JoSDC'24

المسابقة الوطنية لتصميم الشرائح الإلكترونية

Jordan National Semiconductors Design Competition

Innovation Phase Report - JoSDC'2024

Team Name		
RISK-V		
Team Members		
#	Name	E-mail
1	Mohammed Amra	mohammadamra00@gmail.com
2	Yazeed Kamel	yazed.ka51@gmail.com
3	Ahmad Yasin	ahmadmoner1968@gmail.com
4	Qusai Abu-Hosher	qabuhosher@gmail.com
5	Lara Hisham	larahishamahmadby@gmail.com

Jordan National Semiconductor Design Competition (JoSDC'2024)

February - 2025

By

Mohammad Amra
Computer Engineering
mohammadamra00@gmail.com

Yazeed Kamel
Computer Engineering
yazed.ka51@gmail.com

Ahmad Yasin
Computer Engineering
ahmadmoner1968@gmail.com

Qusai abu-Hosher
Computer Engineering
qabuhosher@gmail.com

Lara Hisham
Computer Engineering
larahishamahmadby@gmail.com

1 Acknowledgments

We extend our profound gratitude to Prof.Bassam Jamil for his invaluable guidance and unwavering support throughout the duration of this semiconductors design competition. His insightful feedback and mentorship have significantly contributed to the successful implementation of our design.

We also wish to acknowledge Dr.Ruba A. Alkhasawneh and Prof.Dheya'a Mustafa for their essential role in providing training assistance for the project. Their expertise and guidance in practical training were indispensable, enabling us to acquire the necessary skills and knowledge crucial for the successful completion of the FPGA design. We are deeply appreciative of their dedication and support throughout the training process.

Finally, we express our sincere gratitude to the Computer Engineering Department at Hashemite University and the Computer Engineering Department at Jordan University of Science and Technology. Their provision of essential infrastructure, practical training, and necessary hardware greatly facilitated the execution of this work. The departments' commitment to fostering a conducive environment for learning and research has significantly influenced the overall success of this project. We are grateful for their continuous support and contributions to our academic and practical growth in the field of computer engineering.

2 Abstract

This project presents the design and implementation of a two-way superscalar processor in Verilog HDL, advancing beyond traditional single-issue architectures by enabling the parallel execution of two instructions per clock cycle. Built on a five-stage pipeline—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB)—the processor incorporates key improvements such as dual-instruction fetching via a dual-port memory, dual ALUs for parallel execution, and advanced branch prediction and forwarding units to minimize stalls and flushes. Enhanced components, including a multi-ported register file and sophisticated issue logic, optimize resource utilization and handle dependencies effectively. A comprehensive ecosystem is proposed, featuring a MIPS-compatible assembler for dual-issue instruction encoding, an exception handler for robust fault management, and a Python-based cycle-accurate simulator for validation and performance analysis. Functional testing of updated units (Register File, Issue Logic, ALU, Pipeline Registers) achieved high coverage, validating all procedures and corner cases. Performance evaluations reveal a CPI of 0.5 in most cases, reflecting near-ideal throughput, with stalls occasionally raising it to 0.5–0.7. Benchmark comparisons demonstrate significant improvements over single-cycle (CPI = 1.0) and five-stage pipelined designs ($CPI \approx$1.16–1.34), with reduction in various execution times. These results underscore the processor's superior efficiency, positioning it as a high-performance solution for modern computing demands and a platform for future enhancements.

3 Contents

3.1 Design Explanation

3.1.1 Motivation

In this world everyone wants to get their things/works done fasts. Isn't it ? From Cars to industrial to household machines everyone wants them to work faster. Do you know what's sitting inside these machines making them to work ? They are processors. They may be micro or macro processors depending on the functionality. The basic processor in general executes one instructions per clock cycle. In way to improve their processing speed so that the machines can improve their speed came into being is, the superscalar processor which has pipelining algorithm to enable it to execute two instructions per clock cycle. It was first invented by Seymour Cray's $CDC6600$ invented in 1964 and was later enhanced by Tjaden & Flynn in 1970. The first commercial single-chip superscalar microprocessor $MC88100$ was developed by Motorola in 1988, later Intel introduced its version $I960CA$ in 1989 & the $AMD29000-series$ 29050 in 1990. At present, the typical superscalar processor used is the Intel Core $i7$ processor depending on the Nehalem microarchitecture. Even though, the implementations of superscalar are heading toward enhancing complexity. The design of these processors normally refers to a set of methods that permit the CPU of a computer to attain a throughput of above one instruction for each cycle while executing a single sequential program. Let's further see in this article the Superscalar processor architecture which reduces its execution time and its applications.

3.1.2 What is Superscalar Processor?

A type of microprocessor that is used to implement a type of parallelism known as instruction-level parallelism in a single processor to execute more than one instruction during a CLK cycle by dispatching simultaneously various instructions to special execution units on the processor. **A scalar processor** executes single instruction for each clock cycle; a superscalar processor can execute more than one instruction during a clock cycle. The design techniques of superscalar normally comprise parallel register renaming, parallel instruction decoding, out-of-order executions & speculative execution. So these methods are normally used with complementing design methods like pipelining, branch prediction, caching & multi-core within current designs of microprocessors.

3.1.3 Features

The features of superscalar processors include the following.

- Superscalar architecture is a parallel computing technique utilized in various processors.
- In a superscalar computer, the CPU manages several instruction pipelines to perform numerous instructions simultaneously during a clock cycle.
- Superscalar architectures include all pipelining features although there are several instructions executing simultaneously within the same pipeline.
- Superscalar design methods normally comprise parallel register renaming, parallel instruction decoding, speculative execution & out-of-order execution. So, these methods are normally used with

complementing design methods like caching, pipelining, branch prediction & multi-core in recent microprocessor designs.

3.1.4 Superscalar Processor Architecture

We know that a superscalar processor is a CPU that executes above one instruction for each CLK cycle because processing speeds are simply measured in CLK cycles for each second. Compared to a scalar processor, this processor is very faster. Superscalar processor architecture mainly includes parallel execution units where these units can implement instructions simultaneously. So first, this parallel architecture was implemented within a RISC processor that utilizes simple & short instructions to execute calculations. So due to their superscalar abilities, normally RISC processors have performed better as compared to CISC processors which run at the same megahertz. But, most CISC processors now like the Intel Pentium comprise some RISC architecture also, which allows them to perform instructions in parallel.

3.1.5 Our Design

High-Level Design Overview: Two-Way Superscalar Processor
The TwoWaysTopModule represents a two-way superscalar processor designed to enhance performance by executing two instructions simultaneously whenever possible. It builds on the classic five-stage pipeline-Instruction Fetch, Instruction Decode, Execution, Memory Access, and Writeback—adapted for parallel processing. The processor leverages two distinct execution paths, referred to as ”Way 0” and ”Way 1,” to handle instructions in a single clock cycle, effectively doubling throughput under optimal conditions.

Core Purpose
The primary goal is to maximize instruction-level parallelism while maintaining correctness in the presence of data dependencies, control flow changes, and resource constraints. It achieves this by fetching two instructions per cycle, intelligently assigning them to execution paths based on their type and dependencies, and resolving potential conflicts through advanced prediction and hazard management techniques.

Architectural Features

1. **Dual Execution Paths (Superscalar Design):**

   - The processor operates with two parallel pipelines, allowing two instructions to progress through the stages concurrently.
   - Way 0 is not prioritized for any type of instructions, can handel all types.
   - Way 1 can not handle control relater instructions (e.g., branches, jumps, loads, and stores)

2. **Five-Stage Pipeline:**

    - **Instruction Fetch:** Retrieves two instructions from memory in each cycle, predicting the next program location to maintain a continuous flow.
    - **Instruction Decode:** Interprets both instructions, determines their resource needs, and prepares operands by accessing the register file.
    - **Execution:** Performs computations or address calculations for both instructions, managing control flow decisions like branches.
    - **Memory Access:** Handles data reads or writes to memory for instructions requiring it, such as loads and stores.
    - **Writeback:** Commits results back to the register file, ensuring both execution paths can complete their operations.

3. **Dynamic Instruction Assignment:**

    - Instructions are assigned to Way 0 or Way 1 based on their type and the current state of the processor. Control instructions naturally gravitate toward Way 0 due to their impact on program flow, while Way 1 picks up additional instructions to maximize parallelism.
    - The assignment considers the ”age” of instructions (based on their program counter values) to prioritize older instructions when necessary.

Key Functional Mechanisms

1. **Parallel Instruction Fetching:**

    - The processor fetches two instructions per cycle, using a single program counter that advances sequentially unless altered by control flow instructions.
    - It calculates potential next addresses (e.g., sequential, branch targets, or jump destinations) to ensure the correct instructions are fetched.

2. **Dependency and Hazard Management:**

    - **Data Dependencies:** The design resolves conflicts where one instruction’s result is needed by another (e.g., read-after-write hazards) by forwarding results from later stages to earlier ones, avoiding stalls when possible.
    - **Control Hazards:** Branch and jump instructions are predicted to minimize delays, with mechanisms to correct the pipeline if predictions fail.

- **Resource Conflicts:** It detects and prevents write-after-write hazards by stalling or reordering instructions when both paths attempt to write to the same register.

3. **Branch Prediction:**

    - A dynamic prediction system anticipates branch outcomes based on historical behavior, reducing the penalty of mispredictions by pre-fetching likely paths.
    - When predictions are incorrect, the pipeline adjusts by flushing incorrect instructions and redirecting to the correct path.

Adding the Ghare branch prediction unit has improved the performance of the design, despite the increase in the period, but it decreases the cycles lost due to misprediction, which is better overall and especially for larger benchmarks with more branches.
The table below shows how the 5-stage pipelined (which is when the Ghare was developed) design performs when it uses the Ghare unit:

- It calculates the numbers of branches, false prediction, and the false percentage for each benchmark.
- It also counts the number of instructions executed and the clock cycles, as well as the number of flushes and stalls needed when performing the benchmarks.
- Note that despite the high false percentage, the branch prediction unit performs well when there are more branches (which is the case in real-time codes).
- When executing more branches, the number of the false predictions and flushes and stalls stays the same, while increasing the number of instructions and branches. In other words, the unit memorizes the pattern (which had some false predictions the first time).
- This can lead to more than 90% accuracy.

Benchmark	**# Branch**	**# False**	**False %**	**Inst. Count**	**# Clk**	**# Flush**	**# NOPs**	**EXEC TIME**
Binary_Search	14	3	21.43%	50	54	3	42	1053
Max, Min	27	11	40.74%	85	115	3	96	2242.5
Multiplication using addition	3	2	66.67%	16	26	3	37	680
Remove_Duplicates	147	27	18.37%	542	708	27	349	18335
Scaler Mul	94	25	26.60%	415	530	38	385	8580
Selection Sort	820	777	9.39%	3625	3630	77	2666	70785
Swapping	0	0	0%	22	22	0	20	448.5
Sparse_Matrix	43	15	34.88%	178	200	12	110	3568.5
Insertion Sort	29	17	58.62%	233	235	17	170	4972.5

Table 1: Super Scaler Performance Table - With branchpredector

4. **Register File Access:**

    - A dual-port register file supports simultaneous reads and writes for both execution paths, ensuring operands are available and results are stored efficiently.
    - A dual-port register file supports simultaneous reads and writes for both execution paths, ensuring operands are available and results are stored efficiently.

5. **Memory Operations:**

    - Memory-intensive instructions (loads and stores) are executed with prioritized access to a shared memory unit, maintaining data consistency across both paths.

