

x86-64 Assembly

NASM and GAS

A practical study guide: both syntaxes side by side, the System V ABI, system calls, the stack, diagrams and fully worked examples



Compiled by HARDIK

LinkedIn: www.linkedin.com/in/hardikhardik

Built from personal study notes and extended with diagrams, explanations and tested code examples.

Contents

1 Getting Started	5
1.1 Two assemblers, one processor	5
1.2 Assembling and running	5
1.3 The golden rules	5
1.4 NASM and GAS side by side	6
1.5 Your first program	6
2 Registers and Flags	8
2.1 The register file	8
2.2 Smaller names for the same register	9
2.3 The instruction pointer and the flags	9
3 The System V ABI	11
3.1 What is System V?	11
3.2 What an ABI is	11
3.3 Function-call convention	11
3.4 Who saves what	12
3.5 Prologue and epilogue	12
3.6 Stack alignment	12
4 System Calls	14
4.1 The syscall convention	14
4.2 Common system calls	14
4.3 Flags and file descriptors	15
4.4 Example: Hello, x86-64!	15
4.5 Example: using the C library from main	16
4.6 Example: reading and writing a file	16
5 Sections, Data and Variables	19
5.1 Defining sections	19
5.2 Data types and declarations	19
5.3 RIP-relative addressing	19
5.4 Writing to and reading from globals	20
5.5 Local variables	20
5.6 Constants	21
5.7 Uninitialised variables (.bss)	21
5.8 Alignment	24
6 Signed and Unsigned Comparison	25
6.1 Conditional jumps	25
6.2 The set instructions	25
6.3 Multiplication and division	27
6.4 Zero extension and sign extension	29
6.5 Special extensions for division	30

6.6 String extension: the lods instructions	32
7 Arrays and Scaling	34
7.1 Defining an array	34
7.2 Accessing elements	34
7.3 Complete programs	35
8 Addresses and Pointers	37
8.1 The lea instruction	37
8.2 rbp or rsp?	37
8.3 Using pointers	37
9 Structures, Unions and Bitfields	39
9.1 Structures	39
9.2 Unions	39
9.3 Bitfields	40
9.4 Complete programs	41
10 Control-Flow Patterns	43
10.1 Compare, test and jump	43
10.2 if / else	43
10.3 while loop	44
10.4 for loop	44
10.5 do-while loop	44
10.6 Putting them together	44
10.7 Conditional move	46
11 Functions and the Stack Frame	48
11.1 call and ret	48
11.2 The stack frame	48
11.3 Local variables	49
11.4 More than six arguments	50
11.5 Calling C library functions	51
11.6 Saving callee-saved registers	52
12 Worked Examples	53
12.1 Factorial (recursion, NASM)	53
12.2 Greatest common divisor (GAS)	54
12.3 String length (NASM)	54
12.4 Fibonacci (GAS)	55
12.5 Counting set bits (NASM)	55
12.6 Maximum of an array (GAS)	56
13 Quick Reference and Practice	57
13.1 Instruction cheat sheet	57
13.2 Function checklist	57
13.3 Practice exercises	57
13.4 Where to go next	58

Preface

This book started life as a plain text file of study notes on **x86-64 assembly** written for two assemblers: **NASM** (Intel syntax) and **GAS**, the GNU assembler (AT&T syntax). It has been reorganised into chapters, illustrated with diagrams, and extended with extra explanations and complete example programs. Almost every topic is shown in both syntaxes, side by side, so you can learn one and read the other.

Four kinds of coloured boxes appear throughout:

NOTE

NOTE marks an important point taken from the study notes.

TIP

TIP marks a habit or shortcut that makes your code cleaner or safer.

WATCH OUT

WATCH OUT marks a common mistake or pitfall.

EXTRA

EXTRA marks material added to the notes: new explanations, patterns and examples.

Every complete program in this book was assembled and executed on an x86-64 Linux machine, and the results quoted in the text are the ones actually produced. Most programs finish by passing their answer to the **exit** system call, so you can read it with **echo \$?**. The programs use the Linux x86-64 conventions (System V ABI and Linux system call numbers); they will not run unchanged on Windows.

1 Getting Started

1.1 Two assemblers, one processor

x86-64 is a **CISC** architecture: instructions have different lengths, and many of them can read or write memory directly (unlike ARM, where only load and store instructions touch memory). There are sixteen 64-bit general-purpose registers, a flags register, and an instruction pointer, **rip**.

The same machine code can be written in two different notations:

- **NASM** (Netwide Assembler) uses **Intel syntax**: `mov rax, rbx`.
- **GAS** (GNU Assembler, **as**, also used by **gcc**) uses **AT&T syntax** by default: `movq %rbx, %rax`.

Both produce identical programs. The next section shows how to build each of them.

1.2 Assembling and running

Assembler	Commands
NASM	<pre>nasm -f elf64 hello.asm ld hello.o -o hello ./hello</pre>
GAS (with gcc)	<pre>gcc hello.s -o hello ./hello</pre>
GAS (without gcc)	<pre>as hello.s -o hello.o ld hello.o -o hello ./hello</pre>
NASM + gcc	<pre>nasm -f elf64 hello.asm gcc hello.o -o hello ./hello</pre>

NOTE

Whether your entry point is `_start` or `main` depends on how you link. With plain `ld` there is no C runtime, so execution begins at `_start` and you must exit with a system call yourself. With `gcc` the C runtime starts first and calls `main`; you then return normally and may call C library functions such as `printf`.

EXTRA

On Ubuntu or Debian install the tools with `sudo apt install nasm gcc binutils`. Add `-g` (NASM: `-g -F dwarf`) if you want to debug with `gdb`.

1.3 The golden rules

These three rules from the notes explain most of the differences you will meet:

- **NASM**: the instruction stays the same and the **register name** changes with the data size: `al`, `ax`, `eax`, `rax`.
- **GAS**: the register stays the same and the **instruction gets a suffix** for the data size: `movb`, `movw`, `movl`, `movq` (also `addl`, `subq`, `pushq`, `popq` and so on).
- **Use position-independent code**. Modern executables (PIE) may be loaded at a different address every run, so address data **relative to rip** (Chapter 5).

Size	Bytes	NASM register	NASM keyword	GAS suffix
byte	1	al	byte	b
word	2	ax	word	w
doubleword	4	eax	dword	l (long)
quadword	8	rax	qword	q

1.4 NASM and GAS side by side

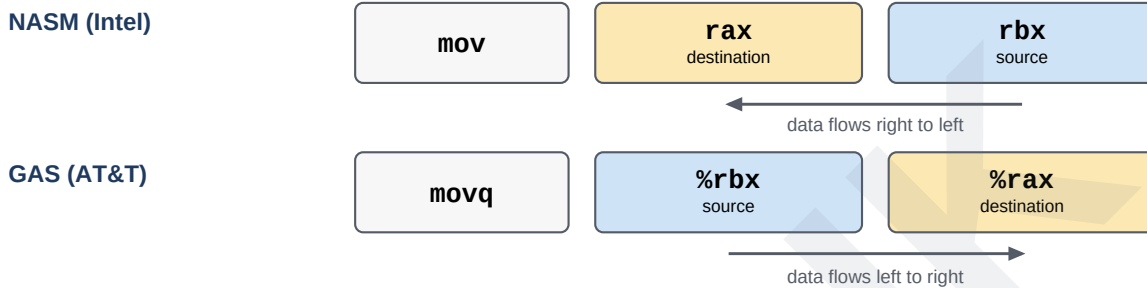


Figure 1.1: Operand order is reversed between the two syntaxes.

Feature	NASM (Intel)	GAS (AT&T)
Operand order	destination, source	source, destination
Registers	rax	%rax
Immediate values	mov rax, 5	movq \$5, %rax
Memory operand	[rbx + rcx*4 + 8]	8(%rbx,%rcx,4)
Size	keyword: mov dword [rbx], 1	suffix: movl \$1, (%rbx)
Relative data access	[rel var]	var(%rip)
Comment	; comment	# comment
Section	section .data	.section .data
Global symbol	global _start	.globl _start
Label	name:	name:

WATCH OUT

Because the operand order is reversed, a comparison reads differently too. NASM `cmp rax, rbx` compares rax with rbx; GAS writes the same thing as `cmpq %rbx, %rax`. Both set the flags from `rax - rbx`.

1.5 Your first program

Exit with status 42. The system call number goes in `rax`, the argument in `rdi`:

NASM

```
section .text
global _start
_start:
    mov rax, 60          ; syscall number 60 = exit
    mov rdi, 42          ; exit status
    syscall
```

GAS

```
.section .text
.globl _start
_start:
    movq $60, %rax       # syscall number 60 = exit
    movq $42, %rdi       # exit status
    syscall
```

2 Registers and Flags

2.1 The register file

x86-64 general-purpose registers (System V roles)

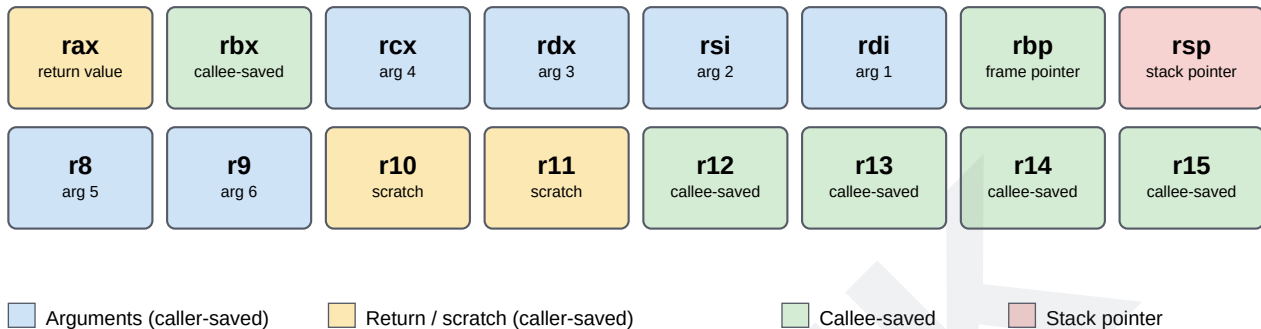
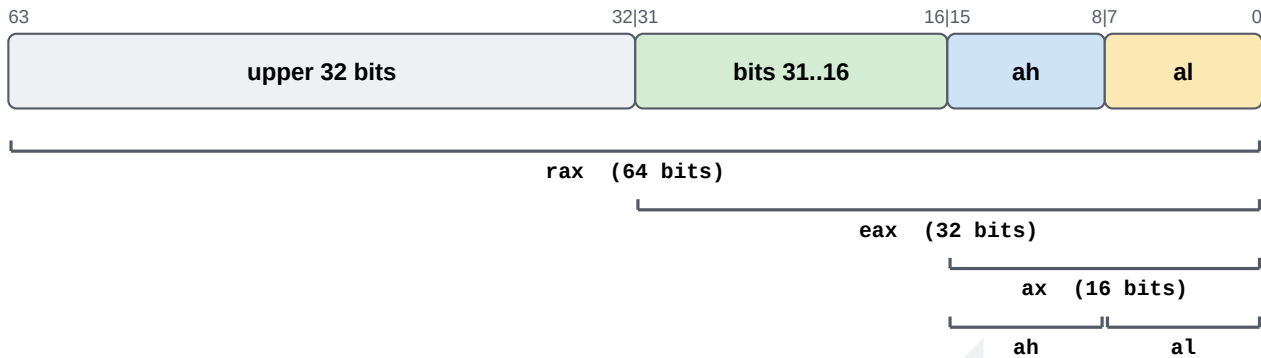


Figure 2.1: The sixteen 64-bit general-purpose registers and their roles in the System V ABI.

