**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 2: ECHO1 (work with keyboard)***

Elaborated:

st.gr. FAF-212 Cristian Brinza

Verified: Rostislav Calin

Chis, ina˘u, 2023

# Task: ECHO1 (work with keyboard)

Create a program in assembler which will "echo" what is typed from the keyboard.

Each ASCII character which will be pressed from the keyboard should appear on the screen and the cursor should move to the next position.

Special actions need to be implemented only for 2 special keys from the keyboard:

"backspace key" - in this case symbol from the left side of the cursor should disappear and the cursor should be moved one position back (If the cursor already is in the first position, than nothing should happen. Special case is if the cursor is on the next line, than when is pressed Backspace in the first column, than cursor should move to the previous line in last column);

"enter key" - in this case all previously introduced string should be printed to the screen starting with the next line and after one "empty" line (but if "enter key" will be pressed as the first key, in this case NO "empty" line should be added and the action should just go to the next line)

(OPTIONAL)

The maximum length of input string should not exceed 256 characters. If the user will want to input more than 256 characters than the input should be stopped and in this case only "backspace" or "enter" keys should be accepted.

> Compiled program should be used in order to create a floppy image and it should be bootable. Use this image to boot the OS in a VirtualBox VM and the text which you intended to print should appear on the screen.

> You can use any assembly compiler.

> Students should be able to modify the code, to recompile it and to boot the VM with new version of program.

> In order to use documentation from TechHelp/XView DOS application, students can install DosBox.

\*\*\* This instructions can update with some new details if it will be required by the teaching process.

# Execution:

1. Register Descriptions: Initial comments describe the purpose of various general-purpose and special-purpose registers like AX, BX, CX, DX, SI, DI, SP, BP, and segment registers. These are essential for data manipulation, addressing, and control in assembly programming.
2. Buffer Setup and Input Reading: This section starts with setting the program's starting address and mode. It defines constants like MAX\_INPUT and BUFFER\_OFFSET, initializes registers, and sets the video mode. It then prepares to read keyboard input and store it in a buffer.
3. Main Input Loop (read\_key): The program enters a loop where it reads keyboard input, checks for special keys like Enter and Backspace, and handles regular character input by saving it to a buffer and echoing it on the screen. It also checks if the buffer has reached its maximum capacity.
4. Handling Enter Key (handle\_enter): When the Enter key is pressed, it checks if any characters were input. If so, it prints the input string and clears the buffer.
5. Buffer Management (clear\_buffer and clear\_input\_buffer): These sections are responsible for clearing the input buffer, either after printing the string or if no input was detected.
6. Backspace Handling (handle\_backspace): This part handles the Backspace key. It erases the last character in the buffer and updates the cursor position on the screen.
7. Full Buffer Handling (stop\_input): If the buffer reaches its maximum size, the program waits for either an Enter or a Backspace key press.
8. Printing Utilities (print\_newline, move\_cursor\_back, print\_string): These are subroutines for printing a newline, moving the cursor back, and printing a string from the buffer.
9. Boot Sector Finalization: The last part of the code ensures the boot sector is exactly 512 bytes with the required boot signature.

;AX, BX, CX, DX: These are the general-purpose registers.

;AX (Accumulator Register): Often used for arithmetic, logic, and data transfer operations.

;BX (Base Register): Typically used for addressing (pointing to data).

;CX (Count Register): Often used as a counter in loops and string operations.

;DX (Data Register): Used alongside AX for certain operations, like multiplication and division.

;SI (Source Index), DI (Destination Index): Used for string and memory array operations. SI generally points to the source, and DI to the destination.

;SP (Stack Pointer), BP (Base Pointer): Used for stack operations. SP points to the top of the stack, while BP is used to reference data in the stack.

;DS (Data Segment), ES (Extra Segment), FS, GS: Segment registers used to access different memory segments.

;Buffer setup and input reading:

;mov si, BUFFER\_OFFSET initializes the SI register to point to the start of the buffer.

;mov [si], byte 0 sets the first byte of the buffer to zero, effectively clearing it.

;In the read\_key section, the code seems to start a process to read keys or characters. The code checks if the buffer is full (reaching 256 characters, indicated by MAX\_INPUT). This is done to prevent buffer overflow, a common issue in low-level programming where data

[org 0x7C00] ; This instruction sets the starting address of the program to 0x7C00.

; It's where the BIOS loads the boot sector of a disk, very typical in older PC architectures.

[bits 16] ; This tells the assembler that we're writing 16-bit code, which is typical for very old PCs

; (like those using Intel 8086 or 80286 processors).

