# REPORT [Lab 1: Booting a PC]

**I chose to finish the colored text challenge**

## Part 1: PC Bootstrap

### Exercise 1

This exercise urges us to get familiar with the assembly language. Thankfully there are several materials available at on the class's reference pages.

Thanks to the ICS, I've been equipped with some basic knowlege of assembly language already. But one thing needs to be cautious about, that in AT&T syntax

> the sources is *always* on the **left**, and the destination is *always* on the **right** [1]

And I am going to read through [2] as suggested along the way

### Exercise 2

This exercise suggests us to use GDB's `si` to trace into the ROM BIOS after the `ljmp $0xf000,$0xe05b` command.

What I find is

```
 1  [f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
 2  [f000:e062]    0xfe062: jne    0xfd2e1
 3  [f000:e066]    0xfe066: xor    %dx,%dx
 4  [f000:e068]    0xfe068: mov    %dx,%ss
 5  [f000:e06a]    0xfe06a: mov    $0x7000,%esp
 6  [f000:e070]    0xfe070: mov    $0xf34c2,%edx
 7  [f000:e076]    0xfe076: jmp    0xfd15c
 8  [f000:d15c]    0xfd15c: mov    %eax,%ecx
 9  [f000:d15f]    0xfd15f: cli
10  [f000:d160]    0xfd160: cld
11  [f000:d161]    0xfd161: mov    $0x8f,%eax
12  [f000:d167]    0xfd167: out    %al,$0x70
13  [f000:d169]    0xfd169: in     $0x71,%al
14  [f000:d16b]    0xfd16b: in     $0x92,%al
15  [f000:d16d]    0xfd16d: or     $0x2,%al
16  [f000:d16f]    0xfd16f: out    %al,$0x92
17  [f000:d171]    0xfd171: lidtw  %cs:0x6ab8
18  [f000:d177]    0xfd177: lgdtw  %cs:0x6a74
19  [f000:d17d]    0xfd17d: mov    %cr0,%eax
20  [f000:d180]    0xfd180: or     $0x1,%eax
21  [f000:d184]    0xfd184: mov    %eax,%cr0
22  [f000:d187]    0xfd187: ljmpl  $0x8,$0xfd18f
23  # The target architecture is set to "i386".
24  # ...
```

At line 1~2, some variable at `%cs:0x6ac8` is compared to 0 and it turns out they're not equal so the control transfers to `0xfd2e1`. I didn't figure out why to do the comparison at first. Then I found this [3] and got know that this comparation is done to determine wheter this is a resume/reboot or a normal start.

At line 3~6, some values(0, `$0x7000`, `0xf34c2`) are assigned to `%ss`, `%esp` and `%edx` . According to [4], `%ss` is used as segment element register. So I guess that this set the start of the stack to the address `$0x7000` . The value stored in `%edx` is later used to call some function.

At line 7, the control flow jumps to a new postion.

line 8~9 and line 17~18 might be the code to load some preconfigured description table.

line 11~16 performs some I/O operation. It seems confusing at first. Thanks to [5,6], I understand that line 11~13 is to disable NMI and line 14~16 is to activate some system device and the first of 2 consecutive `in` is meant to clean the side effect brought by reading from `$0x70`.

At line 18~22, the first bit of `%cr0` is set to `1`, this causes the processor to begin exuting in protected mode [2].

There are still more commands to run by BIOS. But now I know that all this is trying to initialize things around at first.

