**MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA TECHNICAL UNIVERSITY OF MOLDOVA**

**FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

Operating Systems

***Laboratory work 3: Floppy disk input/output operations***

Elaborated:

st.gr. FAF-212 Cristian Brinza

Verified: Rostislav Calin

Chis, ina˘u, 2023

# Task:

Toți studenții de sine stătător vor forma echipe a câte 3 membri din cadrul grupei academice. Componența grupelor va fi aprobată de profesor, inclusiv echipele de 2 sau 4 membri.

Echipele trebuie formate operativ, în limita termenului de timp anunțat de profesor la ora de curs.

Obiectivele lucrării date de laborator sunt focusate pe însușirea abilităților de lucru cu metodele de acces la floppy disk, în special citirea și scrierea datelor, dar nu se limitează doar la acestea. Procedurile date se extind si asupra altor medii de stocare permanentă a datelor precum sunt discurile HDD.

Volumul total la dischetei de 1474560 bytes va fi structural în 96 de blocuri logice a câte 30 de sectoare fiecare. Fiecare student va avea alocat un bloc individual de 15360 bytes, pentru înscrierea datelor sale după cum urmează.

Distribuția blocurilor este reprezentată în fișierul inclus (”Floppy space distribution.xlsx”).

1. În primul și în ultimul sector al blocului fiecărui student (pe dischetă), trebuie înscrisă informația textuală după următorul format (fără ghilimele): ”@@@FAF-21\* Prenume NUME###”.

Acest șir de text trebuie dublat de 10 ori fără caractere adiționale de delimitare.

Exemple:

@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###@@@FAF-212 Vlad URSU###

@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###@@@FAF-211 Andreia-Cristina SIRETANU###

2. Creați un program în limbajul de asamblare care va avea următoarele funcții:

(KEYBOARD ==> FLOPPY) : Citirea de la tastatură a unui string cu lungimea maximală de 256 caracetre (corectarea cu backspace trebuie să funcționeze) și scrierea acestui șir pe dischetă de "N" ori, începând cu adresa {Head, Track, Sector}. Unde "N" poate primi valori în intervalul (1-30000). După detectarea tastei ENTER, dacă lungimea șirului este mai mare ca 0 (zero), trebuie afișată o linie goală și apoi șirul recent introdus. Variabilele "N", "Head", "Track" și "Sector" trebuie citite vizibil de la tastatură. După finisarea operației de scriere pe dischetă, trebuie de afișat la ecran codul de eroare.

**(FLOPPY ==> RAM) : Citirea de pe dischetă a "N" sectoare începând cu adresa {Head, Track, Sector} și transferul acestor date în memoria RAM începând cu adresa {XXXX:YYYY}. După finisarea operației de citire de pe dischetă, trebuie de afișat la ecran codul de eraore. După codul de eroare, trebuie de afișat la ecran tot volumul de date aflat la adresa {XXXX:YYYY} care a fost citit de pe dischetă. Dacă volumul de date afișat este mai mare decât o pagină video atunci este necesar de implementat paginație prin apăsarea tastei ”SPACE”. Variabilele "N", "Head", "Track" și "Sector" precum și adresa {XXXX:YYYY}, la fel trebuie citite de la tastatură.**

(RAM ==> FLOPPY) : Scrierea pe dischetă începând cu adresa {Head, Track, Sector} a unui volum de "Q" bytes, din memoria RAM începând de la adreasa {XXXX:YYYY}. Blocul de date de "Q" bytes trebuie afișat la ecran, iar după finisarea operației de scriere pe dischetă, trebuie de afișat la ecran codul de eroare.

3. După executarea unei funcții de mai sus, programul trebuie să fie gata la executarea următoarei funcții (oricare funcție din cele 3 descrise mai sus).

4. Codul compilat este preferabil să nu depășească 512 bytes. În caz contrar, este necesar de implementat ocolirea acestei restricții și în final trebuie creată imaginea bootabilă a dischetei care să funcționeze în VirtualBox.

