Bootloader in asm

Name: Samiksha Modi
Roll No: 2019331

Logic and Implementation

I implemented a simple bootloader in assembly language. When we press the power button the computer loads the BIOS from some flash memory stored on the motherboard. The BIOS initializes and self tests the hardware then loads the first 512 bytes into the memory from the media device (ie. cd rom or floppy disk). If the last two bytes equal 0xaa55 then the BIOS will jump to location 0x7c00 effectively transferring control to the bootloader.[1] our bootloader should be 512 bytes as well.

I start off with letting the nasm know that it is a 16 bit code. I then give the offset 0x7c00. The BIOS loads the program into this address. So if I want the offset of my value calculated correctly I need to put my origin here, otherwise it would have calculated the offset from 0 giving me the wrong memory address when I do things in the future.

I then enter protected mode. I set the vga text mode to 3. vga is a simple way to print text to screen. I then enable A20 line by calling the 'A-20 Gate activate' function so that I can access more than 1MB of memory. Then I disable the interrupts. Then I load the global descriptor table (gdt) with the lgdt instruction. The gdt has entries telling the cpu about memory segments. Then I set the protected mode bit on the special cpu register cr0. I then do a far jump to the 32 bit code segment. For that we need to change the memory segment, so I change it to CODE_SEG .

After that I set up the gdt. The gdt I set up has 3 parts- a null segment, a code segment and a data segment. I referred to the gdt entry layout present in [1].

In gdt_null segment I just initialised dq to 0x0. It is always set to 0 and is reserved by Intel.We write 64 bits containing 0.

Then in gdt_code segment

1st Double word:

Bits	Function	Description
0-15	Limit 0:15	First 16 bits in the segment limiter
16-31	Base 0:15	First 16 bits in the base address

2nd Double word:

Bits	Function	Description
0-7	Base 16:23	Bits 16-23 in the base address
8-12	Type	Segment type and attributes
13-14	Privilege Level	0 = Highest privilege (OS), 3 = Lowest privilege (User applications)
15	Present flag	Set to 1 if segment is present
16-19	Limit 16:19	Bits 16-19 in the segment limiter
20-22	Attributes	Different attributes, depending on the segment type
23	Granularity	Used together with the limiter, to determine the size of the segment
24-31	Base 24:31	The last 24-31 bits in the base address

Source: http://www.osdever.net/tutorials/view/the-world-of-protected-mode

1st Double Word

- The first 16 bits sets the limit. Since I want maximum possible limit I set it to 0xFFFF.
- Then I set the base address to 0 which is the start of the memory

2nd Double Word

- Bits 0-7 continues on the base address and so I set it to 0.
- Bit 8 is the access flag.I set it to 0. The CPU sets this to 1 when the segment is accessed.
- Bit 9 is the readable/writable bit. It determines if the segment is readable or not. I set this to 1.
- Bit 10 is the direction bit/conforming bit. If this bit is set, then less privileged code segments is allowed to jump to or call this segment. I don’t want that in an OS so I set it to 0.
- Bit 11 specifies if it is a code or data segment. I set this bit to 1 because I have a data segment later.
- Bit 12 is the descriptor type. I set this to 1 for code segment. It should be 0 for data segment.

- Bit 13-14 is the privilege level. The two bits can range from 0-3. 0 is the most privileged and 3 the least privileged. Since this segment is part of my OS, I set it to 0.
- Bit 15 is the present flag. I set this bit to 1.

So now I get 10011010b. In the end I put a b so that it knows I am declaring a byte.

- Bits 16-19 is the last bits in the segment limit. I set this to 1111.
- Bit 20 represents a flag of 'Available to System Programmers'. It is ignored by the cpu so i set it to 0.
- Bit 21 is reserved by Intel and is always 0.
- Bit 22 is the size bit. It tells the CPU that I have 32-bit code if it is set to 1 and 16 16-bit if it is set to 0. So I set it to 1.
- Bit 23 is the granularity bit. If it is set to 1 the limiter multiplies segment limit by 4kB. So I set this to 1.

So now I get 11001111b. In the end I put a b so that it knows I am declaring a byte.

- Bits 24-31 is the 8 bits remaining on the base address. I set this to 0.

The code remains the same for the data segment except for 1 change. Bit 11 is set to 0 as this is a data segment.

Then I make a gdt pointer structure to be able to load this. It is a 16 bit field containing the gdt size followed by a 32 bit pointer the structure itself. I then define the CODE_SEG and DATA_SEG value which are the offsets into the gdt which will be used in the code.