## Part 2: The Boot Loader

### Exercise 3

Both boot.S and main.c are dissambled in boot.asm and all instructions are allocated with some physical address starting at 0x7c00. GDB's x/i shows instructions in memory like boot.asm. But boot.asm has some strange mistakes in the traslation (may because of alignment), which are corrected in GDB's x/i.

```
ljmp    $PROT_MODE_CSEG, $protcseg
    7c2d:   ea                          .byte 0xea
    7c2e:   32 7c 08 00                 xor    0x0(%eax,%ecx,1),%bh
```

now to answer a few questions

> - At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

```
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
ljmp    $PROT_MODE_CSEG, $protcseg
```

By GDB, I know the ljmp is the very instruction to make the processor switch to 32-bit mode. But there is some preworks. First lgdt is used to load a GDT which makes virtual addresses identical to their physical addresses (segment starting at 0). Then the first bit of `%cr0` is set to 1 to enable protected mode. Finally ljmp is used to change the `%cs` to contain the selector for a code segment in the GDT, which is required to activate protected mode. [7,8]

> - What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?

```
0x7d71:      call   *0x10018 # last instruction of the boot loader: call into the kernel just loaded
0x10000c:    movw   $0x1234,0x472 # first instruct of the kernel: warm boot
```

> - *Where* is the first instruction of the kernel?

kern/entry.S

> - How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

The size of each segment of the kernel is record in `struct Proghdr` so that the boot loader knows how many sectors it must read. This information is record as `struct Elf` at the first page of the disk.

## Exercise 4

This is a reading exercise of mercy. I read through the materials, and find that thankfully I've already understood pointers in ICS in the hard way.

```
/**
 * for the 5-th line output
 * old:
 *  a[2]          a[3]
 *  90 01 00 00 | 2D 01 00 00
 * 500 = 0x1F4
 * new:
 *  a[2]          a[3]
 *  90 F4 01 00 | 00 01 00 00
 *  128144        256
*/
```

## Exercise 5

```
ljmp    $PROT_MODE_CSEG, $protcseg
```

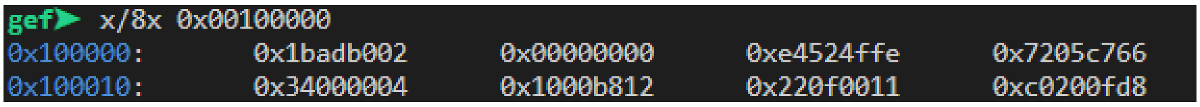
This instruction will jump to some address which the linker will interpret according to the link addresss and the offset resulting `0x7c32` since BIOS loads bootloader at address `0x7c00`.

But if the link address is wrongly configured other than `0x7c00`, although bootloader will still be loaded at `0x7c00`, the addresses of jmp instructions are wrong. So the first jmp instruction executed leads the program to some unexpected address and crashed as I deliberately changed the link address to `0xf000`.

## Exercise 6

At the point the BIOS enters the boot loader, the 8 words of memory at 0x00100000 are merely zeros.

At the point the boot loader enters the kernel, these words are



```
gef➤  x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x1000b812      0x220f0011      0xc0200fd8
```

which is the machine code of the kernel since the boot loader loads the kernel at 0x00100000 according to the ELF header.

## Part 3: The Kernel

### Exercise 7

Before the `movl` instruction, the memory at 0x00100000 is the kernel and the memory at 0xf0100000 is all 0s.

After this instruction, the memory at 0x00100000 is unchanged while the memory at 0xf0100000 becomes the same as the memory at 0x00100000 because the paging is enabled by the `movl` instruction and the initial page directory configured by kern/entrypgdir.c maps the virtual memory at 0xf0100000 to the physical memory at 0x00100000.

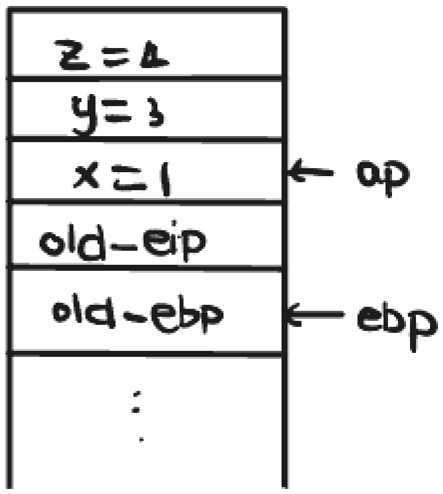
If the mapping weren't in place, the first instruction failed is the instruction after jumping to `relocated`. That's because `EIP` becomes high after the jump so that a instruction at invalid physical address will be tried to be fetched since the mapping isn't in place.

