

The Last Instruction of this for loop is:

```
    7d94:       76 eb                   jbe    7d81 <bootmain+0x44>
```

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

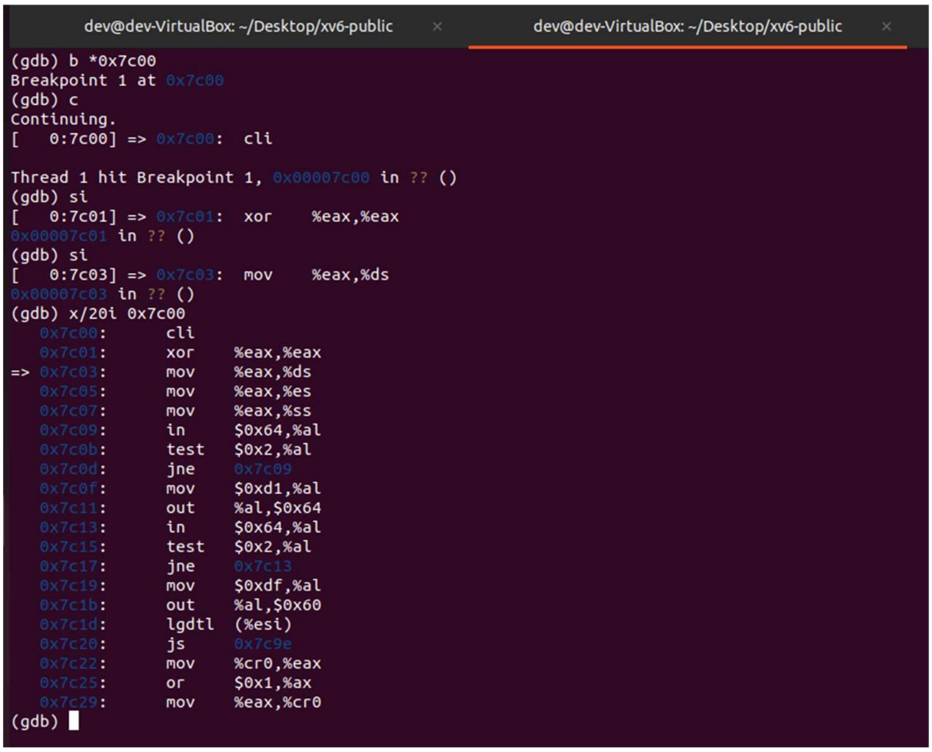
```
    7d81:       ff 15 18 00 01 00       call   *0x10018
```

