LLM OUTPUT

1) Root causes (summary)

- **ADD / SUB swapped** — `add_sub` (or the way the ALU passes inst[5]) used the inverted control / the mux in ALU was reversed. Result: when inst[5]==1 you got the ADD path and when inst[5]==0 you got SUB (or vice versa).

- **Register C never loads when LD[3] asserted** — the load decoder wiring in `micro.v` had an index mismatch: the LD decoder outputs were connected to registers in the wrong order (off-by-one / bit order reversed) so asserting LD[3] did not address register C as expected.

- **OUT A sometimes `zzzzzzzz`** — tri-state enable polarity or the OE wiring had glitches so the tri-state buffer was not being driven at the time the testbench sampled the bus (or multiple drivers/none were enabled briefly). This was made worse by **blocking assignments** used inside the register DFFs (or other register implementation). Blocking assignments can cause intra-clock race: combinational decode logic and tri-state enables might see old/new register values depending on statement scheduling.

- **Race on "ADD then immediate OUT"** — caused by a combination of (a) register updates implemented with blocking assignments (so the apparent ordering of register update vs. other combinational logic depended on source order inside always blocks), and (b) tri-state enable or bus-driver evaluation happening at nearly the same time so the bus could be driven with the *old* A or be briefly undriven (Z). The canonical fix: use **non-blocking (`<=`)** for synchronous register updates and make tri-state enables stable (and correctly polarized). This prevents update-order dependent races: registers update at the same simulation time but in a stable non-blocking semantics, and combinational logic sees the new registered outputs after the edge (no intra-clock blocking ordering ambiguity).

---

2) Exact Verilog edits (patches)

Below are minimal edits to the files you mentioned. I give full replacement snippets per file so you can copy/paste. I annotate why each change is made.

> **Note:** filenames used as in your context (e.g., `rtl/dff.v`, `rtl/add_sub.v`, `rtl/ALU.v`, `rtl/tribuff_8_2to1.v`, `rtl/micro.v`). If your repo uses slightly different names, apply the same changes.

2.1 `rtl/dff.v` — use non-blocking assignments for register updates

Problem: DFF used blocking `=` which causes intra-clock races with combinational logic reading those regs during the same time-step.
Fix: Use non-blocking `<=`. Provide parameterizable width, synchronous reset if needed.

Replace contents (or patch accordingly):

```
// rtl/dff.v
// 8-bit DFF updated to use non-blocking assignment to avoid race
conditions.

module dff #(
    parameter WIDTH = 8
)(
    input  wire             clk,
    input  wire             rst_n,    // active low reset (optional)
    input  wire             en,       // enable load when high (if
used)
    input  wire [WIDTH-1:0] d,
    output reg  [WIDTH-1:0] q
);

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        q <= {WIDTH{1'b0}};
    end else begin
        if (en)
            q <= d;    // non-blocking assignment
        else
            q <= q;    // keep (non-blocking)
    end
end

endmodule
```

Why: `<=` ensures all clocked registers take their new values simultaneously without ordering dependence on statement order. This removes subtle races seen when a combinational piece (like tri-state enable or ALU input mux) depends on register outputs updated at the same rising edge.

2.2 `rtl/add_sub.v` — ensure proper polarity of control bit

Problem: add/sub control was inverted (ADD and SUB swapped).
Fix: make it explicit: `sub`=1 => subtraction; `sub`=0 => addition.

Replace (or patch) file:

```
// rtl/add_sub.v
module add_sub (
    input  wire [7:0] a,
    input  wire [7:0] b,
    input  wire       sub,   // 0 -> add, 1 -> sub
    output wire [7:0] y,
    output wire       carry_or_borrow
);

wire [8:0] add_res;
wire [8:0] sub_res;

assign add_res = {1'b0, a} + {1'b0, b};
assign sub_res = {1'b0, a} - {1'b0, b};

assign y = sub ? sub_res[7:0] : add_res[7:0];
assign carry_or_borrow = sub ? sub_res[8] /*borrow flag*/ :
add_res[8] /*carry flag*/;

endmodule
```

Why: explicit control removes ambiguity. If earlier logic inverted `sub`, this corrects it. Ensure `ALU` passes inst[5] (or correct control bit) directly to `sub`.

