Boot Loader: Where is Assembly relevant in 2017?

Chris Aring

Focusing on bootloaders, I created a bootloader that loads an x86 instruction set architecture operating system

- Device drivers
- Interrupt handlers
- Programs that make use of instructions not implemented in a compiler such as a bitwise rotation
- Code that requires ultra-optimization
- Medical equipment
- Autopilot and various aircraft programs
- Bootloaders
- Computer Viruses

Assembly Languages

Assembly is not a single language, it is a group of low-level programming languages that have a strong connection to their specific architecture's machine code instructions.

Background

When a computer boots the BIOS has control but does not know how to load an operating system or where it is located. After it runs all of its checks and various operations, it hands over control to the boot sector which is located in the first sector of the disk. The boot sector is 512 bytes and ends with 0xAA55. The BIOS will check that bytes 511 and 512 are set to be 0xAA55 to make sure that the boot sector is actually bootable and working. It will then hand over control to the boot loader located in the 512-byte boot sector which can either hand over control to another bootloader or boot into the operating system.

The Environment

CAN BE DIFFICULT!

Basic Bootloader

Blank Boot Sector:

```
e9 fd ff 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 28 more lines with sixteen zero-bytes each ]
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
```

Create your own!

Some Useful tools:

● TIMES	- Produce Multiple Copies of an Instruction
● $	- Current Address
● $$	- Base Address of Current Section

A basic layout

```
loop:
    jmp loop

times 510-($-$$) db 0
dw 0xaa55
```

Compile & Boot

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

```
qemu-system-i386 boot.bin
```

Print

To print text to the screen we can set register *ah* to 0x0e which tells it we want to run the Display Character function. We then set register *al* to the letter we want to type and call BIOS interrupt 0x10 for video services.

Access Memory

Set a few registers to hold information about the disk like the cylinder, sector, and head locations. We raise BIOS interrupt 0x13 which is a code for low-level disk services.

Protected Mode

Necessary to execute 32-bit code and load our kernel.

In this mode we don't have access to the BIOS interrupts but we have access to many more features such as paging and virtual memory.

Before the Switch

1. Global Descriptor Table
2. Disable interrupts using the *cli* flag
3. Load the GDT
4. Set the CPU control register
5. Jump far to flush the processor's pipeline
6. Update a few registers
7. Update the stack and move to our first code.

GDT

The Global Descriptor Table is a data structure that defines characteristics of different memory areas such as the base address, size, and access privileges.

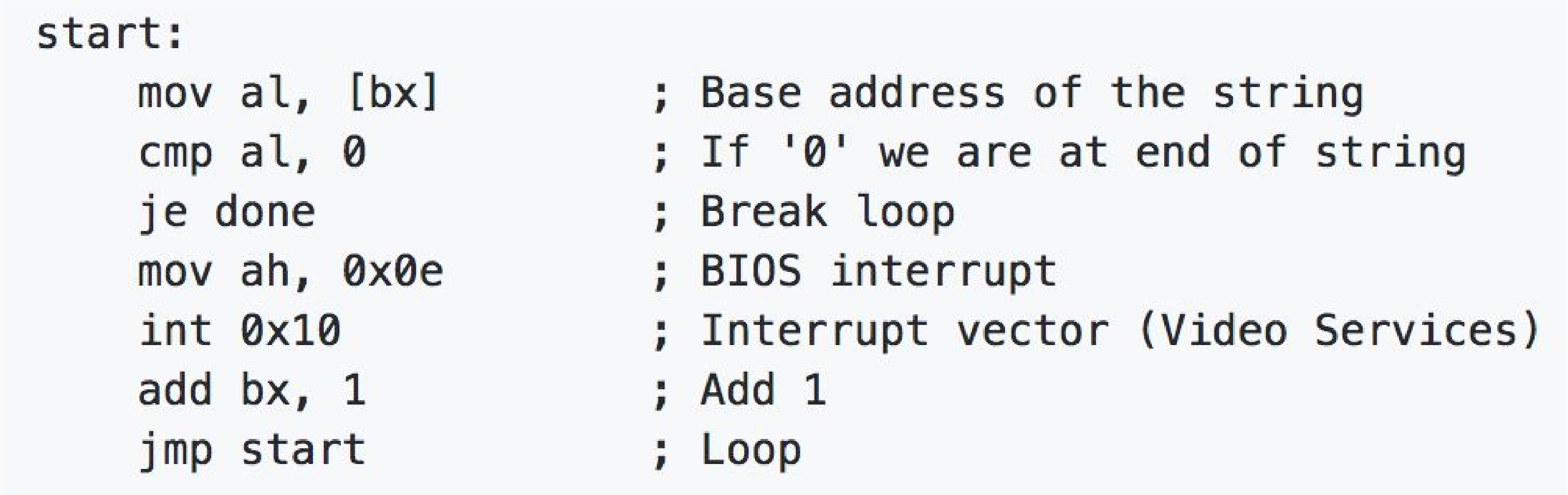
GDT Continued

The GDT was the most confusing part of the project because of how it is split up.

I decided to split it into a data and code section.

Protected Mode Continued

We can now make a new function to print to the window since we can directly access VGA video memory.



```
start:
    mov al, [bx]        ; Base address of the string
    cmp al, 0           ; If '0' we are at end of string
    je done             ; Break loop
    mov ah, 0x0e        ; BIOS interrupt
    int 0x10            ; Interrupt vector (Video Services)
    add bx, 1           ; Add 1
    jmp start           ; Loop
```

The Kernel

SImple - Just print text to window.

Written in C.

Compiling and Linking

Can be difficult if GCC Cross-Compiler was not set up correctly.