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

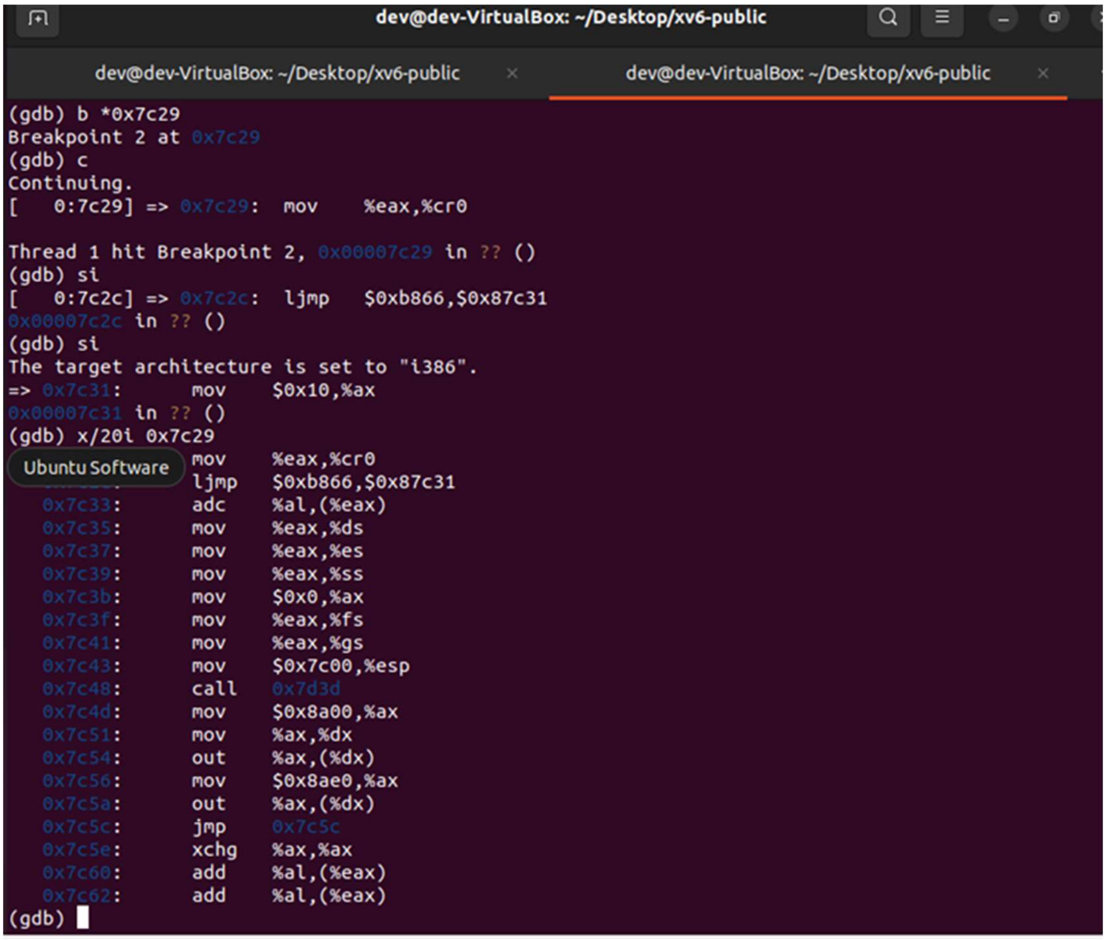
dev@dev-VirtualBox: ~/Desktop/xv6-public   dev@dev-VirtualBox: ~/Desktop/xv6-public

```
(gdb) b *0x7d81
Breakpoint 3 at 0x7d81
(gdb) c
Continuing.
=> 0x7d81:      call   *0x10018

Thread 1 hit Breakpoint 3, 0x00007d81 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x0010000f in ?? ()
(gdb) si
=> 0x100012:    mov    %eax,%cr4
0x00100012 in ?? ()
(gdb) si
=> 0x100015:    mov    $0x109000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010000,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0x801154d0,%esp
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    mov    $0x80103060,%eax
0x0010002d in ?? ()
(gdb) si
=> 0x100032:    jmp    *%eax
0x00100032 in ?? ()
(gdb) si
=> 0x80103060 <main>:   lea    0x4(%esp),%ecx
main () at main.c:20
20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb) 
```

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



```
dev@dev-VirtualBox: ~/Desktop/xv6-public        dev@dev-VirtualBox: ~/Desktop/xv6-public
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si
[   0:7c01] => 0x7c01:  xor    %eax,%eax
0x00007c01 in ?? ()
(gdb) si
[   0:7c03] => 0x7c03:  mov    %eax,%ds
0x00007c03 in ?? ()
(gdb) x/20i 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
   0x7c13:      in     $0x64,%al
   0x7c15:      test   $0x2,%al
   0x7c17:      jne    0x7c13
   0x7c19:      mov    $0xdf,%al
   0x7c1b:      out    %al,$0x60
   0x7c1d:      lgdtl  (%esi)
   0x7c20:      js     0x7c9e
   0x7c22:      mov    %cr0,%eax
   0x7c25:      or     $0x1,%ax
   0x7c29:      mov    %eax,%cr0
(gdb)
```



```
dev@dev-VirtualBox: ~/Desktop/xv6-public
dev@dev-VirtualBox: ~/Desktop/xv6-public        dev@dev-VirtualBox: ~/Desktop/xv6-public
(gdb) b *0x7c29
Breakpoint 2 at 0x7c29
(gdb) c
Continuing.
[   0:7c29] => 0x7c29:  mov    %eax,%cr0

Thread 1 hit Breakpoint 2, 0x00007c29 in ?? ()
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb) si
The target architecture is set to "i386".
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) x/20i 0x7c29
 Ubuntu Software mov    %eax,%cr0
                 ljmp   $0xb866,$0x87c31
   0x7c33:      adc    %al,(%eax)
   0x7c35:      mov    %eax,%ds
   0x7c37:      mov    %eax,%es
   0x7c39:      mov    %eax,%ss
   0x7c3b:      mov    $0x0,%ax
   0x7c3f:      mov    %eax,%fs
   0x7c41:      mov    %eax,%gs
   0x7c43:      mov    $0x7c00,%esp
   0x7c48:      call   0x7d3d
   0x7c4d:      mov    $0x8a00,%ax
   0x7c51:      mov    %ax,%dx
   0x7c54:      out    %ax,(%dx)
   0x7c56:      mov    $0x8ae0,%ax
   0x7c5a:      out    %ax,(%dx)
   0x7c5c:      jmp    0x7c5c
   0x7c5e:      xchg   %ax,%ax
   0x7c60:      add    %al,(%eax)
   0x7c62:      add    %al,(%eax)
(gdb)
```