Operational Flow

- **Cycle Start:** Two instructions are fetched based on the current program counter.
- **Instruction Sorting:** The processor evaluates the instructions' types and dependencies, assigning them to Way 0 or Way 1. Control instructions typically go to Way 0, while others may go to Way 1 if resources allow.
- **Pipeline Progression:** Both instructions move through decoding, execution, memory access, and writeback in lockstep, with results from one stage feeding into the next.
- **Conflict Resolution:** If hazards arise, the processor either forwards data, stalls one path, or flushes and restarts based on branch outcomes.
- **Completion:** Results are written back, and the program counter updates to fetch the next pair of instructions.

Design Goals and Benefits

1. **Performance:**

    - By executing two instructions per cycle, the processor achieves higher throughput compared to a single-issue design, especially for independent instruction sequences.

2. **Flexibility:**

    - The prioritization of Way 0 for control instructions ensures efficient handling of program flow, while Way 1 boosts performance for parallelizable workloads.

3. **Reliability:**

    - Robust hazard detection and correction mechanisms maintain program correctness, even under complex dependency scenarios.

4. **Scalability:**

    - The architecture could be extended with additional ways or enhanced prediction, building on its modular pipeline structure.

Abstraction Summary Imagine the TwoWaysTopModule as a dual-lane highway for instructions. Each lane (Way 0 and Way 1) carries an instruction through five towns (pipeline stages), with traffic controllers (hazard and prediction systems) ensuring smooth flow. Way 0 is reserved for priority vehicles (control instructions), while Way 1 handles additional traffic when the road is clear. The system dynamically adjusts lanes, reroutes traffic when needed, and ensures both lanes reach their destination (writeback) efficiently, delivering two completed instructions per cycle to the outside world.

This high-level view encapsulates the processor's essence: a parallel, pipelined architecture designed for performance, balanced with sophisticated control to handle the complexities of modern instruction sets.

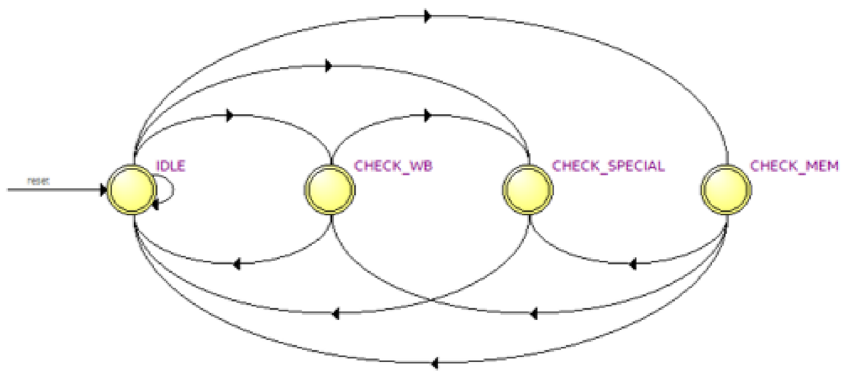


Figure 1: Forwarding unit FSM

3.2 Innovation and improvements

To improve the 5-stage pipelined design, there should be some fixes beforehand:

1. **Some hazard cases were not covered completely in the design, which causes more flush, and more lost cycles.**

2. **Adding a branch prediction unit:**

Branches are one of the most problematic issues when designing a processor. Going from the pipelined design and above through super-scalar or multicore design, processors start to suffer from severe branch penalty which occurs due to fetching and executing a branch instruction which turns out to be not taken after a few cycles, causing the penalty of flushing the pipeline. To deal with the high penalty, this design includes a Tournament branch prediction unit, which includes more than one predictor aiming for maximum accuracy. The predictors are listed as follows:

- **Global predictor:** This predictor focuses on more than one branch inside the code, it requires the following hardware:

    1. **Global history register (GHR):** Which stores the history of these branches, and it works as a shift register which shifts in the latest branches. Its size can be determined based on the size of the codes that will be executed using the processor, to balance between reasonable hardware and accuracy, the GHR records the last five branches.
    2. **Global pattern history table (GHPT):** This table stores predictions based on patterns of the history; it is indexed using the value of GHR XORed with the corresponding LSBs of the PC which are the least significant five bits(this is called a (Gshare) predictor, which prevents collisions caused by using only the PC as an index). Having an index of 5 bits requires 32 entries 2^5. Each entry stores a 2-bit state which corresponds to a prediction (weakly taken, strongly taken, and so on.
    3. **Branch target buffer (BTB):** This buffer is used to store target addresses for predicted branches. It is indexed by the PC, and it has 32 entries. The width of the buffer is 60 bit which consists of 1 bit to indicate the validity of the entry, the tag bits which the bits from the PC other than the index of the BTB ($32 – log32 = 27$bits), and the branch target (same as caches).

- **Local predictor:** This predictor works per branch unlike the global predictor. It is used on the observation that branches have repetitive patterns. The local predictor consists of:

    1. **Local history register (LHR):** Like GHR, this register records the history of branches, but in this case, it works for a single branch at a time. Its size is 16 entries, and each has a width of 4 bits. The functionality works a little bit differently, as each entry store history for a specific branch.
    2. **Local history pattern table (LHPT):** This is very similar to GHPT in terms of functionality, but it is indexed by the LHR thus, the index is4 bits and there are 16 entries as well.

3. **Branch target buffer (BTB):** This works exactly like the one for the global predictor, with small differences such as the width is 4 bits, the size is 16 entries, and the tag size was chosen here instead of being calculated because the buffer is much smaller than the global buffer. Since the risk of collision is reduced, the tag size was chosen to be 28 bits. The resulting BTB width is 60 bits $(1 + 27 + 32)$.

The local predictor is a 2-level predictor, since the LHPT is indexed by the LHR, which is indexed by bits of the PC.

- **Meta predictor:** This predictor has the duty of determining which predictor was better, and based on that, which one will predict the next branch. It has a similar logic to previous predictors, containing a choice table and each entry has a value that chooses either predictor (also works as saturating counter). The index of the table is the same as the global index, because it is bigger than the local, which can cover both.

- **Dynamic predictor:** Since the above predictors need to fill the history before operating accurately, there needs to be a strategy until then. And if the history size is small compared to the number of branches for a certain code, the design can rely on a 2-bit predictor. It predicts depending on the current state which is changed as a saturating counter. (when the state is strongly taken for example, it needs two consecutive wrong predictions to change the prediction).

  But since the design is relatively small compared to processors nowadays, the tournament predictor adds a very big logic(caused by all the tables mentioned above, and all the signals the needed to be propagated through the pipes because the BPU is in the IF stage and the branch compare unit is in the EX stage) which reduces the performance significantly and using a dynamic predictor and the G-share predictors perform like the tournament especially for normal codes which this processor is designed to perform.

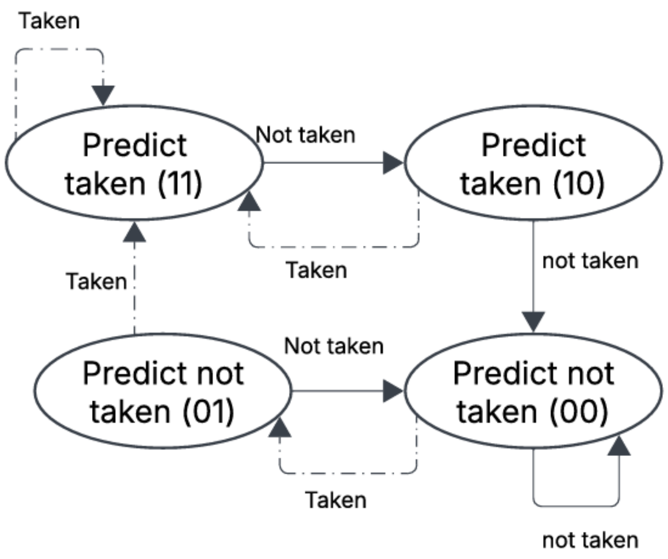


Figure 2: Dynamic predictor FSM

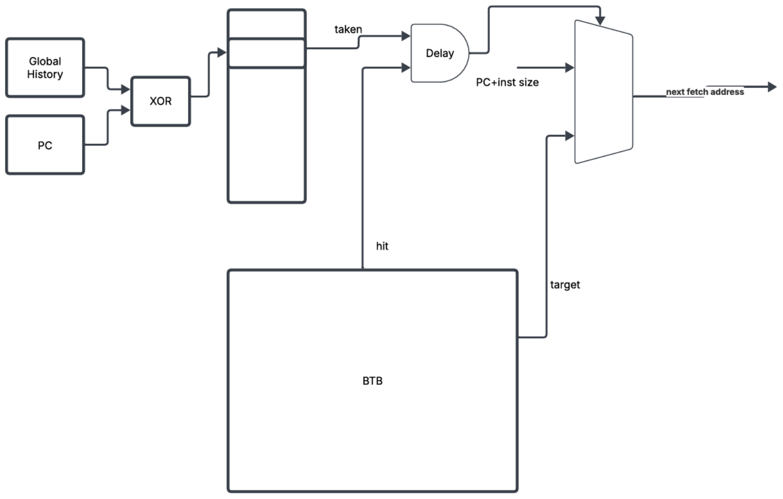


Figure 3: Gshare-style Branch Predictor with a Branch Target Buffer (BTB)

- **Adding an extra Logic (issuing logic IL):** In the timing analysis, the critical path could not be solved, which led to the idea of adding the IS stage. Distributing the logic of five stages to six should resolve the critical path since there is not much logic in any stage the causes units to wait for each other. But due to the extra complexity of the hazard detection and forwarding unit (which now needs to consider an extra stage in the comparisons), the logic became harder to create and understand. Also, the flush penalty is now three cycles instead of two, which is a significant increase that causes more losses than benefits, especially in this small design that deals with small codes. After adding all these reasons up, it turns out that the five-stage pipeline is a better choice.

After fixing the original design, now it needs to be enhanced to the next level, which is the in-order super scalar design. super scalar processors in general consist of more than one pipeline; in this case it is convenient to have a two-way super scalar processor. The design is a two-way hybrid asymmetric processor, where

1. **Way zero:** Which is dedicated for all instructions especially jumps and branches, as well as memory access **(LW, SW, BEQ, BNE, J, JR, JAL, ADD, SUB, AND, OR, XOR, ADDI, SUBI, ANDI, ORI, XORI, NOR, NAND, XNOR, SLL, SRL, SLT, SGT)**.
   - This way has its own control unit, which which is almost the same as the original control unit, this control unit is responsible for those instructions mentioned above.
   - This way contains the data memory where all the memory access operations are done in way zero.
   - This way contains an ALU, and a shifter since it performs every instruction.
   - It also contains the **branch prediction unit (BPU)** and the **branch-compare (BC)**, along with all the muxes responsible for jumps and branches.
   - Adders like PC adder, jump adder, and branch adder are only in this way.
2. **Way one:** Performs all integer instructions which are the ALU instructions **(ADD, SUB, AND, OR, XOR, ADDI, SUBI, ANDI, ORI, XORI, NOR, NAND, XNOR, SLL, SRL, SLT, SGT).**
   - The control unit here produces signals that control the instructions that have been mentioned above.
   - It does not have access to the data memory which makes no need to add more read and write ports.
   - This way consists of the full ALU and the shifter which perform all integer arithmetic and logic operations.

Both ways have access to the register file, so there are four read ports and two write ports. And the same thing applies to the instruction memory so now it has two read ports and no write ports since it is a ROM. Hazard detection and forwarding units are shared between the two ways as well.

