NASM x86-64 Beginner Guide

A Basic Assembly Language Learning Handout

Prepared by

Jubair Ahammad Akter

Department of Computer Science & Engineering
University of Dhaka

Date: January 4, 2026

Contents

1. Introduction

What is NASM?

NASM (Netwide Assembler) is a popular assembler for x86 and x86-64 architectures. It uses a clean, simple syntax and works well with the GNU toolchain on Linux, macOS, and even **Windows (via MinGW or WSL)**.

In this booklet you will learn:

- How to install NASM on Linux, macOS and Windows.
- How to write, compile and run basic assembly programs.
- The difference between `.data`, `.bss` and `.text`.
- How arrays, memory, and the stack work.
- A compact NASM cheat-sheet for quick revision.

2. Installing NASM

2.1 Ubuntu / Debian

Install NASM on Ubuntu

Just run the commands on terminal

```
sudo apt update
sudo apt install nasm
nasm --version
```

2.2 macOS (Homebrew)

Install NASM on macOS

```
brew install nasm
nasm --version
```

2.3 Windows

Install NASM on Windows

- Download NASM from: https://www.nasm.us
- Install and add NASM to your `PATH`.
- Open `cmd` or PowerShell and type:

```
nasm --version
```

3. Sample Program: Adding Two Numbers

NASM Code: Addition Program

```
extern printf

SECTION .data
fmt: db "Sum = %ld", 10, 0

SECTION .text
global main

main:
    push rbp
    mov rbp, rsp

    mov rax, 10
    mov rbx, 20
    add rax, rbx

    mov rdi, fmt
    mov rsi, rax
    xor rax, 0
    call printf

    mov rsp, rbp
    pop rbp
    xor rax, rax
    ret
```

4. Saving Your Assembly Program

Create the File

Create a new text file and save the code as: `hello.asm`

Linux / macOS:

```
nano hello.asm
```

Paste the code, then press `Ctrl+O`, `Enter`, `Ctrl+X` to save and exit.

Windows (PowerShell):

```
notepad hello.asm
```

Save the file in the same folder where you will run NASM commands.

5. Compile and Run

Compilation Commands

Step 1: Assemble to object file

```
nasm -f elf64 hello.asm -o hello.o
```

Step 2: Link with GCC (use -no-pie)

```
gcc -no-pie hello.o -o hello
```

Step 3: Run the program

```
./hello
```

6. NASM Sections: .data vs .bss

6.1 SECTION .data

Initialized Data Examples

```
SECTION .data

; ---------- Input / Output Format Strings ----------
int_in_fmt:      db "%ld", 0             ; read 64-bit integer
int_out_fmt:     db "%ld", 10, 0         ; print int + newline

str_in_fmt:      db "%s", 0              ; read string
str_out_fmt:     db "%s", 10, 0          ; print string + newline

char_in_fmt:     db "%c", 0              ; read one character
char_out_fmt:    db "%c", 10, 0          ; print char + newline

float_in_fmt:    db "%lf", 0             ; read double
float_out_fmt:   db "%lf", 10, 0         ; print double full precision

float_out2_fmt:  db "%.2lf", 10, 0       ; print double with 2 decimal
                                         ; places
                                         ; example: 3.14159 -> 3.14

newline:         db 10, 0                ; '\n'

; ---------- Single Variables ----------
ch:     db 'A'                           ; 1-byte char
str:    db "Hello",0                     ; NULL-terminated string
num:    dq 12345                         ; int64
flt:    dq 3.14                          ; stored as double for
                                         ; scanf/printf
dbl:    dq 3.1415926535                  ; double precision

; ---------- Arrays ----------
int_arr:     dq 10,20,30,40,50           ; int64 array
char_arr:    db "ABCDE",0                ; char array = string
float_arr:   dq 1.2,2.3,3.4              ; double array for %lf
str_arr:     dq str, newline             ; pointers to strings

; ---------- String Constants ----------
name_msg:    db "Jubair can't teach properly",10,0  ; print with
↪  newline
```

Directive Sizes & Notes

Data Type Sizes:

Directive	Size / Usage	Meaning / Full Form
db	1 byte (byte / char)	Define Byte
dw	2 bytes (word)	Define Word
dd	4 bytes (float / int32)	Define Doubleword
dq	8 bytes (double / int64)	Define Quadword
dt	10 bytes (80-bit float)	Define Ten bytes (Extended Float)
do	16 bytes (128-bit)	Define Octaword
dy	32 bytes (256-bit)	Define Yword (AVX 256-bit)
dz	64 bytes (512-bit AVX-512)	Define Zword (AVX-512)

Where Used?

