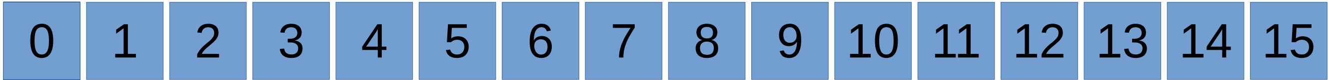
Vector lane ordering
Or: How to implement raw_bitcast on a big-endian architecture

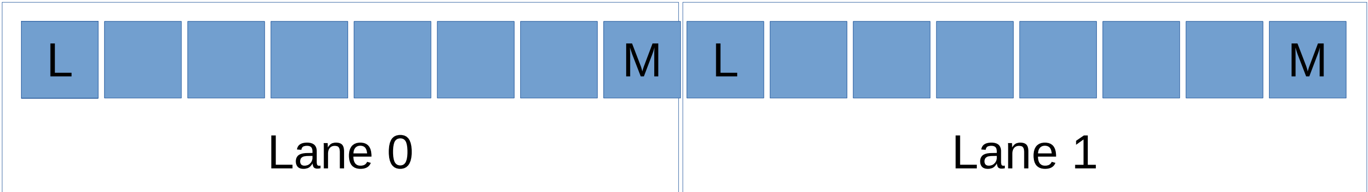
Part 1: Clarifying raw_bitcast semantics
Part 2: Efficient implementation on big-endian machines

Rust / C / C++ SIMD vectors – little-endian architecture

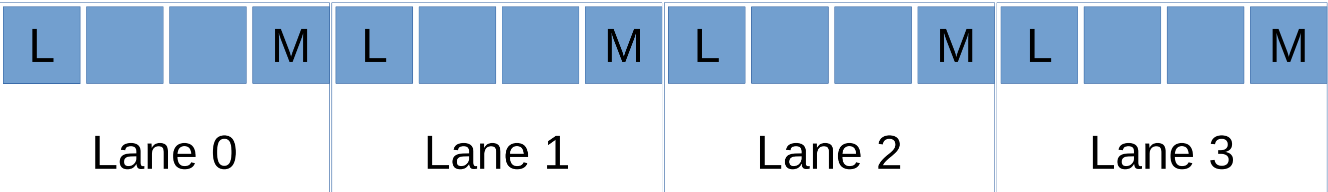
Memory address



I64X2 in memory



I32X4 in memory



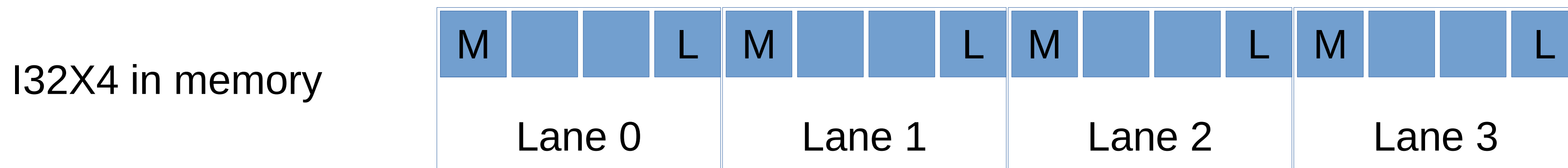
raw_bitcast:
lane 0 of smaller type is **least-**significant part of lane 0 of larger type
→ LE lane order