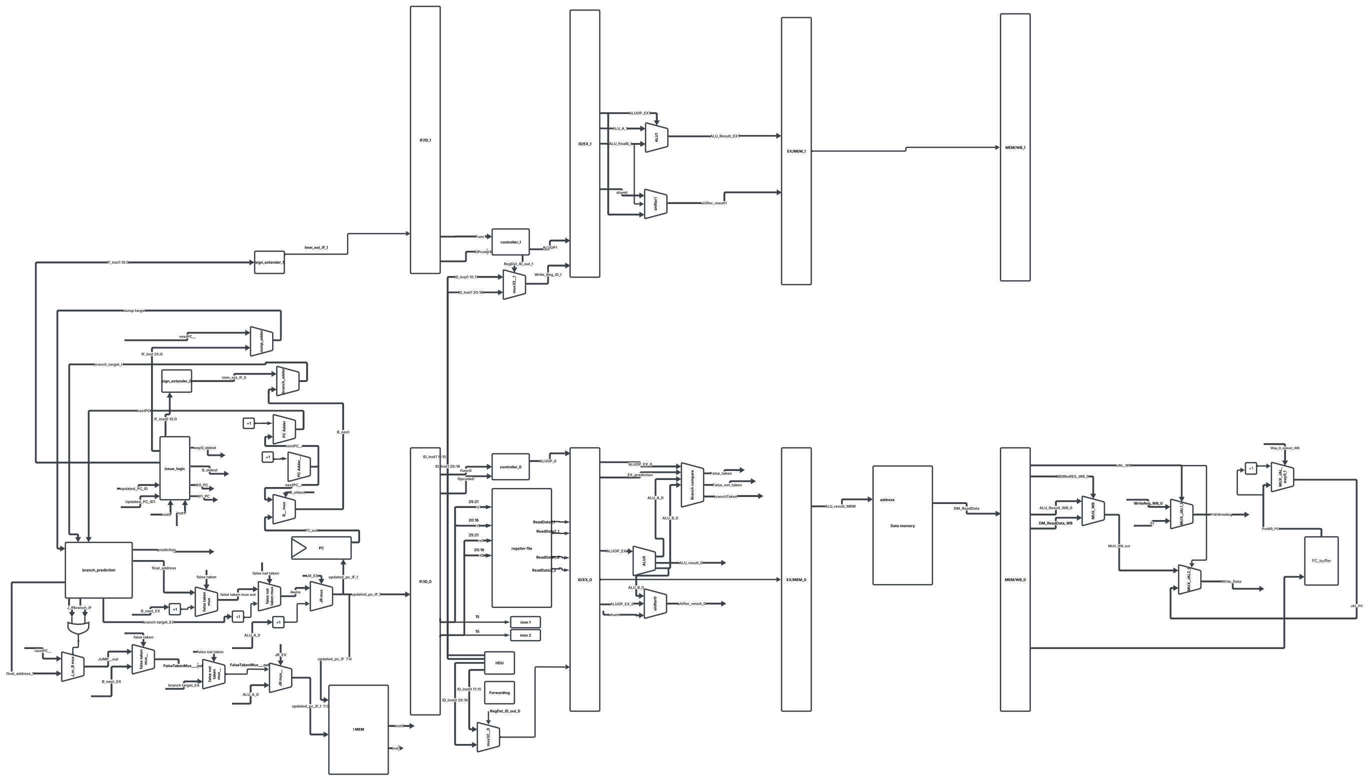


Figure 4: Superscalar Pipelined processor with Branch Prediction Data path

3.3 Testing Plan and Results

As a part of testing plan functional unit coverage was conducted to monitor the behavior of the units on the pre-defined procedures covering all cases that are included in the module and some corner cases that could occur in the design during the execution. The following are the results of unit functional coverage for the units we adjusted from the previous phase showing high score in covering all cases including corner cases.

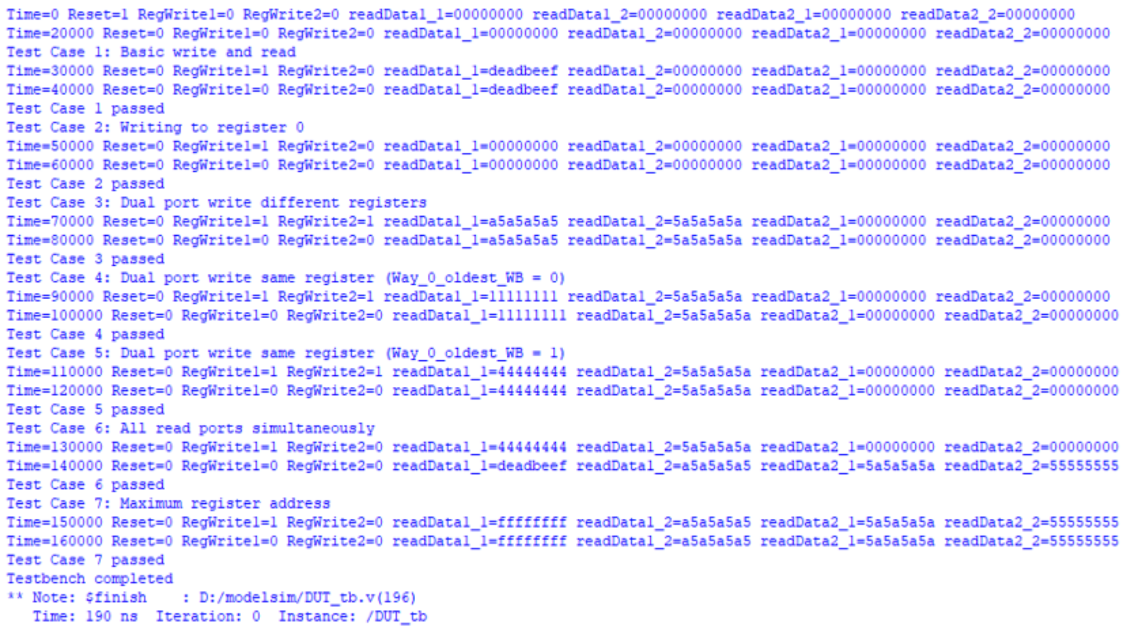
1. **Register File:**

   - **Test Case 1: Basic Write and Read**
     - **Purpose:** Verify the fundamental functionality of writing data to a register and reading it back.
     - **Steps:**
       (a) Enable the write operation.
       (b) Write data to a specific register.
       (c) Disable the write operation.
       (d) Read the data from the same register.
     - **Expected Outcome:** The read data should match the data written to the register.
   - **Test Case 2: Writing to Register 0 (Should Not Write)**
     - **Purpose:** Ensure that writing to register 0 (typically a hardwired zero register) does not alter its value.
     - **Steps:**
       (a) Enable the write operation.
       (b) Attempt to write data to register 0.
       (c) Disable the write operation.
       (d) Read the data from the register 0.
     - **Expected Outcome:** The value in register 0 should remain unchanged (typically zero).
   - **Test Case 3: Dual Port Write to Different Registers**
     - **Purpose:** Verify the ability to write to two different registers simultaneously using dual write ports.
     - **Steps:**
       (a) Enable both write ports.
       (b) Write data to two distinct registers simultaneously.
       (c) Disable the write port.
       (d) Read the values from both registers.
     - **Expected Outcome:** The read data from each register should match the data written to it.

- **Test Case 4: Dual Port Write to Same Register (Priority to First Port)**
  - **Purpose:** Test the behavior when both write ports attempt to write to the same register, with priority given to the first write port.
  - **Steps:**
    (a) Enable both write ports.
    (b) Write different data to the same register using both ports.
    (c) Set the priority to the first write port.
    (d) Disable the write ports.
    (e) Read the value from the register.
  - **Expected Outcome:** The read data should match the data written by the first write port.
- **Test Case 5: Dual Port Write to Same Register (Priority to Second Port)**
  - **Purpose:** Test the behavior when both write ports attempt to write to the same register, with priority given to the second write port.
  - **Steps:**
    (a) Enable both write ports.
    (b) Write different data to the same register using both ports.
    (c) Set the priority to the second write port.
    (d) Disable the write ports.
    (e) Read the value from the register.
  - **Expected Outcome:** The read data should match the data written by the second write port.
- **Test Case 6: All Read Ports Simultaneously**
  - **Purpose:** Verify that all read ports can operate simultaneously and return the correct values from different registers.
  - **Steps:**
    (a) Write data to multiple registers.
    (b) Disable the write operation.
    (c) Read data from multiple registers simultaneously using all read ports.
  - **Expected Outcome:** The read data from each port should match the data written to the corresponding registers.
- **Test Case 7: Edge Case - Maximum Register Address**
  - **Purpose:** Test the ability to write to and read from the highest addressable register (edge case).
  - **Steps:**
    (a) Enable the write ports.

(b) Write different data to the highest addressable register.
(c) Disable the write ports.
(d) Read the value from the highest addressable register.

– **Expected Outcome:** The read data should match the data written to the highest addressable register.



```
Time=0 Reset=1 RegWrite1=0 RegWrite2=0 readData1_1=00000000 readData1_2=00000000 readData2_1=00000000 readData2_2=00000000
Time=20000 Reset=0 RegWrite1=0 RegWrite2=0 readData1_1=00000000 readData1_2=00000000 readData2_1=00000000 readData2_2=00000000
Test Case 1: Basic write and read
Time=30000 Reset=0 RegWrite1=1 RegWrite2=0 readData1_1=deadbeef readData1_2=00000000 readData2_1=00000000 readData2_2=00000000
Time=40000 Reset=0 RegWrite1=0 RegWrite2=0 readData1_1=deadbeef readData1_2=00000000 readData2_1=00000000 readData2_2=00000000
Test Case 1 passed
Test Case 2: Writing to register 0
Time=50000 Reset=0 RegWrite1=1 RegWrite2=0 readData1_1=00000000 readData1_2=00000000 readData2_1=00000000 readData2_2=00000000
Time=60000 Reset=0 RegWrite1=0 RegWrite2=0 readData1_1=00000000 readData1_2=00000000 readData2_1=00000000 readData2_2=00000000
Test Case 2 passed
Test Case 3: Dual port write different registers
Time=70000 Reset=0 RegWrite1=1 RegWrite2=1 readData1_1=a5a5a5a5 readData1_2=5a5a5a5a readData2_1=00000000 readData2_2=00000000
Time=80000 Reset=0 RegWrite1=0 RegWrite2=0 readData1_1=a5a5a5a5 readData1_2=5a5a5a5a readData2_1=00000000 readData2_2=00000000
Test Case 3 passed
Test Case 4: Dual port write same register (Way_0_oldest_WB = 0)
Time=90000 Reset=0 RegWrite1=1 RegWrite2=1 readData1_1=11111111 readData1_2=5a5a5a5a readData2_1=00000000 readData2_2=00000000
Time=100000 Reset=0 RegWrite1=0 RegWrite2=0 readData1_1=11111111 readData1_2=5a5a5a5a readData2_1=00000000 readData2_2=00000000
Test Case 4 passed
Test Case 5: Dual port write same register (Way_0_oldest_WB = 1)
Time=110000 Reset=0 RegWrite1=1 RegWrite2=1 readData1_1=44444444 readData1_2=5a5a5a5a readData2_1=00000000 readData2_2=00000000
Time=120000 Reset=0 RegWrite1=0 RegWrite2=0 readData1_1=44444444 readData1_2=5a5a5a5a readData2_1=00000000 readData2_2=00000000
Test Case 5 passed
Test Case 6: All read ports simultaneously
Time=130000 Reset=0 RegWrite1=1 RegWrite2=0 readData1_1=44444444 readData1_2=5a5a5a5a readData2_1=00000000 readData2_2=00000000
Time=140000 Reset=0 RegWrite1=0 RegWrite2=0 readData1_1=deadbeef readData1_2=a5a5a5a5 readData2_1=5a5a5a5a readData2_2=55555555
Test Case 6 passed
Test Case 7: Maximum register address
Time=150000 Reset=0 RegWrite1=1 RegWrite2=0 readData1_1=ffffffff readData1_2=a5a5a5a5 readData2_1=5a5a5a5a readData2_2=55555555
Time=160000 Reset=0 RegWrite1=0 RegWrite2=0 readData1_1=ffffffff readData1_2=a5a5a5a5 readData2_1=5a5a5a5a readData2_2=55555555
Test Case 7 passed
Testbench completed
** Note: $finish    : D:/modelsim/DUT_tb.v(196)
   Time: 190 ns  Iteration: 0  Instance: /DUT_tb
```

