IITB-RISC-23: Design and Implementation of a 2-Way Out-of-Order Superscalar Processor

EE748 Group 1
Raghav Sapre (22B1241)
Samar Perwez (22B3913)
Devavrat Patni (22B3969)

August 24, 2025

1 Introduction

This report presents the design and implementation of a 2-way out-of-order superscalar processor tailored for the IITB-RISC-23 Instruction Set Architecture (ISA). This out-of-order design enhances performance by allowing instructions to execute as soon as their operands are available, thereby exploiting greater instruction-level parallelism. The processor employs a 6-stages comprising Fetch, Decode, Dispatch, Issue, Execute and Write back stages. Key components include the Reorder Buffer (ROB) for maintaining program order, Reservation Stations (RS) for managing instruction issuance, two ALU Units, and a Load/Store Unit with a Store Buffer for efficient memory operations. The implementation leverages register renaming, via a Renamed Register File (RRF), to eliminate false dependencies and a sophisticated hazard mitigation strategy to ensure correct execution. This report provides a detailed exploration of the processor's architecture, major components, hazard mitigation techniques, and simulation results, drawing on the Verilog scripts provided, including `Assembled_Superscalar.v`, `FetchStage.v`, `Decoder.v`, `RS.v`, `ROB.v`, `RF.v`, `ALU.v`, `load_store_unit.v`, `StoreBuffer.v`, `DataMemory.v`, and `IF_ID_PR.v`.

2 ISA Overview

The IITB-RISC-23 ISA defines a 16-bit instruction set with three formats: R-type for register-based operations, I-type for immediate-based operations, and J-type for jump instructions. It supports eight general-purpose registers (R0 to R7), with R0 serving as the program counter (PC). The ISA encompasses 14 instructions, including arithmetic operations (ADD, NAND, ADI, LLI), memory operations (LW, SW, LM, SM), and control flow change instructions (BEQ, BLT, BLE, JAL, JLR, JRI). Conditional execution relies on Carry (C) and Zero (Z) flags, while Load Multiple (LM) and Store Multiple (SM) instructions enable efficient block memory transfers. This out-of-order processor must adhere to these specifications while optimizing performance through dynamic instruction scheduling.

3 Processor Architecture

The 2-way out-of-order superscalar processor is designed to fetch, decode, and issue two instructions per cycle, executing them out of program order when their operands are available. The Fetch stage retrieves two instructions from the instruction memory, while the Decode stage decodes instructions, performs register renaming, and allocates entries in the ROB and RS. The Issue stage dispatches instructions from the RS to functional units when operands are ready. The Execute stage processes instructions in ALUs or LSU, followed by the Memory Access stage for load and store operations. The load instructions will either do a memory access or forward the data from the latest store instruction with the same address in the `Store Buffer`. Finally, the Write Back stage retires instructions in program order via the ROB, updating the architectural state.

The top-level module, `Assembled_Superscalar.v`, integrates all components, defining the pipeline's connectivity and control signals. It orchestrates instruction flow, ensuring that dependencies are resolved and instructions are retired correctly. The following excerpt illustrates a typical top-level structure:

```
module Assembled_Superscalar #(
    parameter SB_SIZE           = 5,
    parameter ROB_SIZE          = 7,
    parameter RRF_SIZE          = 7,
    parameter R_CZ_SIZE         = 8,
    parameter RS_AL_ENTRY_SIZE  = 145,
    parameter RS_LS_ENTRY_SIZE  = 75,
    parameter ROB_ENTRY_SIZE    = 55
)(
    input wire clk,       // external clock
    // input wire stall,      // external stall
    // input wire flush,      // external flush
    input wire reset
    );
    // ------------------------------------------------------------------------

    // FetchStage inputs
    wire R0w;
    wire [15:0] R0d;

    // FetchStage outputs
    wire [15:0] fetch_I1, fetch_I2; // to decode stage
    wire fetch_I1V, fetch_I2V; // to decode stage
    wire fetch_I1P, fetch_I2P; // to decode stage
    wire [15:0] fetch_I1PC, fetch_I2PC; // to decode stage
    // Additional wires for RS, ROB, and functional units
    // Instantiations of FetchStage, IF_ID_PR, Decoder, etc.
endmodule
```

This module connects the pipeline stages, facilitating out-of-order execution while maintaining ISA compliance.