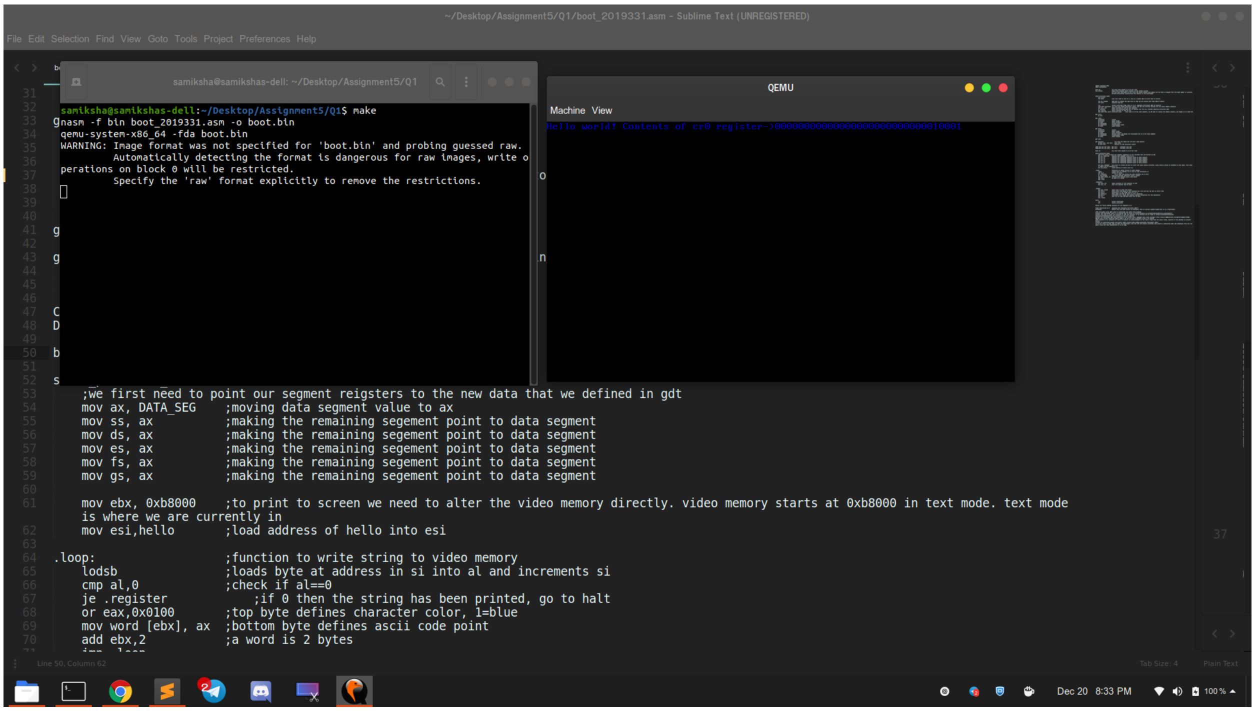
Next I go into 32 bit mode. The first thing I do is point my segment registers to the new data I defined in gdt. So I move my DATA_SEG to ax. Then I move ax to the segment registers ss,ds,es,fs,gs.

