Section 6: Language & Compiler Architecture

Contents

1 Phase Comparison Overview

1.1 Summary Table

Phase	Platform Alignment	Primary Output	What It Adds	Acceptance / Proof
Phase 1 (LCv1)	CPU v2 + Toolchain v2	Single `.asm` → flat `.bin`	Minimal freestanding language subset; calls/stack frames; if/while; deterministic codegen	Program with calls + loops assembles and runs; emits only v2 mnemonics/directives
Phase 2 (LCv2)	CPU v3 + Toolchain v3 + Boot v3	Image-mode `.bin` (vectors + `0x0100` entry)	OS-aware emission: vector table directives, `_start`, syscall builtin, `SYSCALL`/`IRET` handlers	Image boots at `0x0100`; syscall round-trip via vector `0x00`; handlers end with `IRET`
Phase 3 (LCv3)	CPU v4 + Toolchain v4 + Boot ABI v4	`.o` + linker → final `.bin`	Multi-file compilation; `.text`/`.data`; relocations; globals; structs; ABI freeze; kernel entry `R0=bootinfo_ptr`	Cross-module calls/globals link and run; kernel links `kstub.o + kernel.o` and enters compiled `kernel_main`
Phase 4 (LCv4)	CPU v5 + Toolchain v5 + Boot v5	Dual image: `kernel.bin` + `user.bin`	Kernel/user targets; user restrictions; address-range lints; syscall-only services; protection compatibility (vector `0x03`)	Kernel transitions to user mode; user syscalls work; user illegal access is prevented (or traps predictably)

1.2 Reading Guide

- Each phase section is **normative**: it defines constraints the implementation must obey.
- Each phase ends with a **Definition of Done** and **tests** that should become part of your project validation.

2 Phase Specifications

3 Phase 1 (LCv1): Minimal Freestanding Compiler (CPU v2 + Toolchain v2)

3.1 Purpose and Scope

Goal. Deliver the smallest end-to-end compiler that can translate a tiny, predictable systems language subset into assembly accepted by the **v2 assembler/toolchain**, producing a runnable flat binary for the **CPU v2** emulator.

Compatibility target. This phase is explicitly aligned with:

- **CPU v2** features: practical stack and calls (SP/FP, PUSH8/POP8, CALL_ABS, RET) and comparison (CMP).
- **Toolchain v2**: raw/flat binary output via `asm input.asm -o program.bin` (no image mode, no sections, no linker).

Out of scope (by design). To remain compatible with the earlier roadmap and avoid relying on future infrastructure, LCv1 does *not* include:

- Interrupt vectors, SYSCALL/IRET, MMIO conventions, reset-at-`0x0100` image layout (v3+).
- `.text`/`.data` sections, relocations, object files, linker integration (v4+).
- User/kernel protection, dual-image builds, privilege checks (v5).
- Heap allocation/free, complex runtime services, garbage collection, implicit allocation.

3.2 Phase Inputs, Outputs, and Tool Contracts

Inputs.

- A single source file in the LCv1 language subset (defined in §3.3).

Outputs.

- A single assembly file `program.asm` that uses only mnemonics supported by the v2 assembler.
- (Recommended) A symbol listing `program.sym` emitted by the assembler, if implemented/enabled.

Build and run contract. LCv1 must integrate with the existing toolchain plan without introducing new tools:

```
lc1c source.lc -o program.asm
asm program.asm -o program.bin
emu program.bin
```

Where `lc1c` is the Phase 1 compiler executable (hosted in Python or C for fastest iteration).

3.3 Language Subset (LCv1)

3.3.1 Design Principles (LCv1)

LCv1 follows the global language philosophy: **procedural, statically typed, minimal, explicit**. For Phase 1 specifically:

- Prefer features that map directly to CPU v2 (calls, stack locals, branches, arithmetic).
- Avoid features that require runtime services (heap, exceptions, GC, dynamic dispatch).
- Make all behavior explicit (no implicit allocations, no hidden temporaries with side-effects).

3.3.2 Types

LCv1 defines exactly two value categories:

- `i64`: signed 64-bit integer.
- `ptr`: untyped pointer represented as a 64-bit address (byte addressable).

Type rules.

- Arithmetic (`+ - * / %`) is defined on `i64`.
- Comparisons (`== != < <= > >=`) produce an `i64` boolean value `0` or `1`.
- Pointer arithmetic is allowed only in the form:
  - `ptr + i64` and `ptr - i64` (byte-based offsets in Phase 1).
- No implicit casts between `i64` and `ptr`. Casts must be explicit: `cast_i64(ptr)` and `cast_ptr(i64)`.

3.3.3 Declarations and Functions

Top-level. A program is a set of function definitions. No global variables in Phase 1.

Entry point. The program entry is a required function:

```
fn main() -> i64 { ... }
```

The returned value is placed in `R0` before halting (or returning to a wrapper, depending on your emulator convention). If your emulator expects a HALT instruction, the compiler must emit a final HALT after main returns.

Function signatures.

- Up to 4 parameters per function (to stay aligned with register-only argument passing).
- Return type is either `i64` or `ptr`.
- No recursion restrictions (recursion is allowed; it is the developer's responsibility to avoid stack overflow).

3.3.4 Statements and Expressions

Statements.

- Variable declaration: `let x:  i64 = expr;`
- Assignment: `x = expr;`
- If: `if (cond) {...} else {...}`
- While: `while (cond) {...}`
- Return: `return expr;`

Expressions.

- Literals: integer literals (decimal and/or hex).
- Variable references.
- Unary: `-expr`, `ẽxpr` (optional), explicit casts.
- Binary arithmetic and comparisons.
- Function calls: `f(a,b,c,d)` (max 4 args).

Memory dereference (optional in LCv1). If included, keep it minimal and explicit:

- Load: `load64(p:  ptr) -> i64`
- Store: `store64(p:  ptr, v:  i64) -> i64` (returns stored value)

If your assembler/CPU v2 memory ops are limited, you may defer these to Phase 2 or 3. Do not introduce implicit dereference syntax (like `*p`) in Phase 1 unless you can fully specify its semantics.

