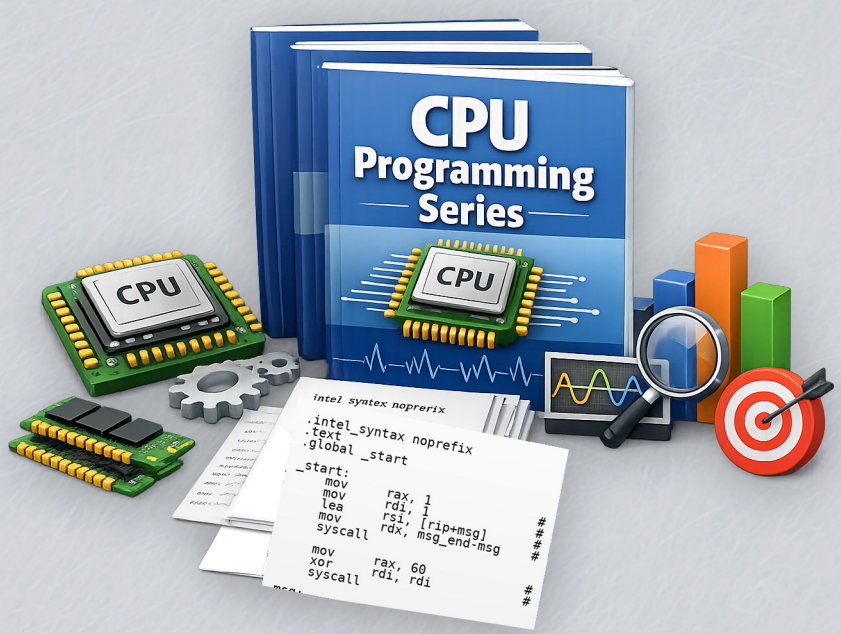
https://simplifycpp.org

CPU Programming Series

RISC-V RV32I Assembly

Clean Architecture, Clean Semantics



15

Prepared by Ayman Alheraki

CPU Programming Series

RISC-V RV32I Assembly

Clean Architecture, Clean Semantics

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Purpose of This Booklet

This booklet provides a rigorous, architecture-first introduction to **RISC-V RV32I assembly programming**. Its purpose is to teach how a clean, modern instruction set operates at the lowest level, focusing on correctness, predictability, and disciplined reasoning rather than shortcuts or platform-specific tricks.
The goal is not memorization of instructions, but understanding:

- how instructions affect architectural state,
- how control flow is constructed explicitly,
- how software correctness emerges from architectural rules.

RV32I is treated as a complete computing model, not as a stepping stone or simplified toy ISA.

Why RISC-V Matters for Programmers

RISC-V matters because it restores clarity to low-level programming. Its design eliminates decades of accumulated legacy behavior and exposes a direct, honest relationship between code and hardware.

For programmers, this means:

- no hidden condition flags,
- no implicit memory access,
- no instruction side effects outside what is explicitly encoded.

Every architectural effect is visible in the instruction stream. As a result, programmers can reason about correctness directly from the code itself, making RISC-V an ideal ISA for systems programming, compiler development, and formal reasoning.

RV32I as a Teaching ISA

RV32I is intentionally minimal yet complete. It contains only what is strictly necessary to execute real programs.
Key teaching properties include:

- register-only arithmetic and logic,
- explicit load/store memory access,
- simple, orthogonal control-flow instructions.

There are no implicit stack operations and no hidden architectural state. For example, a function call is built explicitly using jump-and-link instructions and disciplined register saving:

```
/* Simple function call sequence in RV32I */
jal x1, function   /* save return address in x1 */
...
jalr x0, 0(x1)     /* return to caller */
```

This explicitness makes RV32I ideal for teaching how higher-level language constructs map to machine code.

Scope, Assumptions, and Design Discipline

This booklet focuses strictly on the **RV32I base integer instruction set**. No optional extensions are used.
Assumptions:

- basic familiarity with programming concepts,
- interest in understanding machine-level behavior,
- willingness to reason precisely about state changes.

All examples follow strict design discipline:

- explicit register usage,
- correct stack handling,
- predictable control flow,
- avoidance of undefined or toolchain-specific behavior.

Correctness is always prioritized over cleverness.

How to Read This Booklet

This booklet is meant to be read sequentially. Each chapter builds directly on the architectural rules established earlier.
Readers are encouraged to:

- trace register values manually,
- follow control flow instruction by instruction,

- compare handwritten assembly with compiler-generated output.

All assembly examples use GNU assembler syntax with consistent block-style comments:

```
/* Example: explicit load, compute, store */
lw  x5, 0(x10)      /* load value from memory */
add x5, x5, x11     /* compute in registers */
sw  x5, 0(x10)      /* store result back */
```

The intent is to build confidence in reading, writing, and reasoning about RV32I assembly with precision and architectural awareness.

Chapter 1

RISC-V Philosophy and Architectural Foundations

1.1 RISC Design Principles Revisited

RISC is not a slogan; it is a disciplined architecture strategy. The core principles that matter to programmers are:

- **Simplicity of instruction semantics:** each instruction does one well-defined thing.
- **Regularity:** a small number of instruction formats, predictable decoding, and uniform register usage.
- **Load/store discipline:** arithmetic is performed on registers; memory is accessed only via explicit loads/stores.
- **Compiler friendliness:** the ISA is designed to be a good target for optimizing compilers, not only for hand-written assembly.

RV32I reflects these principles with a fixed 32-bit instruction width, a general-purpose register file, and an explicit separation between computation and memory traffic.

Example: load/store discipline

```
/* Wrong mental model: "add memory to register" does not exist in
 ↪  RV32I */
/* Correct model: load -> compute -> store */

lw   x5, 0(x10)       /* x5 = *(uint32_t*)x10 */
lw   x6, 4(x10)       /* x6 = *(uint32_t*)(x10 + 4) */
add  x7, x5, x6       /* x7 = x5 + x6 */
sw   x7, 8(x10)       /* *(uint32_t*)(x10 + 8) = x7 */
```

This explicitness is not a limitation; it is what makes execution reasoning clean and reliable.

1.2 Open ISA and Long-Term Stability

RISC-V is specified as an open instruction set architecture with a strong emphasis on modularity and forward compatibility. For programmers, the practical consequences are:

- **Stable base:** the base ISA is designed to remain stable across decades.
- **Extensions are explicit:** you opt into M/A/F/D/V and other extensions; the base does not quietly change.
- **Portability by construction:** RV32I code is defined by architectural behavior, not by vendor-specific instruction quirks.

The programmer benefit is a cleaner contract: if you write strictly RV32I, you know exactly what architectural features you are using.

Example: writing extension-free RV32I code

```
/* RV32I-only: multiplication is not available without the M
 ↪  extension */
/* Replace x = a * 10 with: x = (a<<3) + (a<<1) */

slli x6, x5, 3        /* x6 = a * 8 */
slli x7, x5, 1        /* x7 = a * 2 */
add  x6, x6, x7       /* x6 = a * 10 */
```

This forces correct awareness of what the ISA guarantees, and what requires an extension.

1.3 Separation of ISA and Microarchitecture

A key architectural discipline in RISC-V is the strict separation between:

- **ISA (architectural contract):** registers, memory model rules, and instruction semantics.
- **Microarchitecture (implementation):** pipeline depth, caches, predictors, reorder buffers, and execution units.

The same RV32I program must produce the same architectural results regardless of whether it runs on:

- a simple in-order core,
- a deeply pipelined implementation,
- an out-of-order implementation.

This separation is essential for portability and for building correct mental models: as a programmer, you reason from the ISA, not from the microarchitecture.

Example: architectural equivalence despite different implementations

```
/* Architectural behavior: deterministic register results */
addi x5, x0, 1        /* x5 = 1 */
addi x6, x0, 2        /* x6 = 2 */
add  x7, x5, x6       /* x7 = 3 */
/* Pipeline differences do not change x7 = 3 */
```

Performance may vary across implementations, but correctness at the architectural level must not.

1.4 RV32I as the Minimal Executable Core

RV32I is the minimal integer base ISA that can:

- execute real programs,
- express control flow and function calls,
- perform arithmetic and memory access,
- serve as a stable foundation for extensions.

Minimal does not mean weak; it means complete with no unnecessary features. RV32I includes:

- integer arithmetic/logic,
- shifts and comparisons,
- explicit loads/stores,
- conditional branches,
- jump-and-link for calls/returns.

Example: RV32I function call and return (architectural view)

```
/* Caller */
addi x10, x0, 7       /* argument in x10 (a0 by ABI convention) */
jal  x1, func         /* x1 = return address; jump to func */
addi x11, x10, 1      /* use returned value */

/* Callee */
func:
addi x10, x10, 5      /* return value in x10 */
jalr x0, 0(x1)        /* return */
```

Even without any extensions, RV32I can express the fundamental mechanics of higher-level execution.

1.5 Comparison with Legacy ISAs (Conceptual)

This booklet does not treat other ISAs as inferior; it treats them as historically different design points. The conceptual contrast is useful for understanding **why** RV32I feels clean.

Key conceptual contrasts

- **No condition flags:** unlike many legacy ISAs, RV32I comparisons are explicit and branches test register values.
- **Orthogonal encoding:** fixed-width instructions simplify decoding and analysis compared to variable-length encodings.
- **No implicit stack semantics:** calls/returns are explicit jumps with link registers, not implicit magic.

- **Strict load/store model:** arithmetic does not touch memory; memory access is explicit.

Example: branch without flags (RV32I style)

```
/* If (x5 == x6) goto equal; else fall through */
beq  x5, x6, equal
addi x7, x0, 0       /* not equal path */
jal  x0, done
equal:
addi x7, x0, 1       /* equal path */
done:
```

Example: return without a dedicated RET instruction

```
/* Return is a jump via the link register */
jalr x0, 0(x1)       /* PC = x1; no register is written */
```

The result is an ISA that is easier to reason about, easier to teach, and easier to map from compiler IR to machine behavior. That is the meaning of *Clean Architecture, Clean Semantics*.

Chapter 2

RV32I Execution Model

2.1 Instruction Fetch, Decode, Execute

At the architectural level, RV32I executes instructions through a simple and stable contract:

- **Fetch:** the instruction at address `PC` is fetched from instruction memory.
- **Decode:** opcode and fields (register indices, immediates, function bits) determine the operation.
- **Execute:** the operation updates architectural state (registers, memory, and/or `PC`).

RISC-V is designed so that architectural correctness does not depend on the internal pipeline design. Different implementations may be in-order, pipelined, or out-of-order, but must present the same architectural results.

Example: deterministic architectural updates

```
/* Architectural view: each instruction updates state in a
 ↪  well-defined way */
```

```
addi x5, x0, 10       /* x5 = 10 */
addi x6, x0, 20       /* x6 = 20 */
add  x7, x5, x6       /* x7 = 30 */
```

Example: decode selects semantics, not ”implicit modes”

```
/* Same opcode family; funct3/funct7 select operation */
add  x7, x5, x6       /* x7 = x5 + x6 */
sub  x7, x5, x6       /* x7 = x5 - x6 */
```

2.2 Register-Only Computation Model

RV32I follows a strict register computation model:

- Arithmetic and logic operate only on registers.
- Immediates are part of instruction encoding, not memory operands.
- Memory is never modified by an ALU instruction; only explicit stores modify memory.

This rule eliminates a large class of hidden side effects. If an instruction is not a load/store, it does not access memory.

