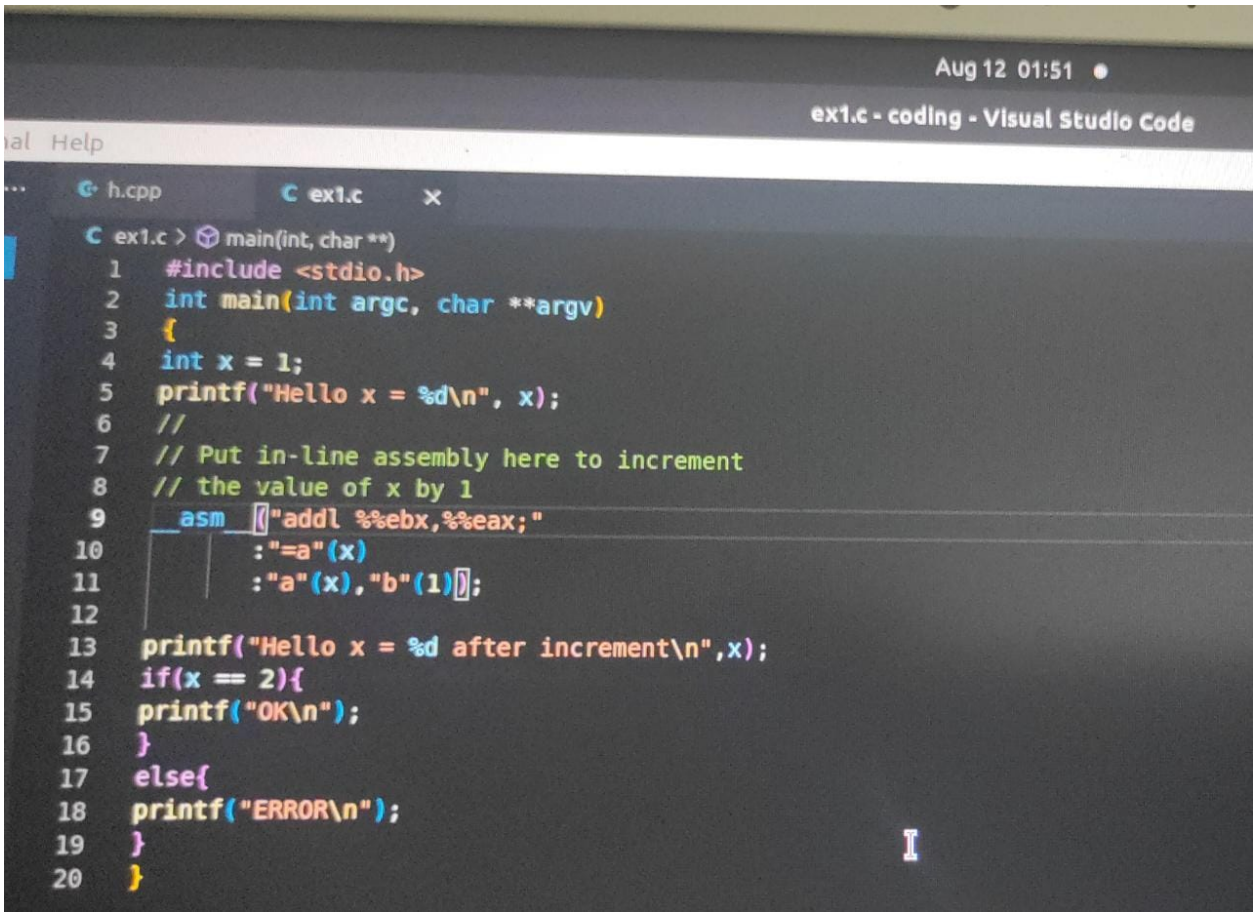
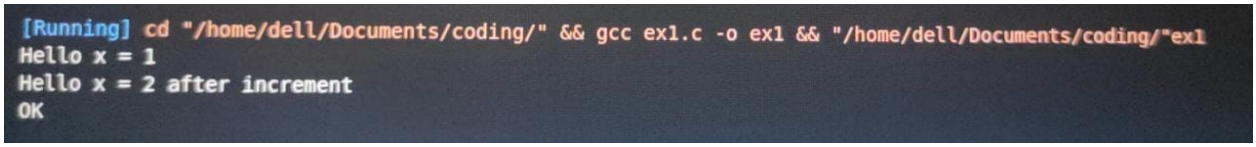
Part A:

Exercise 1

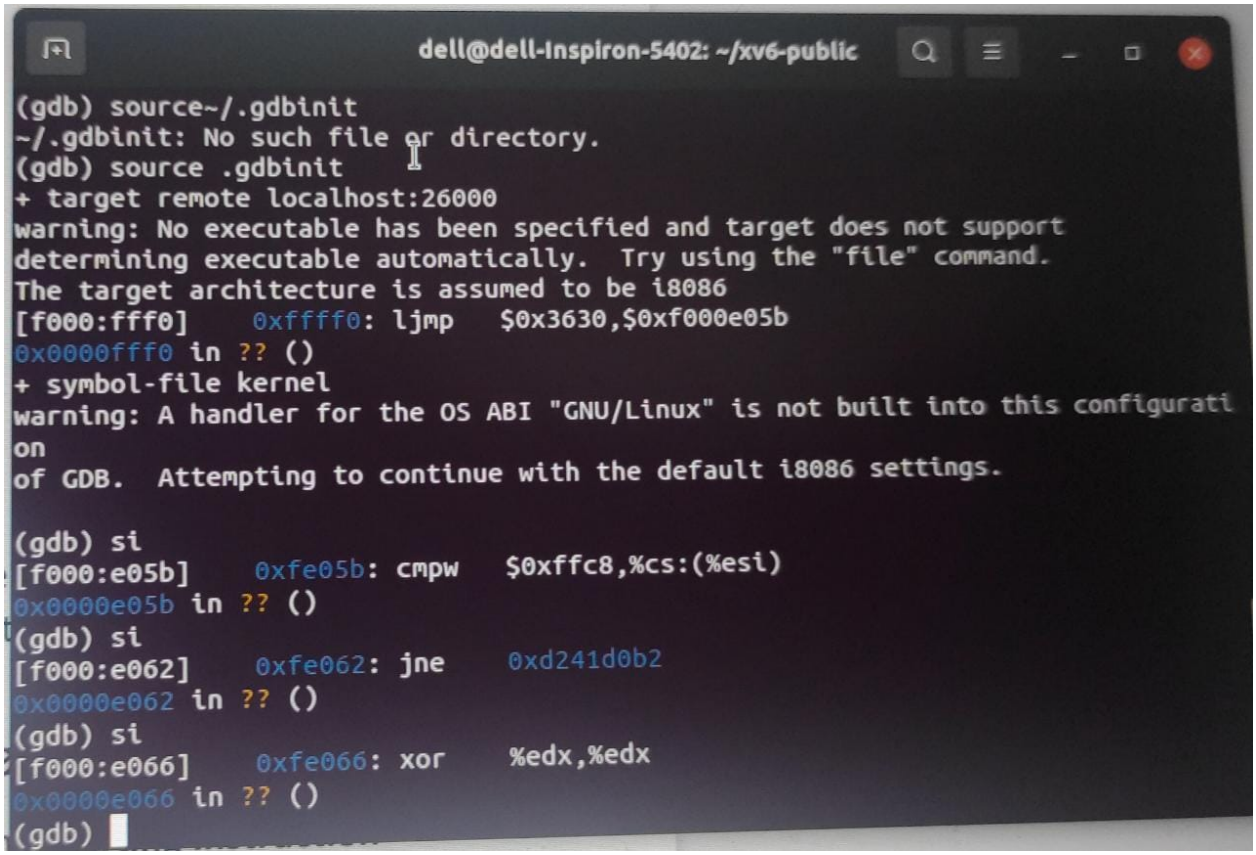




Explanation

Input operands are x and 1 while output operand is 1 . Through this code we add the value of x and 1 and save the output to x. Hence, increasing the value by 1.

