Linux x86-64 Assembly with NASM

A Register-First Guide from Zero to Confident Systems Programming

A5 Technical Booklet

Fedora Linux

NASM

Registers First

Prepared by Ayman Alheraki

How to Use This Book

> **Method:** Every page teaches one mental unit. Read a page, type the code, step through it, and write the register contract in your own words.

Part	What it covers
Foundations	Lessons 1-14: machine state, values, registers, sections, addressing.
Linux syscalls and debugging	Lessons 15-24: kernel boundary, read/write/exit, GDB, flags, jumps.
Subroutines and stack	Lessons 25-34: stack, CALL/RET, ABI, local variables, reusable helpers.
Macros and source structure	Lessons 35-39: NASM macro thinking, local labels, include files.
Practical routines	Lessons 40-58: errors, division, printing, parsing, arrays, strings, structs.
Interop and tooling	Lessons 59-78: C ABI, relocations, make, compiler output, object inspection.
Projects and reference cards	Lessons 79-100 plus appendices: complete programs and quick lookup pages.

This booklet targets Linux x86-64 user-space programming with NASM and ELF64. It deliberately starts from registers and data movement because that is where most learners get lost.

Learning promise

The guide avoids unnecessary mystery. It explains instructions through machine state: registers, memory, flags, stack, syscalls, and ABI contracts. It is not a complete instruction-set encyclopedia. It is a practical path from confusion to confident reading and writing.

1. The Problem: Registers Feel Like a Maze

001

Build a mental map before writing instructions.

Most beginners lose the path because assembly exposes everything that high-level languages hide: registers, addresses, flags, stack, object files, and the operating-system boundary. The solution is not to memorize hundreds of instructions first. The solution is to understand what a value is, where it lives, who owns it, and how it moves.

This guide uses a register-first method. Every lesson answers four questions: what registers are involved, what memory is touched, what contract must be respected, and how to trace the result in a debugger.

NASM is used because its Intel-like syntax is readable for x86-64 learners on Linux. The same ideas apply when later reading GAS, compiler output, or disassembly.

Question	Assembly answer
Where is the value?	In a register, memory address, immediate constant, or stack slot.
Who may overwrite it?	The instruction, syscall, or called routine decides.
How do I verify?	Stop in GDB and inspect registers and memory.

Practice tips

- Do not learn assembly as text. Learn it as machine state changing step by step.

2. The Four Places a Value Can Live

002

Stop confusing values, addresses, labels, and memory cells.

A value can be immediate data encoded in the instruction, a register value inside the CPU, a memory value stored at an address, or a label resolved by the assembler/linker. Most confusion begins when a programmer treats an address as if it were the value stored at that address.

In NASM, brackets mean memory access. Without brackets, a label usually means its address. The distinction is one of the most important rules in the whole language.

Form	Meaning
1234	Immediate constant encoded in the instruction
rax	The contents of register RAX
number	A symbol/address
[number]	Memory contents at the address named number

Address vs value vs immediate

```
section .data
    number dq 1234

section .text
    ; address of variable
    lea rax, [rel number]

    ; value stored at variable
    mov rbx, [rel number]

    ; immediate constant
    mov rcx, 1234
```

Warning
If your code “almost works”, inspect whether you passed an address when a value was required, or a value when an address was required.

3. What NASM Produces on Linux 003

Understand source, object file, linker, and executable.

NASM does not normally create a Linux executable directly when using ELF64. It creates an object file. The linker combines object files, resolves symbols, and writes the final executable. This separation is important because assembly code can call other assembly files, C functions, or library code after linking.

For pure Linux x86-64 examples in this book, the common command is nasm -f elf64 file.asm -o file.o, then ld file.o -o file.

Minimal build and run loop

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

echo $?        # print process exit status
```

Practice tips

- Object files are not executable programs. They are relocatable building blocks.

4. Fedora Toolchain Setup 004

Install the smallest useful assembly toolkit.

On Fedora and similar distributions, NASM, binutils, GCC, GDB, make, and man-pages are enough for the learning path in this book. GCC is useful even when you write assembly by hand because it can generate assembly for comparison and link with the C runtime when needed.

The command below is intentionally conservative. Package names may vary slightly across distributions, but the learning model is identical.

Fedora-oriented installation commands

```
sudo dnf install nasm binutils gcc gdb make man-pages
nasm -v
ld -v
gdb --version
```

Practice tips

- On Debian/Ubuntu, use sudo apt install nasm binutils gcc gdb make manpages-dev.

5. Your First Complete Program 005

See the whole skeleton before studying every detail.

A Linux program can start at a symbol named _start when linked directly with ld. The kernel enters the program at _start. There is no C runtime, no main, and no automatic return to a caller. Therefore, the program must explicitly exit using the exit system call.

This example writes bytes to standard output and exits with code 0. Do not rush past it: every later program is a controlled variation of this skeleton.

hello.asm

```
global _start

section .data
    msg db "Hello from NASM on Linux", 10
    len equ $ - msg

section .text
_start:
    mov eax, 1          ; syscall number: write
    mov edi, 1          ; fd = stdout
    lea rsi, [rel msg]  ; buffer address
    mov edx, len        ; byte count
    syscall

    mov eax, 60         ; syscall number: exit
    xor edi, edi        ; status = 0
    syscall
```

6. The Register Family Tree

006

Understand RAX/EAX/AX/AL as overlapping views.

x86-64 general-purpose registers are 64-bit registers, but many names refer to smaller parts of the same physical register. RAX is 64 bits, EAX is the low 32 bits, AX is the low 16 bits, AL is the low 8 bits, and AH is another old 8-bit slice inside AX.

This overlapping design is powerful, but it can surprise beginners. Writing to EAX clears the upper 32 bits of RAX in 64-bit mode. Writing to AX or AL does not clear the rest.

RAX general	**RBX** general	**RCX** general	**RDX** general
RSI general	**RDI** general	**RSP** stack	**RBP** general
R8 general	**R9** general	**R10** general	**R11** general
R12 general	**R13** general	**R14** general	**R15** general

Name	Bits	Effect when written
rax	64	Replaces all 64 bits
eax	32	Writes low 32 and zeroes high 32
ax	16	Writes low 16 only
al	8	Writes low 8 only

Warning
The zero-extension rule for 32-bit writes is a friend when you know it, and a bug when you forget it.

7. Register Roles in a Program 007

Think in roles, not just names.

A register can be a temporary value, a function argument, a syscall argument, a pointer, a loop counter, or a saved value. The same physical register may play different roles at different points in the program.

Good assembly code writes these roles into comments. Bad assembly code forces the reader to reverse-engineer the intention from the instructions.

Comments explain the role of each register

```
; rdi = pointer to string
; rax = current length
strlen_loop:
    cmp byte [rdi + rax], 0
    je  strlen_done
    inc rax
    jmp strlen_loop
strlen_done:
```

Practice tips

- Before each loop, write a register contract: inputs, outputs, clobbers.

8. The Minimal Register Map You Must Memorize

008

Memorize only the registers that unlock real programs.

You do not need to memorize everything on day one. You need the handful of registers that appear constantly in Linux x86-64 assembly: syscall registers, function-call registers, stack registers, and return registers.

This table is the first serious checkpoint in the book. Keep it visible until it becomes natural.

Purpose	Registers
Syscall number	rax/eax
Syscall args	rdi, rsi, rdx, r10, r8, r9
Function args (SysV ABI)	rdi, rsi, rdx, rcx, r8, r9
Return value	rax, sometimes rdx too
Stack pointer	rsp
Frame pointer	rbp
Callee-saved	rbx, rbp, r12-r15

Warning
Function argument 4 uses RCX, but syscall argument 4 uses R10. This difference is a classic x86-64 Linux trap.

9. MOV Does Not Mean Everything Moves

009

Know what is copied and what is not.

MOV copies data from source to destination. It does not erase the source. It does not allocate memory. It does not automatically know the size of a memory operand unless the other operand or a size keyword tells it.

In memory-to-register and register-to-memory operations, think of MOV as copying bits between storage places.

MOV copies; it does not transfer ownership