3.4 Compiler Architecture (LCv1)

3.4.1 Pipeline

LCv1 implements the full conceptual pipeline, but with minimal complexity:

$$\text{Source} \rightarrow \text{Tokens} \rightarrow \text{AST} \rightarrow \text{(Optional IR)} \rightarrow \text{Assembly (v2)}$$

- Tokens and AST are mandatory.
- An IR is optional in Phase 1; direct AST-to-assembly is acceptable if it stays structured and testable.

3.4.2 Lexer

Responsibilities. Recognize keywords, identifiers, literals, operators, delimiters. Lexer must be deterministic and linear-time.

Diagnostics. Emit precise errors with location:

- invalid character
- malformed number literal
- unterminated block/comment (if comments exist)

3.4.3 Parser (Recursive Descent)

Responsibilities. Build an AST that represents structure (functions, blocks, statements, expressions) independent of surface syntax.

Diagnostics. Parser errors must:

- point to the token that caused the failure,
- show expected tokens/categories,
- continue to recover (best-effort) to report multiple errors per run.

3.4.4 Semantic Analysis

Required checks.

- Scope resolution: variables must be declared before use; lexical scoping across blocks.
- Type checking: enforce the rules in §3.3.
- Function validation: correct arg count/types; return type consistent on all paths.

Failure policy. If semantic analysis fails, code generation must not run.

3.5 Code Generation Contract (CPU v2)

3.5.1 Assembler/ISA subset used

LCv1 codegen must emit only mnemonics available in v2 (and earlier):

- Control flow: direct jumps/branches (ABS/REL), plus **CALL_ABS**, **RET**.
- Stack: **PUSH8**, **POP8**, and use of **SP** and **FP** registers where allowed.
- Comparisons: **CMP** (sets Z for equality/ordering depending on ISA definition in CPU spec).
- Arithmetic and moves from v1 (ADD/SUB/etc. as defined by the assembler spec).

No SYSCALL/IRET, no vector directives, no image-layout directives, no sections.

3.5.2 Calling Convention (LCv1, forward-compatible)

To avoid redesigning later, LCv1 adopts a convention that stays compatible with future phases:

- Arguments 0–3 are passed in `R0..R3`.
- Return value is in `R0`.
- All general registers are treated as **caller-saved** in Phase 1.
- `SP` and `FP` must be restored on return (stack discipline is mandatory).

Phase-1 restriction. Max 4 arguments per function. No stack-passed arguments in Phase 1.

3.5.3 Stack Frame Layout

LCv1 uses `FP` to keep debugging and correctness simple.

Prologue (conceptual). On function entry:

1. Save old FP
2. Set FP to current SP
3. Reserve space for locals (8-byte aligned)

Epilogue (conceptual). On function exit:

1. Deallocate locals (restore SP from FP)
2. Restore old FP
3. `RET`

Alignment. All stack allocations are multiples of 8 bytes.

3.5.4 Conditionals and Loops

Boolean convention. A condition expression evaluates to `i64` where:

$$0 \Rightarrow \text{false}, \quad \neq 0 \Rightarrow \text{true}.$$

Branch lowering.

- Evaluate condition into a register (default: `R0` or a temporary).
- Compare against zero using `CMP`.
- Use Z-based conditional jump(s) to the correct block label.

3.5.5 Register Allocation (Minimal Strategy)

Phase 1 prioritizes correctness:

- Use a small fixed set of temporaries (e.g., `R8`, `R9`, `R10`) for expression evaluation.
- Spill to stack for nested expressions if necessary (explicit, deterministic).
- Do not attempt global register allocation in Phase 1.

3.6 File Layout and Assembly Emission Rules

Single output file. The compiler emits one `.asm` file with:

- one label per function (e.g., `fn_main:`)
- internal labels for blocks (`L_if_else_3`, `L_while_cond_2`, etc.)

No section directives. LCv1 must not emit v3/v4 directives such as `.vectors`, `.entry`, `.text`, `.data`.

Optional `.org`. Only emit `.org` if toolchain v2 explicitly supports it in your implementation and you need a fixed load address. Otherwise, assume the loader places the flat blob at the expected base address for v2.

3.7 Execution Checklist (LCv1)

1. **Define the LCv1 grammar** (functions, blocks, let/assign, if/while, return, expressions).
2. **Implement lexer** with location tracking and minimal, strict diagnostics.
3. **Implement recursive descent parser** and AST nodes for all constructs.
4. **Implement semantic analysis**:
   - symbol table with lexical scopes,
   - type checking,
   - function signature validation.
5. **Implement codegen**:
   - stack frame layout using SP/FP,
   - calling convention (R0..R3 args, R0 return),
   - lowering of if/while via CMP + conditional branches,
   - fixed-register expression evaluation with deterministic spilling.
6. **Integrate with assembler v2**:
   - compile → assemble → run in emulator,
   - ensure only v2 mnemonics/register names are used (SP/FP allowed).
7. **Write tests** (see §3.8) and ensure they are part of CI or the project's standard test flow.

3.8 Validation Tests

3.8.1 Required tests (must pass)

T1: Arithmetic + locals.
- Uses `let`, assignment, arithmetic.
- Verifies expected return value in `R0`.

T2: Control flow.
- Contains at least one `if/else` and one `while`.
- Verifies loop termination and correct result.

T3: Function calls.
- A helper function with 1–4 args is called from `main`.
- Verifies call/return correctness and stack discipline (SP restored).