Exercise 2



Explanation

The "`si`" instruction is used to execute one machine instruction and move to next
(follows a call). The above screenshot shows the first 4 instructions of the xv6 operating system.

- The first instruction is
  `[f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b`
  Here, `f000` is the Starting CS, `fff0` is the Starting IP, `0xffff0` is the Physical Address where this

instruction resides, `ljmp` is the Instruction which means long jump, `0x3630` is the Destination CS, `0xf000e05b` is the Destination IP.

- The cmp instruction is used to perform comparison.It is comparing the value of register cs with 0xffcB.
- The jnz instruction means jump on non zero while jne means jump on not equal. It jumps to the specified location if the Zero Flag is cleared . jnz is commonly used to test if something is not equal to zero whereas jne is commonly found after a cmp instruction.
- The xor instruction performs a logical XOR (exclusive OR) operation. Here it Xor the value of the registers and store it in itself.
- The mov is used to move the content of source register to destination register.

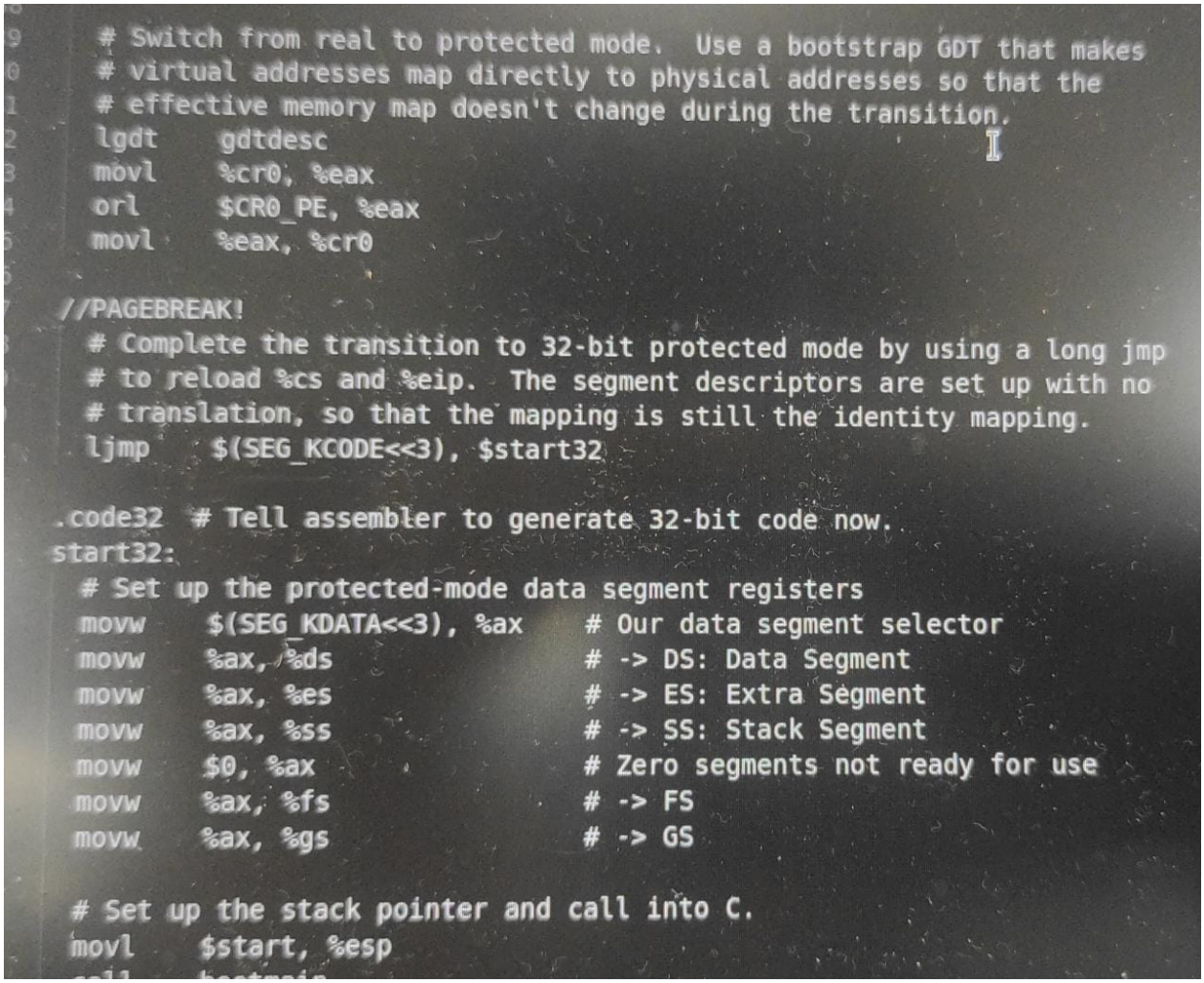
Exercise 3

```
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);
  }
```

```
for(; ph < eph; ph++){
  7d8d: 39 f3                   cmp    %esi,%ebx
  7d8f: 72 15                   jb     7da6 <bootmain+0x5d>
entry();
  7d91: ff 15 18 00 01 00       call   *0x10018

  7d97: 8d 65 f4                lea    -0xc(%ebp),%esp
  7d9a: 5b                      pop    %ebx
  7d9b: 5e                      pop    %esi
  7d9c: 5f                      pop    %edi
  7d9d: 5d                      pop    %ebp
  7d9e: c3                      ret
for(; ph < eph; ph++){
  7d9f: 83 c3 20                add    $0x20,%ebx
  7da2: 39 de                   cmp    %ebx,%esi
  7da4: 76 eb                   jbe    7d91 <bootmain+0x48>
  pa = (uchar*)ph->paddr;
```

```
  entry();
    7d91: ff 15 18 00 01 00     call   *0x10018
}
```

a)

