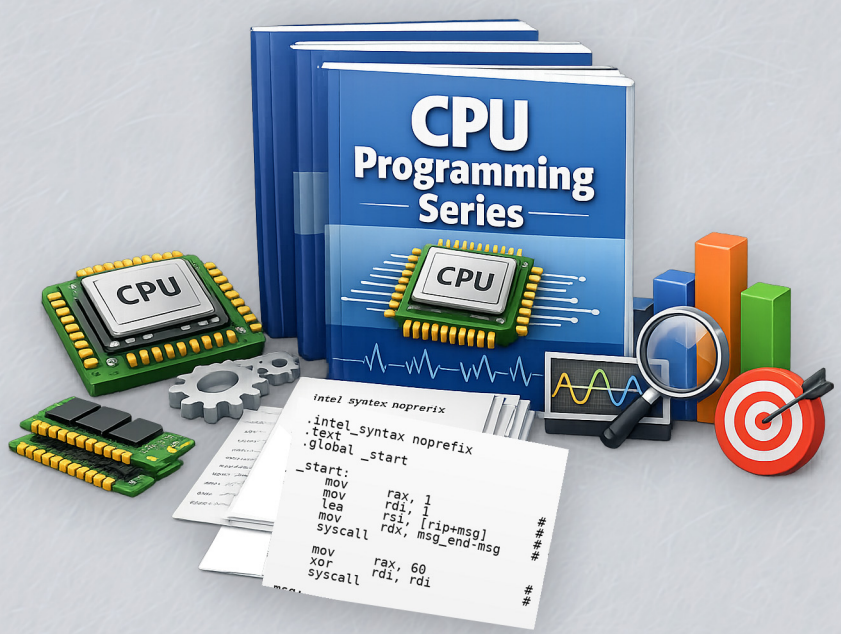
https://simplifycpp.org

CPU Programming Series

x86 Control Flow in Depth

Branches, Conditions, Calls, and Returns



6

Prepared by Ayman Alheraki

CPU Programming Series

x86 Architecture Fundamentals

Registers, Flags, and Addressing Modes

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

P.1 Purpose of This Booklet

This booklet teaches x86 control flow as the CPU actually executes it: decisions derived from flags, redirection of the instruction pointer, and the mechanics of calls and returns. The goal is not to memorize mnemonics, but to gain the ability to read and reason about real machine code produced by modern compilers, and to predict how small instruction-level choices change correctness, security, and performance.

By the end, you should be able to: identify the exact condition tested by a branch, distinguish signed from unsigned decisions, recognize compiler loop forms, explain direct versus indirect control transfers, and understand what `call` and `ret` do at the micro-mechanical level (return address handling and stack interaction).

```
cmp     eax, ebx        # flags := eax - ebx (result not stored)
jl      less_signed     # signed: uses SF and OF
jb      less_unsigned   # unsigned: uses CF
```

P.2 Why Control Flow Deserves a Dedicated Volume

Most low-level bugs that survive review do not come from arithmetic; they come from misunderstood decisions. In x86, the CPU does not know your variable types. It only sees bit

patterns and flags. Control flow is built on this fact, and a single wrong choice of conditional jump can silently invert program logic.
Control flow also sits at the center of: correctness (wrong branch conditions), security (unexpected indirect control transfer targets), and performance (branch predictability and the frequency of mispredictions).
A dedicated volume is necessary because control flow is not one instruction; it is an ecosystem: flag-producing instructions (`cmp`, `test`, arithmetic), flag-consuming instructions (conditional jumps, conditional moves, setcc), and control-transfer instructions (`jmp`, `call`, `ret`, indirect transfers).

```
test    rdi, rdi        # sets ZF if rdi == 0 (bitwise AND, result
↪  discarded)
je      is_null         # branches if ZF==1

cmp     edi, 10         # flags := edi - 10
jae     ge_10_unsigned  # unsigned: CF==0
jge     ge_10_signed    # signed: SF==OF
```

P.3 Scope and Boundaries of Coverage

This booklet focuses on control flow inside the ISA boundary: how flags are produced and consumed, how the instruction pointer is redirected, how near calls and returns operate, and how compilers typically lower high-level constructs into these mechanisms.
Included topics:

- `cmp` and `test` as decision primitives
- Conditional branches and the signed/unsigned distinction
- Direct and indirect jumps

- Jump tables and common `switch` lowering patterns
- `call` and `ret` mechanics, including return-address handling
- Canonical loop patterns and how compilers form them

Excluded topics (handled in later booklets):

- ABI and calling conventions (argument passing, callee/caller-saved rules)
- Stack-frame layout policies (prologues/epilogues as ABI artifacts)
- OS mechanisms (signals, exceptions, kernel transitions, syscalls)
- Deep microarchitecture topics (prediction structures, speculation internals)

Where performance is mentioned, it is framed only through observable ISA-level behavior (e.g., branch directionality, fall-through structure, dependency on flags), without relying on microarchitecture-specific tuning.

P.4 How to Read This Booklet (Concept → Instruction → Pattern)

Every section follows a strict learning sequence.
Concept: define the mechanism in CPU terms (flags, instruction pointer, stack interaction).
Instruction: show the minimal instruction semantics that implement the concept.
Pattern: show how compilers combine instructions to implement real constructs (if/else, loops, switch, calls).
Example 1 (concept: boolean test → instruction: `test` → pattern: null check):

```
test    rax, rax        # ZF=1 if rax==0
jne     nonzero         # common compiler pattern for if(ptr) ...
```

Example 2 (concept: range check → instruction: compare → pattern: jump-table guard):

```
cmp     edi, 7          # max case index = 7
ja      default_case    # unsigned: if edi > 7 then default
jmp     qword ptr [r11 + rdi*8]   # indirect jump through table
```

Example 3 (concept: loop control → instruction: dec/jcc → pattern: counted loop):

```
mov     ecx, 10
.Lloop:
# loop body
dec     ecx
jne     .Lloop
```

Read with the mindset: flags are data, and branches are consumers of that data. The instruction pointer is the true control state. `call`/`ret` are specialized control transfers that additionally use memory (the stack) to preserve return state.

P.5 Relationship to Other Booklets in the CPU Programming Series

This booklet assumes you already understand x86 registers, the key status flags, and effective addressing. Those are the static components. Control flow is the dynamic layer built on top of them.
In the series progression:

- Earlier fundamentals explain how instructions execute and how flags become meaningful.
- This booklet applies those fundamentals to real decision-making and redirection of execution.

- The next booklet on stack and calling conventions builds on the `call`/`ret` mechanics here, adding ABI rules, function prologues/epilogues, argument passing, and stack discipline across real-world toolchains.

Practical outcome: after finishing this booklet, you should be able to read disassembly and answer, with precision:

- What condition is being tested (and is it signed or unsigned)?
- Which instruction produced the flags used by this branch?
- Is this a direct or indirect control transfer, and what are the target computation rules?
- Is this structure a compiler-lowered if/else, loop, or switch?
- What exactly is pushed by `call`, and what exactly is consumed by `ret`?

Chapter 1

Control Flow as a CPU Concept

1.1 What "Control Flow" Really Means at the CPU Level

At the CPU level, **control flow** is the set of rules and mechanisms that determine which instruction executes next. The processor repeatedly fetches the instruction located at the address held in the instruction pointer (IP/RIP), decodes it, executes it, and updates IP/RIP to select the next instruction.

Most instructions advance execution sequentially, but a specific class of instructions modifies IP/RIP to redirect execution. This redirection is the essence of control flow and is independent of any programming language construct.

Control flow transfers fall into distinct architectural categories:

- unconditional transfers such as `jmp`,
- conditional transfers such as `je`, `jl`, and `jb`,
- subroutine transfers such as `call` and `ret`,
- computed transfers such as indirect `jmp` or `call`.

```
mov     eax, 1
add     eax, 2          # sequential execution

jmp     target          # control flow redirection

target:
xor     eax, eax
```

1.2 Sequential Execution vs Flow Redirection

Sequential execution means the instruction pointer advances to the next instruction based on the current instruction length. This behavior dominates normal execution and is what allows instruction streams to be linear in memory.
Flow redirection occurs when an instruction explicitly updates IP/RIP to a non-sequential value. Redirection can be relative, encoded as an offset, or indirect, obtained from a register or memory location.
A conditional jump always creates two conceptual paths:

- the fall-through path when the condition is false,
- the target path when the condition is true.

```
cmp     edi, 0
je      .Lzero          # taken path

mov     eax, 1          # fall-through path
jmp     .Ldone

.Lzero:
mov     eax, 0

.Ldone:
```

```
ret
```

1.3 Instruction Pointer (IP / RIP) and Its Role

The instruction pointer is the architectural register that holds the address of the next instruction to execute. In x86-32 this is commonly known as `EIP`; in x86-64 it is `RIP`. Software cannot arbitrarily modify this register using data-movement instructions.
IP/RIP changes only through control-transfer instructions. This guarantees that all redirection of execution follows explicit architectural rules.

```
call    func            # pushes return address; RIP updated
# execution resumes here after ret

func:
ret                     # RIP restored from stack
```

In x86-64, RIP is also used implicitly in address computation for position-independent code.

```
lea     rax, [rip + .Ltable]
jmp     qword ptr [rax + rdi*8]
```

1.4 Control Flow vs Data Flow

Data flow describes how values move through registers and memory and how they are transformed by arithmetic and logic instructions. Control flow describes how execution moves between instruction sequences.
The two are tightly coupled:

- data-flow instructions often produce flags,
- control-flow instructions consume those flags to select the next execution path.

```
cmp     eax, ebx        # data flow affecting flags
jne     .Lnot_equal     # control flow decision
```

Flags are ephemeral architectural state. Any instruction that sets flags overwrites previous values, making it essential to identify the precise flag-producing instruction associated with each conditional branch.

1.5 Why Control Flow Is Central to Performance and Correctness

Control flow governs which instructions execute and which do not, making it central to program correctness. An incorrect conditional jump or misinterpreted comparison silently alters program behavior.

It is also critical for security because indirect transfers and returns rely on memory-resident targets. Any corruption of these values redirects execution unpredictably.

From a performance perspective, control flow determines the shape of execution paths. Branch-heavy code, deeply nested decisions, and indirect jumps create execution patterns that are fundamentally different from straight-line code.

```
cmp     eax, ebx
jl      .Lsigned_less       # signed comparison
jb      .Lunsigned_below    # unsigned comparison
```

Understanding control flow at the ISA level enables precise reasoning about program behavior, which is a prerequisite for later analysis of calling conventions, stack discipline, and optimization effects covered in subsequent booklets of this series.

Chapter 2

Flags as the Foundation of Decisions

2.1 Status Flags Involved in Control Flow

Control-flow decisions in x86 are built on a small set of architectural status flags in RFLAGS/EFLAGS. Conditional branches do not "compare operands" themselves; they only test flag bits that were produced by an earlier instruction. For control flow, the most important flags are:

- **ZF (Zero Flag)**: set when a result is zero.
- **SF (Sign Flag)**: copies the most significant bit of the result (interpreted as a sign bit in two's complement).
- **CF (Carry Flag)**: indicates carry/borrow in *unsigned* arithmetic (also used by shifts/rotates).
- **OF (Overflow Flag)**: indicates signed overflow in two's complement arithmetic.

These four flags are sufficient to express the semantics of the common conditional jumps:

`je/jne` (ZF), `jb/jae/ja/jbe` (CF and ZF for unsigned), `jl/jge/jg/jle` (SF and OF and ZF for signed).