Register	Typical use	Preserved across calls?
rax	Return value; system call number; multiply and divide	No (scratch)
rbx	General purpose	Yes (callee-saved)
rcx	Argument 4; shift counts; loop counts	No
rdx	Argument 3; high half of multiply and divide	No
rsi, rdi	Arguments 2 and 1; source and destination for string instructions	No
rbp	Frame pointer	Yes
rsp	Stack pointer	Yes (must be restored)
r8, r9	Arguments 5 and 6	No
r10, r11	Scratch (r10 carries argument 4 of a system call)	No
r12 - r15	General purpose	Yes (callee-saved)

2.2 Smaller names for the same register

The 64-bit register `rax` and the smaller names for its parts



Writing `eax` clears bits 63..32. Writing `ax`, `ah` or `al` leaves every other bit unchanged.

Figure 2.2: One physical register, several names.

64-bit	32-bit	16-bit	8-bit (low)	8-bit (high)
<code>rax</code>	<code>eax</code>	<code>ax</code>	<code>al</code>	<code>ah</code>
<code>rbx</code>	<code>ebx</code>	<code>bx</code>	<code>bl</code>	<code>bh</code>
<code>rcx</code>	<code>ecx</code>	<code>cx</code>	<code>cl</code>	<code>ch</code>
<code>rdx</code>	<code>edx</code>	<code>dx</code>	<code>dl</code>	<code>dh</code>
<code>rsi</code>	<code>esi</code>	<code>si</code>	<code>sil</code>	-
<code>rdi</code>	<code>edi</code>	<code>di</code>	<code>dil</code>	-
<code>rbp</code>	<code>ebp</code>	<code>bp</code>	<code>bpl</code>	-
<code>rsp</code>	<code>esp</code>	<code>sp</code>	<code>spl</code>	-
<code>r8 - r15</code>	<code>r8d - r15d</code>	<code>r8w - r15w</code>	<code>r8b - r15b</code>	-

WATCH OUT

A write to a 32-bit register clears the upper 32 bits of the 64-bit register. `mov eax, 5` leaves `rax = 5`. A write to a 16-bit or 8-bit register does **not** clear anything: `mov al, 5` changes only the lowest byte. This is why `xor eax, eax` is the standard way to zero all of `rax`.

TIP

In NASM you pick the register that matches the data size (`al`, `ax`, `eax`, `rax`). In GAS the register is the same and the instruction suffix carries the size (`movb`, `movw`, `movl`, `movq`).

2.3 The instruction pointer and the flags

`rip` holds the address of the next instruction. You never write to it directly; `jmp`, `call` and `ret` change it. The **RFLAGS** register records what the last arithmetic or compare instruction did:

Four RFLAGS bits drive almost every conditional jump



cmp, test, add, sub, and, or, inc, dec ... update them; mov, lea, push and pop do not.

Figure 2.3: The four flags used by conditional jumps.

Flag	Name	Set when
CF	Carry	an addition carries out, or a subtraction borrows (unsigned overflow)
ZF	Zero	the result is zero
SF	Sign	the result's top bit is 1 (negative if you treat it as signed)
OF	Overflow	the signed result does not fit in the register

3 The System V ABI

3.1 What is System V?

System V (pronounced "system five") is a UNIX operating system originally developed by AT&T Bell Labs in the early 1980s. Its specifications describe how programs run on UNIX-like systems, including:

- program execution,
- function calls,
- process and memory management,
- and the binary interface between programs and the operating system.

Today the **System V AMD64 ABI** is the standard used by Linux, the BSDs and macOS on 64-bit x86. Windows uses a different convention, so code in this book is for Linux and similar systems.

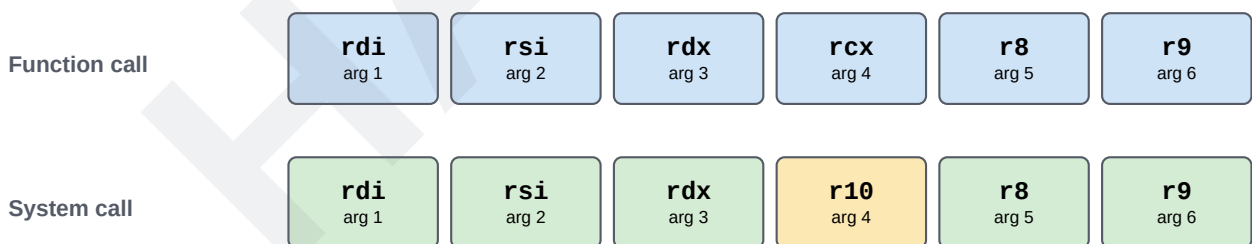
3.2 What an ABI is

ABI stands for **Application Binary Interface**. It is the contract that lets separately built pieces of code work together: the compiler puts values in particular registers and the code it calls expects to find them there. If you write assembly you must follow the same contract or your program will break. The ABI says:

- which registers hold arguments,
- who saves which registers (caller or callee),
- how the stack is aligned,
- how return values are passed.

3.3 Function-call convention

Where do the first six arguments go?



The 4th argument differs: the syscall instruction overwrites rcx, so the kernel uses r10.

Function call: return value in rax. System call: number in rax, result in rax.

Figure 3.1: Function calls and system calls use almost the same registers.

Argument	Register
arg 1	rdi
arg 2	rsi
arg 3	rdx
arg 4	rcx

Argument	Register
arg 5	r8
arg 6	r9
arg 7 and later	on the stack
return value	rax (rdx:rax for a 128-bit result)

NOTE

The seventh and later arguments are pushed on the stack. Floating-point arguments use `xmm0` to `xmm7` instead and are not covered here.

NOTE

A structure larger than 16 bytes is not returned in registers. The caller reserves memory for it, passes its address as a hidden first argument in `rdi`, and the function returns that address in `rax`.

3.4 Who saves what

Kind	Registers	Rule
Caller-saved (scratch)	rax, rcx, rdx, rsi, rdi, r8, r9, r10, r11	A function may destroy them. If you need one after a call, save it yourself first.
Callee-saved	rbx, rbp, r12, r13, r14, r15 (and rsp)	A function that uses one must save it on entry and restore it before returning.

3.5 Prologue and epilogue

Functions that need a stack frame start with a **prologue** and finish with an **epilogue**:

```

; prologue
push rbp
mov rbp, rsp
sub rsp, 16

; epilogue
add rsp, 16      ; release the locals
pop rbp
ret

```

TIP

`leave` does the work of `mov rsp, rbp` followed by `pop rbp`, so the epilogue is often just `leave` then `ret`. Chapter 11 shows complete examples.

3.6 Stack alignment

The stack pointer must be a **multiple of 16** at the moment of a `call` instruction. This matters because many C library functions (`printf`, `malloc`, `scanf` and others) use SSE instructions that fault on misaligned stack data.

- When the operating system starts your program at `_start`, `rsp` is already 16-byte aligned.
- Inside a function you were `called` into, `rsp` is **8 bytes off** (the return address was pushed). A single `push rbp` fixes it, which is one more reason for the standard prologue.
- If you push or allocate an odd number of 8-byte slots before calling another function, add 8 bytes of padding.

EXTRA

System V also defines a 128-byte **red zone** below `rsp` that leaf functions may use without adjusting `rsp`. Signal handlers and the kernel respect it. You will not need it in this book.

HARDIK

4 System Calls

4.1 The syscall convention

A system call asks the Linux kernel to do something for your program: write text, open a file, exit. You put the **system call number in `rax`**, the arguments in registers, and execute the `syscall` instruction. The result comes back in `rax`.

Item	Register
syscall number	<code>rax</code>
arg 1	<code>rdi</code>
arg 2	<code>rsi</code>
arg 3	<code>rdx</code>
arg 4	<code>r10</code>
arg 5	<code>r8</code>
arg 6	<code>r9</code>
return value	<code>rax</code>

WATCH OUT

Argument 4 is `r10`, not `rcx` as in a normal function call. The `syscall` instruction itself destroys `rcx` and `r11`, so do not keep anything in them across a syscall.

NOTE

Use the `syscall` instruction to make the call.

4.2 Common system calls

Name	Number	Description
read	0	read from a file descriptor
write	1	write to a file descriptor
open	2	open a file
close	3	close a file descriptor
lseek	8	move the file pointer
mmap	9	get a pointer to virtual memory
munmap	11	release virtual memory
brk	12	get or move the heap base address
exit	60	exit from the program

4.3 Flags and file descriptors

Open flag	Value
O_RDONLY	0
O_WRONLY	1
O_RDWR	2
O_CREAT	0x40
O_TRUNC	0x200

Flags are combined with OR. To create a file for writing and empty it, use `O_WRONLY | O_CREAT | O_TRUNC = 1 + 0x40 + 0x200 = 0x241`.

Descriptor	Meaning
0	STDIN, standard input
1	STDOUT, standard output
2	STDERR, standard error

EXTRA

Descriptor 3 is not a special stream. It is simply the first number `open` hands out, followed by 4, 5 and so on. Always use the value `open` returns in `rax`. A negative result (from -1 down to -4095) means an error and holds the negated error number.

4.4 Example: Hello, x86-64!

NASM

```
section .data
msg db "Hello, x86-64!", 10
len equ $ - msg           ; length computed by the assembler

section .text
global _start
_start:
    mov rax, 1             ; write
    mov rdi, 1             ; fd 1 = stdout
    lea rsi, [rel msg]     ; buffer (relative address)
    mov rdx, len           ; byte count
    syscall

    mov rax, 60            ; exit
    xor edi, edi           ; status 0
    syscall
```

GAS

```
.section .data
msg:      .ascii "Hello, x86-64!\n"
.set len, . - msg          # length computed by the assembler

.section .text
.globl _start
_start:
    movq $1, %rax           # write
    movq $1, %rdi           # fd 1 = stdout
    leaq msg(%rip), %rsi    # buffer (relative address)
    movq $len, %rdx         # byte count
    syscall

    movq $60, %rax          # exit
    xorl %edi, %edi         # status 0
    syscall
```

`len equ $ - msg` (NASM) and `.set len, . - msg` (GAS) subtract the address of `msg` from the current location, so the byte count is always right even if you edit the string. Note `lea rsi, [rel msg]`: the address is computed relative to `rip`.

4.5 Example: using the C library from main

If you link with `gcc`, you can call C functions. Here `puts` prints a string and appends a newline. Notice that `main` starts with the usual `push rbp`, which also brings the stack back to a 16-byte boundary.

```
.section .rodata
msg:      .asciz "Hello from main"

.section .text
.globl main
main:
    pushq %rbp
    movq %rsp, %rbp        # rsp is now 16-byte aligned
    leaq msg(%rip), %rdi    # argument 1 = string
    call puts@PLT          # libc prints it plus a newline
    xorl %eax, %eax         # return 0
    popq %rbp
    ret

.section .note.GNU-stack,"",@progbits
```

EXTRA

The final `.note.GNU-stack` line tells the linker the program does not need an executable stack. Without it, newer versions of `ld` print a warning. The NASM equivalent is `section .note.GNU-stack noalloc noexec nowrite progbits`.