```
mov rax, 10          ; RAX = 10
mov rbx, rax         ; RBX = copy of RAX
mov [rel value], rbx ; store RBX into memory
mov rcx, [rel value] ; load memory into RCX

section .data
value dq 0
```

10. Sizes: Byte, Word, Dword, Qword

010

Make operand size explicit when needed.

Assembly is precise about data size. A byte is 8 bits, a word is 16 bits, a doubleword is 32 bits, and a quadword is 64 bits. NASM often infers the size from a register, but memory-only operations may require explicit size.

When a program behaves strangely, check whether you loaded 1 byte when you meant 8, or stored 8 bytes when the variable only reserved 1.

NASM type	Bytes	Common register
db	1	al/bl/cl/dl
dw	2	ax/bx/cx/dx
dd	4	eax/ebx/ecx/edx
dq	8	rax/rbx/rcx/rdx

Data declarations and matching loads

```
section .data
    b db 7       ; 1 byte
    w dw 700     ; 2 bytes
    d dd 70000   ; 4 bytes
    q dq 70000   ; 8 bytes

section .text
    mov al,  [rel b]
    mov ax,  [rel w]
    mov eax, [rel d]
    mov rax, [rel q]
```

11. Little Endian Without Fear 011

Understand byte order in memory.

x86-64 is little-endian: the least significant byte of a multi-byte integer is stored at the lowest address. This does not change the numeric value in the register. It only changes how the value is arranged in memory.

When debugging with memory dumps, endian order is visible. When debugging with register dumps, you usually see the whole value as a normal number.

Memory layout of a 64-bit value

```
section .data
    value dq 0x1122334455667788

; memory bytes at value:
; 88 77 66 55 44 33 22 11
```

12. Labels, EQU, and $ 012

Use assembler-time names to make programs readable.

Labels name addresses. EQU defines assembly-time constants. The special symbol $ means the current assembly position. The common expression len equ $ - msg computes a byte length at assembly time, not at runtime.

This is one of the first places where assembly becomes easier than expected: the assembler can compute many constants for you.

Assembler-time length

```
section .data
msg db "ABC", 10
len equ $ - msg        ; len = 4 bytes

section .text
    mov edx, len       ; pass length to write
```

13. Sections: .text, .data, .bss 013

Put code and data in the right regions.

A typical ELF64 program uses .text for executable instructions, .data for initialized data, and .bss for zero-initialized storage. These names are not magic decorations; they guide the linker and loader in how the program should be mapped into memory.

Use .bss for buffers that do not need initial bytes in the file. It keeps the executable smaller.

The three common sections

```
section .text
    ; instructions

section .data
    greeting db "Hi", 10

section .bss
    buffer resb 4096
```

Practice tips

- A large zeroed buffer belongs in .bss, not .data.

14. Addressing Mode Formula 014

Read memory operands as base + index*scale + displacement.

Most x86-64 memory operands can be understood as base + index * scale + displacement. This is perfect for arrays and structures. The scale can be 1, 2, 4, or 8.

Do not try to translate every operand into English. Translate it into a simple address formula.

Part	Example	Meaning
Base	rdi	start address
Index	rcx	element number
Scale	8	element size
Displacement	16	constant byte offset

Effective addresses

```
; Load array[i] where array contains 64-bit integers
; rdi = address of array
; rcx = index i
mov rax, [rdi + rcx*8]

; Store 42 into array[i + 2]
mov qword [rdi + rcx*8 + 16], 42
```

15. The Syscall Boundary 015

Know when your code is talking to the kernel.

A syscall transfers control from user space to the Linux kernel. You place the syscall number in RAX and arguments in the required registers, execute syscall, and read the result from RAX.

A syscall is not a normal function call. It has a different register convention and may destroy RCX and R11. Keep syscall wrappers small and explicit.

Register	Syscall meaning
rax	syscall number before, return value after
rdi	argument 1
rsi	argument 2
rdx	argument 3
r10	argument 4
r8	argument 5
r9	argument 6

Warning
Do not put syscall argument 4 in RCX on Linux x86-64. Use R10.

16. WRITE: Your First Useful Syscall

016

Master the simplest visible output.

The write syscall sends bytes from a buffer to a file descriptor. File descriptor 1 is standard output. The kernel does not know or care whether the bytes are “text” unless the terminal interprets them that way.

The arguments are fd, buffer address, and byte count. If you pass the wrong count, you may print extra bytes or miss the newline.

write register setup

```
; ssize_t write(int fd, const void *buf, size_t count)
mov eax, 1          ; SYS_write
mov edi, 1          ; stdout
lea rsi, [rel msg]  ; address of bytes
mov edx, len        ; number of bytes
syscall
```

17. EXIT: Never Fall Off _start 017

Terminate explicitly when there is no C runtime.

When your entry point is _start and you link with ld directly, there is no caller waiting for a return. Using ret from _start is wrong. Use the exit syscall.

The exit code is visible to the shell with echo $?. This is a useful debugging channel even before you know printing routines.

Exit with status 7

```
mov eax, 60     ; SYS_exit
mov edi, 7      ; process status code
syscall
```

Practice tips

- Use exit codes for tiny tests: return 0 for success, nonzero for a failure condition.

18. READ: Input Is Just Bytes 018

Read from standard input into a buffer.

The read syscall fills a buffer with bytes and returns the number of bytes read in RAX. A terminal line usually includes the newline byte if the user pressed Enter.

Never assume the whole buffer was filled. Always use the returned byte count.

read from stdin

```
section .bss
    buf resb 128

section .text
    mov eax, 0          ; SYS_read
    mov edi, 0          ; stdin
    lea rsi, [rel buf]
    mov edx, 128
    syscall             ; rax = bytes read
```

Warning
The returned byte count may be 0 at EOF, positive on success, or negative on error.

19. GDB: Look at the Machine State

019

Make registers visible while learning.

Assembly becomes far easier when you stop guessing. Use GDB to single-step and inspect registers. The command layout regs gives a live register view in many terminals. The command x examines memory.

The goal is not to become a debugger expert immediately. The goal is to verify one instruction at a time.

Minimal GDB session

```
gdb ./hello
(gdb) starti
(gdb) layout regs
(gdb) si
(gdb) info registers rax rdi rsi rdx
(gdb) x/16xb &msg
(gdb) quit
```

20. Arithmetic: ADD, SUB, INC, DEC

020

Update registers and flags deliberately.

Arithmetic instructions usually update flags. This means they not only compute a result, but also prepare information for conditional jumps. This dual effect is central to assembly control flow.

INC and DEC do not update the carry flag in the same way as ADD/SUB, so use ADD/SUB when carry matters.

Simple arithmetic

```
mov rax, 10
add rax, 5        ; rax = 15
sub rax, 3        ; rax = 12
inc rax           ; rax = 13
dec rax           ; rax = 12
```

21. Flags: The Hidden Result 021

Read ZF, CF, SF, and OF as condition data.

The flags register records properties of the last arithmetic or logical operation. ZF means zero result. CF is important for unsigned carry/borrow. SF records sign bit. OF is important for signed overflow.

Do not read flags emotionally. Read them as facts about the last operation. Then choose the jump instruction that matches signed or unsigned logic.

Flag	Meaning	Common use
ZF	result was zero	je/jz, jne/jnz
CF	unsigned carry/borrow	jae/jb, adc/sbb
SF	sign bit of result	signed comparisons
OF	signed overflow	jg/jl with SF/ZF

22. CMP and TEST

022

Compare without keeping the result.

CMP subtracts internally and sets flags but discards the numeric result. TEST computes bitwise AND internally and sets flags but discards the numeric result. These two instructions are the foundation of branches and loops.

Use cmp for ordering and equality. Use test for zero checks and bit masks.

CMP and TEST patterns

