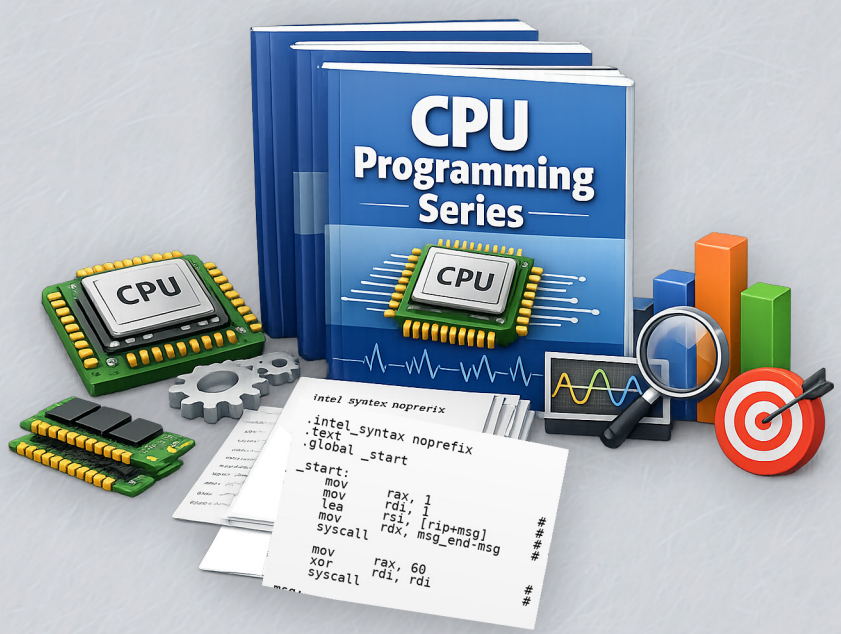
https://simplifycpp.org

CPU Programming Series

ARM 32-bit Assembly Fundamentals

Registers, CPSR, and Load/Store Discipline



11

Prepared by Ayman Alheraki

CPU Programming Series

ARM 32-bit Assembly Fundamentals

Registers, CPSR, and Load/Store Discipline

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Purpose of This Booklet

This booklet establishes a **precise and architecture-faithful foundation** for programming in **ARM 32-bit assembly**, focusing on the elements that determine correctness before performance: registers, CPSR, and the load/store execution discipline.
The goal is not to teach “how to write instructions quickly,” but to ensure the reader understands **what the processor is allowed to do**, **what it will never do**, and **why certain patterns are correct or incorrect by design**. This discipline is essential for writing reliable low-level code, understanding compiler output, and preparing for ABI-level interoperability in later booklets.
All explanations are derived from **official ARM architectural behavior**, not from compiler tricks or OS-specific conventions.

Why ARM Is Different from x86

ARM and x86 differ fundamentally in execution philosophy:

- ARM is a **strict load/store architecture**: memory is accessed only through explicit load and store instructions.

- x86 allows **memory operands** in many arithmetic and logical instructions.
- ARM uses a **uniform register file** with architecturally visible roles.
- x86 relies heavily on **implicit behavior** and historical instruction semantics.

ARM's design enforces clarity:

```
    LDR     R0, [R1]      @ load from memory
    ADD     R2, R0, R3    @ operate on registers only
    STR     R2, [R1]      @ store back to memory
```

In contrast, x86 may combine these steps implicitly. ARM forbids such patterns intentionally to ensure **predictable execution**, simpler pipelines, and consistent compiler behavior. Additionally, ARM exposes **condition flags and conditional execution** as first-class architectural features, whereas x86 relies primarily on branching.

How to Read This Booklet

This booklet must be read **sequentially**.
Each chapter builds on architectural rules introduced earlier. Skipping sections often leads to misunderstanding later material, especially CPSR behavior and control flow rules.
Guidelines for effective reading:

- Focus on **architectural intent**, not instruction memorization.
- Treat examples as **behavioral demonstrations**, not templates.
- Assume no operating system, no ABI, and no compiler assistance.
- Verify mental models against instruction behavior, not assumptions.

All examples are intentionally minimal to expose **cause-and-effect relationships** at the instruction level.

Scope, Limits, and Design Discipline

This booklet deliberately limits its scope.
It covers:

- ARM 32-bit register model
- CPSR structure and flag semantics
- Conditional execution rules
- Load/store memory discipline

It does **not** cover:

- Calling conventions or ABI rules
- Stack frame layouts
- Exception handling or OS interaction
- Performance tuning or pipeline optimization

This separation is intentional. Mixing ABI or OS concepts with architectural fundamentals leads to fragile understanding and incorrect mental models.
By the end of this booklet, the reader should be able to:

- Reason about ARM instructions without relying on a compiler
- Predict CPSR effects accurately
- Write load/store sequences that are architecturally correct
- Prepare confidently for ARM ABI and stack discipline in subsequent booklets

Chapter 1

ARM Architecture Overview

1.1 RISC Philosophy and Design Goals

ARM is a classic example of a **RISC (Reduced Instruction Set Computer)** design approach: a small set of regular instructions, a large register file, and a machine model that encourages predictable execution and efficient implementation.
Key design goals relevant to assembly programmers:

- **Load/Store discipline**: arithmetic and logical operations target registers; memory is accessed explicitly via load/store.
- **Regular encodings and operands**: many instructions share a consistent operand structure (destination + sources).
- **Efficient decoding and pipeline-friendly behavior**: simpler instruction patterns reduce decode complexity and ease scheduling.
- **Architectural transparency**: visible registers, explicit flag-setting, and explicit addressing modes.

A practical consequence is that you write code in **three phases**:

```
@ Phase 1: bring data into registers
LDR     R0, [R1]          @ R0 = *R1

@ Phase 2: compute using registers
ADD     R0, R0, R2        @ R0 = R0 + R2

@ Phase 3: write results back to memory (if needed)
STR     R0, [R1]          @ *R1 = R0
```

On a machine that allows arithmetic directly on memory operands, phase separation can be blurred. On ARM, this separation is **the model**.

1.2 ARM Execution Model (Conceptual)

At a conceptual level, ARM 32-bit execution is best understood through a strict sequence:

- **Fetch**: the instruction at the current PC is fetched.
- **Decode**: fields are decoded (opcode, registers, immediates, condition).
- **Condition check**: many ARM instructions are conditionally executed based on CPSR flags.
- **Execute**: the operation runs (ALU, load/store address generation, branch).
- **Write-back**: results update registers and possibly CPSR (depending on the instruction form).

The architectural state that matters most to you early on:

- Registers **R0–R15** (including SP/LR/PC roles)

- **CPSR** for flags and mode/state bits
- Memory as an external state reached only via **LDR/STR**

Example: conditional execution is a defining architectural feature. The instruction `ADDEQ` executes only if `Z=1` (equal/zero).

```
CMP     R0, R1              @ sets flags based on (R0 - R1)
ADDEQ   R2, R2, #1          @ if Z==1 then R2++
ADDNE   R2, R2, #2          @ if Z==0 then R2+=2
```

This is not a compiler trick; it is a core architectural rule. Whether performance benefits exist depends on microarchitecture, but the **behavior** is architectural.

1.3 Instruction Set vs Microarchitecture

It is critical to separate:

- **ISA (Instruction Set Architecture)**: the contract defined by the architecture. It specifies instruction semantics, registers, flags, memory model constraints, and visible behavior.
- **Microarchitecture**: the internal implementation of a particular CPU (pipeline depth, branch prediction, caches, reorder mechanisms, issue width).

As an assembly programmer building correct foundations, you write to the ISA.
Microarchitecture affects **performance**, but must not be required for **correctness**.
Example: both of the following are ISA-correct, but can differ in performance depending on CPU implementation:

```
    @ Version A: branch-based
    CMP     R0, #0
    BEQ     is_zero
    ADD     R1, R1, #5
is_zero:
```

```
    @ Version B: conditional execution (no branch)
    CMP     R0, #0
    ADDNE   R1, R1, #5
```

Architecturally, both are correct. Microarchitecturally, one might be faster or slower depending on branch prediction, instruction fusion rules, pipeline front-end, etc. This booklet uses examples to teach architectural behavior first; later booklets can address performance.

1.4 ARM 32-bit Operating Modes (Overview Only)

ARM 32-bit defines several **operating modes** that affect the processor's privileged state, exception handling, and (in many designs) which banked registers are active.
This booklet includes only an overview because mode switching is primarily relevant when discussing:

- exceptions and interrupts
- OS/RTOS kernels
- privileged instructions and system control

The essential conceptual points for this booklet:

- The CPU always runs in **one mode at a time**.
- Some modes have **banked registers** (alternate physical registers for some architectural names) to speed exception handling and preserve context.

- Mode and state bits live in **CPSR** (and related saved status registers in exception modes).

A minimal mental model:

- **User mode**: unprivileged application execution.
- **Privileged/exception modes**: entered by exceptions/interrupts; may have banked SP/LR and possibly other registers depending on mode.

Why it matters even in a fundamentals booklet:

- It explains why documentation may show multiple SP/LR instances.
- It prevents incorrect assumptions about register preservation across exceptions.
- It prepares you to interpret CPSR bits and saved status behavior later.