Example: compute a complex expression without touching memory

```
/* Compute: x9 = (x5 + 3) ^ (x6 << 2) */
addi x7, x5, 3        /* x7 = x5 + 3 */
slli x8, x6, 2        /* x8 = x6 << 2 */
xor  x9, x7, x8       /* x9 = x7 ^ x8 */
```

Example: RV32I has no ”add [mem], reg” form

```
/* Correct pattern: load -> compute -> store */
lw   x5, 0(x10)       /* x5 = *(uint32_t*)x10 */
addi x5, x5, 1        /* x5 = x5 + 1 */
sw   x5, 0(x10)       /* *(uint32_t*)x10 = x5 */
```

2.3 Memory Access Discipline

RV32I memory access is explicit and disciplined:

- **Only** loads read memory: `lb, lh, lw, lbu, lhu`.
- **Only** stores write memory: `sb, sh, sw`.
- All addresses are computed as `base register + sign-extended immediate`.
- Loads/stores transfer a specific width (8/16/32 bits) with defined sign or zero extension rules.

Example: signed vs unsigned load semantics

```
/* Memory byte at [x10] = 0xFF */
lb   x5, 0(x10)       /* x5 = 0xFFFFFFFF (sign-extended -1) */
lbu  x6, 0(x10)       /* x6 = 0x000000FF (zero-extended 255) */
```

Example: halfword load and extension

```
/* Memory halfword at [x10] = 0x8001 */
lh   x5, 0(x10)       /* x5 = 0xFFFF8001 (sign-extended) */
lhu  x6, 0(x10)       /* x6 = 0x00008001 (zero-extended) */
```

Example: struct-like field access via base+offset

```
/* Assume x10 points to a record:
   offset 0: uint32_t id
   offset 4: uint16_t flags
   offset 8: uint32_t value
*/
lw   x5, 0(x10)       /* id */
lhu  x6, 4(x10)       /* flags */
lw   x7, 8(x10)       /* value */
```

2.4 Endianness and Data Alignment

Endianness defines byte order in memory for multi-byte values. RV32I systems are commonly little-endian, where the least significant byte is stored at the lowest address.

Example: little-endian layout of a 32-bit word

If `x5 = 0x11223344` and we store it to memory:

```
/* Store a 32-bit value */
sw x5, 0(x10)         /* *(uint32_t*)x10 = 0x11223344 */
```

Then memory at `x10` conceptually contains:

- `[x10+0] = 0x44`
- `[x10+1] = 0x33`
- `[x10+2] = 0x22`
- `[x10+3] = 0x11`

Alignment discipline

Alignment is the rule that multi-byte loads/stores should use naturally aligned addresses:

- halfword (16-bit): address multiple of 2
- word (32-bit): address multiple of 4

Correct low-level code assumes proper alignment unless the environment explicitly guarantees safe unaligned access. This booklet follows strict alignment discipline.

Example: building a 32-bit value from bytes (safe regardless of alignment)

```
/* Load four bytes and assemble a 32-bit little-endian word manually
 ↪  */
lbu  x5, 0(x10)       /* b0 */
lbu  x6, 1(x10)       /* b1 */
lbu  x7, 2(x10)       /* b2 */
lbu  x8, 3(x10)       /* b3 */

slli x6, x6, 8
slli x7, x7, 16
slli x8, x8, 24

or   x5, x5, x6
or   x5, x5, x7
or   x5, x5, x8       /* x5 = assembled 32-bit value */
```

2.5 Program Counter Semantics

The program counter (`PC`) is an architectural concept: it points to the address of the current instruction being executed. RV32I instructions are 32-bit wide, so in straight-line execution:

- `PC` normally advances by `+4` after each instruction.
- Control flow instructions replace the normal `PC+4` with a target address.

There is no general-purpose instruction that directly reads `PC`. Instead, RV32I provides `AUIPC` and `JAL` to construct PC-relative values.

Example: sequential flow (PC advances by 4)

```
/* PC: P
   addi executes at P, then next at P+4, then P+8, ...
*/
addi x5, x0, 1
addi x6, x0, 2
add  x7, x5, x6
```

Example: conditional branch changes PC

```
/* If x5 == x6, PC becomes PC + imm (PC-relative branch) */
beq  x5, x6, equal
addi x7, x0, 0       /* not equal */
jal  x0, done
equal:
addi x7, x0, 1
done:
```

Example: JAL saves return address and jumps

```
/* JAL writes PC+4 into rd, then sets PC to PC + offset */
jal  x1, func          /* x1 = return address; jump to func */
addi x5, x0, 9         /* executes after return */

func:
addi x10, x0, 7
jalr x0, 0(x1)         /* return: PC = x1 */
```

Example: PC-relative address formation with AUIPC

```
/* AUIPC: rd = PC + (imm20 << 12)
   Often paired with ADDI to form a full PC-relative address.
*/
auipc x5, 0            /* x5 = current PC (upper-immediate form) */
addi  x5, x5, 16       /* x5 = PC + 16 (within nearby range) */
```

These rules form the execution model you must internalize: explicit state updates, explicit memory traffic, and explicit control flow.

Chapter 3

Integer Register File (x0–x31)

3.1 General-Purpose Registers Overview

RV32I provides **32 integer registers** named `x0` through `x31`. Each register is **32 bits** wide in RV32. The register file is uniform: instructions select registers by index, and most integer operations can use any register as source or destination.
Key architectural facts:

- Integer registers hold **integers, pointers, addresses, and bitfields**.
- There are **no dedicated accumulator registers** required by the ISA.
- There are **no implicit operands**: if a value is used, you name the register explicitly.
- The ISA specifies **register behavior**; naming conventions (like `a0`, `t0`) belong to ABI usage, not to the ISA itself.

Example: register-only arithmetic is explicit

```
/* x7 = (x5 + x6) - 3 */
```

```
add  x7, x5, x6       /* x7 = x5 + x6 */
addi x7, x7, -3       /* x7 = x7 - 3 */
```

3.2 The Zero Register (x0) — Rules and Implications

`x0` is the **zero register**. Architecturally:

- Reading `x0` always yields **0**.
- Writing to `x0` is **ignored** (the value is discarded).

This single rule has deep consequences. `x0` enables common patterns without special instructions:

- move / copy: `addi rd, rs, 0`
- clear register: `addi rd, x0, 0`
- unconditional jump: `jal x0, label`
- compare-to-zero branches: `beq rs, x0, label`

Example: clearing and copying

```
/* Clear x5 and copy x6 into x7 */
addi x5, x0, 0        /* x5 = 0 */
addi x7, x6, 0        /* x7 = x6 */
```

Example: unconditional control flow without a dedicated JMP

```
/* Unconditional jump */
jal  x0, target
```

```
target:
addi x5, x0, 1
```

Example: writing to x0 discards results (common bug pattern)

```
/* BUG: result is discarded because destination is x0 */
add  x0, x5, x6      /* computed value is lost */
```

Treat `x0` as a **hard-wired constant** and never as a real destination unless the intent is to discard.

3.3 ABI-Friendly Register Roles (Conceptual)

The ISA defines `x0..x31`. The **ABI** defines conventional roles so that separately compiled code can interoperate (calls, returns, and preserved state). Conceptually, ABIs classify registers into:

- argument/return registers,
- temporaries (caller-saved),
- saved registers (callee-saved),
- special-purpose roles (stack pointer, return address).

This booklet uses ABI-friendly roles conceptually because they produce clean, interoperable assembly and match compiler expectations.

Example: function argument and return value discipline (conceptual)

```
/* Conceptual ABI view:
   x10 holds the first argument and return value (a0)
*/
func_inc:
addi x10, x10, 1      /* a0 = a0 + 1 */
jalr x0, 0(x1)        /* return */
```

Even if you write standalone assembly, following ABI roles makes your code easier to integrate with C/C++ and easier to read.

3.4 Temporary vs Saved Registers

Correct low-level design depends on the discipline of **who preserves what** across function calls.

Caller-saved (temporaries)

Caller-saved registers are treated as **volatile across calls**. If the caller needs a value after calling another function, the caller must save it (in a saved register or on the stack) before the call.

Callee-saved (saved registers)

Callee-saved registers are treated as **non-volatile across calls**. If a function uses such a register, it must preserve and restore it before returning.
This conceptual split is the foundation of clean interoperability and predictable call behavior.

Example: caller must preserve a live temporary across a call

```
/* Caller: x5 holds a live value that must survive the call */
addi x5, x0, 99

addi sp, sp, -16      /* allocate stack space */
sw   x5, 12(sp)       /* save x5 */
jal  x1, helper        /* call */
lw   x5, 12(sp)       /* restore x5 */
addi sp, sp, 16       /* deallocate */
```

Example: callee preserves a saved register it uses

```
/* Callee uses x8 as a long-lived variable and must preserve it */
worker:
addi sp, sp, -16
sw   x8, 12(sp)       /* save */
addi x8, x10, 5       /* x8 = a0 + 5 (long-lived local) */
addi x10, x8, 1       /* return value in x10 */
lw   x8, 12(sp)       /* restore */
addi sp, sp, 16
jalr x0, 0(x1)
```

The value of this discipline is that **local reasoning works**: callers and callees can be written independently.

3.5 Register Usage Discipline

Clean RV32I assembly is built on strict register discipline:

- **Define roles:** decide which registers are arguments, locals, temporaries, and preserved state.
- **Minimize live ranges:** keep values alive for the shortest possible region.
- **Avoid accidental clobbering:** assume call boundaries destroy caller-saved registers unless preserved.
- **Use stack frames when needed:** save/restore systematically, not ad-hoc.
- **Never rely on uninitialized registers:** always establish known values.

Example: disciplined use of registers for a small routine

The following routine computes:

`*p = (*p + k) XOR m`

with explicit roles:

- `x10`: pointer `p`
- `x11`: value `k`
- `x12`: value `m`
- `x5`: scratch temporary

```
/* p in x10, k in x11, m in x12 */
lw   x5, 0(x10)       /* x5 = *p */
add  x5, x5, x11      /* x5 = x5 + k */
xor  x5, x5, x12      /* x5 = x5 ^ m */
sw   x5, 0(x10)       /* *p = x5 */
```

Example: common discipline checklist in code form

```
/* Always establish constants explicitly */
addi x5, x0, 0       /* clear scratch */
/* Avoid using x0 as a destination unless discarding is intended */
/* Save what must survive calls; restore before return */
```

If you follow these rules, RV32I assembly remains readable, verifiable, and interoperable—the intended meaning of clean architecture and clean semantics.

Chapter 4

RV32I Instruction Encoding

4.1 Fixed 32-bit Instruction Format

RV32I uses a **fixed 32-bit instruction length**. This single design choice drives much of RISC-V's cleanliness:

- **Predictable fetch and alignment:** instructions are naturally aligned and step in 4-byte units.
- **Fast, regular decode:** the decoder always reads the same width and extracts the same kinds of fields.
- **Stable tooling:** disassemblers and assemblers do not require complex variable-length heuristics.

Architecturally, a 32-bit instruction is partitioned into fields such as:

- **opcode** (low bits) selecting the major instruction class,
- register indices **rd, rs1, rs2**,

- **funct3** and sometimes **funct7** selecting the exact operation,
- an **immediate** value (for I/S/B/U/J forms).