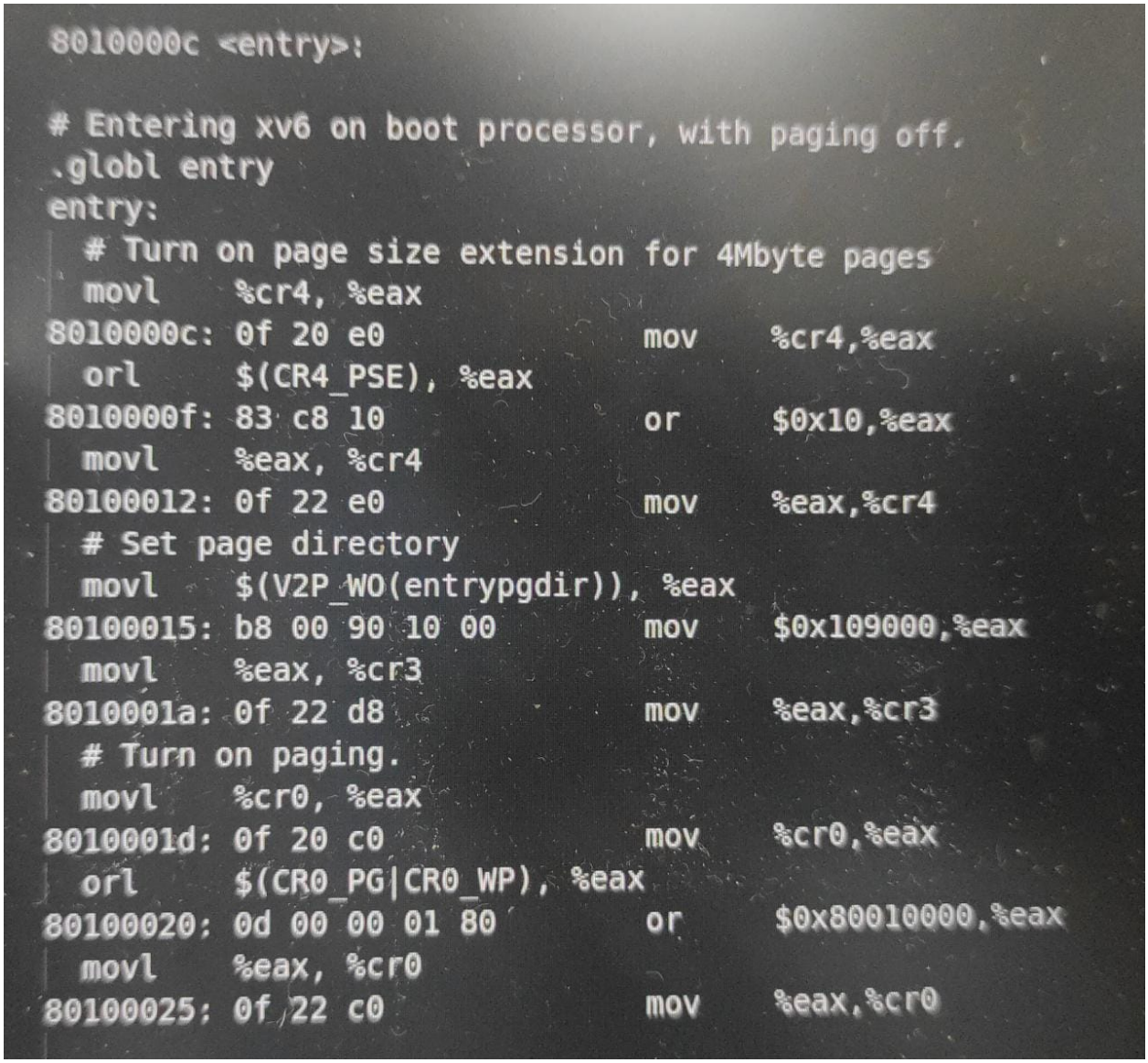


```
    # Switch from real to protected mode.  Use a bootstrap GDT that makes
    # virtual addresses map directly to physical addresses so that the
    # effective memory map doesn't change during the transition.
    lgdt    gdtdesc
    movl    %cr0, %eax
    orl     $CR0_PE, %eax
    movl    %eax,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

//PAGEBREAK!
    # Complete the transition to 32-bit protected mode by using a long jmp
    # to reload %cs and %eip.  The segment descriptors are set up with no
    # translation, so that the mapping is still the identity mapping.
    ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
start32:
    # Set up the protected-mode data segment registers
    movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
    movw    %ax, %ds                # -> DS: Data Segment
    movw    %ax, %es                # -> ES: Extra Segment
    movw    %ax, %ss                # -> SS: Stack Segment
    movw    $0, %ax                 # Zero segments not ready for use
    movw    %ax, %fs                # -> FS
    movw    %ax, %gs                # -> GS

    # Set up the stack pointer and call into C.
    movl    $start, %esp
```

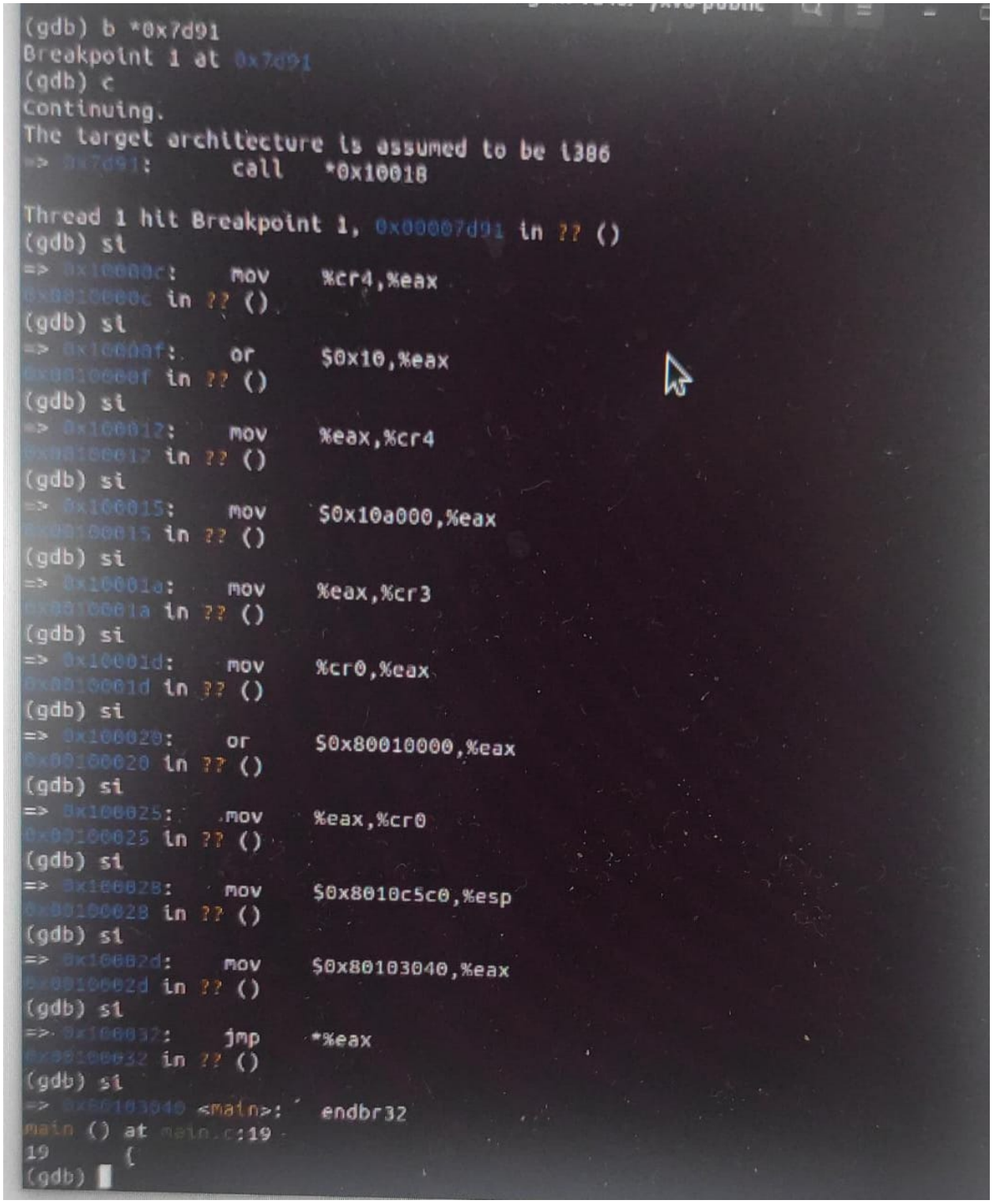
- Reading from the bootasm.s file, we get to know that the line
  ‘**movw $(SEG_KDATA<<3), %ax**’
  Is first instruction to be executed in 32-bit mode.