2.3 `rtl/ALU.v` — ensure the ALU forwards the correct control bit into add_sub and that any ALU mux order is correct

Open `rtl/ALU.v` and find where add_sub is instantiated and fix wiring (example):

```
// rtl/ALU.v
module ALU (
    input  wire [7:0] a,
    input  wire [7:0] b,
    input  wire       inst_sub, // inst[5] from instruction;
0=ADD,1=SUB
    output wire [7:0] y,
    output wire       carry_or_borrow
);

// instantiate add_sub with correct polarity
add_sub u_addsub (
    .a(a),
    .b(b),
    .sub(inst_sub),
    .y(y),
    .carry_or_borrow(carry_or_borrow)
);

endmodule
```

If there was a mux after the add_sub that reversed outputs, fix the ordering so `sub==1` selects subtraction result.

Why: make sure inst[5] from instruction maps directly to `sub` input.

2.4 `rtl/tribuff_8_2to1.v` — correct enable polarity and implement stable tri-state driver

Problem: Tri-buffer enable may have been inverted or driven such that the driver is briefly disabled when expected to drive (leading to Z). Also possible multiple drivers at once. We'll make the module explicit: each driver has dedicated `oe` (active-high) and if `oe` true it drives the bus, else high-Z. Do not add internal steering that could produce multiple drivers — that's a top-level issue.

Replace file:

```
// rtl/tribuff_8_2to1.v
```

```
// Two 8-bit inputs can drive the same output; each input has its
own oe (active-high).
// Whoever asserts its oe drives the bus; otherwise high-Z.
// NOTE: Top level must ensure only one oe is asserted at a time.

module tribuff_8_2to1 (
    input  wire [7:0] in0,
    input  wire       oe0,    // active-high: drive when 1
    input  wire [7:0] in1,
    input  wire       oe1,    // active-high
    output wire [7:0] out
);

// If both oe0 and oe1 are 0 -> out = Z.
// If one is 1, drive that input.
// If both 1 -> conflict (undefined); add optional synthesis-time or
assert to catch this.

assign out = oe0 ? in0 :
             oe1 ? in1 :
             8'bz;

endmodule
```

Why: explicit active-high enables and clean priority (in case of both drivers asserted, priority given to in0) avoids the earlier inverted enable bug. It also centralizes the `Z` behavior so top-level can ensure only one `oe` asserted.

2.5 `rtl/micro.v` — fix LD decoder wiring; ensure OE (output enable) polarity/registration; add assertion to avoid multiple drivers

This is the most sensitive file. I will show the specific parts to edit:

2.5.1 Fix LD decoder → register mapping

Find the load decoder instantiation (something like `decode3to8 ld_dec(inst[5:3], ld);` or `ld = decode3to8(inst[5:3]);`) and verify how `ld` bits are connected to registers. If your register order is `A(0), B(1), C(2), D(3), E(4), F(5)` then `inst[5:3]` value 3 must assert `ld[2]` if you used reversed mapping, or `ld[?]`—the

symptom (C never updates when LD[3] asserted) implies that `ld[3]` was wired to C rather than `ld[2]`. The safe fix: connect load signals by index name rather than positional wiring.