- **db** → characters, small integers (0-255), strings
- **dw/dd/dq** → normal integer and floating data
- **dt/do/dy/dz** → special scientific / SIMD / advanced CPU types

Where to use db / dq / etc? — With Reasons

- Use **db** when working with:
  - ASCII characters (1 byte each)
  - Strings (`db "Hello",0`)
  - Boolean flags & very small integers (0–255)
- Use **dd** for:
  - 32-bit integers (`int`)
  - Single-precision floats (`float`)
- Use **dq** when:
  - Working with 64-bit integers (int64 / long)
  - Storing **double** values (C `double`)
  - Pointers & addresses (64-bit)
  - scanf/printf formats: `%ld`, `%lf`

  (Most common in x86-64!)

6.2 SECTION .bss

Uninitialized Data Examples

```
; ---------- Single Variables (uninitialized) ----------
x:              resq 1      ; reserve 1 QWORD (8 bytes) for int64
ch2:            resb 1      ; reserve 1 byte for a character
flt2:           resd 1      ; reserve 1 DWORD (4 bytes) for a float
dbl2:           resq 1      ; reserve 1 QWORD for a double

; ---------- Arrays: Allocate space only ----------
int_arr1:       resq 1      ; array of 1 int64 → 8 bytes
char_arr1:      resb 1      ; array of 1 char  → 1 byte
str_arr1:       resb 20     ; buffer for a small string (20 chars max)
flt_arr1:       resd 1      ; array of 1 float → 4 bytes

; ---------- Arrays: Large storage ----------
int_arr100:     resq 100    ; 100 × 8 bytes = 800 bytes (int64 array)
char_arr100:    resb 100    ; 100 chars for string input buffer
str_arr100:     resb 200    ; 200 chars for storing multiple strings
flt_arr100:     resd 100    ; 100 floats (4×100 = 400 bytes)

; ---------- Big Input Buffer ----------
buffer:         resb 1024   ; 1 KB (general input/output buffer)
```

Reserve Sizes & Notes

Reserve Directives in `SECTION .bss`:

Directive	Unit Size	Meaning / Full Form
resb	1 byte	Reserve Byte
resw	2 bytes	Reserve Word
resd	4 bytes	Reserve Doubleword
resq	8 bytes	Reserve Quadword
rest	10 bytes	Reserve Ten bytes (80-bit float)

Where Used?

- Large arrays and buffers → saves program file size
- Variables that will be initialized later during runtime
- Dynamic input storage (e.g., strings, sensor values)

Reserve Sizes & Notes

Important Notes:

- Values are **not stored in the executable**
- OS auto-initializes **all bytes to 0** before program starts
- Use **resq** for pointers & int64 (64-bit architecture)
- Array size = `count × unit size`

Where NOT to use? (with reason)

- Do NOT put initialized data here → '.bss' = **uninitialized only**
- Do NOT store constants here → constants must exist in binary → '.data'

7. Registers Overview

64-bit General Purpose Registers

- **RAX** – Accumulator / function return value
- **RBX** – General purpose (callee-saved)
- **RCX** – 4th function argument / loop counter
- **RDX** – 3rd function argument / data register
- **RSI** – 2nd function argument
- **RDI** – 1st function argument
- **RBP** – Base pointer (callee-saved, stack frame reference)
- **RSP** – Stack pointer (top of stack)
- **R8** – 5th function argument
- **R9** – 6th function argument
- **R10 - R15** – General purpose (callee-saved)

System V Argument Passing Order: `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`
Extra arguments → passed on the **stack**.