2.1.1 ZF, SF, CF, OF (Conceptual Review)

- **ZF**: 1 if the last flagged result equals 0.
- **SF**: copies the last flagged result's MSB.
- **CF**: for addition, 1 if there was a carry out of the MSB; for subtraction, 1 if there was a borrow (i.e., unsigned underflow).
- **OF**: 1 if the signed result does not fit in the operand width (two's complement overflow).

Minimal flag-driven decisions:

```
test    rax, rax        # sets ZF if rax==0; sets SF from MSB; clears OF;
↪  CF=0
je      .Lis_zero       # branch if ZF==1

cmp     eax, ebx        # sets flags from eax - ebx
jb      .Lbelow_u       # unsigned: CF==1  (eax < ebx as unsigned)
jl      .Lless_s        # signed:  SF != OF (eax < ebx as signed)
```

2.2 How Arithmetic Instructions Set Flags

Many arithmetic and logical instructions update flags as a side effect. This is not optional: the ISA defines which flags are modified and how. In control-flow analysis, you must identify *the exact instruction that produced the flags* consumed by a later `jcc`.
Core flag-producing instructions for control flow:

- `add`, `sub`, `adc`, `sbb`, `inc`, `dec`
- `cmp` (like `sub` but discards the result)
- `test` (bitwise AND for flags only)
- `and`, `or`, `xor` (common for zero-checks and masking)
- shifts: `shl/sal`, `shr`, `sar` (CF receives shifted-out bit; others depend on result)

Arithmetic example: unsigned vs signed meaning is not stored anywhere; only the chosen conditional jump interprets flags accordingly.

```
mov     al, 250
add     al, 10          # 250 + 10 = 260 -> wraps in 8-bit to 4
                        # CF=1 (carry out), ZF=0 (result not zero), SF=0
                        ↪  (MSB of 4 is 0)
jc      .Lcarry_taken   # unsigned overflow indication (carry)

mov     al, 120
add     al, 120         # 120 + 120 = 240; as signed int8: 120 + 120
↪  overflows
                        # OF=1 (signed overflow), CF may be 0 depending on
                        ↪  carry-out
jo      .Loverflow_taken
```

Subtraction example: in x86, `CF=1` after `sub/cmp` indicates a borrow, which corresponds to "below" in unsigned comparisons.

```
mov     eax, 3
cmp     eax, 5          # computes 3 - 5 (result discarded)
jb      .Lbelow         # CF==1 means unsigned eax < 5
```

Why `test` is preferred for zero checks: it is explicit, does not require an immediate 0, and keeps the original register unchanged.

```
test    rdi, rdi        # sets ZF if rdi==0 without changing rdi
je      .Lnull
```

2.3 Flag Lifetime and Overwriting Rules

Flags are **volatile architectural state**. The CPU does not track which instruction "owns" the flags. Any flag-writing instruction overwrites prior values, which means:

- the flag consumer (`jcc`) is correct only if no intervening instruction modified the relevant flags,
- instruction scheduling (by compiler or hand-written assembly) must preserve intended flag dependencies.

Classic pitfall: inserting any flag-setting instruction between a compare and a jump changes the meaning.

```
cmp     eax, ebx
add     ecx, 1          # overwrites flags (ZF/SF/CF/OF, etc.)
je      .Lequal         # now tests flags from the add, not from cmp (bug)
```

Correct structure: keep the compare adjacent to the branch, or use a non-flag-setting instruction in between.

```
cmp     eax, ebx
je      .Lequal

# or keep flags intact with lea (does not set flags)
cmp     eax, ebx
lea     ecx, [ecx + 1]  # updates ecx without affecting flags
je      .Lequal
```

2.4 Flag Dependencies Between Instructions

A conditional branch depends on flags as inputs. This creates an explicit dependency chain:

$$\texttt{flag-producer} \rightarrow \texttt{flags} \rightarrow \texttt{flag-consumer (jcc)}$$

Understanding control flow requires tracing this chain precisely.

Typical producer-consumer patterns:

Pattern A: compare then branch

```
cmp     rax, rbx        # producer
jge     .Lge_s          # consumer (signed: SF==OF)
```

Pattern B: test/mask then branch

```
test    eax, 0xFF       # producer: ZF indicates whether low byte is zero
jne     .Lhas_lowbyte   # consumer
```

Pattern C: subtract then branch on borrow/carry

```
sub     eax, 1          # producer
jc      .Lunderflow_u   # consumer: CF indicates unsigned
↪  underflow/carry/borrow usage
```

Practical rule when reading disassembly:

- locate the `jcc`,
- locate the closest preceding instruction that defines the required flags,
- verify no intervening instruction clobbers those flags.

2.5 Common Misconceptions About Flags

- **Misconception: "Signed vs unsigned is stored in the CPU."** The CPU stores only bits and flags. Signedness is implied by how you interpret operands and by which conditional jump you choose (`jl` vs `jb`, `jg` vs `ja`).
- **Misconception: "CF and OF mean the same thing."** CF is an *unsigned* carry/borrow indicator; OF is a *signed* overflow indicator. They can differ and frequently do.

- **Misconception: "Flags reflect the last arithmetic you care about."** Flags reflect the last instruction that wrote them, whether you intended it or not.
- **Misconception: "inc/dec behave like add/sub in every way."** `inc/dec` do not update CF, while `add/sub` do. This matters when code uses CF later.
- **Misconception: "cmp stores a result somewhere."** `cmp` only sets flags; it discards the subtraction result.
- **Misconception: "test is a compare."** `test` performs AND and sets flags based on the AND result; it is best understood as a masking/zero-check primitive.

Two short, high-signal examples that expose misconceptions:
Example 1: signed vs unsigned divergence:

```
mov     al, 0xFF        # 255 unsigned, -1 signed (int8)
cmp     al, 1
jb      .Lbelow_u       # true: 255 < 1 (unsigned)? false; CF=0 so not
↪  taken
jl      .Lless_s        # true: -1 < 1 (signed)? true; SF!=OF so taken
```

Example 2: CF not modified by inc:

```
stc                     # CF := 1
inc     eax             # CF unchanged
jc      .Lcarry_still_1 # taken because CF remained 1

clc                     # CF := 0
dec     eax             # CF unchanged
jc      .Lcarry_1       # not taken because CF remained 0
```

Chapter 3

`cmp` and `test` in Depth

3.1 Purpose of `cmp` (Subtraction Without Storing)

`cmp a, b` performs an internal subtraction `a - b` to update status flags, but it discards the result. Architecturally, it behaves like `sub` for flag-setting, except that `a` is not modified. This makes `cmp` the canonical instruction for ordering decisions: equality, less/greater (signed), below/above (unsigned).

Key property: `cmp` does not encode signedness. Signed vs unsigned meaning is selected later by the conditional jump (`jl` vs `jb`, `jg` vs `ja`, etc.).

```
cmp     eax, ebx        # flags := eax - ebx, eax unchanged
je      .Lequal         # ZF==1
jne     .Lne            # ZF==0

jb      .Lbelow_u       # unsigned: CF==1
jae     .Lge_u          # unsigned: CF==0

jl      .Lless_s        # signed:  SF!=OF
jge     .Lge_s          # signed:  SF==OF
```

Equality decision is always ZF-based:

```
cmp     rdi, rsi
je      .Lsame          # rdi == rsi
```

Ordering decision depends on how you interpret the operands:

```
cmp     eax, 10
jl      .Lx_lt_10_signed     # signed: eax < 10
jb      .Lx_lt_10_unsigned   # unsigned: eax < 10
```

3.2 Purpose of `test` (Bitwise AND for Flags Only)

`test a, b` computes `a & b` to update flags, but discards the result and does not modify operands. It is a flag-producing instruction specialized for:

- zero checks (`test reg, reg`),
- bit tests and masking (`test reg, imm`),
- sign checks (`test reg, reg` then `js/jns`).

Unlike `cmp`, `test` does not express ordering. It expresses *bit properties*.

```
test    rax, rax        # sets ZF if rax==0, sets SF from MSB
je      .Lis_zero

test    eax, 1          # tests bit0
jne     .Lodd            # (eax & 1) != 0

test    eax, 0x80000000 # tests sign bit of 32-bit value
jne     .Lmsb_set
```

3.3 Flag Effects of `cmp` vs `test`

Both instructions update ZF and SF from their internal result, but their semantic meaning differs because their internal operation differs.

`cmp a, b`:

- conceptually computes `a - b` (discarded),
- sets ZF/SF based on subtraction result,
- sets CF to indicate unsigned borrow (used by unsigned comparisons),
- sets OF to indicate signed overflow (used by signed comparisons).

`test a, b`:

- conceptually computes `a & b` (discarded),
- sets ZF/SF based on AND result,
- clears OF,
- sets CF to 0 (architecturally not useful for decisions after `test`).

Practical consequence:

- `cmp` enables ordering branches (`jl`, `jb`, `jg`, `ja`, etc.).
- `test` enables bit/zero/sign branches (`je`/`jne`, `js`/`jns`) but not ordering in the subtraction sense.

```
# ordering needs subtraction flags
cmp     eax, ebx
jl      .Lless_s        # depends on SF and OF
jb      .Lbelow_u       # depends on CF
```

```
# bit/zero property uses AND flags
test    eax, eax
je      .Lzero          # depends on ZF

test    eax, 0x20
jne     .Lbit5_set      # depends on ZF after masking
```

3.4 Typical Compiler Emission Patterns

Modern compilers use `cmp` and `test` in highly stereotyped ways.

Pattern A: pointer or integer zero check (most common):

```
test    rdi, rdi
je      .Lnull
```

Pattern B: boolean stored in register (0/1) and compared to zero:

```
test    al, al
jne     .Ltrue
```

Pattern C: compare against immediate (range, threshold):

```
cmp     edi, 10
jl      .Llt_10_s
```

Pattern D: equality against constant (often for enums / tags):

```
cmp     eax, 3
je      .Lcase3
```

Pattern E: switch lowering with bounds check:

```
cmp     edi, 7
ja      .Ldefault       # unsigned bounds check
jmp     qword ptr [rip + .Ljt + rdi*8]
```

Pattern F: bitmask test (flags or feature bits):

```
test    edx, 0x40
je      .Lfeature_off
```

3.5 When Compilers Prefer `test` Over `cmp`

Compilers prefer `test` when the question is "is this value zero?" or "is this bit set?" because `test` expresses the intent directly without requiring an explicit 0 immediate and without consuming an extra constant in the encoding.

Most important cases:

Case 1: register compared to zero

```
# preferred
test    rax, rax
je      .Lzero

# also correct but often larger / less idiomatic
cmp     rax, 0
je      .Lzero
```

Case 2: masking then branching

```
test    eax, 0xFF
je      .Llow_byte_zero
```

Case 3: sign-bit based branch without changing the value

```
test    eax, eax
js      .Lnegative      # branches if SF==1 after test
```

Case 4: preserving flags discipline When code needs a flag-producing instruction whose operands must not change, `test` is safe: it is read-only and explicit about AND semantics.

3.6 Manual Use Cases and Pitfalls

Manual assembly and reverse engineering benefit from using `cmp` and `test` with strict discipline.