- ‘**ljmp $(SEG_KCODE<<3), $start32**’
  completes the transition to 32-bit protected mode from 16 bit-mode.

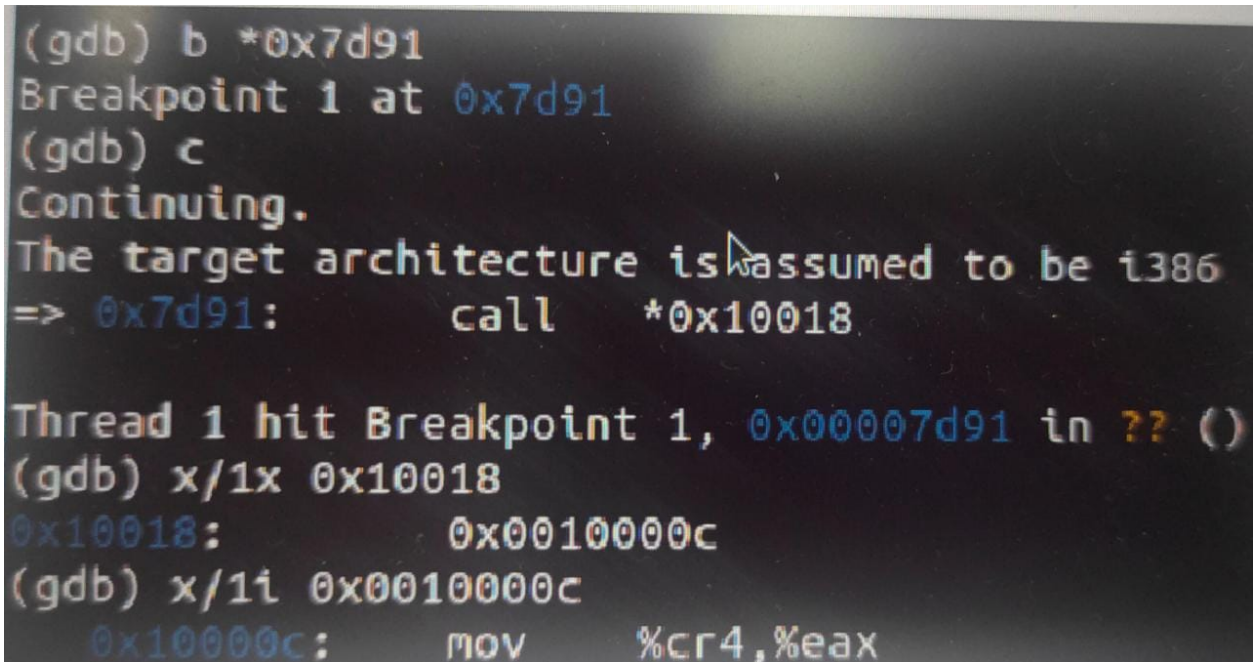
b)



```
8010000c <entry>:

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
8010000c: 0f 20 e0              mov    %cr4,%eax
  orl     $(CR4_PSE), %eax
8010000f: 83 c8 10              or     $0x10,%eax
  movl    %eax, %cr4
80100012: 0f 22 e0              mov    %eax,%cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
80100015: b8 00 90 10 00        mov    $0x109000,%eax
  movl    %eax, %cr3
8010001a: 0f 22 d8              mov    %eax,%cr3
  # Turn on paging.
  movl    %cr0, %eax
8010001d: 0f 20 c0              mov    %cr0,%eax
  orl     $(CR0_PG|CR0_WP), %eax
80100020: 0d 00 00 01 80        or     $0x80010000,%eax
  movl    %eax, %cr0
80100025: 0f 22 c0              mov    %eax,%cr0
```

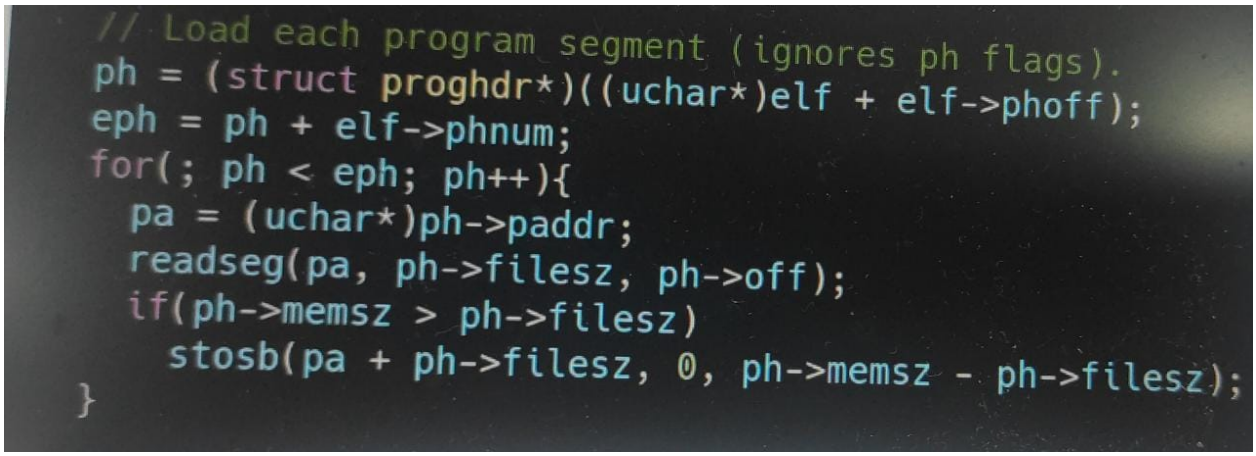

(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91: call   *0x10018
Thread 1 hit Breakpoint 1, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c: mov %cr4,%eax
0x0010000c in ?? ()
(gdb) si
=> 0x10000f: or $0x10,%eax
0x0010000f in ?? ()
(gdb) si
=> 0x100012: mov %eax,%cr4
0x00100012 in ?? ()
(gdb) si
=> 0x100015: mov $0x10a000,%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 $0x8010c5c0,%esp
0x00100028 in ?? ()
(gdb) si
=> 0x10002d: mov $0x80103040,%eax
0x0010002d in ?? ()
(gdb) si
=> 0x100032: jmp *%eax
0x00100032 in ?? ()
(gdb) si
=> 0x80103040 <main>: endbr32
main () at main.c:19
19 {
(gdb)

Using the kernel.asm file, and using the gdb si command after the the end of bootmain(), we
come to know that bootloaders last line is at the address 0x7d91. It contains;
7d91: ff 15 18 00 01 00 call *0x10018
which is call instruction that shifts control to the address stored at 0x10018 since dereferencing operator (*) has been used.
The contents of the address 0x10018, can be obtained by directly using the command x/1x 0x10018, which gives 0x10000c, which is the address which is finally called.