8. Print from Initialized Data

This section demonstrates how to print different types of initialized data from the `.data` segment using `printf` in NASM x86-64.

8.1 Print Integer (%ld)

Printing 64-bit Integer

```
SECTION .data
num:    dq 12345
fmt_i:  db "Integer = %ld",10,0

SECTION .text
global main
extern printf

main:
    mov rdi, fmt_i
    mov rsi, [num]
    xor rax, rax
    call printf
    ret
```

8.2 Print Double (%lf), also (%.2lf)

Printing Double Precision Floating Point also for 2 Decimal Places

```
SECTION .data
val:    dq 3.1415926535
fmt_f:  db "Float = %lf",10,0
fmt_f2: db "Pi = %.2lf",10,0

SECTION .text
global main
extern printf

main:
    mov rdi, fmt_f     ; if you use \%.2lf then you will get 3.14
    mov rsi, [val]
    xor rax, rax
    call printf
    ret
```

8.3 Print Character (%c)

Printing Single Character

```
SECTION .data
ch:     db 'A'
fmt_c:  db "Character = %c",10,0

SECTION .text
global main
extern printf

main:
    movzx rsi, byte [ch]     ; widen char → 64-bit register
    mov rdi, fmt_c
    xor rax, rax
    call printf
    ret
```

8.4 Print String (%s)

Printing Null-Terminated String

```
SECTION .data
msg:    db "Hello NASM!",0
fmt_s:  db "String = %s",10,0

SECTION .text
global main
extern printf

main:
    mov rdi, fmt_s
    mov rsi, msg
    xor rax, rax
    call printf
    ret
```

8.5 Print Array of Integers

Printing Integer Array Using Loop also Array Addressing

```
SECTION .data
arr: dq 10, 20, 30, 40, 50      ; 5 elements
n:   dq 5                       ; equ = equate → Assign a symbolic
↪  constant a value
fmt_arr: db "arr[%ld] = %ld",10,0

SECTION .text
global main
extern printf

main:
    xor rbx, rbx                ; i = 0

print_loop:
    cmp rbx,[n]                  ; i < n ?
    jge done_print

    mov rdi, fmt_arr            ; "arr[i] = value"
    mov rsi, rbx                ; index
    mov rdx, [arr + rbx*8]      ; load arr[i]
    xor rax, rax
    call printf

    inc rbx
    jmp print_loop

done_print:
    ret
```

9. Arithmetic & Logical Operations

This section demonstrates the basic arithmetic and bitwise operations supported by x86-64 NASM with example outputs.

9.1 Addition and Subtraction

Use of add and sub

```
SECTION .data
a: dq 50
b: dq 22
fmt: db "Result = %ld",10,0

SECTION .text
global main
extern printf

main:
    mov rax, [a]
    add rax, [b]        ; rax = a + b  → 72

    mov rdi, fmt
    mov rsi, rax
    xor rax, rax
    call printf

    mov rax, [a]
    sub rax, [b]        ; rax = a - b → 28

    mov rdi, fmt
    mov rsi, rax
    xor rax, rax
    call printf

    ret
```

9.2 MUL: Unsigned Multiplication

Unsigned Multiply: mul

```
SECTION .data
m: dq 6
n: dq 7
fmt: db "Result = %ld",10,0

SECTION .text
global main
extern printf

main:
    mov rax, [m]        ; RAX = m
    mov rcx, [n]
    mul rcx             ; RAX = RAX * n  (unsigned)

    mov rdi, fmt
    mov rsi, rax
    xor rax, rax
    call printf
    ret
```

9.3 IMUL: Signed Multiplication

Signed Multiply: imul

```
SECTION .data
p: dq -8
q: dq 5
fmt: db "Result = %ld",10,0

SECTION .text
global main
extern printf

main:
    mov rax, [p]
    imul qword [q]      ; RAX = p * q → -40

    mov rdi, fmt
    mov rsi, rax
    xor rax, rax
    call printf
    ret
```

9.4 DIV & Remainder

Signed Divide: idiv (with remainder)

```
SECTION .data
u: dq 50
v: dq 7
fmt: db "Quotient = %ld, Remainder = %ld",10,0

SECTION .text
global main
extern printf

main:
    xor rdx, rdx            ; Clear RDX before divide
    mov rax, [u]            ; Dividend → RDX:RAX

    idiv qword [v]          ; Signed divide:
                            ; RAX = quotient (7)
                            ; RDX = remainder (1)
                            ; (We can also do: mov rcx,[v], idiv rcx)

    mov rdi, fmt
    mov rsi, rax            ; quotient
    mov rdx, rdx            ; remainder already in RDX
    xor rax, rax
    call printf
    ret
```

Why `idiv` instead of `div`?