3.6.1 Manual use cases

Zero checks:

```
test    rcx, rcx
je      .Lempty
```

Bit checks:

```
test    eax, 0x8
jne     .Lbit3_set
```

Range checks (bounds):

```
cmp     edi, 100
ja      .Lout_of_range_u     # unsigned
```

Signed ordering:

```
cmp     esi, -1
jg      .Lgreater_than_m1_s  # signed
```

3.6.2 Pitfalls

Pitfall 1: wrong signed/unsigned conditional jump after `cmp`.

```
cmp     eax, ebx
jl      .Lless_s         # signed less
jb      .Lbelow_u        # unsigned below
```

If the data is logically unsigned (sizes, indices, lengths), use unsigned jumps (`jb/ja/jbe/jae`). If the data is logically signed, use signed jumps (`jl/jg/jle/jge`).
Pitfall 2: assuming `test` can replace ordering comparisons. `test` cannot answer "is a ¡ b?" because AND does not encode ordering.
Pitfall 3: clobbering flags between producer and consumer.

```
cmp     eax, ebx
add     ecx, 1          # overwrites flags
je      .Lequal         # now tests flags from add (bug)
```

Pitfall 4: confusing "bit is set" with "value is non-zero" when masking.

```
test    eax, 0x80
jne     .Lmsb_set       # means bit7 set, not that eax is non-zero
```

Pitfall 5: partial-register issues in manual code reading. When `test` targets `al`/`ax`/`eax`, it tests only that width. Ensure you understand which bits are being tested.

```
test    al, al          # tests low 8 bits only
je      .Llow8_zero     # does NOT imply full rax == 0
```

3.6.3 High-signal mini catalog: common `jcc` pairs

```
# equality (from cmp or test)
je      label           # ZF==1
jne     label           # ZF==0

# unsigned (from cmp/sub)
jb      label           # CF==1   below
jae     label           # CF==0   above/equal
ja      label           # CF==0 and ZF==0  above
jbe     label           # CF==1 or  ZF==1  below/equal

# signed (from cmp/sub)
```

```
jl      label           # SF!=OF  less
jge     label           # SF==OF  greater/equal
jg      label           # ZF==0 and SF==OF greater
jle     label           # ZF==1 or  SF!=OF less/equal

# sign and zero after test/logic
js      label           # SF==1
jns     label           # SF==0
```

Chapter 4

Conditional Jumps: The Real Logic

4.1 Anatomy of a Conditional Jump Instruction

A conditional jump (`jcc`) is a control-transfer instruction whose decision is based entirely on the state of the status flags in RFLAGS/EFLAGS. A `jcc` does not compare operands. It does not read general-purpose registers (except indirectly through earlier flag-setting). It evaluates a boolean predicate over flags and either:

- **takes the branch**: IP/RIP becomes the target address, or
- **falls through**: IP/RIP advances to the next sequential instruction.

Architecturally, the branch target for near conditional jumps is encoded relative to the next instruction (a signed displacement). This makes `jcc` position-friendly and naturally suited for basic blocks.
Minimal structure:

```
cmp     eax, ebx        # producer: sets flags
je      .Leq            # consumer: tests ZF
```

```
# fall-through path
; (do something)
jmp     .Ldone          # optional join

.Leq:
; (equal-path code)
.Ldone:
ret
```

Control flow has two paths that must be read explicitly:

- the **taken path** is the jump target block,
- the **not-taken path** is the fall-through block.

4.2 Signed vs Unsigned Comparisons Explained

The CPU does not store signedness. It stores bit patterns and sets flags according to fixed arithmetic rules. Signed vs unsigned comparison is a matter of interpretation chosen by the **conditional jump mnemonic**:

- **Unsigned ordering** uses **CF** (and sometimes ZF) from `a - b`.
- **Signed ordering** uses the relationship between **SF** and **OF** (and sometimes ZF).

After `cmp a, b` (conceptually `a - b`):

- `CF=1` indicates an *unsigned borrow*, meaning `a < b` as unsigned.
- `SF!=OF` indicates *signed less-than*, meaning `a < b` as signed.

A single bit pattern can be both large unsigned and negative signed:

```
mov     al, 0xFF        # 255 unsigned, -1 signed (int8)
cmp     al, 1
jb      .Lbelow_u       # unsigned: 255 < 1  -> false (not taken)
jl      .Lless_s        # signed:  -1 < 1    -> true  (taken)
```

Practical rule:

- indices, sizes, counts, lengths are typically **unsigned** decisions
- arithmetic values that can be negative are typically **signed** decisions

4.3 Mapping Conditions to Flags

Conditional jumps are best learned as flag predicates. The most common groups:

4.3.1 `je` / `jne` (Equality)

Equality is sign-independent and driven by ZF.

```
cmp     eax, ebx
je      .Lequal         # ZF==1
jne     .Lnotequal      # ZF==0

test    rdi, rdi
je      .Lis_zero       # also ZF==1 (zero check)
```

4.3.2 `jl` / `jg` (Signed ordering)

Signed comparisons use SF and OF (and ZF for strictness).

- `jl`: taken if `SF != OF` (signed less)
- `jg`: taken if `ZF==0 and SF==OF` (signed greater)

```
cmp     eax, ebx
jl      .Lless_s        # signed: eax < ebx
jge     .Lge_s          # signed: eax >= ebx

cmp     eax, ebx
jg      .Lgreater_s     # signed: eax > ebx
jle     .Lle_s          # signed: eax <= ebx
```

4.3.3 `jb` / `ja` (Unsigned ordering)

Unsigned comparisons use CF (and ZF for inclusive conditions).

- `jb`: taken if `CF==1` (unsigned below)
- `ja`: taken if `CF==0 and ZF==0` (unsigned above)

```
cmp     eax, ebx
jb      .Lbelow_u       # unsigned: eax < ebx
jae     .Lge_u          # unsigned: eax >= ebx

cmp     eax, ebx
ja      .Labove_u       # unsigned: eax > ebx
jbe     .Lbe_u          # unsigned: eax <= ebx
```

4.3.4 Compact catalog (high-signal)

```
# equality
je      label           # ZF==1
jne     label           # ZF==0

# unsigned (from cmp/sub)
jb      label           # CF==1
jae     label           # CF==0
```

```
ja      label           # CF==0 and ZF==0
jbe     label           # CF==1 or  ZF==1

# signed (from cmp/sub)
jl      label           # SF!=OF
jge     label           # SF==OF
jg      label           # ZF==0 and SF==OF
jle     label           # ZF==1 or  SF!=OF
```

4.4 Why Signedness Is Not in the Instruction

x86 conditional jumps are designed to be minimal consumers of state: they test flags only. The flags were produced by a fixed arithmetic definition (e.g., subtraction for `cmp`). That definition is independent of signedness.
Signedness is a semantic choice made by software:

- a language decides whether a variable is signed/unsigned,
- the compiler chooses the corresponding conditional jump mnemonic,
- the CPU simply executes the mnemonic's flag predicate.

This separation is why the same `cmp` can feed either a signed or unsigned branch without changing the compare itself:

```
cmp     eax, ebx
jl      .Lless_s        # signed interpretation
jb      .Lbelow_u       # unsigned interpretation
```

4.5 Common Bugs from Wrong Jump Selection

Most control-flow bugs in low-level code reduce to one of these errors:

4.5.1 Bug 1: Using signed jumps for unsigned quantities

Lengths, indices, sizes, and pointer-related offsets are typically unsigned. Using `jl/jg` can break logic when the high bit is set.

```
cmp     edi, esi
jl      .Lidx_lt        # bug if idx is unsigned (should be jb)
```

4.5.2 Bug 2: Using unsigned jumps for signed quantities

Values that can be negative must use signed jumps. Otherwise negative numbers appear “large” and comparisons invert.

```
cmp     eax, 0
ja      .Lpositive      # bug for signed test (should be jg or jns pattern)
```

Correct signed positivity checks:

```
cmp     eax, 0
jg      .Lpositive      # signed: eax > 0

test    eax, eax
js      .Lnegative      # signed: SF==1
```

4.5.3 Bug 3: Clobbering flags between `cmp/test` and `jcc`

```
cmp     eax, ebx
add     ecx, 1          # overwrites flags
je      .Lequal         # now tests ZF from add (bug)
```

4.5.4 Bug 4: Wrong width (partial-register decisions)

Branching on `al` does not reflect full `rax`.

```
test    al, al
je      .Llow8_zero     # does not imply rax==0
```

4.5.5 Bug 5: Wrong condition form (strict vs inclusive)

Confusing `ja` with `jae`, or `jg` with `jge`, changes boundary behavior.

```
cmp     eax, ebx
ja      .Labove_u       # strictly >
jae     .Laboveeq_u     # >=
```

4.6 Reading Conditions Like the CPU Does

To read a condition correctly, follow a mechanical procedure:

- Identify the conditional jump instruction (`jcc`) and write down its flag predicate.
- Find the most recent instruction that definitively sets the needed flags (`cmp`, `test`, arithmetic, logic).
- Ensure no intervening instruction overwrites those flags.
- Determine operand width (8/16/32/64) to know which bits drive SF/ZF and which arithmetic width defines CF/OF.
- Decide whether the branch is signed or unsigned based on the `jcc` mnemonic, not on assumptions.

Worked reading example 1 (unsigned bounds check + jump table):

```
cmp     edi, 7          # sets CF/ZF based on edi - 7
ja      .Ldefault       # unsigned: taken if edi > 7  (CF==0 and ZF==0)
jmp     qword ptr [rip + .Ljt + rdi*8]
```

Worked reading example 2 (signed less-than):

```
cmp     eax, ebx
jl      .Lless_s        # taken if SF!=OF (signed eax < ebx)
# fall-through implies signed eax >= ebx
```

Worked reading example 3 (bit test decision):

```
test    edx, 0x20       # checks whether bit5 is set
je      .Lbit5_clear    # taken if (edx & 0x20)==0
```

This CPU-style reading discipline scales: every complex high-level construct reduces to these local flag predicates and control transfers.

Chapter 5

Control Flow Patterns Built from Jumps

5.1 If / Else at the Assembly Level

High-level `if/else` lowers to a conditional jump that selects between two basic blocks, plus an optional join block. There are two canonical shapes:

Shape A: branch over the then-block (fall-through is "then").

```
# if (x == 0) then A else B
test    edi, edi        # ZF=1 if x==0
jne     .Lelse          # if x!=0 jump to else
# then-block A (fall-through)
call    A
jmp     .Ljoin
.Lelse:
call    B
.Ljoin:
ret
```

Shape B: branch over the else-block (fall-through is "else"). This form is common when the "then" block is cold or larger.

```
# if (x != 0) then A else B
test    edi, edi
je      .Lelse
call    A               # then-block
jmp     .Ljoin
.Lelse:
call    B               # else-block
.Ljoin:
ret
```

A third variant removes the explicit join jump by using fall-through placement:

```
# if (cond) A; B;  (B always executes)
cmp     eax, ebx
jne     .LskipA
call    A
.LskipA:
call    B
ret
```

5.2 Nested Conditions

Nested `if` statements become a sequence of conditional branches. Conceptually, each branch filters execution into a narrower path.

Nested: if (c1) { if (c2) A; else B; } else C;

```
# c1
test    edi, edi
je      .LC

# c2 (only evaluated if c1 true)
cmp     esi, 10
jl      .LB
```

```
call    A
jmp     .Ljoin

.LB:
call    B
jmp     .Ljoin

.LC:
call    C

.Ljoin:
ret
```

Common compiler optimization: invert conditions to minimize jumps and keep the most likely path as fall-through.

```
# prefer fall-through for likely path:
test    edi, edi
je      .Lcold_path     # cold path out-of-line
# hot path continues here
cmp     esi, 10
jge     .Lhot_A
# still hot
call    B
ret
.Lhot_A:
call    A
ret
.Lcold_path:
call    C
ret
```

5.3 Short-Circuit Logic (AND / OR)

Short-circuit semantics are naturally implemented with conditional jumps because evaluation stops as soon as the result is determined.