3.8.2 Diagnostics tests (must pass)

- Undefined variable
- Type mismatch (`i64` vs `ptr` without cast)
- Wrong argument count
- Missing return on a function with non-void return type

3.9 Deliverables

1. **Compiler executable** (`lc1c`) runnable on the host machine.
2. **Language reference for LCv1 subset** (this section + a short grammar appendix if desired).
3. **Test suite** for required programs and diagnostics.
4. **Example programs**:

- `examples/arith.lc`
- `examples/control.lc`
- `examples/calls.lc`

5. **Generated assembly samples** in `examples/out/` for inspection/debugging.

3.10 Definition of Done (LCv1)

Phase 1 is complete when:

1. All required tests compile to `.asm`, assemble with the v2 assembler, and run correctly in the emulator.
2. The compiler emits *only* v2-compatible assembly (no v3+ directives or mnemonics).
3. The compiler provides clear syntax and semantic diagnostics with source locations.
4. The calling convention and stack discipline are consistent across all compiled functions.

3.11 Known Risks and Guardrails

- **Feature creep:** do not add structs, globals, sections, syscalls, or heap allocation in LCv1.
- **Unstable ABI:** keep the LCv1 calling convention aligned with future phases (R0..R3 args, R0 return).
- **Hidden runtime:** do not introduce implicit allocations or complex runtime services.
- **Over-optimization:** keep codegen naive; correctness is the only priority in Phase 1.

4 Phase 2 (LCv2): OS-Aware Compiler (CPU v3 + Toolchain v3)

4.1 Purpose and Scope

Goal. Extend the Phase 1 compiler into an **OS-shaped** compiler that can generate **CPU v3** compatible programs using the **toolchain v3 image mode** conventions:

- vector table occupies `0x0000..0x00FF`,
- reset entry begins at `PC=0x0100`,
- `SYSCALL` transfers control through vector `0x00` and returns via `IRET`,
- timer interrupt enters vector `0x01` and returns via `IRET`.

Compatibility target. LCv2 must remain compatible with the previously defined roadmaps:

- **CPU v3**: vectors, `SYSCALL/IRET`, timer IRQ model, MMIO reserved region.
- **Toolchain v3**: assembler directives `.vectors`, `.vector`, `.entry`, `.text`, and **image-mode output** that places vectors at `0x0000` and code starting at `0x0100`.

Out of scope (by design). Phase 2 does *not* include:

- BootInfo / Boot ABI (introduced in Section 5 v4).
- `.data` sections, object files `.o`, relocations, linker integration (toolchain v4+).
- User/kernel mode separation and protection checks (CPU/toolchain v5).
- Heap allocation, GC, exceptions, or any hidden runtime services.

4.2 Phase Inputs, Outputs, and Build Contracts

Inputs.

- A single LCv2 source file (language subset defined in §4.4).
- A target selection: `-target=v3-image` (default for Phase 2).

Outputs.

- A single assembly file `program.asm` that uses only mnemonics/directives supported by the **v3 assembler**.
- A memory image binary `program.bin` produced by the assembler in **image mode** (vectors + text).

Build contract. LCv2 is built and run with the existing toolchain interfaces:

```
lc2c source.lc --target=v3-image -o program.asm
asm program.asm --mode=image -o program.bin
emu program.bin
```

The exact assembler CLI flags may vary, but the output must be an **image** containing:

- vector table at `0x0000`,
- reset entry code starting at `0x0100`,
- zero-filled gaps as needed.

4.3 Binary Layout and Directives (Normative)

4.3.1 Memory Regions

LCv2 code generation must respect the v3 memory layout:

Region	Range	Rule
Vectors	`0x0000..0x00FF`	Present; vector table entries (u16 addresses)
Reset entry	`0x0100..`	`_start` is placed here (reset begins at `PC=0x0100`)
Reserved MMIO	`0xFE00..0xFFFF`	Reserved; compiler must not place code/data here

4.3.2 Directive Emission (Required)

The compiler must emit an assembly prelude compatible with the v3 assembler directive semantics:

```
.vectors
.vector 0, syscall_handler     % vector 0x00
.vector 1, timer_handler       % vector 0x01
.entry _start                  % reset begins at PC=0x0100
.text
```

Vector table contract. Each vector entry is stored as a **2-byte little-endian u16 address** at `0x0000 + 2*id`. The assembler is responsible for writing these entries in image mode based on `.vector` directives.

4.4 Language Subset (LCv2)

4.4.1 Carry-over from LCv1

LCv2 includes everything from LCv1:

- types: `i64`, `ptr`
- statements: `let`, assignment, `if/else`, `while`, `return`
- expressions: literals, vars, arithmetic, comparisons, calls
- calling convention baseline: args in `R0..R3`, return in `R0`

4.4.2 New in LCv2: Syscall Builtin

LCv2 adds a single OS-facing builtin:

```
syscall(n: i64, a1: i64, a2: i64, a3: i64) -> i64
```

Syscall register ABI (normative). The compiler must lower `syscall(n,a1,a2,a3)` as:

- place `n` in `R0`,
- place `a1,a2,a3` in `R1,R2,R3`,
- execute `SYSCALL`,
- treat `R0` as the return value after `IRET`.

Minimal syscall numbers (proof-of-life). LCv2 standardizes the minimal syscalls used in examples/tests:

- `1 = sys_putc`: character in `R1`
- `2 = sys_halt`: halts execution

In Phase 2, these syscalls are primarily used to prove control transfer and return correctness.

4.4.3 Interrupt Model Constraints (v3/v4)

No software interrupt mask. In CPU v3/v4 there is **no architectural interrupt-enable/mask bit**. LCv2 must not introduce language features that imply the ability to disable interrupts. (Interrupt masking is introduced later in v5.)

4.5 Compiler Architecture (LCv2)

4.5.1 Pipeline

LCv2 uses the same conceptual pipeline as LCv1:

$$\text{Source} \rightarrow \text{Tokens} \rightarrow \text{AST} \rightarrow \text{(Optional IR)} \rightarrow \text{Assembly (v3)}$$

with the additional responsibility of emitting **image layout directives** and **system stubs** (`_start`, handlers).

