SHREY VERMA
190101083
v.shrey@iitg.ac.in

Exercise 1.

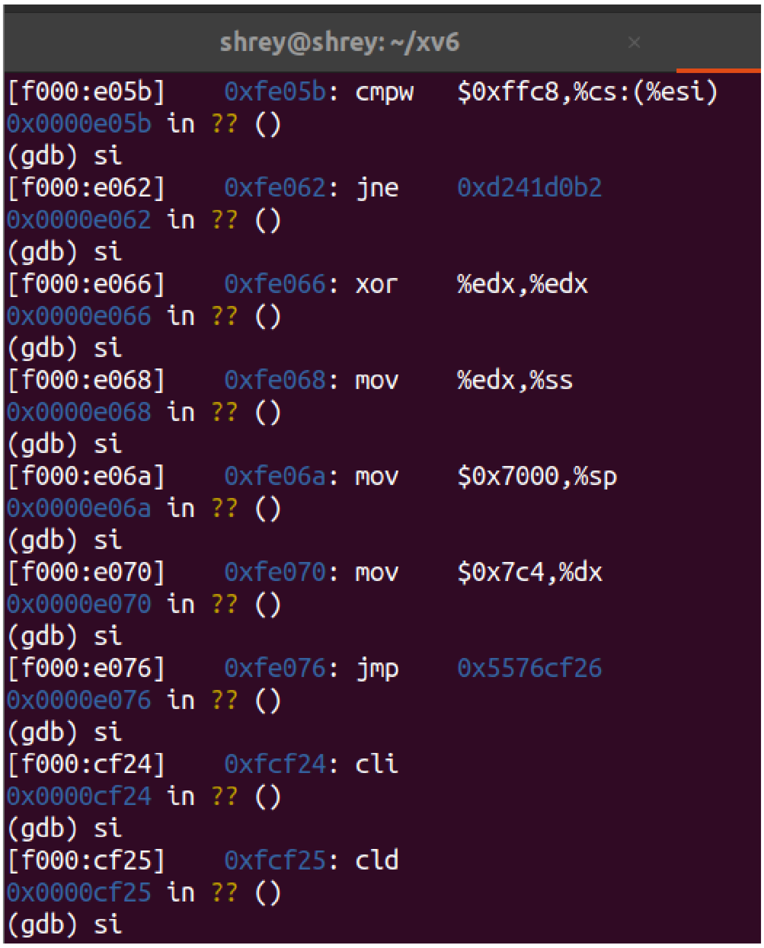
We will use extended inline assembly, as in this we can also specify input/output registers.
Syntax of the same –

```
__asm__ (“assembly code;” : output operands : input operands : clobbered registers);
```

The in-line assembly code for incrementing the value of x by 1 –

```
__asm__ (“inc %%ecx;” : “=c”(x) : “c”(x));
```

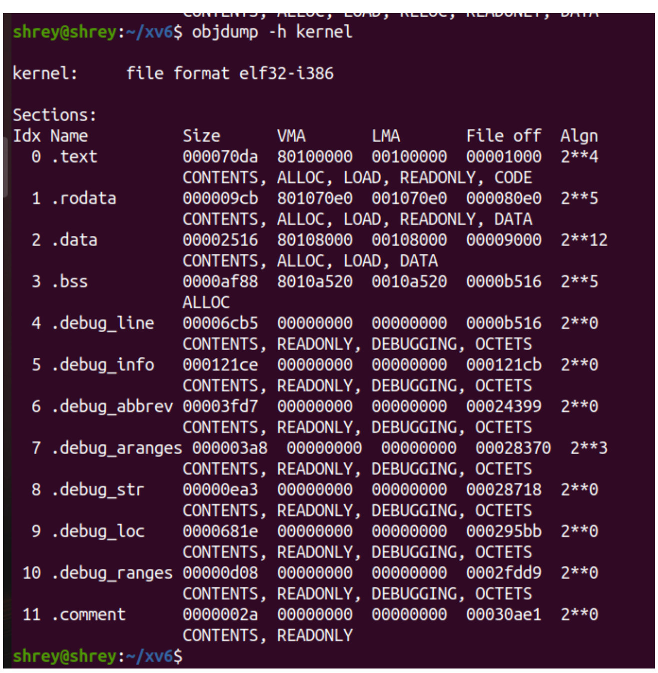
Exercise 2.