Example: fixed-width stepping and PC discipline

```
/* Straight-line RV32I execution: PC advances by 4 bytes each
 ↪  instruction */
addi x5, x0, 1
addi x6, x0, 2
add  x7, x5, x6
```

4.2 R-Type, I-Type, S-Type, B-Type, U-Type, J-Type

RV32I defines a small set of canonical instruction formats. You do not decode by instruction name; you decode by **opcode + format + function fields**.

R-Type: register-register operations

R-Type typically encodes operations like `add`, `sub`, `and`, `or`, `xor`, `sll`, `srl`, `sra`, `slt`, `sltu`. It includes `rd`, `rs1`, `rs2`, `funct3`, `funct7`, `opcode`.

```
/* R-Type examples */
add  x7, x5, x6      /* x7 = x5 + x6 */
sub  x7, x5, x6      /* x7 = x5 - x6 */
slt  x7, x5, x6      /* x7 = (x5 < x6) ? 1 : 0   signed */
sltu x7, x5, x6      /* x7 = (x5 < x6) ? 1 : 0   unsigned */
```

I-Type: immediate arithmetic, loads, and jalr

I-Type is used by immediate ALU ops (`addi, andi, ori, xori, slti, sltiu`) and by loads (`lb, lh, lw, lbu, lhu`) and `jalr`. It includes `rd, rs1, imm[11:0], funct3, opcode`.

```
/* I-Type examples */
addi x5, x0, 123      /* x5 = 123 */
andi x6, x5, 15       /* x6 = x5 & 0xF */

lw   x7, 0(x10)       /* x7 = *(uint32_t*)x10 */
jalr x0, 0(x1)        /* return: PC = x1 */
```

S-Type: stores

S-Type encodes stores (`sb, sh, sw`). The immediate is split across two regions in the instruction. It includes `rs1, rs2, imm parts, funct3, opcode`.

```
/* S-Type examples */
sb x5, 0(x10)         /* *(uint8_t*)x10  = x5 */
sh x5, 2(x10)         /* *(uint16_t*)(x10+2) = x5 */
sw x5, 4(x10)         /* *(uint32_t*)(x10+4) = x5 */
```

B-Type: conditional branches

B-Type encodes conditional branches (`beq, bne, blt, bge, bltu, bgeu`). The immediate is split and represents a PC-relative branch offset.

```
/* B-Type examples */
beq  x5, x6, equal
blt  x5, x6, less     /* signed compare */
bltu x5, x6, uless    /* unsigned compare */
```

U-Type: upper immediates (LUI, AUIPC)

U-Type places a 20-bit immediate into the upper bits of a register:

- `lui`: `rd = imm20 << 12`
- `auipc`: `rd = PC + (imm20 << 12)`

```
/* U-Type examples */
lui   x5, 0x12345     /* x5 = 0x12345000 */
auipc x6, 0           /* x6 = PC + 0 */
```

J-Type: unconditional jump with link (JAL)

J-Type encodes `jal`, a PC-relative jump that also writes `PC+4` to `rd`. The immediate is split and represents the jump offset.

```
/* J-Type example */
jal  x1, func         /* x1 = return address; PC = PC + offset */
```

4.3 Immediate Encoding and Sign Extension

Immediates in RV32I are **encoded inside the instruction**, not fetched from memory. Their key properties:

- Most immediates are **sign-extended** to 32 bits.
- I-Type immediates are typically 12-bit signed values: range `[-2048, +2047]`.
- Branch and jump immediates are PC-relative and have alignment constraints (encoded with an implied low bit of zero).
- Some shifts use an immediate, but the shift amount is constrained to the valid bit-width.

Example: sign extension in addi

```
/* addi uses a 12-bit signed immediate */
addi x5, x0, -1       /* x5 = 0xFFFFFFFF */
addi x6, x0, 2047     /* maximum positive 12-bit immediate */
addi x7, x0, -2048    /* minimum negative 12-bit immediate */
```

Example: branch offset is PC-relative and aligned

```
/* Branch target is PC-relative; assembler computes the immediate
 ↪  encoding */
beq  x5, x0, label
addi x6, x0, 1
label:
addi x6, x0, 2
```

Example: forming a 32-bit constant with LUI + ADDI

A common, disciplined pattern is to place the high bits with `lui` and complete with `addi`:

```
/* Build 0x12345678 in x5 (assembler may adjust immediates as needed)
 ↪  */
lui  x5, 0x12345      /* x5 = 0x12345000 */
addi x5, x5, 0x678    /* x5 = 0x12345678 */
```

4.4 Opcode and Function Fields

Decoding in RV32I is driven by:

- **opcode**: selects the instruction class and format,

- **funct3**: selects a subgroup (often the operation family),
- **funct7**: distinguishes operations that share **opcode** and **funct3** (e.g., `add` vs `sub`).

This design provides orthogonality: many operations share a common base encoding, with function fields selecting semantics cleanly.

Example: add vs sub share fields except funct7

```
/* Both are R-Type; funct7 differentiates them */
add x7, x5, x6
sub x7, x5, x6
```

Example: shift right logical vs arithmetic

```
/* Same conceptual operation family: right shift
   Different semantics: logical fills with 0, arithmetic preserves
    ↪  sign
*/
srl x7, x5, x6       /* logical right shift */
sra x7, x5, x6       /* arithmetic right shift */
```

4.5 Decoding by Design, Not Guesswork

In clean low-level engineering, decoding is a deterministic procedure. You do not infer meaning from patterns or from the tool output alone. The correct method is:

1. Identify the **opcode** to determine the instruction class and format.
2. Extract **rd/rs1/rs2** and the relevant **funct** fields.

3. Interpret the immediate using the rules of that format (including sign extension and bit placement).

4. Apply the architectural semantics to update state.

Example: disciplined decode thinking (conceptual walkthrough)

Consider this code:

```
/* Compute: if (x5 < x6) x7 = 1; else x7 = 0 */
blt  x5, x6, less
addi x7, x0, 0
jal  x0, done
less:
addi x7, x0, 1
done:
```

A correct decoder reasoning path:

- `blt` is B-Type: opcode selects branch class; funct3 selects signed-less-than.
- Immediate is a PC-relative offset encoded across split bitfields.
- If condition holds, `PC` becomes `PC + offset`; otherwise `PC += 4`.
- `addi` is I-Type with sign-extended immediate; writing `x0` is discarded; writing `x7` sets the result explicitly.
- `jal x0, done` is an unconditional jump; return address is discarded because `rd=x0`.

Example: recognizing intentional discard via x0

```
/* Unconditional jump with no link */
jal x0, target        /* rd = x0 discards link, leaving a pure jump
 ↪  */
```

This chapter's discipline rule is simple: **every instruction is decoded by opcode and fields, and every effect is justified by the ISA rules**. That is how RV32I stays clean and why your reasoning remains correct across toolchains and implementations.

Chapter 5

Arithmetic and Logical Instructions

5.1 ADD, SUB, and Integer Arithmetic

RV32I integer arithmetic is performed on 32-bit registers with **two's-complement wraparound** behavior at the architectural level. There are no implicit flags; if you need to test a result, you do it explicitly using comparisons and branches.
Core instructions:

- `add rd, rs1, rs2  rd = rs1 + rs2`
- `sub rd, rs1, rs2  rd = rs1 - rs2`
- `addi rd, rs1, imm  rd = rs1 + signext(imm)`

Example: basic arithmetic sequences

```
/* x7 = (x5 + x6) - 10 */
add  x7, x5, x6
addi x7, x7, -10
```

Example: negate and absolute value pattern (branch-based)

```
/* abs(x5) -> x6 (signed)
   if x5 < 0 then x6 = -x5 else x6 = x5
*/
addi x6, x5, 0         /* x6 = x5 (copy) */
blt  x5, x0, neg
jal  x0, done
neg:
sub  x6, x0, x5         /* x6 = 0 - x5 */
done:
```

Example: add with carry test (manual overflow detection)

RV32I does not expose carry/overflow flags. For unsigned addition, overflow can be detected by comparing the result with one operand:

```
/* Unsigned overflow detection:
   sum = a + b; overflow if sum < a
   a in x5, b in x6
   sum in x7, overflow flag in x8 (0/1)
*/
add  x7, x5, x6
sltu x8, x7, x5         /* x8 = (x7 < x5) ? 1 : 0 */
```

5.2 Shift Operations and Bit Manipulation

Shifts are fundamental for scaling, bit-field work, masking, and constant multiplication. RV32I provides:

- `sll` / `slli` : logical left shift
- `srl` / `srli` : logical right shift (zero-fill)
- `sra` / `srai` : arithmetic right shift (sign-preserving)

Shift amounts in RV32 are effectively within **0..31**. In register shifts, the hardware uses the low bits of the shift-amount register.

Example: scaling by powers of two

```
/* x6 = x5 * 8 */
slli x6, x5, 3
```

Example: extract a byte from a word (shift + mask)

```
/* Extract byte #2 (bits 23:16) from x5 into x6 */
srli x6, x5, 16        /* bring target byte to bits 7:0 */
andi x6, x6, 0xFF      /* mask low byte */
```

Example: arithmetic vs logical right shift

```
/* x5 = 0x80000000 (negative if interpreted signed) */
srli x6, x5, 1         /* x6 = 0x40000000 (zero-fill) */
srai x7, x5, 1         /* x7 = 0xC0000000 (sign-extend fill) */
```

Example: constant multiplication without M extension

If multiplication is unavailable (pure RV32I), use shifts and adds:

```
/* x6 = x5 * 10 = (x5<<3) + (x5<<1) */
slli x7, x5, 3         /* x7 = x5 * 8 */
```

```
slli x8, x5, 1          /* x8 = x5 * 2 */
add  x6, x7, x8         /* x6 = x5 * 10 */
```

5.3 Logical Operations (AND, OR, XOR)

RV32I provides bitwise logical operations in both register and immediate forms:

- `and` / `andi`
- `or` / `ori`
- `xor` / `xori`

These are essential for masking, setting/clearing bits, toggling flags, and building compact tests.