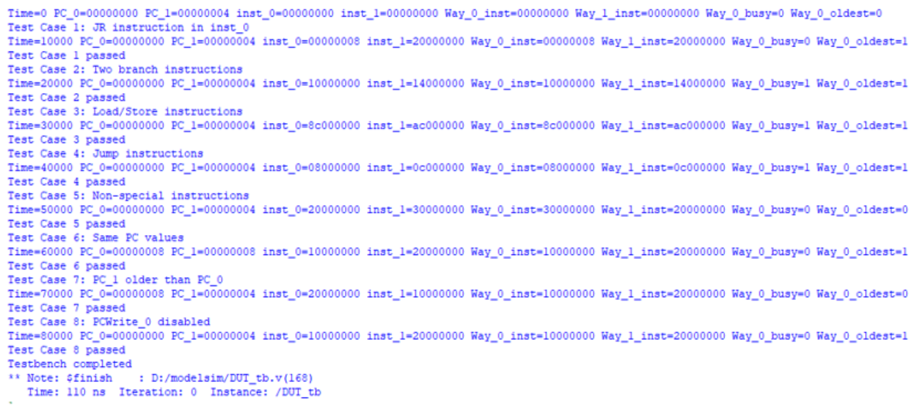
Figure 5: Registers-File coverage

2. **Issue Logic:**

- **Test Case 1: JR Instruction in inst$_0$**
  - **Purpose:** Verify the handling of a JR (Jump Register) instruction in the first instruction slot (inst_0).
  - **Logic:**
    * A JR instruction is a special case that typically requires specific handling (e.g., it may not be issued immediately or may have priority).
    * The test ensures that the logic correctly identifies and processes the JR instruction while handling a non-special instruction in the second slot (inst_1).
- **Test Case 2: Two Branch Instructions**
  - **Purpose:** Test the handling of two branch instructions (e.g., BEQ and BNE) in both instruction slots.
  - **Logic:**
    * Branch instructions often have higher priority or special handling due to their impact on control flow.
    * The test ensures that the logic correctly routes both branch instructions and sets the appropriate busy and priority flags.
- **Test Case 3: Load/Store Instructions**
  - **Purpose:** Verify the handling of load (LW) and store (SW) instructions in the instruction slots.
  - **Logic:**
    * Load and store instructions typically involve memory access and may have specific timing or priority requirements.
    * The test ensures that the logic correctly processes these instructions and sets the appropriate busy and priority flags.
- **Test Case 4: Jump Instructions**
  - **Purpose:** Test the handling of **jump (J) and jump and link (JAL)** instructions.
  - **Logic:**
    * Jump instructions alter the program counter and require special handling.
    * The test ensures that the logic correctly routes these instructions and sets the appropriate flags.
- **Test Case 5: Non-Special Instructions**
  - **Purpose:** Verify the handling of non-special instructions (e.g., arithmetic or logical operations) in both instruction slots.
  - **Logic:**
    * Non-special instructions are typically simpler and do not require special handling.

    * The test ensures that the logic correctly routes these instructions and sets the appropriate flags based on their program counters.

- **Test Case 6: Same Program Counter Values**
  - **Purpose:** Test the behavior when both instructions have the same program counter value (a corner case).
  - **Logic:**
    * When both instructions have the same program counter, the logic must resolve which instruction is older or has priority.
    * The test ensures that the logic correctly handles this scenario and sets the appropriate flags.
- **Test Case 7: Older Program Counter in inst_**1
  - **Purpose:** Verify the behavior when the program counter of inst_1 is older than that of inst_0.
  - **Logic:**
    * The logic should prioritize the older instruction (based on program counter values).
    * The logic should prioritize the older instruction (based on program counter values).
- **Test Case 8: : Disabled PCWrite_**0
  - **Purpose:** Test the behavior when the PCWrite_0 signal is disabled.
  - **Logic:**
    * Disabling PCWrite_0 simulates a scenario where the first instruction slot is not being written.
    * The test ensures that the logic correctly handles this case and routes instructions appropriately.



```
Time=0 PC_0=00000000 PC_1=00000004 inst_0=00000000 inst_1=00000000 Way_0_inst=00000000 Way_1_inst=00000000 Way_0_busy=0 Way_0_oldest=0
Test Case 1: JR instruction in inst_0
Time=10000 PC_0=00000000 PC_1=00000004 inst_0=00000008 inst_1=20000000 Way_0_inst=00000008 Way_1_inst=20000000 Way_0_busy=0 Way_0_oldest=1
Test Case 1 passed
Test Case 2: Two branch instructions
Time=20000 PC_0=00000000 PC_1=00000004 inst_0=10000000 inst_1=14000000 Way_0_inst=10000000 Way_1_inst=14000000 Way_0_busy=1 Way_0_oldest=1
Test Case 2 passed
Test Case 3: Load/Store instructions
Time=30000 PC_0=00000000 PC_1=00000004 inst_0=8c000000 inst_1=ac000000 Way_0_inst=8c000000 Way_1_inst=ac000000 Way_0_busy=1 Way_0_oldest=1
Test Case 3 passed
Test Case 4: Jump instructions
Time=40000 PC_0=00000000 PC_1=00000004 inst_0=08000000 inst_1=0c000000 Way_0_inst=08000000 Way_1_inst=0c000000 Way_0_busy=1 Way_0_oldest=1
Test Case 4 passed
Test Case 5: Non-special instructions
Time=50000 PC_0=00000000 PC_1=00000004 inst_0=20000000 inst_1=30000000 Way_0_inst=30000000 Way_1_inst=20000000 Way_0_busy=0 Way_0_oldest=0
Test Case 5 passed
Test Case 6: Same PC values
Time=60000 PC_0=00000008 PC_1=00000008 inst_0=10000000 inst_1=20000000 Way_0_inst=10000000 Way_1_inst=20000000 Way_0_busy=0 Way_0_oldest=1
Test Case 6 passed
Test Case 7: PC_1 older than PC_0
Time=70000 PC_0=00000008 PC_1=00000004 inst_0=20000000 inst_1=10000000 Way_0_inst=10000000 Way_1_inst=20000000 Way_0_busy=0 Way_0_oldest=0
Test Case 7 passed
Test Case 8: PCWrite_0 disabled
Time=80000 PC_0=00000000 PC_1=00000004 inst_0=10000000 inst_1=20000000 Way_0_inst=10000000 Way_1_inst=20000000 Way_0_busy=0 Way_0_oldest=1
Test Case 8 passed
Testbench completed
** Note: $finish    : D:/modelsim/DUT_tb.v(168)
   Time: 110 ns  Iteration: 0  Instance: /DUT_tb
```

Figure 6: Issuing coverage

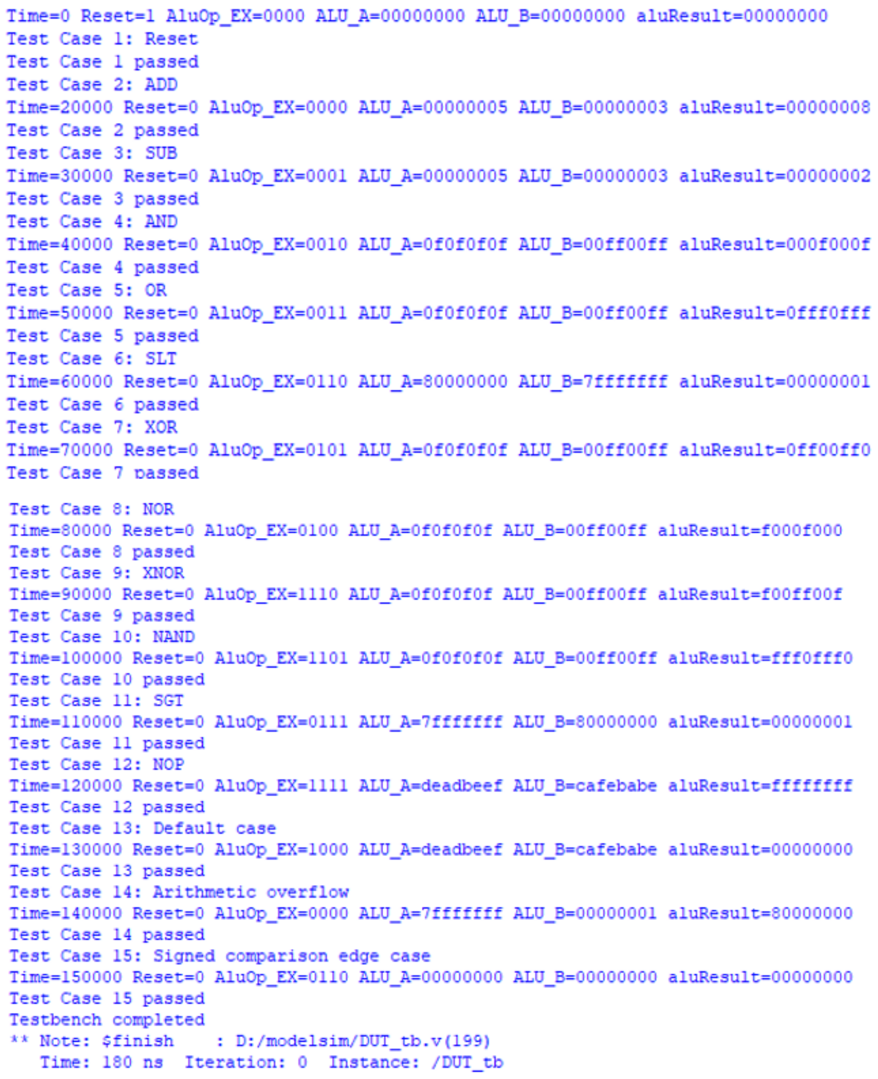
3. **ALU:**

   - **Test Case 1: Reset**
     - **Purpose:** Verify that the ALU resets correctly.
     - **Logic:**
       * When the reset signal is active, the ALU output should be in its default state (typically zero or a known value).
       * Ensures that the ALU initializes properly before performing operations.
   - **Test Case 2: ADD Operation**
     - **Purpose:** Test the addition functionality of the ALU.
     - **Logic:**
       * The ALU should correctly compute the sum of two input values.
       * Verifies basic arithmetic functionality.
   - **Test Case 3: SUB Operation**
     - **Purpose:** Test the subtraction functionality of the ALU.
     - **Logic:**
       * The ALU should correctly compute the difference between two input values.
       * Verifies basic arithmetic functionality.
   - **Test Case 4: AND Operation**
     - **Purpose:** Test the bitwise AND functionality of the ALU.
     - **Logic:**
       * The ALU should perform a bitwise AND operation on the two input values.
       * Verifies logical operation functionality.
   - **Test Case 5: OR Operation**
     - **Purpose:** Test the bitwise OR functionality of the ALU.
     - **Logic:**
       * The ALU should perform a bitwise OR operation on the two input values.
       * Verifies logical operation functionality.
   - **Test Case 6: SLT (Set Less Than) Operation**
     - **Purpose:** Test the signed comparison functionality of the ALU.
     - **Logic:**
       * The ALU should compare two signed values and return a result indicating whether the first value is less than the second.
       * Verifies signed comparison functionality.
   - **Test Case 7: XOR Operation**

  - **Purpose:** Test the bitwise XOR functionality of the ALU.
  - **Logic:**
    * The ALU should perform a bitwise XOR operation on the two input values.
    * Verifies logical operation functionality.
- **Test Case 8: NOR Operation**
  - **Purpose:** Test the bitwise NOR functionality of the ALU.
  - **Logic:**
    * The ALU should perform a bitwise NOR operation on the two input values.
    * Verifies logical operation functionality.
- **Test Case 9: XNOR Operation**
  - **Purpose:** Test the bitwise XNOR functionality of the ALU.
  - **Logic:**
    * The ALU should perform a bitwise XNOR operation on the two input values.
    * Verifies logical operation functionality.
- **Test Case 10: NAND Operation**
  - **Purpose:** Test the bitwise NAND functionality of the ALU.
  - **Logic:**
    * The ALU should perform a bitwise NAND operation on the two input values.
    * Verifies logical operation functionality.
- **Test Case 11: SGT (Set Greater Than) Operation**
  - **Purpose:** Test the signed comparison functionality of the ALU.
  - **Logic:**
    * The ALU should compare two signed values and return a result indicating whether the first value is greater than the second.
    * Verifies signed comparison functionality.
- **Test Case 12: NOP (No Operation)**
  - **Purpose:** Test the behavior of the ALU when no operation is specified.
  - **Logic:**
    * The ALU should output a default or no-operation value when the operation code is NOP.
    * Verifies handling of undefined or no-operation cases.
- **Test Case 13: Default Case**
  - **Purpose:** Test the behavior of the ALU for an undefined or invalid operation code.
  - **Logic:**

        * The ALU should handle undefined operation codes gracefully, typically by outputting a default value.
        * Verifies robustness of the ALU.