b) 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

```
entry();
  7d81:       ff 15 18 00 01 00       call   *0x10018
```

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:

(i) By looking at the first word of memory stored at 0x10018 (by using the command "x/1x 0x10018")
(ii) (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:

(i) By using "x/1i 0x0010000c"
(ii) By looking into kernel.asm

```
dev@dev-VirtualBox: ~/Desktop/xv6-public          dev@dev-VirtualBox: ~/Desktop/xv6-public

(gdb) b *0x7d81
Breakpoint 1 at 0x7d81
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d81:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d81 in ?? ()
(gdb) x/1x 0x10018
0x10018:        0x0010000c
(gdb) x/1i 0x0010000c
   0x10000c:    mov    %cr4,%eax
(gdb)
```

Hence, the first instruction of kernel is:

```
0x10000c:    mov    %cr4,%eax
```

c)

```
// Load each program segment (ignores ph flags).
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();
}
```

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.

<u>Exercise 4</u>. 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

```
dev@dev-VirtualBox:~/Desktop/xv6-public$ gcc pointer.c
cc1: fatal error: pointer.c: No such file or directory
compilation terminated.
dev@dev-VirtualBox:~/Desktop/xv6-public$ gcc pointers.c
dev@dev-VirtualBox:~/Desktop/xv6-public$ ./a.out
1: a = 0x7fff2d3e6530, b = 0x55b4720fd2a0, c = (nil)
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7fff2d3e6530, b = 0x7fff2d3e6534, c = 0x7fff2d3e6531
dev@dev-VirtualBox:~/Desktop/xv6-public$
```

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

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00007188  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801071a0  001071a0  000081a0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000afb0  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006aaf  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   00010e14  00000000  00000000  00011fc5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00004496  00000000  00000000  00022dd9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003b0  00000000  00000000  00027270  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000df4  00000000  00000000  00027620  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 000050b1  00000000  00000000  00028414  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000845  00000000  00000000  0002d4c5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 00000137  00000000  00000000  0002dd0a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .comment      00000026  00000000  00000000  0002de41  2**0
                  CONTENTS, READONLY
```

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.

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

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001c3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dc4  00007dc4  00000238  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      00000026  00000000  00000000  000002e8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000310  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   00000585  00000000  00000000  00000350  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000023c  00000000  00000000  000008d5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   00000283  00000000  00000000  00000b11  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    0000020b  00000000  00000000  00000d94  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line_str 00000046  00000000  00000000  00000f9f  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 0000018d  00000000  00000000  00000fe5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000033  00000000  00000000  00001172  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

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.

<u>Exercise 5</u>. 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:

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

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

dev@dev-VirtualBox:~/Desktop/xv6-public$
```

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:

```
dev@dev-VirtualBox: ~/Desktop/xv6-pu...    dev@dev-VirtualBox: ~/Desktop/xv6-pu...
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) b *0x7c31
Breakpoint 2 at 0x7c31
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7c31:      mov    $0x10,%ax

Thread 1 hit Breakpoint 2, 0x00007c31 in ?? ()
(gdb)
```