Example replacement snippet for register instantiation:

```
// rtl/micro.v (fragment)
// assume decode3to8 outputs 'ld[7:0]' where ld[0] -> reg A, ld[1]
-> reg B, ld[2] -> reg C, etc.

wire [7:0] ld;
decode3to8 u_ld_dec (
    .in(inst[5:3]),
    .out(ld)
);

// Register instances: explicitly connect ld bits to each reg's
enable
// A register
dff #(8) regA (
    .clk(clk),
    .rst_n(rst_n),
    .en(ld[0]),         // load when ld[0] asserted
    .d(aluresult),      // or appropriate bus
    .q(A)
);

// B register
dff #(8) regB (
    .clk(clk),
    .rst_n(rst_n),
    .en(ld[1]),
    .d(<...>),
    .q(B)
);

// C register (this line fixed to use ld[2] not ld[3])
dff #(8) regC (
    .clk(clk),
    .rst_n(rst_n),
    .en(ld[2]),         // correct: C uses ld[2]
    .d(<...>),
    .q(C)
```

```
);

// D register
dff #(8) regD (
    .clk(clk),
    .rst_n(rst_n),
    .en(ld[3]),
    .d(<...>),
    .q(D)
);

// ... and so on for E,F
```

Why: explicit mapping by name avoids off-by-one/reversal mistakes.

2.5.2 Make OE/tri-state enables stable and assert single-driver property

Make output-enable signals active-high and, importantly, add a checked assert (synthesis-friendly $fatal / $error wrapped under `ifndef SYNTHESIS`) to catch multiple drivers in simulation. Optionally register the OE decode outputs (non-blocking) so they are stable right after clock edge if your micro's timing model expects the OE to be stable in the same cycle as register updates.

Example (fragment):

```
// rtl/micro.v (fragment)
wire [7:0] oe_dec; // oe_dec[i] active when register i will drive
bus (OUT X)
decode3to8 u_oe_dec (
    .in(inst[2:0]),
    .out(oe_dec)
);

// Hook tri-state: each register's driver enable is
oe_dec[reg_index]
wire oe_A = oe_dec[0];
wire oe_B = oe_dec[1];
// ...
wire oe_C = oe_dec[2];
// ...

// instantiate tri-state buffer between A/B/C banks and bus
```

```
// for this demo, if using single-driver-per-bus, use tribuff_8_2to1
or separate tri buffers
tribuff_8_2to1 u_triAB (
    .in0(A),
    .oe0(oe_A),
    .in1(B),
    .oe1(oe_B),
    .out(bus)
);

// for C/D etc. instantiate additional tri buffers or smarter bus
muxing
// (Follow your existing top-level wiring but ensure oe signals
mapped correctly)
```

Add simulation-time check (optional but recommended):

```
// guard only for simulation - catches multiple oe asserted
simultaneously driving same bus
`ifndef SYNTHESIS
always @(*) begin
    integer cnt;
    cnt = oe_A + oe_B + oe_C + oe_D + oe_E + oe_F;
    if (cnt > 1) begin
        $display("[%0t] ERROR: multiple bus drivers active!
oe_count=%0d  inst=%b", $time, cnt, inst);
        $fatal; // stop simulation to catch bug early
    end
end
`endif
```

Why: explicit OE mapping and simulation checks prevent more than one driver being active and make the design behavior deterministic; registering OE is optional but helpful if you still see glitches.

2.6 Small final touch: ensure ALU input selection & write-back ordering consistent

If `micro.v` uses combinational routing where the ALU takes values from tri-stated bus or directly from registers, ensure ALU inputs are driven from register outputs (which are stable

after the posedge) and that the write back (ld) is triggered correctly. This is usually already present; the critical parts were above (dff non-blocking and correct decoder wiring).

---

3) Why these changes fix the problems (detailed)

- **ADD/SUB swapped**: explicitly wiring inst[5] into `add_sub.sub` and ensuring the `add_sub` module's `sub` semantics (1 => subtract) fixes swapped behavior. No more polarity inversion.

- **Register C not loading**: the LD decoder bit indices now explicitly map to registers by name (ld[2] → C). This fixes the off-by-one or reversed mapping that prevented C from being loaded when the instruction selected C.

- **OUT A sometimes `zzzzzzzz`**: there were two contributors:

    - Tri-state enable polarity or enabling logic was incorrect — fixed by making oe signals active-high and clarifying the tri buffer semantics.