- **Test Case 14: Arithmetic Overflow**
    - **Purpose:** Test the ALU's handling of arithmetic overflow.
    - **Logic:**
        * The ALU should correctly compute the result when an arithmetic operation (e.g., addition) causes an overflow.
        * Verifies handling of edge cases in arithmetic operations.
- **Test Case 15: Signed Comparison Edge Case**
    - **Purpose:** Test the ALU's handling of edge cases in signed comparisons.
    - **Logic:**
        * The ALU should correctly compare two equal signed values and return the appropriate result.
        * Verifies edge-case handling in signed comparisons.



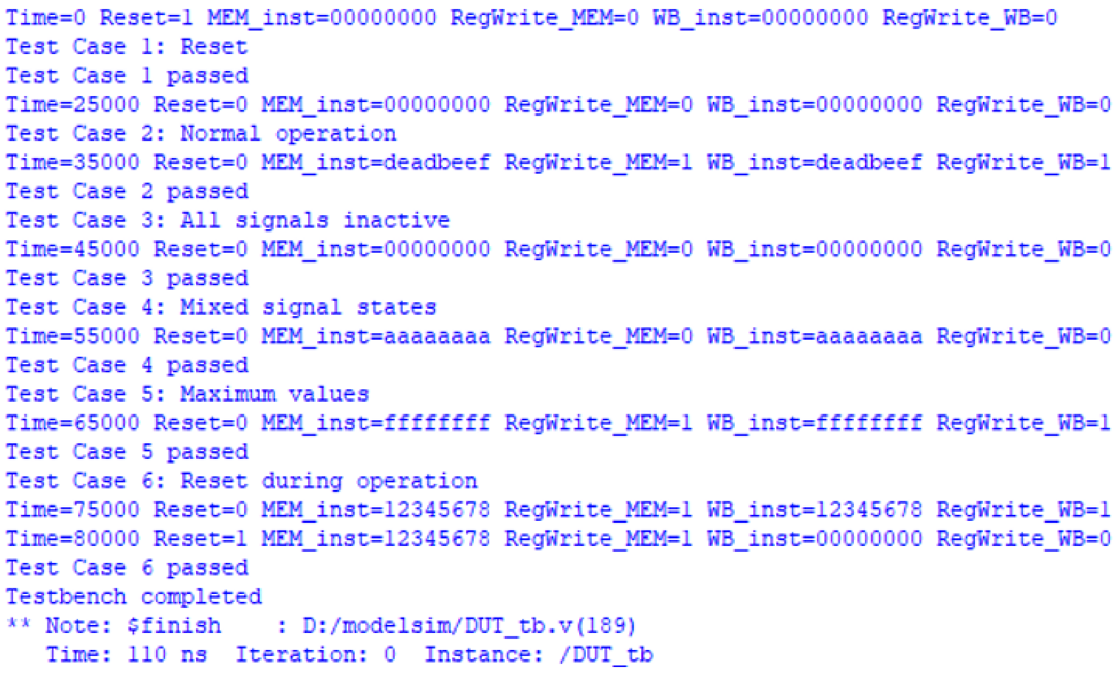
```
Time=0 Reset=1 AluOp_EX=0000 ALU_A=00000000 ALU_B=00000000 aluResult=00000000
Test Case 1: Reset
Test Case 1 passed
Test Case 2: ADD
Time=20000 Reset=0 AluOp_EX=0000 ALU_A=00000005 ALU_B=00000003 aluResult=00000008
Test Case 2 passed
Test Case 3: SUB
Time=30000 Reset=0 AluOp_EX=0001 ALU_A=00000005 ALU_B=00000003 aluResult=00000002
Test Case 3 passed
Test Case 4: AND
Time=40000 Reset=0 AluOp_EX=0010 ALU_A=0f0f0f0f ALU_B=00ff00ff aluResult=000f000f
Test Case 4 passed
Test Case 5: OR
Time=50000 Reset=0 AluOp_EX=0011 ALU_A=0f0f0f0f ALU_B=00ff00ff aluResult=0fff0fff
Test Case 5 passed
Test Case 6: SLT
Time=60000 Reset=0 AluOp_EX=0110 ALU_A=80000000 ALU_B=7fffffff aluResult=00000001
Test Case 6 passed
Test Case 7: XOR
Time=70000 Reset=0 AluOp_EX=0101 ALU_A=0f0f0f0f ALU_B=00ff00ff aluResult=0ff00ff0
Test Case 7 passed

Test Case 8: NOR
Time=80000 Reset=0 AluOp_EX=0100 ALU_A=0f0f0f0f ALU_B=00ff00ff aluResult=f000f000
Test Case 8 passed
Test Case 9: XNOR
Time=90000 Reset=0 AluOp_EX=1110 ALU_A=0f0f0f0f ALU_B=00ff00ff aluResult=f00ff00f
Test Case 9 passed
Test Case 10: NAND
Time=100000 Reset=0 AluOp_EX=1101 ALU_A=0f0f0f0f ALU_B=00ff00ff aluResult=fff0fff0
Test Case 10 passed
Test Case 11: SGT
Time=110000 Reset=0 AluOp_EX=0111 ALU_A=7fffffff ALU_B=80000000 aluResult=00000001
Test Case 11 passed
Test Case 12: NOP
Time=120000 Reset=0 AluOp_EX=1111 ALU_A=deadbeef ALU_B=cafebabe aluResult=ffffffff
Test Case 12 passed
Test Case 13: Default case
Time=130000 Reset=0 AluOp_EX=1000 ALU_A=deadbeef ALU_B=cafebabe aluResult=00000000
Test Case 13 passed
Test Case 14: Arithmetic overflow
Time=140000 Reset=0 AluOp_EX=0000 ALU_A=7fffffff ALU_B=00000001 aluResult=80000000
Test Case 14 passed
Test Case 15: Signed comparison edge case
Time=150000 Reset=0 AluOp_EX=0110 ALU_A=00000000 ALU_B=00000000 aluResult=00000000
Test Case 15 passed
Testbench completed
** Note: $finish    : D:/modelsim/DUT_tb.v(199)
   Time: 180 ns  Iteration: 0  Instance: /DUT_tb
```

Figure 7: ALU coverage

4. **Pipes:**

   - **Test Case 1: Reset**
     - **Purpose:** Verify that the pipeline register resets correctly.
     - **Logic:**
       * When the reset signal is active, all outputs should be in their default state (typically zero or a known value).
       * Ensures that the pipeline initializes properly before processing any data.
   - **Test Case 2: Normal Operation with All Signals Active**
     - **Purpose:** Test the pipeline's behavior when all input signals are active.
     - **Logic:**
       * The pipeline should correctly propagate all input signals (e.g., instruction, register write enable, read data, ALU result, program counter, etc.) to the output.
       * Verifies that the pipeline handles typical operation scenarios correctly.
   - **Test Case 3: All Signals Inactive**
     - **Purpose:** Test the pipeline's behavior when all input signals are inactive.
     - **Logic:**
       * The pipeline should propagate the inactive state of all signals to the output.
       * Verifies that the pipeline handles idle or no-operation scenarios correctly.
   - **Test Case 4: Mixed Signal States**
     - **Purpose:** Test the pipeline's behavior when some signals are active and others are inactive.
     - **Logic:**
       * The pipeline should correctly propagate the mixed states of the input signals to the output.
       * Verifies that the pipeline handles partial activity scenarios correctly.
   - **Test Case 5: Corner Case - Maximum Values**
     - **Purpose:** Test the pipeline's behavior when all input signals are at their maximum possible values.
     - **Logic:**
       * The pipeline should correctly propagate the maximum values of all input signals to the output.
       * Verifies that the pipeline handles edge cases involving maximum values correctly.
   - **Test Case 6: Reset During Operation**
     - **Purpose:** Test the pipeline's behavior when a reset occurs during normal operation.
     - **Logic:**

* When the reset signal is activated during operation, the pipeline should immediately reset all outputs to their default state.
* Verifies that the pipeline handles reset scenarios correctly, even during active operation.



```
Time=0 Reset=1 MEM_inst=00000000 RegWrite_MEM=0 WB_inst=00000000 RegWrite_WB=0
Test Case 1: Reset
Test Case 1 passed
Time=25000 Reset=0 MEM_inst=00000000 RegWrite_MEM=0 WB_inst=00000000 RegWrite_WB=0
Test Case 2: Normal operation
Time=35000 Reset=0 MEM_inst=deadbeef RegWrite_MEM=1 WB_inst=deadbeef RegWrite_WB=1
Test Case 2 passed
Test Case 3: All signals inactive
Time=45000 Reset=0 MEM_inst=00000000 RegWrite_MEM=0 WB_inst=00000000 RegWrite_WB=0
Test Case 3 passed
Test Case 4: Mixed signal states
Time=55000 Reset=0 MEM_inst=aaaaaaaa RegWrite_MEM=0 WB_inst=aaaaaaaa RegWrite_WB=0
Test Case 4 passed
Test Case 5: Maximum values
Time=65000 Reset=0 MEM_inst=ffffffff RegWrite_MEM=1 WB_inst=ffffffff RegWrite_WB=1
Test Case 5 passed
Test Case 6: Reset during operation
Time=75000 Reset=0 MEM_inst=12345678 RegWrite_MEM=1 WB_inst=12345678 RegWrite_WB=1
Time=80000 Reset=1 MEM_inst=12345678 RegWrite_MEM=1 WB_inst=00000000 RegWrite_WB=0
Test Case 6 passed
Testbench completed
** Note: $finish    : D:/modelsim/DUT_tb.v(189)
   Time: 110 ns  Iteration: 0  Instance: /DUT_tb
```

Figure 8: Pipes coverage

3.3.1 Provided Benchmarks

Benchmark	Inst. Count	# Clk	EXEC TIME
Binary_Search	50	54	1053
Max, Min	85	115	2242.5
Multiplication using addition	16	26	680
Remove_Duplicates	542	708	18335
Scaler Mul	415	530	8580
Selection Sort	3625	3630	70785
Swapping	22	22	448.5
Sparse_Matrix	178	200	3568.5
Insertion Sort	233	235	4972.5

Table 2: Provided Benchmarks

- **Binary_Search:** This benchmark, likely implementing a binary search algorithm, executes 50 instructions over 54 clock cycles with a modest 21.43% branch misprediction rate (3 out of 14 branches). With only 3 pipeline flushes and 42 NOPs, it maintains reasonable efficiency, resulting in an execution time of 1053 units. Its low instruction count suggests it's a simple, linear search operation, but branch mispredictions still introduce minor overhead.

