# Superscalar Out-of-Order Processor: Project Report

Team id : 27
Mrudul Jambhulkar (21d070044)
Saurav Raj (21d070064)

May 11, 2025

## Contents

## 1 IF Stage Design

The IF stage fetches two 16-bit instructions per cycle from instruction memory and updates the Program Counter (PC).

Listing 1: IF Stage Module

```verilog
module IF_stage (
    input wire clk, rst, stall,
    input wire [15:0] branch_target,
    input wire branch_taken,
    output reg [15:0] pc,
    output wire [15:0] inst1, inst2, pc1, pc2
);
    wire [15:0] addr1, addr2;
    wire [15:0] next_pc = branch_taken ? branch_target : pc + 16'
       d2;
    Inst_Memory IM (
        .clk(clk), .addr_1(addr1), .addr_2(addr2),
        .IM_output_1(inst1), .IM_output_2(inst2)
    );
    always @(posedge clk or posedge rst) begin
        if (rst) pc <= 16'd0;
        else if (!stall) pc <= next_pc;
    end
    assign addr1 = pc;
    assign addr2 = pc + 16'd1;
    assign pc1 = addr1;
    assign pc2 = addr2;
endmodule
```

## 2 Instruction Memory Initialization

The instruction memory was initialized with six instructions to test ALU (ADD, NAND, ADI, LLI) and memory (LW, SW) operations.

Listing 2: Instruction Memory Initialization

```
instructions[0] = 16'b0001_100_010_011_000; // ADD R4, R2, R3
    // [15:12]=0001 (ADD), [11:9]=100 (R4), [8:6]=010 (R2),
       [5:3]=011 (R3), [2:0]=000 (unused)
    // Operation: R4 = R2 + R3 = 0x0005 + 0x0003 = 0x0008
    // Tests: ALU0/ALU1 arithmetic, register renaming, ROB commit

    instructions[1] = 16'b0010_101_010_011_000; // NAND R5, R2,
       R3
    // [15:12]=0010 (NAND), [11:9]=101 (R5), [8:6]=010 (R2),
       [5:3]=011 (R3), [2:0]=000 (unused)
    // Operation: R5 = ~(R2 & R3) = ~(0x0005 & 0x0003) = ~(0x0001
       ) = 0xFFFE
    // Tests: ALU0/ALU1 logical operation, CDB broadcast

    instructions[2] = 16'b0000_110_001_000100; // ADI R6, R1, 4
    // [15:12]=0000 (ADI), [11:9]=110 (R6), [8:6]=001 (R1),
       [5:0]=000100 (imm=4)
    // Operation: R6 = R1 + 4 = 0x0010 + 4 = 0x0014
    // Tests: Immediate arithmetic, ALU pipeline

    instructions[3] = 16'b0100_111_110_000000; // LW R7, 0(R6)
    // [15:12]=0100 (LW), [11:9]=111 (R7), [8:6]=110 (R6),
       [5:0]=000000 (imm=0)
    // Operation: R7 = M[R6 + 0] = M[0x0014] = 0x1234
    // Tests: LS0 load, data memory access, ROB commit

    instructions[4] = 16'b0101_010_001_001000; // SW R2, 8(R1)
    // [15:12]=0101 (SW), [11:9]=010 (R2), [8:6]=001 (R1),
       [5:0]=001000 (imm=8)
    // Operation: M[R1 + 8] = R2 = M[0x0018] = 0x0005
    // Tests: LS0 store, data memory write

    instructions[5] = 16'b0011_011_000001010; // LLI R3, 10
    // [15:12]=0011 (LLI), [11:9]=011 (R3), [8:0]=000001010 (imm
       =10)
    // Operation: R3 = 10 = 0x000A
    // Tests: Immediate load, ARF update
        // Fill remaining with NOPs (e.g., ADI R0, R0, 0)
        for ( i = 6; i < 256; i = i + 1) begin
            instructions[i] = 16'b0000_000_000_000000;
        end
    end
```

Initial conditions:

- ARF: R1 = 0x0010, R2 = 0x0005, R3 = 0x0003, others = 0.
- Data Memory: M[0x0014] = 0x1234.

