

# HELLO WORLD!!!

When a computer is booted, the first thing that happens is the initialization and testing of the hardware by the BIOS (Basic Input/Output System). The BIOS is a firmware that is built into the computer's motherboard and is responsible for providing low-level services or functions that can be used by programs.

After testing the hardware, the BIOS then finds the boot device, which is typically a hard drive or a USB drive. It reads the first sector of the boot device into memory location 7c00, which is also known as the master boot record (MBR). The MBR contains information about the partition table and the boot loader.

The boot loader is a program that is responsible for loading the operating system into memory and transferring control to it. In the video referenced, a custom boot loader program is being written and will be written into the first sector of the disk, so that it can be executed when the computer is booted.

When the BIOS reads the boot loader program into memory, it transfers control to it. At this point, the boot loader program has control of the computer and can perform any necessary tasks to prepare for loading the operating system.

When the operating system is loaded, it goes through three different modes: real mode, protected mode, and long mode. Real mode is the initial mode that the CPU is in when the computer is booted. In this mode, the CPU can access the first 1 MB of memory and can only execute 16-bit code.

Protected mode is the mode that the CPU switches to in order to prepare for long mode. In this mode, the CPU can access all of the memory in the computer and can execute 32-bit code. It also provides memory protection, which prevents programs from accessing memory that they shouldn't.

Long mode consists of two sub-modes: 64-bit mode and compatibility mode. 64-bit mode is the mode that the operating system is designed for and can execute 64-bit code. Compatibility mode is a mode that allows 16-bit and 32-bit programs to run on a 64-bit operating system without the need for recompilation.

In conclusion, understanding the boot process and the different modes of operation that the CPU can be in is important for developing operating systems and low-level software. It is necessary to understand the limitations and capabilities of each mode in order to write efficient and secure code.

## **BIOS SERVICES**

Throughout our project, we will rely on various services to accomplish essential tasks within our code. These services include functions for printing characters on the screen, loading files from the hard disk into memory (such as the kernel file), retrieving memory information from the computer, and setting the video mode.

One of the primary services we utilize is the print function, which allows us to display characters on the screen. However, when operating in long mode, the traditional print functions provided by the BIOS are no longer accessible. To overcome this limitation, we employ an alternative approach by setting a specific video mode that enables us to print characters on the screen within that mode.

In addition to printing functions, we make use of services that facilitate loading files from the hard disk into memory. This is particularly crucial for loading the kernel file, which serves as the core of the operating system. By loading the kernel file into memory, we ensure that the necessary instructions and data are readily available for the system to execute.

Furthermore, our code incorporates services for obtaining memory information about the computer. This allows us to gather essential details about the system's memory layout, available memory segments, and other relevant information. Such knowledge is crucial for effective memory management within the operating system.

Lastly, we employ services to set the video mode, which determines the display configuration and capabilities. By selecting a specific video mode, we can customize the appearance and behavior of the screen, enabling us to present information and interact with the user in a desired format.

By leveraging these services, we are able to accomplish key functionalities within our project, enabling the creation and operation of a custom Linux kernel from scratch.

## **UEFI - UNIFIED EXTENSIBLE FIRMWARE INTERFACE**

In the initial stages of our project, we focus on the real mode of operation. This mode allows us to perform tasks such as loading the kernel into memory, obtaining hardware information, and setting up the basic system environment. In real mode, the operating system runs in 16-bit mode, utilizing specialized segment registers.

The segment registers, namely cs (Code Segment), ds (Data Segment), es (Extra Segment), and ss (Stack Segment), are 16-bit registers that play a crucial role in

addressing memory. The addressing scheme in real mode follows a specific format: Segment Register:Offset (logical address).

To convert a logical address to a physical address, we utilize the segment registers. We shift the value stored in a segment register left by 4 bits, effectively multiplying it by 16. Then, we add the offset value to the result, which yields a 20-bit physical address.

This addressing mechanism allows us to access different memory segments and manipulate data effectively within the real mode. It sets the foundation for loading the kernel into memory, managing memory allocation, and accessing hardware devices. However, as we progress further into the project, we transition from real mode to long mode, where the operating system operates in a 64-bit environment with expanded capabilities and address space.

```
mov ax,[ds:50h]  ds=0x1000
```

In the provided code snippet, we encounter the instruction "mov ax,[ds:50h]" where ds is assigned a value of 0x1000. In x86 assembly language, ds is one of the segment registers that is 16 bits in size.

