**Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 4: Operational Systems

Elaborated:

st. gr. FAF-213 Afteni Maria, Bajenov Sevastian, Konjevic Alexandra

Verified: Rastislav Călin

Chişinău – 2023

**Topic:** Boot Loader

**Objectives:** Create in assembly language an application that will act as a Boot Loader and will perform the following tasks:

1. It will display a greeting message that includes the author's name and will await keyboard input for the 'source' address on the floppy disk, from where the kernel (or other compiled code intended to be loaded and executed) will be read. The address should be entered in the SIDE, TRACK, SECTOR format and must strictly fall within the range reserved for the author student, as it was in Lab3.

2. It will wait for keyboard input for the 'destination' address in RAM where the data block read from the floppy disk will be loaded. The RAM address should be in the format XXXXh:XXXXh, identical to how it was in Lab3.

3. It will transfer the FLOPPY ==> RAM data and display the error code with which the operation was completed.

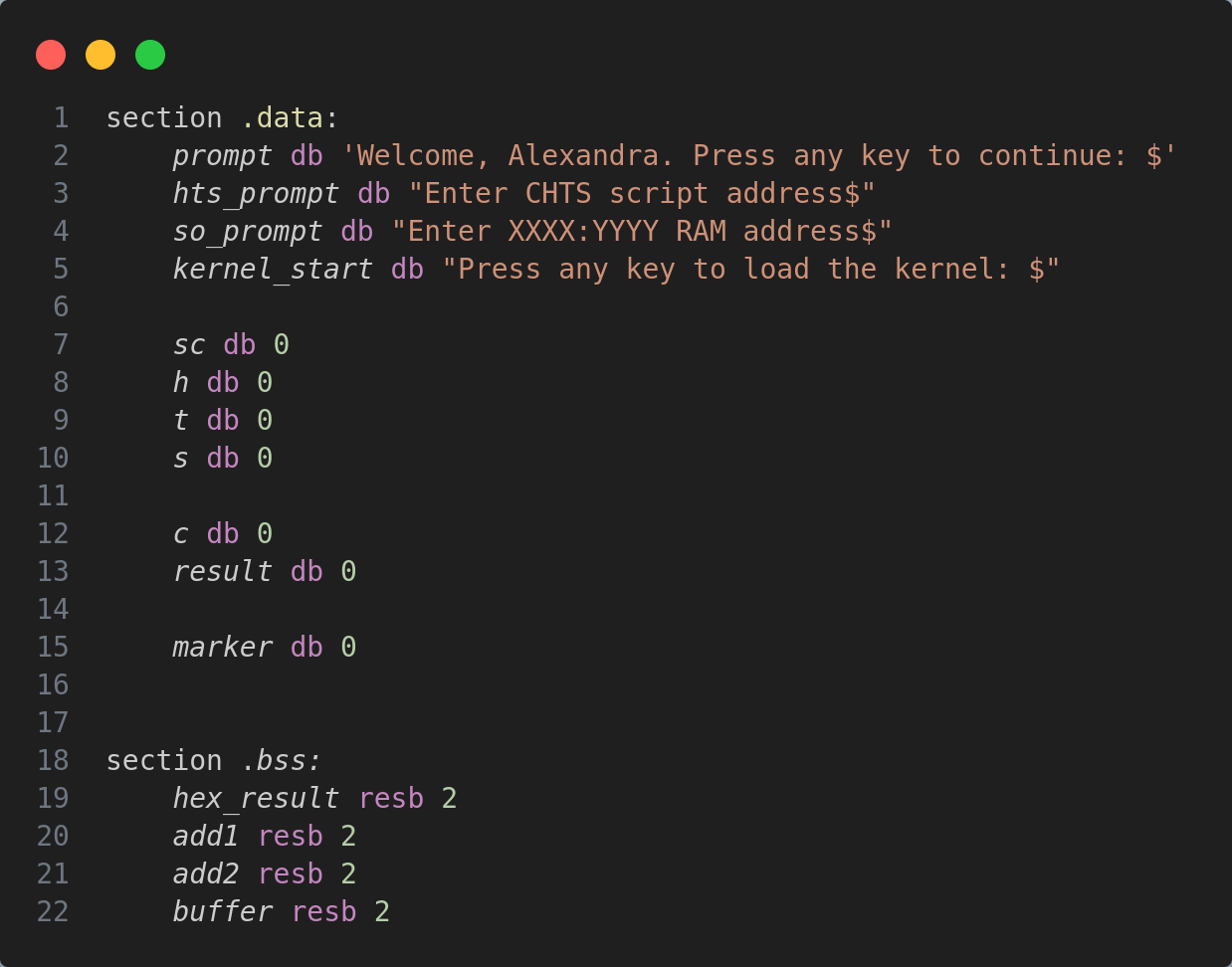
4. It will display a message to press a key to launch the kernel (or execute the desired code).

5. Upon completion of the kernel execution or executed code, it will display a message to press a key to repeatedly execute the Boot Loader.

