This expanded cheat sheet provides a comprehensive guide to the essential NASM instructions for OS development. It includes descriptions of what each instruction does, when to use it, how it works, and real-world scenarios. These instructions are crucial for building bootloaders, kernels, and managing low-level system operations.

Data Movement

mov - Move data from one place to another

- Where & When: Used always. This is the most fundamental operation in assembly. You'll use it to load values into registers (e.g., preparing values for arithmetic or comparison) and copy data between registers or memory.
- Why: Without mov, the CPU cannot perform operations or manipulate data. It's essential to control the flow of information.
- How:

```
mov ax, 0x1234  ; Load 0x1234 into the AX register
mov bx, ax      ; Copy the value of AX into BX
```

---

xchg - Exchange two values

- Where & When: Used when you need to swap two values in a quick, efficient way.
- Why: It's more efficient than using mov twice when you want to swap two values in one instruction.
- How:

```
  xchg ax, bx  ; Swap the values in AX and BX
```

---

push / pop - Save to and restore from the stack

- Where & When: Used in function calls to save registers and return to a prior state.
- Why: The stack is used to store return addresses, saved registers, and local variables.
- How:

```
  push ax  ; Save AX to the stack
  pop bx   ; Restore the top value of the stack into BX
```

---

lea - Load Effective Address

- Where & When: Used when you need the address of a variable or data structure, not the data itself.
- Why: It’s often more efficient than using mov with indirect memory access when you only need the address.
- How:

```
  lea si, [msg]  ; Load the memory address of msg into SI
```

Arithmetic & Logic

add / sub - Basic addition and subtraction

- Where & When: Used frequently for basic arithmetic or adjustments, such as incrementing/decrementing counters, calculating addresses, or performing simple arithmetic.
- Why: Every OS needs to manage memory, iterate over data, or handle arithmetic calculations.
- How:

```
add ax, 5  ; AX = AX + 5
sub bx, 3  ; BX = BX - 3
```

---

inc / dec - Increment or Decrement by 1

- Where & When: Perfect for simple counter operations (e.g., looping through an array or incrementing the stack pointer).
- Why: These are one-byte instructions, making them more efficient than add and sub for operations that only involve +1 or -1.
- How:

```
inc cx  ; Increment CX by 1
dec dx  ; Decrement DX by 1
```

---

mul / div - Multiplication and Division

- Where & When: Used when performing math on data. For example, multiplying or dividing values for memory management, handling file systems, or generating addresses.
- Why: These instructions are specialized for handling multiplication and division, which are important for calculating sizes, offsets, and more.
- How:

```
mov al, 5
mov bl, 3
mul bl  ; AX = AL * BL = 15
```

---

and, or, xor, not - Bitwise logical operations

- Where & When: Use bitwise logic for flags, masking, and manipulating individual bits in memory addresses, data, or register values.
- Why: Operating systems often work with bitflags to track state.
- How:

```
and ax, 0x0F   ; AX = AX AND 0x0F (only keep the lowest 4 bits)
not ax         ; Flip all bits in AX
xor ax, bx     ; AX = AX XOR BX (toggle bits)
```

Comparisons & Flow Control

cmp - Compare two values

- Where & When: Before performing any conditional jump, cmp is used to set the flags based on a subtraction (but doesn’t save the result).
- Why: You can’t make decisions without comparing values first. It's the "setup" instruction for conditional jumps.
- How:

```
cmp ax, bx  ; Set flags based on AX - BX
```

---

je, jne, jg, jl - Conditional jumps

- Where & When: Used after a cmp to perform logic based on the result of the comparison.
- Why: Conditional jumps allow you to make decisions in your code, enabling if-else or looping structures.
- How:

```
cmp ax, bx
je equal   ; Jump to "equal" if AX == BX
jne not_equal ; Jump if AX != BX
```

---

jmp - Unconditional jump

- Where & When: Use when you need to jump to another part of your code, no matter what. It’s often used in loops or to break out of sections of code.
- Why: It's an absolute necessity for looping, branching, and jumps within the code.
- How:

```
jmp start_loop  ; Jump unconditionally to "start_loop"
```

---

call / ret - Function call and return

- Where & When: Use call to invoke a function, and ret to return from a function.
- Why: Function calls and returns form the structure of any program. In an OS, these are the

foundation for interrupt handling, system calls, and other critical operations.

- How:

```
call print_message  ; Call function "print_message"
ret                 ; Return to the calling function
```

System and Interrupt Control

int - Trigger a software interrupt (BIOS or OS)

- Where & When: Use to request services from the BIOS or the OS, like printing to the screen or reading from disk.
- Why: Interrupts provide an interface for low-level hardware and OS interaction.
- How:

```
mov ah, 0x0E    ; Teletype mode for printing
mov al, 'A'     ; Character to print
int 0x10        ; Call BIOS interrupt 0x10 to print 'A'
```

---

iret - Return from an interrupt

- Where & When: Used after handling an interrupt (e.g., during interrupt service routines).
- Why: It tells the CPU to return control to whatever was interrupted.
- How:

```
iret
```

---

cli / sti - Disable/Enable interrupts

- Where & When: Use cli to disable interrupts temporarily (to prevent a time-sensitive operation from being interrupted) and sti to enable interrupts again.
- Why: You disable interrupts when you need to perform a critical section of code without interruption (e.g., switching to protected mode).
- How:

```
cli   ; Disable interrupts
sti   ; Re-enable interrupts
```

---

hlt - Halt the CPU

- Where & When: Use to stop the CPU, typically when the OS is waiting for an interrupt or when it is done executing.
- Why: It's a clean way to pause the CPU until the next interrupt.
- How:

```
hlt   ; Halt the CPU (wait for an interrupt)
```