GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

RISC-V: Data Integrity in High-Performance Data Analysis

Abdallah Abdelnaby

MatrNr: 19334664

Supervisor: Freja Nordsiek

Georg-August-Universität Göttingen
Institute of Computer Science

August 20, 2024

Abstract

In contemporary computing, data reliability and integrity are crucial, especially in systems that require high availability and fault tolerance. Bit flips, which occur when individual bits in memory or storage change state owing to a variety of circumstances, pose a substantial challenge to data integrity and can have serious repercussions. This project implements Triple Modular Redundancy (Triple Modular Redundancy (TMR)) in the RISC-V 32I architecture to improve fault tolerance in essential data storage components, such as the Program Counter (Program Counter (PC)), Register File, Instruction Memory, and Data Memory. We use Xilinx Vivado for design and simulation to assess the influence of TMR on performance measures such as logic delay, net delay, total delay, power consumption, and thermal properties. The findings show that while TMR considerably improves system dependability, but it also includes performance and electricity overheads. Design optimization strategies alleviate these disadvantages by balancing dependability and efficiency. This paper highlights the possibilities and limitations of implementing fault-tolerant methods in open-source architectures such as Reduced Instruction Set Computer - Five (RISC-V).

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

ISA Instruction Set Architecture

RISC-V Reduced Instruction Set Computer - Five

TMR Triple Modular Redundancy

PC Program Counter

x86 A family of backward-compatible instruction set architectures based on the Intel 8086 CPU

Vivado Xilinx's FPGA design suite

HPC High-Performance Computing

1 Introduction

Open-source Instruction Set Architecture (ISA) has had a considerable impact on computer architecture research and industry. Among them, RISC-V has emerged as a revolutionary force, offering a diverse and scalable framework for educational, scientific, and industrial applications. Krste Asanović, Andrew Waterman, and Yunsup Lee founded RISC-V at UC Berkeley in 2010, with contributions from David Patterson[Fou24]. The RISC-V Foundation, a non-profit corporation, promotes and develops the project[Fou24].

The open-source nature of RISC-V ensures the necessary scalability, flexibility, and efficiency for current data analytic applications. This study examines the current status of RISC-V, emphasizing its use in high-performance data analysis and the problems related to data integrity. The talk focuses on implementing TMR to improve data processing reliability and accuracy, especially in important components like the PC and memory.

This article examines the architecture of RISC-V and its implications for data analytics, emphasizing its performance and flexibility benefits. RISC-V standardizes instructions and allows for varied implementations, resulting in resilient and efficient data processing systems [Jou24] [Blo24] [PH17]. The article will examine the architectural aspects of RISC-V, its performance versus other ISAs, and the measures used to prevent data loss and maintain data integrity. This exploration aims to provide a detailed overview of RISC-V's potential and contributions to high-performance data analysis.

1.1 RISC-V in Data Analytics

Scalability, flexibility, and efficiency make RISC-V an invaluable tool for data analytics. Using the open-source nature of RISC-V, data analytics systems can manage large volumes of data more efficiently. According to [Mak24], SiFive's P670 and other RISC-V processors outperform standard A family of backward-compatible instruction set architectures based on the Intel 8086 CPU (x86) and ARM processors. Figure 1 shows that ARM's Cortex-A78 outperforms SiFive's P670 by 5% in peak single-thread performance (SpecINT2k6). However, ARM's Cortex-A78 has twice the compute density (SpecINT2k6/mm^2), making SiFive's P670 better in this sense. SiFive's P670 takes about 50% less space than Cortex-A78, demonstrating its space efficiency.

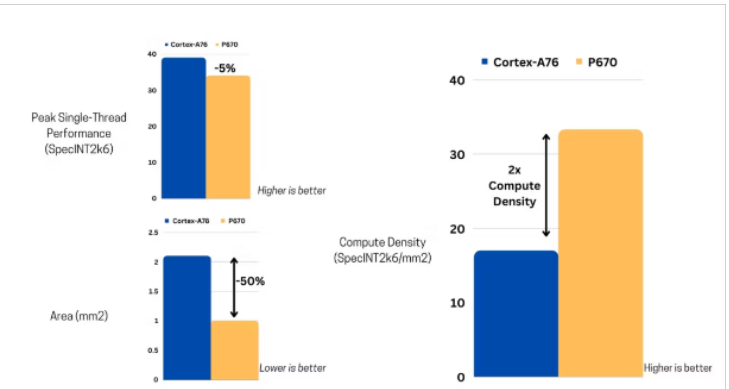

Figure 1: ARM's Cortex-A78 and SiFive's P670 Performance Comparison [Mak24]