Example: set, clear, toggle a bit

Assume bit `k` (0..31) and mask `(1<<k)` in `x6`, value in `x5`:

```
/* Set bit: x5 |= mask */
or   x5, x5, x6

/* Clear bit: x5 &= ~mask (build ~mask first) */
xori x7, x6, -1         /* x7 = ~mask (bitwise NOT via XOR with
 ↪  all-ones) */
and  x5, x5, x7

/* Toggle bit: x5 ^= mask */
xor  x5, x5, x6
```

Example: test if a bit is set

```
/* if (x5 & mask) != 0 then branch */
and  x7, x5, x6
bne  x7, x0, bit_set
```

Example: zero-test without flags

```
/* if x5 == 0 goto is_zero */
beq  x5, x0, is_zero
```

5.4 Signed vs Unsigned Semantics

Registers are just 32-bit patterns. **Signedness is determined by the instruction**, especially for comparisons and right shifts:

- **Signed comparisons:** `slt`, `slti`, `blt`, `bge`
- **Unsigned comparisons:** `sltu`, `sltiu`, `bltu`, `bgeu`
- **Right shift:** `sra/srai` is sign-preserving; `srl/srli` is zero-fill

Example: signed vs unsigned compare difference

```
/* x5 = 0xFFFFFFFF, x6 = 1 */
addi x5, x0, -1        /* -1 signed, 4294967295 unsigned */
addi x6, x0, 1

slt  x7, x5, x6        /* signed: (-1 < 1) => x7 = 1 */
sltu x8, x5, x6        /* unsigned: (0xFFFFFFFF < 1) => x8 = 0 */
```

Example: branch signed vs unsigned

```
/* Same values, different branch outcomes */
blt  x5, x6, signed_less    /* taken */
bltu x5, x6, unsigned_less  /* not taken */
```

5.5 Common Arithmetic Pitfalls

Pitfall 1: expecting flags (carry/overflow/zero)

RV32I does not provide condition flags. All tests must be explicit.

```
/* Wrong mental model: "add sets flags" */
/* Correct: compute then compare */
add  x7, x5, x6
beq  x7, x0, sum_is_zero
```

Pitfall 2: confusing signed vs unsigned comparisons

Always select the correct compare/branch instruction family.

```
/* Use bltu for pointer/size comparisons, blt for signed integers */
bltu x10, x11, ptr_less     /* unsigned compare for addresses */
blt  x5,  x6,  signed_less  /* signed compare for int32_t */
```

Pitfall 3: using the wrong right shift

```
/* For signed division by 2, use arithmetic shift */
srai x6, x5, 1              /* preserves sign */
```

```
/* For bit-field extraction, use logical shift */
srli x7, x5, 1              /* zero-fill */
```

Pitfall 4: immediate range misunderstandings

Many immediate forms (notably I-Type) are 12-bit signed. Large constants require construction via `lui` + `addi` (or PC-relative patterns).

```
/* Build a larger constant: 0x12345678 */
lui  x5, 0x12345
addi x5, x5, 0x678
```

Pitfall 5: accidental discard to x0

```
/* BUG: result discarded */
add x0, x5, x6
```

Pitfall 6: overflow assumptions

Arithmetic wraps modulo 2^{32}. If you need overflow detection, implement it explicitly.

```
/* Unsigned add overflow check: overflow if sum < a */
add  x7, x5, x6
sltu x8, x7, x5
```

This chapter's rule is simple: RV32I is clean because it is explicit. Arithmetic, bit manipulation, and correctness checks are all performed through visible instructions, with no hidden state.

Chapter 6

Load and Store Architecture

6.1 Memory as an Explicit Resource

In RV32I, memory is never accessed implicitly. Every read or write of memory is performed by a dedicated load or store instruction. Arithmetic, logic, and control-flow instructions operate exclusively on registers.
This explicit separation has critical architectural consequences:

- Every memory access is visible in the instruction stream.
- The cost and ordering of memory operations can be reasoned about precisely.
- No instruction combines arithmetic and memory access in a single operation.

This design removes ambiguity and hidden side effects, enabling correct reasoning across different implementations.

Example: explicit memory traffic

```
/* Memory is accessed only by loads and stores */
```

```
lw   x5, 0(x10)       /* load from memory */
add  x5, x5, x11      /* compute in registers */
sw   x5, 0(x10)       /* store back to memory */
```

6.2 Load Instructions (LB, LH, LW, LBU, LHU)

RV32I provides explicit load instructions for different data widths and signedness:

- `lb` : load 8-bit, sign-extended
- `lh` : load 16-bit, sign-extended
- `lw` : load 32-bit
- `lbu` : load 8-bit, zero-extended
- `lhu` : load 16-bit, zero-extended

All loads compute their effective address as:

```
address = rs1 + signext(imm12)
```

Example: signed vs unsigned byte load

```
/* Assume memory[x10] = 0xFF */
lb   x5, 0(x10)       /* x5 = 0xFFFFFFFF (-1 signed) */
lbu  x6, 0(x10)       /* x6 = 0x000000FF (255 unsigned) */
```

Example: halfword load semantics

```
/* Assume memory[x10..x11] = 0x8001 (little-endian) */
lh   x5, 0(x10)       /* x5 = 0xFFFF8001 */
lhu  x6, 0(x10)       /* x6 = 0x00008001 */
```

Example: loading structured data fields

```
/* x10 points to a record:
   offset 0: uint32_t id
   offset 4: uint16_t flags
   offset 8: uint32_t value
*/
lw   x5, 0(x10)       /* id */
lhu  x6, 4(x10)       /* flags */
lw   x7, 8(x10)       /* value */
```

6.3 Store Instructions (SB, SH, SW)

Store instructions write register contents to memory:

- `sb` : store low 8 bits
- `sh` : store low 16 bits
- `sw` : store full 32 bits

The effective address calculation follows the same base-plus-offset rule as loads.

Example: storing values of different widths

```
/* x5 holds a value to store, x10 is base address */
sb x5, 0(x10)         /* store byte */
sh x5, 2(x10)         /* store halfword */
sw x5, 4(x10)         /* store word */
```

Example: updating a structure field

```
/* Update value field at offset 8 */
lw   x6, 8(x10)
addi x6, x6, 1
sw   x6, 8(x10)
```

6.4 Address Calculation Rules

In RV32I, all memory addresses are calculated explicitly using a base register and a signed immediate:

- The base register typically holds a pointer.
- The immediate is a 12-bit signed value (`-2048` to `+2047`).
- The effective address is computed before the memory access.

There are no scaled-index modes, no implicit pointer increments, and no auto-update addressing. Complex addressing must be built explicitly.

Example: manual pointer arithmetic

```
/* Iterate over an array of 32-bit integers */
loop:
lw   x5, 0(x10)      /* load *ptr */
addi x10, x10, 4     /* ptr++ */
addi x11, x11, -1    /* count-- */
bne  x11, x0, loop
```

Example: accessing beyond immediate range

When offsets exceed the 12-bit immediate range, compute the address explicitly:

```
/* Access *(base + 5000) */
addi x12, x10, 5000  /* compute address explicitly */
lw   x5, 0(x12)
```

6.5 Alignment and Misalignment Behavior

RV32I defines natural alignment expectations:

- byte access: any address
- halfword access: address multiple of 2
- word access: address multiple of 4

Correct low-level code assumes proper alignment unless the execution environment explicitly guarantees safe unaligned access. Misaligned accesses may be slower, emulated, or cause traps depending on the system.

Example: aligned word access

```
/* x10 is word-aligned */
lw x5, 0(x10)        /* safe and efficient */
```

Example: misaligned access workaround

To safely handle potentially misaligned data, load bytes and assemble manually:

```
/* Assemble a 32-bit little-endian value from bytes */
lbu  x5, 0(x10)      /* byte 0 */
lbu  x6, 1(x10)      /* byte 1 */
lbu  x7, 2(x10)      /* byte 2 */
lbu  x8, 3(x10)      /* byte 3 */

slli x6, x6, 8
slli x7, x7, 16
slli x8, x8, 24

or   x5, x5, x6
or   x5, x5, x7
or   x5, x5, x8      /* x5 = assembled 32-bit value */
```

Example: alignment discipline in data structures

```
/* Recommended structure layout (conceptual):
   offset 0: uint32_t
   offset 4: uint32_t
   offset 8: uint16_t
   offset 10: uint16_t
*/
```

This chapter's discipline is strict: memory is an explicit resource. You see every load, every store, and every address calculation. That clarity is the foundation of predictable, portable RV32I code.

Chapter 7

Control Flow Instructions

7.1 Conditional Branch Instructions

RV32I provides a small, complete set of **conditional branch** instructions. Branches do not consult flags (there are none). Each branch compares two registers and, if the condition is true, updates the `PC` to a **PC-relative target**. Otherwise, execution falls through to the next instruction.
Branch instructions:

- `beq` : branch if equal
- `bne` : branch if not equal
- `blt` : branch if less than (signed)
- `bge` : branch if greater/equal (signed)
- `bltu` : branch if less than (unsigned)
- `bgeu` : branch if greater/equal (unsigned)

Example: equality and inequality

```
/* if (x5 == x6) goto equal */
beq  x5, x6, equal
addi x7, x0, 0
jal  x0, done
equal:
addi x7, x0, 1
done:
```

Example: if (x5 != 0)

```
/* if (x5 != 0) goto nonzero */
bne  x5, x0, nonzero
addi x6, x0, 0
jal  x0, done2
nonzero:
addi x6, x0, 1
done2:
```

7.2 Branch Comparison Semantics

The key rule: **registers have no inherent signedness**. Signedness is defined by the instruction used.

- **Signed compares:** `blt`, `bge`
- **Unsigned compares:** `bltu`, `bgeu`

This matters strongly for pointers, sizes, and bit patterns that may represent negative values in two's complement.

Example: signed vs unsigned branch difference

```
/* x5 = 0xFFFFFFFF, x6 = 1 */
addi x5, x0, -1        /* -1 signed, 0xFFFFFFFF unsigned */
addi x6, x0, 1

blt  x5, x6, signed_less     /* taken: -1 < 1 */
bltu x5, x6, unsigned_less   /* not taken: 0xFFFFFFFF < 1 is false */

signed_less:
addi x7, x0, 1
jal  x0, done3

unsigned_less:
addi x7, x0, 2

done3:
```

Example: pointer comparisons must be unsigned

```
/* Use unsigned branches for addresses and sizes */
bltu x10, x11, ptr_is_lower
```

7.3 Jump and Link (JAL, JALR)

RV32I expresses calls and returns with explicit jump instructions. There is no implicit call stack manipulation.

JAL

`jal rd, target` performs:

- `rd = PC + 4`
- `PC = PC + offset` (PC-relative target)

JALR

`jalr rd, imm(rs1)` performs:

- `rd = PC + 4`
- `PC = (rs1 + imm) & ~1`

Example: function call and return

```
/* Caller */
addi x10, x0, 7        /* argument in x10 */
jal  x1, func          /* x1 = return address; jump */

/* After return */
addi x11, x10, 1       /* use returned value */

/* Callee */
func:
addi x10, x10, 5       /* return value in x10 */
jalr x0, 0(x1)         /* return: rd=x0 discards link */
```

Example: unconditional jump without link

```
/* Pure jump (no return address saved) */
jal x0, target

target:
addi x5, x0, 1
```

Example: indirect jump via function pointer (conceptual)

```
/* x5 holds a function address */
jalr x1, 0(x5)         /* call *x5, save return in x1 */
```

7.4 Structured Control Flow in Assembly

Structured control flow is not a language feature; it is a discipline. RV32I makes structured patterns explicit and readable when you follow consistent templates.