To understand the effect of this instruction, we need to consider the address calculation. The value of ds, which is 0x1000, is multiplied by 16 (or shifted left by 4 bits) to convert it to a physical address. The resulting physical address is then added to the offset 0x50.

In this case, the calculation would be:  $0x1000 * 16 + 0x50 = 0x10050$ .

Therefore, the instruction "mov ax,[ds:50h]" copies the data stored at the memory address 0x10050 into the register ax. By accessing memory through the combination of the ds segment register and the provided offset, we can retrieve the desired data and store it in the specified register for further processing within the program.

the addresses in this example are hexadecimal value. We can use prefix 0x or suffix h to represent a hexadecimal number.

```
pop bx  ss=0x100  sp=0x500
```

In the given code snippet, we encounter the instruction "pop bx" with the information that ss (the stack segment register) is assigned a value of 0x100 and sp (the stack

pointer register) is assigned a value of 0x500.

When we perform a "pop" operation, it retrieves the topmost value from the stack and stores it in the specified register, in this case, bx. The address of the value being popped from the stack can be determined by combining the value of ss with the value of sp.

In this example, the calculation would be:  $ss * 16 + sp = 0x100 * 16 + 0x500 = 0x1500$ . Therefore, the address of the memory we actually reference when executing the "pop bx" instruction is 0x1500.

It is worth mentioning that the same rule applies to other segment registers, such as CS (Code Segment) and ES (Extra Segment), when calculating the memory address.

Additionally, when accessing data without explicitly specifying a segment register, the default segment register used is ds. So, in the example "mov ax,[ds:50h]", if the ds register is not specified explicitly, it will be assumed as the default register for accessing the data.

These calculations and considerations regarding the segment registers and memory addresses are essential for proper memory management and accessing the correct data within a program.

### **Real Mode:**

In real mode, the processor operates with 16-bit registers as the primary general-purpose registers. These registers include:

- 8-bit registers: al (accumulator low), ah (accumulator high), bl (base low), bh (base high)
- 16-bit registers: ax (accumulator), bx (base), cx (counter), dx (data)

In real mode, these registers are sufficient for performing basic operations and storing data. However, in 64-bit mode, the processor introduces extended registers, such as rax, rbx, rcx, rdx, which are 64 bits in size. These extended registers provide more capacity for working with larger memory addresses and data sets.

In the context of the boot file discussed, which operates in real mode, we don't utilize the 64-bit registers (rax, rbx, rcx, rdx). Instead, we rely on the 16-bit registers (ax, bx, cx, dx) for handling data and performing necessary operations within the limitations of real mode.

It's important to note that as we transition into 64-bit mode and work with more advanced features of the processor, the extended registers become available and provide greater flexibility and capabilities for the operating system and applications.

## CODING PART

```
[BITS 16]
[ORG 0x7c00]

start:
    xor ax,ax
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov sp,0x7c00

PrintMessage:
    mov ah,0x13
    mov al,1
    mov bx,0xa
    xor dx,dx
    mov bp,Message
    mov cx,MessageLen
    int 0x10

End:
    hlt
    jmp End

Message:    db "Hello"
MessageLen: equ $-Message

times (0x1be-($-$$)) db 0

    db 80h
    db 0,2,0
    db 0f0h
    db 0ffh,0ffh,0ffh
    dd 1
    dd (20*16*63-1)

    times (16*3) db 0

    db 0x55
    db 0xaa
```

The first part of our boot code includes two directives: "bits" and "org". The "bits" directive informs the assembler that our boot code is intended to run in 16-bit mode. This is important because the real mode, in which the boot code operates, is a 16-bit mode.

The "org" directive specifies the memory address at which the code is expected to be loaded and executed. In our case, we set it to 7c00. It is crucial to specify this address correctly because the BIOS loads the boot code from the first sector of the storage device into memory address 7c00 and transfers control to that location. If we fail to specify the correct address, any memory references within our code would be incorrect when the program is running.

By using these directives, we ensure that our boot code is set up properly to run in the expected 16-bit mode and at the correct memory address, allowing it to be loaded and executed successfully by the BIOS.

## Now moving to the Assembly code

We use label start to indicate that this is the start of our code.