0x10000c is address of the 1st instruction of the kernel, which contains the below instruction:
mov %cr4,%eax

c)



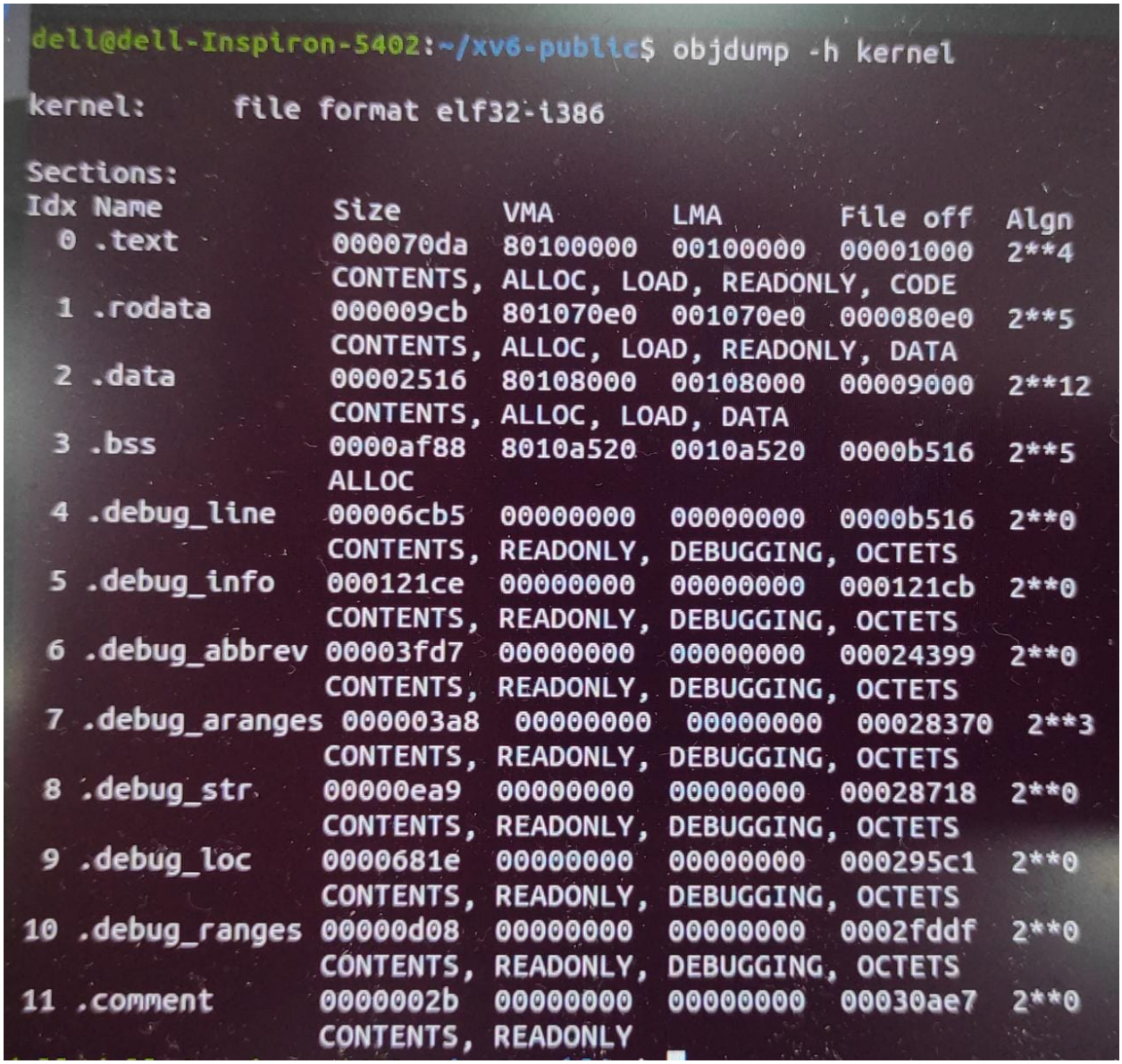
```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
  pa = (uchar*)ph->paddr;
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}
```

The above lines of code are in bootmain.c. It 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 each 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

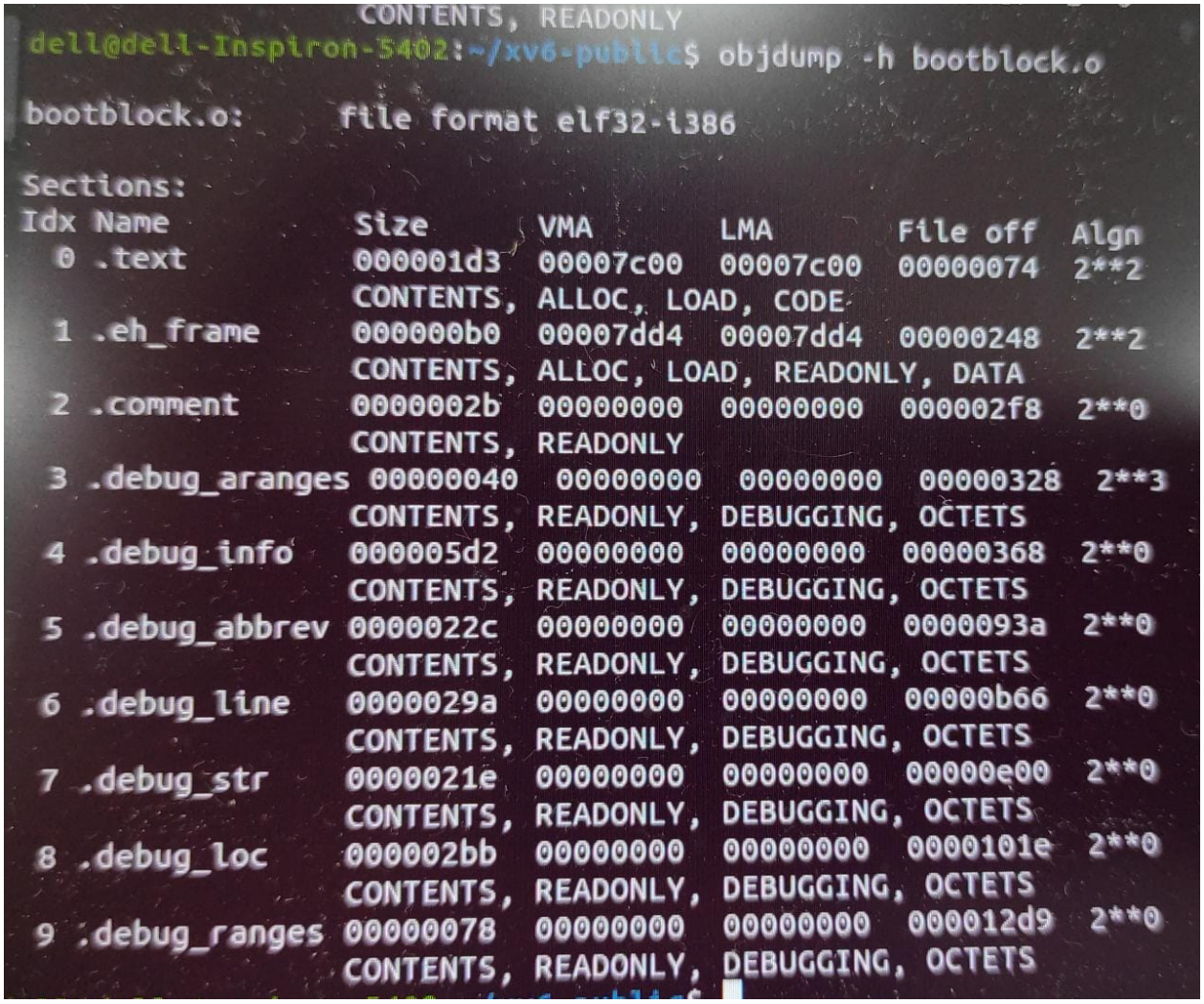
1. Running the **objdump -h kernel** command



```
dell@dell-Inspiron-5402:~/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    00000ea9  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295c1  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fddf  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002b  00000000  00000000  00030ae7  2**0
                  CONTENTS, READONLY
```