# 3 Dispatch Unit

The `Dispatch_Unit` is responsible for managing the flow of up to two instructions from the Instruction Decode (ID) stage to the Reservation Station (RS), Register Rename File (RRF), Architectural Register File (ARF), and Reorder Buffer (ROB). It ensures the correct allocation of RRF tags, resolves source operand availability, and detects hazards and resource conflicts that may require stalling.

## Key Functionalities

- **RRF Tag Allocation:** Scans the 32-entry RRF busy status vector to assign free tags for two instructions, avoiding reuse and ensuring correctness in register renaming.
- **Operand Handling:** Maps architectural register indices to corresponding physical (RRF) tags using ARF mappings. Checks for busy status and fetches operand data for both source operands. Handles special cases for memory operations (e.g., distinguishing between `Rb` and `Rc` usage).
- **Reservation Station Dispatch:** If no hazards or structural conflicts exist and RRF tags are available, dispatches instructions to the RS with all required fields, including operand data, validity flags, immediate values, and control signals (e.g., ALU, memory, branch, jump).
- **Register and ROB Updates:** For instructions that write to registers, updates the ARF with new RRF mappings and sets busy bits. Corresponding entries in the ROB are populated with program counter, opcode, destination registers, and RRF tags.
- **Hazard and Resource Conflict Handling:** The module monitors hazard detection and RS availability signals (`rs_full`, `rs_has_one_slot`) to determine whether to proceed or assert the `stall` signal, preventing dispatch in unsafe conditions.
- **Reset Behavior:** On reset, all output registers and control signals are cleared to ensure a clean system state.

This dispatch unit plays a critical role in enabling out-of-order execution by decoupling instruction issue from operand readiness and managing precise tracking of physical register usage.

# 4 Reservation Station Module

The `Reservation_Station` module implements a reservation station (RS) for dynamic scheduling in a pipelined processor. It handles instruction dispatch, operand readiness, broadcasting via a Common Data Bus (CDB), and instruction issue to functional units such as ALUs and Load/Store Units (LSU).

## 4.1 Inputs and Outputs

- **Inputs:**

    - **Clock and Reset:** `clk`, `rst`
    - **Dispatch Unit:** Provides two instructions per cycle with program counters, opcodes, operand data, operand tags, operand validity flags, destination tags, condition flags, and compare flags.
    - **Common Data Bus (CDB):** Supplies result values and tags from execution units to update waiting operands.
- **Outputs:**
    - Instructions ready to issue are output to `ALU0`, `ALU1`, and `LS0`.
    - A `rs_full` flag is output to indicate if the RS is full and further dispatching should stall.

## 4.2 RS Entry Structure

Each RS entry holds the following fields:

- Program counter (`rs_pc`)
- Opcode (`rs_opcode`)
- Operand data (`rs_opr1_data`, `rs_opr2_data`)
- Operand tags and validity (`rs_opr1_tag`, `rs_opr2_tag`, `rs_opr1_valid`, `rs_opr2_valid`)
- Destination tag (`rs_rrf_dest`)
- Condition codes (`rs_cz`) and compare flags (`rs_cmp`)
- Busy flag (`rs_busy`)

## 4.3 Free Entry Allocation

A combinational logic block scans all 32 entries to find the first and second free entries, storing their indices in `free_entry_1` and `free_entry_2`. If none are found, the RS is marked as full via `rs_full`.

## 4.4 Instruction Dispatching

On the positive edge of the clock:

- If the reset is active, all RS entries are cleared.
- If valid instructions are received and free entries are available, the module stores each instruction's metadata in the allocated RS entries and marks them busy.

