Zaks32-2025: Microcoded 32-bit System — Architecture & Delivery Plan (v1)

Step 1 — Datapath Assembly (Deliverables & Repo Scaffold)

This step corresponds to the **"chips" stage** in Zaks' methodology: define the datapath modules, registers, and buses, with no microcode or sequencer yet.

Core Modules (SystemVerilog Stubs)

- `regfile.sv` — 16 × 32-bit general-purpose registers (GPRs)
- 2 read ports, 1 write port.
- `r0` hardwired to 0.
- `alu.sv`
- Ops: ADD, SUB, AND, OR, XOR, NOT.
- Shifts/rotates handled via external shifter for now.
- Flags: Negative, Zero, Carry, Overflow (NZCV).
- `shifter.sv`
- 32-bit barrel shifter.
- Ops: SLL (shift left logical), SRL (shift right logical), SRA (shift right arithmetic).
- `pc.sv`
- 32-bit register with load and increment (+4).
- `ir.sv`
- Instruction register, 32-bit.
- Provides decoded fields: opcode, rd, rs1, rs2, imm.
- `mar.sv` — Memory Address Register.
- `mdr.sv` — Memory Data Register.
- `flags.sv` — Stores NZCV with write-enable.
- `bus_mux.sv` — Source/dest selection for datapath interconnect.

Example Stub — Register File

```
module regfile (
    input  logic        clk,
    input  logic        we,
    input  logic [3:0]  rs1, rs2, rd,
    input  logic [31:0] wd,
    output logic [31:0] rd1, rd2
);
    logic [31:0] regs[15:0];

    // read ports (combinational)
    assign rd1 = (rs1 == 4'd0) ? 32'd0 : regs[rs1];
    assign rd2 = (rs2 == 4'd0) ? 32'd0 : regs[rs2];

    // write port
    always_ff @(posedge clk) begin
        if (we && rd != 4'd0)
            regs[rd] <= wd;
    end
endmodule
```

Repo Scaffold

```
Zaks32-2025/
  rtl/
    regfile.sv
    alu.sv
    shifter.sv
    pc.sv
    ir.sv
    mar.sv
    mdr.sv
    flags.sv
    bus_mux.sv
  sim/
    tb_regfile.sv
    tb_alu.sv
    tb_shifter.sv
    tb_pc.sv
  Makefile
  README.md
```

Makefile Targets

```
make lint     # verilator --lint-only on all rtl/*.sv
make sim      # run testbenches
```

Step 1 Checklist

- [ ] Create all module stubs in `rtl/`.
- [ ] Write testbenches in `sim/` for regfile, ALU, shifter, PC.
- [ ] Lint + simulate until unit tests pass.
- [ ] Confirm datapath primitives behave correctly.

✅ Once Step 1 is complete, we'll move to **Step 2 — Microinstruction Format + Sequencer + First Microcode Listings** (Zaks' "control" stage).

Step 2 — Control Plane: Microinstruction Format, Sequencer, and Starter Microcode

This phase wires the **soul** of the machine: define the horizontal microinstruction, implement the micro-sequencer, and author initial microcode to realize a minimal ISA. It mirrors Zaks' execution-step breakdowns and TMP/MAR/MDR staging model.

2.1 Microinstruction Word (horizontal)

Width: 96 bits (extensible). **Semantics:** parallel control; one micro-step per clock.

```
package z32u_pkg;
  typedef enum logic [7:0] {
    U_ALU_PASS, U_ALU_ADD, U_ALU_SUB, U_ALU_AND, U_ALU_OR,
    U_ALU_XOR, U_ALU_NOT, U_ALU_SHL, U_ALU_SHR, U_ALU_SAR
  } u_alu_e;

  typedef struct packed {
    // Sequencing
    logic [11:0] next;     // next uPC
    logic        jam_z;    // OR Z into next[0]
    logic        jam_n;    // OR N into next[0]
    logic        dispatch; // high bits of NEXT from IR opcode
    logic        endi;     // return to fetch (uPC ← 0)

    // Datapath ops
    u_alu_e      alu_op;   // ALU function
    logic [3:0]  src_a;    // mux sel A: {REG,PC,MDR,IMM,CONST0,CONST1,...}
    logic [3:0]  src_b;    // mux sel B: {REG,MAR,SHAMT,CONST4,...}
```

```
    logic        a_is_rs1; // choose rs1 vs rd/rs2 when src_a==REG
    logic        b_is_rs2; // choose rs2 vs rd/rs1 when src_b==REG

    // Dest enables (one-hots allowed together)
    logic        wr_rd;    // write back to rd
    logic        wr_pc;
    logic        wr_mar;
    logic        wr_mdr;

    // Misc controls
    logic [1:0]  shift;    // {NONE,SLL,SRL,SRA} (if shifter separate)
    logic        mem_rd;   // memory cycle: read (MDR ← MEM[MAR])
    logic        mem_wr;   // memory cycle: write (MEM[MAR] ← MDR)
    logic        ir_load;  // IR ← MDR
    logic        pc_inc;   // PC ← PC + 4
    logic        flags_we; // NZCV ← ALU
    logic        imm_load; // IMM ← IR.imm (sign/zero select in decode)
    logic        io_space; // access MMIO aperture on bus

    // Spare for interrupts/exc/cache hooks
    logic [31:0] rsv;
  } uinstr_t;
endpackage
```

Notes - `dispatch` jumps into an opcode-indexed page. `endi` collapses back to fetch. - `jam_z/jam_n` provide 2-way conditional branching without an extra ROM read. - Dest signals are independent to allow fused write-backs in one micro-step.