4 Major Blocks Description

4.1 Fetch Stage

The Fetch Stage retrieves two 16-bit instructions per cycle from the instruction memory, using the PC to address the memory. The PC is incremented by 4 (since two 16-bit instructions occupy 4 bytes) to fetch the next pair. The `FetchStage.v` module interfaces with the instruction memory and passes the fetched instructions and PC to the `IF_ID_PR` pipeline register. The PC may be reset to a certain value in cases of mispredicted Jumps and branches.

The `IF_ID_PR.v` module acts as a pipeline register, capturing the fetched instructions and PC, ensuring synchronized transfer to the Decode stage. It includes flush logic to handle control hazards, such as branch mispredictions.

4.2 Decode Stage

The Decode Stage processes the two fetched instructions, expanding them into a wider micro-op format (e.g., 42 bits), that includes opcode, register addresses, immediate values, and control signals. The `Decoder.v` module performs register renaming to eliminate Write-After-Read (WAR) and Write-After-Write (WAW) hazards by mapping architectural registers to physical registers. Each decoded instruction is allocated an entry in the ROB and dispatched to the appropriate RS based on its type (e.g., arithmetic or memory). The renaming process uses the Renamed Register File (RRF) to track physical register mappings. The decoder assigns ROB tags to track instructions and ensures that dependencies are correctly managed through physical register assignments.

4.3 Reservation Stations

The Reservation Stations (RS), implemented in `RS.v`, hold instructions until their operands are available, enabling out-of-order issuance. Each RS entry contains fields for the instruction's opcode, source operand tags or values, destination tag, and a valid bit. The RS monitors the availability of operands, either through architectural register values or the renamed register values. When both operands are ready, the instruction is issued to a functional unit (e.g., ALU or Load/Store Unit). We have implemented a

diversified Reservation Station architecture with one RS each for arithmetic and memory operations. A stall may occur if the RS is full and cannot accept new instructions.

The RS supports multiple functional units, likely including two ALUs for arithmetic/logic operations and a Load/Store Unit for memory operations, ensuring efficient resource utilization.

4.4 Reorder Buffer

The Reorder Buffer (ROB), implemented in `ROB.v`, is critical for maintaining program order in out-of-order execution. Each ROB entry tracks an instruction's status, including its destination physical register, architectural register, and completion status. Instructions are added to the ROB during decoding and retired from the head when they and all preceding instructions are complete. The ROB updates the architectural register file upon retirement, ensuring correct program state. In cases of branch misprediction the ROB handles the Global flush signal to the Reservation Stations, Store Buffer, Fetch Pipeline Register and itself to clean off the wrongly fetched instructions. A stall may occur if the ROB is full and cannot accept new instructions.

4.5 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU), implemented in `ALU.v`, performs all arithmetic and logic operations. It takes two operands and an operation code (opcode) as inputs and produces a result. The ALU supports various operations, including addition, NAND and branch target address calculation, thereby branch prediction verification. It can also handle immediate values for certain instructions. The ALU is designed for high throughput and low latency, enabling efficient execution of instructions.

4.6 Load/Store Unit

The Load/Store Unit, implemented in `load_store_unit.v` and supported by `StoreBuffer.v`, manages memory operations. Loads calculate addresses and check the Store Buffer for pending stores to the same address, enabling store forwarding to reduce latency. Stores are buffered in the Store Buffer until they can be committed to memory, ensuring that memory operations do not block other instructions. The `DataMemory.v` module provides the actual memory interface.

The Store Buffer enhances performance by allowing loads to bypass stores, reducing memory access delays.

4.7 Register File

The Register File (RF), implemented in `RF.v`, is a physical register file with 8 architectural registers and 128 renamed registers to support register renaming. It provides operands to the RS and receives results from functional units via the ROB. The RF handles multiple read and write ports to support parallel operations.

5 Hazard Mitigation

The out-of-order design mitigates pipeline hazards through several mechanisms. Data hazards are addressed by register renaming, which eliminates WAR and WAW hazards by assigning unique physical registers to each destination. RAW hazards are managed through operand forwarding, where the RS captures results directly from functional units. Control hazards, such as those caused by branches, are likely handled by flushing the pipeline when a branch is taken, as currently there is no explicit branch predictor. This ensures that the pipeline is cleared when the PC (R0) is updated, maintaining correctness for control flow changes. Structural hazards are minimized by ensuring sufficient RS entries, ROB entries, and functional units. Stalls are added in all buffers (ROB, SB, RS), to handle resource conflicts.

6 Updates from last submission

- Resolved critical issues with Branch and Jump instructions, ensuring correct control flow and robust pipeline flushing on mispredictions.
- Implemented a dedicated register file for Carry and Zero flags, enabling precise flag management and improved support for conditional execution.