> qemu: fatal: Trying to execute code outside RAM or ROM at 0xf010002c

### Exercise 8

The code for "%o" is essentially the same as the code for "%x". Find the corresponding unsigned interger using `getuint` and set the base to be 8. Then goto number to use `printnum` to produce the output.

1. `cputchar(int)` is the function that `console.c` exports. It is used as the backbone of `putch(int, int *)` by `printf.c`
2. The condition is to determine if the position of the cursor is out of the range of current terminal window. If so, the content in the terminal moves upward for one column and the newline is filled with black ' 's. Then position of cursor is updated by subtracting the size of one column.
3. - In the call to `cprintf()`, `fmt` points to the format string `"x %d, y %x, z %d\n"` while `ap` points to where `x` resides in the stack after initialized by `va_start`



   -
     ```
     vcprintf :
         fmt = 0xf0101a77 "x %d, y %x, z %d\n"
         ap = 0xf010efd4 "\001"
     cons_putc : c = 120
     cons_putc : c = 32
     ```

```
va_arg :
    before: ap = 0xf010efd4 "\001"
    after:  ap = 0xf010efd8 "\003"
cons_putc : c = 49
cons_putc : c = 44
cons_putc : c = 32
cons_putc : c = 121
cons_putc : c = 32
va_arg :
    before: ap = 0xf010efd8 "\003"
    after:  ap = 0xf010efdc "\004"
cons_putc : c = 51
cons_putc : c = 44
cons_putc : c = 32
cons_putc : c = 122
cons_putc : c = 32
va_arg :
    before: ap = 0xf010efdc "\004"
    after:  ap = 0xf010efe0 "` ", <incomplete sequence \360>
cons_putc : c = 52
cons_putc : c = 10
```

4. `He110 World` is output.

   First, `H` is output as in the format string.

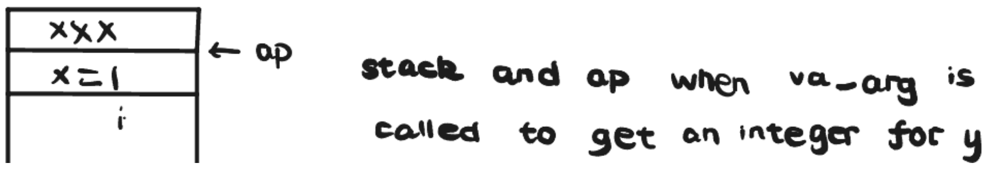
   Then `vprintfmt` detects `%x` and use `va_arg` to get `57616` in `getuint` and `57616` is converted to 16-base as `e110`.

   After that, one whitespace and `Wo` are output according to the format string.

   At last `vprintfmt` detects `%s` and use `va_arg` to get its third argument, a pointer to the address of `i` with value `0x00646c72`. This pointer is interpreted as the start of the string and since x86 is little-endian, with the result `72 6c 64 00` which means `rld\0` in string.

   If the x86 were big-endien, we would want the bytes in `i` to be in the reverse order to produce the same result, which is to say `0x726c6400` to be the value of `i`. Since `57626` is interpreted as an integer in both its initialization and usage, which means that consistent results are produced where the architecture is little-endian or big-endian, we don't have to change it.

5. Some random interger is output after `y=`. Since the format string induces `vprintfmt` to use `ap` pointing to the top of `3` as the first argument of `va_arg`, the word just above `3` is interpreted as an integer and returned.

6. If the calling convention were changed this way, `va_start(ap, fmt)` should make ap pointed to the first byte after `fmt` instead of the first byte before `fmt`. And `va_arg` should in turn search in a reversed manner, which means it will return an appropriate value after current `ap` instead of before it and change the value of `ap` accordingly. ("before" and "after" are defined with the manner of stack's grow. That means, "before" indicates higher address)