4.5.2 Target Profiles

The compiler must support at least:

- `-target=v2-flat` (optional compatibility mode; emits LCv1-style flat assembly)
- `-target=v3-image` (required; emits vectors/entry/text and uses `SYSCALL`/`IRET`)

4.6 Code Generation Contract (CPU v3)

4.6.1 Instruction/Directive Subset Used

LCv2 may use all LCv1 instructions plus v3 additions:

- `SYSCALL` for system call entry
- `IRET` for return from syscall/IRQ handlers
- v3 directive set: `.vectors`, `.vector`, `.entry`, `.text`

No `.data`, no `.align`, no relocations, no object format output.

4.6.2 Calling Convention (Forward-Compatible Baseline)

To stay compatible with later phases and the syscall ABI:

- Function args 0–3 in `R0..R3`, return in `R0`.
- Phase 2 still treats GPRs as **caller-saved** for normal function calls, as in Phase 1.

4.6.3 System Stubs and Required Symbols

LCv2 must generate the following symbols in the output assembly:

`_start` (reset entry at `0x0100`). `_start` is the entry point used by `.entry`. It must:

1. initialize `SP` and `FP` to the project's recommended stack top (as defined by the OS/boot section),
2. call `main()`,
3. halt deterministically (typically via `syscall(2,0,0,0)`).

`syscall_handler` (vector `0x00`). The handler must:

- end with `IRET` (normative),
- implement at least syscalls 1 and 2 for proof-of-life.

`timer_handler` (vector `0x01`). The handler must:

- end with `IRET` (normative),
- be safe even if it does nothing (a minimal handler may immediately `IRET`).

4.6.4 Register Preservation Rules in Handlers (Normative for LCv2 Output)

Because traps/IRQs are asynchronous and may occur between arbitrary instructions, the generated handlers must follow conservative preservation rules.

Rule.

- `syscall_handler` and `timer_handler` must preserve **all registers** except those intentionally used for syscall ABI (`R0..R3`).
- If a handler uses any register outside `R0..R3`, it must save/restore it (e.g., via `PUSH8`/`POP8`).

Minimal safe implementation strategy.

- `timer_handler`: do not touch any registers; immediately `IRET`.
- `syscall_handler`: touch only `R0..R3`; preserve nothing else by construction.

4.6.5 Syscall Handler Semantics (LCv2 Proof-of-Life)

Dispatch.

- If `R0 == 1`: perform `sys_putc` using `R1` as the character.
- If `R0 == 2`: perform `sys_halt`.
- Otherwise: return an error code (recommended: `R0 := -1`) and `IRET`.

Console output mechanism. The exact mechanism for `sys_putc` must match the project's v3 MMIO/debug convention:

- either write to the console MMIO address if defined by the platform,
- or use the emulator's debug channel if that is what Section 5 v3 mandates.

The compiler must not invent a new device protocol; it must use the one already specified for v3.

4.7 Execution Checklist (LCv2)

1. **Add target selection and v3 emission mode**:
    - implement `-target=v3-image` output,
    - keep `-target=v2-flat` (optional) for regression testing.
2. **Implement directive prelude emission** (§4.3):
    - `.vectors`, `.vector 0`, `.vector 1`, `.entry`, `.text`.

3. **Implement required stubs** (§4.6.3):
    - generate `_start` at reset entry,
    - generate `syscall_handler` and `timer_handler`.
4. **Lower syscall builtin**:
    - move args into `R0..R3`,
    - emit `SYSCALL`,
    - treat `R0` as return value.
5. **Preservation discipline for handlers**:
    - ensure handlers touch only `R0..R3` or save/restore any other register used.
6. **Integrate with v3 assembler image mode**:
    - assemble to an image binary that includes vectors and reset entry at proper addresses,
    - verify vector table entries are correct u16 little-endian addresses.
7. **Add tests** (see §4.8).

4.8 Validation Tests

4.8.1 Required tests (must pass)

T1: Syscall round-trip (putc). A program that calls:

```
syscall(1, 'A', 0, 0);
syscall(2, 0, 0, 0);
```

Expected behavior: prints `A` once and halts.

T2: Handler return correctness. A program that executes multiple syscalls in sequence and verifies return values in `R0` (e.g., unknown syscall returns `-1`, known syscall returns `0`).

T3: Timer handler safety. If the emulator triggers timer IRQs in v3:
- run a loop-heavy program long enough to take at least one timer interrupt,
- confirm no corruption of program state (result matches expected output),
- confirm the timer handler returns via `IRET`.

If the emulator does not yet trigger timer IRQs, the timer handler must still be emitted and must assemble.

4.8.2 Structural tests (must pass)

- The emitted binary contains vectors at `0x0000..0x00FF`.
- `_start` is located at `0x0100` in image mode.
- Vector `0x00` points to `syscall_handler`; vector `0x01` points to `timer_handler`.

4.9 Deliverables

1. **Compiler executable** (`lc2c`) with `-target=v3-image`.
2. **LCv2 language reference updates**:
    - `syscall()` builtin definition and lowering rules,
    - explicit statement that v3/v4 have no interrupt mask in software.
3. **Examples**:
    - `examples/sys_putc.lc`
    - `examples/sys_halt.lc`
    - `examples/sys_unknown.lc`
4. **Test suite** covering syscall lowering, image layout, and handler semantics.

4.10 Definition of Done (LCv2)

Phase 2 is complete when:

1. The compiler emits v3-compatible directives and assembly that assembles in **image mode**.
2. The produced image places the vector table at `0x0000..0x00FF` and reset entry at `0x0100`.
3. `syscall()` reliably transfers control through vector `0x00` and returns via `IRET`, with return value in `R0`.
4. The output includes required handlers for vectors `0x00` and `0x01`, both ending with `IRET`.
5. All required tests pass, including syscall proof-of-life and layout checks.