```
cmp rax, rbx
je  equal
jl  less_signed
jb  less_unsigned

test rax, rax
jz   rax_is_zero

test al, 1
jnz  low_bit_is_set
```

23. Jumps: The Instruction Pointer Changes

023

Control flow is just changing RIP.

A jump changes where execution continues. Conditional jumps check flags produced by a previous instruction. Unconditional jumps always move execution to the target label.

Most beginner loops are only three parts: initialize, compare, jump back.

A countdown loop

```
mov rcx, 5
.loop:
    ; body here
    dec rcx
    jnz .loop
```

24. Signed vs Unsigned Jumps 024

Choose jumps based on interpretation.

The CPU does not know whether a byte pattern is signed or unsigned. Your jump instruction tells the reader how to interpret the flags. JA/JB are unsigned. JG/JL are signed.

For sizes and counts, use unsigned logic. For mathematical negative/positive values, use signed logic.

Meaning	Unsigned jump	Signed jump
greater	ja	jg
greater/equal	jae	jge
less	jb	jl
less/equal	jbe	jle
equal	je	je

Warning
Using JL for an unsigned length comparison can turn a large positive size into a “negative” value by interpretation.

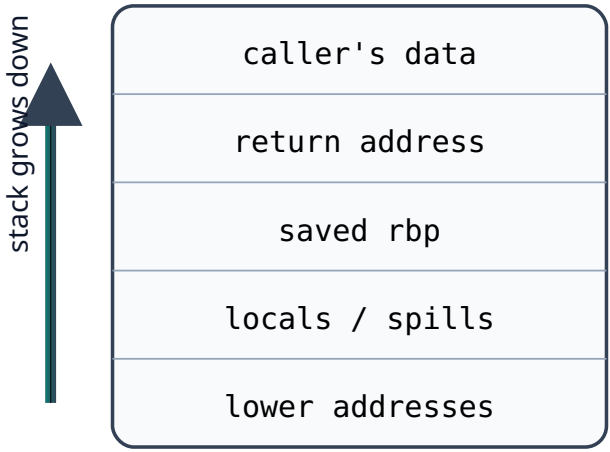
25. The Stack: A LIFO Memory Region

025

Understand RSP before using subroutines.

The stack is memory used for return addresses, saved registers, local variables, and temporary spill storage. It grows downward on x86-64: pushing decreases RSP; popping increases RSP.

CALL pushes a return address. RET pops an address and jumps to it. That is the heart of subroutines.



Push and pop basics

```
push rax        ; rsp -= 8, store rax at [rsp]
pop  rbx        ; load [rsp] into rbx, rsp += 8
```

26. CALL and RET

026

See a subroutine as controlled jump plus return address.

CALL target is not magic. It pushes the address of the next instruction and jumps to target. RET pops that return address into the instruction pointer. If you corrupt the stack, RET returns to nonsense.

This is why stack discipline matters more in assembly than in high-level languages.

A tiny subroutine

```
global _start
section .text
_start:
    call set_answer
    ; rax now contains 42
    mov edi, eax
    mov eax, 60
    syscall

set_answer:
    mov eax, 42
    ret
```

27. Subroutine Contract

027

Write down inputs, outputs, and clobbers.

A subroutine should have a contract. The contract tells the caller which registers contain inputs, where the result appears, and which registers may be destroyed. Without contracts, assembly turns into chaos.

For learning, put the contract above every subroutine. This one practice can remove most register confusion.

A contract-commented subroutine

```
; strlen_z
; in : rdi = pointer to zero-terminated string
; out: rax = length in bytes before zero
; clobbers: rax only
strlen_z:
    xor eax, eax
.loop:
    cmp byte [rdi + rax], 0
    je .done
    inc rax
    jmp .loop
.done:
    ret
```

28. Caller-Saved vs Callee-Saved 028

Know who must preserve which registers.

Under the System V AMD64 calling convention, some registers are caller-saved and some are callee-saved. Caller-saved means the caller must save it if the value is needed after a call. Callee-saved means the called routine must restore it before returning if it modifies it.

This distinction is the key to clean assembly routines that can call each other safely.

Class	Registers	Rule
Caller-saved	rax, rcx, rdx, rsi, rdi, r8-r11	Caller saves if needed across call
Callee-saved	rbx, rbp, r12-r15	Callee restores before ret
Special	rsp	Must be restored exactly

Warning
Never return with a different RSP than you had on entry unless you intentionally changed the calling convention.

29. Function Arguments in System V ABI

029

Call your own routines like C would call functions.

For normal user-space functions on Linux x86-64, the first six integer or pointer arguments use RDI, RSI, RDX, RCX, R8, and R9. Return values usually use RAX. This is different from the syscall register sequence.

If you learn this well, you can write assembly routines callable from C and call C routines from assembly.

System V function argument registers

```
; long add3(long a, long b, long c)
; a in rdi, b in rsi, c in rdx, return in rax
add3:
    lea rax, [rdi + rsi]
    add rax, rdx
    ret
```

30. Stack Alignment Before CALL 030

Keep the ABI promise.

The System V ABI requires stack alignment at function call boundaries. A practical rule: before calling an external C function, ensure RSP is 16-byte aligned at the call instruction so that the called function sees the expected entry alignment.

Alignment bugs can be invisible until a function uses SSE instructions or a library routine assumes alignment.

Alignment adjustment pattern

```
; Example from _start before calling a C function-like routine:
; Save alignment if needed. In many tiny pure-asm examples,
; direct syscalls avoid this issue. Before calling libc, care.
sub rsp  8        ; adjust if required by current entry state
call some_function
add rsp  8
```

Warning
Do not cargo-cult sub rsp,8 everywhere. Know your entry alignment and every push you made.

31. Stack Frame with RBP

031

Use a frame pointer while learning locals.

Modern compilers often omit frame pointers, but beginners benefit from using RBP as a stable base for stack locals. The classic prologue saves the old RBP, makes RBP point to the current frame, then reserves local space.

Once you understand stack frames, you can write optimized leaf routines without RBP.

Classic stack frame

```
my_func:
    push rbp
    mov  rbp, rsp
    sub  rsp, 32        ; local space

    mov qword [rbp-8], 123

    mov  rsp, rbp
    pop  rbp
    ret
```

32. Leaf Functions

032

Use tiny routines without stack frames when safe.

A leaf function does not call another function. If it needs no stack storage and uses only caller-saved registers, it can be extremely small. Many assembly helpers should be leaf functions.

Small leaf routines make register tracking easier because there are fewer contracts to preserve.

A clean leaf function

```
; max_u64(rdi=a, rsi=b) -> rax
max_u64:
    mov rax, rdi
    cmp rax, rsi
    jae .done
    mov rax, rsi
.done:
    ret
```

33. A Reliable Print String Wrapper

033

Hide syscall setup behind a routine.

A wrapper routine reduces repeated code. The caller provides a pointer and length. The routine performs the write syscall. This makes the main program read like a higher-level program while preserving assembly control.

The routine clobbers RAX because syscall returns in RAX. It also uses the syscall argument registers.

Reusable write wrapper

```
; print_buf
; in : rdi = buffer address, rsi = length
; out: rax = bytes written or negative error
; clobbers: rax, rdi, rsi, rdx, rcx, r11
print_buf:
    mov rdx, rsi
    mov rsi, rdi
    mov edi, 1
    mov eax, 1
    syscall
    ret
```

34. Print a Zero-Terminated String

034

Combine two subroutines safely.

A higher-level helper can call strlen_z, then print_buf. Because the pointer in RDI is needed after strlen_z, the routine must preserve it. The easiest beginner method is pushing it on the stack.

This lesson shows why caller-saved registers matter: strlen_z returns length in RAX and may leave other caller-saved registers changed by contract.

Composition by contract