Pattern 1: if / else

```
/* if (x5 < x6) x7=1; else x7=0; (signed) */
blt  x5, x6, then1
addi x7, x0, 0
jal  x0, end1
then1:
addi x7, x0, 1
end1:
```

Pattern 2: while loop

```
/* while (x5 != 0) { x5--; } */
loop2:
beq  x5, x0, end2
addi x5, x5, -1
jal  x0, loop2
end2:
```

Pattern 3: counted for-loop over word array

```
/* for (i=0; i<n; i++) sum += a[i];
   base a in x10, n in x11, sum in x12
*/
addi x12, x0, 0        /* sum = 0 */
loop3:
beq  x11, x0, end3
lw   x5, 0(x10)        /* load a[i] */
add  x12, x12, x5      /* sum += a[i] */
addi x10, x10, 4       /* a++ */
addi x11, x11, -1      /* n-- */
jal  x0, loop3
end3:
```

Pattern 4: switch-like jump table (conceptual, bounds checked)

This is a disciplined pattern: validate range, then compute address and jump.

```
/* switch(x5) with cases 0..3:
   x5 = selector
```

```
   x10 = base address of jump table (array of 32-bit targets)
*/
sltiu x6, x5, 4       /* x6 = (x5 < 4) ? 1 : 0 */
beq   x6, x0, defcase

slli  x7, x5, 2       /* index * 4 */
add   x7, x10, x7     /* &table[index] */
lw    x8, 0(x7)       /* target = table[index] */
jalr  x0, 0(x8)       /* jump to target */

defcase:
jal   x0, default_handler
```

7.5 Common Branching Errors

Error 1: using signed branches for unsigned values

Pointers, sizes, and bit patterns should use `bltu/bgeu`. Using `blt/bge` can break correctness when high-bit addresses occur.

```
/* Correct for addresses */
bltu x10, x11, addr_lower
```

Error 2: inverted conditions and fall-through mistakes

Many bugs come from mixing up the branch-to-then vs branch-to-else style. Use a consistent template:

```
/* Preferred: branch-to-then, explicit jump-to-end */
blt  x5, x6, then_ok
```

```
/* else path */
...
jal  x0, end_ok
then_ok:
...
end_ok:
```

Error 3: forgetting that there are no flags

Do not assume `add` or `sub` sets carry/zero/negative flags. You must compare explicitly.

```
/* Correct zero test after computation */
add  x7, x5, x6
beq  x7, x0, sum_is_zero
```

Error 4: accidental link discard or unintended link creation

```
/* Pure jump: ok */
jal  x0, target

/* Call: must save return address to a real register */
jal  x1, func
```

Error 5: returning to the wrong place (clobbered x1)

If you use the link register (`x1`) inside a non-leaf function, preserve it.

```
/* Non-leaf function must preserve x1 if it makes calls */
nonleaf:
addi sp, sp, -16
sw   x1, 12(sp)       /* save return address */
```

```
jal  x1, helper         /* nested call clobbers x1 */

lw   x1, 12(sp)        /* restore return address */
addi sp, sp, 16
jalr x0, 0(x1)         /* return */
```

The execution model is simple: branches compare registers, jumps update `PC`, and calls/returns are explicit. Clean assembly is achieved by disciplined templates, correct signedness choice, and explicit preservation of live state across calls.

Chapter 8

Function Calls and Stack Discipline

8.1 Stack Growth and Alignment

On RV32 systems, the stack is a contiguous memory region used for:

- preserving registers across calls,
- holding local variables that do not fit in registers,
- passing additional arguments when registers are insufficient,
- supporting debug/profiling conventions (when enabled).

The stack pointer is held in `x2`. The disciplined model is:

- the stack **grows downward** (toward lower addresses),
- allocation is done by **subtracting** from `sp`,
- deallocation is done by **adding** back to `sp`,
- `sp` is kept aligned to the platform ABI requirement (commonly 16-byte alignment).

Example: allocate and deallocate a 16-byte stack frame

```
/* Prologue: allocate 16 bytes */
addi sp, sp, -16

/* ... body ... */

/* Epilogue: free 16 bytes */
addi sp, sp, 16
```

Alignment discipline is not optional if you want correct interoperability with compiler-generated code and predictable behavior for spills, locals, and calls.

8.2 Call and Return Mechanism

RV32I expresses calls and returns explicitly with `jal` and `jalr`. There is no implicit call stack manipulation in the ISA.

Call via JAL

`jal rd, target` performs:

- `rd = PC + 4` (save return address),
- `PC = PC + offset` (jump to target).

Return via JALR

A return is a jump to the saved return address:

- `jalr x0, 0(x1)` sets `PC = x1` and discards the link.

Example: minimal call and return

```
/* Caller */
jal  x1, func          /* x1 = return address; jump */

/* Callee */
func:
addi x10, x10, 1       /* example: a0 = a0 + 1 */
jalr x0, 0(x1)         /* return */
```

Example: indirect call via function pointer

```
/* x5 holds a function address */
jalr x1, 0(x5)         /* call *x5; save return in x1 */
```

8.3 Saving and Restoring Registers

Correct calling discipline depends on **preservation rules**. Conceptually:

- **Caller-saved** registers may be clobbered by the callee.
- **Callee-saved** registers must be preserved by the callee if used.

Even if you do not name the full ABI register classes in text, the discipline is straightforward:

- If a function uses a register that the caller expects preserved, save/restore it.
- If the caller needs a volatile register after a call, the caller saves it.
- If a function makes another call, it must protect its own return address (`x1`) if `x1` will be clobbered.

Example: callee saves a long-lived local register

```
/* Uses x8 as a long-lived local; preserve it */
worker:
addi sp, sp, -16
sw   x8, 12(sp)        /* save x8 */

addi x8, x10, 5        /* x8 = a0 + 5 */
addi x10, x8, 1        /* return = x8 + 1 */

lw   x8, 12(sp)        /* restore x8 */
addi sp, sp, 16
jalr x0, 0(x1)
```

Example: caller preserves a live temporary across a call

```
/* x5 holds a value needed after calling helper */
addi x5, x0, 99

addi sp, sp, -16
sw   x5, 12(sp)        /* save live temporary */
jal  x1, helper
lw   x5, 12(sp)        /* restore */
addi sp, sp, 16
```

8.4 Leaf vs Non-Leaf Functions

Leaf function

A leaf function makes **no further calls**. It can often:

- avoid a stack frame entirely,
- keep all locals in registers,
- return directly with `jalr x0, 0(x1)`.

Example: leaf function without stack frame

```
/* leaf: returns (a0 + a1) in a0 */
sum2:
add  x10, x10, x11     /* a0 = a0 + a1 */
jalr x0, 0(x1)         /* return */
```

Non-leaf function

A non-leaf function makes at least one call. Therefore:

- it must assume `x1` will be overwritten by nested `jal/jalr`,
- it must save `x1` if it needs to return correctly,
- it typically uses a stack frame.

Example: non-leaf must preserve the return address

```
/* non-leaf: calls helper, must save x1 */
nonleaf:
addi sp, sp, -16
sw   x1, 12(sp)        /* save return address */

jal  x1, helper        /* nested call clobbers x1 */
addi x10, x10, 1       /* do some work */
```

```
lw   x1, 12(sp)        /* restore return address */
addi sp, sp, 16
jalr x0, 0(x1)         /* return */
```

8.5 Clean Call Sequences

Clean call sequences are systematic and symmetric. A disciplined function has:

Prologue

- allocate a fixed-size stack frame (aligned),
- save `x1` if non-leaf,
- save any preserved registers used by the function.

Epilogue

- restore preserved registers,
- restore `x1` if saved,
- deallocate the stack frame,
- return with `jalr x0, 0(x1)`.

Example: clean, symmetric prologue/epilogue template

```
/* Template: clean function frame (non-leaf example) */
func:
```

```
addi sp, sp, -32
sw   x1, 28(sp)        /* save return address */
sw   x8, 24(sp)        /* save a preserved local */
sw   x9, 20(sp)        /* save another preserved local */

/* ... function body ...
   may call other functions safely
*/

lw   x9, 20(sp)
lw   x8, 24(sp)
lw   x1, 28(sp)
addi sp, sp, 32
jalr x0, 0(x1)
```

Example: clean caller sequence around a call

```
/* Caller saves what it needs, sets args, calls, then restores */
addi sp, sp, -16
sw   x5, 12(sp)        /* preserve live value */

addi x10, x0, 7        /* arg0 */
addi x11, x0, 3        /* arg1 */
jal  x1, callee

lw   x5, 12(sp)        /* restore live value */
addi sp, sp, 16
```

The discipline rule is strict: treat calls as boundaries where volatile registers may be destroyed, preserve what must survive, keep stack alignment correct, and ensure prologue/epilogue

symmetry. This is the foundation of correct RV32I interoperability and maintainable assembly.

Chapter 9

Writing Clean RV32I Assembly

9.1 Minimalism as a Design Rule

RV32I rewards minimalism because the ISA is explicit and orthogonal. Clean assembly is not the shortest possible code; it is the smallest code that is **obviously correct**.
Minimalism rules:

- Prefer a small, stable set of patterns: load–compute–store, compare–branch, prologue–epilogue.
- Keep live register sets small: reduce the number of values that must survive across calls.
- Avoid clever instruction tricks that hide intent.
- Use `x0` deliberately: copy, clear, or discard—never by accident.

Example: minimal but explicit update of a memory word

```
/* *p += k;  p in x10, k in x11 */
```

```
lw   x5, 0(x10)        /* tmp = *p */
add  x5, x5, x11       /* tmp += k */
sw   x5, 0(x10)        /* *p = tmp */
```

Example: avoid accidental discard to x0

```
/* BUG: computed value is discarded */
add  x0, x5, x6
```

9.2 Readability and Maintainability

Assembly becomes maintainable when the reader can answer, at every line:

- What does this instruction do?
- Why is it here?
- What state does it change?

Maintainability rules:

- Use consistent register roles across the file.
- Document invariants: what each register means at block boundaries.
- Keep control flow structured: one entry, clear exits, explicit join points.
- Make memory access visible and clustered: do not scatter loads/stores unnecessarily.

Example: readable if/else pattern (single join point)

```
/* if (x5 < x6) x7=1; else x7=0; (signed) */
blt  x5, x6, then1
addi x7, x0, 0          /* else */
jal  x0, end1
then1:
addi x7, x0, 1          /* then */
end1:
```

9.3 Naming Conventions and Layout

The assembler accepts numeric registers (`x0..x31`). Clean code uses stable conventions:

- Use ABI-friendly roles conceptually: `x2` as `sp`, `x1` as link (`ra`), `x10..` as arguments/return.
- Use a small set of scratch registers consistently (e.g., `x5--x7`) for temporaries.
- Keep labels structured and local: `loop`, `done`, `then`, `else`, `ret`.
- Place a short block comment at the top of each routine describing inputs/outputs and clobbers.

Layout rules:

- One logical step per line.
- Indent instructions uniformly.
- Group prologue/epilogue symmetrically.
- Keep blank lines between blocks (prologue, body, epilogue).

Example: routine header and clean layout

```
/* sum_array:
   in : x10 = base pointer, x11 = count
   out: x10 = sum
   clobbers: x5, x6
*/
sum_array:
addi x10, x10, 0        /* keep base in x10 */
addi x6,  x0,  0        /* sum = 0 */

loop:
beq  x11, x0, done
lw   x5, 0(x10)          /* load element */
add  x6, x6, x5          /* sum += elem */
addi x10, x10, 4         /* base++ */
addi x11, x11, -1        /* count-- */
jal  x0, loop

done:
addi x10, x6, 0          /* move sum to return register */
jalr x0, 0(x1)
```

9.4 Avoiding Accidental Complexity

Most assembly complexity is self-inflicted. RV32I keeps the ISA clean; your job is to keep the program clean.
Avoid these sources of accidental complexity:

- Hidden state assumptions (there are no flags).

- Mixed signed/unsigned comparisons without explicit intent.
- Unstructured control flow (multiple exits without clear join points).
- Untracked register lifetimes (values silently clobbered across calls).
- Overusing stack frames when registers suffice (or underusing them when preservation is needed).

Example: prevent signed/unsigned mistakes by naming intent

```
/* Pointer compare: must be unsigned */
bltu x10, x11, ptr_lower

/* Signed integer compare */
blt  x5,  x6,  signed_lower
```

Example: avoid implicit ”fall-through bugs” with explicit jumps

```
/* Structured: every path reaches the join label */
beq  x5, x0, is_zero
addi x6, x0, 1
jal  x0, join
is_zero:
addi x6, x0, 0
join:
```

Example: avoid unnecessary memory traffic

```
/* Prefer register temporaries over redundant reloads */
lw   x5, 0(x10)
```

```
addi x5, x5, 1
sw   x5, 0(x10)
```

9.5 Debug-Friendly Code Structure

Debug-friendly assembly is code that you can single-step and verify without guessing. The key is to make state changes easy to observe.
Rules for debug-friendly structure:

- Establish invariants at block boundaries (what each register holds).
- Use short basic blocks with clear labels.
- Keep stack frames simple and symmetric.
- Preserve return address in non-leaf functions (`x1`) and restore it exactly once.
- Avoid self-modifying code and avoid relying on unspecified behavior.