## 4.5 Operand Update from CDB

At each clock edge, if a busy RS entry has invalid operands, the module checks whether the operand tag matches any of the CDB tags (`cdb_tag_0`, `cdb_tag_1`). If a match is found and the entry is valid, the operand data is updated, and its validity is set.

## 4.6 Instruction Issue Logic

This block scans all RS entries for ready instructions (both operands valid and entry busy):

- **Load/Store Instructions** (`opcode == 1000, 1001`) are assigned to `LS0`.
- **ALU Instructions** (e.g., `ADA, NDU, ADC, ADZ, NDC`) are assigned to `ALU0` and `ALU1`.

Ready entries are selected for issue based on availability and assigned to respective functional unit outputs, including their operands, opcode, PC, and flags.

## 4.7 Clearing Issued Entries

On the clock's rising edge, entries that were issued in the current cycle are marked as not busy, making them available for future dispatch.

# 5 Reorder Buffer (ROB)

The Reorder Buffer (ROB) is a crucial structure used to facilitate out-of-order execution while maintaining in-order commitment of instructions. It temporarily holds information related to dispatched instructions and ensures that the architectural state (ARF and PC) is updated only after instructions are safely executed.

**Module:** `ROB.v`

**Inputs:**

- `clk, rst`: Clock and reset signals.
- `iss_valid_0, iss_valid_1`: Valid signals for instructions issued by the dispatch stage.
- `pc_iss_0, pc_iss_1`: Program counters of dispatched instructions.
- `opcode_iss_0, opcode_iss_1`: Opcodes for operation type identification.
- `arf_dest_iss_0, arf_dest_iss_1`: Target architectural registers.
- `rrf_dest_iss_0, rrf_dest_iss_1`: Destination tags in the Rename Register File.
- `cdb_data_0, cdb_data_1`: Result data from Common Data Bus (CDB).
- `cdb_tag_0, cdb_tag_1`: RRF tags for result identification.
- `cdb_valid_0, cdb_valid_1`: Valid signals from CDB.

**Outputs:**

- `arf_value_0, arf_value_1`: Committed values to be written to ARF.
- `arf_dest_0, arf_dest_1`: Corresponding ARF destinations.
- `arf_valid_0, arf_valid_1`: Signals indicating successful commit.

- `dest_pc_out`: Updated program counter (used for branches).
- `dest_pc_valid`: Indicates whether the new PC is valid.

**ROB Entry Structure (54 bits):**

- Bits [53:38]: `dest_pc` (16 bits)
- Bit [37]: Carry flag
- Bit [36]: Zero flag
- Bits [35:32]: Opcode (4 bits)
- Bits [31:16]: Instruction PC (16 bits)
- Bits [15:0]: Computed value
- Bits [8:4]: RRF destination tag (5 bits)
- Bits [3:1]: ARF destination (3 bits)
- Bit [2]: Busy bit
- Bit [1]: Executed bit
- Bit [0]: Issued bit (Committed)

**Working Phases:**

1. **Dispatch:** Instructions are inserted into the ROB with metadata (PC, opcode, ARF/RRF destinations). The `tail` pointer is incremented accordingly.
2. **Update from CDB:** ROB entries are scanned to check if their RRF tag matches the CDB tag. If matched, the value is written into the entry, and the executed bit is set. Destination PC and flags are optionally updated.
3. **Commit:** At most two instructions (head and head+1) can be committed per cycle if both are busy and executed. Results are written to ARF. For branches, `dest_pc_out` is updated.

## Common Data Bus (CDB)

The Common Data Bus (CDB) is responsible for broadcasting results from functional units (ALUs, LS unit) to other pipeline components such as the Reservation Stations (RS), Rename Register File (RRF), and Reorder Buffer (ROB). It allows for simultaneous result forwarding via two channels and uses arbitration to manage multiple valid results in a given cycle.

**Module: `CDB.v`**

**Inputs**

- `clk, rst` – Clock and reset signals.
- `alu0_ex_aluc, alu1_ex_aluc` – Computed results from ALU0 and ALU1.