VMA and LMA of .text section are different which shows that it loads and executes from different addresses.

2.Running the **objdump -h bootblock.o** command

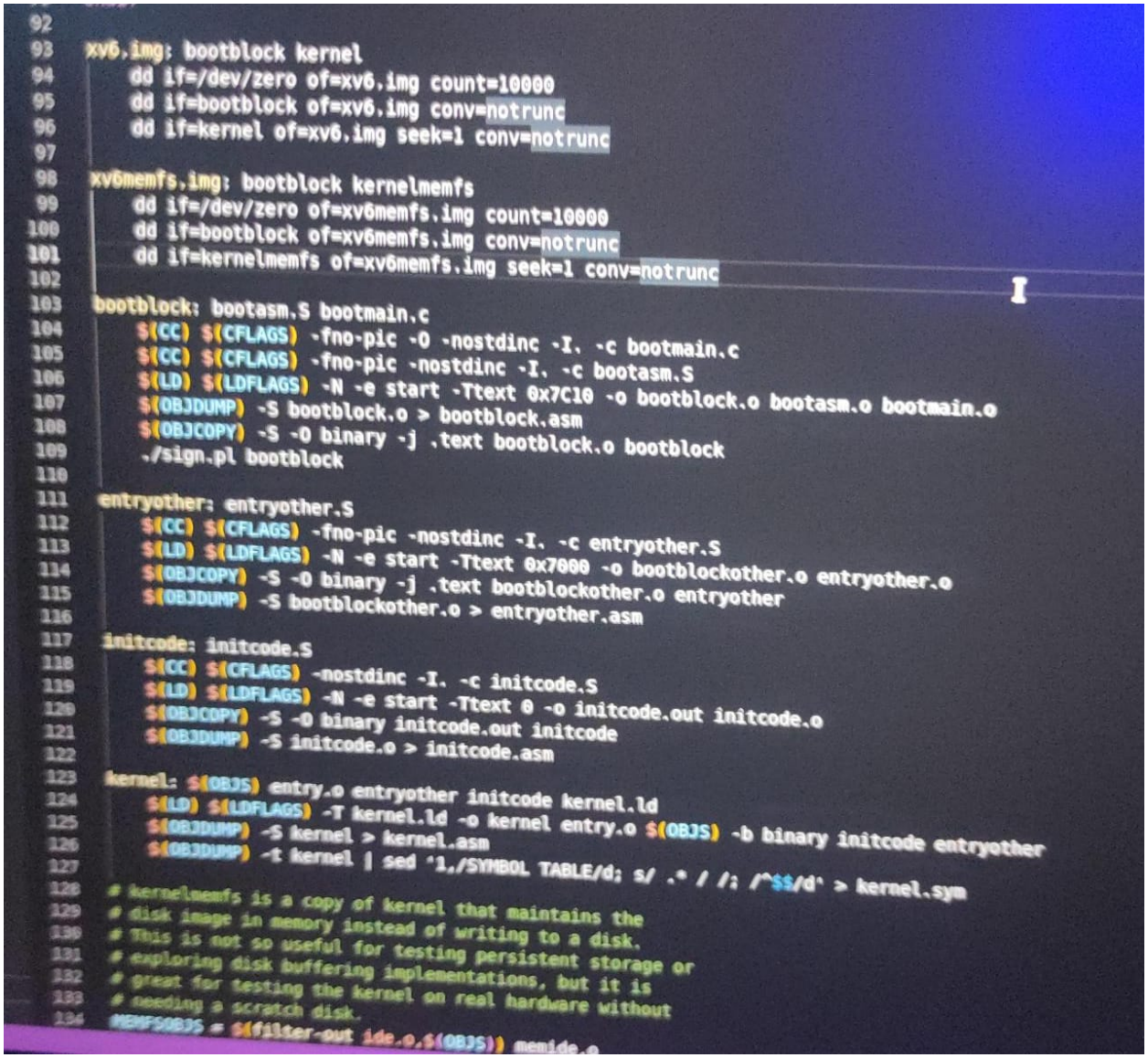


```
                  CONTENTS, READONLY
dell@dell-Inspiron-5402:~/xv6-public$ objdump -h bootblock.o

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      0000002b  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    0000021e  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  0000101e  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012d9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

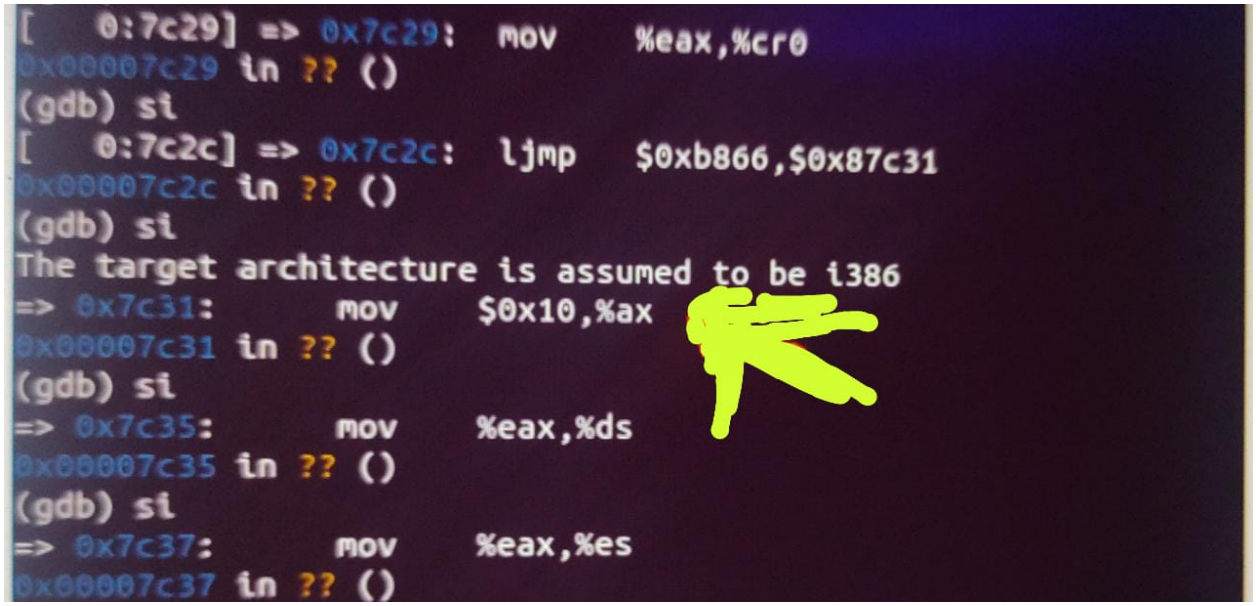
VMA and LMA of .text section are the same which shows that it loads and executes from same addresses.

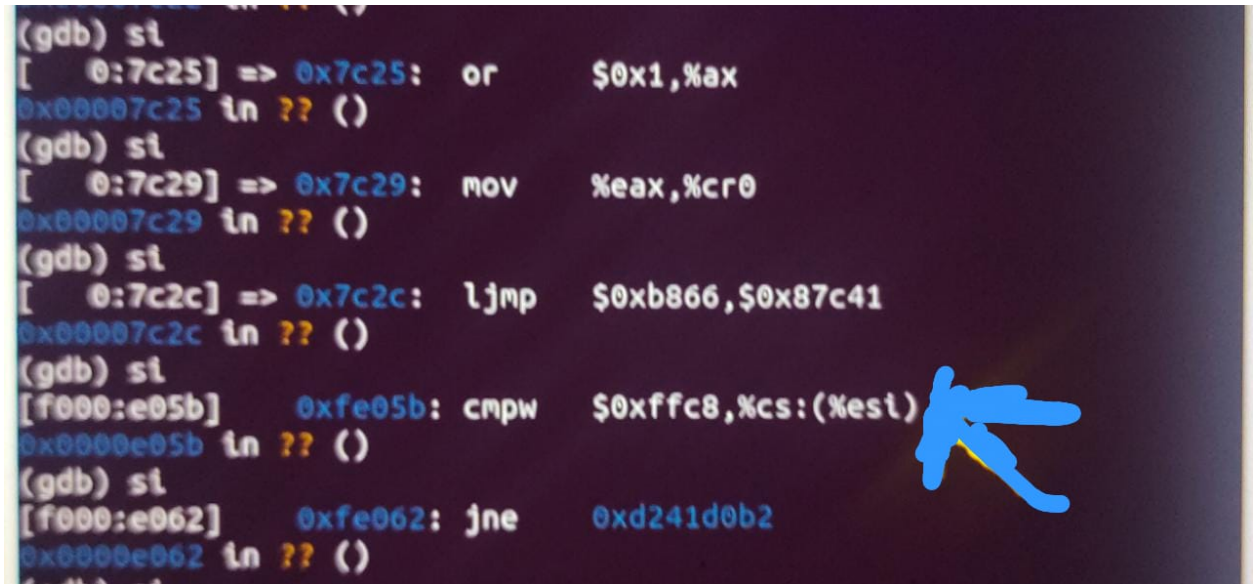