Example: non-leaf function with clear frame and one exit

```
/* inc_then_call:
   in : x10 = value
   out: x10 = result
*/
inc_then_call:
addi sp, sp, -16
sw   x1, 12(sp)         /* save return address */

addi x10, x10, 1        /* local computation */
jal  x1, helper          /* call */
```

```
lw   x1, 12(sp)         /* restore */
addi sp, sp, 16
jalr x0, 0(x1)
```

Example: diagnostic-friendly loop with explicit counter and join

```
/* Count down x11 to zero while accumulating into x10 */
loop2:
beq  x11, x0, done2
add  x10, x10, x11
addi x11, x11, -1
jal  x0, loop2
done2:
jalr x0, 0(x1)
```

Clean RV32I assembly is a discipline: minimal but explicit state changes, consistent register roles, structured control flow, and symmetric stack handling. When you follow these rules, your code remains readable, maintainable, and reliably debuggable.

Chapter 10

RV32I and C Interoperability

10.1 How Compilers Map C to RV32I

A C compiler targets RV32I by lowering C operations into a small set of explicit patterns:

- **Expressions:** computed in registers using `add/sub/and/or/xor/shift` and `addi/andi/ori/xori`.
- **Memory objects:** accessed only via `lw/lh/lhu/lb/lbu` and `sw/sh/sb`.
- **Control**
  flow: implemented with compare-and-branch (`beq/bne/blt/bge/bltu/bgeu`) and explicit jumps (`jal/jalr`).
- **Calls:** `jal` to a symbol or `jalr` through a register; return via `jalr x0, 0(x1)`.

Example: C expression lowering (conceptual)

C:

$$y = (a + 3) \oplus (b \ll 2)$$

Typical RV32I shape:

```
/* a in x10, b in x11, y in x12 */
addi x5, x10, 3       /* t0 = a + 3 */
slli x6, x11, 2       /* t1 = b << 2 */
xor  x12, x5, x6      /* y  = t0 ^ t1 */
```

Example: C load/modify/store pattern

C:

$$*p = *p + k$$

Typical RV32I shape:

```
/* p in x10, k in x11 */
lw   x5, 0(x10)
add  x5, x5, x11
sw   x5, 0(x10)
```

10.2 Register Allocation Observations

Register allocation is the compiler step that assigns many logical temporaries to the limited physical registers `x0..x31`. For interoperability, what matters is not the allocator algorithm, but the **observable patterns**:

- **Short-lived temporaries** often use volatile scratch registers.
- **Long-lived values across calls** are placed in preserved registers or spilled to the stack.
- **High optimization levels** reduce memory traffic by keeping values in registers.
- **Low optimization levels** often create heavier stack frames and more loads/stores.

Example: keeping values in registers vs spilling

Conceptually, a compiler prefers this:

```
/* Keep working set in registers */
add  x5, x10, x11
xor  x6, x5,  x12
add  x10, x6, x13
```

But if registers are pressured or values must survive calls, you see spills:

```
/* Spill a value to stack to survive a call */
addi sp, sp, -16
sw   x5, 12(sp)
jal  x1, helper
lw   x5, 12(sp)
addi sp, sp, 16
```

Practical rule

When reading compiler output: if a register value is used after a call, either it is preserved by convention or it is saved/restored explicitly.

10.3 Stack Frames in Generated Code

A stack frame exists when the compiler needs:

- space for spills (register pressure),
- local objects that must reside in memory (e.g., addressed locals),
- saved registers (including return address when non-leaf),

- outgoing arguments beyond the register-passed set.

Even without naming the full ABI, the structural invariants are consistent:

- `sp` is adjusted by a fixed negative amount in the prologue,
- selected registers are saved into that frame,
- `sp` is restored in the epilogue before return.

Example: typical compiler-shaped non-leaf frame

```
/* Prologue */
addi sp, sp, -32
sw   x1, 28(sp)        /* save return address */
sw   x8, 24(sp)        /* save preserved reg used */
sw   x9, 20(sp)        /* save another preserved reg */

/* ... body ... */
jal  x1, helper

/* Epilogue */
lw   x9, 20(sp)
lw   x8, 24(sp)
lw   x1, 28(sp)
addi sp, sp, 32
jalr x0, 0(x1)
```

Example: leaf function may omit the frame

```
/* Leaf: no stack frame needed if no spills and no calls */
```

```
add  x10, x10, x11
jalr x0, 0(x1)
```

10.4 Calling Convention Overview (Conceptual)

Interoperability requires a calling convention: a contract describing how calls pass data and preserve state. Conceptually, it specifies:

- **Where arguments are placed** (typically in designated registers first, then stack).
- **Where return values appear** (typically in designated return registers).
- **Which registers a callee must preserve** (callee-saved) vs may clobber (caller-saved).
- **Stack alignment rules** and stack frame layout conventions.
- **Return address handling** via the link register.

In clean RV32I assembly intended to interoperate with C, you must:

- accept inputs in the standard argument registers (conceptually),
- return results in the standard return register (conceptually),
- preserve any callee-saved registers you use,
- preserve the return address if you make further calls.

Example: C-callable function shape (conceptual)

```
/* int add3(int x)  -> return x+3
   input in x10, output in x10
*/
```

```
add3:
addi x10, x10, 3
jalr x0, 0(x1)
```

Example: non-leaf C-callable function (conceptual)

```
/* int f(int x) { return helper(x+1); } */
f:
addi sp, sp, -16
sw   x1, 12(sp)        /* save return address */

addi x10, x10, 1       /* x = x + 1 */
jal  x1, helper        /* call helper(x) */

lw   x1, 12(sp)
addi sp, sp, 16
jalr x0, 0(x1)
```

10.5 Understanding Compiler Output

To read compiler-generated RV32I assembly correctly, follow a deterministic procedure:

1) Identify the function boundaries

Look for the label, then prologue (stack adjustment + saves), then epilogue (restores + return).

2) Track register roles

Determine which registers hold:

- arguments/return values,
- long-lived locals,
- temporaries,
- addresses/pointers,
- saved state (return address, preserved registers).

3) Reduce code to core patterns

Most compiler output decomposes into:

- load–compute–store
- compare–branch
- prologue–call–epilogue

Example: compiler-like loop for array sum

This matches common C lowering:

```
/* sum = 0; while (n) { sum += *p; p++; n--; } */
addi x12, x0, 0        /* sum */

loop:
beq  x11, x0, done
lw   x5, 0(x10)        /* load *p */
add  x12, x12, x5      /* sum += */
addi x10, x10, 4       /* p++ */
addi x11, x11, -1      /* n-- */
```

```
jal  x0, loop

done:
addi x10, x12, 0      /* move sum to return */
jalr x0, 0(x1)
```

4) Verify preservation at call boundaries

If you see a call (`jal`/`jalr`), check:

- whether the return address is saved if the function is non-leaf,
- whether values needed after the call are preserved (saved regs or stack spills).

Example: preservation reasoning around a call

```
/* x5 holds a live value that must survive helper */
addi sp, sp, -16
sw   x5, 12(sp)
jal  x1, helper
lw   x5, 12(sp)
addi sp, sp, 16
```

The core rule for interoperability is strict: treat the calling convention as a contract. When your RV32I assembly respects argument placement, return value placement, stack alignment, and preservation rules, it can interoperate cleanly with C across compilers and toolchains.

Chapter 11

Common Mistakes and Debugging Patterns

11.1 Misusing the Zero Register

`x0` always reads as zero and discards any write. This is powerful when used intentionally and disastrous when used accidentally.

Mistake: writing results to `x0`

```
/* BUG: result is discarded */
add  x0, x5, x6
```

Correct: choose a real destination

```
/* Correct */
add  x7, x5, x6
```

Mistake: accidental link discard during call

```
/* BUG: jal writes return address to x0, so return is impossible */
jal  x0, func
```

Correct: save return address into `x1` (link register)

```
/* Correct call */
jal  x1, func
```

Intentional use patterns (safe)

```
/* Pure jump (no link) */
jal  x0, target

/* Clear register */
addi x5, x0, 0

/* Copy register */
addi x6, x5, 0
```

11.2 Broken Stack Frames

Stack bugs are among the most common assembly failures because they corrupt return paths and saved state.

Mistake: mismatched allocation/deallocation

```
/* BUG: allocate 16 but free 12 */
addi sp, sp, -16
```

```
/* ... */
addi sp, sp, 12
jalr x0, 0(x1)
```

Correct: symmetric prologue/epilogue

```
addi sp, sp, -16
/* ... */
addi sp, sp, 16
jalr x0, 0(x1)
```

Mistake: non-leaf function clobbers `x1`

Any nested call overwrites the link register. If the function is non-leaf, it must preserve `x1`.

```
/* BUG: x1 overwritten by helper call, return address lost */
nonleaf:
jal  x1, helper
jalr x0, 0(x1)
```

Correct: save/restore `x1`

```
nonleaf:
addi sp, sp, -16
sw   x1, 12(sp)

jal  x1, helper

lw   x1, 12(sp)
addi sp, sp, 16
jalr x0, 0(x1)
```

Mistake: storing outside allocated frame

```
/* BUG: store at 20(sp) without allocating enough space */
addi sp, sp, -16
sw   x5, 20(sp)
```

Correct: allocate enough space for your offsets

```
addi sp, sp, -32
sw   x5, 20(sp)
```

11.3 Incorrect Branch Logic

Branching bugs are usually not ISA complexity; they are logic errors caused by inconsistent templates or wrong signedness.

Mistake: using signed branch for unsigned data (pointers/sizes)

```
/* BUG: signed compare for addresses can be wrong */
blt  x10, x11, addr_lower
```

Correct: use unsigned branch for addresses/sizes

```
bltu x10, x11, addr_lower
```

Mistake: inverted condition with fall-through confusion

```
/* BUG-PRONE: unclear which path is then/else */
beq  x5, x0, A
/* intended A? or else? */
```

```
addi x6, x0, 1
A:
addi x6, x0, 0
```

Correct: structured template with one join

```
/* if (x5 == 0) x6=0; else x6=1; */
beq  x5, x0, then1
addi x6, x0, 1        /* else */
jal  x0, end1
then1:
addi x6, x0, 0        /* then */
end1:
```

Mistake: assuming flags exist

```
/* BUG: RV32I does not set flags; this is meaningless thinking */
sub  x7, x5, x6
/* "if zero" must be explicit */
```

Correct: explicit comparison of computed result

```
sub  x7, x5, x6
beq  x7, x0, is_zero
```

11.4 Load/Store Confusion

Most memory bugs come from misunderstanding widths, extension, alignment, or address calculation.