Rust / C / C++ SIMD vectors – big-endian architecture

Memory address

I64X2 in memory

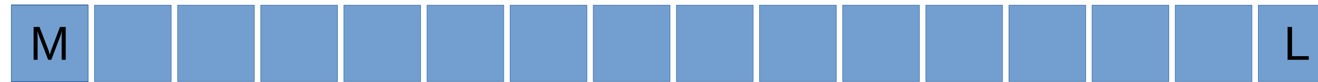
I32X4 in memory



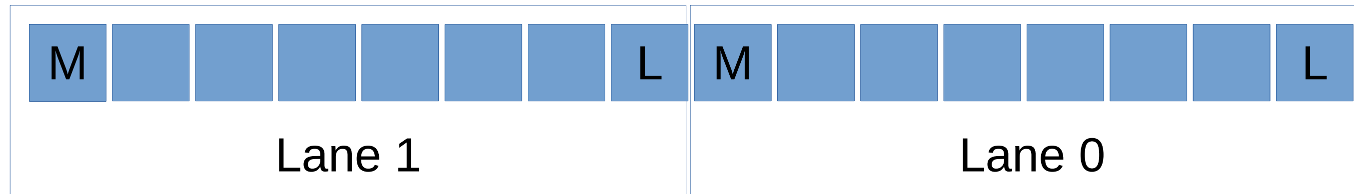
raw_bitcast:
lane 0 of smaller type is **most-** significant part of lane 0 of larger type
→ BE lane order

WebAssembly vector types – defined as values

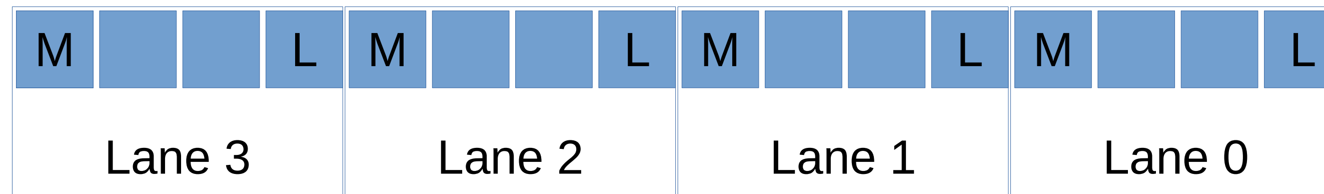
V128 type



V128 as I64X2



V128 as I32X4



raw_bitcast:
lane 0 of smaller type is **least**-significant part of lane 0 of larger type
→ LE lane order