2.2 Micro-Sequencer

```
module sequencer (
  input  logic        clk, rst,
  input  logic  [5:0] ir_opcode,
  input  logic        flag_z, flag_n,
  input  z32u_pkg::uinstr_t ui,
  output logic [11:0] upc
);
  logic [11:0] nxt;
  always_ff @(posedge clk or posedge rst) begin
    if (rst) upc <= 12'd0; else begin
      nxt = ui.next;
      if (ui.dispatch) nxt[11:6] = {2'b00, ir_opcode};
      if (ui.jam_z && flag_z) nxt[0] = 1'b1;
      if (ui.jam_n && flag_n) nxt[0] = 1'b1;
      upc <= ui.endi ? 12'd0 : nxt;
    end
  end
endmodule
```

2.3 Fetch/Decode Microcode Skeleton (human-readable YAML)

```
meta:
  width_bits: 96
  entries: 4096
symbols:
  REG_RS1: 1; REG_RS2: 2; REG_RD: 3; PC: 4; MDR: 5; IMM: 6; MAR: 7
  WR_RD: 1; WR_PC: 2; WR_MAR: 3; WR_MDR: 4
pages:
  0x000:   # common fetch/decode space
    FETCH0: { mem_rd:1, wr_mdr:1, next: FETCH1 }
    FETCH1: { ir_load:1, pc_inc:1, dispatch:1, next: 0x040 }

  0x040:   # opcode dispatch base (ir.opcode in next[11:6])
    # NOP entry point at 0x040 | opcode
```

2.4 Opcode Starters (microcode listings)

Below are minimal sequences to light up the core flows. Naming assumes a 32-bit R-type and I-type ISA as defined in Step 0/1.

NOP

```
NOP:
  - { endi:1 }
```

MOV rd, rs (register→register transfer via ALU PASS)

```
MOV:
  - { src_a: REG_RS1, alu_op: U_ALU_PASS, wr_rd:1, endi:1 }
```

ADD rd, rs1, rs2

```
ADD:
  - { src_a: REG_RS1, src_b: REG_RS2, alu_op: U_ALU_ADD, flags_we:1, wr_rd:
1, endi:1 }
```

LD rd, [rs1+imm]

```
LD:
  - { src_a: REG_RS1, src_b: IMM, alu_op: U_ALU_ADD, wr_mar:1 }   # eff addr
  - { mem_rd:1 }                                                   # MDR ←
```

```
MEM[MAR]
  - { wr_rd:1, endi:1 }                                    # rd ← MDR
```

ST [rs1+imm], rs2

```
ST:
  - { src_a: REG_RS1, src_b: IMM, alu_op: U_ALU_ADD, wr_mar:1 }   # eff addr
  - { src_a: REG_RS2, alu_op: U_ALU_PASS, wr_mdr:1 }              # MDR ← rs2
  - { mem_wr:1, endi:1 }
```

BEQ rs1, rs2, off (set flags then branch on Z)

```
BEQ:
  - { src_a: REG_RS1, src_b: REG_RS2, alu_op: U_ALU_SUB, flags_we:1 }
# compare
  - { src_a: PC, src_b: IMM, alu_op: U_ALU_ADD, wr_pc:1, jam_z:1, endi:1 }
# PC←PC+off if Z
```

2.5 Microassembler I/O (CSV/YAML → HEX)

- Assemble YAML/CSV into 96-bit words (little-endian byte order) for `microstore.sv` init.
- Emit a symbol map (uPC labels → addresses) so testbenches can assert (uPC traces).

2.6 Test Strategy for Step 2

- **uPC trace checks:** For each opcode, assert the expected micro-sequence (e.g., `LD` hits three specific uPCs in order).
- **Architectural state diffs:** Before/after snapshots for rd/mem/PC vs. Python golden ISS.
- **Negative tests:** Ensure `endi` returns to `FETCH0` reliably after each handler.

2.7 Integration Hooks into Step 1 Datapath

- Wire `src_a/src_b` muxes to the existing datapath sources (REG/PC/MDR/IMM/MAR).
- Drive `wr_*` enables to `regfile`, `pc`, `mar/mdr` modules.
- Connect `mem_rd/mem_wr` to the memory controller handshake (blocking in v1).

Ready for Step 3 (later): Interrupt vectors, IO space gating, and micro-optimizations (fused write-back, dual-port microstore).