

# An Introduction to i386 Boot loader Programming

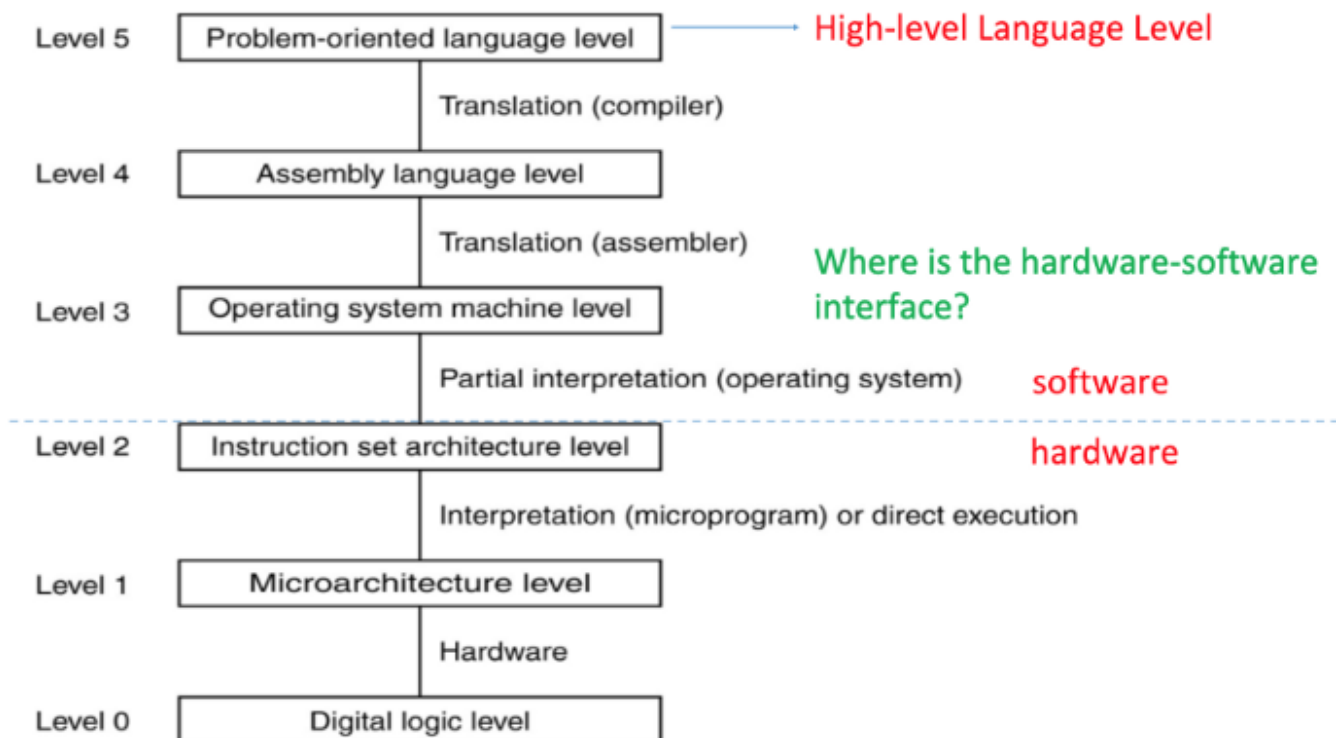


Josh Schiavone

Jul 17 · 5 min read

It seems like right when you turn on your PC, magic happens. It's almost like regular users believe anything below high-level software programs are almost impossible to comprehend. However, that's not the case and I believe it's crucial for a programmer to understand the levels of computing from the hardware all the way up to everyday programs.

Below is a comprehensive diagram on the computer architecture hierarchy:



<https://media.cheggcdn.com/media%2F387%2F387382a8-1961-4d3d-9d3c-3c73d85537e0%2Fphp4pvBdk.png>

Let's break it down.

Level 0: Bare-metal. circuit boards, memory buses, etc.

Level 1 & 2: Micro-architecture. CPU instruction sets, memory communication.

Level 3: The Operating System (i.e. Windows, Linux, etc). Found right above the firmware and the BIOS (Basic Input Output System).

Level 4: Assembly. A human readable language that translates to machine code that the CPU can execute.

Level 5. High-level Programming (C/C++, Java, Python). Trivial for humans to grasp that translates down to assembly language, that translates all the way down to machine code.

In this write up, we will be working in the level 3–4 hierarchy space.

. . .

## Requirements

A) Nasm: The net wide assembler is the most popular assembler and disassembler for the Intel x86 CPU architecture. It is used to write 16-bit, 32-bit and 64-bit programmers.