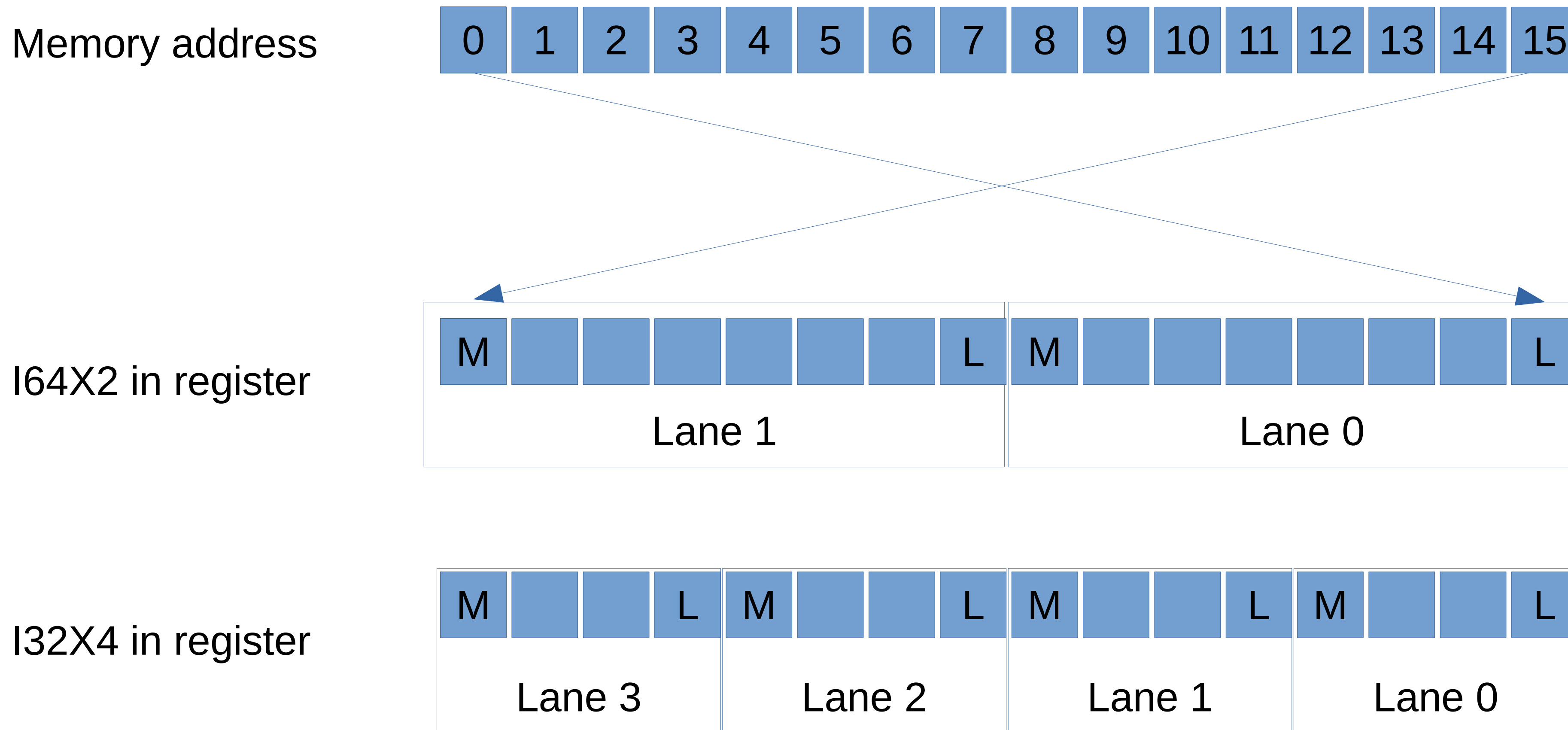
Summary of raw_bitcast semantics

- The raw_bitcast operator is underspecified, it actually exists in two flavors:
  - "Little-endian lane order":
    Lane 0 of smaller type is **least**-significant part of lane 0 of larger type
  - "Big-endian lane order":
    Lane 0 of smaller type is **most**-significant part of lane 0 of larger type

- Current usage of raw_bitcast
  - On a little-endian machine, all uses of raw_bitcast use LE lane order
  - On all machines, raw_bitcast emitted from wasmtime uses LE lane order
  - On a big-endian machine, raw_bitcast emitted from cg_clif uses BE lane order

- Options to define semantics
  - Impliclit (back-end treats functions emitted from wasmtime differently)
  - Treat as memory operation (add MemFlags including endianness options)
  - Separate opcodes, e.g. raw_bitcast_le and raw_bitcast_be
    - Should we have an implicit "native" variant?