This booklet will not require you to write mode-switching code, but it will ensure you understand the architectural reason that ARM documentation distinguishes between user state and privileged/exception state.

Chapter 2

ARM 32-bit Register Model

2.1 General-Purpose Registers (R0–R12)

ARM 32-bit exposes sixteen architectural registers `R0--R15`. In most application-level code, `R0--R12` are treated as general-purpose registers (GPRs). The architecture does not force high-level roles for `R0--R12`; their role is established by discipline, ABI rules (later booklet), and local conventions.
In disciplined assembly, classify usage by intent:

- **Inputs/temporaries**: short-lived values used inside a basic block.
- **Live values**: values that must survive across branches/calls (requires explicit save/restore discipline).
- **Address registers**: base pointers and iterators used by load/store addressing modes.

Example: register-only computation with explicit dataflow

```
    @ R0 = a, R1 = b, R2 = c
```

```
    @ Compute: r = (a + b) ^ c
    ADD     R3, R0, R1          @ R3 = a + b
    EOR     R3, R3, R2          @ R3 = (a + b) XOR c
    @ Result in R3
```

Example: register roles for load/store traversal

```
    @ R0 = base pointer, R1 = count, R2 = running sum
    MOV     R2, #0

loop_sum:
    LDR     R3, [R0], #4        @ R3 = *R0; R0 += 4
    ADD     R2, R2, R3          @ sum += value
    SUBS    R1, R1, #1          @ count--; update flags
    BNE     loop_sum
    @ sum in R2
```

Key lesson: in ARM, arithmetic uses registers; memory is reached via explicit load/store. Keeping addresses, counts, and accumulators in distinct registers reduces mistakes and clarifies correctness.

2.2 Stack Pointer (SP / R13)

`R13` is commonly used as the stack pointer and is referred to as **SP**. Although it is architecturally a register, correct code treats it as a **special register with invariants**:

- SP points to the current stack top for the active mode/state.
- Stack growth direction and alignment requirements are convention/ABI-defined, but alignment discipline is mandatory for correctness in real systems.
- SP should not be used as a general-purpose register in disciplined assembly.

Example: simple stack allocation and deallocation

```
@ Allocate 16 bytes of local stack space
SUB     SP, SP, #16

@ Use stack slots (example)
STR     R0, [SP, #0]      @ store at local slot 0
STR     R1, [SP, #4]      @ store at local slot 1

@ Deallocate
ADD     SP, SP, #16
```

Example: pushing/popping callee context (conceptual)

```
@ Save registers to stack (conceptual; exact set is ABI-dependent)
PUSH    {R4, R5, R6, LR}

@ ... function body ...

POP     {R4, R5, R6, PC}  @ restore and return (PC loaded)
```

Even before ABI details, the conceptual discipline is:

- If you modify SP, you must restore it.
- If you store something at `[SP, #offset]`, that offset must remain valid until you are done.

2.3 Link Register (LR / R14)

`R14` is commonly used as the link register, **LR**. Branch-with-link instructions place a return address into LR. This means:

- LR is a **return address register**.
- Any nested call overwrites LR unless you save it.
- Treat LR as **live** whenever you intend to return.

Example: call and return using LR

```
    BL      callee            @ LR = return address; branch to callee
    @ execution continues here after callee returns

callee:
    @ ... do work ...
    BX      LR                @ return to caller
```

Example: nested call requires saving LR

```
caller:
    BL      f
    B       done

f:
    PUSH    {LR}              @ save return address
    BL      g                 @ overwrites LR
    POP     {LR}              @ restore
    BX      LR

g:
    @ ... do something ...
    BX      LR

done:
```

If you forget to preserve LR across a nested call, the function may return to an unintended address. This is one of the most common early ARM assembly failures.

2.4 Program Counter (PC / R15)

`R15` is the program counter, **PC**. It identifies the next instruction stream location. In ARM 32-bit, PC can appear as an operand in some instructions, but you must treat PC usage as **architecturally special**:

- Writing to PC changes control flow.
- Reading PC yields a value related to the current instruction address, but the exact observed value is affected by the architecture's instruction fetch behavior and state; do not treat it like an ordinary register value.
- Correctness requires explicit branch/return intent, not "PC arithmetic" hacks.

Example: return by loading PC

```
@ Typical return forms
BX      LR              @ return via LR
@ or (conceptual pattern)
MOV     PC, LR          @ control transfer via PC write
```

Example: indirect branch via register

```
@ R0 holds a target address
BX      R0              @ branch to address in R0
```

The discipline in this booklet: use PC writes only to express intentional control flow (branch/return), not as a general computation target.

2.5 Banked Registers (Conceptual View)

ARM 32-bit defines multiple operating modes for exceptions and privileged execution. To support fast exception entry, some registers are **banked** in certain modes, meaning:

- The architectural name (e.g., SP, LR) may refer to a different physical register depending on the current mode.
- Banked registers allow an exception handler to have its own SP/LR without immediately saving the interrupted context.

Conceptually, think of this as:

```
@ Conceptual idea (not actual code):
@ In user mode:   SP_user, LR_user
@ In handler:     SP_exc,  LR_exc
```

Why this matters here:

- It prevents false assumptions about “one global SP” for the entire system.
- It clarifies why architectural documents describe SP/LR variants across modes.
- It prepares you to interpret CPSR mode bits later without requiring OS/kernel material.

2.6 Register Usage Discipline

This booklet enforces a disciplined model that remains correct across toolchains and prepares you for ABI-level rules later.

Rule 1: Treat SP, LR, PC as special

- SP is never a scratch register.
- LR is preserved if you call another function before returning.
- PC is written only for explicit control flow.

Rule 2: Make liveness explicit

If a value must survive a branch or a call, you must explicitly preserve it (in registers that remain live, or by saving it to the stack).

```
    @ R4 holds a long-lived value; preserve across call
    PUSH    {R4, LR}
    @ ... R4 is live ...
    BL      helper
    @ ... still need R4 ...
    POP     {R4, LR}
    BX      LR
```

Rule 3: Separate roles for clarity

Use distinct registers for:

- base addresses
- indices/counters
- temporaries
- accumulated results

```
    @ R0 base, R1 index, R2 temp, R3 result
    LDR     R2, [R0, R1, LSL #2]   @ temp = base[index]
    ADD     R3, R3, R2             @ result += temp
```

Rule 4: Prefer predictable patterns over cleverness

Write code that is obviously correct under the ISA rules:

- explicit loads/stores
- explicit flag-setting points
- explicit save/restore boundaries

This discipline is the foundation for later chapters on CPSR behavior and conditional execution, and for later booklets on ABI, stack frames, and interoperability.

Chapter 3

Program Counter, Link Register, and Control Flow

3.1 PC Semantics and Instruction Fetch

In ARM 32-bit, `R15` is the **Program Counter (PC)**. Architecturally, PC identifies the current instruction stream position, but it must be treated as **special**:

- Writing PC causes an immediate **control-flow change** (branch/return).
- Reading PC can yield a value related to the **current execution address**, but it is not a general-purpose value. The observed PC value depends on the instruction set state and the architectural rules for PC as an operand.
- Correct programs must not rely on "PC behaves like a normal register" assumptions.

Example: PC write expresses explicit control transfer

```
    @ R0 holds target address
```

```
    MOV     PC, R0              @ branch to target by writing PC
```

Example: computed branch via BX (preferred for state-aware control flow)

```
    @ R0 holds target address
    BX      R0                  @ branch to address in R0
```

Discipline: use `B/BL/BX/BLX` (and well-defined return sequences) rather than "PC arithmetic" tricks. This keeps code correct across toolchains and across instruction-set states.

3.2 LR Behavior in Branch-with-Link

`R14` is the **Link Register (LR)**. The **branch-with-link** instruction stores a return address in LR and transfers control to the target.

Example: BL stores return address in LR

```
    BL      callee              @ LR = return address; branch to callee
    @ execution resumes here after callee returns

callee:
    @ ... work ...
    BX      LR                  @ return
```

Key properties:

- LR is overwritten by every `BL` / `BLX`.
- If a function makes a nested call, it must **preserve LR** first.
- LR is not "magic stack return state"; it is a register you must manage.

Example: nested call requires saving LR

```
f:
    PUSH    {LR}              @ preserve return address
    BL      g                 @ overwrites LR
    POP     {LR}              @ restore return address
    BX      LR

g:
    @ ... work ...
    BX      LR
```

3.3 Return Sequences and Pitfalls

A return is a control transfer back to the caller. In disciplined ARM 32-bit assembly, returns are expressed using well-defined patterns:

- `BX LR` is the canonical explicit return.
- Restoring PC from the stack can be used when a function saved LR on entry and wants to combine restore+return.

Example: minimal leaf function return

```
leaf:
    @ no nested calls, LR not clobbered
    ADD     R0, R0, #1
    BX      LR
```

Example: return by popping PC

```
nonleaf:
```

```
    PUSH    {R4, LR}
    @ ... body may call other functions ...
    POP     {R4, PC}          @ restore R4 and return (PC loaded)
```

Common pitfalls:

- **Forgetting to save LR** in non-leaf functions: the function returns to the wrong address.
- **Clobbering LR as a scratch register**: later `BX LR` becomes invalid.
- **Inconsistent save/restore sets**: stack imbalance or corrupted return state.
- **Mixing return forms without discipline**: e.g., saving LR but returning via stale LR.