```
; print_z
; in : rdi = zero-terminated string
; out: rax = write result
print_z:
    push rdi
    call strlen_z
    mov rsi, rax      ; length
    pop rdi           ; original pointer
    call print_buf
    ret
```

35. Why Macros Are Not Functions

035

Separate compile-time expansion from runtime calls.

A macro expands before assembly produces machine code. A subroutine executes at runtime and uses CALL/RET. A macro can reduce typing and enforce patterns, but it does not create a runtime function by itself.

Use subroutines for behavior shared at runtime. Use macros for repeated instruction patterns, constants, boilerplate, and source-level safety.

Feature	Subroutine	Macro
Exists at runtime	Yes	No, expands into code
Has CALL/RET overhead	Yes	No unless macro emits call
Can reduce code size	Yes	Usually no
Can generate specialized code	Limited	Yes
Debugging	Step into routine	Step through expanded instructions

36. NASM Single-Line Macros 036

Use %define for names and tiny expressions.

%define creates a preprocessor macro. It is useful for syscall numbers, buffer sizes, and small expressions. Keep simple macros simple. Do not hide too much behavior in one line.

A good macro makes code easier to inspect. A bad macro turns assembly into a private language.

Readable constants

```
%define SYS_write 1
%define SYS_exit  60
%define STDOUT    1
%define NL        10

mov eax  SYS_write
mov edi  STDOUT
```

37. NASM Multi-Line Macros 037

Generate repeated instruction sequences.

%macro defines a block that expands wherever invoked. Parameters are referenced as %1, %2, and so on. Multi-line macros are useful for syscall wrappers, prologues, epilogues, and repetitive checks.

Use them carefully: every macro invocation duplicates instructions in the final code.

A multi-line macro

```
%macro EXIT 1
    mov eax, 60
    mov edi, %1
    syscall
%endmacro

_start:
    EXIT 0
```

38. Macro-Local Labels 038

Avoid label collisions inside macros.

If a macro emits labels and you invoke the macro more than once, normal labels collide. NASM supports macro-local labels with %%label. Each expansion receives a unique internal name.

This is essential for macros containing loops or branches.

Macro-local label with %%

```
%macro SPIN_N 1
    mov rcx, %1
%%loop:
    dec rcx
    jnz %%loop
%endmacro

SPIN_N 100
SPIN_N 200
```

39. Include Files for Constants and Macros

039

Create a small personal standard library.

As programs grow, put constants and safe macros in include files. This makes examples shorter and reduces the risk of inconsistent syscall numbers or repeated boilerplate.

Keep include files boring. They should clarify, not hide the machine.

Simple include file

```
; file: linux64.inc
%define SYS_read   0
%define SYS_write  1
%define SYS_open   2
%define SYS_close  3
%define SYS_exit   60
%define STDIN      0
%define STDOUT     1

; file: main.asm
%include "linux64.inc"
```

40. Error Results from Syscalls 040

Recognize negative return values.

Linux syscalls return a nonnegative result on success for many calls and a negative errno value on error. A simple pure-assembly program may check whether RAX is negative using JS, because negative values have the sign bit set.

For robust user tools, convert errors to messages or exit codes. For learning, detect them early.

Basic syscall error check

```
syscall
cmp rax, 0
js  .error        ; rax < 0 signed means error-like result

; success path
jmp .done
.error:
    mov edi, 1    ; exit status 1
    mov eax, 60
    syscall
.done:
```

41. Number Output: Why It Feels Hard 041

Convert binary integers to decimal text.

Registers store binary integers. Terminals display bytes, often ASCII. To print a number as decimal, you must repeatedly divide by 10, collect remainders, convert each digit to ASCII, and output digits in reverse order.

This is a perfect exercise because it forces you to use registers, stack/buffer storage, loops, and syscalls together.

Algorithm before assembly

```
; unsigned decimal idea:
; while n != 0:
;   digit = n % 10
;   push '0' + digit
;   n = n / 10
; pop digits to output order
```

42. DIV: RDX:RAX Divided by Operand

042

Use division without mysterious crashes.

Unsigned DIV on 64-bit operands divides the 128-bit value RDX:RAX by the operand. The quotient goes to RAX, and the remainder goes to RDX. Therefore, for ordinary 64-bit division, clear RDX first unless you intentionally have a high half.

Division exceptions often come from forgetting RDX or dividing by zero.

Unsigned divide by 10

```
mov rax, 12345
xor edx, edx      ; high half = 0
mov rcx, 10
div rcx           ; rax = 1234, rdx = 5
```

Warning
Before DIV, always ask: what is in RDX?

43. Print Unsigned Decimal Routine

043

Build a real reusable helper.

This routine converts RDI to decimal and writes it. It uses a stack buffer in .bss to store digits from the end backward. It demonstrates a practical pattern: write bytes into memory, then call write once.

A production-quality routine would preserve more context or use a documented ABI contract. For learning, the contract is explicit.

A useful number printer

```
section .bss
    numbuf resb 32

; print_u64
; in : rdi = unsigned integer
; out: rax = write result
; clobbers: rax, rbx, rcx, rdx, rsi, rdi, r11
print_u64:
    lea rsi, [rel numbuf + 32]
    mov rax, rdi
    mov ebx, 10
    test rax, rax
    jnz .loop
    dec rsi
    mov byte [rsi], '0'
    jmp .emit
.loop:
    xor edx, edx
    div rbx
    add dl, '0'
    dec rsi
    mov [rsi], dl
    test rax, rax
    jnz .loop
.emit:
    lea rdx, [rel numbuf + 32]
    sub rdx, rsi
    mov eax, 1
    mov edi, 1
    syscall
    ret
```

44. Parsing Unsigned Decimal 044

Turn input bytes into a register value.

Parsing is the reverse of printing. Start with result = 0. For each digit, result = result * 10 + digit. Stop at newline, zero, or a nondigit depending on your contract.

Use IMUL or LEA patterns for multiplication by 10. Start with correctness before micro-optimizing.

Simple decimal parser

```
; parse_u64
; in : rdi = pointer to bytes
; out: rax = value, rdi points after last digit
parse_u64:
    xor eax, eax
.loop:
    movzx ecx, byte [rdi]
    cmp cl, '0'
    jb .done
    cmp cl, '9'
    ja .done
    sub cl, '0'
    imul rax, rax, 10
    add rax, rcx
    inc rdi
    jmp .loop
.done:
    ret
```

45. Arrays: One Formula Repeated

045

Use base + index * element_size.

An array is a contiguous sequence of elements. The CPU does not know the element type. You compute the address of the element you want.

For qword arrays, multiply index by 8. For dword arrays, multiply by 4. For bytes, scale is 1 and can be omitted.

Summing a qword array

```
section .data
arr dq 10, 20, 30, 40

section .text
    xor ecx, ecx       ; i = 0
    xor eax, eax       ; sum = 0
.loop:
    add rax, [rel arr + rcx*8]
    inc rcx
    cmp rcx, 4
    jb .loop
```

46. Strings Are Byte Arrays with Agreements

046

Do not assume one string model.

A string can be length-prefixed, zero-terminated, or simply a pointer plus length. Linux write uses pointer plus length. C often uses zero-terminated strings. Your assembly routine must declare which model it expects.

Most bugs in beginner string code come from mixing these models.

Model	End condition	Common use
pointer + length	count register/field	write syscall, binary buffers
zero-terminated	byte 0	C strings, simple examples
line input	newline or returned count	terminal input

47. Memory Copy Loop

047

Copy bytes with clear source/destination roles.

A byte copy routine is a perfect register discipline exercise. Keep source, destination, and count in separate registers. Decide whether the routine returns anything.

This simple version copies forward and does not handle overlapping regions like memmove.

Forward byte copy