## Excercise 9

Kernel initializes its stack at `relocated` in `entry.S`. According to the assembly `kernel.asm` , the stack is located at [0xf0107000,0xf010f000). This space is reserved by `.space` directive. The stack pointer is initialized to point to the higher "end", 0xf010f000.

## Excercise 10

There are 8 words pushed on the stack by eac recursive nesting level of `test_backtrace`. The first three words are callee-saved `%ebp,%esi,%ebx`. Then there are three padding words to keep `%esp` 4-aligned. At last there are one word for the argument of the recursive call and one word for the address of next instruction saved by `call`.

## Excercise 11

Since there is a struct `Command commands[]` defined in `kern/monitor.c` , it's easy to hook `mon_backtrace()` into this list.

The value of `%ebp` can be obtained by using the defined `read_ebp()` function. Since `%eip` is pushed onto stack by `call` just before previous `%ebp` pushed onto stack, the value of `%eip` can be referred by `*((int *)ebp + 1)` . Since arguments are pushed  onto stack just before the `call`, the i-th argument can be referred by `*((int *)ebp + 2 + i)`.

And the `%ebp` for the caller function can be restored by the value saved by function prologue, that is to say, `*(int *)ebp`, the word at the address of current `%ebp`.

when the kernel stack is initialized at `entry.S`, `%ebp` is set to be 0. So the termination condition for `mon_backtrace()` is that `ebp` equals 0.

The return instruction pointer typically points to the instruction after `call` instruction since it's the instruction expected to be executed after the called function returns.

Since there aren't any difference before arguments and other wods in the stack, the backtrace code is unable to detect how many argumens there actually are. Maybe this can be fixed by adding some magical value at the end of argument list so that all words between `%eip` and this magic value are arguments. Or some information provided by compilers is needed.

## Excercise 12

The `gcc` is configured to generate some `.stab*` directives containing debug information in `.s` files. Then in `kern/kernel.ld`, the linker is asked to use this information to generate `.stab` and `.stabstr` sections so that the debug information is contained in the executable file `obj/kern/kernel`. Finally the bootloader loads the symble table in memory as part of loading the kernel binary according to the ELF header.

It's easy to complete `debuginfo_eip` using the `stab_binseatch()` provided. Just need to remember to check the modified value. According to [9], the line number is stored at `n_desc` field.

Then we can use `mon_backtrace` to call `debuginfo_eip` and output the gathered information as required.

`%.*s` is implemented to take one integer and check whether output chars have reached this limit during printing the string.

## Challenge

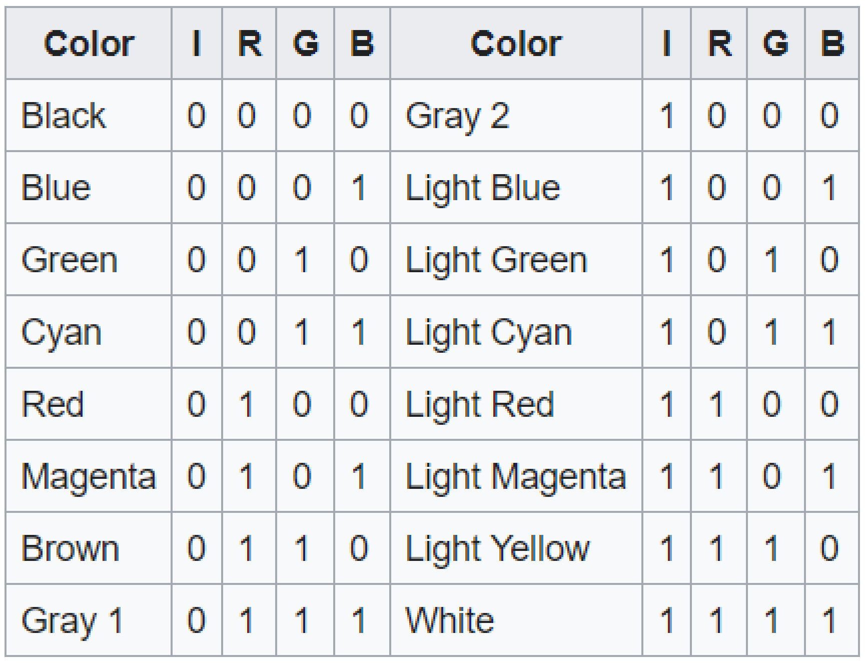
To print colored text, there are 2 parts to consider

1. how to indicate which segment of text should be color and the color used
2. how to make VGA print colored text

For the first part, I followed the suggestion to refer to the ANSI escape sequence [10]. These are sequences that start with "ESC[" which modify the behavior of the terminal like changing the text color, moving the cursor and so on. For cost consideration, I only implemented the logic around color modification.