Exercise 5

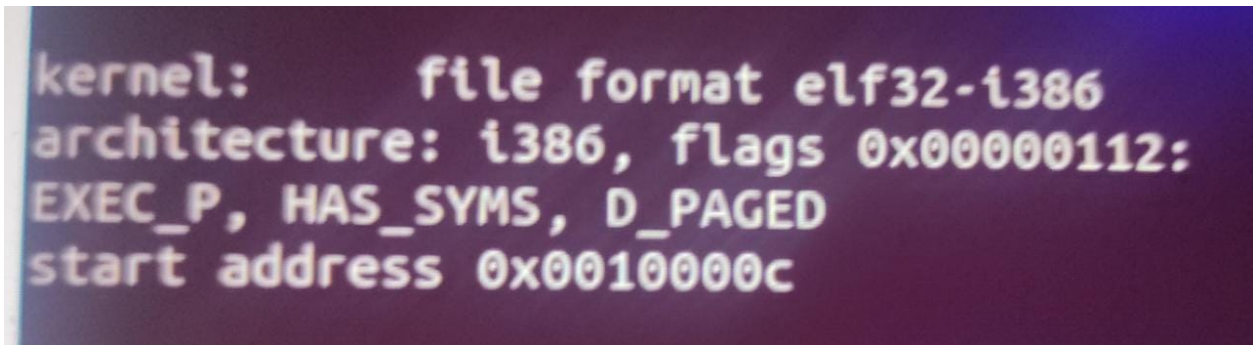
I changed the link address from 0x7c00 to 0x7c10 .Then after running make clean & make commands, i restarted the gdb and started to debug.Since no change has been done to the BIOS, it will run smoothly for both of the versions and hand over the control to the boot loader. Then I made 0x7c00 as the breakpoint and started F to check for differences between the two files. I did it by using si command comparing the outputs of the two files. The first command where a difference was spotted is shown below along with the next 3 instructions. I have attached the output files of gdb in my submission. I have also attached output files of “objdump –h bootmain.o” for both of the versions since the outputs differ due to the change in link address.