- **Max_Min:** Used to find the maximum and minimum values in a dataset, this benchmark runs 85 instructions in 115 clock cycles. It experiences a higher branch misprediction rate of 40.74% (11 out of 27 branches), leading to 3 flushes and 96 NOPs, with an execution time of 2242.5 units. The significant misprediction rate indicates potential challenges in predicting control flow, impacting performance despite the moderate instruction count.

- **Multiplication using addition:** This case simulates multiplication through repeated addition, executing 16 instructions in 26 clock cycles. It has a high branch misprediction rate of 66.67% (2 out of 3 branches), but with only 3 flushes and 37 NOPs, the execution time is 680 units. The small scale and high misprediction rate suggest tight loops or conditional branches that are hard to predict, but the impact is minimal due to the low instruction count.

- **Remove_Duplicates:** Likely removing duplicate entries from a list, this benchmark processes 542 instructions in 708 clock cycles. It has a relatively low branch misprediction rate of 18.37% (27 out of 147 branches), but requires 27 flushes and 349 NOPs, leading to a high execution time of 18335 units. The large number of instructions and overhead from flushes and NOPs indicate a complex operation with frequent branches, challenging the superscalar design's efficiency.

- **Scaler Mul:** This benchmark, possibly scalar multiplication (e.g., in vector or matrix operations), executes 415 instructions in 530 clock cycles. With a 26.60% branch misprediction rate (25 out of 94 branches), it needs 38 flushes and 385 NOPs, resulting in an execution time of 8580 units. The moderate misprediction rate and significant overhead suggest a mix of predictable and unpredictable branches, impacting performance on a larger scale.

- **Selection Sort:** Implementing the selection sort algorithm, this benchmark is the most resource-intensive, executing 3625 instructions in 3630 clock cycles. Despite a low branch misprediction rate of 9.39% (777 out of (20 branches), it requires 77 flushes and 2666 NOPs, leading to a hefty execution time of 70785 units. The massive instruction count and branch activity highlight the algorithm's complexity, straining the superscalar design with frequent pipeline disruptions.

- **Swapping:** This simple benchmark, likely swapping two values, runs 22 instructions in 22 clock cycles with no branch instructions or mispredictions (0% false rate). It requires no flushes and only 20 NOPs, resulting in an execution time of 448.5 units. Its perfect efficiency reflects a straightforward, branch-free operation, making it an ideal case for the superscalar processor.

- **Sparse_Matrix:** Targeting operations on sparse matrices, this benchmark executes 178 instructions in 200 clock cycles. It has a 34.88% branch misprediction rate (15 out of 43 branches), with 12 flushes and 110 NOPs, leading to an execution time of 3568.5 units. The moderate overhead suggests a mix of predictable and unpredictable branches, typical for sparse data structures with irregular access patterns.

- **Insertion Sort:** Implementing the insertion sort algorithm, this benchmark runs 233 instructions in 235 clock cycles. It has a high branch misprediction rate of 58.62% (17 out of 29 branches), requiring 17 flushes and 170 NOPs, with an execution time of 4972.5 units. The significant misprediction rate and overhead indicate frequent control flow changes, challenging the processor's ability to maintain efficiency despite the moderate instruction count.

3.3.2 Additional Benchmarks

Arithmetic Instructions Test

This benchmark verifies basic arithmetic and logical MIPS operations. The results show correct execution and handling of signed and unsigned values.

Step	Test Type	Instruction	Result
1	**Load Values**	`ADDI` $\$13, \$0, 0x0FF0$	$R_{13} = 00000FF0$
2		`ADDI` $\$14, \$0, 0xF0F0$	$R_{14} = FFFFF0F0$
3	**Addition (ADD)**	`ADD` $\$1, \$13, \$14$	$R_1 = R_{13} + R_{14} = E0$
4	**Subtraction (SUB)**	`SUB` $\$2, \$13, \$14$	$R_2 = R_{13} - R_{14} = 00001F00$
5		`SUB` $\$3, \$14, \$13$	$R_3 = R_{14} - R_{13} = FFFFE100$
6	**Self-Subtraction (SUB)**	`SUB` $\$4, \$13, \$13$	$R_4 = R_{13} - R_{13} = 0$
7	**Bitwise AND (AND)**	`AND` $\$5, \$13, \$14$	$R_5 = R_{13} \wedge R_{14} = F0$
8	**Bitwise OR (OR)**	`OR` $\$6, \$13, \$14$	$R_6 = R_{13} \vee R_{14} = FFFFFFF0$
9	**Bitwise XOR (XOR)**	`XOR` $\$7, \$13, \$14$	$R_7 = R_{13} \oplus R_{14} = FFFFFF00$
10	**Bitwise NOR (NOR)**	`NOR` $\$8, \$13, \$14$	$R_8 = \neg(R_{13} \vee R_{14}) = 0000000F$
11	**Set Less Than (SLT)**	`SLT` $\$10, \$14, \$13$	$R_{10} = (R_{14} < R_{13}) = 1$
12	**Set Greater Than (SGT)**	`SGT` $\$11, \$13, \$14$	$R_{11} = (R_{13} > R_{14}) = 1$

Table 3: Arithmatic Instructions Test

Immediate Instruction Test

This benchmark evaluates immediate instructions. The results verify correct sign extension and bitwise behavior.

Step	Instruction	Result
	Immediate Instructions	
1	ADDI $\$9, \$0, 0xFFF0$	$R_9 = FFFFFFF0$
2	ADDI $\$10, \$0, 0xF00F$	$R_{10} = FFFFF00F$
3	ADDI $\$11, \$10, 0xF00F$	$R_{11} = FFFFE01E$
4	ORI $\$31, \$9, 0xFFFF$	$R_{31} = FFFFFFFF$
5	XORI $\$15, \$11, 0x1245$	$R_{15} = FFFFF25B$
6	ANDI $\$0, \$11, 0xFFFF$	
7	SLTI $\$1, \$11, 0xF1F5$	0
8	SLTI $\$1, \$11, 0x0000$	1

Table 4: Immediate Instruction Test

Load Use Case Test

This benchmark tests memory load (LW) and arithmetic operations that depend on loaded values. This test helps analyze load-use hazards, where a register is used immediately after being loaded. Proper forwarding or pipeline stalls may be needed to ensure correctness.

Step	Instruction	Result
	Load Use	
1	LW $\$14, 0x0(\$0)$	✓
2	ADDI $\$1, \$14, 0xA$	✓
3	ADD $\$4, \$14, \$1$	✓
4	ADD $\$5, \$14, \$1$	✓
5	LW $\$14, 0x0(\$0)$	✓
6	ADD $\$20, \$14, \$1$	✓
7	ADDI $\$11, \$14, 0xB$	✓

Table 5: Load Use case Test

3.4 Performance Comparison

The performance evaluation of our two-way superscalar processor, as derived from the provided benchmark data, demonstrates a significant improvement over its single-issue predecessor and alternative configurations, including single-cycle and five-stage pipelined designs without branch prediction. Across various benchmarks, our superscalar design consistently achieved a Cycles Per Instruction (CPI) of approximately 0.5 in most cases, reflecting its ability to execute two instructions per clock cycle under optimal conditions. For instance, in the "insertion sort, as given" benchmark, the superscalar processor completed 443 instructions in 255 clock cycles $CPI \approx 0.57$, compared to the single-cycle design's 251 cycles for 251 instructions CPI = 1.0 and the five-stage pipelined design branch prediction's 304 cycles for 260 instructions (CPI ≈ 1.16. However, stalls due to hazards and branch mispredictions occasionally increased the CPI to between 0.5 and 0.7, as seen in benchmarks like "selection sort," where 6288 instructions required $3,630$ cycles CPI ≈ 0.57 with 77 flushes, indicating some pipeline inefficiencies.

Benchmark	**Inst. Count**	**# Clk**	**EXEC TIME**
Binary_Search	32	32	960
Max, Min	96	96	2880
Multiplication using addition	19	19	570
Remove_Duplicates	508	508	15240
Scaler Mul	450	450	13500
Selection Sort	10765	10765	322950
Swapping	22	22	660
Sparse_Matrix	194	194	5820
Insertion Sort	251	251	7530

Table 6: Single Cycle Performance Table

Benchmark	# Branch	# False	False %	Inst. Count	# Clk	# Flush	# NOPs	EXEC TIME
Binary_Search	14	3	21.43%	50	54	3	42	1053
Max, Min	27	11	40.74%	85	115	3	96	2242.5
Multiplication using addition	3	2	66.67%	16	26	3	37	680
Remove_Duplicates	147	27	18.37%	542	708	27	349	18335
Scaler Mul	94	25	26.60%	415	530	38	385	8580
Selection Sort	820	777	9.39%	3625	3630	77	2666	70785
Swapping	0	0	0%	22	22	0	20	448.5
Sparse_Matrix	43	15	34.88%	178	200	12	110	3568.5
Insertion Sort	29	17	58.62%	233	235	17	170	4972.5

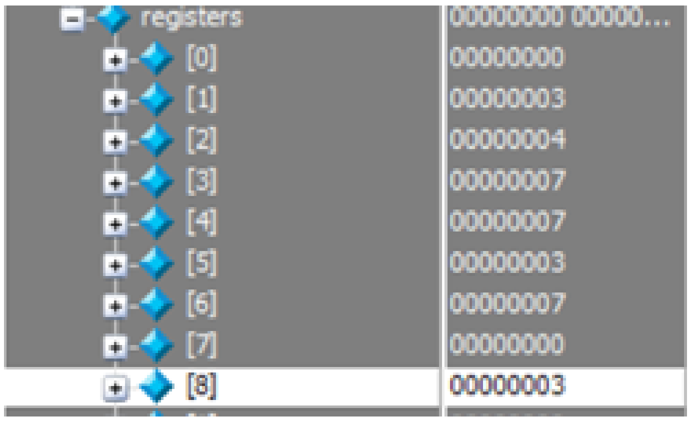
Table 7: Super Scaler Performance Table

Benchmark	# Branch	# False	False %	Inst. Count	# Clk	# Flush	# Stall	EXEC TIME
Swapping	0	0	0%	22	28	0	0	532
Max, Min	27	14	51.85%	105	150	14	9	2850
Multiplication using addition	3	2	66.67%	19	31	0	0	589
Remove_Duplicates	131	23	17.56%	563	613	23	49	11647
Scaler Mul	94	26	27.66%	449	525	36	0	9975
Selection Sort	820	80	9.76%	4102	4266	80	400	81054
Binary_Search	8	3	37.50%	35	47	3	3	893
Sparse_Matrix	43	16	37.21%	201	237	16	12	4503
Insertion Sort	29	20	68.97%	260	304	20	9	5776

Table 8: 5-Stage Pipelined Processor with Branch Prediction Performance Table

3.5 Signal Analysis and Relevant Documentation

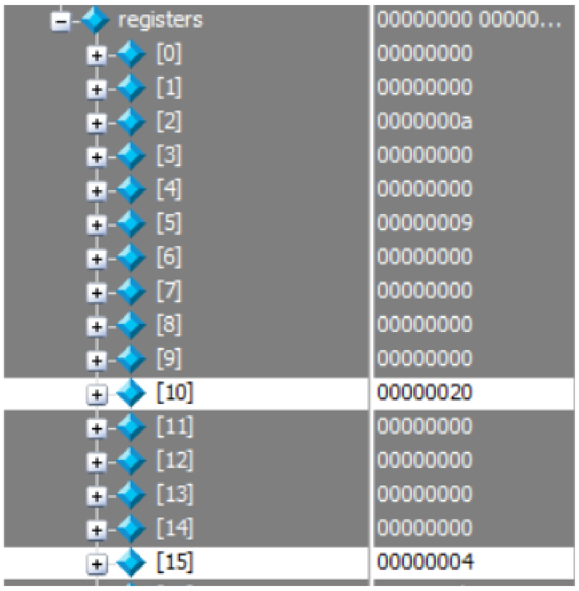
Binary Search:



```
registers   00000000 00000...
  [0]       00000000
  [1]       00000003
  [2]       00000004
  [3]       00000007
  [4]       00000007
  [5]       00000003
  [6]       00000007
  [7]       00000000
  [8]       00000003
```