This block of code initialize the segment registers and stack pointer

1. `[BITS 16]` : This directive sets the assembler mode to 16-bit. It specifies that the code will be executed in the real mode of the processor, where addresses and data are 16 bits wide.
2. `[ORG 0x7c00]` : This directive sets the origin of the program to 0x7c00. This is where the BIOS loads the bootloader into memory and starts executing it.
3. `start:` : This is a label that marks the beginning of the program. The label is used to reference the location in the code.
4. `xor ax, ax` : This instruction performs a logical exclusive OR (XOR) operation between the AX register and itself, which sets the value of the AX register to zero. This is a common way to clear a register's contents, as the XOR of a value with itself always results in zero.
5. `mov ds, ax` : This instruction moves the value of the AX register into the data segment (DS) register. The DS register is used to hold the segment address of the data segment. In real mode, the memory is divided into segments of 64 kilobytes each, with each segment having a base address specified by a segment register. By setting the DS register to zero (the value of the AX register), we set the base address of the data segment to zero, which means that any data accesses using the DS register will start from the beginning of memory.
6. `mov es, ax` : This instruction moves the value of the AX register into the extra segment (ES) register. The ES register is used for accessing memory in other

segments than the data segment. Since we don't need to access memory in other segments in this bootloader program, we can set the ES register to zero as well.

7. `mov ss, ax` : This instruction moves the value of the AX register into the stack segment (SS) register. The SS register holds the segment address of the stack segment. By setting the SS register to zero, we set the base address of the stack segment to zero, which means that the stack starts at the beginning of memory. This is necessary for the bootloader program to have access to the entire memory space for the stack.
8. `mov sp, 0x7c00` : This instruction sets the stack pointer (SP) to the end of the bootloader program, at address 0x7c00. The SP register holds the offset of the current top of the stack, relative to the base address of the stack segment. By setting the SP register to 0x7c00, we set the current top of the stack to the end of the bootloader program, which is the starting address of the program. This means that any subsequent push operations on the stack will decrement the SP register and write values to lower addresses in memory.

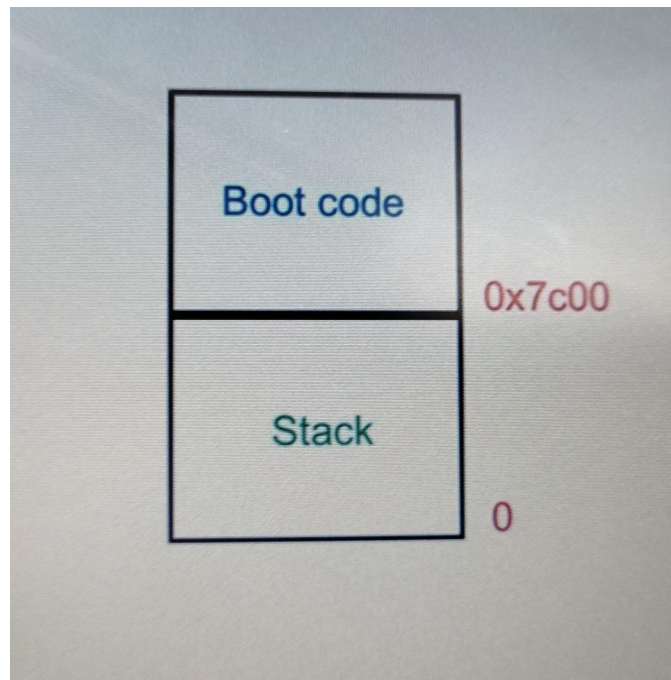
Overall, these instructions set up the environment for the bootloader program to execute in real mode, initialize the registers and segments, and set up the stack at the beginning of the program's memory space.



## SEGMENTSSS

1. Code Segment (CS): This segment register holds the base address of the segment where the program code is stored. The IP register (instruction pointer) holds the offset of the current instruction relative to the base address of the code segment. In the bootloader program, the code is stored at the beginning of memory, so the CS register is set to zero.
2. Data Segment (DS): This segment register is used to hold the base address of the segment where data is stored. In the bootloader program, the DS register is initialized to zero, which means that any data accesses using the DS register will start from the beginning of memory. This segment is used for accessing variables, arrays, and other data stored in memory.
3. Stack Segment (SS): This segment register holds the base address of the stack segment, which is used for storing temporary data, function arguments, and return addresses. The SP register (stack pointer) holds the offset of the current top of the stack relative to the base address of the stack segment. In the bootloader program, the stack is initialized to the end of the program memory, so the SS register is set to zero, and the SP register is set to the address of the end of the program.
4. Extra Segment (ES): This segment register is used for accessing memory in segments other than the data segment. In the bootloader program, the ES register is also initialized to zero, as we don't need to access any other memory segment.





The boot code starts at 7c00 and the stack is set up below 7c00.