5.3.1 AND short-circuit: `if (a && b) A else B`

For AND: if `a` is false, skip evaluating `b` and go directly to else.

```
# a
test    edi, edi
je      .Lelse          # if a==0 -> else

# b
test    esi, esi
je      .Lelse          # if b==0 -> else

call    A               # (a && b) true
jmp     .Ljoin
.Lelse:
call    B
.Ljoin:
ret
```

5.3.2 OR short-circuit: `if (a || b) A else B`

For OR: if `a` is true, do not evaluate `b`.

```
# a
test    edi, edi
jne     .Lthen          # if a!=0 -> then

# b
```

```
test    esi, esi
jne     .Lthen          # if b!=0 -> then

call    B               # both false
jmp     .Ljoin

.Lthen:
call    A
.Ljoin:
ret
```

5.3.3 Mixed short-circuit: `if (a && (b || c))`

```
# a must be true
test    edi, edi
je      .Lfalse

# (b || c)
test    esi, esi
jne     .Ltrue
test    edx, edx
jne     .Ltrue

.Lfalse:
call    F               # false path
ret

.Ltrue:
call    T               # true path
ret
```

5.4 Boolean Expressions Without Boolean Types

The CPU has no boolean type. Conditions are represented through:

- flags (ZF/SF/CF/OF) consumed by `jcc`,
- integer conventions (0 = false, non-zero = true),
- bit masks (specific bits encode condition state).

Boolean from comparison (classic):

```
cmp     eax, ebx        # sets ZF
sete    al              # al := (ZF==1) ? 1 : 0
movzx   eax, al         # zero-extend to int
```

Boolean from non-zero test:

```
test    rdi, rdi        # sets ZF based on rdi
setne   al              # al := (rdi != 0)
movzx   eax, al
```

Boolean from unsigned ordering:

```
cmp     eax, ebx
setb    al              # al := (unsigned eax < ebx)
movzx   eax, al
```

Boolean used directly without materializing a 0/1 value:

```
test    eax, eax
je      .Lfalse
# true-path (eax != 0)
```

Boolean as bitmask state:

```
# if (flags & 0x20) ...
test    edx, 0x20
je      .Lbit_clear
# bit set path
```

5.5 Compiler Reordering of Conditions

Compilers may reorder condition evaluation when semantics permit, primarily to:

- reduce the number of jumps,
- keep hot paths as fall-through,
- merge identical exit paths,
- eliminate redundant tests (common-subexpression elimination on conditions).

5.5.1 Branch inversion and fall-through shaping

A source-level `if (cond) A else B` can become either `jcc else` or `jcc then` depending on layout goals.

```
# form 1
test    edi, edi
je      .Lelse
call    A
jmp     .Ljoin
.Lelse:
call    B
.Ljoin:
ret

# form 2 (inverted)
test    edi, edi
jne     .Lthen
call    B
jmp     .Ljoin
.Lthen:
```

```
call    A
.Ljoin:
ret
```

5.5.2 Early-exit normalization

Many nested conditions compile into a series of early exits:

```
# if (!p) return;
# if (len == 0) return;
test    rdi, rdi
je      .Lret
test    esi, esi
je      .Lret
# work
call    Work
.Lret:
ret
```

5.5.3 Combining condition checks

Two checks can be merged when they target the same outcome:

```
# if (x==0 || y==0) fail;
test    edi, edi
je      .Lfail
test    esi, esi
je      .Lfail
# success path
call    OK
ret
.Lfail:
call    Fail
ret
```

5.5.4 Reordering under short-circuit constraints

Short-circuit order is preserved when observable side effects exist. When operands are pure tests (no side effects), compilers may choose an equivalent order to improve code shape, but the key property remains: AND must fail fast, OR must succeed fast.
Reading discipline:

- treat conditional jumps as the truth source for condition meaning,
- reconstruct the boolean structure from branch edges and fall-through,
- identify shared join blocks and early exits.

Chapter 6

Loop Patterns in x86

6.1 Loop Concepts Without High-Level Languages

A loop is a control-flow structure that causes a basic block (or a region of blocks) to execute repeatedly. At the ISA level, a loop is defined by:

- a **loop header** (entry point),
- a **back-edge** (a jump that returns execution to the header),
- an **exit condition** that eventually prevents taking the back-edge.

Two fundamental loop questions exist in machine code:

- **When do we exit?** (which flag predicate ends the repetition)
- **How do we make progress?** (which register/memory state changes each iteration)

In x86, the CPU does not have a loop construct; it has conditional and unconditional jumps. Loops are built by arranging: `flag producer` → `jcc back-edge or exit`.
Minimal infinite loop (no exit condition):

```
.Lloop:
jmp     .Lloop
```

Minimal conditional loop (exit check at top):

```
.Lhead:
test    edi, edi
je      .Lexit
# body
dec     edi
jmp     .Lhead
.Lexit:
ret
```

6.2 `dec` + `jnz` Pattern

The most recognizable counted loop at the assembly level is: `dec reg` followed by `jnz label`. `dec` decrements the register and sets ZF if the result becomes zero. `jnz` loops while ZF==0.

Canonical form:

```
mov     ecx, 10
.Lloop:
# body
dec     ecx             # sets ZF when ecx becomes 0
jnz     .Lloop           # branch back while ecx != 0
```

Key properties:

- Progress is explicit: the counter moves toward zero.
- The exit condition is zero: loop stops when counter becomes 0.
- The back-edge is the `jnz`.

Common variation using `sub`:

```
sub     ecx, 1
jne     .Lloop
```

Common variation using `add` with negative step:

```
add     ecx, -1
jne     .Lloop
```

6.3 Count-Controlled Loops

Count-controlled loops repeat a known number of iterations or until a counter reaches a boundary. Compilers typically implement these using a counter register and a comparison against an end value.

6.3.1 Down-counter to zero

This is the `dec/jnz` family (or `sub/jne`):

```
mov     ecx, n
test    ecx, ecx
je      .Lexit
.Lloop:
# body
dec     ecx
jnz     .Lloop
.Lexit:
ret
```

6.3.2 Up-counter with compare

Very common for array indexing:

```
xor     eax, eax          # i = 0
.Lloop:
# body uses i in eax
inc     eax
cmp     eax, esi          # compare i with n
jl      .Lloop            # signed i < n (works if n is non-negative)
```

For unsigned lengths (typical for sizes), compilers commonly use unsigned conditions:

```
xor     eax, eax          # i = 0
.Lloop:
# body
inc     eax
cmp     eax, esi
jb      .Lloop            # unsigned i < n
```

6.3.3 Pointer-walking count loop

Instead of an index, a pointer advances:

```
mov     rdi, base         # p = base
lea     rsi, [base + rdx]# end = base + size
.Lloop:
cmp     rdi, rsi
jae     .Lexit            # unsigned p >= end
# body uses [rdi]
add     rdi, 1
jmp     .Lloop
.Lexit:
ret
```

6.3.4 Do-while shape (check at bottom)

```
.Lloop:
```

```
# body executes at least once
dec     ecx
jnz     .Lloop
```

6.4 Condition-Controlled Loops

Condition-controlled loops are driven by a predicate that depends on data, not just a simple iteration count (e.g., search until match, parse until terminator, iterate while condition holds).

6.4.1 While loop (check at top)

```
.Lhead:
# condition
cmp     eax, ebx
jge     .Lexit          # while (eax < ebx) ...
# body
add     eax, 1
jmp     .Lhead
.Lexit:
ret
```

6.4.2 Do-while loop (check at bottom)

```
.Lbody:
# body
add     eax, 1
cmp     eax, ebx
jl      .Lbody          # do { ... } while (eax < ebx)
ret
```

6.4.3 Search loop (break on match)

```
mov     rdi, base
lea     rsi, [base + rdx]     # end
.Lloop:
cmp     rdi, rsi
jae     .Lnot_found
mov     al, byte ptr [rdi]
cmp     al, cl
je      .Lfound
inc     rdi
jmp     .Lloop

.Lfound:
# rdi points to match
ret

.Lnot_found:
# not found
ret
```

6.4.4 Sentinel-driven loop (terminate on zero byte)

```
mov     rdi, s
.Lloop:
mov     al, byte ptr [rdi]
test    al, al
je      .Lend
inc     rdi
jmp     .Lloop
.Lend:
ret
```

6.5 Why the `loop` Instruction Is Rarely Used

x86 includes the `loop` family (`loop`, `loope/loopz`, `loopne/loopnz`), which decrements `(E)CX` and branches if the condition holds. Despite being a single mnemonic for "decrement and branch", modern compilers rarely emit it for general code.
Practical reasons in modern toolchains:

- **Limited register choice**: it is tied to `(E)CX`, while compilers prefer flexible register allocation.
- **Predictable canonical forms**: compilers standardize on `dec/jnz` or `sub/jne` because these patterns are universally supported, easy to schedule, and interact predictably with surrounding code.
- **Better composition**: separate decrement and branch instructions allow more freedom for instruction scheduling and for inserting other operations without changing semantics.

Typical form compilers prefer:

```
dec     ecx
jnz     .Lloop
```

What `loop` looks like when encountered:

```
mov     ecx, 10
.Lloop:
# body
loop    .Lloop            # ecx := ecx - 1; if ecx != 0 then branch
```

6.6 Recognizing Compiler-Generated Loops

To recognize a loop in disassembly, search for a **back-edge**: a jump to a lower-address label (often earlier in the function) or a jump to a label that dominates the loop body.
Practical recognition checklist:

- Identify a label that is the target of a backward `jmp/jcc`.
- Find the **counter or condition** updated each iteration.
- Identify the flag producer (`cmp/test/dec/add/sub`) paired with the loop-controlling `jcc`.
- Determine whether the loop is count-controlled (counter vs bound) or condition-controlled (predicate from data).
- Check for early exits: conditional jumps to an exit label inside the loop body.

Common compiler loop shapes:

6.6.1 Counted loop with compare at bottom

```
xor     eax, eax          # i = 0
.Lloop:
# body
inc     eax
cmp     eax, esi
jl      .Lloop
```

6.6.2 Counted loop with pre-check for zero iterations

```
test    esi, esi
je      .Lexit
```

```
xor     eax, eax
.Lloop:
# body
inc     eax
cmp     eax, esi
jl      .Lloop
.Lexit:
ret
```

6.6.3 Loop with internal break

```
.Lloop:
# body
cmp     eax, 0
je      .Lbreak
# continue path
add     edx, 1
jmp     .Lloop
.Lbreak:
ret
```

6.6.4 Unsigned bounds loop (typical for sizes and pointers)

```
cmp     rdi, rsi
jae     .Lexit
# body
add     rdi, 4
jmp     .Ltop
```

Reading discipline for loops:

- Translate each `jcc` into a flag predicate.

- Reconstruct the loop condition and exit condition from the branch direction (back-edge vs exit).
- Decide signed vs unsigned based on the `jcc` mnemonic, not on assumptions.

Chapter 7

Indirect Jumps and Jump Tables

7.1 What an Indirect Jump Really Is

A direct jump encodes its target in the instruction (typically as a relative displacement). An **indirect jump** does not. It obtains the target address from a register or from memory and then sets IP/RIP to that value. In other words, the destination is **data-dependent** at runtime.
Forms (Intel syntax):

- `jmp reg`
- `jmp [mem]`
- `jmp qword ptr [base + index*scale + disp]`

Minimal examples:

```
jmp     rax                         # RIP := rax

jmp     qword ptr [rbx]             # RIP := *rbx

jmp     qword ptr [r11 + rdi*8]     # RIP := *(r11 + rdi*8)
```

Indirect jumps are fundamental for:

- jump tables for `switch`-like dispatch,
- dynamic dispatch stubs and trampolines,
- computed gotos and state machines,
- return instructions (`ret`) as an indirect control transfer (target from stack).