Key advantages of RISC-V in data analytics include scalability, flexibility, and efficiency. Scalability refers to the ability to expand computing resources efficiently to manage

large datasets. Flexibility is achieved through customizable instruction sets tailored to specific data processing needs, and efficiency is demonstrated by optimized performance for various data analytics tasks with lower power consumption and cost [PH17] [HP18].

Figure 2 highlights the distribution of x86 instructions in integer programs. The table indicates that instructions like Load, Conditional Branch, and Compare are most frequently executed by compilers in integer programs, accounting for a total of 96% of all executed instructions. This insight emphasizes the critical role of these instructions in data processing tasks and the potential for optimization within RISC-V architectures.

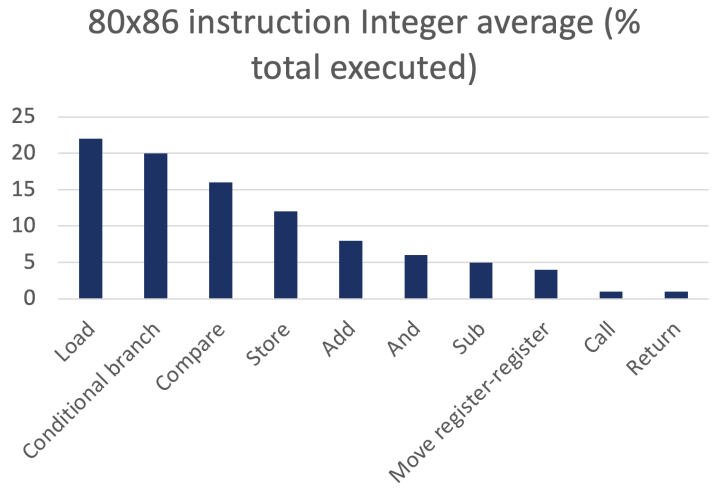

Figure 2: x86 Used Instruction in Integer Programs

1.2 RISC-V 32I ISA Types

The RISC-V 32I Instruction Set Architecture (ISA) defines a standard base set of instructions for 32-bit processors, which is crucial for various applications in data analytics and computing [Ali23]. The 32I ISA includes multiple instruction formats designed to provide flexibility and efficiency in different operations. According to [Fou24] and [HP18], The primary instruction types in the RISC-V 32I ISA as shown in 3 are R-type, I-type, S-type, B-type, U-type, and J-type. Each type serves a distinct purpose and has a specific bit distribution pattern.

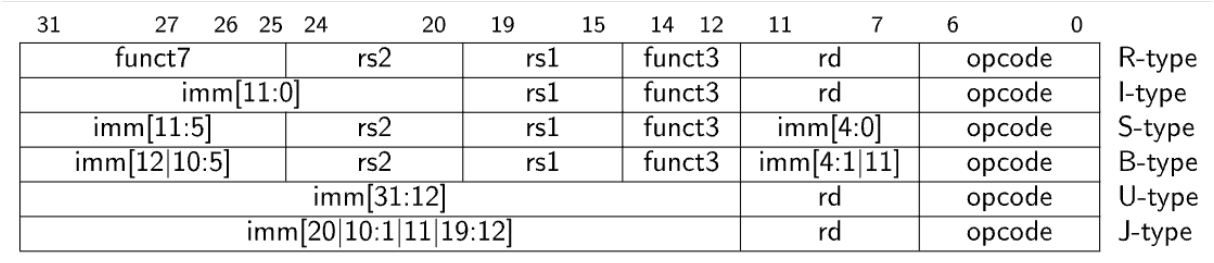

Figure 3: RISC-V 32I Instruction Types [Fou24]

- R-Type Instructions R-type instructions are used for register-register operations. They include arithmetic and logical operations such as addition, subtraction, and

logical AND, OR, XOR. The format of an R-type instruction consists of the following fields: opcode, rd (destination register), funct3, rs1 (source register 1), rs2 (source register 2), funct7.