Pitfall demo: LR overwritten by nested call

```
bad_f:
    @ WRONG: does not preserve LR
    BL      g                 @ overwrites LR
    BX      LR                @ returns to somewhere inside g's caller path
```

Corrected:

```
good_f:
    PUSH    {LR}
    BL      g
    POP     {LR}
    BX      LR
```

3.4 Control Flow without a Stack

Not all control flow requires stack usage. If a routine:

- does not call other routines (leaf),
- does not need to preserve live values beyond available registers,

then it can be written without any stack operations.

Example: branch-based loop (no stack)

```
    @ R0 = pointer, R1 = count, R2 = sum
    MOV     R2, #0

loop:
    LDR     R3, [R0], #4      @ load and advance pointer
    ADD     R2, R2, R3        @ sum += value
    SUBS    R1, R1, #1        @ count-- and set flags
    BNE     loop              @ if not zero, continue
    @ sum in R2
```

Example: multi-way control flow using compares and conditional branches

```
    @ R0 holds x
    CMP     R0, #0
    BEQ     case_zero
    BLT     case_neg
    B       case_pos

case_zero:
    MOV     R1, #0
    B       done

case_neg:
    MOV     R1, #-1
```

```
    B       done

case_pos:
    MOV     R1, #1

done:
```

The key idea: structured control flow (loops, if/else, switch-like dispatch) is expressible using branches and flags without stack usage. The stack becomes necessary when you must preserve state across calls or exceed available registers.

3.5 ARM Branch Instructions Overview

ARM 32-bit provides a small set of branch/control-transfer instructions. The architectural concepts you must master:

- `B` : unconditional branch (PC-relative target).
- `BL`: branch with link (stores return address in LR).
- `BX`: branch to address in register (used for returns and indirect branches).
- `BLX`: branch with link and optional instruction-set state exchange (register target form commonly used for indirect calls).
- `CBZ/CBNZ` (where available): compare-and-branch on zero/nonzero for a register (reduces explicit `CMP` in common patterns).

Example: direct call vs indirect call

```
    BL      direct_target     @ direct call (symbol)
```

```
    @ R4 holds function pointer
    BLX     R4                @ indirect call; LR updated
```

Example: canonical return

```
    BX      LR                  @ return to caller
```

Example: compact loop using CBZ (if supported)

```
    @ R1 = count
loop2:
    CBZ     R1, done2
    SUB     R1, R1, #1
    B       loop2
done2:
```

Discipline summary

- Use `BL/BLX` for calls, and assume LR is overwritten.
- Use `BX LR` or stack-restore-to-PC patterns for returns.
- Use conditional branches for structured flow; avoid inventing ad-hoc PC tricks.
- Keep control flow readable: label blocks, use consistent patterns, and preserve return state explicitly.

Chapter 4

CPSR: Current Program Status Register

4.1 CPSR Layout and Bit Fields

The **Current Program Status Register (CPSR)** holds the architectural status of the processor: condition flags, control bits, and the current mode/state. In ARM 32-bit, CPSR is not a general-purpose register; it is a **system status register** that affects instruction execution (especially conditional execution) and privileged behavior.
For disciplined assembly at the architectural level, you must understand CPSR as three conceptual regions:

- **Condition flags (top bits)**: reflect arithmetic/logic results and drive conditional execution.
- **Execution control bits**: control instruction-set state and interrupt masking (system-level concept).
- **Mode field**: identifies the current operating mode (user/privileged/exception modes).

Conceptual view (do not treat as a full bit-accurate diagram)

```
@ CPSR conceptual grouping:
@ [flags: N Z C V] [control/state bits] [mode bits]
```

This booklet focuses on the **flag semantics** and their correctness impact. Control/state bits and mode bits are introduced only to avoid incorrect assumptions.

4.2 Condition Flags (N, Z, C, V)

ARM condition flags are updated by many operations and are stored in CPSR:

- **N (Negative)**: reflects the sign bit of the result (bit 31 of a 32-bit result).
- **Z (Zero)**: set if the result is zero.
- **C (Carry)**: indicates unsigned carry out in addition; in subtraction it indicates *no borrow* (unsigned compare rule).
- **V (Overflow)**: indicates signed overflow (two's complement overflow).

Example: basic flag production via SUBS/CMP

```
@ CMP is architecturally a subtraction that updates flags and discards
↪  the result
CMP     R0, R1            @ flags reflect (R0 - R1)

@ Equivalent conceptual form:
SUBS    R2, R0, R1        @ R2 = R0 - R1, and flags updated
```

Unsigned vs signed meaning

- Unsigned comparisons typically use **C and Z**.
- Signed comparisons typically use **N and V** (and Z).

Example: unsigned compare (HI/LS) vs signed compare (GT/LT)

```
    CMP     R0, R1
    BHI     unsigned_r0_gt_r1     @ unsigned: (C==1 && Z==0)
    BLS     unsigned_r0_le_r1

    CMP     R0, R1
    BGT     signed_r0_gt_r1       @ signed: (Z==0 && N==V)
    BLT     signed_r0_lt_r1       @ signed: (N!=V)
```

4.3 Flag Updates and Instruction Effects

Not every instruction updates flags. In ARM 32-bit, many data-processing instructions have a flag-setting form (commonly expressed using an `S` suffix). The discipline is:

- Use flag-setting instructions only when you intend to **consume the flags**.
- Avoid accidental flag clobbering before a conditional branch or conditional instruction.

Example: ADD vs ADDS

```
    ADD     R2, R0, R1        @ flags not updated (typical form)
    ADDS    R2, R0, R1        @ flags updated (N,Z,C,V updated per result)
```

Example: loop counter with SUBS (canonical pattern)

```
    @ R1 = count
loop:
    @ ... body ...
    SUBS    R1, R1, #1        @ updates flags based on new count
    BNE     loop              @ uses Z flag
```

Example: accidental flag clobbering

```
    CMP     R0, #0
    ADDS    R1, R1, #1        @ WRONG if you still needed CMP flags
    BEQ     is_zero           @ BEQ now tests flags from ADDS, not CMP
```

Corrected:

```
    CMP     R0, #0
    BEQ     is_zero
    ADD     R1, R1, #1        @ safe: does not overwrite flags (in this
    ↪  form)
```

Example: using flags immediately (tight, readable)

```
    CMP     R0, R1
    MOVEQ   R2, #0            @ if equal
    MOVLT   R2, #-1           @ if signed less-than
    MOVGT   R2, #1            @ if signed greater-than
```

4.4 Flag Preservation Rules

CPSR flags are **global architectural state** for the currently executing context. They are not automatically preserved across:

- most arithmetic/logic instructions in flag-setting form
- comparisons (`CMP`, `CMN`, `TST`, `TEQ`)
- many instructions that have an `S` variant

Core rules for disciplined code:

- If you need flags, **consume them immediately**.
- Assume a function call may destroy flags unless a strict convention promises otherwise (do not rely on it in fundamentals).
- Avoid long sequences between flag-producing instruction and flag-consuming control flow.

Example: consume flags immediately

```
    CMP     R0, #0
    BEQ     zero_case
    @ safe to do non-flag-setting work here
```

Example: preserve decision result in a register instead of preserving flags

Instead of trying to “save flags”, convert the decision into a value:

```
    CMP     R0, #0
    MOVEQ   R2, #1            @ R2 = (R0==0) ? 1 : 0
    MOVNE   R2, #0
    @ later code uses R2, not CPSR flags
```

This is the recommended discipline for maintainable assembly: CPSR is for short-lived decisions, registers are for durable state.

4.5 Common CPSR Misunderstandings

Misunderstanding 1: Carry and overflow mean the same thing

They represent different arithmetic domains:

- **C** is about **unsigned** arithmetic (carry / no-borrow).
- **V** is about **signed** arithmetic overflow.

Example: C can change without V, and V can change without C

```
@ Unsigned carry example: 0xFFFFFFFF + 1 -> 0 with carry
MOV     R0, #0
MVN     R0, R0            @ R0 = 0xFFFFFFFF
ADDS    R1, R0, #1        @ R1 = 0, C=1, Z=1, V=0

@ Signed overflow example: 0x7FFFFFFF + 1 -> 0x80000000 with V set
MOV     R0, #0x7F
LSL     R0, R0, #24       @ R0 = 0x7F000000
ORR     R0, R0, #0xFF
LSL     R0, R0, #8
ORR     R0, R0, #0xFF
LSL     R0, R0, #8
ORR     R0, R0, #0xFF     @ R0 = 0x7FFFFFFF (constructed without
↪  literals)
ADDS    R1, R0, #1        @ V=1 (signed overflow), C depends on
↪  unsigned carry-out (here C=0)
```

Misunderstanding 2: CMP is “just compare” with no arithmetic meaning

`CMP` is subtraction that updates flags. Understanding it as subtraction is essential for correct unsigned/signed interpretation.

Misunderstanding 3: Flags survive across unrelated instructions

Flags are overwritten frequently. If you need a prior comparison, branch/conditionalize immediately or materialize the decision into a register.

Misunderstanding 4: Using S-variants everywhere is harmless

Overusing flag-setting instructions increases the chance of accidental clobbering and makes code fragile. Use `S` forms only when flags are intentionally consumed.