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

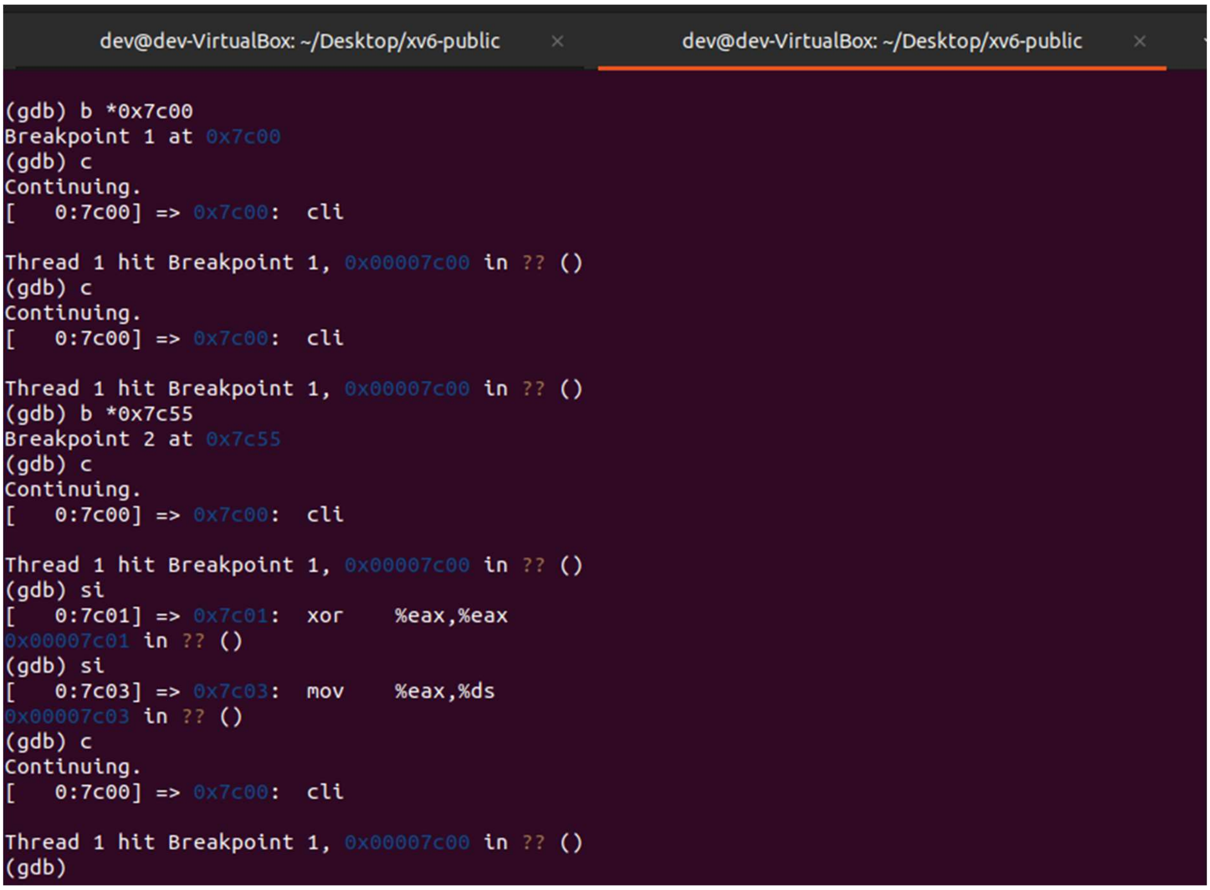
make clean

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.



```
dev@dev-VirtualBox: ~/Desktop/xv6-public        dev@dev-VirtualBox: ~/Desktop/xv6-public
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) b *0x7c55
Breakpoint 2 at 0x7c55
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si
[   0:7c01] => 0x7c01:  xor    %eax,%eax
0x00007c01 in ?? ()
(gdb) si
[   0:7c03] => 0x7c03:  mov    %eax,%ds
0x00007c03 in ?? ()
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(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.

<u>Exercise 6</u>:

Answer:

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

```
dev@dev-VirtualBox: ~/Desktop/xv6-public        dev@dev-VirtualBox: ~/Desktop/xv6-public

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x0010000
0x10000:        0x00000000      0x00000000      0x00000000      0x00000000
0x10010:        0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/8i 0x0010000
   0x10000:     add    %al,(%eax)
   0x10002:     add    %al,(%eax)
   0x10004:     add    %al,(%eax)
   0x10006:     add    %al,(%eax)
   0x10008:     add    %al,(%eax)
   0x1000a:     add    %al,(%eax)
   0x1000c:     add    %al,(%eax)
   0x1000e:     add    %al,(%eax)
(gdb)
```

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

```
dev@dev-VirtualBox: ~/Desktop/xv6-public        dev@dev-VirtualBox: ~/Desktop/xv6-public
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) x/8i 0x00100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0xf,%al
   0x10000d:    and    %ah,%al
   0x10000f:    or     $0x10,%eax
   0x100012:    mov    %eax,%cr4
   0x100015:    mov    $0x109000,%eax
(gdb)
```

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.