4.6 Example: reading and writing a file

This NASM program creates a file, writes a line into it, closes it, reopens it, reads the line back and prints it to the screen. It keeps the descriptor in `r12`, a callee-saved register, so the value survives the system calls.

```

section .data
path    db "/tmp/hardik_demo.txt", 0
text    db "Hello file", 10
tlen    equ $ - text

section .bss
buf      resb 64

section .text
global _start
_start:
; ---- fd = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0644) ----
mov rax, 2
lea rdi, [rel path]
mov rsi, 0x241          ; O_WRONLY (1) | O_CREAT (0x40) | O_TRUNC (0x200)
mov rdx, 0x0644         ; permissions rw-r--r--
syscall
mov r12, rax            ; save the descriptor in a callee-saved register

; ---- write(fd, text, tlen) ----
mov rax, 1
mov rdi, r12
lea rsi, [rel text]
mov rdx, tlen
syscall

; ---- close(fd) ----
mov rax, 3
mov rdi, r12
syscall

; ---- fd = open(path, O_RDONLY) ----
mov rax, 2
lea rdi, [rel path]
xor esi, esi            ; O_RDONLY = 0
xor edx, edx
syscall
mov r12, rax

; ---- n = read(fd, buf, 64) ----
xor eax, eax            ; read = 0
mov rdi, r12
lea rsi, [rel buf]
mov rdx, 64
syscall
mov r13, rax            ; number of bytes actually read

; ---- write(1, buf, n) to the screen ----
mov rax, 1
mov rdi, 1
lea rsi, [rel buf]
mov rdx, r13
syscall

; ---- close(fd) and exit(0) ----
mov rax, 3
mov rdi, r12
syscall
mov rax, 60
xor edi, edi
syscall

```

TIP

Translate it to GAS as an exercise: reverse the operand order, add `%` and `$`, and replace `lea rsi, [rel path]` with `leaq path(%rip), %rsi`.

HARDIK

5 Sections, Data and Variables

5.1 Defining sections

Purpose	NASM	GAS
Code	section .text	.section .text
Initialised, writable data	section .data	.section .data
Read-only constants	section .rodata	.section .rodata
Uninitialised data (zeroed)	section .bss	.section .bss

5.2 Data types and declarations

C type	Size	NASM	GAS
char	1 byte	db	.byte
short	2 bytes	dw	.word
int	4 bytes	dd	.long
long, pointer	8 bytes	dq	.quad

NOTE

Always use **8 bytes for a pointer** in a 64-bit (x86-64) program, and 4 bytes in a 32-bit (x86-32) program.

WATCH OUT

In GAS, **.word** means **2 bytes** and **.long** means **4 bytes**. This is the opposite of ARM, where **.word** is 4 bytes. NASM's **dw** / **dd** follow the same sizes as GAS's **.word** / **.long**.

Declaring global variables

<pre>; NASM section .data c db 1 s dw 2 i dd 3 l dq 4</pre>	<pre># GAS .section .data c: .byte 1 s: .word 2 i: .long 3 l: .quad 4</pre>
---	---

5.3 RIP-relative addressing

Access every variable **relative to rip**, in both assemblers. The operating system may place the executable at a different address each time it runs (ASLR and PIE). The distance between an instruction and the variable it uses does not change, so a rip-relative reference always works, while an absolute address would break.

RIP-relative addressing: the distance stays the same wherever the program is loaded

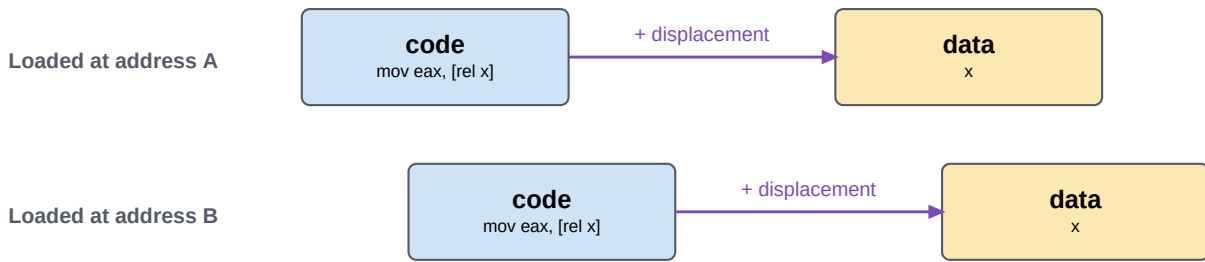


Figure 5.1: The displacement between code and data is fixed, so rip-relative access survives relocation.

	NASM	GAS
Relative reference	[rel var]	var(%rip)
Make it the default	default rel (at the top of the file)	always write (%rip)

5.4 Writing to and reading from globals

NASM

```
; write                                ; read (register size matches the data)
mov byte [rel c], 10                   mov al, [rel c]
mov word [rel s], 20                   mov ax, [rel s]
mov dword [rel i], 30                  mov eax, [rel i]
mov qword [rel l], 40                  mov rax, [rel l]
```

GAS

```
# write                                # read (suffix matches the data)
movb $10, c(%rip)                      movb c(%rip), %al
movw $20, s(%rip)                      movw s(%rip), %ax
movl $30, i(%rip)                      movl i(%rip), %eax
movq $40, l(%rip)                      movq l(%rip), %rax
```

TIP

When you store an immediate into memory, NASM needs a size keyword (**byte**, **word**, **dword**, **qword**) because the assembler cannot guess how wide the memory operand is. When a register is involved the register already tells the size, so no keyword is needed.

5.5 Local variables

A local variable lives on the stack. Allocate space by subtracting from **rsp**, then address it relative to **rbp** (fixed) or **rsp** (moves when you push):

```
sub rsp, 16          ; NASM: room for locals    subq $16, %rsp      # GAS
mov dword [rbp-4], 7 ; a local int              movl $7, -4(%rbp)
```

WATCH OUT

Allocating exactly 4 bytes (`sub rsp, 4`) works for a single variable, but it breaks the 16-byte alignment rule as soon as you call another function. Round the allocation up to a multiple of 16.

5.6 Constants

	NASM	GAS
Define	<code>EMP_ID equ 0</code> <code>EMP_DAY equ 4</code>	<code>.set EMP_ID, 0</code> <code>.set EMP_DAY, 4</code>
Use	<code>cmp rcx, EMP_ID</code>	<code>cmpq \$EMP_ID, %rcx</code>

WATCH OUT

In GAS, `.set` needs a **comma**: `.set EMP_ID, 0`. Without it the assembler stops with "expected comma". Constants are not variables: they take no memory and are replaced by their value at assembly time. In GAS, put a `$` in front when you use one as an immediate.

5.7 Uninitialised variables (.bss)

Variables without a starting value go in `.bss`. They take no space in the executable file; the loader reserves the memory and fills it with zeros.

NASM

Directive	Purpose	Size per unit
<code>resb</code>	Reserve bytes	1 byte
<code>resw</code>	Reserve words	2 bytes
<code>resd</code>	Reserve double words	4 bytes
<code>resq</code>	Reserve quad words	8 bytes

```
section .bss
var1    resb 1      ; 1 byte
var4    resq 1      ; 8 bytes (64-bit)
buffer  resb 64     ; 64-byte buffer
```

```
mov rax, [rel var4]
mov byte [rel var1], 5
lea rdi, [rel buffer]
```

GAS

Directive	Meaning	Example
<code>.skip N</code>	Reserve N bytes	<code>.skip 64</code>
<code>.zero N</code>	Reserve N zeroed bytes	<code>.zero 32</code>
<code>.lcomm sym, size</code>	Local uninitialised variable	<code>.lcomm buf, 64</code>
<code>.comm sym, size, align</code>	Global uninitialised variable	<code>.comm buf, 64, 8</code>

```
.section .bss
.align 8          # align to 8 bytes
var4:  .skip 8     # 8 bytes

movq var4(%rip), %rax
movq $123, var4(%rip)
```

WATCH OUT

On x86-64 Linux the GNU assembler accepts `.lcomm` with only a name and a size. A third (alignment) argument is accepted by `.comm` but not by `.lcomm`.

Complete programs

The programs below write and read every data size, use `.bss` variables, and exit with the total 112 (100 from the four globals, plus 5 and 7 from the `.bss` bytes).

NASM

```
section .data
c    db 1
s    dw 2
i    dd 3
l    dq 4

section .bss
var1  resb 1
var4  resq 1
buffer resb 64

section .text
global _start
_start:
; ---- write to global variables ----
mov byte [rel c], 10
mov word [rel s], 20
mov dword [rel i], 30
mov qword [rel l], 40

; ---- read them back (register size matches data size) ----
movzx eax, byte [rel c]      ; eax = 10
movzx ebx, word [rel s]      ; ebx = 20
mov  ecx, [rel i]            ; ecx = 30
mov  rdx, [rel l]            ; rdx = 40
add  rax, rbx
add  rax, rcx
add  rax, rdx                ; rax = 100

; ---- uninitialised (.bss) variables ----
mov byte [rel var1], 5
mov qword [rel var4], 123
lea rdi, [rel buffer]        ; rdi = &buffer
mov byte [rdi], 7
add al, [rel var1]           ; +5
add al, [rdi]                ; +7 -> 112
cmp qword [rel var4], 123
jne .bad

movzx edi, al                ; exit status = 112
mov eax, 60
syscall

.bad:
mov edi, 1
mov eax, 60
syscall
```

GAS

```
.set EMP_ID, 0                # constants
.set EMP_DAY, 4

.section .data
c: .byte 1
s: .word 2
i: .long 3
l: .quad 4

.section .bss
.align 8
var4: .skip 8
var1: .skip 1
buffer: .zero 64

.section .text
.globl _start
_start:
    # ---- write to global variables ----
    movb $10, c(%rip)
    movw $20, s(%rip)
    movl $30, i(%rip)
    movq $40, l(%rip)

    # ---- read them back (suffix matches data size) ----
    movzbl c(%rip), %eax        # eax = 10
    movzwl s(%rip), %ebx        # ebx = 20
    movl i(%rip), %ecx          # ecx = 30
    movq l(%rip), %rdx          # rdx = 40
    addq %rbx, %rax
    addq %rcx, %rax
    addq %rdx, %rax             # rax = 100

    # ---- uninitialised (.bss) variables ----
    movb $5, var1(%rip)
    movq $123, var4(%rip)
    leaq buffer(%rip), %rdi
    movb $7, (%rdi)
    addb var1(%rip), %al         # +5
    addb (%rdi), %al            # +7 -> 112
    cmpq $123, var4(%rip)
    jne bad

    movzbl %al, %edi            # exit status = 112
    movl $60, %eax
    syscall

bad:
    movl $1, %edi
    movl $60, %eax
    syscall
```

5.8 Alignment

Naturally aligned data is faster, and some SSE instructions require it. Use `align 8` (NASM) or `.align 8` (GAS) before an 8-byte variable. In GAS on x86-64 Linux, `.align N` means **N bytes**, not 2 to the power N, so `.align 8` is an 8-byte boundary.

6 Signed and Unsigned Comparison

6.1 Conditional jumps

After `cmp a, b` (NASM) or `cmp b, a` (GAS) you choose a jump by whether you treat the numbers as **unsigned** or **signed**. The comparison itself is the same; only the jump differs.