7.2 Jump Tables as a Control Flow Optimization

A jump table is a contiguous array of code addresses (or relative offsets) used to implement multi-way branching efficiently. Instead of multiple comparisons and branches, dispatch becomes:

- optional bounds check,
- compute table entry address,
- indirect jump to the selected target.

High-level intent:

- reduce the number of conditional branches,
- provide near-constant dispatch time for dense case ranges.

Canonical shape:

```
cmp     edi, MAX_CASE
ja      .Ldefault
jmp     qword ptr [rip + .Ljt + rdi*8]
```

7.3 Switch Statements at the Assembly Level

Compilers lower a `switch` using one of several strategies depending on case density and range:

- **jump table** for dense ranges,
- **binary search / decision tree** for sparse sets,
- **linear chain** for small numbers of cases.

7.3.1 Linear chain (small number of cases)

```
cmp     edi, 1
je      .Lcase1
cmp     edi, 3
je      .Lcase3
jmp     .Ldefault

.Lcase1:
# ...
jmp     .Lend
.Lcase3:
# ...
jmp     .Lend
.Ldefault:
# ...
.Lend:
ret
```

7.3.2 Decision tree (sparse cases)

```
cmp     edi, 100
```

```
jl      .Llow
je      .Lcase100
cmp     edi, 1000
je      .Lcase1000
jmp     .Ldefault

.Llow:
cmp     edi, 7
je      .Lcase7
jmp     .Ldefault
```

7.3.3 Jump table (dense range)

```
# switch(x) with cases 0..7
cmp     edi, 7
ja      .Ldefault
jmp     qword ptr [rip + .Ljt + rdi*8]

.Lcase0:
# ...
ret
.Lcase1:
# ...
ret
# ...
.Lcase7:
# ...
ret
.Ldefault:
# ...
ret
```

7.4 Address Computation for Jump Tables

On x86-64, jump tables are commonly addressed using RIP-relative addressing so code remains position-independent. The base of the table is computed implicitly by the addressing mode, and the index selects an entry using scaled indexing.
A typical table entry size is 8 bytes when storing absolute 64-bit addresses, hence `index*8`.

```
# rdi holds the case index
jmp     qword ptr [rip + .Ljt + rdi*8]   # target := table[rdi]
```

Another common pattern uses an explicit base register:

```
lea     r11, [rip + .Ljt]               # r11 = &table[0]
jmp     qword ptr [r11 + rdi*8]         # jump to table[rdi]
```

Some toolchains use tables of 32-bit relative offsets instead of full addresses, then add the base:

```
lea     r11, [rip + .Ljt]               # base
movsxd  rax, dword ptr [r11 + rdi*4]    # sign-extend 32-bit offset
add     rax, r11                        # absolute target = base + offset
jmp     rax
```

The offset-table form reduces table size (4 bytes per entry) and is common when targets are within a nearby code region.

7.5 Bounds Checking Patterns

Because an indirect jump is data-dependent, correct jump-table dispatch requires preventing out-of-range indices. Compilers emit bounds checks before the indirect jump.

7.5.1 Unsigned upper-bound check (most common)

For indices, unsigned checks are standard. If `x > MAX` then default:

```
cmp     edi, 7
ja      .Ldefault       # unsigned: edi > 7
jmp     qword ptr [rip + .Ljt + rdi*8]
```

7.5.2 Range normalization (shift to start at 0)

For cases like 10..15, compilers normalize:

```
sub     edi, 10         # x -= 10
cmp     edi, 5
ja      .Ldefault       # if (x-10) > 5 then default
jmp     qword ptr [rip + .Ljt + rdi*8]
```

7.5.3 Two-sided bounds check (when needed)

If inputs might be negative but treated as signed, a compiler may guard both sides explicitly:

```
cmp     edi, 0
jl      .Ldefault       # signed: x < 0
cmp     edi, 7
jg      .Ldefault       # signed: x > 7
jmp     qword ptr [rip + .Ljt + rdi*8]
```

In practice, many compilers avoid signed two-sided checks by converting the index into an unsigned range via normalization and then using a single unsigned bound check.

7.6 Security and Correctness Considerations

Indirect jumps are powerful and dangerous because the target comes from data. Correctness and security rely on controlling that data.

7.6.1 Correctness hazards

- **Missing bounds checks**: an out-of-range index reads a table entry beyond the intended region and jumps to an unintended address.
- **Wrong scaling**: using `index*4` for an 8-byte table (or vice versa) produces misaligned targets and invalid jumps.
- **Wrong width**: using a 32-bit index where a 64-bit address computation is needed can accidentally truncate or mis-compute addresses.
- **Signedness mismatch**: using signed comparisons for an index can allow negative values to pass if interpreted incorrectly.

Scaling pitfall example:

```
# bug if table entries are 8 bytes but scale uses *4
jmp     qword ptr [rip + .Ljt + rdi*4]   # wrong: reads wrong half-entry
↪  addresses
```

7.6.2 Security hazards

- **Control-flow hijack potential**: if an attacker can influence the table base or index, they can redirect execution.
- **Return is an indirect transfer**: `ret` loads the target from memory (stack). Stack corruption can redirect control flow.
- **Speculative effects**: even with architectural bounds checks, unsafe table access patterns can still be risky if the index is not properly constrained before use in address computation. The safe design principle is to ensure indices are validated before they influence any control-transfer target computation.

Return as indirect control transfer:

```
ret                                     # RIP := pop()   (target is
↪  memory-resident)
```

Robust jump-table pattern (normalize + unsigned bound check + indirect jump):

```
sub     edi, BASE_CASE                  # normalize so first case becomes 0
cmp     edi, MAX_INDEX
ja      .Ldefault
jmp     qword ptr [rip + .Ljt + rdi*8]
```

Reading discipline:

- identify the index, normalization, and bounds check,
- confirm the table base computation (RIP-relative or register base),
- confirm entry size and scaling,
- confirm the indirect jump consumes the computed address as intended.

Chapter 8

`call` Instruction Internals

8.1 What `call` Really Does

At the ISA level, `call` is a control-transfer instruction that performs two actions atomically as one architectural operation:

- it computes the **return address** (the address of the instruction immediately following the `call`),
- it transfers control to the **target** by updating IP/RIP.

Unlike `jmp`, `call` preserves a return point so that `ret` can later resume execution. Conceptually, `call` is:

`push(return_address); jmp(target)`

Minimal example:

```
call    func            # push return address; RIP := func
# execution resumes here after func executes ret
```

```
mov     eax, 1

func:
xor     eax, eax
ret
```

Key idea: the CPU does not know “functions”. It only knows control transfer plus a saved return address on the stack.

8.2 Pushing the Return Address

The return address pushed by a near `call` is the **next instruction pointer** value. In 64-bit mode, the stack pointer (`RSP`) is decremented by 8 and the 8-byte return address is written at `[RSP]`. In 32-bit mode, `ESP` is decremented by 4.
64-bit conceptual effect:

```
# before: RSP = S
call    func
# after:
#   RSP = S - 8
#   [RSP] = address(next instruction)
#   RIP = address(func)
```

You can observe the saved return address by reading `[rsp]` immediately after a call target begins, before it changes the stack further.
Example: first instruction in callee reads the return address:

```
func:
mov     rax, qword ptr [rsp]    # rax := return address
# ... then usual function work
ret
```

Important consequences:

- **Control integrity depends on stack integrity**: if the stored return address is corrupted, `ret` will jump to the wrong place.
- **Nested calls create a return chain**: each call pushes another return address, forming a LIFO structure.

Nested call chain shape:

```
call    A
# ...

A:
call    B
ret

B:
ret
```

8.3 Near vs Far Calls (Conceptual)

x86 defines **near** and **far** calls:

- **Near call**: changes IP/RIP within the current code segment context.
- **Far call**: changes both IP and the code segment selector, transferring to a different segment context (historically used with segmentation and privilege transitions in certain environments).

In modern 64-bit user-mode code, near calls dominate. Far calls exist architecturally but are uncommon in typical application code generation. For this booklet's scope, treat far calls as a conceptual category that explains why the ISA includes encodings beyond the usual near `call`.

Near call forms you will actually see in disassembly:

- `call rel32` (direct relative)
- `call r/m64` (indirect via register or memory)

8.4 Direct vs Indirect Calls

A **direct call** encodes the target in the instruction stream (relative displacement). An **indirect call** obtains the target address from a register or memory at runtime.

8.4.1 Direct (relative) call

The common form in position-friendly code is relative-to-next-instruction:

```
call    func            # encoded as relative displacement to func
```

8.4.2 Indirect call through register

Used for function pointers, virtual calls, trampolines, and dynamic dispatch:

```
mov     rax, qword ptr [rdi]    # rax = function pointer
call    rax                     # push return address; RIP := rax
```

8.4.3 Indirect call through memory

Used when the pointer is stored in memory and called directly:

```
call    qword ptr [rip + fp]    # target loaded from memory, then call
```

8.4.4 Virtual-dispatch style shape (conceptual)

This is the characteristic pattern where a pointer is loaded then called:

```
mov     rax, qword ptr [rdi]         # load vptr-like base
mov     rax, qword ptr [rax + 16]    # load method pointer
call    rax
```

8.5 How Compilers Choose Call Forms

Compilers select call forms based on what is known about the target at compile time and how the code must relocate.

8.5.1 Known symbol target

If the callee is a known function symbol, the compiler prefers a **direct relative call** because it is compact and naturally supports position-relative encoding.

```
call    known_function
```

8.5.2 Unknown target at runtime

If the call target is computed (function pointer, callback, dispatch table), the compiler must use an **indirect call**.

```
call    rax
```

8.5.3 Position-independent and relocation-friendly shapes

Even for known symbols, toolchains may route calls through indirection mechanisms in some builds (for example through a stub or table entry) to support dynamic linking and late binding. At the ISA level, what you observe is still either a direct relative call or an indirect call via a memory-loaded pointer.
Two observable patterns:

```
# direct relative call
call    func

# indirect call via memory-resident entry
call    qword ptr [rip + entry]
```

8.5.4 Inlining eliminates calls

When profitable and legal, compilers may remove the call entirely by inlining the callee body. In disassembly, the absence of a `call` does not mean "no function" in the source; it may simply mean the call was replaced by straight-line code.

8.5.5 Reading discipline for calls

When you see a call:

- identify whether it is direct (relative) or indirect (register/memory),
- identify where the target comes from if indirect,
- remember the return address is pushed automatically,
- verify the next control-transfer after callee completes is typically `ret`.

Quick recognition examples:

```
call    0x401000                # direct, absolute shown by disassembler
↪  (still relative encoding)
call    func                    # direct symbol
call    rax                     # indirect via register
call    qword ptr [rdi + 8]     # indirect via memory
```

Chapter 9

`ret` Instruction and Returning Control

9.1 Stack-Based Return Mechanism

The x86 return mechanism is stack-based. A `call` transfers control to a target while saving a **return address** on the stack. A `ret` returns by restoring the instruction pointer from that saved address.
The architectural model is strict LIFO:

- each `call` pushes one return address,
- each `ret` pops one return address,
- returns occur in reverse order of calls.