Figure 9: Binary search benchmark results

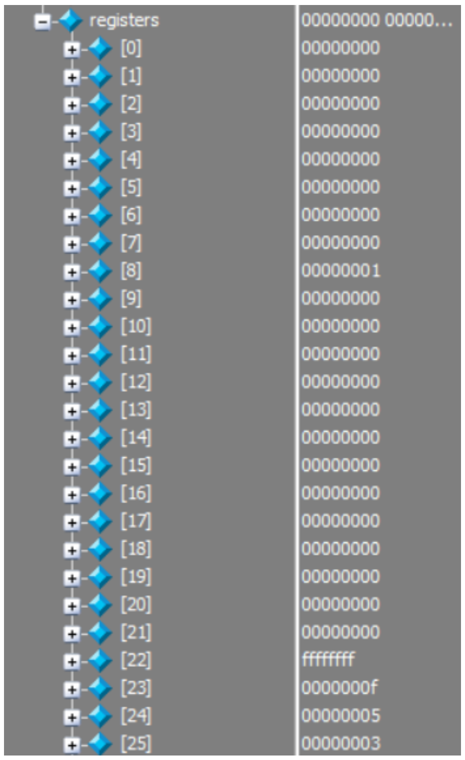
Multiplication using Addition:



```
registers   00000000 00000...
  [0]       00000000
  [1]       00000000
  [2]       0000000a
  [3]       00000000
  [4]       00000000
  [5]       00000009
  [6]       00000000
  [7]       00000000
  [8]       00000000
  [9]       00000000
  [10]      00000020
  [11]      00000000
  [12]      00000000
  [13]      00000000
  [14]      00000000
  [15]      00000004
```

Figure 10: Multiplication using addition benchmark results

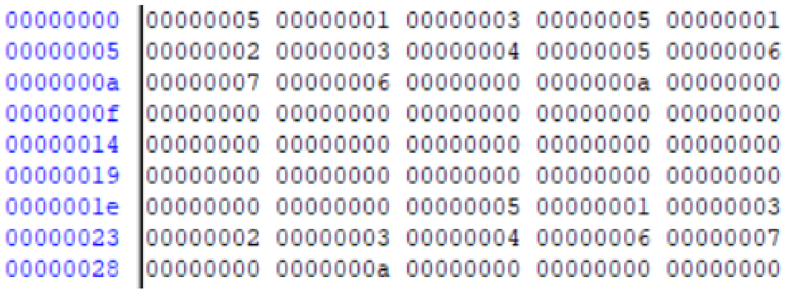
Max & Min:



```
registers   00000000 00000...
[0]         00000000
[1]         00000000
[2]         00000000
[3]         00000000
[4]         00000000
[5]         00000000
[6]         00000000
[7]         00000000
[8]         00000001
[9]         00000000
[10]        00000000
[11]        00000000
[12]        00000000
[13]        00000000
[14]        00000000
[15]        00000000
[16]        00000000
[17]        00000000
[18]        00000000
[19]        00000000
[20]        00000000
[21]        00000000
[22]        ffffffff
[23]        0000000f
[24]        00000005
[25]        00000003
```

Figure 11: Max and Min benchmark results

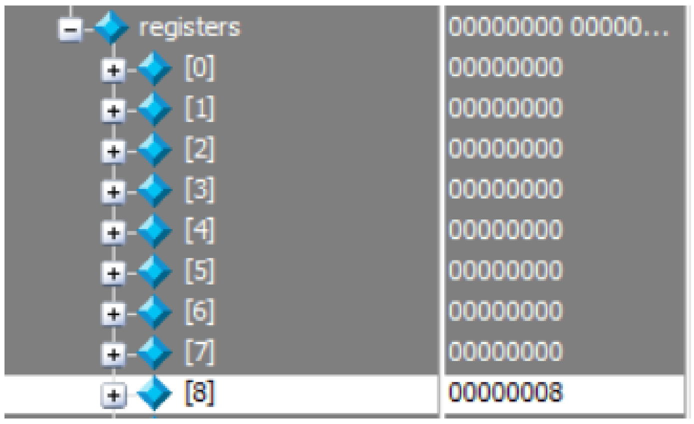
Remove duplicates:



```
00000000 |00000005 00000001 00000003 00000005 00000001
00000005 |00000002 00000003 00000004 00000005 00000006
0000000a |00000007 00000006 00000000 0000000a 00000000
0000000f |00000000 00000000 00000000 00000000 00000000
00000014 |00000000 00000000 00000000 00000000 00000000
00000019 |00000000 00000000 00000000 00000000 00000000
0000001e |00000000 00000000 00000005 00000001 00000003
00000023 |00000002 00000003 00000004 00000006 00000007
00000028 |00000000 0000000a 00000000 00000000 00000000
```

Figure 12: Remove duplicates benchmark results

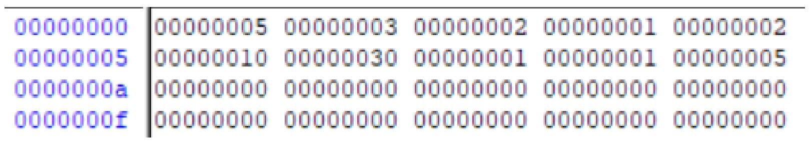
Sparse Matrix Count:



```
registers  00000000 00000...
  [0]      00000000
  [1]      00000000
  [2]      00000000
  [3]      00000000
  [4]      00000000
  [5]      00000000
  [6]      00000000
  [7]      00000000
  [8]      00000008
```

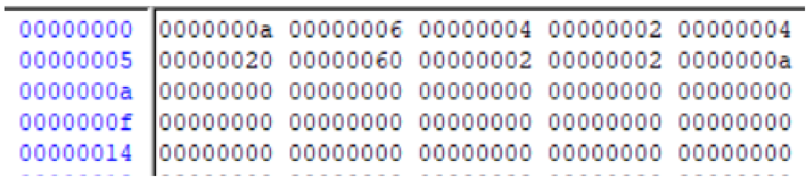
Figure 13: Sparse Matrix Count benchmark results

Scaler Multiplication:



```
00000000 00000005 00000003 00000002 00000001 00000002
00000005 00000010 00000030 00000001 00000001 00000005
0000000a 00000000 00000000 00000000 00000000 00000000
0000000f 00000000 00000000 00000000 00000000 00000000
```

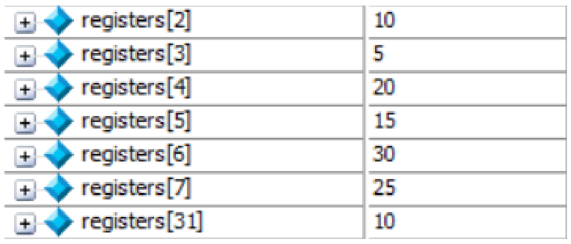
Figure 14: Scaler Multiplication benchmark results - Before



```
00000000 0000000a 00000006 00000004 00000002 00000004
00000005 00000020 00000060 00000002 00000002 0000000a
0000000a 00000000 00000000 00000000 00000000 00000000
0000000f 00000000 00000000 00000000 00000000 00000000
00000014 00000000 00000000 00000000 00000000 00000000
```

Figure 15: Scaler Multiplication benchmark results - After

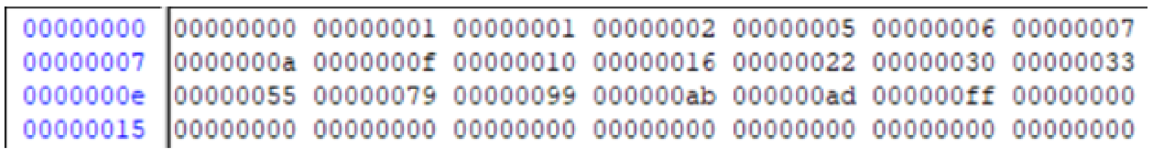
Swapping:



```
registers[2]    10
registers[3]    5
registers[4]    20
registers[5]    15
registers[6]    30
registers[7]    25
registers[31]   10
```

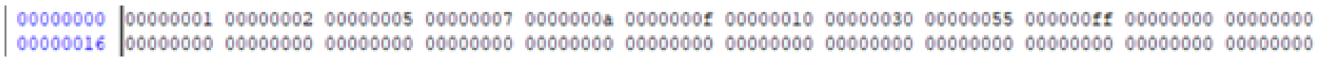
Figure 16: Swapping benchmark results

Sorting algorithms:



```
00000000 00000000 00000001 00000001 00000002 00000005 00000006 00000007
00000007 0000000a 0000000f 00000010 00000016 00000022 00000030 00000033
0000000e 00000055 00000079 00000099 000000ab 000000ad 000000ff 00000000
00000015 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 17: Selection sort benchmark results



```
00000000 00000001 00000002 00000005 00000007 0000000a 0000000f 00000010 00000030 00000055 000000ff 00000000 00000000
00000016 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 18: Insertion sort benchmark results

3.6 Software Tools and implementations

3.6.1 Exceptions Handler:

MIPS processors, widely used in embedded systems and academic settings, require robust mechanisms to handle exceptions to maintain execution integrity. Arithmetic errors such as division by zero, overflow, and illegal instructions can lead to unpredictable program behavior. Exception handling in MIPS processors is essential for software reliability, preventing crashes, and providing mechanisms for debugging and real-time error correction. This document details a software-based approach to exception handling, ensuring smooth program execution and effective recovery from runtime issues.

1. **Design and Implementation**

   - **System Overview:** The system is structured into three main components:
     (a) **Arithmetic Overflow and Underflow Exception Handler**: Detects and corrects arithmetic overflow and underflow errors.
     (b) **Illegal Instruction Handler**: Identifies and corrects invalid MIPS instructions.
     (c) **Main Control Unit**: Coordinates the execution of handlers and processes MIPS assembly code.

   Implementation Details

   - **Arithmetic Overflow and Underflow Exception Handling**
     - Identifies arithmetic overflow in addition and subtraction operations.
     - Uses integer bounds to detect overflow and underflow conditions.
     - Replaces erroneous instructions with corrected values to maintain execution flow.
   - **Illegal Instruction Handling**
     - Parses MIPS assembly code to detect improperly formatted or unsupported instructions.
     - Corrects instructions based on predefined valid formats.
   - **Main Control Flow**
     - Reads MIPS assembly code from an input file.
     - Initializes registers and sets default values.
     - Calls exception handler functions sequentially for each instruction.
     - Writes corrected MIPS code to an output file, which then serves as input to the scheduler.

2. **Input and Output File Formats**

   The input file is a plain text document containing MIPS assembly instructions, with each instruction adhering to standard MIPS syntax. Comments and labels are supported, allowing developers to structure their code efficiently. A typical input file contains arithmetic operations, memory accesses, and branching instructions, making it essential for the exception handling system to process each line accurately.

The output file maintains the corrected MIPS assembly code after processing exceptions. Each modified instruction includes comments explaining the corrections, allowing developers to understand the adjustments made by the system. If an illegal instruction is detected, it is replaced with a valid format, ensuring smooth execution.

3. **Results and Benefits**

   The exception handling system effectively detects and corrects arithmetic overflow and underflow errors and illegal instructions, ensuring robust software execution. By implementing structured parsing and correction mechanisms, the system maintains minimal overhead while enhancing reliability. The efficiency of error detection allows programs to execute without unexpected failures, providing a more stable computing environment.