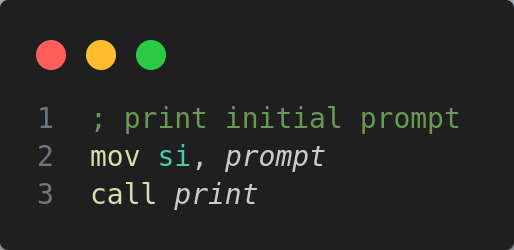
**Implementation:**

To create a boot loader system capable of loading a kernel or other compiled code from the floppy disk into RAM, and then, executing it, there were created two bootloaders: *`bootloader.asm`* and *`mini\_boot.asm`*. The first part (*`bootloader.asm`*) contains the main functionality, while the second part (*`mini\_boot.asm`*) is a small boot loader meant to load the main boot loader.

The code begins with the org *`7d00h`* directive, setting the origin point where the bootloader code will reside in memory.

Various data sections (`*section .data`* and *`section .bss`*) are declared to hold prompts, user inputs, and intermediate variables:

The line that follows after this - *`dw 0AA55h`* - is marking the end of a boot sector, indicating to the BIOS that the sector is bootable and the code contained within it should be executed during the boot process.

The next step is displaying the initial prompt (*`Welcome, Alexandra. Press any key to continue: $`*):

The function *`print`* is being called:

# 

# This function also calls another function: *`cursor`* which sets the position of the cursor using the function *`03h`* of the interruption *`10h`*:

# 

# Next, the printing function declares the *`print\_char`* label, which loads the character from the memory address pointed to by *`si`* into the *`al`* register and compares the character loaded with the ASCII value of *'$'*. This symbol *'$'* marks the end of the prompt. If the character loaded is *'$'*, it jumps to the *`end\_print`* label, which effectively ends the printing process.

# Printing characters occurs in the following mode: the function sets up the video display function (*`0eh`*) of BIOS interrupt *`10h`* for displaying characters. After that, there is incremented the memory address in *`si`* to point to the next character in the string and the function jumps back to the *`print\_char`* label to continue printing the next character.

# After the prompts are displayed, the user input is read, using the function *`00h`* of the interruption *`16h`:*

# And this character is displayed:

# 

# After that, there is printed a new line, by calling the function *`newline`*:

# This function sets the position of the cursor on a new line.

# Next, the next prompt is displayed, to get the CHTS script address (*`Enter CHTS script address$`*):

# The next block of code is responsible for displaying the *`>`* symbol in front of the user input:

# 

# Next, we have the instruction that moves the value 0 into the memory location pointed to by the label *`[result]`*. After that, the functions *`clear`* and *`read\_buffer`* are called:

# The clear function sets the value 0 to the memory locations pointed by *`c`* and *`si`*, and moves the value of the *`buffer`* to the register *`si`*.

# Next, we have the *`read\_buffer`* function:

# First of all, it reads a character from the keyboard (with function *`00h`* of the *`16h`* interruption), after that, handles the enter or backspace keys. If the keys are neither of them, then the introduced character is added to the buffer, and the buffer pointer is incremented. The character read from the keyboard is also displayed on the screen.

# The handling of the enter key uses also additional helper functions:

# This function is responsible for converting a sequence of ASCII characters representing a decimal number into its corresponding numerical value. The instructions *`xor ax, ax`* and *`xor ab, ab`* set the values of this registers to zero. *`atoi\_d`* label marks the beginning of the loop where the actual conversion takes place. `*lodsb`*: This instruction loads the byte addressed by the *`si`* register into the *`al`* register and increments *`si`* automatically, this means it loads the next character from memory into *`al`* on each iteration.

# Conversion steps: *`sub al, '0'`* subtracts the ASCII value of character '0' from the ASCII value of the current character in *`al`*. ASCII values are numeric representations of characters, and subtracting the ASCII value of '0' effectively converts the character to its numerical value. Next, the *`bh`* register is cleared. *`imul bx, 10`* multiplies the value in *`bx`* (which initially holds the previously processed part of the number) by 10. This prepares for the next digit to be added. *`imul`* stands for "integer multiply". It effectively multiplies BX by 10 and stores the result back in *`bx`*. *`add bl, al`* adds the converted numerical value in *`al`* to the *`bl`*. It accumulates the digits to form the final integer value. Next, this value is stored into the memory location labeled as *`result`*. This is the accumulating result of the conversion. *`dec byte [c]`* decrements the byte in memory pointed to by *`c`*. *`cmp byte [c], 0`*: Compares the value pointed to by *`c`* with zero to check if the conversion is complete. If there are more characters to convert (*`[c]`* is not zero), the loop continues (*`jne atoi\_d`*) to process the next character. If all characters have been processed, the subroutine proceeds to *`ret`*, indicating the end of the conversion.

# Also, the handling of enter uses the function:

# 

# The *`atoh`* (ASCII to Hexadecimal) subroutine is responsible for converting a sequence of ASCII characters representing a hexadecimal number into its corresponding numerical value.

# First of all, it clears the value of register *`bx`* and sets the value of register *`di`* to the value of *`hex\_result`*. `*atoh\_s`* label marks the beginning of the loop where the actual conversion takes place.

