BOOT LOADER

Simple boot loader to print “Hello World”

In this section, we see assembly program for creating a **16-bit bootloader**, a minimal program that runs when a computer boots. It is written for the x86 architecture and executes in real mode.

x86 CPU and memory layout

)Primary memory (IVT, BDA-Boot Area, - Conversational Memory-EBDA, ROM BIOS area)

- 0-3FF IVT
- 400 4FF BDA
- 4FF above Bios Stack Area
- 7C00 (512Bytes) Boot block
- 640K Conventional memory (0x – 0x9FFFF)
- 0x90000 – 0x9FFFF – EBDA
- 0xA0000 - 0xBFFFF – VGA frame buffer and Char buffer area
- 0xC0000 – 0xDFFFF – Option ROM area
- 0xE0000 – 0xFFFFF – ROM BIOS
- 0xFFFFF – 0x10FFEF -UMA

Setup

- NASM (**Netwide Assembler**) is a popular assembler for the x86 architecture.
- It converts assembly language source code into machine code or flat binaries.

- QEMU (**Quick Emulator**) is a versatile virtualization and emulation tool.
- The `qemu-system-x86` package emulates x86 CPUs and systems, allowing you to run x86-based operating systems or bare-metal programs without real hardware

sudo apt install nasm

sudo apt install qemu-system-x86

General-Purpose Registers

These are versatile and used for arithmetic, data storage, and memory addressing.

1. `AX` **(Accumulator Register):**
   - Used as a general-purpose register in this program.
   - Acts as an intermediary when initializing segment registers (`DS`, `ES`, `SS`).
2. `SI` **(Source Index Register):**
   - Used as a pointer to the message string in memory.
   - The `SI` register points to the address of the message (`msg`), enabling string traversal.
   - The `lodsb` instruction automatically uses `SI` to read a byte from the address `DS:SI` and increments `SI` to point to the next byte.
3. `SP` **(Stack Pointer):**
   - Points to the top of the stack in memory.
   - The program sets `SP` to `0x7c00`, which is the bootloader's starting address.

4. `AL` **(Lower Byte of AX):**
    - Used to store individual characters from the message string.
    - `lodsb` loads a character into `AL`, which is then passed to the BIOS interrupt (`int 0x10`) for display.
5. `AH` **(Upper Byte of AX):**
    - Used to specify BIOS interrupt functions.
    - In this code, it is set to `0x0E` to indicate the teletype output function of BIOS interrupt `0x10`:

Segment Registers

These registers are used to define the base address of memory segments. In real mode, the CPU combines segment registers with offsets to form physical memory addresses.

1. `DS` **(Data Segment):**
    - Points to the segment where data (like the message string) resides.
    - Initialized to `0x00` to reference the bootloader's memory:
2. `ES` **(Extra Segment):**
    - Often used for additional data operations.
    - In this program, `ES` is initialized to `0x00`, though it is not actively used.
3. `SS` **(Stack Segment):**
    - Defines the segment where the stack resides.
    - Set to `0x00` so that the stack operates within the same segment as the rest of the program.

Instruction Pointer (Implicit Register)

- The `IP` (Instruction Pointer) keeps track of the next instruction to execute.

- It works implicitly and is updated automatically by the CPU as the program executes instructions or jumps.

Flags Register

- Contains status flags and control flags that reflect the outcome of operations or control the CPU behavior.

1. **Interrupt Flag (IF):**
    - Managed explicitly in the code:
        - `cli`: Clears the interrupt flag, disabling interrupts.
        - `sti`: Sets the interrupt flag, enabling interrupts.

Code Flow:

- **Interrupt Handling:**
    - `cli`: Disables interrupts temporarily to avoid interference during setup.
    - `sti`: Re-enables interrupts after initialization.
- **Segment Initialization:**
    - `mov ax, 0x00`: Clears the `AX` register (used as a general-purpose register).
    - `mov ds, ax`, `mov es, ax`, `mov ss, ax`: Set the data segment (`DS`), extra segment (`ES`), and stack segment (`SS`) to 0. This standardizes segment registers to a known state.
- **Stack Setup:**
    - `mov sp, 0x7c00`: Points the stack pointer (`SP`) to `0x7c00`. This is a safe place for the stack, avoiding overwriting the bootloader code.
- **Message Address Setup:**
    - `mov si, msg`: Loads the address of the message (`msg`) into the `SI` register, which is used as a pointer to iterate through the string

- **Loop through the message:**
  - `lodsb`: Loads the next byte from memory (pointed to by `SI`) into `AL` and increments `SI`.
  - `cmp al, 0`: Checks if the character is the null terminator (`0x00`), marking the end of the string.
  - `je done`: If null terminator is found, jump to the `done` label to finish execution.
- **BIOS Teletype Output:**
  - `mov ah, 0x0E`: Specifies the teletype function of BIOS interrupt `0x10`.
  - `int 0x10`: Displays the character in `AL` on the screen.
- **Repeat:**
  - `jmp print`: Loops back to process the next character.
- `cli`: Disables interrupts again for safety.
- `hlt`: Halts the CPU, effectively stopping the program.

`msg`: Stores the string `"Hello World!"` with a null terminator (`0x00`) to mark the end.

- **Padding:**
  - `times 510 - ($ - $$) db 0`: Fills unused space in the 512-byte boot sector with zeros. `$` is the current address, and `$$` is the start address.
  - Ensures the bootloader is exactly 512 bytes.
- **Boot Signature:**
  - `dw 0xAA55`: Marks the last two bytes with the mandatory boot sector signature (`0xAA55`). BIOS checks this to verify the sector is bootable.

Purpose of the Makefile

- Automates the process of assembling the bootloader (from `boot.asm`) into a binary format (`boot.bin`).
- Simplifies cleanup by removing generated binary files.

1. Target: `all`

- **Purpose**: Builds the bootloader binary.
- **Command**:
  - `nasm`: Invokes the NASM assembler to compile the assembly source file.
  - `f bin`: Specifies the output format as a flat binary file (raw executable without headers).
  - `./src/boot.asm`: Path to the assembly source file.
  - `o ./bin/boot.bin`: Specifies the output binary file's location and name.

2. Target: `clean`

- **Purpose**: Removes the generated binary file.
- **Command**:
  - `rm -f ./bin/boot.bin`:
    - `rm`: Deletes files.
    - `f`: Forces deletion without prompting for confirmation, even if the file doesn't exist.
    - `./bin/boot.bin`: Specifies the file to be deleted.

Go to bin folder in terminal and execute the either of this command to create or to clean bin file:

1. make all
2. make clean

Then run the below command in the same bin directory to see the output,

qemu-system-x86_64 -hda ./boot.bin