```
; memcpy_simple
; in : rdi = dst, rsi = src, rdx = count
; out: rax = original dst
memcpy_simple:
    mov rax, rdi
.loop:
    test rdx, rdx
    jz .done
    mov cl, [rsi]
    mov [rdi], cl
    inc rsi
    inc rdi
    dec rdx
    jmp .loop
.done:
    ret
```

48. REP MOVSB: A Hardware-Assisted Pattern

048

Know the compact string instruction form.

x86 has string instructions that use implicit registers. REP MOVSB copies RCX bytes from RSI to RDI. It is concise but hides operands, so beginners should first understand the explicit loop.

Once the explicit loop is clear, REP MOVSB becomes a readable idiom.

String instruction copy

```
; in: rdi = dst, rsi = src, rcx = count
cld             ; ensure forward direction
rep movsb       ; copy rcx bytes
```

Warning
The direction flag should be clear for forward string operations. Many ABIs expect DF clear across calls.

49. Command-Line Arguments at _start

049

Read argc and argv from the initial stack.

When the kernel starts a program at _start, the initial stack contains argc followed by argv pointers, then environment pointers. If you bypass the C runtime, you can read them directly from RSP.

This is advanced compared with a normal C main, but it reveals how programs really receive startup data.

Initial stack layout

```
_start:
    mov rdi, [rsp]          ; argc
    lea rsi, [rsp + 8]      ; argv array address
    ; argv[0] is [rsi]
    ; argv[1] is [rsi+8] if argc > 1
```

50. Environment Pointers

050

Find envp after argv.

After argc and argv pointers, the stack contains a NULL pointer, then environment pointers, then another NULL. You can scan past argv to reach envp.

This is not necessary for basic assembly, but it helps you understand the loader and process startup.

Locate envp

```
mov rcx, [rsp]          ; argc
lea rbx, [rsp + 8]      ; &argv[0]
lea rbx, [rbx + rcx*8 + 8] ; skip argv pointers and NULL
; rbx now points to envp[0]
```

51. Opening Files with Syscalls 051

Use file descriptors beyond stdin/stdout.

The open syscall returns a file descriptor on success. You then use read, write, and close on that descriptor. Modern Linux code often uses openat, but open remains easy for learning on x86-64 syscall tables.

Always check for negative return values before using a returned file descriptor.

Open a file read-only

```
%define SYS_open  2
%define SYS_close 3
%define O_RDONLY  0

lea rdi, [rel path]
mov esi, O_RDONLY
xor edx, edx
mov eax, SYS_open
syscall             ; rax = fd or negative error
```

52. Echo Program: Read Then Write

052

Build a tiny useful filter.

A classic assembly exercise is an echo program: read bytes from stdin into a buffer, then write exactly the number of bytes read to stdout. This teaches one of the strongest habits in systems programming: trust returned counts, not buffer capacity.

The same pattern is the basis of filters, file copy tools, and protocol readers.

stdin to stdout

```
section .bss
buf resb 4096

section .text
.loop:
    mov eax, 0
    mov edi, 0
    lea rsi, [rel buf]
    mov edx, 4096
    syscall
    test rax, rax
    jle .done

    mov rdx, rax
    mov eax, 1
    mov edi, 1
    lea rsi, [rel buf]
    syscall
    jmp .loop
.done:
    mov eax, 60
    xor edi, edi
    syscall
```

53. Heap-Like Memory with BRK 053

See process memory growth simply.

The brk syscall changes the end of the process data segment. Calling brk with zero is a common way to query the current break. Then you can request a larger break.

For serious programs, libc malloc or mmap is easier and safer. But learning brk shows how low-level allocation begins.

brk basics

```
; get current program break
mov eax, 12        ; SYS_brk
xor edi, edi
syscall            ; rax = current break

; request 4096 more bytes
lea rdi, [rax + 4096]
mov eax, 12
syscall
```

54. mmap: Page-Oriented Memory

054

Map pages directly from the kernel.

mmap can allocate anonymous memory or map files. The syscall has six arguments, so it demonstrates the full x86-64 syscall register set. Argument 4 goes in R10, not RCX.

This is one of the best examples for proving that function calls and syscalls use different fourth argument registers.

Anonymous mmap syscall

```
%define SYS_mmap 9
%define PROT_READ  1
%define PROT_WRITE 2
%define MAP_PRIVATE   2
%define MAP_ANONYMOUS 32

xor edi,  edi                       ; addr = NULL
mov esi,  4096                      ; length
mov edx,  PROT_READ | PROT_WRITE
mov r10d, MAP_PRIVATE | MAP_ANONYMOUS
mov r8d,  -1                        ; fd
xor r9d,  r9d                       ; offset
mov eax,  SYS_mmap
syscall
```

55. Bit Masks

055

Operate on individual bits with confidence.

Bit operations are common in systems code: permissions, flags, hardware registers, protocol fields, and packed data. AND clears or tests, OR sets, XOR toggles, and NOT inverts.

Write masks in binary when the bit pattern matters for teaching. Write masks in hex when compactness matters.

Flag manipulation

```
%define FLAG_READ  0b001
%define FLAG_WRITE 0b010
%define FLAG_EXEC  0b100

or  eax  FLAG_WRITE     ; set write bit
and eax  ~FLAG_EXEC     ; clear exec bit
test eax  FLAG_READ     ; set ZF if read bit is absent
```

56. Shifts and Multiplication by Powers of Two

056

Use SHL/SHR/SAR deliberately.

SHL shifts left and often corresponds to multiplying unsigned values by powers of two. SHR shifts right logically. SAR shifts right arithmetically and preserves sign for signed values.

Do not use shifts as decoration. Use them when the operation is genuinely bit-level or power-of-two arithmetic.

Shift patterns

```
mov rax, 7
shl rax, 3        ; 7 * 8 = 56
shr rax, 1        ; logical divide by 2 for unsigned-like value

mov rbx, -8
sar rbx, 1        ; signed-like divide by 2 -> -4
```

57. LEA Is Arithmetic, Not Only Addresses

057

Use LEA for non-destructive integer formulas.

LEA computes an effective address expression and stores the result, but it does not access memory. This makes it a useful arithmetic instruction for sums and scaled additions.

LEA does not update flags, which is sometimes helpful when you need to preserve flags from a comparison.

LEA arithmetic

```
lea rax, [rdi + rsi*4]      ; rax = rdi + 4*rsi
lea rbx, [rax + rax*2]      ; rbx = 3*rax
```

58. Structures by Offsets 058

Manually encode layout agreements.

A structure is memory plus an agreement about offsets and sizes. In assembly, you often define constants for field offsets. Then use base + offset to access fields.

When interoperating with C, the C compiler decides the exact layout including padding. Verify with sizeof and offsetof or generated assembly.

Structure offsets

```
%define POINT_X 0
%define POINT_Y 8
%define POINT_SIZE 16

; rdi = pointer to Point { long x; long y; }
mov rax, [rdi + POINT_X]
add rax, [rdi + POINT_Y]
```

59. Alignment: Correctness and Speed

059

Respect natural boundaries when practical.

Modern x86-64 can handle many unaligned accesses, but alignment still matters for performance and ABI correctness. Align frequently accessed data and stack frames where required.

NASM align can pad the current section to a boundary. Use it for tables and data that benefit from predictable placement.

Align data to 16 bytes

```
section .data
align 16
vector_data dq 1, 2, 3, 4
```

60. Calling C from Assembly 060

Use the ABI to access libc routines.

When linking with GCC and libc, you can call functions such as puts or printf. Now you must obey the function calling convention, stack alignment, and special rules for variadic functions like printf.

For printf with vector arguments, AL communicates the number of vector registers used. For simple integer/string calls, clear EAX before calling printf.

Assembly main calling printf