Minimal call-return chain:

```
call    A
# resumes here after A returns
ret
```

```
A:
call    B
ret

B:
ret
```

9.2 How `ret` Uses the Stack

A near `ret` is an **indirect control transfer** whose target is loaded from memory at `[SP]` and placed into IP/RIP. Then SP is incremented to remove the popped return address.
Conceptually, in 64-bit mode:

```
RIP := [RSP]; RSP := RSP + 8
```

In 32-bit mode:

```
EIP := [ESP]; ESP := ESP + 4
```

This can be modeled as a pop into the instruction pointer:

```
ret                     # RIP := pop()
```

You can observe the mechanism by explicitly popping into a general register, then jumping:

```
pop     rax             # rax := return address
jmp     rax             # similar control transfer to ret
```

A callee can also read the return address without returning yet:

```
mov     rax, qword ptr [rsp]    # rax := saved return address (top of
↪  stack)
```

9.3 `ret imm16` and Stack Cleanup

x86 defines a form `ret imm16` (near return with immediate). It pops the return address into IP/RIP and then additionally increments SP by an immediate byte count.
Conceptually in 64-bit mode:

```
RIP := [RSP]; RSP := RSP + 8 + imm16
```

In 32-bit mode:

```
EIP := [ESP]; ESP := ESP + 4 + imm16
```

This exists to support conventions where the callee removes argument bytes from the stack. While common historically in some 32-bit conventions, it is uncommon in typical 64-bit System V and Windows x64 calling convention usage where stack argument cleanup rules are different and argument passing is primarily register-based.
Illustration of the effect (conceptual):

```
# callee returns and discards 16 bytes of caller-provided stack arguments
ret     16
```

Equivalently observable as:

```
pop     rax             # pop return address
add     rsp, 16         # discard argument area
jmp     rax             # return
```

Reading discipline:

- treat `ret imm16` as: return + extra stack pointer adjustment,
- never assume it is “optional”: it changes stack layout expectations.

9.4 What Happens If the Stack Is Corrupted

Because `ret` loads its destination from memory, a corrupted stack means `ret` transfers control to an unintended address. This is not an "exception" at the ISA level; it is simply a jump to whatever value is at the top of the stack.
Common corruption sources:

- writing beyond a local stack object (overflow),
- mis-matched push/pop sequences,
- incorrect manual stack-pointer adjustment (`add rsp, ...` / `sub rsp, ...`),
- returning with a different SP than the one used on entry.

Simple example: mismatch of pushes and pops:

```
func:
push    rbx
# ... body ...
ret                     # bug: return address is not on top (rbx value is)
```

Correct form:

```
func:
push    rbx
# ... body ...
pop     rbx
ret
```

Example: wrong stack adjustment:

```
sub     rsp, 32         # reserve space
# ... use [rsp]..[rsp+31] ...
add     rsp, 24         # bug: should restore 32
ret                     # ret reads wrong address
```

Example: attacker-influenced return target (conceptual hazard):

```
# if [rsp] is overwritten, ret jumps to overwritten value
ret
```

Architecturally, the CPU does not validate that the return target is “reasonable”. It only uses the value provided by memory.

9.5 Why `ret` Is Simple but Dangerous

`ret` is among the simplest x86 control-transfer instructions: a single memory read plus a stack-pointer increment. That simplicity is exactly why it is dangerous:

- it is a control transfer whose target is data,
- the data source is a mutable memory region (the stack),
- small stack discipline errors have catastrophic control-flow consequences.

Practical implications for low-level reasoning:

- When reading code, always identify what value will be at `[rsp]` at the moment `ret` executes.
- Treat every stack adjustment (`push/pop/sub/add rsp`) as part of the control-flow correctness proof.
- Recognize that returns are a major boundary between correct execution and control-flow failure.

A minimal “return correctness checklist” in disassembly:

```
# 1) Was every pushed value popped (or accounted for) before ret?
# 2) Is rsp restored to the entry value (modulo the return address)?
# 3) Is there any instruction that overwrote stack memory near [rsp]?
# 4) Is ret imm16 used, and does it match the expected stack arguments?
ret
```

This booklet treats `ret` as a control-flow primitive. Later booklets extend this understanding into full calling conventions, stack frames, and ABI rules, but the architectural danger remains the same: `ret` jumps wherever the stack tells it to.

Chapter 10

Call / Return Flow as a System

10.1 Control Flow vs Data Flow During Calls

A function call is simultaneously a control-flow event and a data-flow event.
Control flow: `call` changes IP/RIP to the callee and saves the return address; `ret` restores IP/RIP from the stack. This creates a strict edge in the control-flow graph:

$$\texttt{caller} \rightarrow \texttt{callee} \rightarrow \texttt{caller\_next}$$

Data flow: arguments, return values, and temporary state move through registers and memory. Even without committing to a specific ABI in this booklet, two universal data-flow facts remain:

- a caller must place inputs where the callee expects them (often registers, sometimes memory),
- a callee must place the return value where the caller expects it (commonly a register).

Control-transfer minimal core:

```
call    func            # control: push return address; jump to func
# caller resumes here
```

Data-flow example (inputs and output as registers, conceptual):

```
mov     edi, 2          # input a
mov     esi, 3          # input b
call    add2            # returns result in eax (common convention)
# eax now holds result (data flow)

add2:
lea     eax, [rdi + rsi]
ret
```

Key reading rule: `call/ret` provide the control skeleton; register/memory moves provide the data contract. Mixing the two is a common analysis mistake (e.g., assuming `call` "passes" values by itself).

10.2 Nested Calls and Return Chains

Nested calls create a return chain stored on the stack. Every `call` pushes one return address. Therefore, a sequence of calls produces a stack of return targets:

$$\mathrm{RA}_1, \mathrm{RA}_2, \ldots, \mathrm{RA}_n$$

and `ret` pops them in reverse order.

Three-level nesting:

```
call    A
# resumes here after A returns
ret

A:
```

```
call    B
ret

B:
call    C
ret

C:
ret
```

Architectural view (64-bit conceptual):

```
# At entry of A:
#   [rsp]     = return address into caller
# After A calls B:
#   [rsp]     = return address into A
#   [rsp+8]   = return address into caller
```

Practical implication: the stack top always holds the *next* return target. If stack discipline is broken at any level, the whole chain collapses.

10.3 Visualizing Call Depth

Call depth is simply the number of active (not-yet-returned) calls. At the ISA level, depth correlates with how many return addresses have been pushed and not popped.
A minimal visualization technique for understanding depth in disassembly is to treat each `call` as "depth++" and each `ret` as "depth–". Even without drawing full stack frames, you can reconstruct call nesting.
Example with explicit depth commentary:

```
# depth = 0
call    A               # depth = 1
```

```
# depth = 0 after A returns
ret

A:
call    B               # depth = 2 (while inside B)
ret                     # depth back to 0 when A returns to caller

B:
call    C               # depth = 3 (while inside C)
ret                     # returns to B (depth = 2)
C:
ret                     # returns to B (depth = 2)
```

A practical debugging anchor in many environments is that return addresses form a chain that unwinding tools follow. Architecturally, that chain exists because `call` stores return targets on the stack.

10.4 Tail Calls (Conceptual Introduction)

A **tail call** occurs when a function ends by calling another function and immediately returning the result of that call. Conceptually:

- normal call: caller calls callee, then returns later
- tail call: caller transfers control to callee as its final action

At the machine-code level, a tail call often appears as a **jump** (`jmp`) to the callee instead of `call`, because no return to the current function is needed. The return address already on the stack (from this function's caller) can remain the one used by the final callee's `ret`.
Tail-call shape:

```
# tail-call style: no new return address pushed
jmp     target_func
```

Non-tail-call shape:

```
call    target_func
ret
```

Conceptual example:

```
F:
# ... prepare args for G ...
jmp     G               # tail call: G returns directly to F's caller

G:
# ... compute ...
ret
```

Important constraints (kept conceptual here):

- the call must be in the tail position (no work remains after it),
- the calling contract between caller and callee must be compatible enough to reuse the return path.

For this booklet, treat tail calls as a control-flow optimization that turns “call then return” into “jump”, changing the observable call/return structure.

10.5 Control Flow Integrity Basics

Control flow integrity is the principle that control transfers should only go to valid, intended targets. At the ISA level, the most sensitive transfers are **indirect**:

- `call reg` / `call [mem]`
- `jmp reg` / `jmp [mem]`

- `ret` (target from stack memory)

These instructions take their destination from data. Therefore, correctness and security depend on the integrity of that data.
Key architectural observations:

- `ret` is a memory-driven jump: `RIP := [RSP]` then `RSP += 8`.
- an indirect `call`/`jmp` is register/memory-driven: `RIP := target`.
- bounds checks and validation patterns in code exist precisely because targets can become unsafe when indices or pointers are uncontrolled.

Examples of indirect transfers:

```
call    rax                         # indirect call to address in rax
jmp     qword ptr [r11 + rdi*8]     # indirect jump via table
ret                                 # indirect return via stack
```

Basic correctness discipline for call/return integrity in low-level reasoning:

- ensure stack pointer restoration matches the pushes/pops performed,
- ensure indirect call targets originate from controlled sources,
- ensure jump-table indices are range-checked before indexing,
- ensure no instruction clobbers return addresses or target pointers.

Minimal illustration of why stack integrity matters:

```
# if [rsp] is corrupted, ret jumps to an unintended address
ret
```

In later booklets, these basics connect directly to ABI rules, stack frames, and hardened calling/return mechanisms. In this volume, the essential message is architectural: call/return forms a system whose correctness depends on preserving the return-address chain and controlling indirect targets.

Chapter 11

Compiler Control Flow Strategies

11.1 Branch Minimization

Modern compilers attempt to reduce the number of conditional branches when it is profitable and legal, because branches create multiple execution paths and impose control-flow overhead. At the ISA level, branch minimization usually appears as:

- turning small if/else logic into conditional data selection,
- merging multiple conditions that share an outcome,
- using jump tables for dense multi-way dispatch,
- replacing branches with flag-to-value materialization when a boolean value is needed.

11.1.1 Materialize a boolean instead of branching

Instead of:

```
cmp     eax, ebx
jne     .Lfalse
mov     eax, 1
jmp     .Lend
.Lfalse:
xor     eax, eax
.Lend:
```

Compilers often do:

```
cmp     eax, ebx
sete    al              # al := (eax==ebx)
movzx   eax, al
```

11.1.2 Merge conditions that share an exit

```
# if (!p || n==0) return;
test    rdi, rdi
je      .Lret
test    esi, esi
je      .Lret
# work path
call    Work
.Lret:
ret
```

11.1.3 Use a jump table for dense cases

```
cmp     edi, 7
ja      .Ldefault
jmp     qword ptr [rip + .Ljt + rdi*8]
```

11.2 Branch Inversion

Branch inversion is the systematic transformation:

```
if (cond) then A else B  ⟺  if (!cond) then B else A
```

Compilers invert branches to:

- make the most likely path fall-through,
- reduce the number of unconditional jumps,
- improve block layout for instruction cache locality.

Two equivalent shapes:

```
# form 1
test    edi, edi
je      .Lelse
call    A
jmp     .Ljoin
.Lelse:
call    B
.Ljoin:
ret

# form 2 (inverted)
test    edi, edi
jne     .Lthen
call    B
jmp     .Ljoin
.Lthen:
call    A
.Ljoin:
ret
```

Reading discipline: do not attach meaning to the mnemonic alone; attach meaning to the predicate and which block is fall-through.

11.3 Fall-Through Optimization

Fall-through optimization is the layout decision that places the most frequently executed successor block immediately after a conditional branch, so that when the branch is not taken, execution continues sequentially without an extra jump.
Typical shape: “guard then early exit” (hot path is fall-through).

```
test    rdi, rdi
je      .Lerror          # uncommon: jump away
# hot path continues here
call    FastPath
ret
.Lerror:
call    SlowPath
ret
```

Another shape: late join elimination by layout.

```
cmp     eax, ebx
jne     .Lskip
call    A                # only on equal
.Lskip:
call    B                # always executes
ret
```

Compilers also form diamond patterns (then/else) but attempt to place the join block directly after whichever path is likely to continue.