Meaning	Unsigned jump	Signed jump
$a > b$	<code>ja</code> (jump above)	<code>jg</code> (jump greater)
$a \geq b$	<code>jae</code> (above or equal)	<code>jge</code> (greater or equal)
$a < b$	<code>jb</code> (jump below)	<code>jl</code> (jump less)
$a \leq b$	<code>jbe</code> (below or equal)	<code>jle</code> (less or equal)
$a == b$	<code>je</code>	<code>je</code>
$a != b$	<code>jne</code>	<code>jne</code>

Jump	Flags tested
<code>ja / jnbe</code>	$CF = 0$ and $ZF = 0$
<code>jae / jnb / jnc</code>	$CF = 0$
<code>jb / jnae / jc</code>	$CF = 1$
<code>jbe / jna</code>	$CF = 1$ or $ZF = 1$
<code>jg / jnl</code>	$ZF = 0$ and $SF = 0F$
<code>jge / jnl</code>	$SF = 0F$
<code>jl / jnge</code>	$SF != 0F$
<code>jle / jng</code>	$ZF = 1$ or $SF != 0F$
<code>je / jz</code>	$ZF = 1$
<code>jne / jnz</code>	$ZF = 0$
<code>js</code>	$SF = 1$ (result negative)
<code>jns</code>	$SF = 0$ (result not negative)

NOTE

These signed and unsigned rules work the same in GAS and NASM.

EXTRA

`js` and `jns` do not extend or convert anything. They branch on the sign flag after an instruction that set it, for example `test eax, eax` followed by `js negative`.

6.2 The set instructions

`setcc` stores 1 or 0 in an 8-bit register depending on the flags, which turns a comparison into a value.

Meaning	Unsigned	Signed
a > b	seta	setg
a >= b	setae	setge
a < b	setb	setl
a <= b	setbe	setle
a == b	sete	sete
a != b	setne	setne

The same bits give different answers. Here **rax** holds -1 (all ones) and **rbx** holds 1:

NASM

```

section .text
global _start
_start:
    mov rax, -1                ; 0xFFFFFFFFFFFFFFFF
    mov rbx, 1
    cmp rax, rbx               ; flags from rax - rbx
    setl cl                    ; signed: -1 < 1 -> 1
    setb dl                    ; unsigned: 18446744073709551615 < 1 -> 0
    seta r8b                   ; unsigned: above -> 1
    movzx ecx, cl
    movzx edx, dl
    movzx r8d, r8b
    shl ecx, 2                 ; 1 * 4 = 4
    shl edx, 1                 ; 0 * 2 = 0
    lea edi, [rcx + rdx]       ; 4
    add edi, r8d                ; + 1 = 5
    mov eax, 60
    syscall

```

GAS

```

.section .text
.global _start
_start:
    movq $-1, %rax             # 0xFFFFFFFFFFFFFFFF
    movq $1, %rbx
    cmpq %rbx, %rax            # flags from rax - rbx (AT&T: source first!)
    setl %cl                    # signed: -1 < 1 -> 1
    setb %dl                    # unsigned: 18446744073709551615 < 1 -> 0
    seta %r8b                   # unsigned: above -> 1
    movzbl %cl, %ecx
    movzbl %dl, %edx
    movzbl %r8b, %r8d
    shll $2, %ecx               # 1 * 4 = 4
    shll $1, %edx               # 0 * 2 = 0
    leal (%rcx,%rdx), %edi      # 4
    addl %r8d, %edi             # + 1 = 5
    movl $60, %eax
    syscall

```

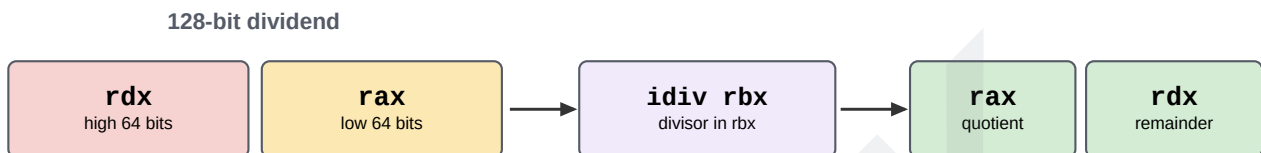
Signed, -1 is less than 1, so **setl** gives 1. Unsigned, the same bits are 18446744073709551615, which is not below 1, so **setb** gives 0 and **seta** gives 1. The program exits with 5 (4 + 0 + 1).

6.3 Multiplication and division

	Unsigned	Signed
Multiply	<code>mul</code>	<code>imul</code>
Divide	<code>div</code>	<code>idiv</code>

Division

`div` and `idiv` divide the **128-bit value in `rdx:rax`** by the operand. The quotient goes to `rax` and the remainder to `rdx`. Because the dividend is 128 bits wide you must set `rdx` first:



Signed: `cqo` copies the sign of `rax` into every bit of `rdx`, then `idiv`

Unsigned: `xor edx, edx` clears `rdx`, then `div`

(for 32-bit division use `eax / edx` with `cdq` or `xor edx, edx`)

Figure 6.1: Division works on the register pair `rdx:rax`.

- **Signed:** `cqo` (NASM) or `cqto` (GAS) copies the sign of `rax` into every bit of `rdx`, then `idiv`.
- **Unsigned:** clear `rdx` with `xor edx, edx`, then `div`.

WATCH OUT

If you forget to set `rdx`, the dividend is garbage. If the quotient does not fit in 64 bits (or you divide by zero) the CPU raises an exception and the program crashes with SIGFPE.

Multiplication

`imul rax, 5` (two-operand form) multiplies and keeps the low 64 bits in the destination. The one-operand `mul rbx` computes `rax * rbx` and puts the full 128-bit product in `rdx:rax`. Also available: the three-operand form `imul r8, rax, 10`, meaning `r8 = rax * 10`.

Both programs below exit with the total 138; the comments show every intermediate value.

NASM

```
section .data
x db 0xF0                                ; 240 unsigned, -16 signed

section .text
global _start
_start:
; ---- signed division: -17 / 5 ----
mov rax, -17
mov rbx, 5
cqo                                     ; sign-extend rax into rdx:rax
idiv rbx                               ; rax = -3 (quotient), rdx = -2 (remainder)
neg rax                                ; 3
neg rdx                                ; 2
imul r8, rax, 10                        ; r8 = 30
add r8, rdx                             ; r8 = 32

; ---- unsigned division: 17 / 5 ----
mov rax, 17
xor edx, edx                            ; upper half of the dividend = 0
mov rbx, 5
div rbx                                 ; rax = 3, rdx = 2
add r8, rax                             ; 35
add r8, rdx                             ; 37

; ---- zero vs sign extension of the same byte ----
movzx eax, byte [rel x]                 ; 240
movsx rcx, byte [rel x]                 ; -16
add rax, rcx                            ; 224
sub rax, 200                             ; 24
add r8, rax                             ; 61

; ---- multiplication ----
mov rax, 7
imul rax, 5                             ; signed: rax = 35
add r8, rax                             ; 96
mov rax, 6
mov rbx, 7
mul rbx                                 ; unsigned: rdx:rax = 42
add r8, rax                             ; 138

mov rdi, r8                             ; exit status = 138
mov rax, 60
syscall
```

GAS

```
.section .data
x: .byte 0xF0                # 240 unsigned, -16 signed

.section .text
.globl _start
_start:
    # ---- signed division: -17 / 5 ----
    movq $-17, %rax
    movq $5, %rbx
    cqto                    # sign-extend rax into rdx:rax
    idivq %rbx              # rax = -3 (quotient), rdx = -2 (remainder)
    negq %rax               # 3
    negq %rdx               # 2
    imulq $10, %rax, %r8    # r8 = 30
    addq %rdx, %r8          # r8 = 32

    # ---- unsigned division: 17 / 5 ----
    movq $17, %rax
    xorl %edx, %edx         # upper half of the dividend = 0
    movq $5, %rbx
    divq %rbx               # rax = 3, rdx = 2
    addq %rax, %r8          # 35
    addq %rdx, %r8          # 37

    # ---- zero vs sign extension of the same byte ----
    movzbl x(%rip), %eax    # 240
    movsbq x(%rip), %rcx    # -16
    addq %rcx, %rax         # 224
    subq $200, %rax         # 24
    addq %rax, %r8          # 61

    # ---- multiplication ----
    movq $7, %rax
    imulq $5, %rax          # signed: rax = 35
    addq %rax, %r8          # 96
    movq $6, %rax
    movq $7, %rbx
    mulq %rbx               # unsigned: rdx:rax = 42
    addq %rax, %r8          # 138

    movq %r8, %rdi          # exit status = 138
    movq $60, %rax
    syscall
```

6.4 Zero extension and sign extension

When you move a small value into a bigger register, the CPU must fill the new upper bits. Two different instructions exist because the right fill depends on whether the value is signed:

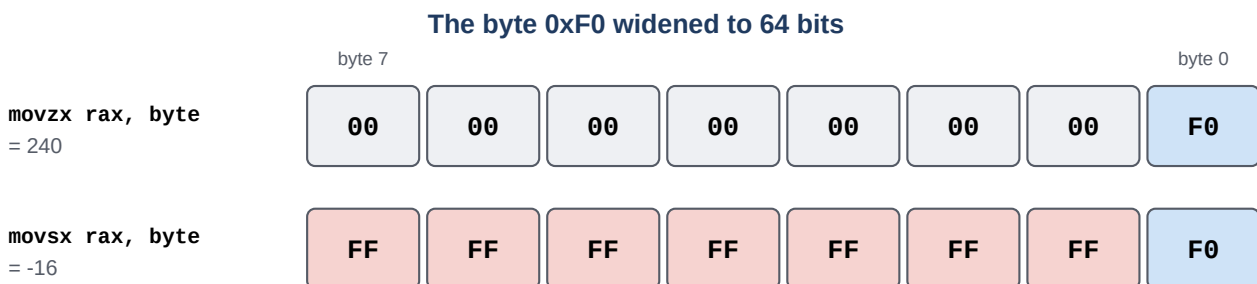


Figure 6.2: The same byte, widened by zero extension and by sign extension.

Zero extension (unsigned values)

Fill the upper bits with zeros. NASM uses one instruction, `movzx`, for every size combination. GAS spells out the sizes in the name:

GAS	Meaning
<code>movzbw</code>	byte to word
<code>movzbl</code>	byte to doubleword
<code>movzbq</code>	byte to quadword
<code>movzwl</code>	word to doubleword
<code>movzwq</code>	word to quadword

```
movzx rax, byte [rel x]      ; NASM
movzbq x(%rip), %rax        # GAS
movzwl (%rdi), %eax         # GAS: load a 16-bit value, widen to 32 bits
```

TIP

There is no `movzx` from 32 to 64 bits because it is not needed: any write to a 32-bit register already clears the upper half. Use a plain `mov eax, [rel i]` (NASM) or `movl i(%rip), %eax` (GAS).