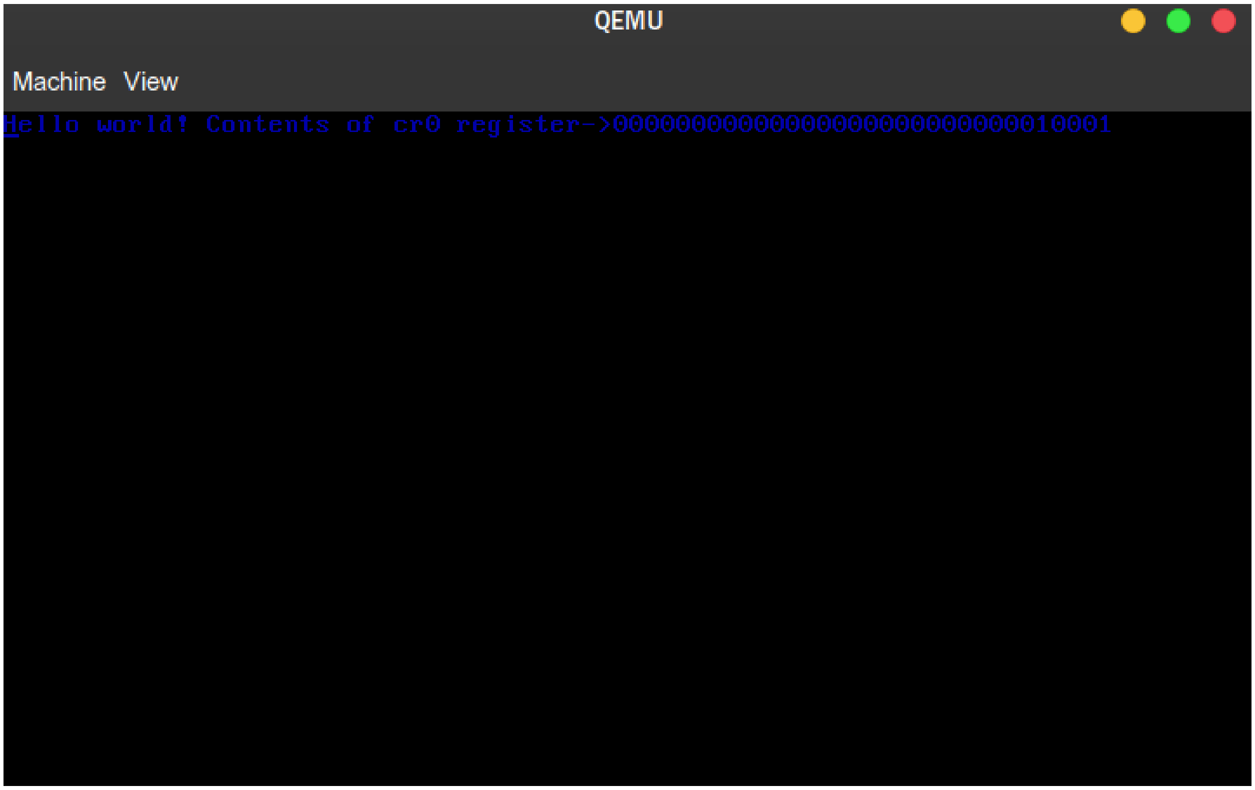
I can't call the BIOS anymore but I can write to the screen by altering the video memory directly. Video memory starts at 0xb8000 in text mode. Text mode is where we currently are.

The top byte of the eax regiter defines the character colour in the the buffer as int value from 0-15 with 0=black, 1=blue and 15=white. The bottom byte defines an ascii code point. I use this to print hello world in blue on the screen. I move 0xb8000 to ebx register. I load the address of the string hello into esi. Then I create a loop. It writes this string to video memory. I do lodsb which loads byte at address in si into al and increments si. I then check if al==0. If it is 0 it means the whole string has been printed and I move on to print the cr0 register contents. Otherwise I keep printing the string character by character until I'm done by incrementing the value in ebx register by 2 and running the loop again and again.

To print the contents of the cr0 register, I move the contents of cr0 to the edx register. The cr0 register has 32 bits so I just need to print the 32 bits.I move 32 into ecx register. I am printing the contents in binary. For this I create a loop2. I move 0x130 to the eax register which sets text color to blue from black. Then I left shift the value in the edx register by1 bit and put the top bit in the carry flag. I then determine if the lower byte is 0 or 1. I then move this value to be printed to the location stored in ebx. I then increment ebx by 2 since a word is 2 bytes.I then dec ecx to ensure that I stop after I print the 32 bits in the cr0 register. After I'm done printing the contents of the cr0 register, I halt execution.

I then pad the remaining 510 bytes with 0 and write dw 0xaa55 which marks this 512 byte sector as a bootloader.





Instructions to boot the image

```
Makefile
1 #Name:Samiksha Modi
2 #Roll No:2019331
3 all:
4 	nasm -f bin boot_2019331.asm -o boot.bin
5 	qemu-system-x86_64 -fda boot.bin
6
```

I have submitted the Makefile.
Line 4 in Makefile compiles the required binary.
Line 5 in Makefile tells QEMU to boot using the bootable binary

Sources Used

The following sites were used to understand and write this program

1. The bootloader code in my program can be found at http://3zanders.co.uk/2017/10/16/writing-a-bootloader2/

2. The code to print the contents of the cr0 register in my program can be found at https://stackoverflow.com/questions/65361946/print-the-contents-of-the-cr0-register
3. To understand what was happening in the Link1 code I watched this video multiple times https://www.youtube.com/watch?v=pXzortxPZR8 Moving Into 32-Bit Protected Mode | Make a 64 bit OS From Scratch!! | Part 4 by Poncho
4. http://www.osdever.net/tutorials/view/the-world-of-protected-mode To understand global descriptor table

The comments in my program are there to explain my understanding of the code I took from the above links. I do not claim to have come up with the bootloader code and the cr0 content printing code myself.I understood what was happening from the above links and then implemented it on my own.