Ragionato per un paio di secondi

Z-Ant IR & Zig Renderer – Architecture Deep Dive

(A practical orientation for anyone who will add new UOp-lowerings or renderer back-ends)

1. Big-Picture Flow

```
ONNX  ──▶  High-level Z-Ant passes  ──▶  UOpBuilder(slice)  ──▶  Zig renderer  ──▶
           (operator lowering)             (SSA micro-IR)         (pretty prints)
```

Stage	Responsibility	Key files
Lower Xxx passes	Break one ONNX node into a handful of *UOps* (lowering).	`lower_add.zig` , `lower_reduce_mean.zig` , …
UOpBuilder	Owns the growing SSA slice. Guarantees *single assignment*. Dupes all owned memory.	`ir.zig` section 4
Renderer (Zig)	Walks the slice once and prints valid Zig code that performs the same scalar work.	`zig_renderer.zig` + renderers per opcode
Runtime	We just `zig build` the produced code and call `generated_kernel()` from tests.	unit tests

2. The IR: `UOp` , `UOpType` , `Any` , `DType`

2.1 `UOpType`

A closed enum of *all* primitive operations that any backend must understand. *They are intentionally tiny* – think of them as the “assembly” of tensor code.

```
DEFINE_GLOBAL   // top-level tensor
VIEW            // alias + broadcast info
GEP             // pointer arith (GetElementPtr)
LOAD / STORE    // scalar mem-ops
RANGE / ENDRANGE// counted loops
ADD, MUL, NEG…  // scalar ALU
```

```
MULACC          // fused FMA into DEFINE_ACC
...
```

2.2 `UOp`

```
pub const UOp = struct{
    id   : usize,          // index in the slice (SSA id == pos)
    op   : UOpType,        // opcode tag
    dtype: DType,          // element type of *result*
    src  : []const usize,  // producer ids (already defined ops)
    arg  : ?Any,           // optional payload
};
```

- **SSA guarantee**: every ID is defined **once**; later ops only read it.
- `src` is always duplicated by the builder → you own a private slice.

2.3 `Any` – payload

Single-slot tagged union that carries the odd bits of metadata:

Variant	Used by	Example
`int` / `float`	`CONST`	literal “5”, “0.5f”
`loop_bounds`	`RANGE`	`{start=0,end=128}`
`mem_info`	`GEP`	`{ base=<id>, offset, stride }`
`view_meta`	`VIEW`	`{ shape=[2,3], strides=[3,1] }`

Add more when a new primitive needs it.

3. Lowering helpers – `UOpBuilder`

```
const id = b.push(.ADD, .f32, &.{lhs, rhs}, null);
```

- `push()` duplicates **both** `src` and (if needed) the slices stored inside `Any.view_meta` .
- `toOwnedSlice()` hands ownership of the finished program to the caller.
- `deinit()` frees *everything* (each `src` , each duplicated `shape/stride`).

> **Rule of thumb**: *If you allocate memory inside `Any` , duplicate it before storing and free it in `deinit()` .*

4. Renderer architecture

```
zig_renderer.zig
 ├── identify_buffers()   // builds BufferInfo map
 ├── render_as_function() // top-level entry
 │     1. emits Zig fn signature
 │     2. allocates buffers (DEFINE_GLOBAL)
 │     3. creates ptr_map (id → var name)
 │     4. walks slice -> render_uop()
 │     5. returns output slice
 └── render_uop()         // one big switch
        ├─ MemoryRender.render()   // LOAD / STORE / CONST / DEFINE_GLOBAL
        ├─ GepRender.render()      // pointer math
        ├─ ArithmeticRender.render() // ADD/MUL/…
        ├─ UnaryRender.render()    // NEG/CAST/EXP2…
        ├─ ControlFlowRender       // RANGE / ENDRANGE
        ├─ manage VIEW meta        // adds to view_map, no code
        └─ DEFINE_ACC / MULACC     // inline helpers
```

4.1 Maps kept during rendering

Name	Type	Purpose
`buffer_map`	`HashMap(usize, BufferInfo)`	semantic info for each DEFINE_GLOBAL (shape, name, is_input)
`view_map`	`HashMap(usize, ViewInfo)`	stores `{shape,strides}` for each VIEW id
`ptr_map`	`HashMap(usize, []const u8)`	final Zig *identifier* (variable name) that holds the value/pointer for an SSA id
`rendered_ids`	`HashSet(usize)`	ensures we print each op once

4.2 Naming convention (auto-generated)

Prefix	What it is
`input_`	function parameter slice
`output_`	final output buffer slice
`addr_`	**usize** holding a calculated address (`GEP`)
`buf_`	scalar temporaries (`ADD` result, etc.)