5. Fiecare echipă trebuie să perfecteze un raport structurat, inclusiv cu concluzie, similar cu lucrările anterioare. Raportul trebuie să includă codul-sursă complet și funcțional, precum și procedura completă (testată) de compliare și de producere a imaginii bootabile, care să funcționeze corect în VirtualBox. Dacă descrierea procedurii de creare a imaginii bootabile va lipsi din raport sau dacă procedura nu va fi completă sau nu va funcționa (nu va putea fi reprodusă), întreaga echipă riscă să primească nota 5(CINCI)!! Din acest considerent este necesar ca fiecare membru a echipei să testeze procedura. În raport este necesar să specificați inclusiv versiunea compilatorului folosit și a tuturor altor componente sau dependențe care au fost folosite în procesul de construcție (build) a imaginii bootabile.

# Execution:

This assembly code is designed for a boot sector program, typically used in the initial stages of a computer's boot process. The program is written for the x86 architecture and is intended to be loaded at memory address 0x7C00, a standard location for boot sector code. Let's break it down into its components:

**Initialization and Setup:**

[org 0x7C00]: This directive sets the code origin to 0x7C00, indicating where the program will be loaded in memory.

cli: Clears the interrupt flag, disabling interrupts.

mov ax, 0x07C0, mov ds, ax, mov es, ax, mov ss, ax: These instructions set the data, extra, and stack segments to 0x07C0.

mov sp, 0x7C00: Sets the stack pointer to 0x7C00, initializing the stack.

sti: Sets the interrupt flag, enabling interrupts.

mov [DriveNumber], dl: Stores the drive number (from which the boot occurred) into the DriveNumber variable.

Data and Buffers:

The program defines various data segments like String, Buffer, DriveNumber, DataStart, and MenuPrompt. These are used for storing input, output, and messages.

Main Program Loop (start):

The main loop of the program displays a menu, waits for user input, and then jumps to the appropriate task based on the input.

Tasks:

TASK\_1, TASK\_2, and TASK\_3 are labels for different tasks: writing a string to disk, reading from disk, and writing RAM data to disk.

Subroutines:

DISPLAY\_STRING: Displays a null-terminated string.

GET\_CHAR: Waits for and retrieves a single character input.

READ\_STRING: Reads a string from the keyboard into the String buffer.

WRITE\_TO\_DISK\_1, READ\_FROM\_DISK, WRITE\_TO\_DISK\_3: Perform disk I/O operations.

DISPLAY\_DATA: Displays data from the Buffer.

PREPARE\_RAM\_DATA: Prepares data from RAM for writing to disk.

READ\_SECTOR, WRITE\_SECTOR: Low-level disk I/O operations.

DISPLAY\_CHAR: Outputs a single character.

Boot Signature:

dw 0xAA55: This is the boot signature, a required marker at the end of the boot sector.

Error Handling:

The .error label in READ\_SECTOR and WRITE\_SECTOR provides a basic framework for handling disk I/O errors, though it's left incomplete.

Sector Constants:

START\_SECTOR\_1, END\_SECTOR\_1, etc., are constants defining sector ranges for disk operations.

**Code:**

[org 0x7C00]

cli

mov ax, 0x07C0

mov ds, ax

mov es, ax

mov ss, ax

mov sp, 0x7C00

sti

mov [DriveNumber], dl

jmp start

String times 512 db 0

Buffer times 512 db 0

DriveNumber db 0

DataStart db "Your data here", 0

START\_SECTOR\_1 equ 1021

END\_SECTOR\_1 equ 1050

START\_SECTOR\_2 equ 931

END\_SECTOR\_2 equ 960

START\_SECTOR\_3 equ 1681

END\_SECTOR\_3 equ 1710

MenuPrompt db "Choose an option:", 0Dh,0Ah, "1. Write String to Disk", 0Dh,0Ah, "2. Read from Disk", 0Dh,0Ah, "3. Write RAM Data to Disk", 0Dh,0Ah, "Option: ", 0

start:

; Display Menu

mov si, MenuPrompt

call DISPLAY\_STRING

; Wait for user input

call GET\_CHAR

cmp al, '1'

je TASK\_1

cmp al, '2'

je TASK\_2

cmp al, '3'

je TASK\_3

; Invalid input, restart

jmp start

TASK\_1:

CALL READ\_STRING

CALL WRITE\_TO\_DISK\_1

jmp $

TASK\_2:

CALL READ\_FROM\_DISK

CALL DISPLAY\_DATA

jmp $

TASK\_3:

CALL PREPARE\_RAM\_DATA

CALL WRITE\_TO\_DISK\_3

jmp $

; [Insert the subroutine definitions here: READ\_STRING, WRITE\_TO\_DISK\_1, READ\_FROM\_DISK, DISPLAY\_DATA, PREPARE\_RAM\_DATA, WRITE\_TO\_DISK\_3, etc.]

dw 0xAA55 ; Boot signature

; Subroutine Definitions

; Add the definitions for subroutines like READ\_STRING, WRITE\_TO\_DISK\_1, DISPLAY\_STRING, GET\_CHAR, etc.