7 Simulation Results

Many program codes were written to test the functionality of the processor. The programs were assembled into machine code using a custom python script and loaded into the instruction memory. The processor was simulated using a Verilog simulator, and waveforms were analyzed to verify correct operation.

1. Program 1:

```
LLI  R1  0
LLI  R2  0
LLI  R3  0
LLI  R4  0
LLI  R5  0
LLI  R6  0
LLI  R7  0
LLI  R1  10
LLI  R2  20
LLI  R3  30
LLI  R4  40
LLI  R5  50
LLI  R6  60
LLI  R7  70
LLI  R2  90
LLI  R0  4
LLI  R3  100
LLI  R4  110
LLI  R5  120
LLI  R6  130
LLI  R7  140
```

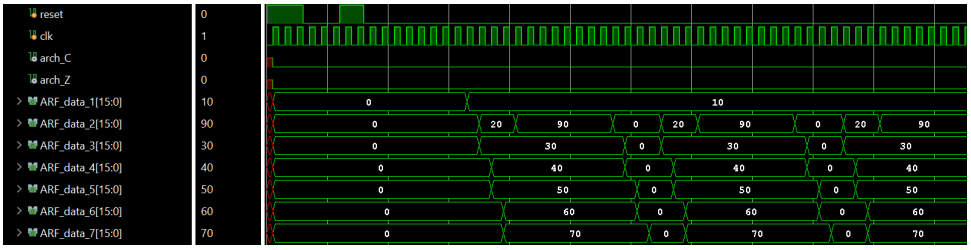

Figure 1: Waveform for Program 1

2. Program 2:

```
LLI  R1  10
LLI  R2  10
LLI  R3  10
LLI  R4  10
LLI  R5  10
LLI  R6  10
LLI  R7  10
LW  R1  R7  10
```

```
LW R2 R7 14
LW R3 R7 18
LW R4 R7 22
LW R5 R7 26
LW R6 R7 30
```

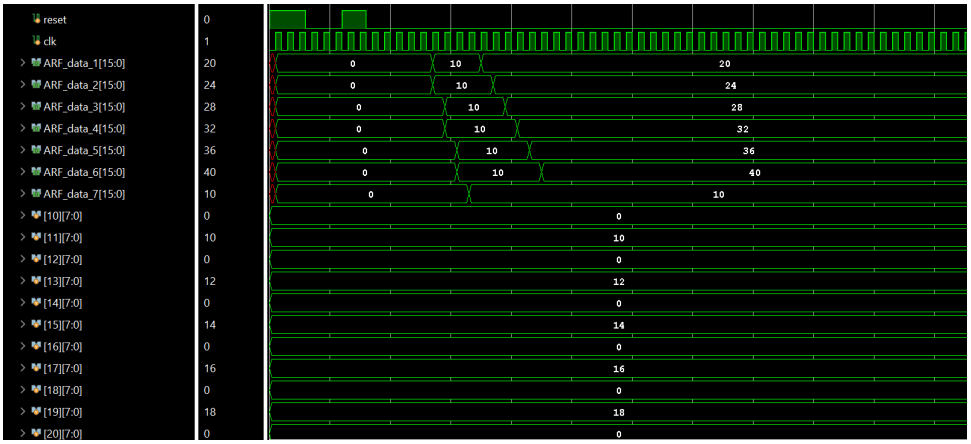

Figure 2: Waveform for Program 2

3. Program 3:

```
LLI R1 20
LLI R2 10
LLI R3 30
LLI R4 50
LLI R5 70
LLI R6 90
LLI R7 110
SW R1 R2 0
SW R3 R2 2
SW R4 R2 4
SW R5 R2 6
SW R6 R2 8
SW R7 R2 10
```