4.11 Known Risks and Guardrails

- **Do not invent new devices:** `sys_putc` must use the v3-defined console mechanism.
- **Do not assume interrupt masking:** v3/v4 do not provide an IE bit; avoid language features that imply it.
- **No linker assumptions:** keep Phase 2 output as a single assembly unit; no `.o`/relocations.
- **Handler safety:** keep timer handler minimal and non-invasive unless/until a full trap-frame ABI is frozen.

5 Phase 3 (LCv3): Object Files, Linker Integration, and ABI Freeze (CPU v4 + Toolchain v4)

5.1 Purpose and Scope

Goal. Upgrade the language and compiler into a **project-scale** tool that supports:

- multi-file builds via **object files (.o)** and a **linker**,
- explicit **.text/.data** emission, alignment, and data layout,
- a frozen, written **ABI and calling convention** that all future code (kernel, user, libraries, asm) must obey,
- direct integration with the **Boot ABI v4** (`R0 = bootinfo_ptr` on kernel entry).

Compatibility target. LCv3 must align with the previously defined v4 milestones:

- **CPU v4**: base+offset addressing mode, wide aligned loads/stores, extended register set.
- **Toolchain v4**: assembler emits **.o** with **relocations**, supports **.text/.data/.align**, and the linker produces final binaries from multiple objects.
- **Boot ABI v4**: kernel entry receives `bootinfo_ptr` in `R0`.

Out of scope (by design). Phase 3 does *not* include:

- user/kernel protection and dual-image boot (v5),
- privilege restrictions and trap-based protection rules (v5),
- a heap allocator or GC (still forbidden for the minimal systems runtime philosophy).

5.2 Build Model (Multi-File) and Tool Contracts

Inputs.

- One or more LC source files (modules).
- A build graph specifying which sources compile into which output (can be a simple list in Phase 3).

Outputs.

- Per-source object files: `*.o`
- A final linked binary:
  - `kernel.bin` (for kernel target builds), or
  - `program.bin` (for standalone programs).

Normative build flow. The Phase 3 compiler must fit into the v4 toolchain model:

```
lc3c -c a.lc -o a.o
lc3c -c b.lc -o b.o
ld  a.o b.o -o program.bin
emu program.bin
```

The assembler may be used internally or as a backend stage, but the **observable interface** is `.o + ld`.

5.3 Target Profiles

LCv3 introduces explicit targets:

- `-target=standalone`: produces a runnable program entry (`_start`) + vectors if desired by image-mode.
- `-target=kernel`: produces a linkable `kernel_main` with the Boot ABI v4 contract.

Important rule. In `-target=kernel`, the compiler must **not** emit vector tables or reset entry stubs by default. Those belong to the kernel stub/boot layer (Section 5). The compiler's job is to emit linkable code and data. This prevents conflicts with the existing boot plan where `kstub` owns the early reset flow.

5.4 ABI and Calling Convention (Frozen at Phase 3)

5.4.1 Register Roles

General registers. LCv3 assumes the CPU v4 general-purpose register file `R0..R15` exists (or the project's defined v4 count).

Special registers. `SP` and `FP` are dedicated to stack operations and stack frames.

5.4.2 Function Call ABI (Normative)

Arguments and return.

- Arguments 0–3 are passed in `R0..R3`.
- Return value is passed in `R0`.

Caller-saved vs callee-saved (frozen). To maintain compatibility with the OS/syscall conventions described earlier and the kernel's expectations, Phase 3 freezes the following rule set:

- **Callee-saved:** `R8..R15`, plus `FP`.
- **Caller-saved:** `R0..R7`.

Rationale.

- `R0..R3` must remain volatile due to syscall and argument passing usage.
- Reserving `R8..R15` as callee-saved gives the compiler stable temporaries for codegen and optimization.

Obligation. Any function that modifies a callee-saved register must restore it before returning.

5.4.3 Stack Frame ABI (Normative)

Alignment. The stack pointer `SP` must remain 8-byte aligned at all public call boundaries.

Prologue and epilogue (conceptual).

- Prologue saves old `FP`, sets `FP := SP`, reserves locals, and saves any callee-saved registers used.
- Epilogue restores callee-saved registers, deallocates locals, restores `FP`, and returns.

5.4.4 Boot ABI v4 Integration (Normative)

When compiled as a kernel entry, LCv3 must expose:

```
fn kernel_main(bootinfo: ptr) -> i64 { ... }
```

and it must assume:

- on entry, `R0 = bootinfo_ptr`,
- `SP/FP` are already initialized by the boot stub (Section 5),
- interrupts are not maskable in v4 (no IE bit yet), so do not assume they can be disabled in software.

5.5 Object File Emission Model (Toolchain v4)

5.5.1 Sections

LCv3 object output must support:

- **.text**: executable code
- **.data**: initialized global data
- (Optional) **.bss**: zero-initialized data, if the linker supports it; otherwise emulate with **.data** + zero literals.

5.5.2 Symbols

Global symbols. Functions and globals that are referenced across modules must be emitted as global symbols.

Local symbols. Block labels and temporary compiler-generated symbols must be local to the object.

5.5.3 Relocations

The compiler must rely on the assembler/linker relocation model for:

- calls to functions defined in other objects,
- addresses of global data symbols defined in other objects,
- jump tables (if introduced) and other address-bearing constants.

Important restriction. LCv3 must not hardcode absolute addresses of external symbols. Always emit relocation-capable references.

5.6 Language Extensions (LCv3)

5.6.1 Global Variables

LCv3 introduces global variables in `.data`:

```
let g_counter: i64 = 0;
```

Rules.

- Global initializers must be compile-time constants (integer literals, address-of global symbol if supported).
- No global constructors or runtime initialization.

5.6.2 Structs (Packed and Aligned Layout)

LCv3 introduces structs with explicit, deterministic layout rules.

Declaration.

```
struct Point {
  x: i64;
  y: i64;
}
```

Layout rules (normative).