```
92
93  xv6.img: bootblock kernel
94      dd if=/dev/zero of=xv6.img count=10000
95      dd if=bootblock of=xv6.img conv=notrunc
96      dd if=kernel of=xv6.img seek=1 conv=notrunc
97
98  xv6memfs.img: bootblock kernelmemfs
99      dd if=/dev/zero of=xv6memfs.img count=10000
100     dd if=bootblock of=xv6memfs.img conv=notrunc
101     dd if=kernelmemfs of=xv6memfs.img seek=1 conv=notrunc
102
103 bootblock: bootasm.S bootmain.c
104     $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
105     $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
106     $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C10 -o bootblock.o bootasm.o bootmain.o
107     $(OBJDUMP) -S bootblock.o > bootblock.asm
108     $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
109     ./sign.pl bootblock
110
111 entryother: entryother.S
112     $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c entryother.S
113     $(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o
114     $(OBJCOPY) -S -O binary -j .text bootblockother.o entryother
115     $(OBJDUMP) -S bootblockother.o > entryother.asm
116
117 initcode: initcode.S
118     $(CC) $(CFLAGS) -nostdinc -I. -c initcode.S
119     $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o initcode.out initcode.o
120     $(OBJCOPY) -S -O binary initcode.out initcode
121     $(OBJDUMP) -S initcode.o > initcode.asm
122
123 kernel: $(OBJS) entry.o entryother initcode kernel.ld
124     $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
125     $(OBJDUMP) -S kernel > kernel.asm
126     $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
127
128 # kernelmemfs is a copy of kernel that maintains the
129 # disk image in memory instead of writing to a disk.
130 # This is not so useful for testing persistent storage or
131 # exploring disk buffering implementations, but it is
132 # great for testing the kernel on real hardware without
133 # needing a scratch disk.
134 MEMFSOBJS = $(filter-out ide.o,$(OBJS)) memide.o
```

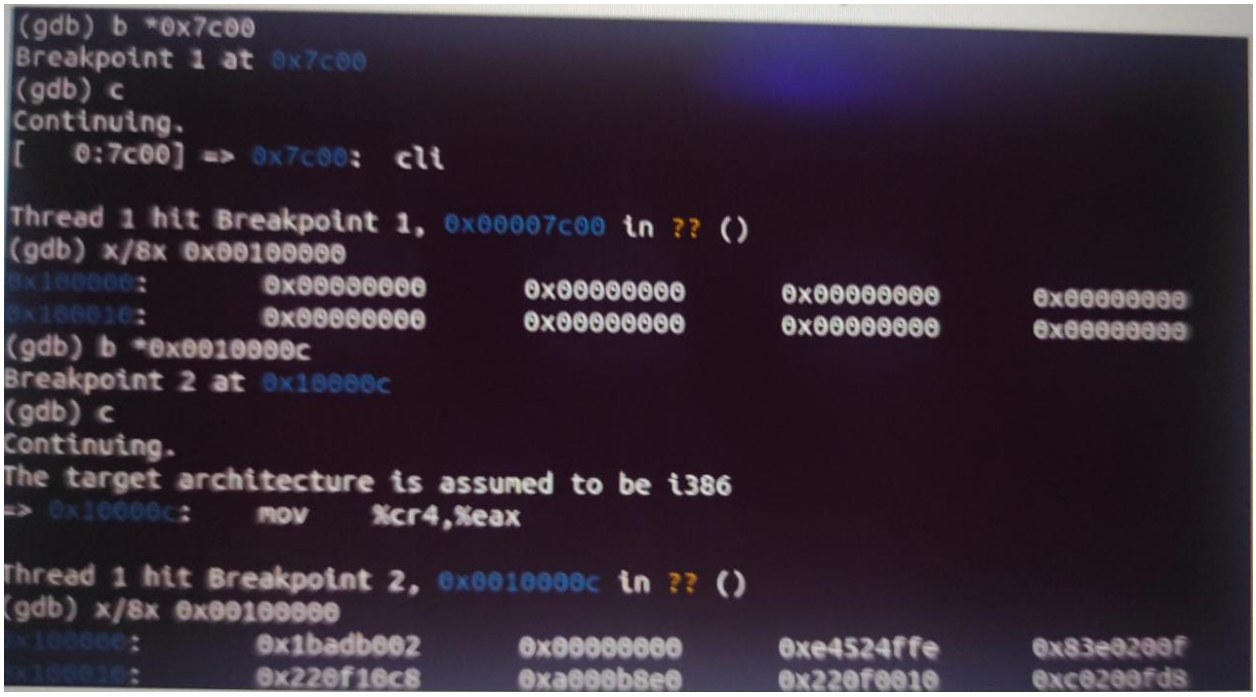

[0:7c29] => 0x7c29: mov %eax,%cr0
0x00007c29 in ?? ()
(gdb) si
[0:7c2c] => 0x7c2c: ljmp $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0x7c31: mov $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35: mov %eax,%ds
0x00007c35 in ?? ()
(gdb) si
=> 0x7c37: mov %eax,%es
0x00007c37 in ?? ()


(gdb) si
[0:7c25] => 0x7c25: or $0x1,%ax
0x00007c25 in ?? ()
(gdb) si
[0:7c29] => 0x7c29: mov %eax,%cr0
0x00007c29 in ?? ()
(gdb) si
[0:7c2c] => 0x7c2c: ljmp $0xb866,$0x87c41
0x00007c2c in ?? ()
(gdb) si
[f000:e05b] 0xfe05b: cmpw $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne 0xd241d0b2
0x0000e062 in ?? ()


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

Exercise 6

We examine the 8 words of memory at 0x00100000 at two different instances of time, the first when the BIOS enters the boot loader and the second when the boot loader enters the kernel. Use the command “x/8x 0x00100000” but before that we will have to set our breakpoints. The first breakpoint will be at 0x7c00 because this is the point where the BIOS hands control over to the boot loader. The second breakpoint will be at 0x0010000c because this is the point when the kernel is passed control by the bootloader. We get different values at both the breakpoints. The explanation to this is as follows. The address 0x00100000 is actually 1MB which is the address from where the kernel is loaded into the memory. Before the kernel is loaded into the memory, this address contains no data (i.e.garbage value). By default, all the uninitialized values are set to 0 in xv6.Hence, when we tried to read the 8 words of memory at 0x00100000 at the first breakpoint, we got all zeroes since no data had been loaded until that point. When we check the values at the second breakpoint, the kernel has already been loaded into the memory and thus this address now contains meaningful data instead of zeroes.


(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 *0x0010000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c: mov %cr4,%eax
Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x83e0200f
0x100010: 0x220f10c8 0xa000b8e0 0x220f0010 0xc0200fd8

Part B
Exercise 1
The 5 files which are edited to add the system call are :

Syscall.h: This file assigns a number to every system call in xv-6 system. Before adding draw there were 21 system calls , hence draw was assigned number 22.
Line:#define SYS_draw 22

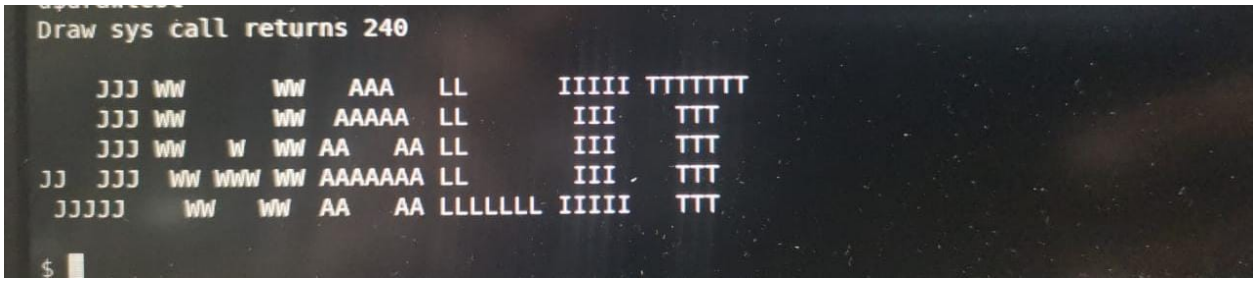
Syscall.c: It contains an array of function pointers(syscalls[]) which uses index defined in systemcall.h to point to the respective system call function stored at a different memory location. We also put a function prototype here(but not implementation).
Line1:Extern int sys_draw(void);
Line2:[SYS_draw] sys_draw,

Sysproc.c: This is where the implementation of our system call is written

user.h and usys.S: They act as an interface for our system to access the system call. The function prototype is added in user.h(included as header file in our program) while instruction to treat it as a system call is included in usys.S
usys.S: SYSCALL(draw)
user.h: int draw(void*,int)

Exercise 2

Makefile:
The program(draw) is included in the UPROG list which are written to fs.img as well as EXTRA list which is included when making dist.
draw.c:
The user level application draw.c contains the instructions to allocate space to the buffer and then call the system call to write the ascii JWALIT string in the buffer. In the end, the image is printed to the console



Made by
Jwalit Bharatkumar Devalia
200123026
Mathematics and Computing
Lab 0