Prefix	What it is
`idx_`	loop induction variable (`RANGE`)
`acc_`	accumulator (DEFINE_ACC)
`view_`	alias id (no code emitted)

5. `GepRender` – deep dive

Goals: turn a high-level `GEP` into

```
const addr_7 = @intFromPtr(base.ptr) + (offset_expr) * @sizeOf(f32);
```

Steps:

1. **Base pointer selection**
    - If `base_id` is a DEFINE_GLOBAL slice that is `input_*` or `output_*` → need `.ptr` because slices are `{ ptr, len }` .
    - If it is an internal pointer (`addr_*` , `acc_*`) → already a raw pointer.
2. **Offset expression** *If there is a VIEW* we respect its per-axis `stride` and optional broadcast (stride == 0). *1-D index form*: supports rank-1 and rank-2 by unflattening. *Full index form*: just ∑ (idx × stride) skipping broadcast axes. *Raw buffer (no VIEW)* assumes plain row-major layout (rank-1 or rank-2).
3. **Emit final line** – multiply offset by `@sizeOf(dtype)` .

Helpers:

```
castIndex()           // "idx_i32" → "@as(usize,@intCast(idx_i32))"
emitTerm()            // prints "+ (expr*stride)" skipping stride==0
ArenaAllocator        // every temporary string is arena-allocated ⇒ freed at end
```

6. `MemoryRender` – LOAD / STORE / CONST

- **CONST** → `var buf_4: f32 = 0.5; _ = &buf_4;`
- **LOAD** → read through the calculated pointer, result bound to `var buf_9` .
- **STORE** → write scalar value *directly* via pointer cast.

All three look up variable names in `ptr_map`.

7. Control-flow (`RANGE` / `ENDRANGE`)

```
var idx_3: i32 = 0; // RANGE
while (idx_3 < end) : (idx_3 += 1) {
    ...
}                    // ENDRANGE
```

Indentation depth is tracked in `loop_indent` so nested loops indent correctly.

8. Adding a new lowering pass

1. **In `LowerXxx.zig`**
    - Calculate output `shape`, `strides`, etc.
    - Emit VIEWs / GEPs / ALU UOps through `UOpBuilder`.
2. **Unit-test**
    - Dump the slice (`uop.dump`) to make sure the sequence is valid.
3. **Rendering**
    - If your new op re-uses existing primitives (often the case), no renderer work is needed.
    - Otherwise add a small renderer file: follow the pattern of `arithmetic_render.zig`.
4. **Update `render_uop` switch** to call your renderer.

9. Adding a brand-new primitive

1. Extend `UOpType`, add a case.
2. Decide if it needs metadata → add a field to the `Any` union.
3. Teach `UOpBuilder.push()` to duplicate that field when it appears.
4. Write a renderer and hook it in `render_uop`.
5. Write one minimal lowering + test so CI exercises it.

10. Memory ownership rules

Who allocates	What	Who frees
Lowering pass	slices in `Any`	**UOpBuilder.deinit**
Renderer	`acc_*`, `view_*` names via `allocPrint`	Freed at end of `render_as_function` (only those prefixes)
`identify_buffers`	`BufferInfo.name`, `.shape`	Freed in `ZigRenderer.deinit`

Never store a pointer into caller-owned memory inside a `UOp` without duplicating it first.

11. Common gotchas

- Forgetting to append `.ptr` for slice parameters ⇒ ‘expected pointer, found []const f32’.
- VIEW rank mismatch: `uop.src.len-1` must equal `view_meta.strides.len` in the multi-index form.
- Leaked arena strings: only allocate with the dedicated arena inside the renderer; it is `deinit()` ed every GEP invocation.

12. Checklist for new contributors

1. **Lowering** produces valid SSA? run `dump()` and eyeball ordering.
2. **Slice safety**: every `shape[]`, `strides[]` duplicated inside Any?
3. **Renderer**: variable names go through `ptr_map`, never hard-code id → name.
4. **Unit tests**:
   - `zig build test` passes without “leaked” or “expected pointer” errors.
   - Dumped Zig shows correct address arithmetic.
5. **Documentation**: update this file if you add new primitives.

> With these conventions in mind, extending Z-Ant either on the *front* (lowerings) or the *back* (renderers) should be a predictable and safe process. Happy hacking!