# *`mov al, [si]`* loads the byte addressed by the *`si`* register into the *`al`* register. This retrieves the next character from memory for processing. After that, if the value from *`al`* is greater than 65, the code jumps to *`atoh\_l`*. If the character is less than or equal to *'9'*, it proceeds directly to the next step. *`jg atoh\_l`* (if the character is greater than 'A', it means the character is in the range 'A' to 'F').

# The function subtracts 55 from the ASCII value (*`al`*), converting characters 'A' to 'F' into their respective hexadecimal values ('A' = 10, 'B' = 11, ..., 'F' = 15). *`sub al, 48`*: If the character is '0' to '9', subtracts 48 from the ASCII value (al) - this converts ASCII characters '0' to '9' into their respective numerical values (0 to 9).

# *`continue`* label moves the current value in *`bx`* into memory location *`[di]`* (hex\_result); multiplies the value in *`bx`* by 16 to shift it left by one hexadecimal digit. After that, it adds the converted value in *`ax`* to *`bx`* and stores the updated value in *`[di]`*. It then increments the *`si`* register to move to the next character and decrements the byte in memory pointed to by *`c`* to track the length of the input string. If there are more characters to convert (*`[ci]`* is not zero), the loop continues (*`jnz atoh\_s`*).

# The handling of the backspace is done this way:

# The *`hdl\_backspace`* segment checks if there's a character in the buffer to be deleted. If there's a character present, it erases the last character in the buffer, moves the cursor back one position on the screen, prints a space character to visually erase the character on the display, and then resumes waiting for new input by jumping back to the read character section. If there are no characters in the buffer, it continues to wait for new input without any erasing action.

# After the value of sector count is read, and the functions *`clear`* and *`read\_buffer`* are called, the same steps are used to read the value of head, track, sector, ram address, segment and offset.

# After the buffer is read, the register *`ax`* is set to the value of *`hex\_result`*, and the value from *`ax`* is moved to the *`[add2]`*.

# After that, the function *`load\_kernel`* is called:

# *`mov ah, 0h`* sets *`ah`* register to 0, indicating the "reset disk system" function for the BIOS interrupt. *`mov ax, [add2]`* moves the value stored in memory at location *`add2`* into the *`ax`* register. The function *`02h`* of the interruption *`13h`* is executed, and the values of the *`sector\_count`, `track`, `sector `, `head`* are read into the memory at *`ES:BX`*. After that, the error code is displayed on the screen.

# After the *`load\_kernel`* function is executed, the next sequence of code displays a prompt asking the user to load a kernel, waits for the user to input a key, displays the input as a character on the screen, and then combines and jumps to the memory address formed by the contents of *`add1`* and *`add2`*.

*`mini\_boot.asm`*:

# 

# *`mini\_boot.asm`* is a smaller boot loader that loads the main boot loader (*`bootloader.asm`*) and jumps to its entry point. The code follows a sequence of steps:

# Display prompts to the user for input (source address, destination address in RAM).

# Read input and handle user interactions (keypresses, backspaces).

# Convert ASCII input to usable numeric values (e.g., converting ASCII input for address to its actual hex value).

# Load the kernel from the floppy disk to the specified RAM address.

# Display messages and prompts at various stages.

# Use BIOS interrupt calls (int 10h, int 13h, etc.) to handle input/output and disk operations.

# Running the code

# In order to create a bootable image for the virtual machine, first of all we need to assemble the program from the file with the individual task , using the command *`nasm task.asm`*. This command creates a binary file*.* After that, we need to execute the script *`boot\_floppy\_script.sh`*, passing it three parameters – the binary file, the bootloader (*`bootloader.asm`*) and the second bootloader (*`mini\_boot.asm`*): *`../boot\_floppy\_script.sh rainbow bootloader.asm mini\_boot.asm`*. The script is the following:

The script takes three arguments from the command line: $1 (binary\_file), $2 (bootloader), and $3 (mini\_bootloader). It checks if the binary file provided as $1 exists. If it doesn’t exist, it displays an error message and exits the script with a non-zero status. It uses NASM with the *`-f bin`* option to assemble the *`mini\_bootloader`* file into *`mini\_bootloader.bin`*. Similarly, it assembles the *`bootloader`* file into *`bootloader.bin`*.

After that, the script creates an empty floppy disk image named *`floppy.img`* with a size of 1.44MB using the truncate command. It then is essentially moving the *`mini\_bootloader.bin`* file and renaming it to the name stored in the variable *`$floppy\_image`* (which is *`floppy.img`* as per the script).

Using dd (data duplicator), it writes the *`bootloader.bin`* content to the created floppy image starting at the second block (512 bytes per block) using *`seek=1`*. This places the bootloader in the boot sector. Similarly, it writes the content of the binary file (*`$binary\_file`*) to the floppy image starting at the fourth block (*`seek=3`*). This presumably adds the program or kernel to the floppy image.

# Conclusion:

# In conclusion, the laboratory work involving the implementation of the provided code has provided a valuable learning experience in low-level programming, bootloader development, and BIOS interrupt handling. It underscores the importance of meticulousness, thorough testing, and a deep understanding of hardware interactions in the realm of systems programming.