; For example:

; Display string subroutine

DISPLAY\_STRING:

pusha

mov ah, 0x0E

.repeat:

lodsb

cmp al, 0

je .done

int 0x10

jmp .repeat

.done:

popa

ret

; Get character input subroutine

GET\_CHAR:

push ax

mov ah, 0x00

int 0x16

pop ax

ret

; Read string from keyboard

READ\_STRING:

xor bx, bx ; Reset buffer index

.read\_loop:

mov ah, 0x00 ; BIOS keyboard read

int 0x16 ; Wait for key press

cmp al, 0x0D ; Enter key

je .exit\_read

cmp al, 0x08 ; Backspace

je .backspace

mov [String + bx], al ; Store character

inc bx

jmp .read\_loop

.backspace:

dec bx ; Move back in buffer

jmp .read\_loop

.exit\_read:

mov byte [String + bx], 0 ; Null-terminate string

ret

; Write string to disk

WRITE\_TO\_DISK\_1:

mov ah, 0x03 ; Write sectors

mov al, 1 ; Write one sector at a time

mov ch, 0 ; Cylinder number

mov dh, 0 ; Head number

mov dl, [DriveNumber] ; Drive number

mov cx, START\_SECTOR\_1 ; Start sector

call WRITE\_SECTOR

mov cx, END\_SECTOR\_1 ; End sector

call WRITE\_SECTOR

ret

; Read from disk

READ\_FROM\_DISK:

mov ah, 0x02 ; Read sectors

mov al, 1 ; Read one sector at a time

mov ch, 0 ; Cylinder number

mov dh, 0 ; Head number

mov dl, [DriveNumber] ; Drive number

mov cx, START\_SECTOR\_2 ; Start sector

call READ\_SECTOR

mov cx, END\_SECTOR\_2 ; End sector

call READ\_SECTOR

ret

; Display data

DISPLAY\_DATA:

mov bx, Buffer

mov cx, 512 ; Number of bytes to display

.display\_loop:

mov al, [bx]

call DISPLAY\_CHAR

inc bx

loop .display\_loop

ret

; Prepare RAM Data

PREPARE\_RAM\_DATA:

mov si, DataStart

mov di, Buffer

mov cx, 512

rep movsb

ret

; Write RAM data to disk

WRITE\_TO\_DISK\_3:

mov ah, 0x03

mov al, 1

mov ch, 0

mov dh, 0

mov dl, [DriveNumber]

mov cx, START\_SECTOR\_3

call WRITE\_SECTOR

mov cx, END\_SECTOR\_3

call WRITE\_SECTOR

ret

READ\_SECTOR:

; Assume ES:BX points to the buffer

push ax

push bx

push cx

push dx

mov bx, Buffer ; Data buffer address

int 0x13

jc .error

pop dx

pop cx

pop bx

pop ax

ret

.error:

; Handle disk error

pop dx

pop cx

pop bx

pop ax

ret

WRITE\_SECTOR:

; Assume ES:BX points to the buffer

push ax

push bx

push cx

push dx

mov bx, Buffer ; Data buffer address

int 0x13

jc .error

pop dx

pop cx

pop bx

pop ax

ret

.error:

; Handle disk error

pop dx

pop cx

pop bx

pop ax

ret

DISPLAY\_CHAR:

mov ah, 0x0E ; Teletype output

int 0x10

ret

**Conclusion:**

Through this laboratory work, we have gained a basic understanding of assembly language programming and the process of creating a bootable floppy image. We have successfully created assembly code that is a basic boot sector program for an x86 system, designed to execute right after a computer starts. Upon execution, it displays a menu for the user to choose from three tasks: writing a string to disk, reading from disk, or writing prepared RAM data to disk. The code handles user input, performs the selected disk operations using BIOS interrupts, and includes routines for displaying messages and reading strings from the keyboard.

**References:**

1. "Assembly Language for x86 Processors" by Kip R. Irvine
2. VirtualBox User Manual
3. NASM Documentation