; Define constants

MAX\_INPUT EQU 256 ; EQU is used to define a constant. Here, MAX\_INPUT is set to 256.

BUFFER\_OFFSET EQU 0x8000 ; Another constant, BUFFER\_OFFSET, is defined as 0x8000.

; This will be used as a memory address offset for a buffer.

; Entry point

start:

cli ; "Clear Interrupt Flag": Disables interrupts, preventing the CPU

; from handling interrupts until they're explicitly re-enabled.

xor ax, ax ; AX is a general-purpose register. "xor ax, ax" is a common way to set AX to 0.

; XORing a register with itself clears it.

mov ds, ax ; DS (Data Segment) register is set to 0. Segment registers in 16-bit

; mode are used to address memory locations.

mov es, ax ; ES (Extra Segment) register is also set to 0.

mov fs, ax ; FS register is set to 0.

mov gs, ax ; GS register is set to 0.

mov ss, ax ; SS (Stack Segment) register is set to 0.

mov sp, 0xFFFF ; SP (Stack Pointer) register is set to 0xFFFF.

; It points to the top of the stack in memory.

sti ; "Set Interrupt Flag": Re-enables interrupts.

; Set video mode to 80x25 text mode

mov ah, 0x00

mov al, 0x03

int 0x10 ; Calls the BIOS interrupt 0x10 to set the video mode.

; AH=0 and AL=3 sets it to 80x25 text mode.

mov si, BUFFER\_OFFSET ; SI (Source Index) is a register used for indexed addressing of memory.

; It's set to the buffer's starting point.

mov [si], byte 0 ; Sets the first byte of the buffer to 0, effectively clearing it.

read\_key:

; Check if we've reached 256 characters

mov di, si ; DI (Destination Index) is another register for indexed addressing.

; It's set to the current position in the buffer.

sub di, BUFFER\_OFFSET ; Subtracts the buffer offset from DI to find out how many characters

; have been read into the buffer.

cmp di, MAX\_INPUT ; Compare DI with MAX\_INPUT (256). This checks if the buffer is full.

je stop\_input ; If DI equals MAX\_INPUT, jump to the 'stop\_input' label.

; Get key from keyboard

xor ah, ah ; Clear AH register.

int 0x16 ; Call interrupt 0x16. This waits for a key press and stores the result in AX.

; Check for Enter key (0x1C is the scan code for Enter)

cmp ah, 0x1C ; Compare AH with 0x1C (Enter key's scan code).

je handle\_enter ; If Enter key is pressed, jump to 'handle\_enter'.

; Check for Backspace key (0x0E is the scan code for Backspace)

cmp ah, 0x0E ; Compare AH with 0x0E (Backspace key's scan code).

je handle\_backspace ; If Backspace key is pressed, jump to 'handle\_backspace'.

; Save the key to the buffer and echo it

mov [si], al ; Move the key from AL to the buffer location pointed by SI.

inc si ; Increment SI to point to the next position in the buffer.

mov ah, 0x0E

int 0x10 ; Call interrupt 0x10 to display the character on screen.

jmp read\_key ; Jump back to 'read\_key' to read the next key.

handle\_enter:

mov di, si

sub di, BUFFER\_OFFSET

test di, di ; Test if DI is 0 (no characters input).

jz clear\_buffer ; If DI is 0, jump to 'clear\_buffer'.

; Move to a new line and print the input string

call print\_newline

call print\_newline ; Print an extra empty line for spacing.

mov di, BUFFER\_OFFSET ; Reset DI to the start of the buffer.

call print\_string ; Call a subroutine to print the string from the buffer.

;The program checks if the buffer is full. If it is, it stops accepting input.

;It waits for a key press and checks if it's the Enter or Backspace key.

;If a regular key is pressed, it's saved to the buffer and echoed to the screen.

;Upon pressing Enter, it tests if any characters were input. If not, it clears the buffer; otherwise, it prints the input string.

clear\_buffer:

; Clear the buffer after printing or if it's empty

call clear\_input\_buffer ; Calls a subroutine to clear the input buffer.