Misunderstanding 5: Treating CPSR as a general-purpose variable

CPSR is system state. In disciplined assembly, CPSR flags are **ephemeral control signals**, not long-lived program variables. Long-lived state should be stored in registers or memory with explicit save/restore boundaries.

Chapter 5

Conditional Execution Model

5.1 ARM Condition Codes

ARM 32-bit defines a rich set of **condition codes** that allow most instructions to be conditionally executed based on CPSR flags. Each condition evaluates a Boolean expression over **N, Z, C, V**.
Conceptually, conditions fall into categories:

- **Equality**: EQ (Z==1), NE (Z==0)
- **Unsigned comparisons**: HI, HS/CS, LO/CC, LS
- **Signed comparisons**: GT, GE, LT, LE
- **Flag tests**: MI (N==1), PL (N==0), VS (V==1), VC (V==0)
- **Always**: AL

Example: mapping flags to intent

```
    CMP     R0, R1
    BEQ     equal_case          @ Z==1
    BHI     unsigned_gt         @ C==1 && Z==0
    BLT     signed_lt           @ N!=V
```

Correct usage requires selecting the condition that matches the **data domain** (unsigned vs signed). Mixing domains yields logically correct code only by accident.

5.2 Conditional Instruction Encoding

In ARM 32-bit (ARM state), most data-processing instructions include a **condition field**. If the condition evaluates false, the instruction is treated as a **no-op** with respect to architectural state (no register write, no flag update).

Example: conditional data movement

```
    CMP     R0, #0
    MOVEQ   R1, #0              @ if R0==0
    MOVNE   R1, #1              @ if R0!=0
```

Example: conditional arithmetic

```
    CMP     R2, R3
    ADDGT   R4, R4, #5          @ signed greater-than
    ADDLE   R4, R4, #1          @ signed less-or-equal
```

Encoding discipline:

- Condition applies to the **entire instruction**.

- If the condition fails, **no side effects** occur.
- Flag-setting variants (`S`) update flags *only if executed*.

5.3 Flag-Driven Execution Flow

Conditional execution enables a style where **flags drive short control decisions** without branches. The canonical pattern is:

- Produce flags (`CMP`, `SUBS`, `TST`, etc.).
- Immediately consume flags with conditional instructions.

Example: saturating increment

```
    @ R0 holds value, saturate at 100
    CMP     R0, #100
    ADDLT   R0, R0, #1        @ increment only if R0 < 100 (signed)
```

Example: select without branching

```
    @ R0 = x, R1 = y, select max(x,y) into R2 (signed)
    CMP     R0, R1
    MOVGE   R2, R0
    MOVLT   R2, R1
```

Example: flag clobber avoidance

```
    CMP     R0, #0
    MOVEQ   R1, #0
    MOVNE   R1, #1
    @ No flag-setting instructions in between
```

Discipline: flags are **short-lived**. Consume them immediately or materialize the decision into a register.

5.4 Conditional Execution vs Branching

Conditional execution and branching are complementary tools.

Branch-based form

```
    CMP     R0, #0
    BEQ     zero
    ADD     R1, R1, #5
    B       done
zero:
    ADD     R1, R1, #1
done:
```

Conditional-execution form

```
    CMP     R0, #0
    ADDEQ   R1, R1, #1
    ADDNE   R1, R1, #5
```

Architectural correctness is identical. Differences are practical:

- Conditional execution avoids branches for **short, local decisions**.
- Branching is clearer for **long or complex control paths**.
- Microarchitectural performance varies; correctness does not.

Example: loop control favors branching

```
    @ R1 = count
loop:
    @ body
    SUBS    R1, R1, #1
    BNE     loop
```

Loops rely on branches; conditional execution shines in straight-line code.

5.5 Advantages and Limitations

Advantages

- **Reduced branching**: fewer control-flow changes for short decisions.
- **Clear intent**: explicit mapping from flags to actions.
- **Predictable behavior**: architectural no-op on failed condition.
- **Compact code**: replaces small if/else blocks.

Limitations

- **Not suitable for long sequences**: readability and maintenance suffer.
- **Flag pressure**: accidental flag clobbering can break logic.
- **State restrictions**: conditional execution applies broadly in ARM state; availability and forms may differ in other states.
- **Performance is implementation-dependent**: do not assume it is always faster than branching.

Guidelines

- Use conditional execution for **short, local, side-effect-free decisions**.
- Use branches for **loops, calls, and multi-block control flow**.
- Produce flags intentionally; consume them immediately.
- Prefer clarity over cleverness; readable code is more reliable.

This conditional execution model is a defining characteristic of ARM 32-bit. Mastery requires precise flag handling and disciplined instruction selection—skills that directly support later topics such as ABI code generation and low-level optimization.

Chapter 6

Load/Store Architecture Fundamentals

6.1 Why ARM Is Load/Store

ARM 32-bit is a **load/store architecture**: arithmetic and logical instructions operate on **register operands**, and memory is accessed only through **explicit load and store** instructions. This is not a style preference; it is the architectural contract.
Practical consequences for assembly programmers:

- Computation is **register-centric**: move data into registers, compute, then store results.
- Address calculation is explicit and structured through addressing modes.
- Correctness depends on understanding **when** memory is read/written and **which** instructions can touch memory.

Example: the three-phase load/compute/store model

```
    @ R0 = pointer p, R1 = value x
    LDR     R2, [R0]          @ R2 = *p       (memory read)
```

```
    ADD     R2, R2, R1          @ R2 = R2 + x   (register compute)
    STR     R2, [R0]            @ *p = R2       (memory write)
```

In a non-load/store ISA, an arithmetic instruction might directly use a memory operand. On ARM, it cannot. This separation makes reasoning about memory effects precise.

6.2 Memory Access Rules

In disciplined ARM 32-bit assembly, only a small family of instructions may read or write memory:

- **Loads**: `LDR`, `LDRB`, `LDRH` and related variants
- **Stores**: `STR`, `STRB`, `STRH` and related variants
- **Multiple transfers**: `LDM`, `STM` (including push/pop style forms)

Core rules:

- Address used by a load/store is computed from a **base register** plus an **offset/index** (depending on addressing mode).
- The architecture defines which byte/halfword/word is transferred and how the destination register is updated.
- The assembly programmer must ensure the address is **valid**, properly aligned when required, and points to accessible memory.

Example: explicit access width

```
    @ R0 points to bytes
    LDRB    R1, [R0]            @ load 8-bit, zero-extended into R1
    LDRH    R2, [R0]            @ load 16-bit, zero-extended into R2
    LDR     R3, [R0]            @ load 32-bit into R3
```

Example: sign vs zero extension discipline

```
    @ R0 points to signed byte and signed halfword
    LDRSB   R1, [R0]          @ load signed 8-bit, sign-extend into R1
    LDRSH   R2, [R0]          @ load signed 16-bit, sign-extend into R2
```

If you use the wrong variant, your arithmetic may become silently incorrect (especially in signed comparisons and indexing logic).

Example: address calculation is explicit

```
    @ R0 = base, R1 = index
    @ Load base[index] where element size is 4 bytes
    LDR     R2, [R0, R1, LSL #2]
```

This is a core ARM idiom: index scaling is part of the addressing mode, not a separate instruction.

6.3 Register-to-Register vs Memory Operations

In ARM 32-bit, nearly all data-processing instructions are **register-to-register** (or register-immediate). Memory cannot be used as an operand in these instructions.

Example: arithmetic is register-only

```
    @ R2 = R0 + R1 (register operation)
    ADD     R2, R0, R1
```

If you want to add a memory value to a register, you must load first:

```
    @ R0 = pointer p, want: R2 = R2 + *p
    LDR     R1, [R0]          @ R1 = *p
    ADD     R2, R2, R1        @ R2 += R1
```

Example: register staging clarifies memory side effects

```
    @ Update a struct field: *(p+8) += x
    @ R0 = p, R1 = x
    LDR     R2, [R0, #8]      @ load field (read)
    ADD     R2, R2, R1        @ compute
    STR     R2, [R0, #8]      @ store field (write)
```

This pattern makes memory effects explicit and auditable: one read, one write, no hidden memory operands.

6.4 Side Effects and Ordering Constraints

Loads and stores have observable side effects: they interact with memory and with the outside world when memory-mapped I/O is involved. Even when you are not writing kernel code, disciplined assembly must respect these facts.

Side-effect categories

- **Architectural state side effects**: register write-back in certain addressing modes.
- **Memory side effects**: reads/writes to RAM or device-mapped regions.
- **Flag side effects**: most loads/stores do not update flags, but do not assume a universal rule; keep flag logic isolated.

Example: write-back changes the base register

```
    @ R0 points to an array
    LDR     R1, [R0], #4      @ R1 = *R0; then R0 = R0 + 4  (post-index
    ↪  write-back)
```

If you forget that write-back occurred, later addresses will be wrong.

Example: pre-indexed with write-back

```
    @ Advance pointer then load
    LDR     R1, [R0, #4]!     @ R0 = R0 + 4; then R1 = *R0
```

The discipline is: treat write-back forms as **both a load/store and a pointer update**. Do not mix them casually with other pointer arithmetic unless you are very explicit.