11.4 Common Patterns Emitted by Modern Compilers

Control flow in optimized code is dominated by repeated canonical patterns.

11.4.1 Zero and null checks via `test`

```
test    rdi, rdi
je      .Lnull
```

11.4.2 Bounds checks using unsigned jumps

```
cmp     edi, 15
ja      .Lout           # unsigned x > 15
```

11.4.3 Loop back-edges using `dec`/`jnz` or compare/jcc

```
dec     ecx
jnz     .Lloop

inc     eax
cmp     eax, esi
jb      .Lloop          # unsigned i < n
```

11.4.4 Short-circuit logic lowered to multiple conditional branches

```
# if (a && b) ...
test    edi, edi
je      .Lfalse
test    esi, esi
je      .Lfalse
# true path
```

11.4.5 Switch lowering: bounds check + indirect jump

```
sub     edi, BASE
cmp     edi, MAX
ja      .Ldefault
jmp     qword ptr [rip + .Ljt + rdi*8]
```

11.4.6 Tail-call style end-of-function transfer

```
# last action is call to another function
jmp     Target
```

11.4.7 Materialized boolean via `setcc`

```
cmp     eax, ebx
setl    al              # signed less
movzx   eax, al
```

11.5 Why Assembly Often Looks "Unnatural"

Assembly generated by modern compilers frequently looks unnatural to programmers because it is not produced to be readable; it is produced to satisfy correctness constraints while optimizing multiple cost models (code size, register pressure, control flow shape, and scheduling freedom).
Primary reasons it looks surprising:

11.5.1 Reason 1: high-level structure is flattened into basic blocks

Source constructs such as `if/else`, `for`, and `switch` become jumps between labeled blocks. The resulting layout reflects optimization choices more than source structure.

11.5.2 Reason 2: conditions are inverted and reordered

The compiler may invert predicates, reorder checks, or use early exits to keep hot paths as fall-through. This can reverse the apparent meaning if you assume the first branch corresponds to the source-level `if` directly.

11.5.3 Reason 3: booleans are not booleans

The compiler may avoid branches by materializing predicates into integers (`setcc`), or by keeping predicates in flags without ever forming a 0/1 variable.

11.5.4 Reason 4: common subexpressions and dead paths are eliminated

Conditions can disappear or merge. Branches may vanish entirely if code is proven unreachable or constant-folded, and loops may be transformed.

11.5.5 Reason 5: layout is chosen for execution flow, not for narrative

Blocks may be placed out of source order; cold blocks can be moved far away; join points can be merged; unconditional jumps can appear simply to reach an arranged layout.
Example that looks unnatural but is structurally optimal (guard then early return):

```
test    rdi, rdi
je      .Lret0
# main work
call    Work
mov     eax, 1
ret
.Lret0:
xor     eax, eax
ret
```

Reading discipline for compiler-generated control flow:

- treat each label as a basic block,
- translate each `jcc` into a flag predicate,
- follow edges: taken target vs fall-through,
- reconstruct the high-level meaning from the graph, not from linear order.

Chapter 12

Reading and Debugging Control Flow

12.1 Tracing Execution with the Instruction Pointer

To trace control flow, treat IP/RIP as the single source of truth for "where execution is". Every step of execution is:

- fetch instruction at IP/RIP,
- execute it,
- update IP/RIP to next sequential address or a branch/call/return target.

Practical tracing rule:

- for `jcc`: evaluate the flag predicate to decide taken vs fall-through,
- for `jmp`: always taken (IP/RIP becomes target),
- for `call`: IP/RIP becomes target and a return address is pushed,
- for `ret`: IP/RIP is popped from the stack.

Minimal trace example (two-path branch):

```
test    edi, edi
je      .Lzero          # if ZF==1 -> jump
mov     eax, 1          # fall-through path
ret
.Lzero:
xor     eax, eax        # taken path
ret
```

Trace procedure:

- compute ZF from `test edi,edi`,
- if ZF==1, next IP/RIP is `.Lzero`,
- else, next IP/RIP is the `mov` instruction.

Call/return trace (return chain):

```
call    A               # push RA0; RIP:=A
mov     eax, 7          # resumes here (RA0)
ret

A:
call    B               # push RA1; RIP:=B
ret                     # pops RA0

B:
ret                     # pops RA1
```

Key invariant: at any moment, `[rsp]` holds the next return target for `ret`.

12.2 Understanding Disassembly Output

Disassembly is a rendering of bytes into instructions plus labels and addresses. Correct reading requires separating:

- **instruction semantics** (architectural truth),
- **symbolization** (names added by tools),
- **layout** (block order chosen by compiler and linker).

Common disassembly elements:

- addresses and offsets: where each instruction resides,
- labels: basic block entry points (tool-generated or symbol-based),
- operand forms: registers, immediates, and memory addressing modes,
- control-flow annotations: conditional/unconditional targets.

Example of typical annotated output shape:

```
.L0:
cmp     edi, 7
ja      .Ldefault
jmp     qword ptr [rip + .Ljt + rdi*8]

.Ldefault:
xor     eax, eax
ret
```

Interpretation rule:

- ignore label naming style; treat labels as nodes,

- focus on `jcc/jmp/call/ret` edges,
- confirm operand width (8/16/32/64) for correct flag meaning.

12.3 Recognizing High-Level Constructs in Assembly

High-level constructs map to repeatable control-flow graphs.

12.3.1 If statement

```
test    edi, edi
je      .Lskip
call    A
.Lskip:
ret
```

12.3.2 If/else diamond

```
cmp     eax, ebx
jne     .Lelse
call    ThenPath
jmp     .Ljoin
.Lelse:
call    ElsePath
.Ljoin:
ret
```

12.3.3 While loop (top-tested)

```
.Lhead:
cmp     eax, ebx
jge     .Lexit
```

```
# body
inc     eax
jmp     .Lhead
.Lexit:
ret
```

12.3.4 Do-while loop (bottom-tested)

```
.Lbody:
# body
dec     ecx
jnz     .Lbody
ret
```

12.3.5 Switch with jump table

```
cmp     edi, 5
ja      .Ldefault
jmp     qword ptr [rip + .Ljt + rdi*8]
```

12.3.6 Short-circuit AND

```
test    edi, edi
je      .Lfalse
test    esi, esi
je      .Lfalse
# true path
call    T
ret
.Lfalse:
call    F
ret
```

Recognition method:

- identify the **edges** (jumps) first,
- classify the structure by its shape (diamond, back-edge, multi-way),
- then attach meaning by mapping each `jcc` to its flag predicate.

12.4 Debugging Wrong Jumps

Wrong jumps are usually one of five root causes:

- wrong signed/unsigned `jcc` selection,
- wrong boundary condition (`jg` vs `jge`, `ja` vs `jae`),
- flags clobbered between producer and consumer,
- wrong operand width (partial register),
- wrong condition source (branch tests flags from the wrong instruction).

12.4.1 Signed vs unsigned bug

```
cmp     eax, ebx
jl      .Lless          # bug if values are unsigned (should be jb)
```

Fix:

```
cmp     eax, ebx
jb      .Lbelow_u
```

12.4.2 Boundary bug (strict vs inclusive)

```
cmp     edi, 10
ja      .Lgt_u              # strictly >
# if intended: >= then should be jae
```

12.4.3 Flags clobbered bug

```
cmp     eax, ebx
add     ecx, 1              # overwrites flags
je      .Lequal             # now tests ZF from add (bug)
```

Fix approach: keep compare adjacent to branch or use a non-flag-setting instruction where needed (e.g., `lea` for arithmetic updates).

12.4.4 Wrong width bug

```
test    al, al
je      .Lzero              # tests only low 8 bits
```

Fix: test the intended width:

```
test    eax, eax            # 32-bit
je      .Lzero
```

Debugging procedure:

- write down the `jcc` predicate in flags,
- locate the last flag-producing instruction,
- ensure nothing between them changes the required flags,
- confirm the correct signed/unsigned family was used,
- confirm operand width matches the source-level intent.

12.5 Control Flow Bugs That Look Like Data Bugs

Many failures appear as corrupted values but are actually wrong-path execution. A wrong branch causes the program to:

- skip initialization,
- call the wrong helper,
- miss bounds checks,
- execute a cleanup path twice,
- fall through into code that assumes a different state.

Example: missing initialization due to wrong condition:

```
test    edi, edi
jne     .Lskip_init       # bug: should be je to skip init only when x==0
mov     dword ptr [rbx], 0 # init (skipped unexpectedly)
.Lskip_init:
# later code reads [rbx] and seems "corrupted"
```

Example: wrong signedness causes out-of-range access that looks like data corruption:

```
# index in edi should be unsigned
cmp     edi, esi
jl      .Lin_range        # bug: signed compare
# out-of-range path not taken for large unsigned values with high bit set
.Lin_range:
mov     eax, dword ptr [rdi*4 + rbx]   # reads unintended memory
```

Example: wrong early-exit leads to double-use of resources:

```
test    rdi, rdi
je      .Lcleanup
call    UseResource
.Lcleanup:
call    ReleaseResource    # runs even when resource was never acquired
ret
```

Practical debugging mindset:

- when data looks impossible, first verify the control path that produced it,
- prove which branch was taken by reconstructing flags and predicates,
- confirm that required setup code executed before dependent code.

Control flow is the gatekeeper of state. If execution enters the wrong block, the resulting state can mimic random data corruption even when memory operations are technically correct.

Appendices

Appendix A — Control Flow Instruction Summary

This appendix provides a compact, architecture-accurate summary of the x86 control-flow instructions used throughout this booklet. Each group is organized by *what architectural state it consumes or produces*, not by source-language meaning.

`cmp` and `test`

`cmp a, b` performs an internal subtraction (`a - b`) and updates flags without storing the result. It is the primary instruction for equality and ordering decisions.
`test a, b` performs an internal bitwise AND (`a & b`) and updates flags without storing the result. It is used for zero checks, bit tests, and sign checks.
Flag effects summary:

- `cmp`: sets ZF, SF, CF, OF according to subtraction rules
- `test`: sets ZF, SF; clears OF; CF is set to 0

Canonical forms:

```
cmp     eax, ebx        # ordering and equality (signed/unsigned via jcc)
test    rax, rax        # zero / non-zero check
test    eax, 0x20       # bit test (mask)
```

Conditional Jumps (Grouped by Logic)

Conditional jumps consume flags produced by prior instructions. They do not compare operands themselves.

Equality (ZF-based):

```
je      label           # ZF==1  (equal / zero)
jne     label           # ZF==0  (not equal / non-zero)
```

Unsigned ordering (CF and ZF):

```
jb      label           # CF==1              below
jae     label           # CF==0              above or equal
ja      label           # CF==0 and ZF==0    above
jbe     label           # CF==1 or  ZF==1    below or equal
```

Signed ordering (SF and OF, plus ZF):

```
jl      label           # SF!=OF             less
jge     label           # SF==OF             greater or equal
jg      label           # ZF==0 and SF==OF   greater
jle     label           # ZF==1 or  SF!=OF   less or equal
```

Sign-based (from test/logic):

```
js      label           # SF==1  (negative)
jns     label           # SF==0  (non-negative)
```

Reading rule: signedness is selected by the jump mnemonic, not by `cmp`.

`jmp` (Direct and Indirect)

`jmp` unconditionally updates IP/RIP. It does not push a return address.