1) **cmp** - Comparing two operands
   Here, the first is given directly by register’s address and the second one is given in segmented form with the offset given in memory form.
2) **jne** - Conditional Jump only when not equal, it is run after a *cmp* command.
   Done to verify the correctness of output of previous *cmp* function.
3) **xor** – Takes *xor* of the two operands
   Here *xor* of *edx* and *edx* will be 0 hence set *edx* = 0.
4) **mov** – Moves instruction that moves data between the locations given as operands.
   Here we load the value of *edx* i.e. 0 to *SS* (Stack Segment) Register

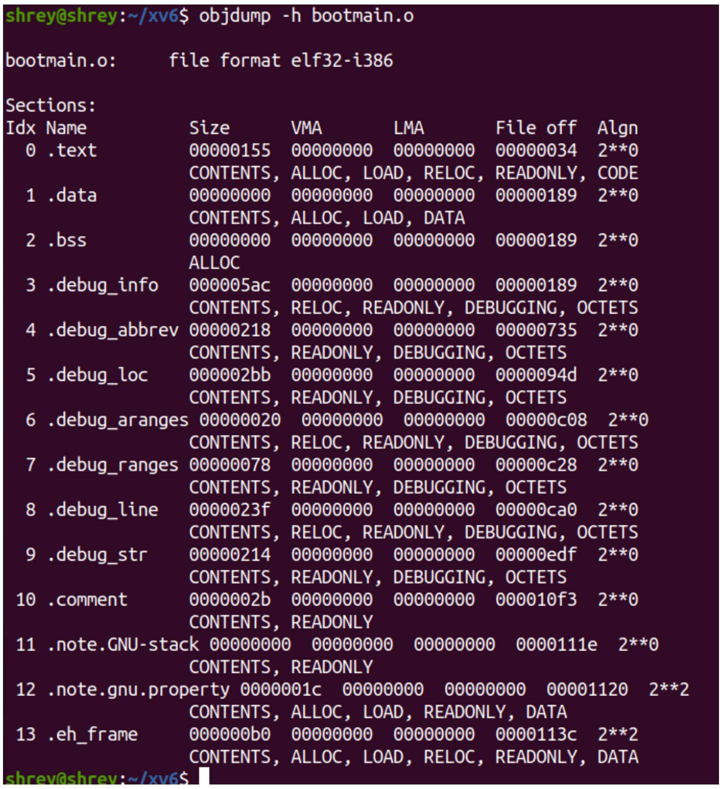
5) Copies value 0x074 in register *sp* (Stack Pointer).
6) Copies the value 0x7c4 in register *dx*.
7) **jmp –** Transfers program control flow to the instruction at the memory location indicated by the operand.
8) **cli –** Clears interrupt flag, affects no other flags. External interrupts disabled at the end of *cli* instruction.
9) **cld -** Clears the direction flag; affects no other flags or registers. Direction flag determines the direction forward/backward of string processing.

Exercise 4.



```
shrey@shrey:~/xv6$ 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    00000ea3  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295bb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fdd9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030ae1  2**0
                  CONTENTS, READONLY
shrey@shrey:~/xv6$
```