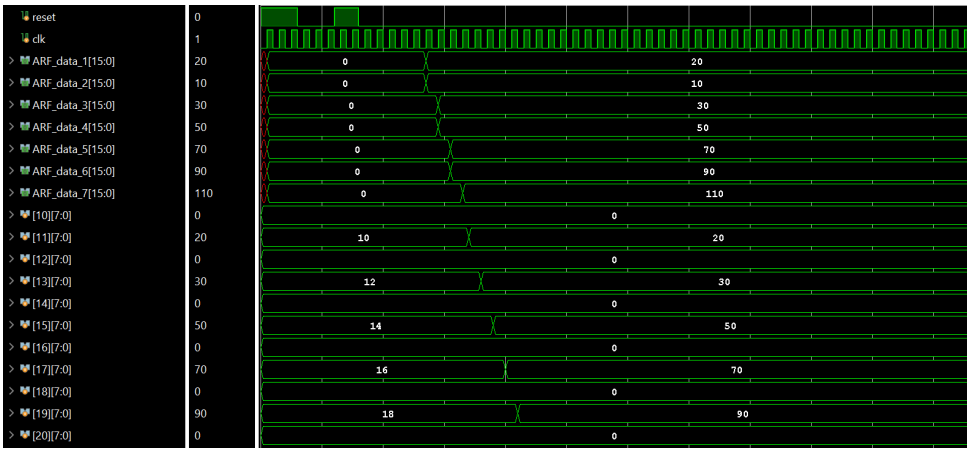

Figure 3: Waveform for Program 3

4. Program 4:

```
LLI  R1  5
LLI  R2  10
LLI  R3  30
LLI  R4  50
LLI  R5  70
LLI  R6  90
LLI  R7  110
ADA  R1  R1  R1
BLE  R1  R4  −2
```

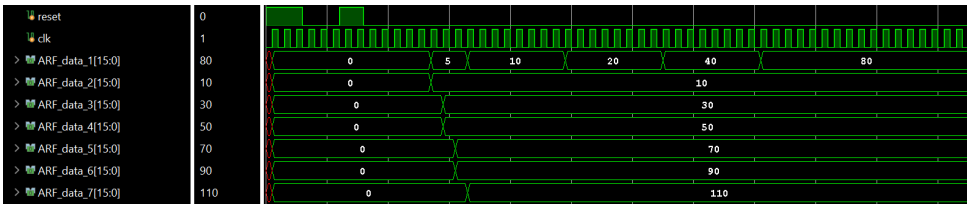

Figure 4: Waveform for Program 4

5. Program 5:

```
LLI  R1  5
LLI  R2  10
LLI  R3  30
LLI  R4  40
LLI  R5  70
LLI  R6  90
LLI  R7  110
ADA  R2  R2  R2
ADA  R3  R3  R3
ADA  R1  R1  R1
BLT  R1  R4  −2
```

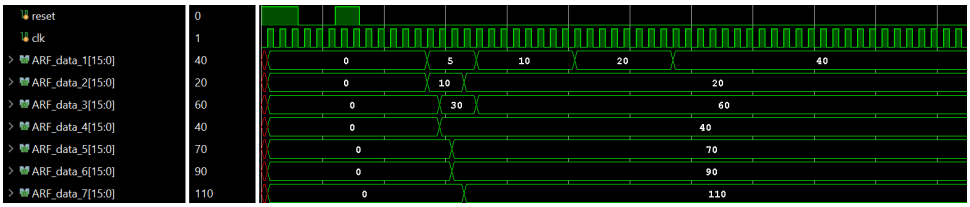

Figure 5: Waveform for Program 5

6. Program 6:

```
LLI  R1  5
LLI  R2  10
LLI  R3  30
LLI  R4  40
LLI  R5  70
LLI  R6  90
LLI  R7  110
ADA  R5  R5  R5
ADA  R3  R3  R3
ADA  R2  R2  R2
JAL  R1  −2
```

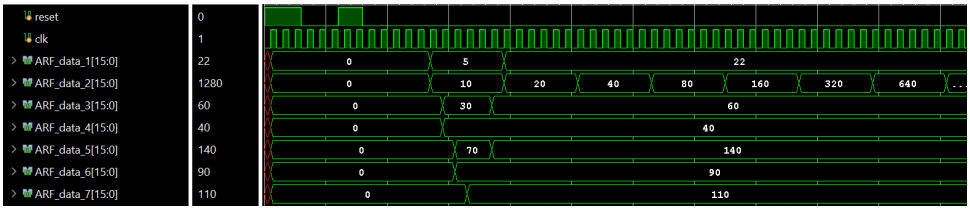

Figure 6: Waveform for Program 6

7. Program 7:

```
LLI  R1  5
LLI  R2  10
LLI  R3  30
LLI  R4  18
LLI  R5  70
LLI  R6  90
LLI  R7  110
ADA  R5  R5  R5
ADA  R3  R3  R3
ADA  R2  R2  R2
JLR  R1  R4
```