In real mode, the stack pointer points to 2 bytes below the memory address 7c00 because the default operand size for push and pop instructions is 2 bytes in 16-bit mode. As we push data onto the stack, the stack pointer (SP) register decrements to accommodate the new data.

To print characters on the screen, we utilize BIOS services through software interrupts. These interrupts can be accessed by using the "int" instruction along with the corresponding interrupt number. In our case, the print function is accomplished by calling interrupt 10. It is important to note that the interrupt number is represented in hexadecimal format.

When using the print function, we need to provide certain parameters to specify the desired operation. While there are various parameters available, in the case of printing a string, we focus on the parameters required for this specific task.

Overall, by understanding the stack behavior and utilizing the appropriate BIOS interrupt, we can effectively print characters on the screen using the print function in real mode.

1. `mov ah, 0x13` : This instruction moves the value 0x13 (19 in decimal) into the AH register. In this context, 0x13 represents the video services function number for

displaying text in the BIOS video mode. Here 13 is a function code which is for printing string

2. `mov al, 1`: This instruction moves the value 1 into the AL register. The AL register is used to specify various parameters for video services. In this case, the value 1 indicates that we want to write the string of characters to the screen. **Register AL specifies the write mode, we set it to 1, meaning that the cursor will be placed at the end of the string.**

3. `mov bx, 0xa`: This instruction moves the value 0xa (10 in decimal) into the BX register. The BX register is used as a general-purpose register in this case, and here it is used to hold the page number where the text will be displayed on the screen. 0xa means the character is printed in dark green color

Bh higher part of bx represents the page number and Bl the lower part of bx holds the information of character attributes

4. `xor dx, dx`: This instruction performs an XOR operation between the DX register and itself, effectively setting the DX register to 0. The DX register is used to hold the starting column and row position on the screen where the text will be displayed. Setting it to 0 means the text will start at the top left corner of the screen.

Dh higher part of dx represents rows and Dl represents columns

5 `mov bp, Message`: This instruction moves the memory address of the string "Message" into the BP register. The BP register is used as a base pointer, and here it is used to hold the memory address of the string that will be displayed on the screen.

In case if we want to store the data directly we need to use square bracket to it `mov bp, [Message]`

6. `mov cx, MessageLen`: This instruction moves the value of "MessageLen" into the CX register. The CX register is used to hold the length of the string to be displayed. The specific value of "MessageLen" is not shown in the given code snippet, but it would typically represent the length of the string stored in the "Message" variable. **cx is used to represent number of characters to print**

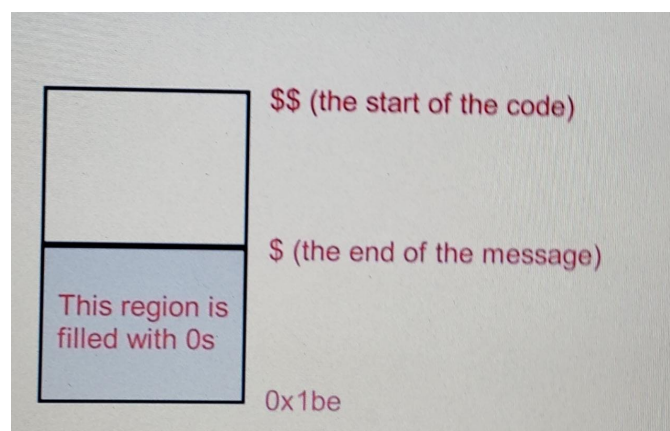
`hlt` instruction places the processor in a halt state.

When an interrupt is triggered, the processor will execute the instruction following the halt instruction, which in this case is a jump instruction. This jump instruction redirects the execution flow back to the halt state, creating an infinite loop.

The directive "times" is used to repeat a command a specified number of times. In this example, the expression specifies how many times the "db" (define byte) command is repeated. The double dollar sign indicates the beginning of the current section.

Since we only have one section in this example, the double dollar sign marks the start of our code. On the other hand, the expression with a dollar sign minus double dollar sign calculates the size from the start of the code to the end of the message.

By using this expression, we can repeat the "db" command to fill the space between the end of the message and the offset 1be with zeros. This ensures that the reserved space is properly filled and aligned within the code.