Figure 4.1 objdump -h kernel



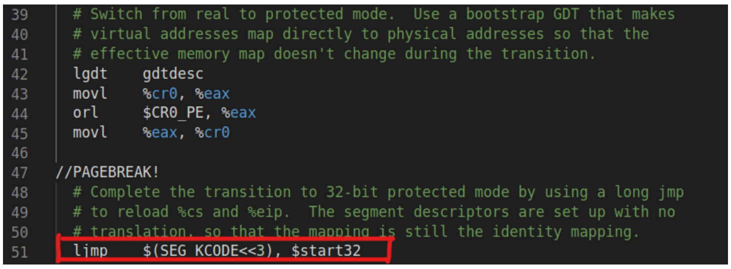
```
shrey@shrey:~/xv6$ objdump -h bootmain.o

bootmain.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000155  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000189  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000189  2**0
                  ALLOC
  3 .debug_info   000005ac  00000000  00000000  00000189  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  4 .debug_abbrev 00000218  00000000  00000000  00000735  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_loc    000002bb  00000000  00000000  0000094d  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_aranges 00000020  00000000  00000000  00000c08  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  7 .debug_ranges 00000078  00000000  00000000  00000c28  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line   0000023f  00000000  00000000  00000ca0  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  9 .debug_str    00000214  00000000  00000000  00000edf  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .comment      0000002b  00000000  00000000  000010f3  2**0
                  CONTENTS, READONLY
 11 .note.GNU-stack 00000000  00000000  00000000  0000111e  2**0
                  CONTENTS, READONLY
 12 .note.gnu.property 0000001c  00000000  00000000  00001120  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 13 .eh_frame     000000b0  00000000  00000000  0000113c  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
shrey@shrey:~/xv6$
```

Figure 4.2 objdump -h bootmain.o

We can observe several sections in the output of these commands. Some of them are –

1. VMA – Link address containing memory address from which section begins to execute.
2. LMA – Link address containing memory address from which section should be loaded. Usually VMA = LMA
3. .text: The program's executable instructions.
4. .rodata: Read-only data, such as ASCII string constants produced by the C compiler. (We will not bother setting up the hardware to prohibit writing, however.)
5. .data: The data section holds the program's initialized data, such as global variables declared with initializers like int x = 5;

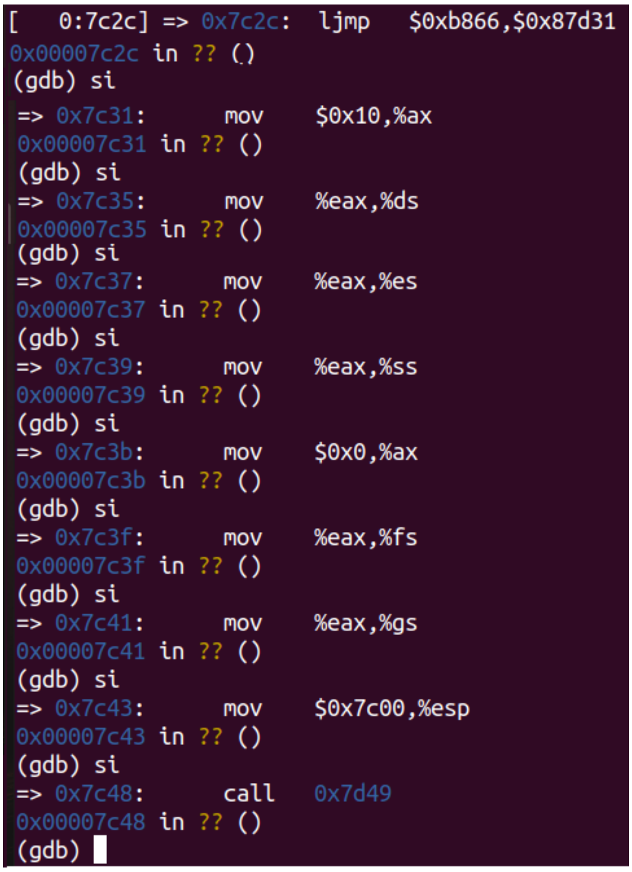
Exercise 5.



```
39  # Switch from real to protected mode.  Use a bootstrap GDT that makes
40  # virtual addresses map directly to physical addresses so that the
41  # effective memory map doesn't change during the transition.
42  lgdt    gdtdesc
43  movl    %cr0, %eax
44  orl     $CR0_PE, %eax
45  movl    %eax, %cr0
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
```