Ordering constraints (conceptual, correctness-first)

At the ISA level, a single-threaded program observes memory effects in program order. However, when interacting with:

- device memory (memory-mapped I/O),
- concurrency,
- interrupts/exceptions,

you must assume that ordering and visibility rules become part of correctness. This booklet does not teach the full system memory model, but it enforces safe assembly habits:

- Keep memory-mapped I/O access sequences explicit and isolated.
- Avoid “clever” reordering of loads/stores around flag-dependent control flow.
- Treat multi-instruction read-modify-write sequences as **non-atomic** unless you later use proper atomic/locking mechanisms.

Example: read-modify-write is not atomic

```
    @ *(p) += 1  (not atomic across threads/interrupts)
    LDR     R1, [R0]
    ADD     R1, R1, #1
    STR     R1, [R0]
```

This sequence is correct for single-threaded logic, but it is not safe as an atomic increment if another agent can modify the same memory concurrently.

Example: disciplined separation of concerns

```
    @ Good practice: separate address evolution, data movement, and
    ↪  decisions
    LDR     R2, [R0]          @ data
    CMP     R2, #0            @ decision flags
    BEQ     done              @ control flow
    ADD     R2, R2, #1        @ compute
    STR     R2, [R0]          @ commit
done:
```

Summary discipline checklist

- Only load/store instructions touch memory; everything else is register compute.
- Choose the correct width and sign-extension form.
- Treat write-back addressing as a side effect on the base register.
- Consume flags immediately; avoid accidental clobbering between compare and branch/conditional instruction.
- Remember that multi-instruction updates are not atomic when concurrency or interrupts exist.

Chapter 7

Addressing Modes

7.1 Immediate Addressing

ARM 32-bit provides immediate operands for many data-processing instructions. Immediate addressing means the operand is encoded directly in the instruction, not loaded from memory. This is crucial for tight code: constants, masks, increments, and small offsets can be used without extra loads.

Example: arithmetic with immediate

```
    ADD     R0, R0, #1          @ R0 += 1
    SUB     R1, R1, #16         @ R1 -= 16
```

Example: bit manipulation with immediate masks

```
    ORR     R2, R2, #0x1        @ set bit 0
    BIC     R3, R3, #0x4        @ clear bit 2 (bit clear)
```

Example: immediate in address calculation via offset addressing

Immediate values are also used inside memory addressing forms:

```
    LDR     R0, [R1, #12]     @ load word from address (R1 + 12)
```

Discipline: immediate operands are ideal for small constants and offsets; large constants require construction or literal loading (addressed later in the series).

7.2 Register Addressing

Register addressing means the operand comes from a register. This is the core of ARM data processing: most operations are register-to-register.

Example: register-to-register arithmetic

```
    ADD     R2, R0, R1        @ R2 = R0 + R1
    EOR     R3, R3, R4        @ R3 = R3 XOR R4
```

For memory operations, register addressing appears as using a base register and optionally an index register.

Example: register offset addressing in load/store

```
    @ R0 = base, R1 = offset
    LDR     R2, [R0, R1]      @ load from (R0 + R1)
```

Example: scaled register indexing (array element access)

```
    @ R0 = base address, R1 = index
    @ element size = 4 bytes
    LDR     R2, [R0, R1, LSL #2]   @ R2 = base[index]
```

This is a key ARM idiom: index scaling is integrated into the addressing mode.

7.3 Offset Addressing

Offset addressing forms compute an effective address using a base register plus an offset. The offset can be immediate or register-based, and it does not necessarily update the base register.

Example: immediate offset (no write-back)

```
    @ Load a struct field at offset 8
    LDR     R1, [R0, #8]      @ R1 = *(R0 + 8)
    STR     R2, [R0, #12]     @ *(R0 + 12) = R2
```

Example: register offset (no write-back)

```
    @ R0 = base, R3 = dynamic offset
    LDR     R1, [R0, R3]      @ R1 = *(R0 + R3)
```

Example: scaled index offset (no write-back)

```
    @ array of 32-bit elements
    LDR     R2, [R0, R1, LSL #2]
```

Discipline: when you use offset addressing without write-back, you are expressing: *compute address, access memory, base remains unchanged*.

7.4 Pre-indexed and Post-indexed Modes

Indexing modes define **when** the base register is updated relative to the memory access.

Pre-indexed addressing (compute first, then access)

Pre-indexed means the effective address is computed before the load/store. It can optionally include write-back.

```
    @ Pre-index without write-back (common conceptual form)
    LDR     R1, [R0, #4]      @ R1 = *(R0 + 4), R0 unchanged
```

Pre-index with write-back

```
    @ The '!' means write-back: update base to the effective address
    LDR     R1, [R0, #4]!     @ R0 = R0 + 4; then R1 = *R0
```

Post-indexed addressing (access first, then update)

Post-indexed means the memory access uses the original base register value, and then the base register is updated afterward.

```
    LDR     R1, [R0], #4      @ R1 = *R0; then R0 = R0 + 4
```

These forms are widely used for iteration because they combine memory access and pointer progression.

Example: iterate a word array using post-index

```
    @ R0 = ptr, R1 = count, R2 = sum
    MOV     R2, #0
loop:
    LDR     R3, [R0], #4      @ load then advance
    ADD     R2, R2, R3
    SUBS    R1, R1, #1
    BNE     loop
```

Example: reverse traversal using pre-index write-back

```
    @ R0 points just past end of array (end pointer)
    @ Each iteration: move back then load
    LDR     R3, [R0, #-4]!    @ R0 = R0 - 4; R3 = *R0
```

7.5 Write-back Semantics

Write-back means the base register is modified as a side effect of the addressing mode. This has major correctness implications.

Key rules

- Write-back updates the base register to the computed effective address (pre-index) or base+offset after access (post-index).
- Write-back changes the register value even if you do not use it later; it is still a side effect that can break logic if overlooked.
- Do not combine write-back with separate pointer updates unless you are explicitly modeling both.

Example: write-back vs manual pointer update (avoid double-advance)

```
    @ WRONG: post-index already advances R0; adding again double-advances
    LDR     R1, [R0], #4
    ADD     R0, R0, #4        @ wrong: R0 advanced twice
```

Correct (pick one):

```
    @ Option A: use post-index only
    LDR     R1, [R0], #4
```

```
    @ Option B: no write-back, update explicitly
    LDR     R1, [R0]
    ADD     R0, R0, #4
```

Example: safe pointer+data discipline

```
    @ R0 = ptr, R1 = count
loop2:
    LDR     R2, [R0], #4      @ side effect: ptr advanced
    @ R0 now points to next element
    SUBS    R1, R1, #1
    BNE     loop2
```

Summary discipline checklist

- Use immediate offsets for fixed fields and small strides.
- Use scaled register offsets for indexed arrays.
- Prefer post-index write-back for forward iteration.
- Prefer pre-index write-back for reverse iteration.
- Treat write-back as a real side effect: track it like you track register clobbers.

Chapter 8

Data Movement and Alignment

8.1 Byte, Halfword, and Word Access

ARM 32-bit load/store instructions are explicitly typed by access width. Correct assembly requires selecting the instruction that matches the data layout in memory.

- **Byte (8-bit)**: `LDRB`, `STRB`
- **Halfword (16-bit)**: `LDRH`, `STRH`
- **Word (32-bit)**: `LDR`, `STR`

Loads into registers place the value into a 32-bit register. For smaller widths, loads typically **zero-extend** unless a signed-load variant is used.

Example: zero-extension vs sign-extension

```
    @ R0 points to memory
    LDRB    R1, [R0]          @ unsigned byte -> zero-extended into R1
```

```
    LDRH    R2, [R0]          @ unsigned halfword -> zero-extended into R2

    LDRSB   R3, [R0]          @ signed byte -> sign-extended into R3
    LDRSH   R4, [R0]          @ signed halfword -> sign-extended into R4
```

Example: writing narrow fields without corrupting neighbors

```
    @ Store only one byte
    STRB    R1, [R0]          @ writes low 8 bits of R1 to memory

    @ Store only one halfword
    STRH    R2, [R0, #2]      @ writes low 16 bits of R2 to memory
```

Discipline: never use `STR` when you intend to update a byte field. A 32-bit store overwrites four bytes and can destroy adjacent fields.

8.2 Alignment Requirements

Alignment is the constraint that an address must be a multiple of the access size for naturally aligned transfers:

- Byte: alignment always satisfied.
- Halfword: typically requires address divisible by 2.
- Word: typically requires address divisible by 4.

Why it matters:

- Misalignment can trigger exceptions on some systems/configurations.
- Even when allowed, misalignment can change behavior or reduce performance.
- Correct assembly assumes alignment unless explicitly handling unaligned layouts.

Example: alignment-aware layout

```
@ Suppose a struct layout:
@ offset 0: uint32_t a
@ offset 4: uint16_t b
@ offset 6: uint8_t  c
@ offset 7: padding (for 4-byte alignment of next field)

LDR     R1, [R0, #0]      @ a (word aligned)
LDRH    R2, [R0, #4]      @ b (halfword aligned)
LDRB    R3, [R0, #6]      @ c (byte)
```

Example: computing alignment test in assembly

```
@ Test if R0 is word-aligned (R0 % 4 == 0)
TST     R0, #3
BNE     not_aligned
@ aligned path
```

8.3 Unaligned Access Behavior

Unaligned behavior is **not a safe assumption**. The architectural reality is that whether unaligned word/halfword accesses are supported, trapped, or transformed depends on system configuration and the specific ARM core/profile and control settings.
Therefore, disciplined assembly follows one of two correct approaches:

- **Approach A (preferred)**: enforce alignment and treat misalignment as an error or slow path.
- **Approach B**: explicitly implement unaligned loads/stores using byte operations and shifts.