- `alu0_rrf_dest, alu1_rrf_dest` – Destination RRF tags for ALU results.
- `alu0_valid, alu1_valid` – Validity signals for ALU outputs.
- `ls0_ex_aluc` – Load result from Load-Store Unit (LS0).
- `ls0_rrf_dest` – Destination RRF tag for LS0 result.
- `ls0_valid` – Valid signal for LS0 result.

**Outputs**

- `cdb_data_0, cdb_data_1` – Result values broadcasted on channels 0 and 1.
- `cdb_tag_0, cdb_tag_1` – Corresponding RRF destination tags.
- `cdb_valid_0, cdb_valid_1` – Valid signals indicating if the data and tag are valid.

**Arbitration Policy**

- Priority order: **ALU0** $>$ **ALU1** $>$ **LS0**.
- Maximum of two valid results can be broadcast in a cycle (dual-channel).
- If ALU0 is valid, it takes `cdb_channel_0`. Then:
  - If ALU1 is also valid, it takes `cdb_channel_1`.
  - Else if LS0 is valid, LS0 takes `cdb_channel_1`.
- If ALU0 is not valid but ALU1 is, then:
  - ALU1 gets `cdb_channel_0`.
  - If LS0 is valid, it takes `cdb_channel_1`.
- If only LS0 is valid, it uses `cdb_channel_0`.

## 6 Testbench

A testbench was created to simulate the processor with a 50ns clock and 100ns reset, running for 50 cycles. It monitors PC, ARF, ROB commits, and stalls.

Listing 3: Testbench Snippet

```
`timescale 1ns / 1ps

module Top_Level_tb ();

// Testbench signals
reg clk;
reg rst;

// Instantiate the top-level module
Top_Level uut (
    .clk(clk),
    .rst(rst)
);

// Clock generation: 50ns period (20MHz)
initial begin
    clk = 0;
    forever #25 clk = ~clk;
end

// Reset and simulation control
initial begin
    // Initialize signals
    rst = 1;
    #100; // Hold reset for 100ns
    rst = 0;

    // Run simulation for 50 cycles (2500ns)
    #2500;
    $display("Simulation completed.");

    // Final state check
    $display("Final ARF State:");
    $display("R1 = %h (Expected: 0x0010)", uut.arf.arf_data_1);
    $display("R2 = %h (Expected: 0x0005)", uut.arf.arf_data_2);
    $display("R3 = %h (Expected: 0x000A)", uut.arf.arf_data_3);
    $display("R4 = %h (Expected: 0x0008)", uut.arf.arf_data_4);
    $display("R5 = %h (Expected: 0xFFFE)", uut.arf.arf_data_5);
    $display("R6 = %h (Expected: 0x0014)", uut.arf.arf_data_6);
    $display("R7 = %h (Expected: 0x1234)", uut.arf.arf_data_7);
    $display("Data Memory[0x0018] = %h (Expected: 0x0005)", uut.
       data_memory.mem[24]);

    $finish;
end
```

```

// Monitor key signals
initial begin
    $monitor("Time=%0t␣|␣PC=%h␣|␣ARF:␣R1=%h,␣R2=%h,␣R3=%h,␣R4=%h,
       ␣R5=%h,␣R6=%h,␣R7=%h␣|␣ROB␣Commit:␣valid_0=%b,␣dest_0=%h,␣
       value_0=%h␣|␣Stall=%b",
             $time,
             uut.if_pc,
             uut.arf.arf_data_1,
             uut.arf.arf_data_2,
             uut.arf.arf_data_3,
             uut.arf.arf_data_4,
             uut.arf.arf_data_5,
             uut.arf.arf_data_6,
             uut.arf.arf_data_7,
             uut.rob.arf_valid_0,
             uut.rob.arf_dest_0,
             uut.rob.arf_value_0,
             uut.stall);
end

endmodule
```