B) QEMU: A simplistic virtualization program for Linux to run operating systems as virtual machines, without boot-able media.

### Installation:

Ubuntu & Debian:

```
sudo apt install nasm qemu
```

Arch Linux:

```
sudo pacman -S nasm qemu
```

OSX:

```
sudo brew install nasm qemu
```

. . .

## Core Concepts

### Absolute Addresses:

An absolute address can be defined with the assembly “org” instruction which allows us to set our location counter address to execute our code at link time. Let’s take a look at the MBR’s process from a memory standpoint.



The master boot record has a maximum fixed size of 512 bytes. In all cases, the boot sector is loaded at address 0x7c00. We will declare our MBR entry point at this address in the code.

## Magic Numbers:

Magic numbers are the final two bytes of the Master Boot Record (511–512) which must be written as a hexadecimal value to initialize the boot sector process. This section of the MBR must contain 'AA' and '55'. These two bytes classify it as a valid MBR.

. . .

## x86 Assembly Concepts

Basic command breakdown:

Below I will demonstrate the basic assembly commands that we will use to write a simple boot loader.

`mov register, value ; simply stores a value into the given register`

`jmp label: ; jump to a specified point (i.e, function or  
back to a loop)`

`call function: ; executes a defined function`

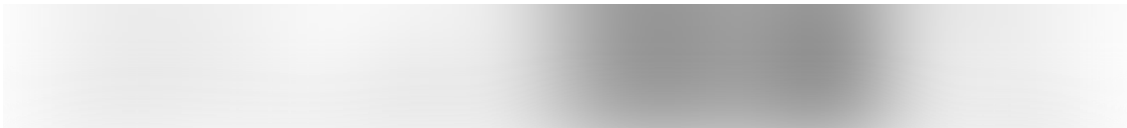
`cmp dest, src: ; will compare two operands`

`jle label: ; will jump back somewhere if previous compare is  
less or equal. (l = less, e = equal)`

`int value: ; calls a kernel interrupt`

## Registers





[https://www.tutorialspoint.com/assembly\\_programming/images/register1.jpg](https://www.tutorialspoint.com/assembly_programming/images/register1.jpg)

These are our general purpose assembly registers for our CPU to store data. We will be programming the boot sector in 16-bit mode so we will stick with, AH, AL (8-bit), BH, BL (8-bit). We won't be touching the counter and data registers but we certainly can if we wanted to. To view all the other CPU registers, they can be found here:

[https://wiki.osdev.org/CPU\\_Registers\\_x86](https://wiki.osdev.org/CPU_Registers_x86).

. . .

## Programming The Boot loader

Firstly, we will create a boot.asm file where we will write our boot loader code.