```
ESC[Ps;Ps;...Psm
Ps = 0
    -> clear all color attribute
30 <= Ps <= 37
    -> foreground colors
40 <= Ps <= 47
    -> background colors
```

For the second part, I delved into the memory storage scheme of CGA and finally understood its structure. There are 2 bytes for each character on the termial where the lower byte is the ASCII code and the higher byte indicates its color. What's more, the lower 4 bits of the color byte indicates foreground color while the upper 4 bits indicates the background color. And the corresponding 4 bits have similar meaning for both lower and higher parts of the color byte.[11]



| Color | I | R | G | B | Color | I | R | G | B |
|---|---|---|---|---|---|---|---|---|---|
| Black | 0 | 0 | 0 | 0 | Gray 2 | 1 | 0 | 0 | 0 |
| Blue | 0 | 0 | 0 | 1 | Light Blue | 1 | 0 | 0 | 1 |
| Green | 0 | 0 | 1 | 0 | Light Green | 1 | 0 | 1 | 0 |
| Cyan | 0 | 0 | 1 | 1 | Light Cyan | 1 | 0 | 1 | 1 |
| Red | 0 | 1 | 0 | 0 | Light Red | 1 | 1 | 0 | 0 |
| Magenta | 0 | 1 | 0 | 1 | Light Magenta | 1 | 1 | 0 | 1 |
| Brown | 0 | 1 | 1 | 0 | Light Yellow | 1 | 1 | 1 | 0 |
| Gray 1 | 0 | 1 | 1 | 1 | White | 1 | 1 | 1 | 1 |

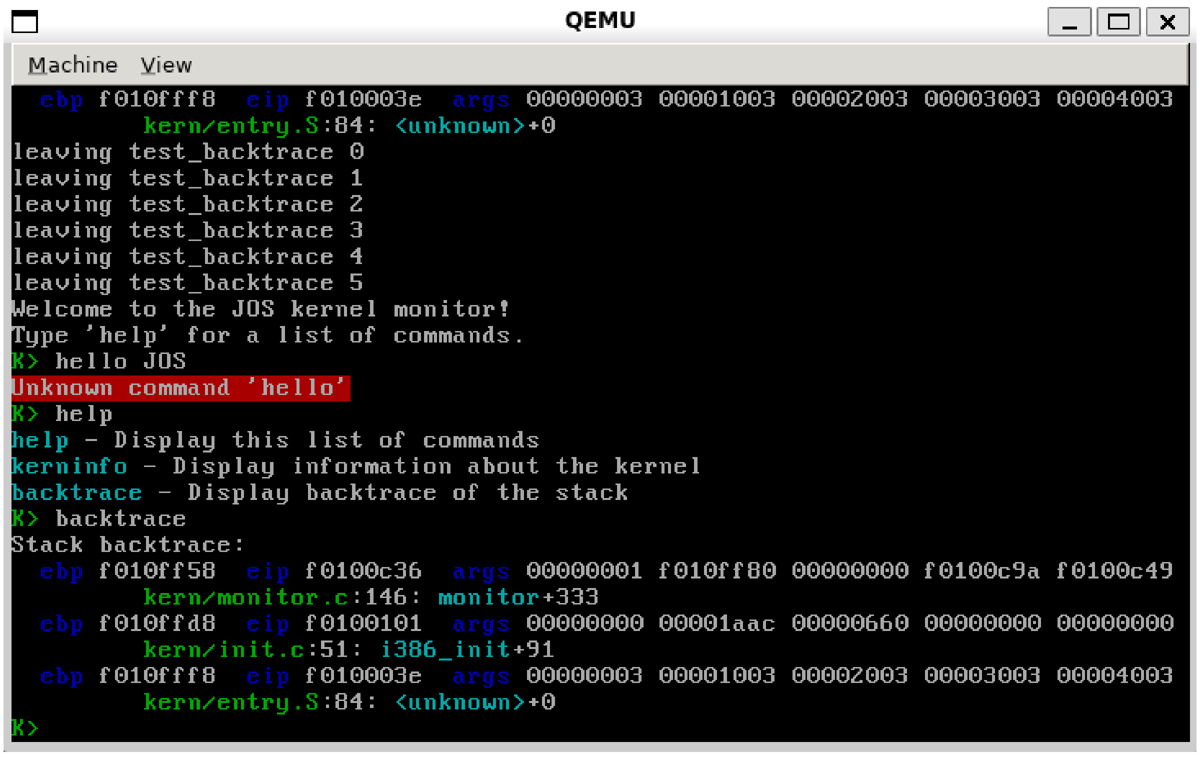
CGA palette internal bit arrangement (4-bit RGBI) [12]

And accidentally (although I infer that it might not be that accidental), the binary color representation of CGA is exactly the same as the ones place of Ps in ANSI escape sequence.

So the rest comes natural. I implement a simple status machine of ANSI espace sequence at `cons_putc` in `kern/console.c`. It only recognize the ESC[Ps;Ps;...Psm pattern described above but this should be sufficient for this challenge. A static variable named `color` is maintained as the color indicated by the most recent ANSI escape sequence.

Now, to print some text in specific color, we just need to add the indication before the text and "ESC[0m" after it to reset the `color`.