Sign extension (signed values)

Fill the upper bits with copies of the sign bit. NASM uses `movsx` (and `movsxd` for 32 to 64 bits):

GAS	Meaning
<code>movsbw</code>	byte to word
<code>movsbl</code>	byte to doubleword
<code>movsbq</code>	byte to quadword
<code>movswl</code>	word to doubleword
<code>movswq</code>	word to quadword
<code>movslq</code>	doubleword to quadword

```
movsx rax, byte [rel x]      ; NASM
movsxd rax, dword [rel d]    ; NASM: 32 -> 64 bits
movsbq x(%rip), %rax        # GAS
movswl (%rdi), %eax         # GAS: load a 16-bit value, widen to 32 bits
movslq d(%rip), %rax        # GAS: 32 -> 64 bits
```

6.5 Special extensions for division

These instructions have no operands. They always work on the accumulator registers (`al`, `ax`, `eax`, `rax`, `dx`, `edx`, `rdx`), which is exactly what division needs:

NASM	GAS	Meaning
<code>cbw</code>	<code>cbtw</code>	AL to AX (sign-extended)
<code>cwde</code>	<code>cwtl</code>	AX to EAX
<code>cdqe</code>	<code>cltq</code>	EAX to RAX

NASM	GAS	Meaning
cwd	cwtd	AX to DX:AX
cdq	cltd	EAX to EDX:EAX
cqo	cqto	RAX to RDX:RAX

NOTE

These are used before signed division (**idiv**). The GNU assembler also accepts the Intel names (**cdqe**, **cqo** and so on) in AT&T mode.

The **cbw** / **cwde** / **cdqe** chain widens one value step by step, and writing **eax** clears the upper half of **rax**:

NASM

```

section .data
d dd -100

section .text
global _start
_start:
; ---- the cbw / cwde / cdqe chain ----
mov al, 0xF0          ; -16 in 8 bits
cbw                  ; ax = 0xFFF0
cwde                 ; eax = 0xFFFFF0
cdqe                 ; rax = -16
neg rax              ; 16
mov r8, rax

; ---- writing a 32-bit register zero-extends into 64 bits ----
mov eax, 0xFFFFFFFF   ; rax = 0x00000000FFFFFFFF
shr rax, 31           ; 1
add r8, rax           ; 17

; ---- 32 -> 64 bit sign extension needs movsxd ----
movsxd rcx, dword [rel d] ; rcx = -100
neg rcx               ; 100
add r8, rcx           ; 117

mov rdi, r8           ; exit status = 117
mov rax, 60
syscall

```

GAS

```
.section .data
d: .long -100

.section .text
.globl _start
_start:
    # ---- the cbtw / cwtl / cltq chain (AT&T names of cbw / cwde / cdqe) ----
    movb $0xF0, %al          # -16 in 8 bits
    cbtw                     # ax = 0xFFF0
    cwtl                     # eax = 0xFFFFF0
    cltq                     # rax = -16
    negq %rax                # 16
    movq %rax, %r8

    # ---- writing a 32-bit register zero-extends into 64 bits ----
    movl $0xFFFFFFFF, %eax   # rax = 0x00000000FFFFFFFF
    shrq $31, %rax           # 1
    addq %rax, %r8           # 17

    # ---- 32 -> 64 bit sign extension: movslq ----
    movslq d(%rip), %rcx     # rcx = -100
    negq %rcx                # 100
    addq %rcx, %r8           # 117

    movq %r8, %rdi           # exit status = 117
    movq $60, %rax
    syscall
```

6.6 String extension: the lods instructions

The string instructions read through `rsi` and then advance it automatically (forward when the direction flag is clear, which is the normal state; `cld` makes sure).

Instruction	Meaning
<code>lodsb</code>	<code>AL <- [RSI]</code> , then <code>RSI = RSI + 1</code>
<code>lodsw</code>	<code>AX <- [RSI]</code> , then <code>RSI = RSI + 2</code>
<code>lods d</code>	<code>EAX <- [RSI]</code> , then <code>RSI = RSI + 4</code>
<code>lods q</code>	<code>RAX <- [RSI]</code> , then <code>RSI = RSI + 8</code>

EXTRA

Companions you will see: `stosb/w/d/q` (store `AL/AX/EAX/RAX` to `[RDI]`), `movsb/w/d/q` (copy `[RSI]` to `[RDI]`) and `scasb/w/d/q` (compare `AL/AX/EAX/RAX` with `[RDI]`). With a `rep` prefix they repeat `rcx` times. GAS writes `lodsb` the same way.

```

section .data
s db 1, 2, 3, 4, 0           ; zero-terminated list of bytes

section .text
global _start
_start:
    cld                     ; direction flag = 0: rsi moves forward
    lea rsi, [rel s]
    xor ebx, ebx             ; total = 0
.next:
    lodsb                   ; al = [rsi], then rsi = rsi + 1
    test al, al
    jz .done
    movzx eax, al
    add ebx, eax
    jmp .next
.done:                       ; ebx = 1 + 2 + 3 + 4 = 10
    mov edi, ebx
    mov eax, 60
    syscall

```

HARDIK

7 Arrays and Scaling

7.1 Defining an array

```
; NASM                                # GAS
arr dd 3, 7, 2, 9, 4                 arr: .long 3, 7, 2, 9, 4
```

7.2 Accessing elements

To address an element you need the **base address**, the **index**, and the **element size**. x86 builds the address in one step: $\text{base} + \text{index} * \text{scale} + \text{displacement}$, where the scale can be 1, 2, 4 or 8.

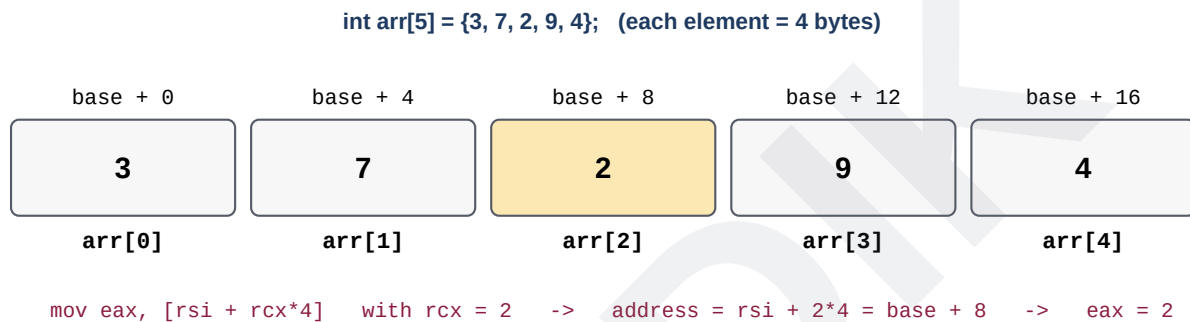


Figure 7.1: Element 2 of an int array lives $2 \times 4 = 8$ bytes after the base address.

Element size	NASM	GAS
8 bytes	<code>mov rdx, [rax + rcx*8]</code>	<code>movq (%rax,%rcx,8), %rdx</code>
4 bytes	<code>mov edx, [rax + rcx*4]</code>	<code>movl (%rax,%rcx,4), %edx</code>
2 bytes	<code>mov dx, [rax + rcx*2]</code>	<code>movw (%rax,%rcx,2), %dx</code>
1 byte	<code>mov dl, [rax + rcx]</code>	<code>movb (%rax,%rcx), %dl</code>

The first step is always to put the array's address in a register with **lea** (Chapter 8):

```
lea rax, [rel arr]           ; NASM: rax = &arr
leaq arr(%rip), %rax         # GAS:  rax = &arr
```

The scale tells the CPU how many bytes each element occupies. To read a particular **byte inside** an element, add a displacement after the scaled index:

```
mov dl, [rax + rcx*4 + 1]    ; NASM: second byte of element rcx
movb 1(%rax,%rcx,4), %dl    # GAS:  second byte of element rcx
```

WATCH OUT

Use the right width. Reading `[rax + rcx*4]` into the 64-bit register `rdx` fetches 8 bytes, which is two `int` elements, not one. For a 4-byte element use the 32-bit register (`edx`) or the `movl` form.

WATCH OUT

No RIP-relative address with an index register. `arr(%rip,%rcx,4)` is rejected by the assembler, and NASM's `[rel arr + rcx*4]` produces a warning and an unusable address. Load the base with `lea` first, then index from that register.

NOTE

x86 does not allow a memory-to-memory operation. `mov [rax], [rbx]` is an error: go through a register.

7.3 Complete programs

Each program sums an `int` array with a loop, then reads one element from a byte, word and quadword array to show scales 1, 2 and 8. The exit status is $25 + 3 + 30 + 3 = 61$.

NASM

```
section .data
arr dd 3, 7, 2, 9, 4      ; int arr[5]
b   db 1, 2, 3           ; char b[3]
w   dw 10, 20, 30        ; short w[3]
q   dq 1, 2, 3           ; long q[3]

section .text
global _start
_start:
    lea rsi, [rel arr]    ; rsi = &arr
    xor ecx, ecx         ; i = 0
    xor eax, eax         ; sum = 0
.loop:
    cmp rcx, 5
    jge .done
    add eax, [rsi + rcx*4] ; sum += arr[i] (scale 4 = 4-byte ints)
    inc rcx
    jmp .loop
.done:
    ; eax = 3+7+2+9+4 = 25
    mov ecx, 2
    lea rsi, [rel b]
    movzx edx, byte [rsi + rcx] ; b[2] = 3 (scale 1)
    add eax, edx
    lea rsi, [rel w]
    movzx edx, word [rsi + rcx*2] ; w[2] = 30 (scale 2)
    add eax, edx
    lea rsi, [rel q]
    mov rdx, [rsi + rcx*8] ; q[2] = 3 (scale 8)
    add eax, rdx ; 25 + 3 + 30 + 3 = 61

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

GAS

```
.section .data
arr:    .long 3, 7, 2, 9, 4    # int arr[5]
b:      .byte 1, 2, 3         # char b[3]
w:      .word 10, 20, 30      # short w[3]
q:      .quad 1, 2, 3         # long q[3]

.section .text
.globl _start
_start:
    leaq arr(%rip), %rsi      # rsi = &arr
    xorl %ecx, %ecx          # i = 0
    xorl %eax, %eax          # sum = 0
loop:
    cmpq $5, %rcx
    jge done
    addl (%rsi,%rcx,4), %eax   # sum += arr[i]    (scale 4 = 4-byte ints)
    incq %rcx
    jmp loop
done:
                                # eax = 3+7+2+9+4 = 25
    movl $2, %ecx
    leaq b(%rip), %rsi
    movzbl (%rsi,%rcx), %edx   # b[2] = 3      (scale 1)
    addl %edx, %eax
    leaq w(%rip), %rsi
    movzwl (%rsi,%rcx,2), %edx # w[2] = 30      (scale 2)
    addl %edx, %eax
    leaq q(%rip), %rsi
    movq (%rsi,%rcx,8), %rdx   # q[2] = 3      (scale 8)
    addl %edx, %eax            # 25 + 3 + 30 + 3 = 61

    movl %eax, %edi
    movl $60, %eax
    syscall
```

8 Addresses and Pointers

8.1 The lea instruction

lea (load effective address) **computes** an address and puts it in a register. It does not read memory. Use it to get a pointer to a variable on the stack or in the data section.

NASM	GAS	Result
lea rax, [rbp]	leaq (%rbp), %rax	rax = rbp
lea rax, [rsp]	leaq (%rsp), %rax	rax = rsp
lea rax, [rbp + 20]	leaq 20(%rbp), %rax	rax = rbp + 20
lea rax, [rsp + 20]	leaq 20(%rsp), %rax	rax = rsp + 20
lea rax, [rel var]	leaq var(%rip), %rax	rax = address of var

NOTE

In NASM the size of **lea** comes from the destination register, so you never write **qword** inside it: **lea rax, [rbp]**, not **lea rax, qword [rbp]**.

EXTRA

Because **lea** just does the address arithmetic, it is also a compact calculator that does not touch the flags: **lea rax, [rdi + rdi*2]** computes $rdi * 3$, and **lea eax, [rdi + rsi]** computes $rdi + rsi$.

8.2 rbp or rsp?

Use rbp (frame pointer) for	Use rsp (stack pointer) for
Local variables	Small leaf functions
Complex functions	Hand-optimised code
Debugging	When you fully control every push and call
Learning and clarity	

rbp stays fixed during the function, so **[rbp - 4]** is always the same variable. **rsp** moves with every **push** and **pop**, so offsets from it keep changing.

8.3 Using pointers

A pointer is just an 8-byte integer holding an address. Read through it with a memory operand: **mov eax, [rdi]** (NASM) or **movl (%rdi), %eax** (GAS) loads the **int** that **rdi** points to. Write through it the same way.

Example: swap(int *a, int *b), NASM

