Printing User Input in Reverse

Jerry, Caroline, Fernando

CODE DEMO

What is Assembly?

Assembly is a low-level programming language that communicates directly with the computer's hardware. Unlike Python or Java, it's super close to what the CPU actually understands, that's why it's fast and powerful, but also harder to read and write.

Each instruction in Assembly corresponds closely to a machine code instruction. It's like speaking the computer's own language.

Registers Overview

The CPU has small storage units called *registers*. These are like tiny ultra-fast variables. Common ones include:

- `eax`, `ebx`, `ecx`, `edx`: general-purpose registers.
- `esi`, `edi`: often used for indexing.
- `esp`, `ebp`: used for stack operations.

Memory Addressing

Assembly uses memory addresses to read/write data. Think of memory like a long street with houses, each has a number (address). You can move data to and from these addresses using registers.

System Calls & Int 0x80

To interact with the operating system, like reading input or printing output, we use *system calls*.
In 32-bit Linux Assembly, we trigger these with `int 0x80`, which interrupts the OS and requests a service like:

- `sys_read` to get input
- `sys_write` to print
- `sys_exit` to quit

We tell the OS *which* call we want by putting a number in `eax`, and we use other registers (`ebx`, `ecx`, `edx`) to pass arguments.

Sections: .bss, .text

Assembly programs are divided into sections:

- `.bss`: uninitialized variables (like buffers)
- `.data`: initialized constants (we don't use this one here)
- `.text`: where the code lives: the instructions that actually run

Common instructions: mov, cmp, jmp, etc.

Some key instructions we use:

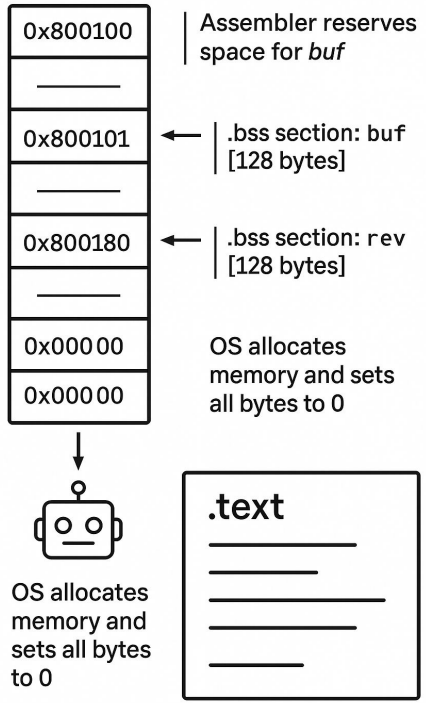
- `mov`: moves data between registers/memory.
- `cmp`: compares two values.
- `jmp`: jumps to another part of code.
- `int`: triggers a system interrupt (like `int 0x80`).
- `inc`, `dec`: increment/decrement values.

These are the building blocks of Assembly logic.

.bss and .text sections of the code

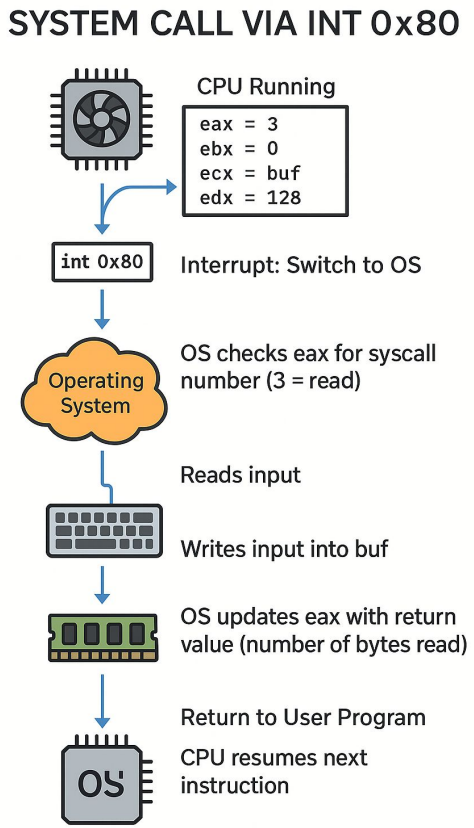
```
section .bss
   buf     resb 128
   rev     resb 128


section .text
   global _start
```

- `.bss` is where we reserve space for variables (`buf`, `rev`).
- `.text` holds our actual instructions.