In offset 1be we have what is being called as the partition entries , there are four entries with each entries being 16 bytes each

```
db 80h
db 0,2,0
db 0f0h
db 0ffh,0ffh,0ffh
dd 1
dd (20*16*63-1)
```

We can see that we have only defined the first entry and rest of the entries are marked as 0

The pattern is as follows

- First byte is boot indicator we simply set it as 80 that this is a bootable partition
- The next three bytes constructs a starting c,h,s value. c stands for cylinder , h heads and s sector

- starting chs / type / ending chs
- The first byte represents head value
- The second byte is divided into two parts bits [0-5] is used as sector value [6-7] is cylinder value
- The last byte holds the lower 8 bits of cylinder value
- The head and cylinder values starts from 0 so cylinder 0 is the first cylinder
- Sector values starts from 1 meaning sector 1 is first sector
- Fourth byte is partition type we set it to f0 then the next three bytes construct the ending c,h,s value and here we simply set them to ff
- ff is max value we can set in a byte
- The next double word represents LBA address of starting sector
- LBA means logical block address
- In boot process we load our file using LBA instead of c,h,s value
- The last double word specifies how many sectors the partition has
- The value set here is 10mb
- the value in the partition entry does not reflect the real partition.
- The reason we define this entry is that some BIOS will try to find the valid looking partition entries.
- If it is not found, the BIOS will treat usb flash drive as, for example, floppy disk.
- We want BIOS to boot the usb flash drive as hard disk. So we add the partition entry to construct a seemingly valid entries.
- The last two bytes of our boot file is signature which should be 55AA
- Size of boot file is 512 byte and sector size is 512 byte
- When we write the boot file into the boot sector, it gets the same data as boot file.
- With boot the image prepared, we use build script to build our project and write the binary file into boot image.
- If you know the makefile, you can build the project using Makefile.

In the script, we have two commands to write, the first one assembles the boot file.

```
nasm -f bin -o boot.bin boot.asm  
or  
nasm -f bin -o boot.bin /mnt/c/Users/viswa/Desktop/boot/boot.asm
```

In order to specify the desired output file format as a binary file, we utilize the "-o" flag followed by the name of the output file we want to generate. In this particular case, the output file we intend to produce is a binary file, and the input file we aim to assemble is named "boot.asm".

Once the assembly process is successfully completed without encountering any errors, we should observe the generation of a file called "boot.bin". This file contains the assembled binary code that we can work with.

Moving forward, our objective is to write this binary file into the boot image, thereby making it bootable. To achieve this, we utilize the "dd" command. The command structure for this operation is as follows:

```
dd if=boot.bin of=boot.img bs=512 count=1 conv=notrunc
```

In this command, "if" signifies the input file, which is "boot.bin", while "of" represents the output file, which is "boot.img". The "bs" flag is used to define the block size, which in this case is set to 512 bytes. Additionally, the "count" parameter specifies the number of blocks to be copied, with a value of 1 indicating a single block. Finally, the option "conv=notrunc" ensures that the output file is not truncated before writing the data, preserving any existing content within "boot.img".

To enable the status bar in Visual Studio Code, we click on the "View" option in the menu bar and select "Appearance". From the dropdown menu, we choose "Show Status Bar" to display the status bar at the bottom of the screen.

Once the status bar is visible, we may notice the presence of "CRLF" (Carriage Return Line Feed), which represents the line break format used in Windows. However, for compatibility with the build script in Linux, we need to change it to "LF" (Line Feed). We can do this by clicking on the line break indicator in the status bar and selecting "LF".

Next, we need to save the file as a shell script. We can choose the "Save As" option and provide the desired name for the file, such as "build", ensuring that the file

extension is ".sh" to indicate it as a shell script.

Once the build script is saved, we need to open the Ubuntu terminal. We can do this by launching the Ubuntu application or accessing the terminal emulator provided by the Ubuntu operating system.

In the Ubuntu terminal, we can type the appropriate command to execute the build script. The specific command will depend on the location and name of the build script file. For example, if the build script is located in the current directory and named "build.sh", we can run it by entering "./build.sh" in the terminal.

Executing the build script will initiate the desired actions defined within the script, such as assembling files, generating output files, or performing other specified tasks as required for the project.

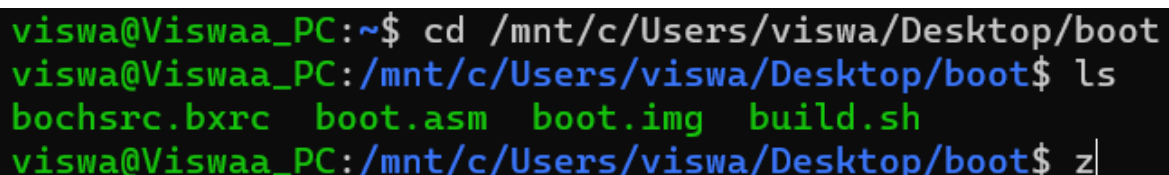
```
cd /mnt/c/Users/viswa/Desktop/boot
```

After opening the Ubuntu terminal, you can use the command "ls" to list all the files in the current directory. This will display the names of the available files, including the boot image file, which is the previously created blank 10 MB image.