; Print newlines for spacing after clearing buffer or printing string

call print\_newline ; Calls a subroutine to print a newline.

call print\_newline ; Calls it again for an extra newline for spacing.

jmp read\_key ; Jump back to 'read\_key' to read the next key.

clear\_input\_buffer:

; Subroutine to clear the input buffer

mov di, BUFFER\_OFFSET ; DI points to the start of the buffer.

mov cx, MAX\_INPUT ; CX is set to the number of bytes to clear (256).

mov al, 0 ; AL is set to 0, the value to fill the buffer with.

rep stosb ; Repeatedly store the value in AL to the buffer, clearing it.

mov si, BUFFER\_OFFSET ; Reset SI to the start of the buffer.

ret ; Return from the subroutine.

handle\_backspace:

; Check if we're at the start position

cmp si, BUFFER\_OFFSET

je read\_key ; If SI is at the buffer's start, do nothing and read the next key.

dec si ; Decrement SI to move back the buffer pointer.

mov byte [si], 0 ; "Erase" the character in the buffer by setting it to zero.

call move\_cursor\_back ; Call a subroutine to move the cursor back and erase the character on the screen.

jmp read\_key ; This instruction causes the program to jump back to the 'read\_key' label.

; It's part of a loop to read the next key from the keyboard.

;Clearing the Buffer: If the Enter key is pressed or the buffer is full, the program clears the buffer and prints new lines for spacing.

;The clear\_input\_buffer Subroutine: This is a routine to clear the buffer. It uses the rep stosb instruction, which stores a byte (in this case, 0) into a string of memory locations, effectively clearing the buffer.

H;andling Backspace: When the Backspace key is pressed, the program checks if the cursor is at the start of the buffer. If not, it moves the cursor back, erases the character from the buffer, and calls a subroutine to handle the cursor movement on the screen.

;Prevent Overflow: The buffer has a fixed size (256 bytes in your code). Clearing the buffer after processing the input ensures that it doesn't overflow. An overflow could occur if the program keeps accepting input without ever clearing the buffer. Buffer overflow can lead to various issues, including crashing the program or, in worse cases, security vulnerabilities.

;Reset for New Input: After the input is processed (for example, after it's displayed on the screen), the buffer is cleared to make room for new input. This is similar to resetting a form after submission in modern applications. It ensures that subsequent inputs start fresh and do not contain remnants of previous data.

;Maintain Accuracy: Not clearing the buffer could result in the previous input affecting the new input. For instance, if the buffer isn't cleared and the user enters fewer characters than the last time, the end of the new input would still contain parts of the old input.

;Consistency and Predictability: Clearing the buffer ensures that the program behaves consistently each time it processes input. This predictability is crucial in programming, especially at the low level where there's direct interaction with hardware.

; Move cursor back and erase character on screen

stop\_input:

; In case of 256 character limit, wait for Enter or Backspace

xor ah, ah ; Clears the AH register. 'xor' is a bitwise operation; when used with

; the same register (ah, ah), it sets that register to 0.

int 0x16 ; Calls interrupt 0x16. In assembly, an 'interrupt' is a way to pause

; the current program and call a special function (in this case, to wait

; for a key press).

cmp ah, 0x1C ; Compares the AH register with 0x1C. 'cmp' is used for comparison.

; 0x1C is the scan code for the Enter key.

je handle\_enter ; 'je' means 'jump if equal'. If AH is equal to 0x1C (Enter key),

; it jumps to the 'handle\_enter' label.

cmp ah, 0x0E ; This is another comparison, this time checking for the Backspace key.

je handle\_backspace ; If AH is equal to 0x0E (Backspace key), jump to 'handle\_backspace'.

jmp stop\_input ; If neither Enter nor Backspace is pressed, it jumps back to

; 'stop\_input' to repeat the process.

print\_newline:

; Subroutine to print a newline

mov ah, 0x0E ; Prepares to call a video interrupt.

mov al, 0x0A ; 0x0A is the ASCII code for Line Feed (LF), moves the cursor down.

int 0x10 ; Calls interrupt 0x10, which interacts with the display/screen.

mov al, 0x0D ; 0x0D is the ASCII code for Carriage Return (CR), moves the cursor to the start.

int 0x10 ; Calls interrupt 0x10 again to print the Carriage Return.

ret ; 'ret' is return. It returns control to where the subroutine was called.

move\_cursor\_back:

; Get the current cursor position

mov ah, 0x03 ; Function 0x03: Read cursor position

mov bh, 0 ; Page number

int 0x10 ; BIOS video interrupt

; AH now contains the cursor row (in AL), and DL contains the cursor column

; Check if we are at the start of a line

cmp dl, 0 ; Compare the column with 0

je check\_row ; If at start, check the row

move\_back\_one:

; Existing logic to move back one position

mov ah, 0x0E

mov al, 0x08 ; 0x08 is the ASCII code for Backspace.

int 0x10 ; This moves the cursor back one position.

mov al, ' ' ; ' ' is the ASCII code for space.