```
section .data
x dd 3
y dd 4

section .text
global _start

; void swap(int *a, int *b)      pointers are 8 bytes: rdi and rsi
swap:
    mov eax, [rdi]              ; tmp = *a
    mov edx, [rsi]              ; *a = *b
    mov [rdi], edx              ; *b = tmp
    mov [rsi], eax
    ret

_start:
    lea rdi, [rel x]            ; &x
    lea rsi, [rel y]            ; &y
    call swap                    ; x = 4, y = 3
    mov eax, [rel x]            ; 40
    imul eax, 10                 ; 40
    add eax, [rel y]            ; 43
    mov edi, eax
    mov eax, 60
    syscall
```

The same in GAS

```
.section .data
x: .long 3
y: .long 4

.section .text
.globl _start

# void swap(int *a, int *b)      pointers are 8 bytes: rdi and rsi
swap:
    movl (%rdi), %eax           # tmp = *a
    movl (%rsi), %edx           # *a = *b
    movl %edx, (%rdi)           # *b = tmp
    movl %eax, (%rsi)
    ret

_start:
    leaq x(%rip), %rdi          # &x
    leaq y(%rip), %rsi          # &y
    call swap                    # x = 4, y = 3
    movl x(%rip), %eax
    imull $10, %eax             # 40
    addl y(%rip), %eax          # 43
    movl %eax, %edi
    movl $60, %eax
    syscall
```

After **swap**, **x** is 4 and **y** is 3, so the program exits with $4 * 10 + 3 = 43$.

9 Structures, Unions and Bitfields

9.1 Structures

In assembly a structure is just a block of memory, and a **member is an offset** from the start of the block. To reach a member, use `rbp` or `rsp` (or any pointer register) plus the member's offset, with a register size that matches the member.

```
struct pair {
    int a;          /* offset 0 */
    int b;          /* offset 4 */
};
struct pair p;
p.a = 3;
p.b = 4;
int sum = p.a + p.b;

; NASM                                # GAS
sub rsp, 16                          subq $16, %rsp
mov dword [rsp], 3                    movl $3, 0(%rsp)
mov dword [rsp + 4], 4                movl $4, 4(%rsp)
mov eax, [rsp]                       movl 0(%rsp), %eax
add eax, [rsp + 4]                    addl 4(%rsp), %eax
; sum
```

EXTRA

Compilers insert **padding** so each member sits at a multiple of its own size. In `struct { char c; int i; }` the `int` starts at offset 4 (not 1) and the whole structure occupies 8 bytes. Always compute offsets from the declared layout, not by adding up member sizes.

9.2 Unions

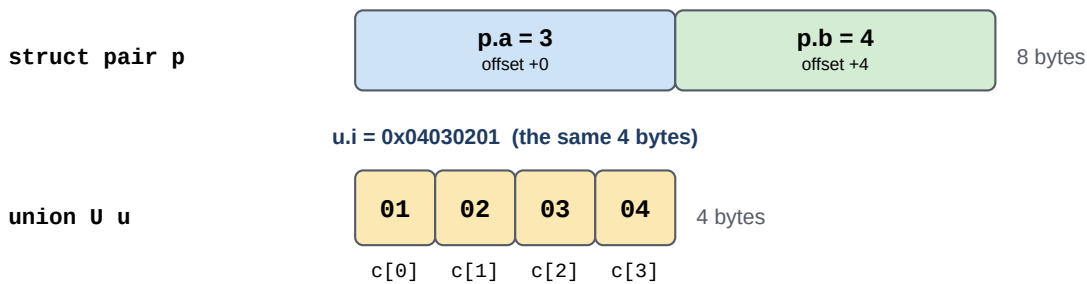
A union places all its members at **the same address**. Reading a different member simply reinterprets the same bytes.

```
union U {
    int i;
    char c[4];
};
```

NOTE

x86 is **little endian**: the least significant byte of an integer is stored at the lowest address. After `u.i = 0x04030201` the bytes in memory are `01 02 03 04`, so `u.c[0]` is `0x01`.

```
; reading each byte (NASM)            # reading each byte (GAS)
mov al, [rbp - 4]                      movb -4(%rbp), %al    # u.c[0]
mov al, [rbp - 3]                      movb -3(%rbp), %al    # u.c[1]
mov al, [rbp - 2]                      movb -2(%rbp), %al    # u.c[2]
mov al, [rbp - 1]                      movb -1(%rbp), %al    # u.c[3]
```



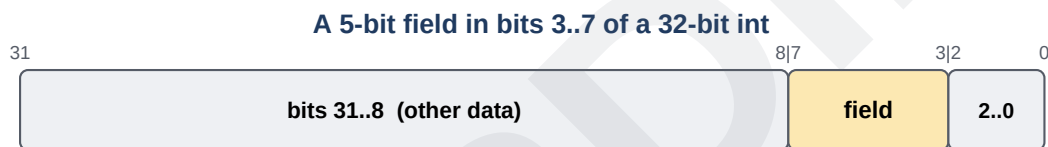
Little endian: the lowest byte of the int is stored first, so c[0] = 0x01.

Figure 9.1: A structure puts members side by side; a union overlays them.

9.3 Bitfields

A bitfield packs several small values into one integer. The assembly follows two fixed recipes:

- **Read:** load the value, **shift** right by the field's lowest bit number, **mask** with the field's width.
- **Write:** load the value, **clear** the field with an AND mask, **OR** in the new value shifted into place, **store** it back.



READ: `shr eax, 3` ; field moves down to bit 0
 `and eax, 0x1F` ; keep only the 5 field bits

WRITE: `and eax, ~(0x1F << 3)` ; clear the field
 `or eax, (17 << 3)` ; put the new value in

Figure 9.2: Reading and writing a 5-bit field that starts at bit 3.

```

; reading 5 bits, bit 3 to bit 7
; NASM
mov eax, [rbp - 4]
shr eax, 3
and eax, 0x1F

# GAS
movl -4(%rbp), %eax
shrl $3, %eax
andl $0x1F, %eax

; writing the value 17 at bit 3
mov eax, [rbp - 4]
and eax, ~(0x1F << 3)
or eax, (17 << 3)
mov [rbp - 4], eax

movl -4(%rbp), %eax
andl $~(0x1F << 3), %eax
orl $(17 << 3), %eax
movl %eax, -4(%rbp)

```

WATCH OUT

If the new value is not a constant, first mask it to the field width (`and ecx, 0x1F`) and shift it (`shl ecx, 3`) before the `or`, otherwise extra bits spill into neighbouring fields.

EXTRA

How a compiler lays out bitfields (which end it starts from, whether they cross word boundaries) is implementation-defined. Hand-written code should state the exact bit positions it assumes.

9.4 Complete programs

These programs use one stack area for a struct, a union and a bitfield. They exit with $7 + 1 + 4 + 17 = 29$.

NASM

```
section .text
global _start
_start:
    push rbp
    mov rbp, rsp
    sub rsp, 16                ; room for the locals below

    ; ---- struct pair { int a; int b; } p;   at [rbp-16] ----
    mov dword [rbp-16], 3      ; p.a = 3
    mov dword [rbp-12], 4      ; p.b = 4
    mov eax, [rbp-16]
    add eax, [rbp-12]          ; sum = p.a + p.b = 7
    mov r8d, eax

    ; ---- union U { int i; char c[4]; } u;   at [rbp-4] ----
    mov dword [rbp-4], 0x04030201 ; u.i
    movzx eax, byte [rbp-4]      ; u.c[0] = 1 (little endian: lowest byte first)
    movzx ebx, byte [rbp-1]      ; u.c[3] = 4
    add r8d, eax
    add r8d, ebx                ; 12

    ; ---- bitfield: bits 3..7 of the int at [rbp-8] ----
    mov dword [rbp-8], 0xFFFFFFFF
    mov eax, [rbp-8]            ; write the value 17 into bits 3..7
    and eax, ~(0x1F << 3)      ; 1) clear the field
    or eax, (17 << 3)           ; 2) OR in the new value
    mov [rbp-8], eax            ; 3) store back
    mov eax, [rbp-8]            ; read it back
    shr eax, 3                  ; 1) shift the field down to bit 0
    and eax, 0x1F               ; 2) mask to 5 bits -> 17
    add r8d, eax                ; 29

    mov edi, r8d                ; exit status = 29
    mov eax, 60
    syscall
```

GAS

```
.section .text
.globl _start
_start:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp                # room for the locals below

    # ---- struct pair { int a; int b; } p;   at -16(%rbp) ----
    movl $3, -16(%rbp)           # p.a = 3
    movl $4, -12(%rbp)           # p.b = 4
    movl -16(%rbp), %eax
    addl -12(%rbp), %eax          # sum = p.a + p.b = 7
    movl %eax, %r8d

    # ---- union U { int i; char c[4]; } u;   at -4(%rbp) ----
    movl $0x04030201, -4(%rbp)   # u.i
    movzbl -4(%rbp), %eax        # u.c[0] = 1 (little endian: lowest byte first)
    movzbl -1(%rbp), %ebx        # u.c[3] = 4
    addl %eax, %r8d
    addl %ebx, %r8d              # 12

    # ---- bitfield: bits 3..7 of the int at -8(%rbp) ----
    movl $0xFFFFFFFF, -8(%rbp)
    movl -8(%rbp), %eax          # write the value 17 into bits 3..7
    andl $~(0x1F << 3), %eax    # 1) clear the field
    orl $(17 << 3), %eax         # 2) OR in the new value
    movl %eax, -8(%rbp)         # 3) store back
    movl -8(%rbp), %eax          # read it back
    shrl $3, %eax               # 1) shift the field down to bit 0
    andl $0x1F, %eax            # 2) mask to 5 bits -> 17
    addl %eax, %r8d              # 29

    movl %r8d, %edi              # exit status = 29
    movl $60, %eax
    syscall
```

10 Control-Flow Patterns

10.1 Compare, test and jump

Control flow has two steps: an instruction sets the flags, then a conditional jump reads them. `cmp a, b` subtracts and discards the result; `test a, b` ANDs and discards it. `test eax, eax` is the idiomatic way to ask "is eax zero or negative?".

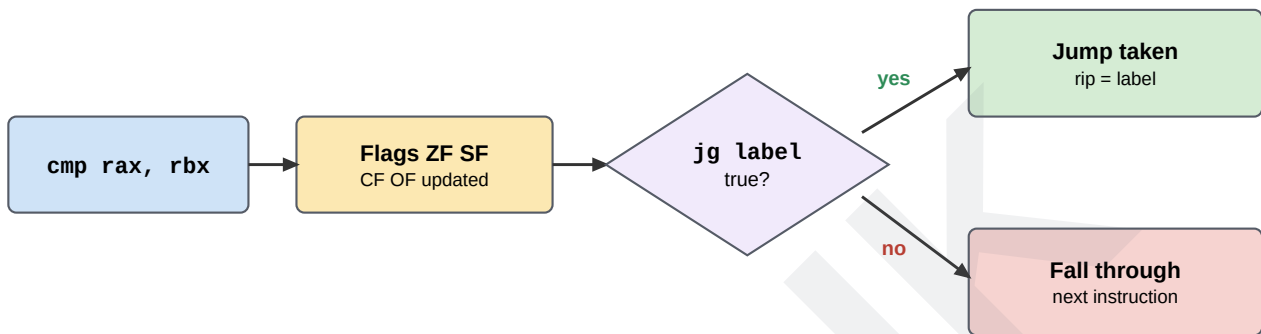


Figure 10.1: A compare sets the flags; a conditional jump tests them and either jumps or falls through.

C code	NASM	GAS
if (a > b) goto L	cmp rax, rbx jg L	cmpq %rbx, %rax jg L
if (a == 0) goto L	test rax, rax jz L	testq %rax, %rax jz L
if (a <= b) goto L (unsigned)	cmp rax, rbx jbe L	cmpq %rbx, %rax jbe L

10.2 if / else

TIP

When translating `if (cond) { A } else { B }`, jump on the **opposite** condition to skip block A. That keeps the true-case code falling straight through.