Implementation on little-endian machines



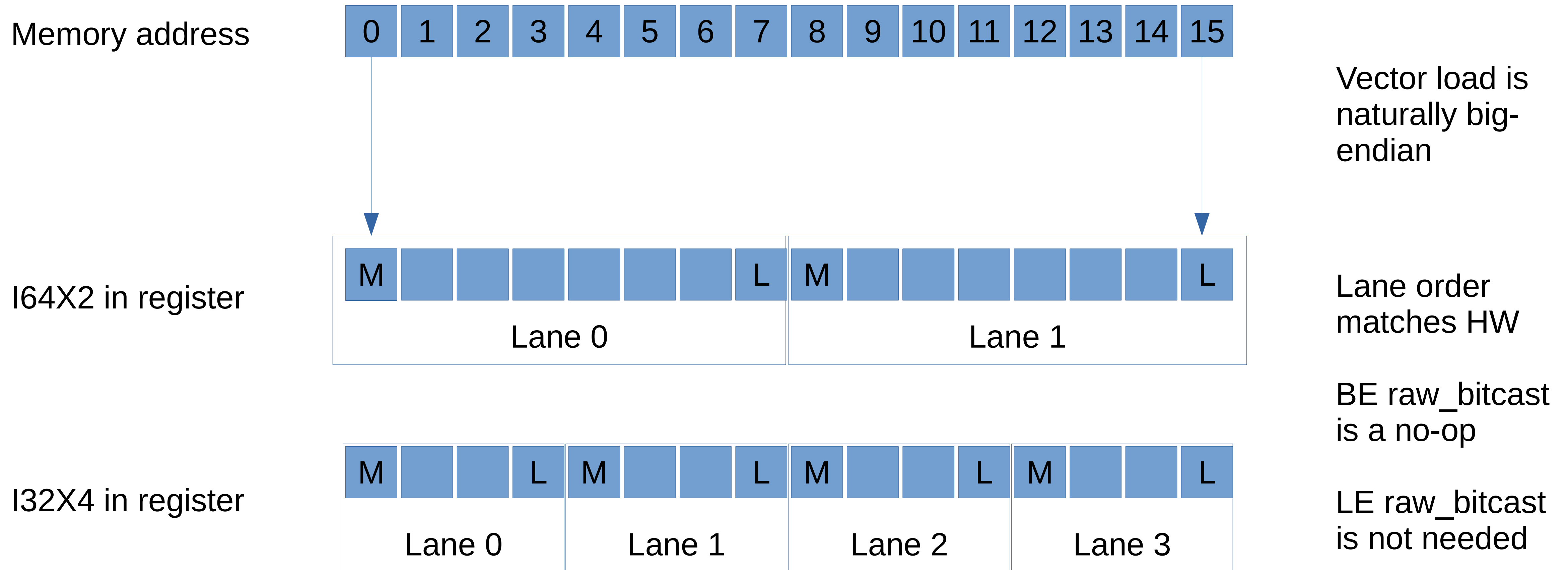
Vector load is naturally little-endian

Lane order matches HW

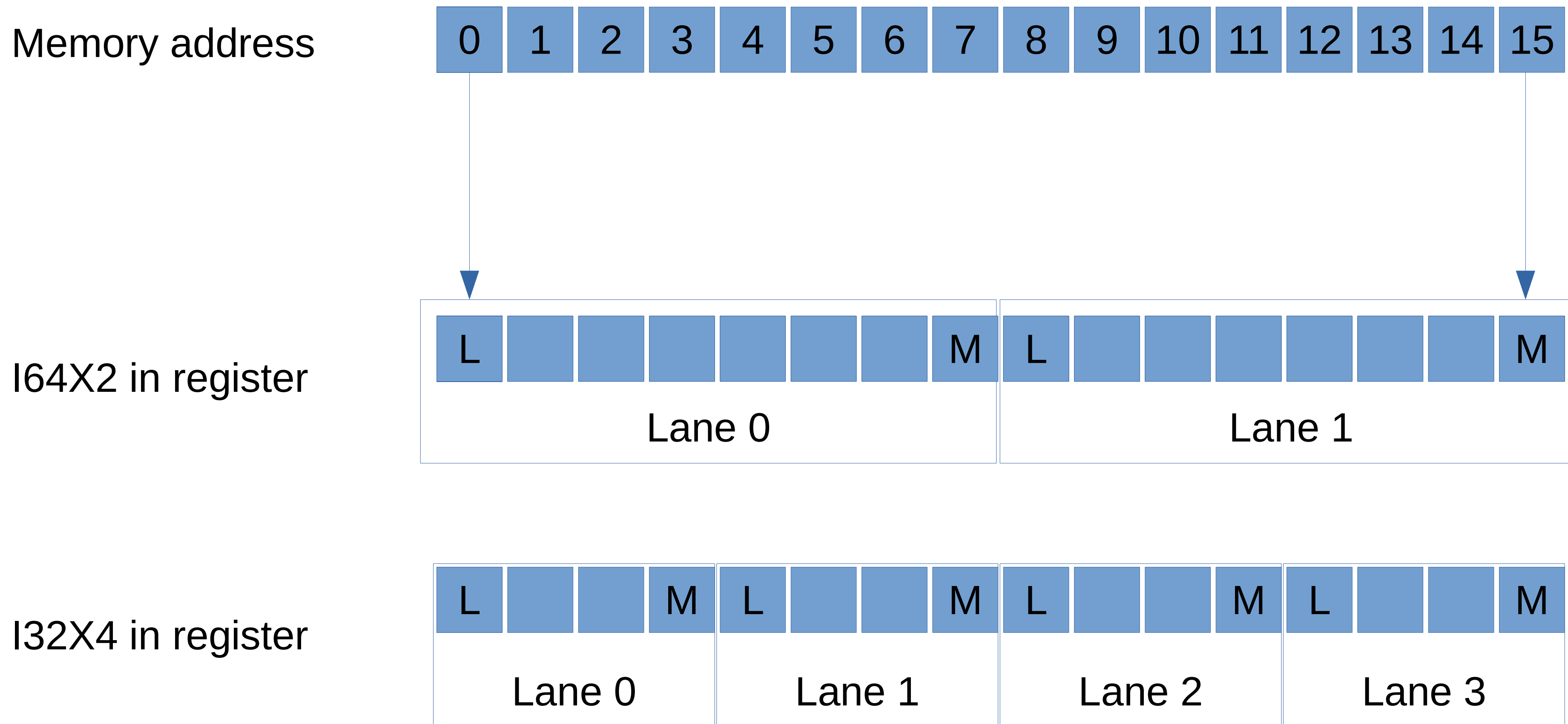
LE raw_bitcast is a no-op