    - The bigger timing issue (race) came from using blocking `=` in DFFs. With blocking assignments, the order of statements inside `always @(posedge clk)` can cause one flop to update earlier than another, which, when combined with combinational decode logic reading flops to generate OE, results in the bus being driven with stale values or briefly undriven. Non-blocking assignments (`<=`) guarantee simultaneous update semantics at the clock edge — this removes the order-dependent behavior and the race. After the posedge, combinational decode sees the new A, and when `OUT A` is asserted, the tri-state buffer drives the bus with the new A reliably (no old A or intermediate Z).

- **Single-driver assertion**: adding a simulation-time guard ensures any accidental multiple-driver scenarios are flagged early; often Z occurs when all drivers are disabled or when drivers are in contention — catching confusion early prevents intermittent Z behaviour.

---

4) Expected `tb` monitor logs — before and after

Below I show the relevant failing trace you provided (before) and the expected fixed behaviour (after). Times and signals follow your original failing run.

Before (failing run you provided)

```
Time(ns)  data_in     inst       data_out
---------------------------------------
   0      00000000   00000000    zzzzzzzz
  10      00001111   00001000    zzzzzzzz   // IN A (A should be 0x0F
after next edge)
  20      00000010   00010000    zzzzzzzz   // IN B (B should be
0x02)
  30      00000000   01000010    zzzzzzzz   // ADD B  (A <= A + B)
expect 0x11 after edge
  40      00000000   00000001    00001111   // OUT A  (sees old A)
  50      00000000   00000001    zzzzzzzz   // second OUT A
(sometimes Z)
```

After (expected passing run with edits applied)

```
Time(ns)  data_in     inst       data_out
---------------------------------------
   0      00000000   00000000    zzzzzzzz
  10      00001111   00001000    zzzzzzzz   // IN A (load asserted,
new A will appear after edge)
  20      00000010   00010000    zzzzzzzz   // IN B (load asserted)
  30      00000000   01000010    zzzzzzzz   // ADD B  (A <= A + B)
  40      00000000   00000001    00010001   // OUT A  (now shows new
A = 0x11)
  50      00000000   00000001    00010001   // second OUT A (still
shows 0x11, not Z)
```

Notes:

- The important change is at time 40: `data_out` becomes `00010001` (0x11) — the updated A — instead of old `00001111` (0x0F) or `zzzzzzzz`.

- `data_out` remains valid on subsequent cycles while OE remains asserted.

---

5) Additional small improvements / best practices

- **Use non-blocking (`<=`) for all synchronous registers** — consistent rule prevents many timing bugs.
- **Add simulation assertions**:
    - Assert at-most-one-driver on any shared bus.
    - Assert that decoder outputs are one-hot when expected (or zero for NOP).
    - Example assertion snippet (wrap in `ifndef SYNTHESIS`).
- **Document signal polarities** (comments at decode and tri-state instantiation) — reduce confusion later.
- **Add a bus-keeper or weak pull (if target FPGA allows)** — to avoid floating `Z` in case of no driver. In RTL simulation prefer to avoid and catch logic errors instead.
- **Make decoder outputs one-hot and add a default case** in the decode module to avoid X propagation.
- **Small timing fix**: if your micro architecture requires combinational instructions to immediately drive the bus after a register update in the *same* active edge (that is, if you need combinational result *without* waiting a clock), prefer design where the register writeback and subsequent read are separated into different clock phases; but since your requirement was "ADD then immediate OUT" in next cycle, the non-blocking DFF approach handles it properly.

---

6) Final remarks & simulation note

- I could not run your testbench here; I **did not** change your instruction encoding semantics beyond fixing the control polarity and index wiring. If your module or port names differ, apply the same small edits: (1) change DFFs to non-blocking, (2) fix

add/sub polarity, (3) correct LD decoder → register index mapping, (4) make OE polarity active-high in tri-state module and ensure only one driver active.

- If after applying these changes you still see an intermittent old-value or Z at the bus when doing back-to-back ops, please paste the exact `micro.v` (full file) and the `decode3to8` and tri-buffer instantiation lines — I will produce a direct 1:1 edit for your repo. The edits above are intentionally minimal and address the exact symptoms and the typical root causes you described.