Example: safe aligned/unaligned split

```
    @ R0 = pointer to 32-bit value (may be unaligned)
    TST     R0, #3
    BNE     slow_unaligned
    LDR     R1, [R0]          @ fast aligned load
    B       done

slow_unaligned:
    @ Manual unaligned load into R1 using bytes (little-endian assumed
    ↪  here)
    LDRB    R1, [R0, #0]
    LDRB    R2, [R0, #1]
    LDRB    R3, [R0, #2]
    LDRB    R4, [R0, #3]
    ORR     R1, R1, R2, LSL #8
    ORR     R1, R1, R3, LSL #16
    ORR     R1, R1, R4, LSL #24

done:
```

This is architecturally explicit: it works regardless of unaligned support, at the cost of extra instructions.

Example: manual unaligned 16-bit load

```
    @ R0 = pointer to 16-bit value (may be unaligned)
    LDRB    R1, [R0, #0]
    LDRB    R2, [R0, #1]
    ORR     R1, R1, R2, LSL #8
```

8.4 Endianness Considerations

Endianness determines the byte order used to represent multi-byte values in memory.

- **Little-endian**: lowest-address byte is the least significant byte.
- **Big-endian**: lowest-address byte is the most significant byte.

Most modern ARM systems are configured little-endian, but assembly that handles raw bytes must be explicit about endianness assumptions.

Example: assembling a 32-bit word from bytes

Little-endian assembly (byte 0 is least significant):

```
    @ R0 points to 4 bytes: b0 b1 b2 b3
    LDRB    R1, [R0, #0]
    LDRB    R2, [R0, #1]
    LDRB    R3, [R0, #2]
    LDRB    R4, [R0, #3]
    ORR     R1, R1, R2, LSL #8
    ORR     R1, R1, R3, LSL #16
    ORR     R1, R1, R4, LSL #24
```

For big-endian, the shifts would be reversed. The discipline is: if you write byte-assembly code, document and enforce the assumed endianness.

Example: swapping endianness (byte reverse concept)

When you must convert between byte orders, use byte-level logic (or architecture-supported byte-reversal instructions where available in the profile), but always verify the correctness using test vectors.

```
@ Conceptual byte swap using shifts/masks (portable)
@ Input in R0, output in R1
AND     R1, R0, #0xFF
MOV     R1, R1, LSL #24

AND     R2, R0, #0xFF00
MOV     R2, R2, LSL #8
ORR     R1, R1, R2

AND     R2, R0, #0xFF0000
MOV     R2, R2, LSR #8
ORR     R1, R1, R2

AND     R2, R0, #0xFF000000
MOV     R2, R2, LSR #24
ORR     R1, R1, R2
```

8.5 Memory Safety at Assembly Level

Assembly provides no automatic memory safety. Correctness depends on disciplined rules:

- Never access memory outside the valid region.
- Keep pointer arithmetic explicit and auditable.
- Match access width to the actual object layout.
- Respect alignment; handle unaligned only via explicit safe paths.
- Avoid accidental overwrites: use `STRB/STRH` for narrow fields.

Example: bounds-checked byte copy (conceptual safety pattern)

```
    @ Copy N bytes from src to dst
    @ R0 = dst, R1 = src, R2 = count
copy_loop:
    CMP     R2, #0
    BEQ     copy_done
    LDRB    R3, [R1], #1
    STRB    R3, [R0], #1
    SUB     R2, R2, #1
    B       copy_loop
copy_done:
```

Example: preventing width mismatch overwrite

```
    @ Suppose [R0] is a packed byte field.
    @ WRONG: STR overwrites 4 bytes.
    @ STR     R1, [R0]

    @ Correct: store only one byte.
    STRB    R1, [R0]
```

Example: alignment-aware word copy with fallback

```
    @ Copy 32-bit words when aligned; otherwise copy bytes.
    @ R0 = dst, R1 = src, R2 = word_count

    ORR     R3, R0, R1
    TST     R3, #3
    BNE     copy_bytes

copy_words:
    CMP     R2, #0
```

```
    BEQ     done
    LDR     R4, [R1], #4
    STR     R4, [R0], #4
    SUBS    R2, R2, #1
    BNE     copy_words
    B       done

copy_bytes:
    @ Convert word_count to byte_count (word_count * 4)
    MOV     R2, R2, LSL #2
byte_loop:
    CMP     R2, #0
    BEQ     done
    LDRB    R4, [R1], #1
    STRB    R4, [R0], #1
    SUB     R2, R2, #1
    B       byte_loop

done:
```

This pattern is correctness-first: it respects alignment constraints and remains valid even when unaligned word loads are not supported.

Chapter 9

Instruction Sequencing Discipline

9.1 Instruction Dependencies

Correct assembly is fundamentally about respecting **dependencies**: when one instruction produces a value or state that a later instruction consumes.
In ARM 32-bit, the most important dependency classes are:

- **Register data dependency**: a later instruction reads a register written earlier.
- **Address dependency**: a load/store depends on an address computed earlier.
- **Memory dependency**: a later load depends on a prior store (same location) in program logic.
- **Control dependency**: a later instruction is reached only if a prior branch condition is taken.
- **Flag dependency**: a later conditional instruction/branch consumes CPSR flags produced earlier.

Example: register data dependency

```
    ADD     R2, R0, R1        @ produces R2
    EOR     R3, R2, #0xFF     @ consumes R2
```

Example: address dependency

```
    ADD     R0, R0, #16       @ produces new pointer
    LDR     R1, [R0, #4]      @ consumes pointer as base address
```

Example: memory dependency (read after write)

```
    STR     R2, [R0]          @ write *p
    LDR     R3, [R0]          @ later read *p (expects the stored value)
```

Discipline: keep related producer/consumer instructions close together and avoid interleaving unrelated work that may clobber registers or flags.

9.2 Flag Dependency Hazards

Flags are a shared global state in CPSR. A **flag hazard** occurs when the flags you intend to use are overwritten before you consume them.
Hazard pattern:

- instruction A sets flags
- instruction B (unintentionally) sets flags again
- instruction C branches/conditionalizes based on flags, but now sees B's flags, not A's

Example: classic hazard

```
    CMP     R0, #0
    ADDS    R1, R1, #1        @ clobbers flags
    BEQ     zero_case         @ tests ADDS flags, not CMP flags
```

Correct patterns:

Pattern 1: consume flags immediately

```
    CMP     R0, #0
    BEQ     zero_case
    ADD     R1, R1, #1
```

Pattern 2: avoid S-variants unless needed

```
    CMP     R0, #0
    ADD     R1, R1, #1        @ does not update flags in this form
    BEQ     zero_case         @ safe: still tests CMP flags
```

Pattern 3: materialize decision into a register

```
    CMP     R0, #0
    MOVEQ   R2, #1
    MOVNE   R2, #0
    @ Later logic uses R2; flags no longer required
```

Discipline: flags are for **short-lived decisions**. Registers are for **durable state**.

9.3 Ordering Rules

At the architectural level for single-threaded code, instructions appear to execute in program order. However, assembly must still respect ordering rules in practice:

- A value must be produced before it is consumed.
- Memory updates in multi-instruction sequences are **not atomic**.
- Loads/stores with write-back introduce additional register updates that must be accounted for.

Example: write-back ordering is part of correctness

```
    @ Post-index: access then update
    LDR     R1, [R0], #4      @ R1 = *R0; then R0 += 4
    LDR     R2, [R0]          @ uses updated R0
```

Example: pre-index with write-back ordering

```
    LDR     R1, [R0, #4]!     @ R0 += 4; then R1 = *R0
    LDR     R2, [R0]          @ uses same updated base
```

Example: read-modify-write sequence is ordered but not atomic

```
    LDR     R1, [R0]          @ read
    ADD     R1, R1, #1        @ modify
    STR     R1, [R0]          @ write
```

This is correct for single-threaded logic, but another agent (thread/interrupt) can observe intermediate states. Treat such sequences as **non-atomic** unless later booklets introduce atomic mechanisms.

9.4 Predictable Execution Patterns

Predictable assembly favors **standard, recognizable idioms**:

- compare then branch immediately
- loop with `SUBS` + `BNE`
- load/compute/store with clear staging registers
- preserve LR before nested calls

Pattern: compare then branch

```
    CMP     R0, R1
    BEQ     equal
    BHI     unsigned_gt
    @ fall-through
```

Pattern: counter loop

```
    @ R0 = pointer, R1 = count
loop:
    LDR     R2, [R0], #4
    @ ... body ...
    SUBS    R1, R1, #1
    BNE     loop
```

Pattern: conditional select without branch

```
    @ R0 = a, R1 = b, result in R2 (signed max)
    CMP     R0, R1
    MOVGE   R2, R0
    MOVLT   R2, R1
```

Pattern: call discipline

```
PUSH    {LR}
BL      helper
POP     {LR}
BX      LR
```

Predictability here is a correctness tool: these idioms make it hard to accidentally clobber flags, lose the return address, or mis-handle pointer updates.