```
; if (a > b) x = 1; else x = 2;      a = eax, b = ebx, x = ecx
    cmp eax, ebx
    jle .else                      ; opposite of >
    mov ecx, 1
    jmp .endif
.else:
    mov ecx, 2
.endif:
```

10.3 while loop

```
; while (i < n) { ...; i++; }      i = ecx, n = edx
.loop:
    cmp ecx, edx
    jge .done                    ; opposite of <
    ; ... loop body ...
    inc ecx
    jmp .loop
.done:
```

10.4 for loop

A **for** loop is a **while** loop with the initialiser before it and the increment at the bottom. The complete programs below contain one.

10.5 do-while loop

```
; do { ...; n--; } while (n != 0);
.do:
    ; ... body ...
    dec ecx                    ; decrement AND set the flags
    jnz .do                   ; repeat while the result was not zero
```

EXTRA

dec then **jnz** is the idiomatic counted loop: the decrement already sets ZF, so no separate **cmp** is needed.

10.6 Putting them together

These programs combine a **for** loop, an **if / else** and a **do-while** loop, and exit with $55 + 1 + 4 = 60$.

NASM

```
section .text
global _start
_start:
    ; for (i = 1; i <= 10; i++) sum += i;
    xor eax, eax                ; sum = 0
    mov ecx, 1                  ; i = 1
.for:
    cmp ecx, 10
    jg .endfor                  ; opposite of <=
    add eax, ecx
    inc ecx
    jmp .for
.endfor:                        ; eax = 55

    ; if (sum > 50) r = 1; else r = 2;
    cmp eax, 50
    jle .else                   ; opposite of >
    mov ebx, 1
    jmp .endif
.else:
    mov ebx, 2
.endif:                         ; ebx = 1

    ; do { count++; n--; } while (n != 0);    n starts at 4
    mov ecx, 4
    xor edx, edx
.do:
    inc edx
    dec ecx                     ; dec sets the flags
    jnz .do                    ; repeat while the result was not zero
                                ; edx = 4

    add eax, ebx
    add eax, edx                ; 55 + 1 + 4 = 60
    mov edi, eax
    mov eax, 60
    syscall
```

GAS

```
.section .text
.globl _start
_start:
    # for (i = 1; i <= 10; i++) sum += i;
    xorl %eax, %eax          # sum = 0
    movl $1, %ecx            # i = 1
.Lfor:
    cmpl $10, %ecx
    jg .Lendfor              # opposite of <=
    addl %ecx, %eax
    incl %ecx
    jmp .Lfor
.Lendfor:                    # eax = 55

    # if (sum > 50) r = 1; else r = 2;
    cmpl $50, %eax
    jle .Lelse               # opposite of >
    movl $1, %ebx
    jmp .Lendif
.Lelse:
    movl $2, %ebx
.Lendif:                     # ebx = 1

    # do { count++; n--; } while (n != 0);    n starts at 4
    movl $4, %ecx
    xorl %edx, %edx
.Ldo:
    incl %edx
    decl %ecx                 # dec sets the flags
    jnz .Ldo                  # repeat while the result was not zero
                                # edx = 4

    addl %ebx, %eax
    addl %edx, %eax           # 55 + 1 + 4 = 60
    movl %eax, %edi
    movl $60, %eax
    syscall
```

NOTE

NASM local labels begin with a dot (**.loop**) and belong to the previous normal label. In GAS, labels beginning with **.L** are local and do not appear in the symbol table.

10.7 Conditional move

cmovcc copies a register only if the condition holds, so a small **if** needs no jump at all. It is x86's counterpart to ARM's conditional execution.

NASM

```
section .text
global _start

; long max(long a, long b)
max:
    mov rax, rdi                ; result = a
    cmp rdi, rsi
    cmovl rax, rsi              ; if (a < b) result = b    (signed)
    ret

_start:
    mov edi, 3
    mov esi, 9
    call max                    ; rax = 9
    mov rdi, rax
    mov eax, 60
    syscall
```

GAS

```
.section .text
.globl _start

# long max(long a, long b)
max:
    movq %rdi, %rax             # result = a
    cmpq %rsi, %rdi             # compare a with b    (AT&T: 2nd operand - 1st)
    cmovl %rsi, %rax            # if (a < b) result = b    (signed)
    ret

_start:
    movl $3, %edi
    movl $9, %esi
    call max                    # rax = 9
    movq %rax, %rdi
    movl $60, %eax
    syscall
```

`cmp rdi, rsi` followed by `cmovl rax, rsi` reads "if rdi is less than rsi (signed), copy rsi into rax". The program returns the larger of 3 and 9.

11 Functions and the Stack Frame

11.1 call and ret

`call label` pushes the address of the next instruction on the stack and jumps to the label. `ret` pops that address into `rip`, returning to the caller. Arguments are in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`; the result is in `rax`.

```
/* C */  add(5, 7);

mov edi, 5          ; NASM
mov esi, 7
call add            ; eax = 12 after the call

movl $5, %edi       # GAS
movl $7, %esi
call add
```

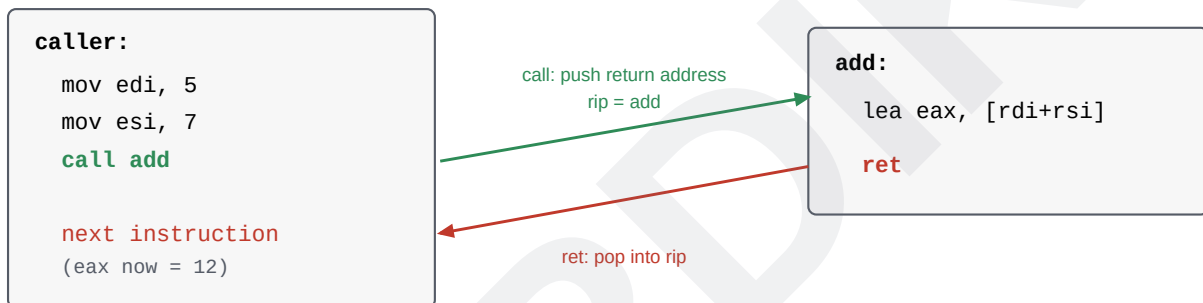


Figure 11.1: `call` saves the return address on the stack; `ret` jumps back to it.

11.2 The stack frame

A function that needs local variables builds a **stack frame** between `rbp` and `rsp`. After the standard prologue the layout is fixed, so every item has a constant offset from `rbp`:

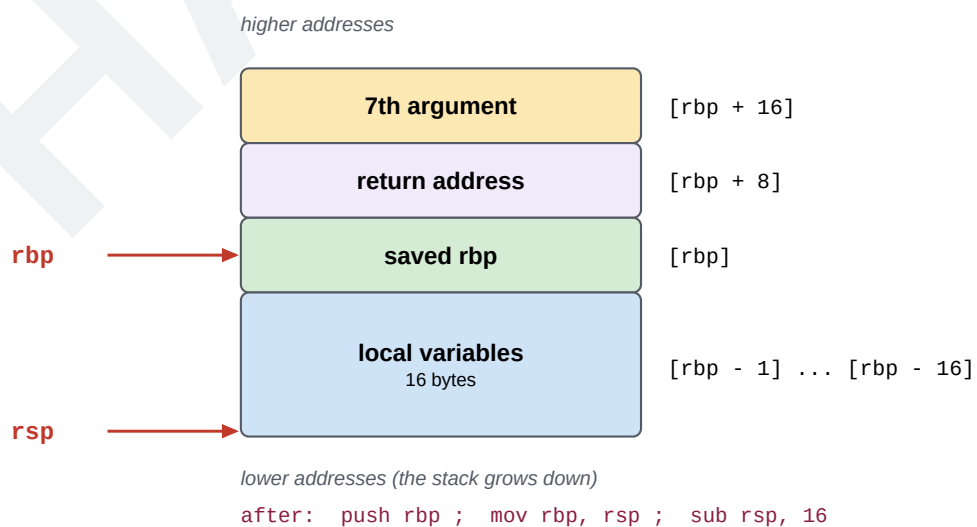


Figure 11.2: A stack frame after the prologue, with the offset of each item from `rbp`.

11.3 Local variables

```
/* C */
int sum(int a, int b) { int temp; temp = a + b; return temp; }
```

NASM

```
section .text
global _start

; int sum(int a, int b) { int temp; temp = a + b; return temp; }
sum:
    push rbp
    mov rbp, rsp
    sub rsp, 16                ; allocate locals (keeps rsp 16-byte aligned)
    lea eax, [rdi + rsi]
    mov [rbp-4], eax           ; temp = a + b
    mov eax, [rbp-4]           ; return temp
    leave                      ; same as: mov rsp, rbp / pop rbp
    ret

_start:
    mov edi, 5
    mov esi, 7
    call sum                   ; eax = 12
    mov edi, eax               ; exit status = 12
    mov eax, 60
    syscall
```

GAS

```
.section .text
.globl _start

# int sum(int a, int b) { int temp; temp = a + b; return temp; }
sum:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp            # allocate locals (keeps rsp 16-byte aligned)
    leal (%rdi,%rsi), %eax
    movl %eax, -4(%rbp)        # temp = a + b
    movl -4(%rbp), %eax        # return temp
    leave                      # same as: movq %rbp,%rsp / popq %rbp
    ret

_start:
    movl $5, %edi
    movl $7, %esi
    call sum                   # eax = 12
    movl %eax, %edi            # exit status = 12
    movl $60, %eax
    syscall
```

`sum` allocates 16 bytes (not 4) so the stack stays 16-byte aligned. `leave` undoes the prologue (it restores `rsp` and pops `rbp`), and `ret` returns. The program exits with $5 + 7 = 12$.

EXTRA

`sum` calls nothing (it is a **leaf function**), so the frame is not strictly needed: `lea eax, [rdi + rsi]` then `ret` would do. The longer version is shown to demonstrate locals.

11.4 More than six arguments

The first six integer arguments travel in registers; the seventh is pushed on the stack. Inside the function it sits **above** the saved `rbp` and the return address, at `[rbp + 16]`.

- `[rbp + 0]` is the saved `rbp`.
- `[rbp + 8]` is the return address.
- `[rbp + 16]` is the seventh argument, `[rbp + 24]` the eighth, and so on.

WATCH OUT

Keep the stack aligned: in `_start` the stack is aligned, so push **two** slots (one of padding and one for the argument) before the `call`, and remove both afterwards.

NASM