Mistake: using `lb` when you need `lbu`

```
/* BUG: byte 0xFF becomes -1 (sign-extended) */
lb   x5, 0(x10)
```

Correct: use zero-extending load for unsigned bytes

```
lbu  x5, 0(x10)        /* 0xFF becomes 255 */
```

Mistake: forgetting word scaling in array indexing

```
/* BUG: uses index directly as byte offset for int32 array */
add  x12, x10, x11     /* should be index*4 */
lw   x5, 0(x12)
```

Correct: scale index by element size

```
/* addr = base + index*4 */
slli x12, x11, 2
add  x12, x10, x12
lw   x5, 0(x12)
```

Mistake: storing word with misalignment risk

```
/* BUG-PRONE: sw requires word alignment for portable code */
sw   x5, 1(x10)
```

Correct: maintain alignment or use byte assembly when needed

```
/* Assemble/store bytes when alignment is not guaranteed */
sb   x5, 0(x10)
```

```
srli x6, x5, 8
sb   x6, 1(x10)
srli x6, x5, 16
sb   x6, 2(x10)
srli x6, x5, 24
sb   x6, 3(x10)
```

11.5 Debugging by Reasoning, Not Trial

Clean debugging in RV32I is deterministic because the ISA is explicit. The correct method is to reason about **architectural state** and verify invariants.

Step 1: Define state invariants

At each label, write what must be true:

- which registers hold arguments, locals, temporaries,
- what `sp` points to and what is stored at each offset,
- which registers must be preserved across calls,
- loop invariants (e.g., pointer advances, counter decreases).

Step 2: Trace control flow and PC changes

Branches and jumps explicitly update `PC`. Confirm:

- every conditional has a clear fall-through and a clear target,
- every multi-path structure has a join label,
- returns use the correct restored return address.

Step 3: Track memory access precisely

For each load/store, verify:

- effective address (`base + offset`),
- width (byte/halfword/word),
- sign vs zero extension,
- alignment assumptions.

Debug pattern: validate stack correctness with a frame map

```
/* Frame map example (32 bytes):
   28(sp): saved x1
   24(sp): saved x8
   20(sp): saved x9
   16(sp): local spill slot
*/
addi sp, sp, -32
sw   x1, 28(sp)
sw   x8, 24(sp)
sw   x9, 20(sp)
/* ... */
lw   x9, 20(sp)
lw   x8, 24(sp)
lw   x1, 28(sp)
addi sp, sp, 32
jalr x0, 0(x1)
```

Debug pattern: isolate arithmetic correctness with explicit checks

```
/* Check unsigned overflow: sum = a+b; overflow if sum < a */
add  x7, x5, x6
sltu x8, x7, x5
bne  x8, x0, overflow
```

Debug pattern: verify loop termination deterministically

```
/* Invariant: x11 decreases to zero, x10 advances by 4 */
loop:
beq  x11, x0, done
lw   x5, 0(x10)
add  x12, x12, x5
addi x10, x10, 4
addi x11, x11, -1
jal  x0, loop
done:
```

The core rule: do not guess. Write down the intended invariants, step through state transitions, and confirm every `sp` adjustment, every preserved register, every branch condition, and every address computation. RV32I is explicit enough that disciplined reasoning always converges on the bug.

Chapter 12

RV32I as a Foundation for Extensions

12.1 Why RV32I Is Intentionally Minimal

RV32I is designed to be the smallest complete integer ISA that can execute real programs while remaining clean, regular, and easy to implement. The minimalism is intentional and structural:

- **Orthogonality:** a small set of formats covers the full instruction set.
- **Explicitness:** no implicit memory operands, no condition flags, no hidden architectural state.
- **Compiler friendliness:** straightforward lowering from common IR operations.
- **Implementation scalability:** the same base ISA targets tiny microcontrollers and larger cores.

Minimal does not mean limited in reasoning power. RV32I is sufficient to express:

- arithmetic and bit manipulation,

- memory access through loads and stores,
- structured control flow through branches and jumps,
- calls and returns through link-based jumps,
- system-level control through the privileged architecture (outside the user ISA scope).

Example: multiplication without M (pure RV32I)

```
/* x6 = x5 * 10 = (x5<<3) + (x5<<1) */
slli x7, x5, 3
slli x8, x5, 1
add  x6, x7, x8
```

Example: bit-field extraction without special instructions

```
/* Extract bits [15:8] from x5 into x6 */
srli x6, x5, 8
andi x6, x6, 0xFF
```

The discipline benefit is direct: when you understand RV32I, you understand the core execution model that every extension builds upon.

12.2 Relationship to M, A, F, D Extensions

RISC-V is modular: extensions add capabilities without changing the meaning of base instructions. RV32I remains the foundation, while extensions expand the available operations.

M: Integer Multiply/Divide

The M extension adds integer multiply/divide/remainder operations. Without M, multiplication/division must be synthesized.

```
/* Without M: x7 = x5 * 9 = (x5<<3) + x5 */
slli x6, x5, 3
add  x7, x6, x5
```

A: Atomics

The A extension introduces atomic read-modify-write primitives and ordering features for lock-free and concurrent algorithms. Without A, atomicity must be provided by the environment (e.g., interrupts disabled, kernel primitives), not by RV32I instructions.

F and D: Floating-Point

F adds single-precision floating-point; D adds double-precision. Without F/D, floating-point arithmetic is typically handled by:

- software floating-point libraries, or
- integer fixed-point arithmetic.

Example: fixed-point scale idea (integer-only)

```
/* Fixed-point concept: value represents real = x / 256
   Multiply by 3.5 => multiply by 896 and shift right 8
   (3.5 * 256 = 896)
   x5 input, x6 output
*/
```

```
addi x7, x0, 896
/* Without M, large-scale multiply requires decomposition; shown
 ↪  conceptually:
   In real systems, this is performed with M or software routines.
*/
```

In practice, once you move beyond teaching and analysis, real workloads often select M and optionally A/F/D depending on target domain.

12.3 Forward Compatibility Concepts

Forward compatibility is a design contract: new extensions and implementations must not break the base ISA's meaning. For programmers, the important concepts are:

- **Base stability:** RV32I semantics remain stable even as extensions appear.
- **Feature detection:** portable code must not assume extension availability unless the environment guarantees it.
- **Graceful fallback:** write base-ISA code when portability matters; use extensions behind clear boundaries when performance or capability demands it.

Example: boundary-based design (conceptual)

```
/* Pattern idea:
   - Base implementation: shift/add
   - Accelerated implementation: use mul if M is available
   Selection occurs outside the core routine (build-time or runtime).
*/
```

A clean architecture approach is to treat extensions as **optional accelerators** or **capability unlocks**, not as silent assumptions inside generic code paths.

12.4 Preparing for Advanced Booklets

Understanding RV32I thoroughly prepares you for advanced topics because every extension and every ABI layer assumes you already grasp:

- register-only computation,
- explicit memory traffic,
- PC-relative control flow,
- disciplined stack frames,
- caller/callee preservation reasoning.

This booklet therefore sets prerequisites for later booklets that may cover:

- **RV32IM**: multiplication/division and compiler lowering changes
- **Atomics (A)**: lock-free primitives and memory-order reasoning
- **Floating-point (F/D)**: register sets, calling rules, and mixed-mode code
- **Privilege architecture**: traps, CSR access, interrupts, and system calls
- **ABI deep dive**: full register classification, stack layout rules, and interoperability checklists

Example: extension-aware thinking applied to a routine

```
/* sum += a[i] * b[i]
   Base (RV32I): requires software multiply or shift/add
    ↪  decomposition
```

```
   With M: becomes a simple mul + add in the loop
*/
loop:
beq  x11, x0, done
lw   x5, 0(x10)         /* a[i] */
lw   x6, 0(x12)         /* b[i] */

/* Base RV32I: multiply not available here; would call helper or
 ↪  expand */
/* With M: mul x7, x5, x6 */

add  x13, x13, x7       /* sum += product (conceptual) */
addi x10, x10, 4
addi x12, x12, 4
addi x11, x11, -1
jal  x0, loop
done:
```

The clean path forward is always the same: master the base ISA's explicit rules first, then add extensions as controlled capabilities. That is how RV32I serves as a stable foundation for both learning and real-world systems work.

Appendices

Appendix A — RV32I Instruction Summary

This appendix provides a compact, architecture-accurate summary of the RV32I base instruction set. It is intended as a quick reference for programmers who already understand the execution model and need a reliable overview without ambiguity or hidden semantics.

Arithmetic and Logical Instructions

RV32I arithmetic and logical instructions operate strictly on registers. All results are 32-bit and wrap modulo 2^{32}. There are no condition flags.

- **Register–Register (R-Type):**
  - `add`, `sub`
  - `sll`, `srl`, `sra`
  - `and`, `or`, `xor`
  - `slt`, `sltu`
- **Register–Immediate (I-Type):**
  - `addi`

  - `andi`, `ori`, `xori`
  - `slti`, `sltiu`
  - `slli`, `srli`, `srai`

```
/* Arithmetic and logical examples */
add  x7, x5, x6         /* x7 = x5 + x6 */
sub  x7, x5, x6         /* x7 = x5 - x6 */
slli x8, x5, 3          /* x8 = x5 << 3 */
sltu x9, x5, x6         /* x9 = (x5 < x6) unsigned */
```

Load and Store Instructions

Memory access is explicit. All addresses are computed as `rs1 + signext(imm12)`.

- **Loads (I-Type):**
  - `lb` (8-bit, sign-extended)
  - `lh` (16-bit, sign-extended)
  - `lw` (32-bit)
  - `lbu` (8-bit, zero-extended)
  - `lhu` (16-bit, zero-extended)
- **Stores (S-Type):**
  - `sb` (8-bit)
  - `sh` (16-bit)
  - `sw` (32-bit)

```
/* Load/store examples */
lw   x5, 0(x10)         /* load 32-bit word */
lbu  x6, 1(x10)         /* load unsigned byte */
sw   x5, 4(x10)         /* store word */
```

Control Flow Instructions

Control flow is explicit and PC-relative. Comparisons are performed directly between registers.

- **Conditional Branches (B-Type):**
  - `beq`, `bne`
  - `blt`, `bge` (signed)
  - `bltu`, `bgeu` (unsigned)
- **Unconditional Jumps:**
  - `jal` (J-Type, PC-relative, with link)
  - `jalr` (I-Type, register-indirect, with link)
- **Upper Immediate Instructions (U-Type):**
  - `lui`
  - `auipc`

```
/* Control flow examples */
beq  x5, x6, equal
jal  x1, func
jalr x0, 0(x1)
```

Encoding Overview Tables

RV32I uses a fixed 32-bit instruction length. The instruction format is determined by the opcode.

Instruction Formats Overview (Conceptual):

- **R-Type:** `opcode | rd | funct3 | rs1 | rs2 | funct7`
- **I-Type:** `opcode | rd | funct3 | rs1 | imm[11:0]`
- **S-Type:** `opcode | imm[4:0] | funct3 | rs1 | rs2 | imm[11:5]`
- **B-Type:** `opcode | imm | funct3 | rs1 | rs2 | imm`
- **U-Type:** `opcode | rd | imm[31:12]`
- **J-Type:** `opcode | rd | imm`

Key decoding principles:

- The **opcode** selects the instruction class and format.
- **funct3** and **funct7** refine the operation.
- Immediates may be split across fields and are sign-extended unless explicitly unsigned.

```
/* Same opcode family, different funct fields */
add  x7, x5, x6        /* R-Type, funct7 selects add */
sub  x7, x5, x6        /* R-Type, funct7 selects sub */
```

This summary captures the complete RV32I user-level instruction set. Every extension, ABI rule, and advanced execution feature builds upon these encodings and semantics.