```
buf = readline("\033[32mK>\033[0m "); // green prompt
// \033 is the octal ASCII code for ESC
```



```
  ebp f010fff8  eip f010003e  args 00000003 00001003 00002003 00003003 00004003
         kern/entry.S:84: <unknown>+0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> hello JOS
Unknown command 'hello'
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Display backtrace of the stack
K> backtrace
Stack backtrace:
  ebp f010ff58  eip f0100c36  args 00000001 f010ff80 00000000 f0100c9a f0100c49
         kern/monitor.c:146: monitor+333
  ebp f010ffd8  eip f0100101  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:51: i386_init+91
  ebp f010fff8  eip f010003e  args 00000003 00001003 00002003 00003003 00004003
         kern/entry.S:84: <unknown>+0
K>
```

## Reference

[1] Brennan's Guide to Inline Assembly

[2] Intel 80386 Reference Programmer's Manual

[3] Why BIOS need to compare a value in (seemly) randomized address to zero in the second instruction?

[4] Low Level Programming Basic Concepts

[5] Phil Storrs PC Hardware book, The more common I/O address assignments

[6] BIOS reads twice from different port to the same register in a row

[7] bootloader - switching processor to protected mode

[8] Lecture 7: System boot

[9] 68 - 0x44 - N_SLINE

[10] ANSI escape sequences

[11] Colour Graphics Adapter: Notes

[12] *IBM Personal Computer Hardware Reference Library: IBM Enhanced Color Display*