- Each field is placed at the lowest offset that satisfies its alignment.
- In LCv3, `i64` and `ptr` have alignment 8.
- Struct alignment is the max alignment of its fields.
- Struct size is padded up to a multiple of struct alignment.

Field access. Field access is lowered using **CPU v4 base+offset addressing** whenever possible.

5.6.3 Pointers and Address-of

LCv3 allows taking the address of globals and locals:

- `addr(x)` returns `ptr` to a local or global.
- `load64(p)` and `store64(p, v)` are now **required** in Phase 3 (no longer optional).

5.7 Code Generation Contract (CPU v4)

5.7.1 Use of v4 Addressing

Rule. For loads/stores of struct fields and locals, the compiler should prefer:

- base+offset mode with `FP` or a base register,
- aligned wide operations when alignment permits.

Fallback. If an offset does not fit the addressing mode constraints (if any exist in your ISA spec), compute address in a temp register and use a generic load/store.

5.7.2 Register Allocation (Improved but Still Simple)

LCv3 may introduce a local register allocator:

- expression trees allocated with a small pool of temporaries,
- reuse callee-saved registers `R8..R15` inside functions (with correct save/restore),
- spill when necessary with deterministic rules.

5.8 Kernel Integration Plan (v4 Boot ABI)

5.8.1 Required Kernel Entry Signature

The compiler must support emitting an externally visible kernel entry:

```
fn kernel_main(bootinfo: ptr) -> i64
```

5.8.2 BootInfo Access Pattern (Normative)

Because BootInfo layout is defined in Section 5 v4, LCv3 must:

- define a matching struct type in a shared header/module,
- access its fields via known offsets (struct layout rules),
- never assume BootInfo is at a fixed address; always use `bootinfo_ptr` from `R0`.

5.8.3 Link Composition

A typical kernel build must be:

- `kstub.o` (handwritten asm that owns vectors/reset) + `kernel.o` (compiled) + optional libs
- linked into `kernel.bin` by `ld`.

Important constraint. The compiler must not emit its own vector table when building `-target=kernel` to avoid conflicting ownership with `kstub`.

5.9 Execution Checklist (LCv3)

1. **Freeze the ABI** (§5.4) as a project contract.
2. **Implement multi-file compilation**:
   - module-level symbol tables,
   - separate compilation units.
3. **Emit object files** (§5.5):
   - `.text`/`.data` sections,
   - global and local symbols,
   - relocation-capable references.
4. **Add language features**:
   - globals with constant initialization,
   - structs with deterministic layout,
   - required load/store builtins.
5. **Implement v4-aware codegen**:
   - base+offset addressing for locals/fields,
   - aligned wide ops where appropriate.
6. **Integrate kernel entry build**:
   - `-target=kernel` emits `kernel_main` symbol only,
   - link with `kstub.o` into `kernel.bin`,
   - verify Boot ABI v4 (`R0=bootinfo_ptr`).
7. **Add tests** (see §5.10).

5.10 Validation Tests

5.10.1 Object/link tests (must pass)

T1: Cross-module call.

- `a.lc` defines `fn add(a,b)->i64`
- `b.lc` calls `add` from `main`
- Must link and run correctly; confirms call relocations.

T2: Cross-module global access.

- `a.lc` defines `let g:  i64 = 7;`
- `b.lc` reads `g` and returns `g+1`
- Confirms data symbol relocations.

5.10.2 Struct layout tests (must pass)

T3: Struct field offsets.

- Define a struct and compute addresses of fields using `addr` + known offsets.
- Validate loads/stores operate on correct memory locations.

5.10.3 Kernel ABI tests (must pass)

T4: BootInfo pointer entry.

- Link `kstub.o + kernel.o`
- `kernel_main` reads a known BootInfo field and returns a recognizable value.
- Confirms `R0` entry passing and struct offset usage.

5.11 Deliverables

1. **Compiler executable** (`lc3c`) supporting `-c` object emission and `-target=kernel`.
2. **ABI specification** included in the project docs (this section is normative).
3. **Module/library skeleton**:
   - `lib/abi.lc` (shared ABI declarations)
   - `lib/bootinfo.lc` (BootInfo struct matching Section 5 v4)
4. **Test suite** for relocations, structs, and kernel entry.

5.12 Definition of Done (LCv3)

Phase 3 is complete when:

1. The compiler emits correct `.o` objects with `.text/.data` and relocations.
2. Multi-file projects link via the v4 linker and run correctly.
3. The ABI is frozen and enforced:
   - args in `R0..R3`, return in `R0`,
   - `R8..R15` + `FP` are callee-saved, `R0..R7` are caller-saved,
   - `SP` 8-byte aligned at call boundaries.
4. Kernel integration works:
   - `kernel_main(bootinfo_ptr)` is reachable from `kstub`,
   - BootInfo fields are read correctly via base+offset access.

5.13 Known Risks and Guardrails

- **ABI drift:** after Phase 3, ABI changes are breaking changes. Treat this section as a contract.

- **Absolute addressing:** never bake absolute addresses for external symbols; always rely on relocations.
- **Compiler vs boot ownership:** kernel builds must not emit vectors/reset stubs (owned by Section 5).
- **Runtime creep:** globals must remain constant-initialized; no constructors or hidden init passes.

6 Phase 4 (LCv4): Dual-Image, Protection-Aware Kernel/User Compilation (CPU v5 + Toolchain v5)

6.1 Purpose and Scope

Goal. Finalize the language and compiler architecture so it can build **both kernel and user programs** under the v5 platform rules:

- **dual-image output model** (kernel image + user image),
- **CPU v5 protection model** (kernel-only ranges + user-accessible ranges),
- **explicit privilege separation** where user programs may only request services via **syscalls** (vector `0x00`),
- **compiler-enforced restrictions** for user targets (static checks that prevent privileged behavior).