In addition to the boot image file, you will also need the Bochs configuration file. This file contains the necessary settings and configurations for the Bochs virtual machine emulator, which will be used to run and test the boot image.

By having both the boot image file and the Bochs configuration file in the same directory, you can proceed with the further steps of the project, such as running the boot image using Bochs and testing the functionality of the Linux kernel that you are developing.

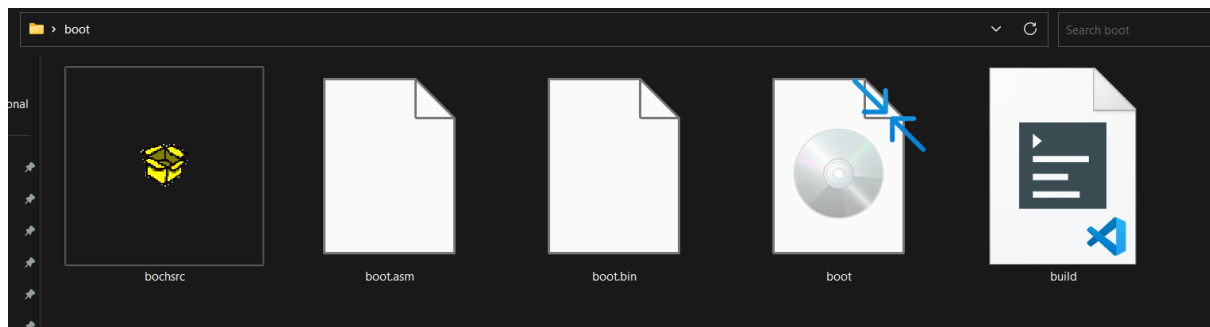
You can find the screenshot attached below



```
viswa@Viswaa_PC:~$ cd /mnt/c/Users/viswa/Desktop/boot
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ ls
bochsrc.bxrc boot.asm boot.img build.sh
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ z|
```

You can find the necessary commands which were executed shown below:

```
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o boot.bin /mnt/c/Users/viswa/Desktop/boot/boot.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=boot.bin of=boot.img bs=512 count=1 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00324975 s, 158 kB/s
```



To inspect the binary file "boot.bin" and view its contents in a readable format, you can use the "hexdump" command with the "-C" option. This option displays the file in a hexadecimal and ASCII representation, allowing you to see the content as both character strings and hexadecimal values.

By running the command "hexdump -C boot.bin", you will be able to examine the contents of the "boot.bin" file in a structured manner. The hexadecimal values will be displayed alongside the corresponding ASCII characters, providing you with a comprehensive view of the file's content. This can be helpful for verifying the correctness of the binary file and analyzing its structure or specific data patterns.

```
hexdump -C boot.bin
```

```
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ hexdump -C boot.bin
00000000  31 c0 8e d8 8e c0 8e d0  bc 00 7c b4 13 b0 01 bb  |1.....|....|
00000010  0a 00 31 d2 bd 1f 7c b9  05 00 cd 10 f4 eb fd 48  |..1...|.....H|
00000020  65 6c 6c 6f 00 00 00 00  00 00 00 00 00 00 00 00  |ello.....|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 80 00  |.....|
000001c0  02 00 f0 ff ff ff 01 00  00 00 bf 4e 00 00 00 00  |.....N...|
000001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |.....U.|
00000200
```

Upon inspecting the contents of the "boot.bin" file using the "hexdump" command, we observe that the last two bytes are represented by the values "55" and "AA". These values, when combined, form the hexadecimal signature "55AA" which serves as a boot sector signature.

The output of the "hexdump" command also provides additional information. The first column displays the memory address in hexadecimal format, and in this case, the address "200" corresponds to the decimal value of 512. The right column shows the ASCII representation of the data, allowing us to identify the message "hello" present in the file.

To execute the program within the virtual machine, we have already copied the Bochs configuration file to the desired directory. However, before running the virtual machine, we need to make some modifications to the configuration file. Opening the file in Visual Studio Code, we locate line 13, where we must update the path to the current directory where the boot image is located. This ensures that the virtual machine can properly locate and load the boot image during execution.