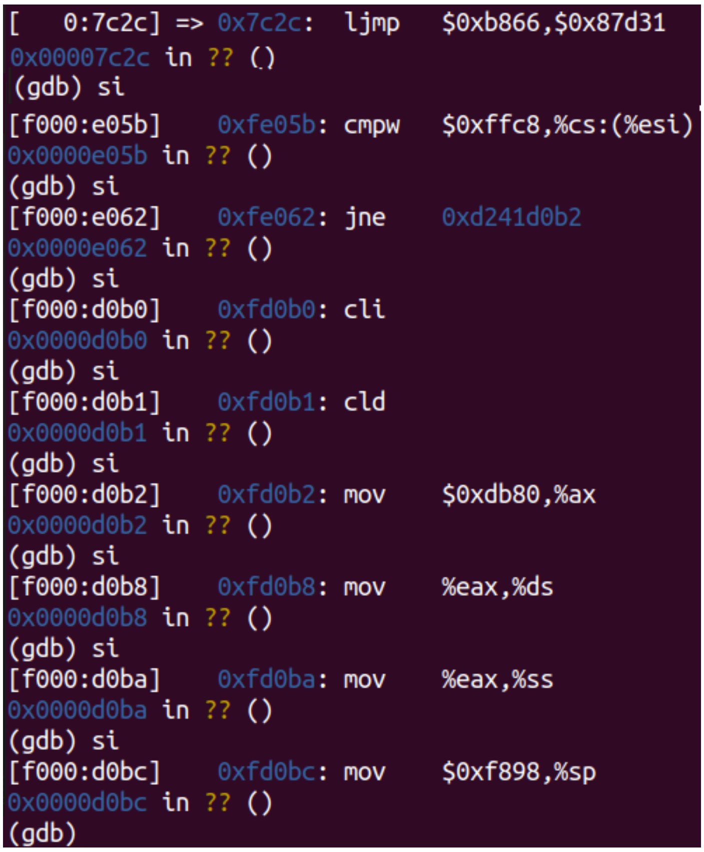
Figure 1 - Transition from 16bit to 32bit

The instruction on the line 51(ljmp) of the `botasm.s` will be the first to break when the wrong address is entered in the makefile.



```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87d31
0x00007c2c in ?? ()
(gdb) si
=> 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) si
=> 0x7c3b:      mov    $0x0,%ax
0x00007c3b in ?? ()
(gdb) si
=> 0x7c3f:      mov    %eax,%fs
0x00007c3f in ?? ()
(gdb) si
=> 0x7c41:      mov    %eax,%gs
0x00007c41 in ?? ()
(gdb) si
=> 0x7c43:      mov    $0x7c00,%esp
0x00007c43 in ?? ()
(gdb) si
=> 0x7c48:      call   0x7d49
0x00007c48 in ?? ()
(gdb)
```

*Figure 3 – Execution with **correct** address*



```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87d31
0x00007c2c 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) si
[f000:d0b1]    0xfd0b1: cld
0x0000d0b1 in ?? ()
(gdb) si
[f000:d0b2]    0xfd0b2: mov    $0xdb80,%ax
0x0000d0b2 in ?? ()
(gdb) si
[f000:d0b8]    0xfd0b8: mov    %eax,%ds
0x0000d0b8 in ?? ()
(gdb) si
[f000:d0ba]    0xfd0ba: mov    %eax,%ss
0x0000d0ba in ?? ()
(gdb) si
[f000:d0bc]    0xfd0bc: mov    $0xf898,%sp
0x0000d0bc in ?? ()
(gdb)
```

*Figure 2 – Execution with **wrong** address*

The correct link address is *0x7c00*. Changes were made in the makefile and changed the link address to *0x7d00*, and then executed the *`make qemu`* command after clearing the old make files using *`make clean`* command.