Compatibility target. LCv4 must align with the existing v5 milestones from the earlier sections:

- **CPU v5**: interrupt-enable flag `I`, kernel-mode flag `K`, protection traps on illegal user access, and vector `0x03` reserved for protection faults.
- **Toolchain v5**: dual-image build flow (`kernel.bin` + `user.bin`), plus any packing/launch convention defined in the toolchain roadmap.
- **Boot v5**: kernel boots first, validates BootInfo, sets up user entry + stack, clears `K`, and jumps to user entry.

Out of scope (by design). Even at Phase 4, the language remains minimal and explicit:

- no garbage collection,
- no implicit allocation or hidden runtime initialization,
- no exceptions.

6.2 V5 Memory Map and Privilege Rules (Normative)

6.2.1 Regions

The compiler must respect the v5 memory partitioning used by the OS/boot plan:

Region	Range	Rule
Vectors	`0x0000..0x00FF`	Kernel-only; vector table / reserved low memory
User accessible	`0x0100..0xFDFF`	User may execute/read/write *within* this range
MMIO + reserved	`0xFE00..0xFFFF`	Kernel-only; MMIO and reserved

6.2.2 Protection rule (CPU-enforced)

When `K = 0` (user mode), the CPU raises a protection fault (vector `0x03`) on any:

- instruction fetch, load, or store to an address in `0x0000..0x00FF` or `0xFE00..0xFFFF`,
- attempt to execute privileged instructions (if the ISA defines any as privileged under `K=0`).

6.2.3 Compiler rule (static enforcement)

To prevent user-mode crashes-by-construction, LCv4 must statically enforce **user safety rules** (see §6.6).

6.3 Targets and Build Products

6.3.1 Targets

LCv4 defines two explicit compilation targets:

`-target=kernel`. Produces objects and a final `kernel.bin` intended to run in kernel mode `K=1`. Kernel code may:

- install vectors and handlers,
- access MMIO (`0xFE00..0xFFFF`),
- access low memory vectors region (`0x0000..0x00FF`),
- transition to user mode and jump to user entry (Boot v5 contract).

`-target=user`. Produces objects and a final `user.bin` intended to run in user mode `K=0`. User code must:

- be linked/located entirely within `0x0100..0xFDFF`,
- use syscalls (vector `0x00`) as the only OS service mechanism,
- avoid any MMIO or low-memory access patterns.

6.3.2 Normative dual-image build flow

A v5 build is defined as:

```
# Kernel
lc4c --target=kernel -c kmain.lc -o kmain.o
ld  kstub.o kmain.o klib.o -o kernel.bin

# User
lc4c --target=user -c app.lc -o app.o
ld  ucrt0.o app.o ulib.o -o user.bin

# Run (the emulator/loader uses both images as defined by toolchain v5)
emu kernel.bin user.bin
```

Ownership constraint (important).

- Kernel vectors/reset flow remain owned by the boot/kernel stub layer (Section 5), not by the compiler.
- For `-target=user`, the compiler must not emit vectors and must not assume it controls reset; it must emit a user entry symbol (see below) compatible with the kernel's user-jump contract.

6.4 ABI and Entry Contracts (Reusing Phase 3 ABI)

6.4.1 Function call ABI

Phase 4 inherits and does not change the Phase 3 ABI:

- args 0–3 in `R0..R3`, return in `R0`,
- callee-saved: `R8..R15` + `FP`,
- caller-saved: `R0..R7`,

- `SP` 8-byte aligned at public call boundaries.

6.4.2 Kernel entry (Boot v5 compatibility)

Kernel entry remains defined by the boot layer (kstub/boot) and must continue to pass `bootinfo_ptr` in `R0` to the compiled kernel entry point:

```
fn kernel_main(bootinfo: ptr) -> i64
```

6.4.3 User entry (Boot v5 compatibility)

The user program must expose a single externally visible entry symbol that the kernel will jump to after switching to user mode:

```
fn user_main() -> i64
```

User return policy. Because user code runs under kernel supervision, LCv4 standardizes that:

- user code must terminate by calling `syscall(SYS_EXIT, code, 0, 0)` (preferred), or
- if it returns from `user_main`, the compiler-generated user crt0 must translate the return value into `SYS_EXIT`.

6.5 CPU v5 Flags and Interrupt Handling (Normative Semantics)

6.5.1 `K` (Kernel Mode) flag

- `K=1`: privileged (kernel) execution.
- `K=0`: user execution; protection rules enforced.

6.5.2 `I` (Interrupt Enable) flag

- `I=1`: IRQs may be taken.
- `I=0`: IRQs are masked (not taken).

Compiler visibility.

- Kernel target may optionally provide intrinsics to set/clear `I`.
- User target must not expose `I` manipulation intrinsics (or must make them a compile-time error).

6.6 User Safety Rules (Compiler-Enforced)

LCv4 must implement **static restrictions** for `-target=user` to ensure code is valid under v5 protection.

6.6.1 Address-range lints (mandatory)

The compiler must reject user builds that include:

- integer-to-pointer casts that yield a provably out-of-range constant address,
- taking the address of known kernel symbols or known MMIO symbols,
- any constant absolute address in `0x0000..0x00FF` or `0xFE00..0xFFFF`.

Minimum rule. If the compiler cannot prove an address is safe (e.g., pointer obtained from unknown arithmetic), it may:

- allow it (and rely on runtime CPU traps), but emit a warning, or
- reject it under a strict mode (recommended: `-Werror=unsafe-user-ptr`).

6.6.2 Forbidden operations in user target (mandatory)

User builds must not allow:

- inline assembly that emits privileged instructions,
- direct MMIO write helpers or device register access helpers,
- installation of vectors or emission of `.vectors`/`.vector`/`.entry` directives.

6.6.3 Syscall-only OS services (mandatory)

The only standardized mechanism for a user program to request kernel services is:

```
syscall(n: i64, a1: i64, a2: i64, a3: i64) -> i64
```

lowered identically to Phase 2/3 (R0..R3 + `SYSCALL`).