9.5 Architectural Constraints

Instruction sequencing must respect constraints imposed by the ISA:

- Only load/store instructions access memory; arithmetic is register-only.
- CPSR flags are overwritten by comparisons and S-variants; treat flags as volatile control state.
- PC/LR/SP are special registers with control-flow and stack invariants.
- Write-back addressing modes modify the base register as a side effect.

Constraint example: do not “compute on memory”

```
@ Wrong mental model: "ADD [R0], #1" does not exist on ARM
@ Correct staging:
LDR     R1, [R0]
ADD     R1, R1, #1
STR     R1, [R0]
```

Constraint example: keep flag producers adjacent to consumers

```
    CMP     R2, #0
    BEQ     done              @ consume immediately
    @ safe: flag-neutral work or explicit decisions after the branch
```

Constraint example: avoid losing LR

```
    @ Any nested BL clobbers LR; save it first
    PUSH    {LR}
    BL      g
    POP     {LR}
    BX      LR
```

Discipline checklist

- Track register liveness; do not reuse registers prematurely.
- Treat flags as volatile; consume immediately or materialize decisions.
- Treat write-back as a real side effect; never “double-advance” pointers.
- Keep code structured: producer/consumer proximity, clear labels, standard idioms.
- Prefer correctness patterns first; performance tuning comes after correctness is proven.

Chapter 10

Common Errors and Dangerous Assumptions

10.1 Misusing LR

LR (R14) holds the return address after `BL`/`BLX`. The most common ARM assembly failures happen when LR is treated like a normal scratch register or when nested calls overwrite it.

Error 1: Non-leaf function that does not preserve LR

```
bad_f:
    @ WRONG: LR will be overwritten by BL g
    BL      g
    BX      LR                @ returns to the wrong place
```

Correct:

```
good_f:
    PUSH    {LR}
```

```
    BL      g
    POP     {LR}
    BX      LR
```

Error 2: Using LR as temporary storage

```
bad_lr_temp:
    MOV     LR, R0              @ WRONG: destroys return address
    @ ...
    BX      LR                  @ branches to arbitrary value
```

Correct discipline: LR is **return state**. Use a GPR (`R0--R12`) or stack slots for temporaries.

Error 3: Mixing return mechanisms without discipline

```
mixed_return:
    PUSH    {LR}
    @ ...
    BX      LR                  @ WRONG: LR on stack not restored; stack
    ↪  becomes imbalanced
```

Correct: choose one strategy and complete it.

```
consistent_return:
    PUSH    {LR}
    @ ...
    POP     {LR}
    BX      LR
```

Or return by restoring PC:

```
consistent_return2:
    PUSH    {LR}
    @ ...
    POP     {PC}                @ return by loading PC
```

10.2 Incorrect PC Assumptions

PC (R15) is not a normal register. Writing PC transfers control. Reading PC yields an architecturally defined value tied to the instruction stream, but not a general-purpose "current address" you can casually compute with.

Error 1: Treating PC like a general data register

```
bad_pc_math:
    ADD     R0, PC, #4        @ often used for position logic, but unsafe
     ↪  as a general habit
    @ R0 may not mean what you assume across states/encodings
```

Correct discipline in this booklet:

- Use PC writes only for explicit branches/returns.
- Use labels and assembler-supported relocation for addresses (introduced later in the series).
- Avoid "PC arithmetic" tricks in fundamentals code.

Error 2: Returning by writing PC from an untrusted register

```
bad_return:
    MOV     PC, R0            @ WRONG if R0 is not a verified return target
```

Correct: return only via the established return state (`BX LR` or restoring `PC` from the stack).

```
safe_return:
    BX      LR
```

10.3 CPSR Corruption

CPSR flags drive conditional execution and branching. “Corruption” in normal application-level assembly usually means **accidentally overwriting flags** between a flag-producing instruction and a flag-consuming instruction.

Error 1: Flag clobber between CMP and branch

```
bad_flags:
    CMP     R0, #0
    ADDS    R1, R1, #1        @ clobbers flags
    BEQ     zero_case         @ tests ADDS flags, not CMP flags
```

Correct patterns:

```
good_flags_1:
    CMP     R0, #0
    BEQ     zero_case
    ADD     R1, R1, #1
```

```
good_flags_2:
    CMP     R0, #0
    ADD     R1, R1, #1        @ no flags updated
    BEQ     zero_case         @ still uses CMP flags
```

Error 2: Using S-variants everywhere

```
bad_overuse_s:
    ADDS    R2, R2, #1
    SUBS    R3, R3, #4
    ANDS    R4, R4, R5
    @ flags are continually overwritten; later conditions become fragile
```

Correct discipline: use `S` only when you intend to consume flags immediately.

Error 3: Assuming flags survive a call

```
bad_call_flags:
    CMP     R0, #0
    BL      helper            @ assume flags preserved (unsafe assumption)
    BEQ     zero_case         @ may not test CMP result anymore
```

Correct: branch/conditionalize before calls or materialize the decision:

```
good_call_flags:
    CMP     R0, #0
    MOVEQ   R2, #1
    MOVNE   R2, #0
    BL      helper
    CMP     R2, #1
    BEQ     zero_case
```

10.4 Misaligned Memory Access

Misalignment is one of the most common sources of crashes or silent wrong behavior in low-level code. Halfword/word loads generally require natural alignment (2/4-byte), and unaligned support is configuration- and core-dependent.

Error 1: Word load from unaligned address

```
bad_unaligned:
    @ R0 may be unaligned
    LDR     R1, [R0]          @ may fault or behave unexpectedly on some
    ↪  systems
```

Correct: enforce alignment or use a safe slow path.

```
good_align_check:
    TST     R0, #3
    BNE     slow_unaligned
    LDR     R1, [R0]
    B       done

slow_unaligned:
    @ Portable unaligned load (little-endian assumed)
    LDRB    R1, [R0, #0]
    LDRB    R2, [R0, #1]
    LDRB    R3, [R0, #2]
    LDRB    R4, [R0, #3]
    ORR     R1, R1, R2, LSL #8
    ORR     R1, R1, R3, LSL #16
    ORR     R1, R1, R4, LSL #24
done:
```

Error 2: Wrong-width store overwriting neighbors

```
bad_width_store:
    @ Want to store a byte field, but this overwrites 4 bytes
    STR     R1, [R0]
```

Correct:

```
good_width_store:
    STRB    R1, [R0]          @ store only low 8 bits
```

10.5 Invalid Load/Store Patterns

Load/store instructions are precise: they define access width, addressing mode, and optional write-back. Many bugs come from misunderstanding write-back and base register evolution.

Error 1: Double-advancing a pointer (write-back + manual update)

```
bad_double_advance:
    LDR     R1, [R0], #4      @ post-index already advances R0
    ADD     R0, R0, #4        @ WRONG: advances twice
```

Correct (choose one):

```
good_advance_a:
    LDR     R1, [R0], #4
```

```
good_advance_b:
    LDR     R1, [R0]
    ADD     R0, R0, #4
```

Error 2: Confusing pre-index and post-index semantics

```
    @ Pre-index with write-back:
    @ R0 updated first, then used
    LDR     R1, [R0, #4]!

    @ Post-index:
    @ R0 used first, then updated
    LDR     R2, [R0], #4
```

If you swap these forms unintentionally, your code reads the wrong element or skips/duplicates elements.

Error 3: Using the wrong sign-extension form

```
bad_sign:
    @ byte is signed in memory, but loaded as unsigned
    LDRB    R1, [R0]          @ zero-extends; negative values become large
    ↪  positives
```

Correct:

```
good_sign:
    LDRSB   R1, [R0]          @ sign-extends correctly
```

Error 4: Read-modify-write assumed atomic

```
bad_atomic_assumption:
    @ Not atomic across threads/interrupts
    LDR     R1, [R0]
    ADD     R1, R1, #1
    STR     R1, [R0]
```

Correct discipline: treat this as single-threaded logic only. Atomicity requires dedicated mechanisms (introduced in later booklets).

Summary: dangerous assumptions checklist

- LR is not a scratch register; preserve it in non-leaf functions.
- PC is a control-flow register; avoid PC arithmetic tricks in fundamentals.
- Flags are volatile; consume immediately and avoid accidental clobbering.
- Alignment is not optional; validate or implement safe unaligned paths.
- Write-back is a side effect; track it like register clobbers.
- Width and sign-extension must match the memory data model exactly.

Appendices

Appendix A — Minimal Instruction Reference (Conceptual)

This appendix provides a concise, correctness-focused reference to the essential ARM 32-bit instruction classes used throughout this booklet. It is intentionally conceptual and architectural: it explains what each class does and how it behaves, without encoding tables, ABI rules, OS assumptions, or microarchitectural tuning.

Data Processing Instructions

Data processing instructions compute on registers (and immediates). They do not access memory.