When GDB was run to compare the difference between the instructions before the instruction

`ljmp $0xb866, $0x87c31` the same instructions were executed. But after this instruction all the instruction executed incorrectly in the wrong version.

```
shrey@shrey: ~/xv6
shrey@shrey:~/xv6$ objdump -f kernel

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

shrey@shrey:~/xv6$
```

Figure - objdump -f kernel Entry Point Address: 0x0010000c

Exercise 6.

```
shrey@shrey: ~/xv6
(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      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb)
```

The kernel is loaded into the main memory starting from the address 0x00100000. When we inspect the given address at the first breakpoint all the words are found to be 0s. This is because before the start of the execution of boot loader there is no useful data at this location.

However, when the data at the same location is observed after the second breakpoint, we find some useful data. This implies that now the kernel has been loaded to the main memory.

Exercise 3.

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/10i $eip
=> 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)
```

Figure 1 - Disassembly of next 10 instructions after breakpoint

We can see in Fig 1 that we set a breakpoint at address 0x7c00, which is the address at which the boot sector starts loading.

Next, we use the GDB x/Ni command to trace further 10 instructions from the current position of the program counter/instruction pointer (denoted as eip).

```
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

seta20.2:
  inb     $0x64,%al                   # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al                   # 0xdf -> port 0x60
  outb    %al,$0x60
```

Figure 2 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

00007c13 <seta20.2>:

seta20.2:
  inb     $0x64,%al                   # Wait for not busy
    7c13: e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c15: a8 02                   test   $0x2,%al
  jnz     seta20.2
    7c17: 75 fa                   jne    7c13 <seta20.2>

  movb    $0xdf,%al                   # 0xdf -> port 0x60
    7c19: b0 df                   mov    $0xdf,%al
  outb    %al,$0x60
    7c1b: e6 60                   out    %al,$0x60
```

Figure 3 bootblock.asm

Figure 3 and 4 show the source code and disassembled instructions in bootasm.s and bootblock.asm respectively. They also represent the first 10 instructions traced after the address 0x7c00.

```
// Read a single sector at offset into dst.
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>
}
```

Figure 4 readsect() in bootblock.asm

```
// 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);
}
```

Figure 5 readsect() in bootmain.c

Let's analyse the readsect() function in both bootmain.c as well as the exact assembly instructions (bootblock.asm) that correspond to each of the statements mentioned in bootblock.asm. We can see that for each line in one there is a corresponding line in another.

```
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   }
```

Figure 6 bootblock.asm loading kernel

Let us have a look at bootblock.asm.

From line number 315 to 319, we are reading the bootloader section from the disk

From line number 321 to 348, we are reading the remaining sectors of the kernel from the disk.

When the bootloader is finished executing, as we can see in line number 319, we make a call to address *0x10018.

So, let us set the 2nd breakpoint at address 0x7d91 in GDB and continue until the breakpoint is reached.

The section of code in bootasm.s (fig 7) shows the transition from the 16 bit real mode to 32 bit protected mode.

All lines of code before line number 42 were executed in 16-bit real mode.

After line number 54 we complete the transition to 32-bit protected mode and all instructions thereafter will be executed in this mode.

```
39    # Switch from real to protected mode.  Use a bootstrap GDT that makes
40    # virtual addresses map directly to physical addresses so that the
41    # effective memory map doesn't change during the transition.
42    lgdt    gdtdesc
43    movl    %cr0, %eax
44    orl     $CR0_PE, %eax
45    movl    %eax, %cr0
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
```

Figure 7 Transition from 16 to 32 bit in bootasm.s

```
shrey@shrey: ~/xv6
(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) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb)
```

Figure 8 Boot loader to kernel transition

In the GDB terminal we can see that when we execute *si* after the last bootloader instruction at 0x7d91 we get the first kernel instruction loaded.

```c
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // 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();
}
```

Figure 9 bootmain() in bootmain.c

Now, we will try to understand how does the bootloader decides how many instructions it has to read to fetch the entire kernel from the disk. When we analyse the bootmain() function in bootmain.c,

We find that the boot-loader runs for a loop from **ph** to **eph** to load the sectors from the kernel. Their values are stored in ELF (Executable and Linkable Format) header.

ph is given by elf -> phoff and **eph** is given by elf -> phnum which determines the total number of iterations.