6.7 User Runtime Stub (ucrt0) and Linking Model

6.7.1 Rationale

User binaries must be fully linkable and runnable without owning the machine reset flow. Therefore LCv4 defines a minimal user runtime stub object `ucrt0.o` that:

- provides the exported entry symbol expected by the kernel (either `_user_start` or `user_main` depending on your boot contract),
- initializes `SP/FP` if the kernel does not do so (prefer kernel-provided stack setup; do not duplicate responsibilities),
- calls `user_main()`,
- converts a return value into `SYS_EXIT`.

6.7.2 User link placement (normative)

The linker script (or equivalent toolchain v5 mechanism) must place:

- `.text` and `.data` of the user image entirely within `0x0100..0xFDFF`,
- user stack top within `0x0100..0xFDFF` (as provided in BootInfo v5).

The compiler must not emit absolute placements conflicting with this policy.

6.8 Kernel/Boot Integration Requirements (v5)

6.8.1 BootInfo usage (kernel target)

Kernel code compiled under `-target=kernel` must:

- read user image entry and user stack parameters from BootInfo (as defined in Section 5 v5),
- validate that user entry lies within `0x0100..0xFDFF`,
- validate that user stack top lies within `0x0100..0xFDFF`,
- prepare user mode transition (set `K:=0`) and jump to user entry.

6.8.2 Protection fault path (v5)

Vector `0x03` is reserved for protection faults. LCv4 requires that:

- kernel builds include a `prot_fault_handler` (typically in the kernel stub/asm layer),
- user builds do not define or install this vector.

6.9 Compiler Architecture (LCv4)

6.9.1 Pipeline

LCv4 maintains the same conceptual pipeline as previous phases:

$$\text{Source} \to \text{Tokens} \to \text{AST} \to \text{IR} \to \text{Object (.o)} \to \text{Link} \to \text{Image}$$

with target-dependent validation and codegen rules.

6.9.2 Target-dependent validation stages

Common validation.

- typing, scoping, struct layout, ABI correctness.

User-only validation.

- address-range linting,
- forbidden feature checks,
- link placement checks (at minimum: ensure no absolute out-of-range constants are emitted).

Kernel-only validation.

- allow privileged intrinsics (MMIO, vector ownership not emitted by compiler but allowed in linked asm stubs),
- allow `I` flag intrinsics if defined.

6.10 Execution Checklist (LCv4)

1. **Add dual-target compilation**:
   - `-target=kernel` and `-target=user`
   - target-dependent feature gating
2. **Add user safety validation** (§6.6):
   - constant address range checking
   - forbidden feature checks (inline asm, MMIO helpers, vector directives)
3. **Define and implement user crt0 object model** (§6.7):
   - entry symbol policy
   - `SYS_EXIT` termination policy
4. **Integrate with v5 dual-image toolchain**:
   - build and link `kernel.bin` and `user.bin`
   - run via emulator with both images
5. **Kernel-side BootInfo validation hooks**:
   - compiled kernel reads BootInfo user entry/stack fields
   - enforces range checks before transitioning to user mode
6. **Add tests** (see §6.11).

6.11 Validation Tests

6.11.1 Dual-image boot tests (must pass)

T1: Kernel-to-user transition.

- Kernel validates BootInfo user entry and stack range.
- Kernel switches to user mode (`K:=0`) and jumps to user entry.
- User program runs and exits via syscall.

T2: User syscall round-trip.

- User program calls `syscall(SYS_PUTC, 'U', 0, 0)` then `SYS_EXIT`.
- Output contains the character and the system terminates cleanly.

6.11.2 Protection tests (must pass)

T3: Illegal low-memory access traps.

- User program attempts to read from `0x0000` via a forbidden constant pointer.
- Compiler must reject it (preferred) or warn; if allowed, CPU must trap to vector `0x03`.

T4: Illegal MMIO access traps.

- User program attempts to write to `0xFE00` via a forbidden constant pointer.
- Compiler must reject it (preferred) or warn; if allowed, CPU must trap to vector `0x03`.

6.11.3 Static restriction tests (must pass)

- Inline asm usage in user target causes a compile-time error.
- Any emitted vector directives in user target causes a compile-time error.
- Any constant absolute pointer outside `0x0100..0xFDFF` causes a compile-time error.

6.12 Deliverables

1. **Compiler executable** (`lc4c`) supporting:
   - `-target=kernel` and `-target=user`,
   - v5-aligned validation and codegen rules.
2. **User crt0 object** (`ucrt0.o`) and minimal **user library** (`ulib.o`) providing syscall wrappers:
   - `putc()`, `exit(code)`, optional `yield()` if defined in the syscall table.
3. **Kernel library** (`klib.o`) with BootInfo parsing helpers (no hidden init).
4. **Test suite** covering:
   - dual-image boot path,
   - user syscall correctness,
   - compile-time restriction enforcement,
   - protection fault behavior (vector `0x03`).

6.13 Definition of Done (LCv4)

Phase 4 is complete when:

1. The project builds **two images** (`kernel.bin` and `user.bin`) compatible with the v5 toolchain plan.
2. Kernel boots, validates BootInfo user entry/stack, transitions to user mode (`K:=0`), and transfers control to user code.
3. User code cannot (by compiler checks) include privileged constructs and can only request services via syscalls.
4. Protection violations from user code reliably trigger the protection handler (vector `0x03`) if they occur at runtime.
5. All Phase 4 tests pass.

6.14 Known Risks and Guardrails

- **Range rule mismatch:** keep `0x0100..0xFDFF` as the user-accessible window and reserve `0x0000..0x00FF` + `0xFE00..0xFFFF`.

- **Ownership conflicts:** vectors/reset remain owned by kernel stub/boot; the compiler must not emit them for kernel/user by default.
- **Over-permissive user pointers:** if strict safety is desired, enable `-Werror=unsafe-user-ptr` and treat unknown pointer arithmetic as an error.
- **ABI stability:** do not change Phase 3 ABI; kernel/user interop and libraries depend on it.