BE raw_bitcast is not needed

Implementation on big-endian machines – default (e.g. cg_clif)



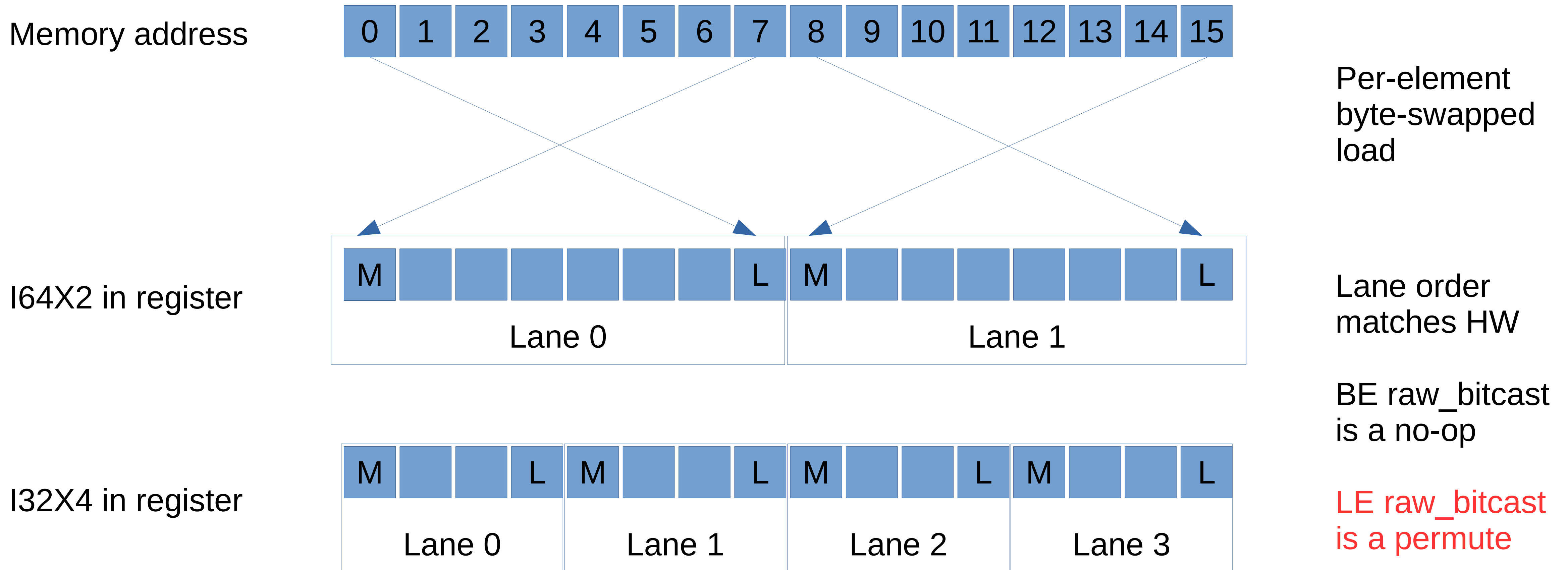
Implementation on big-endian machines – little endian memory