```
global main
extern printf

section .data
fmt db "value = %ld", 10, 0

section .text
main:
    push rbp
    mov rbp, rsp
    lea rdi, [rel fmt]
    mov esi, 42
    xor eax, eax      ; no vector args for printf
    call printf
    xor eax, eax
    pop rbp
    ret
```

61. Calling Assembly from C

061

Export a NASM routine with global.

A NASM routine can be called from C if it follows the ABI. Declare the symbol global in NASM, declare it extern in C, assemble to an object file, and link with GCC.

This is one of the most practical uses of assembly: write a focused low-level routine and call it from a higher-level program.

Exported assembly function

```
; add3.asm
global add3
section .text
add3:
    lea rax, [rdi + rsi]
    add rax, rdx
    ret

; build idea:
; nasm -f elf64 add3.asm -o add3.o
; gcc main.c add3.o -o app
```

62. Recursion and the Stack 062

Respect call depth and saved state.

Recursive assembly functions are possible, but they make stack discipline unavoidable. Every call creates another return address and may need saved arguments or locals.

Iteration is usually clearer for early assembly learning. Use recursion only after CALL/RET and stack frames are fully understood.

Simple recursive factorial

```
; factorial(n) in rdi -> rax
factorial:
    cmp rdi, 1
    ja .recurse
    mov eax, 1
    ret
.recurse:
    push rdi
    dec rdi
    call factorial
    pop rdi
    imul rax, rdi
    ret
```

63. Jump Tables

063

Use data to select code paths.

A jump table stores addresses of labels or functions. It is useful for interpreters, dispatchers, and switch-like logic. Bounds-check the index before jumping.

Indirect jumps are powerful and dangerous. A wrong index becomes a jump into nonsense.

A tiny jump table

```
cmp rdi, 2
ja .bad
lea rbx, [rel table]
jmp [rbx + rdi*8]

.case0: ; ...
    ret
.case1: ; ...
    ret
.case2: ; ...
    ret
.bad:
    ret

table dq .case0, .case1, .case2
```

64. Position-Independent Thinking

064

Use RIP-relative addressing for normal data.

On x86-64 Linux, RIP-relative addressing is the natural way to reference nearby static data. NASM uses rel to request relative addressing in many examples.

This habit helps when code is linked in modern environments and avoids unnecessary absolute addresses.

RIP-relative references

```
lea rsi, [rel msg]    ; good general habit for static data
mov rax, [rel value]

section .data
msg db "text", 10
value dq 123
```

65. Relocations and Symbols 065

Understand what the linker resolves.

A symbol may not have its final runtime address when NASM creates the object file. The object file records relocation information. The linker resolves addresses when producing the executable or shared object.

Use readelf and objdump to inspect what the assembler and linker produced.

Inspect object files and executables

```
readelf -h hello.o
readelf -r hello.o
nm hello.o
objdump -d -Mintel hello
```

66. Makefile for Assembly Projects

066

Remove command-line friction.

Once you have more than one file, write a Makefile. The goal is not sophistication. The goal is repeatability. Build commands should be easy to rerun and easy to review.

Use separate targets for clean, run, and debug.

Minimal Makefile

```
ASM=nasm
LD=ld

all: hello

hello: hello.o
	$(LD) $< -o $@

%.o: %.asm
	$(ASM) -f elf64 $< -o $@

run: hello
	./hello

clean:
	rm -f *.o hello
```

67. Reading Compiler Output 067

Let C teach you ABI patterns.

C compilers are excellent teachers if you read their assembly output at low optimization first, then at higher optimization. Write tiny C functions, compile to assembly, and compare with your own NASM routines.

Remember that GCC outputs AT&T syntax by default. Use -masm=intel to make it easier for NASM learners to read.

Generate Intel-syntax compiler output

```
gcc -O0 -S -masm=intel demo.c -o demo_O0.s
gcc -O2 -S -masm=intel demo.c -o demo_O2.s
less demo_O0.s
```

68. Objdump as a Reality Check 068

See the machine code you actually shipped.

Source code is intention. Machine code is reality. Objdump shows the instructions in an executable or object file after assembly and linking. This is essential when you use macros, alignment, or linker-generated code.

Use Intel disassembly style while learning NASM.

Inspect final program

```
objdump -d -Mintel ./hello
objdump -s -j .data ./hello
readelf -S ./hello
```

69. GDB Memory Examination 069

Treat memory like a visible table.

The x command examines memory. The format x/16xb means examine 16 units, in hexadecimal, as bytes. x/4gx means examine 4 giant words, 8 bytes each. This turns abstract memory into visible bytes.

Always inspect both the register that holds an address and the memory at that address.

Common GDB memory commands

```
(gdb) info registers rsi rdx
(gdb) x/16xb $rsi
(gdb) x/s $rsi
(gdb) x/4gx $rsp
```

70. Debugging Register Confusion

070

Use a checklist, not emotion.

When a program fails, resist the urge to rewrite everything. Ask simple questions: Is the register initialized? Is it still valid after a call or syscall? Is it a value or an address? Is the operand size correct? Was RSP restored?

This checklist catches most beginner bugs.

Symptom	Likely cause
Segmentation fault	Bad pointer, wrong bracket usage, corrupted return address
Wrong printed bytes	Wrong length, wrong buffer pointer, uninitialized data
Return to random place	Stack imbalance before ret
Works before call, fails after	Caller-saved register overwritten
Huge unexpected number	Signed/unsigned interpretation or missing zero-extension

71. Common Bracket Mistakes 071

Use brackets only when you mean memory.

NASM brackets request memory access. Many errors are simple bracket errors: mov rax, value loads an address-like immediate, while mov rax, [rel value] loads the stored contents.

When reading code, pronounce [x] as “memory at x”. This simple habit prevents many mistakes.

Brackets as memory access

```
section .data
value dq 99

section .text
    lea rax, [rel value]      ; address of value
    mov rbx, [rel value]      ; contents: 99
    mov [rel value], qword 7 ; store 7
```

72. Clobbers: What an Operation Destroys

072

Track destructive effects explicitly.

A clobber is a register or flag that an operation overwrites. DIV clobbers RAX and RDX. Syscall clobbers RAX and also RCX/R11. A function call may clobber caller-saved registers.

Write clobbers into subroutine comments until it becomes automatic.

Operation	Common clobbers
syscall	rax, rcx, r11, plus result-specific registers by convention
div r/m64	rax, rdx
mul r/m64	rax, rdx
call function	caller-saved registers may change
cmp/test	flags

73. Multiplication: IMUL Forms 073

Choose the readable form first.

IMUL has several forms. For learning, the three-operand form is often the clearest: imul destination, source, immediate. It computes without requiring RDX:RAX like one-operand multiply/divide forms.

Use LEA for simple scale-add formulas and IMUL for general multiplication.

Readable multiplication patterns

```
imul rax, rdi, 10      ; rax = rdi * 10
imul rax, rsi          ; rax = rax * rsi
lea  rax, [rdi+rdi*4] ; rax = rdi * 5
```

74. The Red Zone

074

Know the 128-byte area below RSP.

On System V AMD64 user-space, leaf functions may use the red zone below RSP for temporary storage without adjusting RSP. However, it is not suitable for kernel code or code where asynchronous contexts may break the assumptions.

Beginners can ignore the red zone at first and use explicit stack frames. Later, it becomes a useful optimization concept.

Prefer explicit stack space while learning

```
; Conservative beginner style:
sub rsp, 32
; use [rsp]..[rsp+31]
add rsp, 32
ret
```

75. Flags After Logical Instructions

075

Know how AND/OR/XOR affect conditions.

Logical instructions set ZF and SF according to the result and clear CF and OF. This makes XOR EAX,EAX a common zeroing idiom that also sets ZF.

If you need to preserve flags from a previous compare, do not insert logical instructions before the conditional jump.

Flags can be overwritten accidentally