int 0x10 ; This prints a space, 'erasing' the character on the screen.

mov al, 0x08 ; Backspace again to reposition the cursor.

int 0x10

ret

check\_row:

; Check if current row is greater than 0

test al, al ; Test if AL (row number) is 0

jz move\_back\_one ; If row is 0, just move back one position

; Logic to move cursor to the end of the previous line

; Decrease row

dec al ; Decrease the row number

; Set column to the end of line (assuming 80 characters per line)

mov dl, 79 ; Set the column to the last position

; Correctly set the row and column for the BIOS call

push ax ; Save AX

mov ah, al ; Move the row number to AH

mov al, dl ; Move the column number to AL

mov bh, 0 ; Page number

mov ah, 0x02 ; Function 0x02: Set cursor position

int 0x10 ; BIOS video interrupt

pop ax ; Restore AX

ret

print\_string:

;Handling Full Buffer: If the buffer is full (256 characters), the program waits for either an Enter or a Backspace key press. Other keys are ignored.

;Printing New Line: The print\_newline subroutine prints a new line by issuing a Line Feed followed by a Carriage Return. These are control characters used to move the cursor to the start of a new line.

;Moving Cursor Back: The move\_cursor\_back subroutine simulates a backspace action. It moves the cursor back, overwrites the last character with a space, and then moves the cursor back again.

;Printing a String: The print\_string subroutine is set up to print a string from the buffer. However, the full implementation of this loop is not shown in the snippet.

; Subroutine to print a string

.print\_char: ; This is a label used as a part of a loop in the subroutine.

mov al, [di] ; This loads the byte at the memory location pointed to by DI into AL.

; In this context, it's loading a character from the buffer into AL.

or al, al ; The 'or' instruction here is used to set the Zero Flag (ZF) if AL is 0.

; It's a common way to check if a value is zero in assembly language.

jz .done ; 'jz' means 'jump if zero'. If AL is zero (end of the string), it jumps

; to the label '.done'.

mov ah, 0x0E ; Prepares AH with the function number for the video interrupt to print a character.

int 0x10 ; Calls interrupt 0x10, which prints the character in AL to the screen.

inc di ; Increments DI, moving to the next character in the buffer.

jmp .print\_char ; Jumps back to '.print\_char' to process the next character.

.done:

ret ; Returns from the subroutine once the end of the string is reached.

; Boot signature

times 510-($-$$) db 0 ; This fills the rest of the boot sector with zeros.

; The 'times' directive repeats an instruction a specified number of times.

; Here, it ensures the boot sector is exactly 512 bytes.

dw 0xAA55 ; This is the boot signature, 0xAA55. It's required at the end of the boot sector

; to be recognized as a valid bootable disk by the BIOS.

;Printing a String: The print\_string subroutine prints each character of a string stored in a buffer. It uses a loop to go through each character until it encounters a zero byte, which signifies the end of the string.

;Boot Sector Padding: The times 510-($-$$) db 0 instruction is used to fill the remainder of the boot sector with zeros. The boot sector must be exactly 512 bytes, and this ensures that requirement is met.

;Boot Signature: The dw 0xAA55 instruction places the boot sector signature at the end. This signature is necessary for the BIOS to recognize the sector as a bootable disk.

Key components:

* **BITS 16**: This tells the assembler that we're working in 16-bit mode, which is the mode used by the BIOS when the computer starts.
* **ORG 0x7C00**: The BIOS loads the bootloader at memory address **0x7C00**. This directive tells the assembler to start the program at this address.
* **cli**: This instruction disables interrupts. Interrupts are signals that can pause the CPU to handle external events. By disabling them, we ensure our code runs without interruption.
* **xor, mov**: These are data movement and manipulation instructions. They set up the initial environment for our bootloader.
* **call**: This instruction calls a subroutine. In this case, it's calling the **display\_methods** subroutine.
* **hlt**: This instruction halts the CPU. It's used here to stop the CPU after our bootloader has finished running.
* **jmp hang**: This creates an infinite loop, ensuring the CPU doesn't try to execute random data after the bootloader.
* **times 510-($-$$) db 0**: This is a NASM directive. It fills the bootloader with zeros until it's 510 bytes long. This is because the bootloader must be exactly 512 bytes, with the last two bytes being the signature **0xAA55**.
* **dw 0xAA55**: This is the bootloader signature. The BIOS looks for this signature to identify a bootable disk.

**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 a simple bootloader that displays text and read text on the screen and tested it in a virtual environment using VirtualBox.

**References:**

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