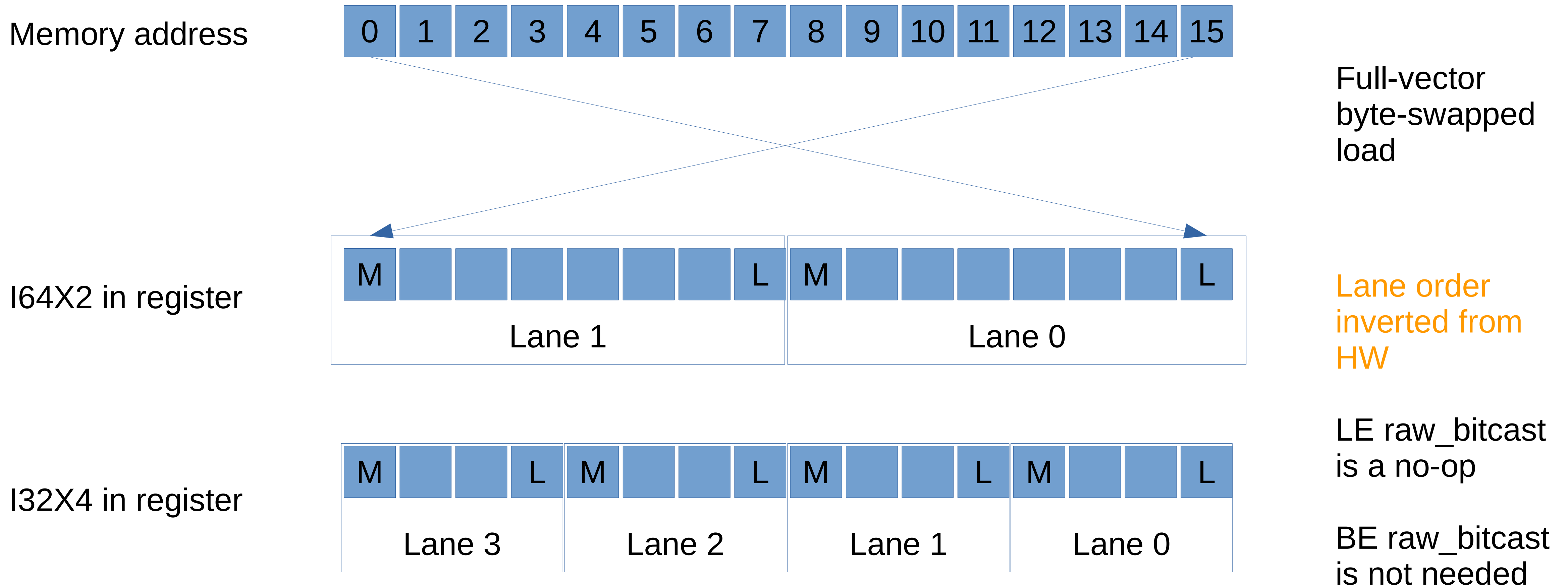
Vector load is naturally big-endian

Bytes in each lane are in wrong order – register not usable for arithmetic operations!