Appendix B — Assembly Style Checklist

This appendix is a practical checklist for writing and reviewing clean RV32I assembly. It is designed to prevent the highest-impact classes of bugs: clobbered state across calls, broken stack frames, incorrect signedness in comparisons, and accidental complexity in control flow.

Register Usage Rules

- **Treat `x0` as a constant, not a register.**
  - Writing to `x0` discards results: never use `x0` as a destination unless you intentionally discard.
  - Use `x0` intentionally for clear/copy/jump patterns.
- **Define register roles at function entry.**
  - Identify which registers carry inputs, which are locals, which are temporaries.
  - Keep a small set of scratch registers for short-lived temporaries.
- **Minimize live ranges.**
  - Do not keep values live across calls unless necessary.
  - Recompute cheap values instead of preserving large live state.
- **Respect call boundaries.**
  - Assume volatile registers can be clobbered by callees.
  - Preserve any value needed after a call (saved register or stack slot).
- **Never use uninitialized registers.**
  - Establish constants and initial values explicitly.

```
/* Correct: explicit clear, copy, and intentional discard */
addi x5, x0, 0          /* clear x5 */
addi x6, x5, 0          /* copy x5 -> x6 */
jal  x0, target          /* pure jump (discard link intentionally) */
target:
```

```
/* BUG: accidental discard into x0 */
add  x0, x5, x6          /* result lost */
```

Stack Discipline Checklist

- **Stack grows downward.** Allocation uses `addi sp, sp, -N`.
- **Prologue/epilogue must be symmetric.**
  - If you allocate `N`, you must deallocate exactly `N`.
- **Maintain ABI-required alignment.**
  - Keep `sp` aligned (commonly 16 bytes) at call boundaries.
- **Non-leaf functions must preserve the return address (`x1`).**
  - Any nested call overwrites `x1`; save/restore it if you call out.
- **Save/restore any callee-preserved registers you use.**
- **Never store outside your allocated frame.**

```
/* Clean non-leaf template: aligned, symmetric, preserves x1 */
func:
addi sp, sp, -16
sw   x1, 12(sp)
```

```
jal  x1, helper

lw   x1, 12(sp)
addi sp, sp, 16
jalr x0, 0(x1)
```

```
/* BUG: mismatched frame size */
addi sp, sp, -16
/* ... */
addi sp, sp, 12         /* wrong */
jalr x0, 0(x1)
```

Control Flow Safety

- **Remember: RV32I has no flags.**
  - Every decision must be an explicit compare/branch on registers.
- **Choose signed vs unsigned branches correctly.**
  - Use `blt/bge` for signed integers.
  - Use `bltu/bgeu` for pointers, sizes, and address comparisons.
- **Use structured templates with a single join label.**
  - Avoid multi-exit spaghetti control flow unless deliberately justified.
- **Make unconditional jumps explicit.**
  - Use `jal x0, label` for a pure jump.
- **Loop invariants must be visible.**

- Counters must move toward termination.
- Pointers must advance by element size.

```
/* Clean if/else with explicit join */
blt  x5, x6, then1
addi x7, x0, 0          /* else */
jal  x0, end1
then1:
addi x7, x0, 1          /* then */
end1:
```

```
/* Signed vs unsigned correctness */
blt  x5,  x6,  signed_less   /* signed */
bltu x10, x11, addr_less     /* unsigned for addresses */
```

Clean Code Review Guide

Use this review pass order. It catches most real bugs quickly.

1) Function contract

- What are inputs/outputs? Which registers carry them?
- Which registers are clobbered? Is it documented in a short header comment?

2) Frame correctness

- Is the frame size constant and symmetric?
- Is `x1` saved/restored in non-leaf functions?
- Are offsets within the allocated frame?
- Is stack alignment preserved at call sites?

3) Register lifetime correctness

- Any value used after a call: where is it preserved?
- Any register assumed constant: is it actually written later?
- Any accidental write to `x0`?

4) Control flow correctness

- Are branch conditions correct and signedness correct?
- Is each multi-path structure joined explicitly?
- Do loops terminate (counter decreases, pointer advances)?

5) Memory correctness

- Addresses: correct base+offset? correct scaling?
- Width: `lb/lbu`, `lh/lhu`, `lw` chosen correctly?
- Alignment assumptions made explicit or handled safely?

```
/* Review example: array sum loop invariants are visible */
addi x12, x0, 0         /* sum */
loop:
beq  x11, x0, done      /* counter to zero */
lw   x5, 0(x10)         /* load */
add  x12, x12, x5       /* accumulate */
addi x10, x10, 4        /* pointer advances by 4 */
addi x11, x11, -1       /* counter decreases */
jal  x0, loop
done:
```

This checklist enforces the meaning of clean architecture and clean semantics: explicit state, explicit control flow, disciplined preservation rules, and predictable memory behavior.

Appendix C — Cross-References

This appendix situates the RV32I booklet within the broader CPU Programming Series, highlights architectural correspondences with x86 and ARM materials, and outlines clear next steps for continued study.

Relation to x86 and ARM Booklets

RV32I, x86, and ARM differ historically and stylistically, yet share core execution concepts. Understanding RV32I clarifies these commonalities by removing legacy complexity.

- **Registers vs Flags:**
  - RV32I: no flags; all decisions use explicit register comparisons.
  - x86: arithmetic updates flags implicitly.
  - ARM (AArch64): condition flags exist but are optional and explicit.
- **Memory Access Model:**
  - RV32I: strict load/store; no memory operands in ALU ops.
  - x86: many ALU ops may reference memory directly.
  - ARM: load/store, with richer addressing modes.
- **Control Flow:**
  - RV32I: PC-relative branches and explicit link-based calls.
  - x86: CALL/RET with implicit stack behavior.

- ARM: BL/RET with link register and optional stack usage.

```
/* RV32I: explicit compare and branch */
blt  x5, x6, less

/* x86 (conceptual): implicit flags after cmp */
/* cmp rax, rbx; jl less */

/* AArch64 (conceptual): explicit condition flags */
/* cmp x0, x1; b.lt less */
```

Studying RV32I first makes x86 and ARM behavior easier to interpret, because all implicit mechanisms become explicit patterns you already understand.

Position in the CPU Programming Series

This booklet occupies a foundational position in the series:

- It introduces a **clean, modern RISC execution model**.
- It establishes disciplined reasoning about registers, memory, PC, and stack frames.
- It provides a neutral reference point before engaging with richer or legacy ISAs.

Conceptually, the progression is:

- **Early booklets:** CPU fundamentals, execution flow, and basic ABI ideas.
- **This booklet (RV32I):** minimal complete ISA with explicit semantics.
- **Later booklets:** architecture-specific depth (x86-64, ARM/AArch64), extensions, and system-level topics.

RV32I acts as the **baseline mental model** against which other architectures are compared.

Recommended Next Topics

After mastering RV32I, the recommended next topics are grouped by increasing complexity:

RV32I Extensions

- RV32IM: integer multiply/divide and compiler lowering changes.
- RV32IA: atomic operations and concurrency foundations.
- RV32IF/D: floating-point registers, calling rules, and mixed-mode code.

ABI and Toolchain Depth

- Full calling convention rules and register classification.
- Stack frame layout as produced by optimizing compilers.
- Debugging and profiling conventions.

Privilege and System Programming

- Trap and exception handling.
- CSR access and control flow.
- Interrupts and system calls.

Cross-Architecture Comparison

- x86-64 System V ABI vs RV32I calling discipline.
- ARM AArch64 calling convention and stack usage.
- Mapping compiler IR to different ISAs.

```
/* Example mindset carried forward:
   - Explicit register roles
   - Explicit memory access
   - Explicit control flow
   These invariants remain valid across architectures.
*/
```

The intended path is deliberate: master the explicit, minimal model first; then add complexity as controlled layers. This ensures that every advanced topic builds on solid architectural understanding rather than on memorized patterns.

References

RISC-V Architectural Specifications (Conceptual Use)

The primary conceptual foundation for this booklet is the official RISC-V architectural definition of the base integer instruction set. These specifications define the architectural contract that every compliant implementation must honor, independent of microarchitecture. Core conceptual areas derived from the specifications include:

- the definition of architectural state (register file, program counter),
- instruction semantics for RV32I,
- load/store memory model and alignment rules,
- control-flow behavior and PC-relative addressing,
- stability guarantees of the base ISA.

Throughout this booklet, the specifications are used strictly as a **semantic reference**, not as an implementation guide. The emphasis is on understanding architectural behavior visible to software, not on hardware pipelines or timing.

ISA Encoding and Execution Semantics

Instruction encoding knowledge in this booklet is grounded in the canonical RV32I encoding rules:

- fixed 32-bit instruction width,
- opcode-driven instruction class selection,
- orthogonal use of funct3 and funct7 fields,
- immediate field splitting and sign extension rules,
- deterministic execution semantics independent of implementation.

Encoding concepts are applied only to the extent required to:

- decode compiler-generated assembly correctly,
- reason about instruction behavior without guesswork,
- validate correctness of hand-written assembly.

```
/* Example: same opcode class, different funct fields */
add  x7, x5, x6         /* R-Type: addition */
sub  x7, x5, x6         /* R-Type: subtraction */
```

The execution model described in this booklet always reflects the architectural definition: fetch, decode, execute, and explicit state update.

ABI and Calling Convention Documentation

Interoperability between assembly and higher-level languages depends on ABI rules. The conceptual ABI material used in this booklet covers:

- argument passing conventions,
- return value placement,
- caller-saved and callee-saved register roles,
- stack growth direction and alignment,
- function prologue and epilogue structure.

The ABI is treated as a **contract**, not an optimization guideline. All assembly examples follow ABI-consistent behavior to ensure compatibility with C and C++ code generated by modern compilers.

```
/* ABI-consistent non-leaf function shape (conceptual) */
func:
addi sp, sp, -16
sw   x1, 12(sp)
jal  x1, helper
lw   x1, 12(sp)
addi sp, sp, 16
jalr x0, 0(x1)
```

This discipline allows assembly routines to be linked safely with compiler-generated object code.

Compiler and Toolchain Behavioral References

Understanding real-world RV32I code requires awareness of how modern compilers lower high-level constructs into assembly. The behavioral knowledge referenced in this booklet includes:

- common lowering patterns for expressions, loops, and conditionals,
- register allocation and spill behavior,
- stack frame generation for leaf and non-leaf functions,
- instruction selection constraints imposed by RV32I,
- optimization effects on control flow and memory access.

The focus is on **observable behavior**, not on compiler internals.

```
/* Typical compiler-generated loop pattern */
loop:
beq  x11, x0, done
lw   x5, 0(x10)
add  x12, x12, x5
addi x10, x10, 4
addi x11, x11, -1
jal  x0, loop
done:
```

Such patterns are stable across modern toolchains because they directly reflect RV32I architectural constraints.

Academic and Professional Architecture Materials

The conceptual framing of this booklet is aligned with long-standing academic and professional practices in computer architecture education:

- RISC design principles emphasizing simplicity and regularity,
- separation of ISA and microarchitecture,
- architectural reasoning based on visible state transitions,
- explicit modeling of memory and control flow,
- correctness-first reasoning over performance speculation.

These principles are consistent with how RISC architectures are taught in university curricula and how professional systems engineers reason about low-level code across architectures.

```
/* Architecture-first reasoning:
   - explicit state updates
   - explicit control flow
   - no hidden side effects
*/
```

This reference framework ensures that the content of this booklet remains stable, portable, and relevant as tools evolve, while preserving correctness and clarity as its primary goals.