```
cmp rax, rbx
; inserting xor ecx, ecx here would overwrite flags
je .equal

xor eax, eax     ; rax = 0 and ZF = 1
```

76. Zeroing Registers

076

Use simple idioms and know their effects.

xor reg, reg is a classic way to set a register to zero. mov eax, 0 is also clear and often assembled efficiently. Writing to a 32-bit register zero-extends to the full 64-bit register.

Choose clarity in beginner code. Later, read performance notes with more context.

Zeroing idioms

```
xor eax  eax     ; rax = 0
mov edi  0       ; rdi = 0 because EDI write zero-extends
sub ecx  ecx     ; rcx = 0, but less visually common
```

77. Conditional Move

077

Avoid branches for small selections.

CMOVcc copies a value if a condition is true, based on flags. It can implement min/max-like logic without a branch. This is useful but should not be forced everywhere.

Start with branches. Use conditional moves when they make the dataflow clearer or performance measurement supports them.

Unsigned min with CMOVA

```
; rax = min unsigned(rdi, rsi)
mov rax, rdi
cmp rdi, rsi
cmova rax, rsi
```

78. System Call Numbers 078

Do not hardcode blindly without a table source.

Syscall numbers are architecture-specific. The examples in this guide target Linux x86-64. Do not copy numbers to ARM64, i386, or macOS and expect them to work.

In installed Linux headers, syscall constants are available through system headers. In pure NASM examples, we define only the small subset we use.

Name	x86-64 number used here
read	0
write	1
open	2
close	3
mmap	9
brk	12
exit	60

Warning
The number is not the whole syscall interface. The register convention matters too.

79. Mini Project: hello.asm

079

Assemble, link, run, and inspect.

This project is the first complete exercise. Type it manually. Build it. Run it. Then inspect it with objdump and step through it with GDB. Do not copy-paste mechanically; the goal is to connect source lines to register changes.

After it works, intentionally break one register at a time and observe the failure.

Project 1: visible output

```
global _start
section .data
msg db "Assembly is machine state made visible", 10
len equ $ - msg
section .text
_start:
    mov eax, 1
    mov edi, 1
    lea rsi, [rel msg]
    mov edx, len
    syscall
    mov eax, 60
    xor edi, edi
    syscall
```

80. Mini Project: strlen and print_z

080

Write higher-level-looking assembly.

This project introduces a string routine and a print routine. The main program should call print_z twice. This proves that routines are reusable only when their contracts are stable.

The exercise is not about saving a few instructions. It is about making assembly understandable.

Project 2 skeleton

```
global _start
section .data
a db "first line", 10, 0
b db "second line", 10, 0
section .text
_start:
    lea rdi, [rel a]
    call print_z
    lea rdi, [rel b]
    call print_z
    mov eax, 60
    xor edi, edi
    syscall
; include strlen_z, print_buf, print_z routines here
```

81. Mini Project: decimal counter 081

Make binary state human-readable.

Write a program that prints numbers from 1 to 10, one per line. Use print_u64, then write a newline. This forces you to preserve the loop counter across calls or keep it in a callee-saved register.

If the loop breaks, ask which call clobbered your counter.

Counter loop using RBX

```
mov rbx, 1
.loop:
    mov rdi, rbx
    call print_u64
    call print_newline
    inc rbx
    cmp rbx, 11
    jb .loop
```

82. Mini Project: echo filter

082

Handle returned byte counts correctly.

Run the echo program with keyboard input and with redirected files. Compare behavior. Notice that read returns the count and write must use that count. This is how robust byte-stream code begins.

Then modify it to stop after the first chunk. Then modify it to count total bytes.

Test echo with redirection

```
./echo < input.txt > output.txt
cmp input.txt output.txt
echo $?
```

83. Mini Project: count bytes

083

Accumulate a result across syscalls.

A byte-counting program repeatedly reads chunks, adds the returned count to a total, and prints the final total. The total must survive read and write calls. Store it in a callee-saved register or memory.

This project is small but teaches the same design used by real Unix filters.

Byte counter core

```
xor r12, r12          ; total bytes, callee-saved by convention
.loop:
    ; read into buffer
    ; if rax <= 0, done
    add r12, rax
    jmp .loop
.done:
    mov rdi, r12
    call print_u64
```

84. Mini Project: sum integers from argv

084

Combine startup stack and parsing.

Read command-line arguments from the initial stack, parse each as an unsigned integer, and print the sum. This project combines argc/argv layout, loops, parse_u64, and print_u64.

It is also a practical demonstration that raw program startup is just memory and pointers.

Project flow

```
; pseudo-flow:
; argc = [rsp]
; argv = rsp + 8
; for i = 1 .. argc-1:
;     parse argv[i]
;     sum += value
; print sum
```

85. Mini Project: file copy

085

Use open/read/write/close together.

A file copy program opens the source, creates/truncates the destination, loops reading and writing chunks, then closes descriptors. The full production version needs careful error handling and partial-write handling.

Even a simplified version teaches the file descriptor model and the discipline of checking return values.

File copy core

```
; simplified loop after fd_in and fd_out exist
.read_loop:
    mov eax, 0
    mov edi, r12d        ; fd_in
    lea rsi, [rel buf]
    mov edx, 4096
    syscall
    test rax, rax
    jle .done

    mov rdx, rax
    mov eax, 1
    mov edi, r13d        ; fd_out
    lea rsi, [rel buf]
    syscall
    jmp .read_loop
```

86. Performance: First Correctness, Then Measurement

086

Avoid premature micro-optimization.

Assembly invites the illusion that every line must be clever. In reality, incorrect clever code is slower than correct clear code because it wastes debugging time and creates hidden bugs.

Write a clear version. Test it. Inspect it. Then measure. Only optimize the part that matters.

Priority	Question
Correctness	Does it produce the right bytes for edge cases?
ABI safety	Does it preserve required registers and stack alignment?
Clarity	Can you trace registers without guessing?
Measurement	Did a benchmark show the bottleneck?
Optimization	Can a change improve the bottleneck safely?

87. Cache Intuition for Assembly Programmers

087

Know why memory access patterns matter.

Registers are fastest. Cache is fast but not free. Main memory is much slower. Sequential access is usually friendlier to the cache than random access. This is why tight loops over arrays can be fast and pointer-chasing can be expensive.

This guide does not turn into a microarchitecture manual, but a systems programmer should develop basic cache intuition.

Practice tips

- Prefer simple linear scans when they solve the problem.
- Do not replace a clear loop with complex code unless measurement justifies it.

88. SIMD: A Door, Not a First Step

088

Understand where vector registers fit.

x86-64 also has XMM/YMM/ZMM vector registers for SIMD work. They are important for high-performance numeric and byte-processing code, but they should come after the scalar foundation is stable.

The System V ABI uses XMM registers for floating-point/vector arguments and returns. Later study of SSE/AVX will build on the same ABI discipline learned here.

Scalar foundation before SIMD

```
; scalar first: understand this deeply
add rax, rbx

; SIMD later: many packed operations exist
; paddb xmm0, xmm1
```

89. When to Use Assembly in Real Projects

089

Apply assembly where it earns its cost.

Assembly is valuable for boot code, kernels, embedded runtimes, JITs, ABI boundaries, highly tuned kernels, reverse engineering, exploit mitigation research, and deep debugging. It is not the best language for every program.

The greatest value of learning assembly is not only writing it. It is understanding what C, C++, Rust, Go, and operating systems eventually become near the machine.

90. Assembly Reading Routine 090

A daily practice method.

Choose a tiny C function. Compile it with -O0 and -O2. Read both assembly outputs. Then write your own NASM version and compare. This practice teaches compiler thinking, ABI contracts, and instruction selection.

Keep examples tiny: add, max, strlen, memcpy, parse integer, count bytes. A small function studied deeply teaches more than a large file skimmed quickly.

Compiler-output exercise