Direct jump (target encoded as relative displacement):

```
jmp     label
```

Indirect jump (target from register or memory):

```
jmp     rax
jmp     qword ptr [rbx]
jmp     qword ptr [rip + table + rdi*8]
```

Indirect jumps are used for:

- jump tables (`switch` lowering),
- state machines,
- computed control transfer.

Correctness requires bounds checking before indirect targets are computed.

`call` and `ret`

`call` performs a control transfer while saving the return address on the stack.

```
call    target          # push return address; RIP := target
```

Direct call (relative target):

```
call    func
```

Indirect call (target from register or memory):

```
call    rax
call    qword ptr [rip + fp]
```

`ret` restores control by loading IP/RIP from the stack.

```
ret                     # RIP := pop()
```

`ret imm16` additionally adjusts the stack pointer after popping the return address.

```
ret     16              # pop return address; add extra stack adjustment
```

Key architectural facts:

- `call` creates a return chain by pushing addresses
- `ret` is an indirect control transfer driven by memory
- stack integrity is essential for correct control flow

This summary table is intended as a quick architectural reference when reading or debugging control flow in x86 disassembly.

Appendix B — Common Errors and Dangerous Assumptions

This appendix catalogs high-frequency control-flow errors observed in real x86 code. Each item explains the faulty assumption, why it fails architecturally, and how to recognize or avoid it when reading or writing assembly.

Mixing Signed and Unsigned Jumps

Faulty assumption: signedness is carried by registers or by `cmp`. **Architectural reality**: signedness is selected solely by the `jcc` mnemonic.
Using a signed jump for an unsigned quantity (sizes, indices, lengths) inverts logic when the high bit is set.
Bug pattern:

```
cmp     edi, esi
jl      .Lin_range      # bug if edi/esi are unsigned sizes
```

Correct unsigned form:

```
cmp     edi, esi
jb      .Lin_range      # unsigned: edi < esi
```

Inverse bug (using unsigned jumps for signed values):

```
cmp     eax, 0
ja      .Lpositive      # bug for signed test (negative values look large)
```

Correct signed forms:

```
cmp     eax, 0
jg      .Lpositive      # signed: eax > 0

test    eax, eax
js      .Lnegative
```

Reading rule: decide signed vs unsigned from the `jcc` used, never from variable names or source-level intent.

Assuming Flags Persist

Faulty assumption: flags produced by a compare remain valid until the branch.
Architectural reality: any flag-writing instruction overwrites them.
Bug pattern (flag clobber):

```
cmp     eax, ebx
add     ecx, 1          # overwrites ZF/SF/CF/OF
je      .Lequal         # now tests flags from add (bug)
```

Safe patterns:

```
cmp     eax, ebx
je      .Lequal         # adjacent use

cmp     eax, ebx
lea     ecx, [ecx + 1]  # arithmetic without flags
je      .Lequal
```

Recognition checklist:

- identify the `jcc`,
- locate the immediately preceding flag producer,
- verify no intervening instruction sets the needed flags.

Trusting Stack State Blindly

Faulty assumption: the stack is correct by construction. **Architectural reality**: `ret` is an indirect jump whose target is read from memory.
Classic mismatches:

```
func:
push    rbx
# ...
ret                     # bug: return address is not on top
```

Correct discipline:

```
func:
push    rbx
# ...
pop     rbx
ret
```

Wrong stack adjustment:

```
sub     rsp, 32
# ...
add     rsp, 24         # bug: should restore 32
ret
```

Key rule: before `ret`, ensure the stack pointer matches the value expected by the caller (modulo the return address). Treat every `push/pop` and `sub/add rsp` as control-flow critical.

Misreading Compiler Intent

Faulty assumption: assembly should mirror source structure linearly. **Architectural reality**: compilers reshape control flow for layout, fall-through, and canonical patterns.

Common misreads:

Inverted conditions (hot path as fall-through):

```
test    rdi, rdi
je      .Lerror          # uncommon path
# hot path continues
call    Work
ret
.Lerror:
call    HandleError
ret
```

Early exits instead of nesting:

```
test    rdi, rdi
je      .Lret
test    esi, esi
je      .Lret
call    Work
.Lret:
ret
```

Boolean materialization instead of branching:

```
cmp     eax, ebx
setl    al               # signed less
movzx   eax, al
```

Jump-table dispatch instead of chains:

```
cmp     edi, 7
ja      .Ldefault
jmp     qword ptr [rip + .Ljt + rdi*8]
```

Reading discipline:

- reconstruct the control-flow graph (blocks and edges),
- translate each `jcc` into a flag predicate,
- ignore source order expectations; follow taken vs fall-through paths,
- verify signedness, width, and flag provenance for each decision.

These assumptions recur because they feel intuitive at the source level but are false at the ISA level. Correct control-flow reasoning requires strict adherence to architectural semantics, not narrative reading of the instruction stream.

Appendix C — Preparation for Next Booklets

This appendix bridges pure control-flow mechanics to the structured, rule-driven world of stacks, calling conventions, and ABIs. It defines what the reader must already understand—and be able to recognize in disassembly—before moving forward.

Readiness for Stack and Calling Conventions

Before studying any calling convention, the reader must be fluent in the architectural stack model and its interaction with control flow.
You should be able to:

- trace how `call` pushes a return address and how `ret` consumes it,
- verify stack-pointer correctness across `push/pop` and `sub/add rsp`,

- identify stack-resident data vs control data (return addresses),
- reason about nested calls and return chains without relying on source code.

Minimal stack discipline example (correct):

```
func:
sub     rsp, 32          # reserve local space
# use [rsp]..[rsp+31]
add     rsp, 32
ret
```

Classic stack-corruption bug to recognize:

```
func:
push    rbx
sub     rsp, 16
# ...
add     rsp, 16
ret                      # bug: rbx still on stack above return address
```

Required reasoning skill:

- at the point of `ret`, you must be able to state exactly what value is at `[rsp]`,
- you must be able to reconstruct how that value got there.

This readiness ensures that when calling conventions define which registers are preserved or how arguments are passed, those rules can be mapped onto a correct underlying stack model.

Readiness for ABI-Level Control Flow

ABIs formalize control flow across compilation units, libraries, and language boundaries. Prior to studying ABI rules, the reader must already understand the architectural primitives ABIs are built upon.
You should be able to:

- distinguish direct vs indirect calls in disassembly,
- recognize when a call target comes from a register or memory,
- understand that `ret` is an indirect jump whose target is memory-driven,
- identify tail-call shapes where `jmp` replaces `call+ret`.

Indirect call recognition:

```
mov     rax, qword ptr [rdi]
call    rax             # ABI-visible indirect call
```

Tail-call transfer shape:

```
# last action of function
jmp     target_func     # no new return address pushed
```

Return-chain integrity:

```
call    A
# resumes here
ret

A:
call    B
ret

B:
ret
```

ABI-level reasoning builds on these facts:

- which registers must survive calls,
- which registers are free to clobber,

- how arguments and return values flow across calls,
- how stack alignment and layout are enforced.

Without a precise understanding of architectural control transfers, ABI rules appear arbitrary; with it, they become systematic constraints.

Transition from Flow Mechanics to Function Semantics

This booklet intentionally treated functions as *control-flow nodes*, not semantic units. The next stage transitions from mechanics to meaning.
What changes:

- `call/ret` stop being viewed as isolated instructions and become part of a contract,
- registers stop being anonymous and gain ABI-defined roles,
- stack space stops being ad-hoc and becomes a structured frame,
- control flow becomes constrained by inter-procedural rules.

Control-flow-only view (this booklet):

```
call    func
# ...
ret
```

ABI-aware view (next booklets):

```
# arguments prepared per ABI
call    func
# return value observed in defined register
```

Required mental shift:

- from “where does execution go” to “who owns which state at each boundary”,
- from local correctness to cross-function correctness,
- from single-function reasoning to system-wide call discipline.

If you can:

- trace execution using RIP alone,
- validate every jump, call, and return path,
- prove stack correctness at each return,

then you are prepared to move from control-flow mechanics to full calling conventions and ABI-level execution models.
This appendix marks the boundary between *how control flows* and *what a function means*.

References

R.1 Official x86 and x86-64 Architecture Manuals

The authoritative foundation for all control-flow behavior in this booklet is the official x86 and x86-64 architecture specifications. These manuals precisely define:

- architectural state (RIP/EIP, RFLAGS, general-purpose registers),
- exact semantics of `cmp`, `test`, `jcc`, `jmp`, `call`, and `ret`,
- flag-setting rules and their consumption by conditional branches,
- near vs far control transfers at the ISA level.

Every control-flow rule presented in this booklet is derived from these architectural definitions, not from compiler behavior or operating-system conventions.
Required reader capability after this booklet:

- read an instruction description and extract its precise control-flow effect,
- distinguish architectural guarantees from implementation details,
- reason about control flow solely from the ISA contract.

R.2 Compiler Documentation and Generated Code Behavior

Modern compilers implement control flow by mapping high-level constructs onto canonical x86 instruction patterns. Compiler documentation explains:

- why certain branch forms (`test/jcc`, `cmp/jcc`) dominate output,
- why indirect jumps and jump tables are preferred for dense multi-way dispatch,
- how branch inversion and fall-through shaping influence layout,
- why certain instructions (`loop`) are avoided in favor of simpler patterns.

In this booklet, compiler behavior is treated as *observable output*, not as authority. The correct interpretation strategy is:

- start from ISA semantics,
- observe how compilers exploit those semantics,
- never infer architecture rules from compiler habits.

Representative compiler-emitted patterns used throughout this booklet:

```
test    rdi, rdi
je      .Lexit

cmp     eax, esi
jb      .Lloop

jmp     qword ptr [rip + table + rdi*8]
```

R.3 Instruction Set Reference Sources

Instruction-set reference sources provide instruction-by-instruction details, including:

- exact flag effects,
- operand-size behavior,
- valid addressing modes,
- control-transfer classification (direct vs indirect).

These references are essential when:

- validating whether an instruction modifies flags,
- confirming whether a form is allowed in 64-bit mode,
- checking corner cases such as partial-register writes or implicit operands.

This booklet consistently relies on instruction-reference semantics for:

- distinguishing `cmp` vs `test`,
- separating signed vs unsigned branch logic,
- understanding `ret imm16` stack effects,
- identifying which instructions are safe between compare and branch.

R.4 Academic and Professional CPU Architecture Materials

Academic and professional architecture materials provide the conceptual framing used throughout this series:

- control-flow graphs (CFGs),
- basic blocks and edges,
- dominance, back-edges, and loop structure,
- separation of architectural state from microarchitectural execution.

These materials explain:

- why control flow must be reasoned about structurally, not linearly,
- why mispredicted or misdirected control flow dominates performance and correctness issues,
- how call/return form a structured control discipline.

The methodology used in this booklet—reconstructing execution by following IP/RIP transitions—comes directly from this body of work.

R.5 Cross-References to Other Booklets in This Series

This booklet is intentionally positioned between architectural fundamentals and ABI-level execution.
Upstream dependencies:

- instruction execution model,
- registers and flags semantics,
- basic data representation rules.

Downstream continuations:

- stack layout and frame structure,
- calling conventions and register preservation rules,
- ABI-defined argument passing and return values,
- inter-procedural control flow across modules.

Conceptual handoff:

- this booklet answers *where execution goes*,
- subsequent booklets answer *what state is owned and preserved*.

If the reader can:

- trace execution by following RIP through `jcc`, `jmp`, `call`, and `ret`,
- validate every branch predicate from flags,
- prove stack correctness at each return,

then this booklet has fulfilled its role as the control-flow foundation for the rest of the CPU Programming Series.