### Before

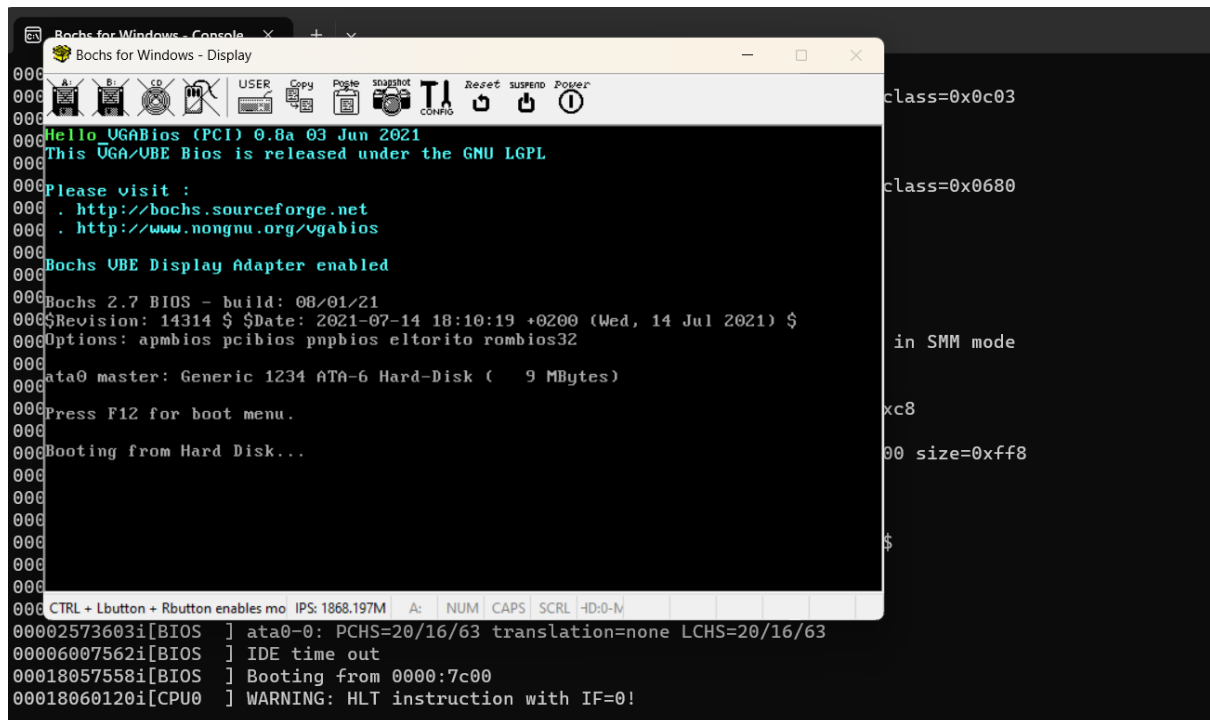
```
ata0-master: type=disk, path="boot.img", mode=flat, cylinders=20, heads=16, spt=63, sect_size=512, model="Generic 1234", biosdetect=auto, translation=auto
```

### After

```
ata0-master: type=disk, path="C:\Users\viswa\Desktop\boot\boot.img", mode=flat, cylinders=20, heads=16, spt=63, sect_size=512, model="Generic 1234", biosdetect=auto, translation=auto
```

Running Bochs is a simple process - we just need to double-click on it. Once Bochs is launched, we can see the message "hello" printed on the screen, indicating that the boot process is successful.





After the initial process of booting from the USB flash drive, the boot loader loaded from the flash drive will proceed to boot from the hard disk. The screen will display the message "Booting from Hard Disk..." as the system transitions to the next stage of the boot process.

At this point, the boot loader will execute the necessary instructions to locate and load the operating system kernel from the hard disk. The boot loader plays a crucial role in initializing the system and transferring control to the kernel, which will then take over and continue the boot process.

Once the boot process is complete, the operating system will be fully loaded into memory, and the computer will be ready for use, allowing users to interact with the system and perform various tasks according to the functionality provided by the installed operating system.

## How to convert it to USB flash drive

To create a bootable USB flash drive in Windows, we can use a tool like Rufus. After opening Rufus, we click on the "SELECT" button and choose the boot image file by clicking "Open". In the "Device" option, we verify that the correct USB device is selected. A warning may appear, but we can simply click "OK" to proceed. The process of creating the bootable USB flash drive is usually fast. Once the bootable USB is created, we insert it into the test computer and power it on.