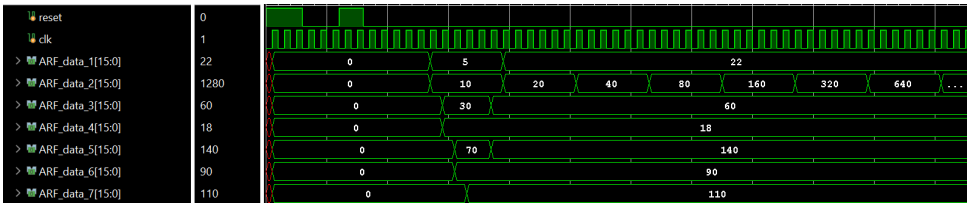

Figure 7: Waveform for Program 7

8. Program 8:

```
LLI  R1  5
LLI  R2  10
LLI  R3  30
LLI  R4  8
LLI  R5  70
LLI  R6  90
LLI  R7  110
ADA  R5  R5  R5
ADA  R3  R3  R3
ADA  R2  R2  R2
JRI  R4  10
```

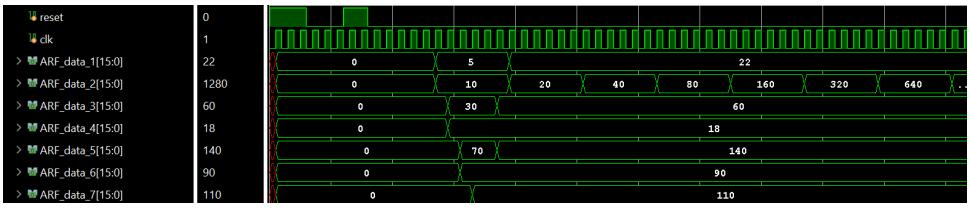

Figure 8: Waveform for Program 8

9. Program 9:

```
LLI  R1  5
LLI  R2  3
LLI  R3  8
LLI  R4  2
LLI  R5  9
LLI  R6  4
LLI  R7  20
ADA  R1  R1  R2
ADA  R3  R3  R4
ADA  R5  R5  R6
ADA  R2  R1  R3
ADA  R4  R3  R5
ADA  R6  R5  R1
SW  R2  R7  10
SW  R4  R7  20
SW  R6  R7  30
LW  R1  R7  10
LW  R3  R7  20
LW  R5  R7  30
ADA  R2  R1  R7
ADA  R4  R3  R2
ADA  R6  R5  R4
ADA  R1  R6  R2
ADA  R3  R1  R4
ADA  R5  R3  R6
ADA  R7  R5  R1
ADA  R2  R7  R3
ADA  R4  R2  R1
```

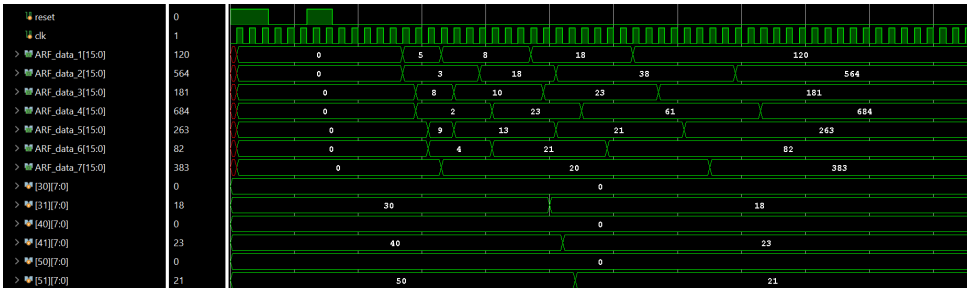

Figure 9: Waveform for Program 9

8 Conclusion

The 2-way out-of-order superscalar processor for the IITB-RISC-23 ISA represents a significant advancement over the in-order design, achieving higher performance through dynamic instruction scheduling. The integration of register renaming, a Reorder Buffer, Reservation Stations, and a Load/Store Unit with a Store Buffer enables efficient out-of-order execution while maintaining program correctness. The Verilog implementation, spanning modules like `Decoder`, `RS`, `ROB`, and `load_store_unit`, demonstrates a robust design that balances complexity and performance. Future enhancements could include branch prediction to reduce control hazard penalties or additional functional units to further increase parallelism. This project showcases the application of out-of-order execution principles in a constrained ISA, providing a foundation for advanced processor design exploration.

9 Work Distribution

The work on this project was divided among team members as follows:

- **Samar Perwez (22B3913):** Implemented the Decode stage and the Register File. Helped in the ideation of the Fetch and ROB stage and final testing.
- **Devavrat Patni (22B3969):** Implemented the Fetch stage and the Reservation Stations. Tested the final integrated architecture.
- **Raghav Sapre (22B1241):** Implemented the Reorder Buffer, Store Buffer and Flag RRF. Helped in testing the final integrated architecture.