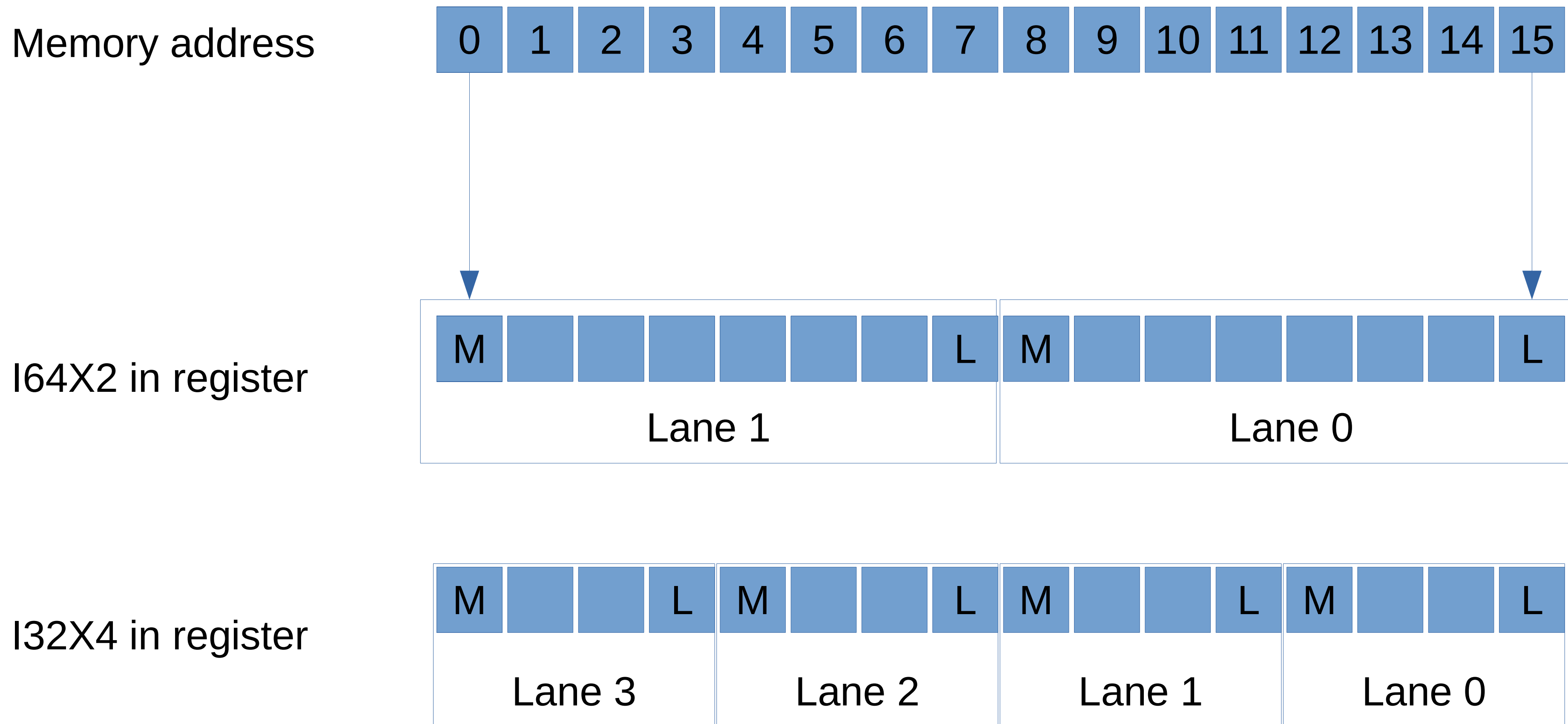
Implementation on big-endian machines – little endian memory