- Arithmetic: `ADD`, `SUB`, `RSB`
- Logical: `AND`, `ORR`, `EOR`, `BIC`
- Move/transform: `MOV`, `MVN`
- Shifted operand forms integrated into the instruction

```
ADD     R2, R0, R1              @ R2 = R0 + R1
SUB     R3, R2, #16             @ R3 = R2 - 16
```

```
    EOR     R4, R4, R5              @ R4 = R4 XOR R5
    ADD     R6, R0, R1, LSL #2      @ R6 = R0 + (R1 << 2)
```

Conceptual rule: all computation happens in registers; memory values must be loaded first.

Load and Store Instructions

Load/store instructions are the only instructions that access memory. They define access width, address calculation, and optional base write-back.

- Word: `LDR`, `STR`
- Halfword: `LDRH`, `STRH`
- Byte: `LDRB`, `STRB`
- Signed loads: `LDRSB`, `LDRSH`

```
    LDR     R1, [R0]                @ R1 = *R0
    STR     R2, [R0, #8]            @ *(R0 + 8) = R2

    LDRB    R3, [R0, #1]            @ byte load, zero-extend
    STRB    R3, [R0, #1]            @ byte store

    LDRH    R4, [R0, #2]            @ halfword load, zero-extend
    STRH    R4, [R0, #2]            @ halfword store

    LDRSB   R5, [R0]                @ signed byte load, sign-extend
    LDRSH   R6, [R0]                @ signed halfword load, sign-extend
```

Write-back and indexing:

```
    LDR     R1, [R0], #4            @ post-index: R1 = *R0; then R0 += 4
    LDR     R2, [R0, #4]!           @ pre-index:  R0 += 4; then R2 = *R0
```

Conceptual rule: load/store always has memory side effects; write-back additionally modifies the base register.

Branch Instructions

Branch instructions update control flow by changing PC.

- `B` — unconditional branch
- `BL` — call (branch with link; LR updated)
- `BX` — branch to register (return/indirect branch)
- `BLX` — indirect call (branch with link to register)

```
    B       target

    BL      func              @ LR = return address; branch to func
    @ ...
func:
    BX      LR                @ return

    BX      R0                @ indirect branch to address in R0
```

Conceptual rule: any `BL`/`BLX` overwrites LR; preserve LR in non-leaf functions.

Flag-Setting Instructions

Flag-setting instructions update CPSR condition flags. Flags are short-lived control signals and must be consumed immediately.

- Compare: `CMP`, `CMN` (sets flags, discards result)
- Test: `TST`, `TEQ` (sets flags based on logical test)
- Data processing with `S` suffix: e.g., `ADDS`, `SUBS`, `ANDS`

```
    CMP     R0, #0
    BEQ     is_zero

    TST     R1, #1
    BNE     bit0_set

    ADDS    R2, R2, #1        @ updates N,Z,C,V
    SUBS    R3, R3, #1        @ common loop counter pattern
    BNE     loop
```

Conceptual rule: avoid accidental flag clobbering between flag production and flag consumption.

Appendix B — Preparation for Next Booklets

This appendix defines the exact architectural and mental readiness expected before progressing to the next booklets in the ARM track. It clarifies what the reader must already understand and practice correctly, ensuring that later topics (ABI, stack frames, OS interaction) are built on solid foundations rather than assumptions.

Readiness for ARM Stack Discipline

Before studying stack frames and procedure prologues, the reader must already be comfortable with the following concepts:

- SP (`R13`) is a special register with invariants, not a general-purpose register.
- Stack growth direction and alignment must be respected consistently.
- Every modification to SP must be balanced and auditable.

You should be able to reason correctly about stack effects even without ABI rules.

```
@ Allocate local space
SUB     SP, SP, #16

@ Use stack slots
STR     R0, [SP, #0]
STR     R1, [SP, #4]

@ Deallocate local space
ADD     SP, SP, #16
```

Required mindset:

- The stack is memory with strict discipline.
- Every push has a matching pop.
- Misaligned or unbalanced stacks lead to undefined behavior at higher levels.

Readiness for ARM Calling Conventions

Calling conventions formalize rules that you are already applying informally in this booklet. Before advancing, you must fully understand:

- `BL`/`BLX` overwrite LR.
- Non-leaf functions must preserve LR explicitly.
- Registers have lifetimes; some values must survive calls, others do not.

You should already be able to write correct call/return logic without knowing which registers are caller-saved or callee-saved.

```
@ Non-leaf function skeleton
PUSH    {LR}
BL      helper
POP     {LR}
BX      LR
```

Required mindset:

- A function call is a controlled control-flow transfer.
- Return correctness is architectural, not optional.
- ABI rules will formalize what you already practice.

Readiness for ABI Interoperability

ABI interoperability means your assembly code can safely interact with compiler-generated code and foreign modules.
Before moving forward, you must already be able to:

- Distinguish architectural rules from ABI rules.
- Preserve control flow, stack integrity, and register state explicitly.
- Avoid relying on undocumented compiler behavior.

Example of ABI-neutral correctness:

```
@ Preserve state explicitly before external interaction
PUSH    {R4, R5, LR}
BL      external_function
POP     {R4, R5, LR}
BX      LR
```

Required mindset:

- ABI is a contract layered on top of the architecture.
- Architecture correctness comes first; ABI correctness builds on it.
- Clean boundaries prevent subtle, hard-to-debug failures.

Readiness for Embedded and OS-Level ARM Code

Embedded systems and OS-level code impose stricter correctness requirements.
Before proceeding, you should already understand:

- Load/store ordering matters when interacting with memory-mapped I/O.
- Read-modify-write sequences are not atomic by default.
- Alignment, access width, and side effects are correctness-critical.

Example of disciplined memory access pattern:

```
    @ Read-modify-write (single-threaded logic)
    LDR     R1, [R0]
    ADD     R1, R1, #1
    STR     R1, [R0]
```

Required mindset:

- Hardware observes memory operations.
- There is no safety net at this level.
- Predictable, explicit code is more important than clever code.

Summary Readiness Checklist

You are ready to move on if you can confidently:

- Track SP, LR, PC, and CPSR effects across instruction sequences.
- Write correct control flow without relying on ABI shortcuts.
- Reason about memory access width, alignment, and side effects.
- Read compiler-generated ARM code and understand why it is correct.

The next booklets will formalize these practices into ABI rules, stack frames, and system-level interactions. This appendix ensures that the transition is additive, not corrective.

References

ARM Architecture Reference Manuals

This booklet is grounded in the architectural behavior defined by the official ARM 32-bit architecture specifications. These manuals define the processor-visible contract: registers, CPSR behavior, instruction semantics, addressing modes, and architectural constraints.
Key conceptual guarantees drawn from these manuals and reflected throughout this booklet:

- The architectural meaning of registers `R0--R15`, including SP, LR, and PC.
- CPSR flag semantics and their role in conditional execution.
- Load/store-only access to memory.
- Precise definitions of addressing modes and write-back behavior.

All examples in this booklet follow architectural rules that remain valid regardless of:

- specific ARM core implementation,
- operating system presence,
- compiler or toolchain choice.

ARM Instruction Set Documentation

Instruction behavior described in this booklet follows the canonical ARM instruction definitions:

- Data processing instruction semantics
- Load/store instruction side effects
- Branch and control-flow behavior
- Flag-setting and conditional execution rules

Examples are written to expose architectural intent rather than encoding detail.

```
    @ Architectural meaning: compute in registers
    ADD     R2, R0, R1

    @ Architectural meaning: explicit memory access
    LDR     R3, [R4]
    STR     R3, [R4]
```

The focus is on **what the instruction does**, not on opcode formats or binary layouts. Encoding tables, instruction widths, and binary representations are intentionally excluded, as they do not affect architectural correctness at this stage.

Compiler-Generated ARM Code Behavior

Modern compilers targeting ARM 32-bit adhere strictly to the architectural rules described in this booklet. Understanding compiler-generated output requires recognizing the same patterns presented here.
Common compiler-emitted patterns explained by this booklet:

- Load/compute/store sequences
- `SUBS` + conditional branch loop structures
- Explicit preservation of LR in non-leaf functions
- Use of conditional execution for short decisions

Example of a compiler-style loop pattern:

```
loop:
    LDR     R2, [R0], #4
    ADD     R3, R3, R2
    SUBS    R1, R1, #1
    BNE     loop
```

Understanding this code requires:

- knowledge of write-back semantics,
- awareness of flag dependencies,
- understanding of control-flow discipline.

This booklet equips the reader to read such output confidently and to reason about its correctness independently of optimization level.

Cross-References to Other Booklets in This Series

This booklet is part of a structured CPU Programming Series and is intentionally scoped to architectural fundamentals.
It directly builds upon:

- execution model fundamentals,

- register and flag semantics,
- binary data representation concepts.

It directly prepares the reader for subsequent booklets covering:

- ARM stack frames and stack discipline
- ARM calling conventions and ABI rules
- Interoperability with C/C++ compilers
- Embedded and OS-level ARM programming

Conceptual dependency example:

```
    @ This booklet explains why this works
    PUSH    {LR}
    BL      helper
    POP     {LR}
    BX      LR
```

Later booklets will formalize:

- which registers must be preserved,
- exact stack layout rules,
- ABI-mandated alignment requirements.

This separation ensures that each booklet adds new knowledge without revising or correcting earlier material, preserving a clean and cumulative learning path.