- **idiv** = signed division Works with negative numbers correctly
- **div** = unsigned division Only for positive integers
- **RDX:RAX** together hold the 128-bit dividend
- **RAX** receives the quotient
- **RDX** receives the remainder

9.5 Bitwise Logical Operations

Use of AND, OR and XOR

```
SECTION .data
r: dq 0b1101                ; 13 in decimal
s: dq 0b1011                ; 11 in decimal

fmt_and: db "AND  = %ld",10,0
fmt_or:  db "OR   = %ld",10,0
fmt_xor: db "XOR  = %ld",10,0

SECTION .text
global main
extern printf

main:
    ; ----- AND -----
    mov rax, [r]
    and rax, [s]            ; 1101 & 1011 → 1001 (9)

    mov rdi, fmt_and
    mov rsi, rax
    xor rax, rax
    call printf

    ; ----- OR -----
    mov rax, [r]
    or rax, [s]             ; 1101 | 1011 → 1111 (15)

    mov rdi, fmt_or
    mov rsi, rax
    xor rax, rax
    call printf

    ; ----- XOR -----
    mov rax, [r]
    xor rax, [s]            ; 1101 ^ 1011 → 0110 (6)

    mov rdi, fmt_xor
    mov rsi, rax
    xor rax, rax
    call printf

    ret
```

10. Loop Example

10.1 Sum of First 10 Numbers (1–10)

Loop using `inc` and `cmp`

```
SECTION .data
fmt: db "Sum = %ld",10,0

SECTION .text
global main
extern printf

main:
    xor rax, rax     ; sum = 0
    mov rbx, 1       ; i = 1

loop_start:
    add rax, rbx
    inc rbx
    cmp rbx, 11
    jle loop_start   ; continue until i <= 10

    mov rdi, fmt
    mov rsi, rax
    xor rax, rax
    call printf
    ret
```

11. Conditional Execution Using cmp and Jumps

11.1 IF-ELSE Example Code

IF-ELSE using cmp and je/jne

```
SECTION .data
num: dq 10
msg_pos: db "Number is Positive",10,0
msg_zero: db "Number is Zero",10,0
msg_neg: db "Number is Negative",10,0

SECTION .text
global main
extern printf

main:
    mov rax, [num]
    cmp rax, 0
    je is_zero
    jl is_negative

is_positive:
    mov rdi, msg_pos
    xor rax, rax
    call printf
    jmp end

is_zero:
    mov rdi, msg_zero
    xor rax, rax
    call printf
    jmp end

is_negative:
    mov rdi, msg_neg
    xor rax, rax
    call printf

end:
    ret
```

11.2 Jump Instructions Overview

Signed vs Unsigned Conditional Jumps

Signed Comparisons:

- `jg` — Jump if greater
- `jge` — Jump if greater or equal
- `jl` — Jump if less
- `jle` — Jump if less or equal

Unsigned Comparisons:

- `ja` — Above (greater)
- `jae` — Above or equal
- `jb` — Below (less)
- `jbe` — Below or equal

Equality:

- `je` — equal (Zero Flag = 1)
- `jne` — not equal (Zero Flag = 0)