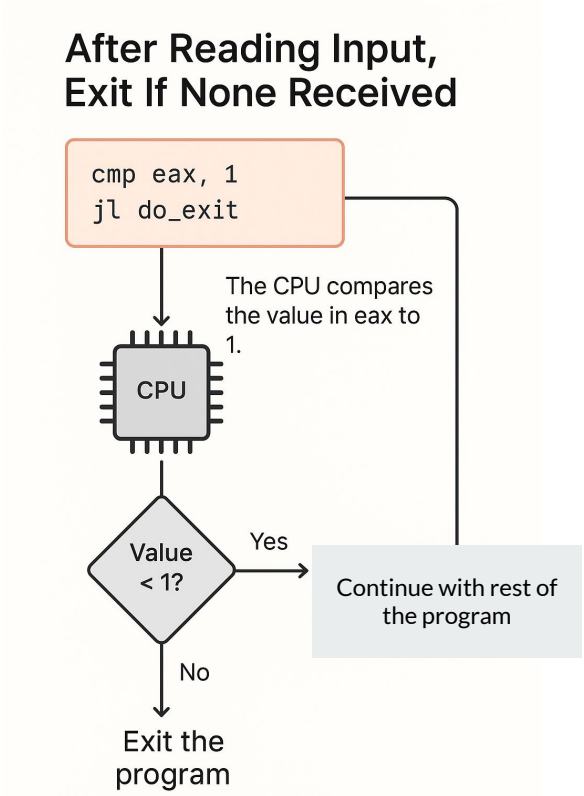
Starting program and calling sys_read

```
_start:
 mov    eax, 3          ; System call 3: sys_read
 mov    ebx, 0          ; File descriptor 0: stdin
 mov    ecx, buf        ; Pointer to input buffer
 mov    edx, 128        ; Read up to 128 bytes
 int    0x80            ; Trigger interrupt
```



Early exit if no input

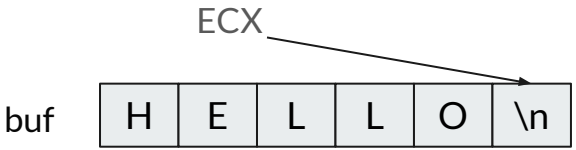
```
; Exit if len < 1
    cmp     eax, 1      ; CoMPare EAX with 1
    jl      do_exit ; Jump to do_exit if Less
```



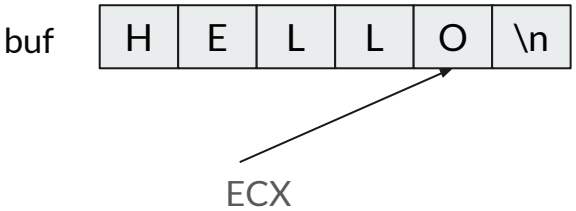
Find the last Character in input

```
mov     ebx, eax ; Save len to EBX
mov     ecx, buf ; Save ptr to ECX
```


```
add     ecx, ebx ; Goto end of ptr
```




```
dec     ecx  ; Point to last char
```



Skipping Newline by Updating String Length (ebx)

```
mov     al, [ecx]           ; Put the last char into AL
cmp     al, 0x0A            ; Check if \n
jne     skip_strip          ; Jump to skip_strip if Not Equal \n
dec     ebx                 ; Len -1 if \n
```

Preparing for the reverse Logic

```
skip_strip:
  xor     esi, esi              ; Clear ESI, efficiently using XOR
```

Set ESI to the result of ESI XOR ESI

Reverse Loop


```
skip_strip:
    xor     esi, esi            ; Clear ESI, efficiently using XOR

rev_loop:
    cmp     esi, ebx            ; while(esi < ebx)
    jge     write_out           ; If ESI > or = EBX: break
    ; Copy len from EBX because EBX is the condition
    mov     ecx, ebx
    dec     ecx                 ; Move ptr forward 1 char
    sub     ecx, esi            ; Skip chars that already copied
    ; Save that char, use AL (lower byte for EAX) for 1 byte
    mov     al, [buf + ecx]
    mov     [rev + esi], al     ; Write from left to right
    inc     esi                 ; esi++
    jmp     rev_loop
```

```
// Equal to
ssize_t esi = 0;
while (esi < len) {
    rev[esi] = buf[len - 1 - esi];
    esi++;
}
```

Output

```
write_out:
   mov     eax, 4              ; System call 4: sys_write
   mov     ebx, 1              ; File descriptor 1: stdout
   mov     ecx, rev            ; Pointer to rev
   mov     edx, esi            ; Length of output
   int     0x80
```

Exit

```
do_exit:
   mov      eax, 1              ; System call 1: sys_exit
   xor      ebx, ebx            ; return 0
   int      0x80
```