Lateralus Bytecode Format (LBC)

Register-based bytecode for the Lateralus interpreter and JIT

Lateralus Language

bad-antics · April 2025 · Lateralus Language Research

ABSTRACT The Lateralus Bytecode (LBC) format is a compact, register-based bytecode used by the Lateralus interpreter and as the input format for the JIT compiler. LBC is designed for fast decoding (4-byte fixed-width instructions), compact constant tables (inline 16-bit literals, out-of-line 64-bit constants), and type-tagged values (3-bit tag allows unboxed representation for common types). This paper specifies the instruction set, the file format, and the encoding decisions that balance code density against decoding speed.

1. Design Goals

LBC serves two consumers: the tree-walking interpreter (which needs fast dispatch) and the JIT compiler (which needs type information at each instruction). The format must satisfy both.

- **Fixed-width instructions**: 4 bytes per instruction enables O(1) jump targets (no bytecode index scan) and predictable fetch cost.
- **Register-based (not stack-based)**: register machines require fewer instructions to express common patterns; stack machines require more push/pop traffic that obscures data flow.
- **Type tags**: each register value carries a 3-bit type tag that the JIT uses for specialization without requiring a full type inference pass at JIT time.
- **Pipeline-aware opcodes**: LBC includes opcodes for pipeline stage dispatch and error short-circuit that map directly to the four pipeline operator semantics.

2. Instruction Format

Each LBC instruction is 32 bits encoded as:

```
31      24 23    16 15     8 7      0
+----------+--------+---------+--------+
|  opcode  |  dst   |   src1  |  src2  |
+----------+--------+---------+--------+
  8 bits     8 bits    8 bits    8 bits

For instructions with an immediate value:
31      24 23    16 15               0
+----------+--------+-----------------+
|  opcode  |  dst   |  imm16 (signed) |
+----------+--------+-----------------+
```

The 8-bit register fields allow 256 registers per function. Functions that need more than 256 temporaries are split by the compiler. The 16-bit immediate covers all common small constants; large constants use a LOAD_CONST opcode that indexes a 64-entry constant table.

3. Opcode Table

The opcode space is 256 values, organized into groups:

```
0x00-0x0F:  Arithmetic   ADD, SUB, MUL, DIV, MOD, NEG, ABS, ...
0x10-0x1F:  Logic        AND, OR, XOR, NOT, SHL, SHR, SAR
0x20-0x2F:  Compare      EQ, NE, LT, LE, GT, GE, IS_NULL
0x30-0x3F:  Control      JMP, JT, JF, CALL, RET, TAIL_CALL
0x40-0x4F:  Memory       LOAD, STORE, ALLOC, FREE, MEMCPY
0x50-0x5F:  Constant     LOAD_CONST, LOAD_INT16, LOAD_F64, LOAD_STR
```

```
0x60-0x6F:  Pipeline     PIPE_CALL, PIPE_ERR_CHECK, PIPE_FANOUT, PIPE_AWAIT
0x70-0x7F:  Type         TAG_CHECK, CAST, TYPEOF, IS_OK, IS_ERR
0x80-0xFF:  Reserved
```

The Pipeline group (0x60-0x6F) encodes the four operator semantics directly. PIPE_CALL is a regular function call in the context of a total pipeline stage. PIPE_ERR_CHECK tests the result tag and jumps to the error exit block if it is Err. PIPE_FANOUT dispatches one input to N parallel stage functions.

4. Value Representation

LBC values are 64-bit NaN-boxed: floating-point values use the full IEEE 754 double-precision range; non-float values are encoded in the NaN payload with a 3-bit tag:

```
Tag  Meaning         Encoding
-----------------------------------------
000  f64             IEEE 754 double (non-NaN)
001  i64             lower 61 bits sign-extended
010  bool            bit 0 = 0/1 (false/true)
011  null / unit     all-zero payload
100  heap pointer    lower 48 bits = address
101  Result::Ok      lower 48 bits = payload ptr
110  Result::Err     lower 48 bits = error ptr
111  reserved
```

The NaN-boxing scheme allows the common integer and boolean operations to be performed without unboxing: an ADD of two i64 values strips the tag, adds, and re-tags in three instructions. The tag field is read by the JIT for type specialization without a separate type-inference phase.

5. The Constant Table

Each function has a private constant table of up to 64 entries. Constants are 64-bit values: integers, floats, or pointers to string literals in the string pool. The constant table is emitted in the LBC file immediately before the instruction stream of its function.

```
// Function constant table layout
u16 count          // number of constants (max 64)
u64 constants[N]   // constant values, NaN-boxed
```

The LOAD_CONST instruction takes a 5-bit constant index (0-63) and loads the constant into the destination register. For constants outside this range, the compiler emits multiple instructions to construct the value.

6. LBC File Format

An LBC file starts with a header, followed by the string pool, followed by a list of function records:

```
// LBC file layout
magic:       u32   = 0x4C424300  ('LBC\0')
version:     u16   = 1
flags:       u16   = 0           (reserved)
n_functions: u32
string_pool_size: u32
string_pool: [u8; string_pool_size]
functions:   [FunctionRecord; n_functions]

// FunctionRecord layout
name_offset:  u32   // offset into string pool
```

```
n_regs:       u8    // number of registers
n_params:     u8    // number of parameters
const_count:  u8    // number of constants
reserved:     u8
constants:    [u64; const_count]
n_instr:      u32
instructions: [u32; n_instr]
```

The format is intentionally simple to make the interpreter's startup fast: a single sequential read loads the entire file; no seeking is required.

7. Interpreter Dispatch Loop

The interpreter uses a computed-goto dispatch loop (in C99 via GCC labels-as-values extension, and via a switch statement on MSVC). Each opcode handler is a label; dispatch is a single indirect jump from the opcode byte.

```
// Computed-goto dispatch (conceptual)
void *dispatch_table[256] = {
    [OP_ADD] = &&op_add,
    [OP_SUB] = &&op_sub,
    // ...
};

#define DISPATCH() \
    ip += 4; \
    goto *dispatch_table[*ip];

op_add:
    regs[DST] = i64_add(regs[SRC1], regs[SRC2]);
    DISPATCH();
```

The dispatch loop processes approximately 400 million instructions per second on a 2024-era laptop for integer-heavy workloads. Float and memory operations are 30-40% slower due to hardware latency.

8. Relation to LBC v2 Plans

LBC v1 (described here) is a stable format for the Lateralus 1.x series. LBC v2 (planned for Lateralus 2.0) will add: a 3-address instruction form with 5-bit register fields for functions with fewer than 32 registers (enables 2-byte instructions for common patterns), a structured exception table for the |?> operator (currently encoded as conditional branches), and a debug information section (source maps, variable names, type information for debuggers).

LBC v1 and v2 will be distinguished by the version field in the file header; the interpreter will support both versions in parallel during the transition period.