Multi-Target Compilation in Lateralus

RISC-V, x86-64, WebAssembly, and the C99 transpiler backend

Lateralus Language

bad-antics · February 2025 · Lateralus Language Research

ABSTRACT Lateralus targets four output formats: native RISC-V machine code (the primary target for Lateralus OS), native x86-64 machine code, WebAssembly for browser and WASI deployments, and C99 transpilation for maximum portability. All four backends share a common intermediate representation (Lateralus IR) and differ only in code generation. This paper describes the backend architecture, the IR design choices that make multi-target compilation practical, and the performance characteristics of each target.

1. Why Four Targets

A language that targets only one machine architecture limits its deployment contexts. Lateralus is designed to run on Lateralus OS (RISC-V), on standard Linux/macOS workstations (x86-64), in the browser via WebAssembly, and on any platform with a C99 compiler via transpilation.

The four targets are not equally important: RISC-V is the primary target for embedded and OS work; x86-64 is the primary target for developer tooling and servers; WASM is the target for browser-side Lateralus programs; and C99 is the escape hatch for exotic platforms and for embedding Lateralus logic in C codebases.

2. The Common Intermediate Representation

Lateralus IR (LIR) is a typed, SSA-form representation used by all four backends. LIR's design is driven by the requirement that all backends must be able to produce equivalent code without platform-specific knowledge in the front-end or optimizer:

- No architecture-specific instructions: LIR uses abstract operations (add, load, store) that map to different instructions on each target.
- Type information is preserved: LIR values carry their Lateralus type, enabling backend-specific type-directed optimizations (e.g., RISC-V can use FPU instructions for f64 operations).
- Pipeline nodes are preserved: the optimizer sees pipeline IR nodes and can apply pipeline-specific rewrites before lowering to flat LIR.

```
// Example LIR snippet (simplified)
function add_and_double(x: i32, y: i32) -> i32:
    %sum  = add i32 %x, %y
    %two  = const i32 2
    %result = mul i32 %sum, %two
    ret i32 %result
```

3. RISC-V Backend

The RISC-V backend is the most mature and the reference implementation against which the other backends are compared. It targets the RV64GC ISA (64-bit, general, compressed instruction extension) and uses the RISC-V calling convention.

```
// x86-64 → RISC-V instruction mapping for the add_and_double example
add_and_double:
    add  a0, a0, a1    # sum = x + y
    slli a0, a0, 1     # result = sum * 2 (shift left = multiply by 2)
    ret
```

The RISC-V backend uses a linear-scan register allocator with the 16 integer and 16 floating-point registers. Spills go to the stack frame. The calling convention matches the RISC-V psABI for compatibility with C libraries.

3.1 Compressed Instructions

The compressed extension (C) reduces code size by 20-30% for typical workloads by encoding common 32-bit instructions in 16 bits. The backend emits compressed instructions for: c.addi, c.li, c.lw, c.sw, and the compressed branch and jump forms.

4. x86-64 Backend

The x86-64 backend targets the System V AMD64 ABI on Linux/macOS and the Microsoft x64 ABI on Windows. Both ABIs are supported; the choice is a compilation flag.

The x86-64 backend uses a graph-coloring register allocator over the 16 general-purpose registers (minus RSP, RBP, and the two scratch registers reserved for LIR lowering). SIMD instructions are emitted for vectorizable iterator pipelines (e.g., a map(|x| x * 2) over a &[f64] becomes an AVX2 vector multiply).

```
// x86-64 output for add_and_double (AT&T syntax)
add_and_double:
    leal  (%rdi,%rsi), %eax   # sum = x + y
    addl  %eax, %eax          # result = sum * 2
    ret
```

5. WebAssembly Backend

The WASM backend emits binary WebAssembly (not WAST text format). LIR's stack-based value-passing maps naturally to WASM's stack machine; the main translation challenge is call convention differences and memory model.

Lateralus's ownership model simplifies WASM compilation: owned values are stack-allocated when possible (no heap, no GC), and borrows become pointers into linear memory. The WASM backend uses the 32-bit address space of WASM linear memory; i64 values are split or use the i64 WASM type directly.

5.1 WASI Integration

WASM System Interface (WASI) provides standard I/O and filesystem access. The std::io module is compiled to WASI calls when targeting WASM. Programs that use only std::io for I/O are fully portable between native and WASM targets.

6. C99 Transpiler Backend

The C99 transpiler translates LIR to portable C99 code. The output is not pretty-printed for readability; it is optimized for correctness and compilability on any C99-compliant compiler.

```
/* C99 output for add_and_double */
#include <stdint.h>

static int32_t add_and_double(int32_t x, int32_t y) {
    int32_t sum    = x + y;
    int32_t result = sum * 2;
    return result;
}
```

The C99 backend is useful for three scenarios: targeting platforms without a native Lateralus backend (ARM, MIPS, PowerPC), embedding Lateralus functions in a larger C codebase, and as a debugging aid when the native backend produces incorrect code.

6.1 C99 Limitations

The C99 backend cannot express: variable-length arrays of non-trivial types (uses heap allocation instead), generic functions (monomorphizes to C function variants with mangled names), and async pipelines (requires a C setjmp-based coroutine emulation).

7. Backend Performance Comparison

We compiled a representative benchmark suite (10 algorithms: quicksort, matrix multiply, JSON parser, HTTP request handler, SHA-256, AES-128, Fibonacci, prime sieve, regex match, and a 5-stage data pipeline) and measured throughput relative to a hand-written C implementation.

```
Backend     Throughput vs C  Compile time  Binary size
------------------------------------------------------------
RISC-V            92%           0.8 s/KLOC     1.0×
x86-64            95%           0.6 s/KLOC     1.1×
WASM              78%           0.9 s/KLOC     1.2×
C99 (via gcc)     91%           1.4 s/KLOC     1.05×
```

The x86-64 backend comes closest to C performance, primarily due to AVX2 vectorization. The WASM deficit is from linear-memory indirection for heap objects. The C99 path via GCC is slower to compile but produces comparable output to the native backend.

8. Cross-Compilation

Cross-compilation (building for a different target than the host) is supported for all four backends. The compiler is itself a Lateralus binary; building a Lateralus compiler that targets RISC-V on an x86-64 host requires only the RISC-V standard library headers and a RISC-V sysroot.

```
# Cross-compile to RISC-V from x86-64
ltl build --target riscv64-linux-gnu my_program.lt

# Cross-compile to WASM
ltl build --target wasm32-wasi my_program.lt
```

The build system automates sysroot discovery for common target triples. Custom targets can be specified with a JSON target descriptor that lists the ABI, word size, and available extensions.