12. Input Using scanf

12.1 Integer, Character & String Input (NASM + C Library)

[Single Program: Read Int, Char and String]

```
; Read: integer, character, string
; Then print them back using printf
;
; Build (Linux x86_64):
;   nasm -f elf64 scanf_demo.asm -o scanf_demo.o
;   gcc scanf_demo.o -no-pie -o scanf_demo
;   ./scanf_demo

SECTION .data
prompt_i: db "Enter int: ",0
prompt_c: db "Enter char: ",0
prompt_s: db "Enter string: ",0

fmt_i:    db "You entered int: %ld",10,0
fmt_c:    db "You entered char: %c",10,0
fmt_s:    db "You entered string: %s",10,0

in_i:     db "%ld",0
in_c:     db " %c",0        ; leading space skips newline/whitespace
in_s:     db "%49s",0       ; safe: max 49 chars + null

SECTION .bss
num: resq 1
ch:  resb 1
str: resb 50

SECTION .text
global main
extern printf, scanf

main:
    ; ---- prompt + read integer ----
    mov rdi, prompt_i
    xor rax, rax
    call printf

    mov rdi, in_i
    lea rsi, [num]
    xor rax, rax
    call scanf

    ; ---- prompt + read character ----
    mov rdi, prompt_c
```

```
    xor rax, rax
    call printf

    mov rdi, in_c
    lea rsi, [ch]
    xor rax, rax
    call scanf

    ; ---- prompt + read string ----
    mov rdi, prompt_s
    xor rax, rax
    call printf

    mov rdi, in_s
    lea rsi, [str]
    xor rax, rax
    call scanf

    ; ---- print results ----
    mov rdi, fmt_i
    mov rsi, [num]
    xor rax, rax
    call printf

    mov rdi, fmt_c
    movzx rsi, byte [ch]
    xor rax, rax
    call printf

    mov rdi, fmt_s
    lea rsi, [str]
    xor rax, rax
    call printf

    xor eax, eax
    ret
```

13. Array Input in .bss and Usage of mov vs lea

13.1 Scanning an Integer Array & Printing It

[Store Array in .bss from Keyboard (5 numbers)]

```
; Read 5 integers into an array (arr) stored in .bss
; Then print them with index.
;
; Build:
;   nasm -f elf64 array_scan.asm -o array_scan.o
;   gcc array_scan.o -no-pie -o array_scan
;   ./array_scan

SECTION .data
msg:   db "Enter 5 numbers:",10,0
fmt_i: db "%ld",0
out:   db "arr[%ld] = %ld",10,0

SECTION .bss
arr:   resq 5

SECTION .text
global main
extern printf, scanf

main:
    ; print message
    mov rdi, msg
    xor rax, rax
    call printf

    ; i = 0
    xor rcx, rcx

read_loop:
    cmp rcx, 5
    jge show

    ; scanf("%ld", &arr[i])
    mov rdi, fmt_i
    lea rsi, [arr + rcx*8]      ; lea = address
    xor rax, rax
    call scanf

    inc rcx
    jmp read_loop

show:
```

```
    xor rcx, rcx

print_loop:
    cmp rcx, 5
    jge done

    ; printf("arr[%ld] = %ld\n", i, arr[i])
    mov rdi, out
    mov rsi, rcx                ; index
    mov rdx, [arr + rcx*8]      ; mov = value
    xor rax, rax
    call printf

    inc rcx
    jmp print_loop

done:
    xor eax, eax
    ret
```

`mov` vs `lea` (Very Important)

- **`mov rax, [arr]`** loads the **value** stored at memory address `arr`
- **`lea rax, [arr]`** loads the **address** of `arr` (pointer)
- **For `scanf`** you must pass an **address** ⇒ use `lea`
- **For `printf`** when printing array values, you pass the **value** ⇒ use `mov`

14. Functions in Assembly

14.1 Function Example: Add Two Numbers

[Function + Example Call (System V ABI)]

```
; Function: add_two(a, b) => returns a+b in RAX
; In x86_64 System V calling convention:
;   1st arg -> RDI
;   2nd arg -> RSI
;   return  -> RAX

SECTION .text
global add_two
global main
extern printf

SECTION .data
msg: db "Sum = %ld",10,0

add_two:
    mov rax, rdi
    add rax, rsi
    ret

main:
    ; call add_two(50, 70)
    mov rdi, 50
    mov rsi, 70
    call add_two

    ; print result in RAX
    mov rdi, msg
    mov rsi, rax
    xor rax, rax
    call printf

    xor eax, eax
    ret
```