Implementation on big-endian machines – LE memory, inverted



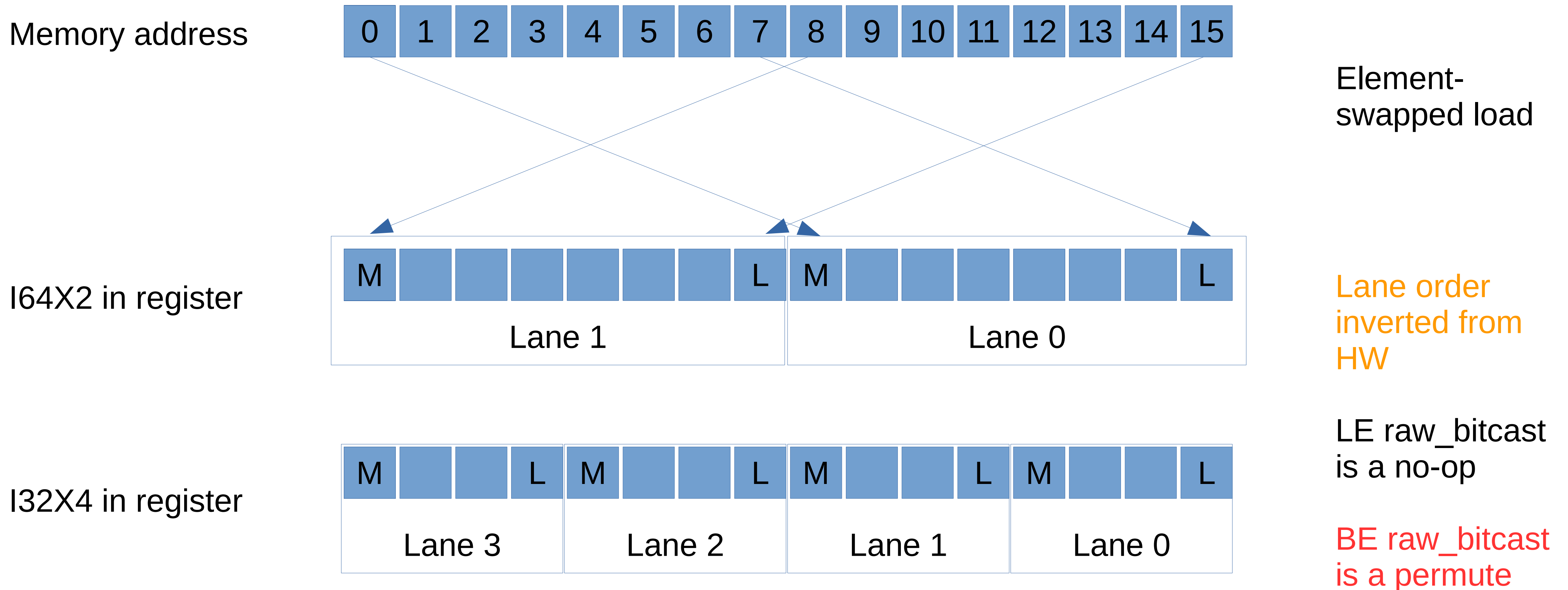
Implementation on big-endian machines – BE memory, inverted



Vector load is naturally big-endian

Lane order does not match source semantics – not usable!

Implementation on big-endian machines – BE memory, inverted



Summary of implementation options

- The back-end has the choice between two options
  - In-register lane order matches HW order (BE lane order)
  - In-register lane order inverted from HW order (LE lane order)

- Impact of lane order choice on visible semantics
  - Either implementation option can fully implement CLIF semantics
  - In particular, either option can implement both LE and BE raw_bitcast
  - Implementation option potentially visible in the ABI (vector argument/return regs)
    - SystemV ABI requires BE lane order, Wasmtime ABI free to define
    - Can be made transparent via lane swaps at ABI boundaries
    - More efficient to choose one defined lane order per ABI

- Impact of lane order choice on implementation
  - Either option can be fully implemented, including both LE/BE memory load/store
  - Affected instructions: memory ops, explicit lane number ops, raw_bitcast
  - Efficiency: raw_bitcast is no-op if and only if requested lane order matches implementation lane order, permute (element swap) otherwise

Proposed solution

- Choose in-register lane order based on the current function's ABI
  - LE lane order if function using Wasmtime ABI
  - BE lane order otherwise

- Effect
  - Fully implemented CLIF semantics, including LE and BE raw_bitcast
  - No element swaps needed at ABI boundaries
  - Efficient implementation of all variants of memory ops & explicit lane order ops
  - raw_bitcast implementation:
    - LE raw_bitcast in Wasmtime ABI functions is no-op
    - BE raw_bitcast in functions using other ABI is no-op
    - Result: **every** raw_bitcast used by **all current front ends** is no-op!

- Staged implementation
  - Phase 1: Back-end only, assuming every raw_bitcast is a no-op
  - Phase 2: Implement explicit raw_bitcast semantics via CLIF extension