```
section .text
global _start

; long f(long a, long b, long c, long d, long e, long g, long h)
f:
    push rbp
    mov rbp, rsp
    lea rax, [rdi + rsi]      ; a + b
    add rax, rdx              ; + c
    add rax, rcx              ; + d
    add rax, r8               ; + e
    add rax, r9               ; + g
    add rax, [rbp + 16]       ; + h (7th arg: above saved rbp + return addr)
    pop rbp
    ret

_start:
    mov edi, 1
    mov esi, 2
    mov edx, 3
    mov ecx, 4
    mov r8d, 5
    mov r9d, 6
    sub rsp, 8                ; padding so rsp is 16-byte aligned at the call
    push 7                    ; 7th argument goes on the stack
    call f                    ; rax = 28
    add rsp, 16                ; remove argument and padding

    mov rdi, rax               ; exit status = 28
    mov eax, 60
    syscall
```

GAS

```
.section .text
.globl _start

# long f(long a, long b, long c, long d, long e, long g, long h)
f:
    pushq %rbp
    movq %rsp, %rbp
    leaq (%rdi,%rsi), %rax    # a + b
    addq %rdx, %rax          # + c
    addq %rcx, %rax          # + d
    addq %r8, %rax           # + e
    addq %r9, %rax           # + g
    addq 16(%rbp), %rax       # + h (7th arg: above saved rbp + return addr)
    popq %rbp
    ret

_start:
    movl $1, %edi
    movl $2, %esi
    movl $3, %edx
    movl $4, %ecx
    movl $5, %r8d
    movl $6, %r9d
    subq $8, %rsp            # padding so rsp is 16-byte aligned at the call
    pushq $7                 # 7th argument goes on the stack
    call f                   # rax = 28
    addq $16, %rsp           # remove argument and padding

    movq %rax, %rdi          # exit status = 28
    movl $60, %eax
    syscall
```

The sum of 1 through 7 is 28, which becomes the exit status.

11.5 Calling C library functions

When you link with `gcc` you can call `printf`. Three details matter: the stack must be 16-byte aligned (the `push rbp` in `main` does that), `al` must hold the number of vector registers used (0 here), and the call must go through the PLT so the program stays position independent.

NASM

```
default rel ; every memory reference is rip-relative
extern printf

section .data
fmt db "sum = %d", 10, 0

section .text
global main
main:
    push rbp ; also re-aligns rsp to 16 bytes
    mov rbp, rsp
    lea rdi, [fmt] ; arg 1: format string
    mov esi, 42 ; arg 2: the number
    xor eax, eax ; al = number of vector registers used (0)
    call printf wrt ..plt ; call through the PLT (needed for PIE)
    xor eax, eax ; return 0
    pop rbp
    ret

section .note.GNU-stack noalloc noexec nowrite progbits
```

GAS

```
.section .rodata
fmt: .asciz "sum = %d\n"

.section .text
.globl main
main:
    pushq %rbp # also re-aligns rsp to 16 bytes
    movq %rsp, %rbp
    leaq fmt(%rip), %rdi # arg 1: format string
    movl $42, %esi # arg 2: the number
    xorl %eax, %eax # al = number of vector registers used (0)
    call printf@PLT # call through the PLT (needed for PIE)
    xorl %eax, %eax # return 0
    popq %rbp
    ret

.section .note.GNU-stack,"",@progbits
```

TIP

`default rel` at the top of a NASM file makes every memory reference rip-relative, so you can write `lea rdi, [fmt]` instead of `lea rdi, [rel fmt]`.

WATCH OUT

Skipping the alignment is the classic cause of a crash inside `printf`: an SSE instruction inside the C library faults on a misaligned stack.

11.6 Saving callee-saved registers

If your function uses `rbx`, `rbp` or `r12` to `r15`, push them on entry and pop them (in reverse order) on exit. The recursive example in the next chapter does this with `rbx`.

12 Worked Examples

EXTRA

Every program in this chapter was assembled and run. Each one ends by passing its answer to `exit`, so you can read it with `echo $?`. Examples alternate between NASM and GAS; translating each into the other syntax is good practice.

12.1 Factorial (recursion, NASM)

Recursion shows why callee-saved registers matter: each call keeps its own `n` in `rbx` and the previous value is restored on return. `push rbx` also realigns the stack to 16 bytes before the next `call`.

```
section .text
global _start

; long fact(long n) { if (n <= 1) return 1; return n * fact(n - 1); }
fact:
    push rbx                ; rbx is callee-saved: we use it, so save it
                           ; (the push also re-aligns rsp to 16 bytes)

    cmp rdi, 1
    jle .base
    mov rbx, rdi             ; remember n
    dec rdi
    call fact                ; rax = fact(n - 1)
    imul rax, rbx            ; n * fact(n - 1)
    pop rbx
    ret

.base:
    mov eax, 1
    pop rbx
    ret

_start:
    mov edi, 5
    call fact                ; rax = 120
    mov rdi, rax
    mov eax, 60
    syscall
```

12.2 Greatest common divisor (GAS)

```
.section .text
.globl _start

# long gcd(long a, long b)    Euclid's algorithm with remainders
gcd:
    movq %rdi, %rax
    movq %rsi, %rcx
.Lloop:
    testq %rcx, %rcx
    jz .Ldone                # while (b != 0)
    xorl %edx, %edx          # upper half of the dividend = 0
    divq %rcx                # rdx = a % b
    movq %rcx, %rax          # a = b
    movq %rdx, %rcx          # b = remainder
    jmp .Lloop
.Ldone:
    ret                      # result is in rax

_start:
    movl $48, %edi
    movl $18, %esi
    call gcd                  # rax = 6
    movq %rax, %rdi
    movl $60, %eax
    syscall
```

12.3 String length (NASM)

```
section .data
s db "assembly", 0

section .text
global _start

; size_t my_strlen(const char *s)
my_strlen:
    xor eax, eax              ; count = 0
.next:
    cmp byte [rdi + rax], 0    ; end of string?
    je .done
    inc rax
    jmp .next
.done:
    ret

_start:
    lea rdi, [rel s]
    call my_strlen             ; rax = 8
    mov rdi, rax
    mov eax, 60
    syscall
```

12.4 Fibonacci (GAS)

```
.section .text
.globl _start

# long fib(long n)      fib(0) = 0, fib(1) = 1
fib:
    xorl %eax, %eax      # a = 0
    movl $1, %ecx        # b = 1
.Lloop:
    testq %rdi, %rdi
    jz .Ldone
    leaq (%rax,%rcx), %rdx # t = a + b
    movq %rcx, %rax       # a = b
    movq %rdx, %rcx       # b = t
    decq %rdi
    jmp .Lloop
.Ldone:
    ret                  # return a

_start:
    movl $10, %edi
    call fib              # rax = 55
    movq %rax, %rdi
    movl $60, %eax
    syscall
```

12.5 Counting set bits (NASM)

```
section .text
global _start
_start:
    mov eax, 0xF0F1      ; the number to examine
    xor ecx, ecx         ; count = 0
.loop:
    test eax, eax
    jz .done
    test eax, 1          ; lowest bit set?
    jz .skip
    inc ecx              ; yes: count it
.skip:
    shr eax, 1           ; move the next bit into place
    jmp .loop
.done:
    ; ecx = 9
    mov edi, ecx
    mov eax, 60
    syscall
```

12.6 Maximum of an array (GAS)

```
.section .data
arr:    .long 3, 7, 2, 9, 4

.section .text
.globl _start

# int array_max(const int *p, long n)    rdi = p, rsi = n
array_max:
    movl (%rdi), %eax                # best = p[0]
    movl $1, %ecx                    # i = 1
.Lloop:
    cmpq %rsi, %rcx
    jge .Ldone
    movl (%rdi,%rcx,4), %edx          # edx = p[i]
    cmpl %eax, %edx
    jle .Lskip                        # signed compare: keep best if p[i] <= best
    movl %edx, %eax                  # new maximum
.Lskip:
    incq %rcx
    jmp .Lloop
.Ldone:
    ret

_start:
    leaq arr(%rip), %rdi
    movl $5, %esi
    call array_max                    # eax = 9
    movl %eax, %edi
    movl $60, %eax
    syscall
```

NOTE

The signed compare **jle** keeps the best value so far when **p[i] <= best**. For unsigned data, use **jbe** instead.

13 Quick Reference and Practice

13.1 Instruction cheat sheet

NASM	GAS	Meaning
mov rax, rbx	movq %rbx, %rax	copy
add rax, 5	addq \$5, %rax	add
sub rax, rbx	subq %rbx, %rax	subtract
imul rax, rbx	imulq %rbx, %rax	signed multiply
idiv rbx	idivq %rbx	signed divide rdx:rax
cmp rax, rbx	cmpq %rbx, %rax	compare (flags only)
test rax, rax	testq %rax, %rax	AND (flags only)
lea rax, [rbx+8]	leaq 8(%rbx), %rax	address arithmetic
movzx eax, byte [rdi]	movzbl (%rdi), %eax	zero-extend load
movsx rax, byte [rdi]	movsbq (%rdi), %rax	sign-extend load
push rbp / pop rbp	pushq %rbp / popq %rbp	stack
call f / ret	call f / ret	function call and return
jmp / je / jne / jl / ja	same	jumps
syscall	syscall	system call

13.2 Function checklist

- Arguments in **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9** (extras on the stack); result in **rax**.
- If you use **rbx**, **rbp** or **r12** to **r15**, save them and restore them.
- Keep **rsp** 16-byte aligned at every **call**.
- Address every variable relative to **rip**.
- Use the register size (NASM) or suffix (GAS) that matches the data.
- System calls: number in **rax**, arguments in **rdi**, **rsi**, **rdx**, **r10**, **r8**, **r9**; **rcx** and **r11** are destroyed.

13.3 Practice exercises

EXTRA

Check each answer with the exit status.

- **Sum of 1 to n.** Write a loop that adds $1 + 2 + \dots + 10$ and exits with 55.
- **Even or odd.** Use **test eax, 1** to return 1 for even numbers and 0 for odd ones.
- **Absolute value.** Return $|n|$ using **neg** and **cmovs**, with no jump.
- **Power.** Compute 2 to the power n with **shl**, then 3 to the power n with **imul** in a loop.
- **Reverse a string.** Reverse a zero-terminated string in place with two pointers.

- **Minimum and maximum.** Extend the array-maximum example so it also returns the minimum.
- **Seven arguments.** Write `f(a, b, c, d, e, g, h)` that returns `a*b + c*d + e*g + h`.
- **Copy a file.** Extend the file example to copy one file into another with `read` and `write`.

13.4 Where to go next

- Compile a small C function with `gcc -S -O1 -masm=intel` and read what the compiler produced.
- Learn `gdb`: `layout asm`, `stepi` and `info registers` show every instruction and flag.
- Explore SSE and AVX, the vector instructions that use `xmm` and `ymm` registers.
- Compare with ARM, where only `LDR` and `STR` touch memory and almost every instruction can be conditional.

Good luck, and happy hacking. — HARDIK

Connect with me on LinkedIn: www.linkedin.com/in/hardikhardik