   Performance analysis indicates that the structured approach optimizes error resolution through lookup tables and efficient string processing techniques. As a result, error correction occurs in real time without introducing significant delays. This is particularly beneficial for embedded systems and real-time applications, where stability and low latency are crucial.

   The system has practical applications in academia, where it assists students in debugging MIPS assembly code and understanding error correction mechanisms. Additionally, it enhances the reliability of embedded systems where MIPS processors are commonly used, ensuring that mission-critical applications remain stable and functional.

4. **Future Improvements**

   - **Support for More Exception Types** as well as additional instructions.
   - **Optimized Performance**: Improving algorithm efficiency for real-time processing.
   - **Graphical Debugging Interface**: Developing a GUI tool for visualizing instruction flow and errors.
   - **Integration with MIPS Simulators**: Extending support to work alongside MIPS emulators for automated debugging.

The software-based exception handling system for MIPS processors significantly improves code reliability and execution stability. By implementing arithmetic error detection and instruction correction, it serves as a crucial debugging tool and enhances program robustness. The structured approach ensures usability across various applications, making it a valuable tool for both academic and industrial use. Future improvements, such as enhanced reporting, graphical debugging, and real-time analysis, will further strengthen its capabilities. The system provides an essential layer of reliability for MIPS assembly programming, ensuring that software executes smoothly without unexpected failures.

3.6.2 Instructions scheduler:

When dealing with more than one instruction at a time and to prevent dependencies, the design requires a scheduler which can be hardware-based, but it would add so much logic leading to a slower design. The software scheduler can reorder the instructions without messing up the order and the dependencies. It can also control what instructions to be fetched based on the type, which obliges the asymmetric configuration of the design, by not allowing the following cases to be fetched together:

(a) Two instructions of any type that have dependencies regardless of the type.

(b) Two memory access instructions at the same time.

(c) Two ALU instructions at the same time.

- The scheduler controls these instructions by reordering the instructions, which are done by performing instructions that have no dependency on any other instruction to fill up the gaps instead of NOPs (if possible). If there is no way to reorder, then the problems will be solved by adding NOP instructions until the data dependencies are resolved.
- The scheduler can also rename registers if needed. And overall, the scheduler is the first step of avoiding hazards.
- Generally, the scheduler does most of the work to avoid hazards, but it leaves the forwarding to be performed by its designated unit.

3.6.3 Assembler:

The assembler is a software that turns the MIPS instructions (both standard and pseudo) into machine code. This tool is quite useful for testing the design with any kind of benchmark, as it provides the code with no trouble. The assembler has been created and used in recent phases, but it had some issues when dealing with labels, for instance, it could not deal with pseudo instructions when the label is needed for loops and jumps, because the pseudo instruction is broken down into two instructions and this should be considered to have a reliable labeling mechanism. Other cases when the instruction is after a label but not in the same line as it can cause inconvenience when testing each code. This also applies to comments in the code which was not handled well in the recent assembler and now the assembler is fully functional.

3.7 Executive Summary

Two-Way Superscalar Processor Design

This project details the development of an advanced two-way superscalar processor, a well known architecture designed to enhance computational performance by enabling the simultaneous execution of two instructions per clock cycle. Implemented in Verilog Hardware Description Language (HDL), this processor transcends the limitations of traditional single-issue processors by leveraging parallelism at the instruction level, effectively doubling throughput under ideal conditions. The design is centered around a pre crafted five-stage pipeline—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB)—tailored to manage dual instruction streams with precision and efficiency.

At its core, the processor supports dual-instruction fetch mechanism that retrieves two instructions in parallel during each clock cycle, supported by a specialized dual-port instruction memory. This capability lays the foundation for superscalar operation, allowing the system to process multiple instructions concurrently. To maximize pipeline efficiency and minimize disruptions, the design integrates advanced branch prediction and forwarding units. These components work in tandem to anticipate control flow changes and resolve data dependencies, reducing the frequency of pipeline stalls and flushes that could otherwise hinder performance. The branch prediction system employs sophisticated techniques to forecast the outcomes of conditional branches, while the forwarding logic ensures that inter-instruction dependencies are handled seamlessly.

In the execution stage, the processor features dual Arithmetic Logic Units (ALUs), each dedicated to one of the two instruction streams. This parallel execution capability enables independent arithmetic, logical, and shift operations, significantly boosting computational throughput. The memory access stage utilizes a shared data memory system, carefully orchestrated to support read and write operations from both execution paths. To address potential data hazards arising from simultaneous instruction execution, an advanced forwarding logic mechanism is implemented, allowing intermediate results to be bypassed directly to dependent instructions without waiting for write-back completion.

The instruction decode stage is equipped with dual control units, each generating the necessary control signals for its respective instruction stream. This ensures that both instructions are decoded and prepared for execution independently yet cohesively. A critical component of this stage is the issue logic, which determines the pairing and ordering of instructions to optimize resource utilization and avoid conflicts such as Write-After-Write (WAW) hazards. The register file is designed to support this dual-issue paradigm, providing multiple read and write ports to accommodate simultaneous access by both instruction paths, ensuring data availability without bottlenecks.

The processor also includes comprehensive support for control flow instructions, such as jumps, branches, and jump-and-link operations, which are critical for real-world program execution. These are managed through dedicated logic that coordinates with the branch prediction unit to maintain pipeline integrity and correctness, even in the presence of complex control transfers. The write-back stage finalizes the execution process by committing results to the register file or memory, with additional logic to handle special cases like jump-and-link instructions that require program counter values to be stored.

To support a complete ecosystem for this processor, the development of an assembler is proposed as a critical component. The assembler translates high-level assembly instructions (MIPS) into the processor's machine code, adhering to its custom instruction set architecture (MIPS ISA). Given the superscalar nature of the design, the assembler must be capable of handling dual-issue semantics, ensuring that instructions are encoded in a way that maximizes parallelism while respecting dependencies and resource constraints. It would include features such as instruction reordering for optimal pairing, support for branch prediction hints, and directives for memory alignment. This tool would enable programmers to write efficient code tailored to the processor's capabilities, bridging the gap between software and hardware.

Complementing the assembler, an exception handler is envisioned to enhance the processor's reliability and robustness. The exception handling system would manage events such as arithmetic overflows, memory access violations, and illegal instructions, which are inevitable in real-world applications. Implemented as a combination of hardware and firmware, the handler would detect exceptions during execution, save the processor state (e.g., program counter and registers), and transfer control to predefined routines. For the two-way design, this requires careful coordination to handle exceptions from either execution path, potentially pausing one stream while resolving an issue in the other. The exception handler would also support interrupts, enabling integration with external devices, and provide a mechanism for returning to normal execution, ensuring the processor remains responsive and fault-tolerant.

A cycle-accurate simulator forms the third pillar of this ecosystem, offering a virtual environment to test and validate the processor's behavior. Unlike the Verilog implementation, which focuses on hardware synthesis, the simulator would model the processor's operation at a cycle-by-cycle level, accurately replicating pipeline dynamics, instruction timing, and resource contention. Written in a high-level language (Python), it would simulate the dual-issue pipeline, branch prediction outcomes, hazard detection, and memory interactions, providing detailed statistics such as cycles per instruction (CPI), stall frequency, and branch misprediction rates. This tool would be invaluable for debugging the hardware design, verifying the assembler's output, and testing exception handling scenarios without requiring physical hardware. Additionally, it could serve as an educational platform, allowing developers to experiment with the processor's behavior under various workloads.

Together, these components—the assembler, exception handler, and cycle-accurate simulator—create a holistic ecosystem that transforms the two-way superscalar processor from a theoretical design into a practical computing solution. The assembler empowers software development, the exception handler ensures operational resilience, and the simulator facilitates verification and performance analysis. This integrated approach not only validates the processor's functionality but also positions it for real-world deployment, where it can handle diverse workloads with efficiency and reliability. The ecosystem lays the groundwork for future enhancements, such as expanding the ISA, optimizing branch prediction algorithms, or integrating cache systems, further elevating the processor's capabilities.

As part of the testing plan for the two-way superscalar processor design, comprehensive functional unit coverage was conducted to evaluate the behavior of updated units—namely the Register File, Issue Logic, ALU, and Pipeline Registers—transitioning from the previous single-issue architecture. This testing ensured that all predefined procedures and critical corner cases were addressed, achieving high coverage scores across all scenarios which will be shown in coming sections in this report.
For the Register File, tests validated basic read/write operations, dual-port simultaneous writes with priority handling, and edge cases like writing to register zero (ensuring it remains unchanged) and accessing the maximum register address, confirming reliable multi-port functionality. The Issue Logic was rigorously tested for special instructions (e.g., JR, branches, jumps, load/store) and non-special cases, alongside corner scenarios like identical program counters and older instructions in the second slot, ensuring correct instruction routing and priority assignment. The ALU underwent exhaustive testing across all operations—arithmetic (ADD, SUB), logical (AND, OR, XOR, etc.), and comparisons (SLT, SGT)—with additional focus on reset behavior, arithmetic overflow, and undefined operation handling, verifying robust computation capabilities. Pipeline Registers were tested for reset functionality, normal propagation of active/inactive signals, mixed signal states, maximum value handling, and mid-operation resets, ensuring seamless data flow across stages. Collectively, these results demonstrate that the updated units successfully support the dual-issue paradigm, covering typical use cases and edge conditions with high reliability, as validated through directed test cases designed to probe each unit's full operational scope.

In the performance evaluation of our two-way superscalar processor design, we achieved an impressive Cycles Per Instruction (CPI) of 0.5 in most cases, reflecting the architecture's ability to execute two instructions per clock cycle under optimal conditions. This near-ideal CPI highlights the effectiveness of dual instruction fetching, parallel ALUs, and efficient issue logic in maximizing throughput. However, in scenarios involving pipeline stalls—stemming from data hazards, branch mispredictions, or resource contention—the CPI ranged between 0.5 and 0.7. These stalls, while infrequent, slightly elevated the average CPI, underscoring the robustness of our hazard detection and forwarding mechanisms in mitigating delays, though not entirely eliminating them, particularly in complex instruction sequences.

The performance evaluation of our two-way superscalar processor, as derived from the provided benchmark data, demonstrates a significant improvement over its single-issue predecessor and alternative configurations, including single-cycle and five-stage pipelined designs without branch prediction. Across various benchmarks, our superscalar design consistently achieved a Cycles Per Instruction (CPI) of approximately 0.5 in most cases, reflecting its ability to execute two instructions per clock cycle under optimal conditions. For instance, in the "insertion sort, as given" benchmark, the superscalar processor completed 443 instructions in 255 clock cycles (CPI ≈0.57), compared to the single-cycle design's 251 cycles for 251 instructions ($CPI = 1.0$) and the five-stage pipelined design branch prediction's 304 cycles for 260 instructions (CPI ≈1.16). However, stalls due to hazards and branch mispredictions occasionally increased the CPI to between 0.5 and 0.7, as seen in benchmarks like ”selection sort,” where 6288 instructions required 3, 630 cycles (CPI ≈0.57) with 77 flushes, indicating some pipeline inefficiencies. In conclusion, this project successfully demonstrates the principles of parallel instruction processing, pipeline optimization, and dependency management in a modern processor context. It provides a robust platform for further research and development in high-performance computing architectures, showcasing the potential of superscalar techniques to meet the growing demands of contemporary and future computing workloads.

4 List of Figures

5 List of Tables

6 References

1. Computer Organization and Design: The Hardware/ Software Interface, by David A. Patterson and John L. Hennessy, Fifth Edition

2. Computer Architecture: A Quantitative Approach, by David A. Patterson and John L. Hennessy Fifth Edition

3. Intel Community