To start off, we will write an assembly program to print a character to QEMU.

```
[org 0x7c00] ; MBR start address

mov ah, 0x0E ; BIOS command to move our cursor forward
mov al, 'X'   ; Store the character to print in the al register
int 0x10     ; BIOS interrupt - equivalent to print function

jmp $

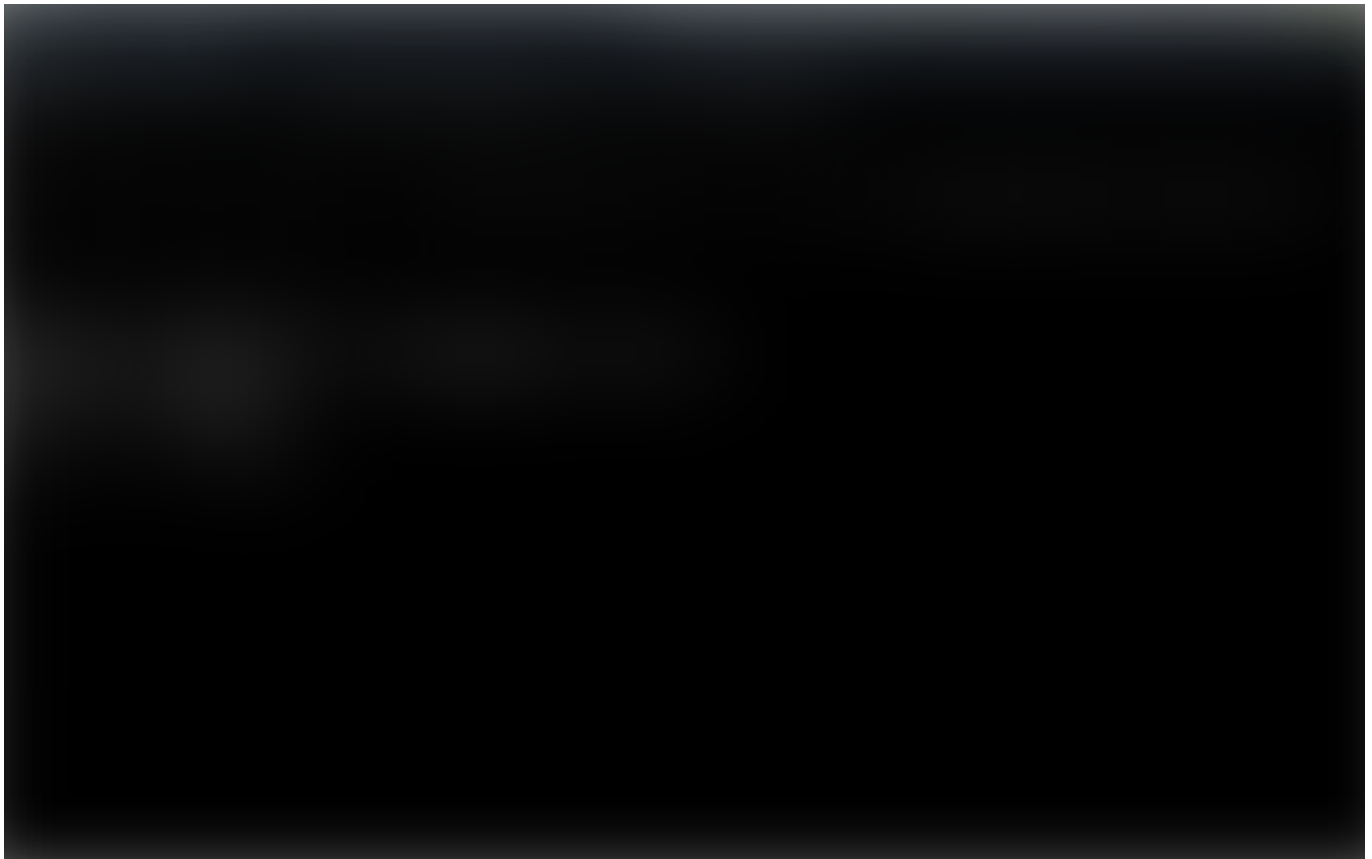
times 510-($-$$) db 0 ; byte padding
dw 0xAA55             ; magic MBR number
```

Assemble the program:

```
nasm boot.asm -f bin -o boot.bin
```

Run the boot sector with QEMU:

```
qemu-system-x86_64 -fda boot.bin
```



Awesome. We now printed the character 'X' from the OS layer. Although, it's not very smart storing single characters in separate memory addresses and combining them as strings. Let's write a function to print each character in a string, using loops.

```
[org 0x7c00] ; MBR start address
```

```
mov si, STR ; store our first string in the si register  
call printf
```

```
mov si, STR_B ; store our second string in the si register  
call printf
```

```
jmp $ ; infinite loop
```

```
printf:  
  pusha ; push everything onto the stack, only 16-bit mode  
str_loop: ; loop function  
  mov al, [si] ; move first char of STR to a-low  
  cmp al, 0 ; look for null terminator  
  jne print_char ; continue to print each char in STR  
  popa ; pop all stack values
```

```
ret
```

```
print_char:
```

```
mov ah, 0x0E ; call our BIOS code for cursor shift
```

```
int 0x10
```

```
add si, 1 ; move forward memory location by one
```

```
jmp str_loop
```

```
STR: db 'Bootsector loaded successfully', 0x0A, 0x0D, 0
```

```
STR_B: db "Dummy OS – Power off the machine to load a real OS", 0x0A,  
0x0D, 0
```

```
; padding & magic number
```

```
times 510-($-$$) db 0
```

```
dw 0xAA55 ; initialize boot sector
```

Above will produce the following output:



We now booted into a BIOS that does absolutely nothing but shows strings. From this point, the opportunities are endless. You can now start developing an operating system from the ground up that actually serves a purpose.

. . .

## Conclusion

In this write up, we covered the computer architecture hierarchy, absolute addresses, magic numbers, basic assembly instructions, CPU registers, and actually developing a minimalist boot loader.


If you're interested to see some of my other projects, feel free to visit my GitHub:  
<https://github.com/josh0xA/>.


2020 © Josh Schiavone

[Bootloader](#)   [Assembly](#)   [Josh Schiavone](#)   [Programming](#)   [How To Make A Bootloader](#)

[About](#)   [Help](#)   [Legal](#)

Get the Medium app

 A button that says 'Download on the App Store', and if clicked it will lead you to the iOS App store

 A button that says 'Get it on, Google Play', and if clicked it will lead you to the Google Play store