Since we are running in legacy mode, we need to ensure that the CSM (Compatibility Support Module) is enabled in the BIOS settings if the motherboard has a UEFI system. Typically, the CSM support is enabled by default. We access the BIOS settings, navigate to the CSM option, and ensure that it is enabled. After making the necessary changes, we exit the BIOS settings and reboot the computer.

As we haven't set up the boot device priority, we open the boot menu during the startup process. In this menu, we can choose the device from which we want to boot. Since the test computer does not have a hard disk, the only option available will be the USB flash drive. Once we select the USB flash drive as the boot device, the computer should proceed to boot from it. If everything goes well, we should see the message "HELLO WORLD" printed on the screen, indicating a successful boot.

## **TEST DISK EXTENSION**

In order to efficiently load our kernel from the disk into memory and initiate the jump to the kernel, we decided to test our disk extension service. During the boot process, we rely on the BIOS disk services to facilitate the loading of files from the disk.

To read a file from the disk, we need to provide the CHS (Cylinder - Head - Sector) value, which allows us to locate the specific sector we want to read. However, this process can be time-consuming. To simplify our boot file and enhance its performance, we opted to use the logical block address, which is employed by the disk extension service to access the disk.

We have defined a label called "TestDiskExtension" to designate the portion of our code where we will test and implement the disk extension service. By leveraging this service, we aim to streamline the process of loading our kernel and improve the overall efficiency of our booting mechanism.

```
mov ah,0x41  
mov bx,0x55aa
```

It's important to note that the value of the drive ID is stored in the dl register when the BIOS transfers control to our boot code. Since we will be invoking disk services multiple times later in the boot process, we need to pass the appropriate drive ID to the dl register before each disk service call.

To facilitate this, we define a variable called DriveId and store the value of dl in it. This allows us to conveniently access and utilize the drive ID whenever we need to

make disk service calls. Within the "TestDiskExtension" section of our code, we include the necessary command, using square brackets to access the location where the drive ID value is stored.

By properly managing and utilizing the drive ID, we can ensure that our disk services operate smoothly and accurately during the boot process.

```
mov [DriveId],dl
```

After setting up the appropriate drive ID value in the dl register, we proceed to call interrupt 13h, which is the BIOS interrupt for disk services. By invoking this interrupt, we can access various disk-related services provided by the BIOS.

If the requested disk service is supported by the BIOS, it will be executed, and the carry flag (CF) will be cleared, indicating a successful operation. However, if the requested service is not supported by the BIOS, the carry flag will be set, indicating a failure.

By checking the state of the carry flag after calling interrupt 13h, we can determine whether the disk service was successfully executed or not. This allows us to handle any potential errors or unsupported services appropriately during the boot process.

```
jc NotSupport
```

After calling interrupt 13h, we have a label called "NotSupport" that is placed at the address of the halt instruction. If the disk service is not supported by the BIOS, the carry flag will be set, indicating a failure. In this case, the program flow will jump to the "NotSupport" label.

At this label, we compare the value in the bx register with the hexadecimal value aa55. If the comparison fails, it means that the disk extension support is not available. This check helps us verify if the disk extension services are supported by the system.

If the disk extension support is not available, appropriate actions can be taken, such as displaying an error message or taking alternative measures to handle the situation.

```
jne NotSupport //this command means jump if not equal
```

If the disk extension support is available and the comparison succeeds, it means that the disk extension services are supported by the system. In this case, we change the message to "Disk extension is supported" and proceed with the desired actions, such as loading the kernel from disk to memory and jumping to the kernel code.

**You can find the final assembly code below:**

```
[BITS 16]
[ORG 0x7c00]

start:
    xor ax,ax
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov sp,0x7c00

TestDiskExtension:
    mov [DriveId],dl
    mov ah,0x41
    mov bx,0x55aa
    int 0x13
    jc NotSupport
    cmp bx,0xaa55
    jne NotSupport

PrintMessage:
    mov ah,0x13
    mov al,1
    mov bx,0xa
    xor dx,dx
    mov bp,Message
    mov cx,MessageLen
    int 0x10

NotSupport:
End:
    hlt
    jmp End

DriveId:    db 0
Message:    db "Disk extension is supported"
MessageLen: equ $-Message

times (0x1be-($-$$)) db 0

    db 80h
    db 0,2,0
    db 0f0h
    db 0ffh,0ffh,0ffh
    dd 1
    dd (20*16*63-1)
```

```
times (16*3) db 0
```

```
db 0x55
```

```
db 0xaa
```