```
cat > demo.c <<'EOF'
long max2(long a, long b) { return a > b ? a : b; }
EOF
gcc -O2 -S -masm=intel demo.c -o demo.s
cat demo.s
```

91. Reference Card: Syscall Registers

091

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	rax = syscall number before syscall; return value after syscall.
2	Arguments: rdi, rsi, rdx, r10, r8, r9.
3	RCX and R11 are clobbered by syscall mechanics.
4	Negative RAX commonly indicates an error result.

92. Reference Card: Function Registers

092

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	Arguments: rdi, rsi, rdx, rcx, r8, r9.
2	Return: rax, and sometimes rdx for wider integer returns.
3	Caller-saved: rax, rcx, rdx, rsi, rdi, r8-r11.
4	Callee-saved: rbx, rbp, r12-r15. RSP must be restored.

93. Reference Card: Memory Operands

093

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	[label] means memory at label.
2	[reg] means memory at address stored in reg.
3	[base + index*scale + disp] is the central x86 addressing formula.
4	Use byte/word/dword/qword when NASM cannot infer size.

94. Reference Card: Jumps

094

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	je/jz: equal or zero.
2	jne/jnz: not equal or not zero.
3	ja/jb: unsigned above/below.
4	jg/jl: signed greater/less.
5	Use cmp for comparisons and test for bit/zero checks.

95. Reference Card: Stack Discipline

095

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	push decreases rsp by 8 in 64-bit mode and stores a qword.
2	pop loads a qword and increases rsp by 8.
3	call pushes a return address then jumps.
4	ret pops a return address and jumps.
5	Every routine must return with RSP balanced.

96. Reference Card: NASM Declarations

096

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	db: define bytes.
2	dw: define words (2 bytes).
3	dd: define doublewords (4 bytes).
4	dq: define quadwords (8 bytes).
5	resb/resq reserve uninitialized storage in .bss.
6	equ defines assembler-time constants.

97. Reference Card: Debug Commands

097

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	starti: start at first instruction.
2	si: step one instruction.
3	ni: next instruction, stepping over calls.
4	info registers: inspect registers.
5	x/16xb ADDR: inspect 16 bytes.
6	disassemble: view instructions.

98. Reference Card: Build Commands

098

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	nasm -f elf64 file.asm -o file.o
2	ld file.o -o file
3	./file
4	objdump -d -Mintel file
5	readelf -S file
6	gdb ./file

99. Reference Card: Common Failures

099

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	Segfault: bad pointer or corrupted stack.
2	No output: wrong fd, buffer, length, or syscall number.
3	Extra output: length too large.
4	Wrong loop: flags overwritten before jump.
5	Wrong after call: caller-saved register not preserved.

100. Final Map: From Confusion to Control

100

Compress the lesson into a quick practical card.

Use this page as a fast reminder while coding. The full explanations earlier in the book show why each rule matters.

Rule #	Statement
1	Registers hold temporary machine state.
2	Memory holds bytes addressed by pointers.
3	The stack stores return addresses, saved registers, and locals.
4	Syscalls cross into the Linux kernel.
5	The ABI is the contract that lets code units cooperate.
6	Macros shape source code; subroutines shape runtime behavior.

Appendix A. Complete hello.asm 101

Study a complete assemble-ready example or project skeleton.

This appendix page is designed for typing, building, stepping in GDB, and modifying. Keep the register contracts visible while experimenting.

Appendix A. Complete hello.asm

```
global _start

section .data
    msg db "Hello from Linux x86-64 NASM", 10
    len equ $ - msg

section .text
_start:
    mov eax, 1
    mov edi, 1
    lea rsi, [rel msg]
    mov edx, len
    syscall

    mov eax, 60
    xor edi, edi
    syscall
```

Appendix B. Complete strlen_z + print_z

102

Study a complete assemble-ready example or project skeleton.

This appendix page is designed for typing, building, stepping in GDB, and modifying. Keep the register contracts visible while experimenting.

Appendix B. Complete strlen_z + print_z

```
global _start

section .data
    msg db "Subroutines make assembly readable", 10, 0

section .text
_start:
    lea rdi, [rel msg]
    call print_z
    mov eax, 60
    xor edi, edi
    syscall

strlen_z:
    xor eax, eax
.loop:
    cmp byte [rdi + rax], 0
    je .done
    inc rax
    jmp .loop
.done:
    ret

print_buf:
    mov rdx, rsi
    mov rsi, rdi
    mov edi, 1
    mov eax, 1
    syscall
    ret

print_z:
    push rdi
    call strlen_z
    mov rsi, rax
    pop rdi
    call print_buf
    ret
```

Appendix C. Complete echo.asm 103

Study a complete assemble-ready example or project skeleton.

This appendix page is designed for typing, building, stepping in GDB, and modifying. Keep the register contracts visible while experimenting.

Appendix C. Complete echo.asm

```
global _start
%define SYS_read  0
%define SYS_write 1
%define SYS_exit  60
%define STDIN     0
%define STDOUT    1
%define BUFSIZE   4096

section .bss
    buf resb BUFSIZE

section .text
_start:
.read_loop:
    mov eax, SYS_read
    mov edi, STDIN
    lea rsi, [rel buf]
    mov edx, BUFSIZE
    syscall
    test rax, rax
    jle .done

    mov rdx, rax
    mov eax, SYS_write
    mov edi, STDOUT
    lea rsi, [rel buf]
    syscall
    jmp .read_loop

.done:
    mov eax, SYS_exit
    xor edi, edi
    syscall
```

Appendix D. Complete parse and print exercise skeleton

Study a complete assemble-ready example or project skeleton.

This appendix page is designed for typing, building, stepping in GDB, and modifying. Keep the register contracts visible while experimenting.

Appendix D. Complete parse and print exercise skeleton

```
; Exercise: connect parse_u64 and print_u64.
; Goal: read a line, parse the leading unsigned integer,
;       print the value again, then print a newline.
;
; Required routines:
;   read_line   -> rax = bytes read
;   parse_u64   -> rax = value
;   print_u64   -> writes decimal representation
;   print_nl    -> writes newline
;
; Key register plan:
;   rdi = pointer argument for parse_u64 / print_u64 value
;   r12 = saved parsed number if later calls clobber rax
;   rax = return register for routines and syscalls
```

Appendix E. Minimal C interop files

105

Study a complete assemble-ready example or project skeleton.

This appendix page is designed for typing, building, stepping in GDB, and modifying. Keep the register contracts visible while experimenting.

Appendix E. Minimal C interop files

```
; add3.asm
global add3
section .text
add3:
    lea rax, [rdi + rsi]
    add rax, rdx
    ret

; main.c
; #include <stdio.h>
; long add3(long, long, long);
; int main(void) {
;     printf("%ld\n", add3(10, 20, 30));
; }
;
; build:
; nasm -f elf64 add3.asm -o add3.o
; gcc main.c add3.o -o test_add3
```

Selected technical references

NASM Manual, Chapter 9: Output Formats
NASM documents the -f output format option and ELF64 object-file support for Linux/System V-like systems.

NASM Manual, Chapter 5: The NASM Preprocessor
NASM documents %define, %macro, macro expansion order, and macro-local features used in this guide.

Linux man-pages, syscall(2)
The syscall interface and architecture-specific register tables are summarized in the Linux manual pages.

System V AMD64 ABI Draft 0.99.6
The user-space calling convention, stack frame, register preservation, and Linux kernel convention differences are specified there.

Fedora Packages: nasm
Fedora packages NASM as a portable x86 assembler using Intel-like syntax.