- I-Type Instructions I-type instructions are used for immediate operations. These instructions involve an immediate value rather than a second source register. They are commonly used for arithmetic operations with a constant, load instructions, and certain control instructions. The format includes opcode, rd, funct3, rs1, and a 12-bit immediate value.

- S-Type Instructions S-type instructions handle store operations where data is stored from a register to memory. The format of S-type instructions includes fields for opcode, funct3, rs1, rs2, and a 12-bit immediate value split between two parts for addressing purposes.

- B-Type Instructions B-type instructions are used for conditional branches. These instructions compare two registers and branch to a target address if the condition is met. The format includes opcode, funct3, rs1, rs2, and a 12-bit immediate value split between two parts for the branch offset.

- U-Type Instructions U-type instructions provide a large immediate value, which is useful for upper immediate operations, such as loading a 20-bit upper immediate value into a register. The format includes opcode, rd, and a 20-bit immediate value.

- U-Type Instructions J-type instructions are used for jump and link operations, where the program counter (PC) is updated with a target address, and the return address is stored in a register. The format includes opcode, rd, and a 20-bit immediate value for the jump target.

1.3 Problem Statement and Objectives

In the scope of modern computing, the reliability and integrity of data are paramount, especially in systems that demand high availability and fault tolerance. One prevalent issue that threatens data integrity is the phenomenon of bit flips, where individual bits in memory or storage spontaneously change state from 0 to 1 or vice versa. According to [Rad24], bit flips can be caused by various factors, including cosmic rays, electromagnetic interference, or manufacturing defects. These seemingly random errors can have significant consequences, ranging from minor glitches to critical system failures.

A notable real-world incident highlighting the impact of bit flips is the Belgium election mishap in 2003 [Rad24]. During the regional elections, a single bit flip in a computer's memory caused an incorrect vote tally, awarding an extra 4,096 votes to a candidate by mistake. This error was only identified due to the presence of paper ballots, underscoring the potential severity of bit-flip errors in electronic systems.

Given the increasing dependence on digital systems in critical applications such as aerospace, medical devices, and financial systems, there is a compelling need for robust mechanisms to ensure data integrity and system reliability. The RISC-V 32I architecture, with its open-source and flexible design, presents an ideal platform to implement and evaluate fault-tolerant strategies such as Triple Modular Redundancy (TMR).

The primary objectives of this project are:

1. Identify Critical Data Storage Components:

- Determine the key components within the RISC-V 32I architecture that are most susceptible to bit flip errors and would benefit from enhanced fault tolerance.
- Specifically, focus on the Program Counter (PC), Register File, Instruction Memory, and Data Memory.

2. Implement Triple Modular Redundancy (TMR):
   - Design and implement TMR for the identified critical components.
   - Replicate each component three times and integrate a majority voter to determine the correct output, mitigating the impact of single bit flip errors.
3. Develop and Verify Majority Voter Logic:
   - Create a reliable majority voter circuit using the formula Output = AB + AC + BC to ensure accurate data retrieval from the triplicated components.
   - Verify the functionality of the majority voter through simulation and testbenches.
4. Evaluate Performance Metrics:
   - Conduct comprehensive simulations using Xilinx's FPGA design suite (Vivado) to assess the impact of TMR on system performance, power consumption, and thermal characteristics.
   - Generate detailed timing reports, including total delay, net delay, and logic delay of the longest path, with and without design optimization (opt_design enabled).
   - Compare the performance of the TMR-enhanced system against a non-redundant baseline to quantify the overhead and benefits of fault tolerance.
5. Demonstrate System Reliability:
   - Validate the effectiveness of the TMR implementation in preventing and correcting bit flip errors.
   - Ensure that the system can maintain reliable operation even in the presence of faults, making it suitable for deployment in high-reliability applications.

By achieving these objectives, the project aims to enhance the robustness of the RISC-V 32I architecture, providing a resilient solution against bit flip errors and improving overall system reliability.

2 Methodology

2.1 RISC-V 32I Architecture

The RISC-V 32I architecture is a minimalist, open-source instruction set architecture (ISA) designed for simplicity and efficiency as stated in [Jou24]. It includes a variety of components that work together to execute instructions efficiently. Below [Cai24] provides Figure 4 which is an overview of the main components and their functions:

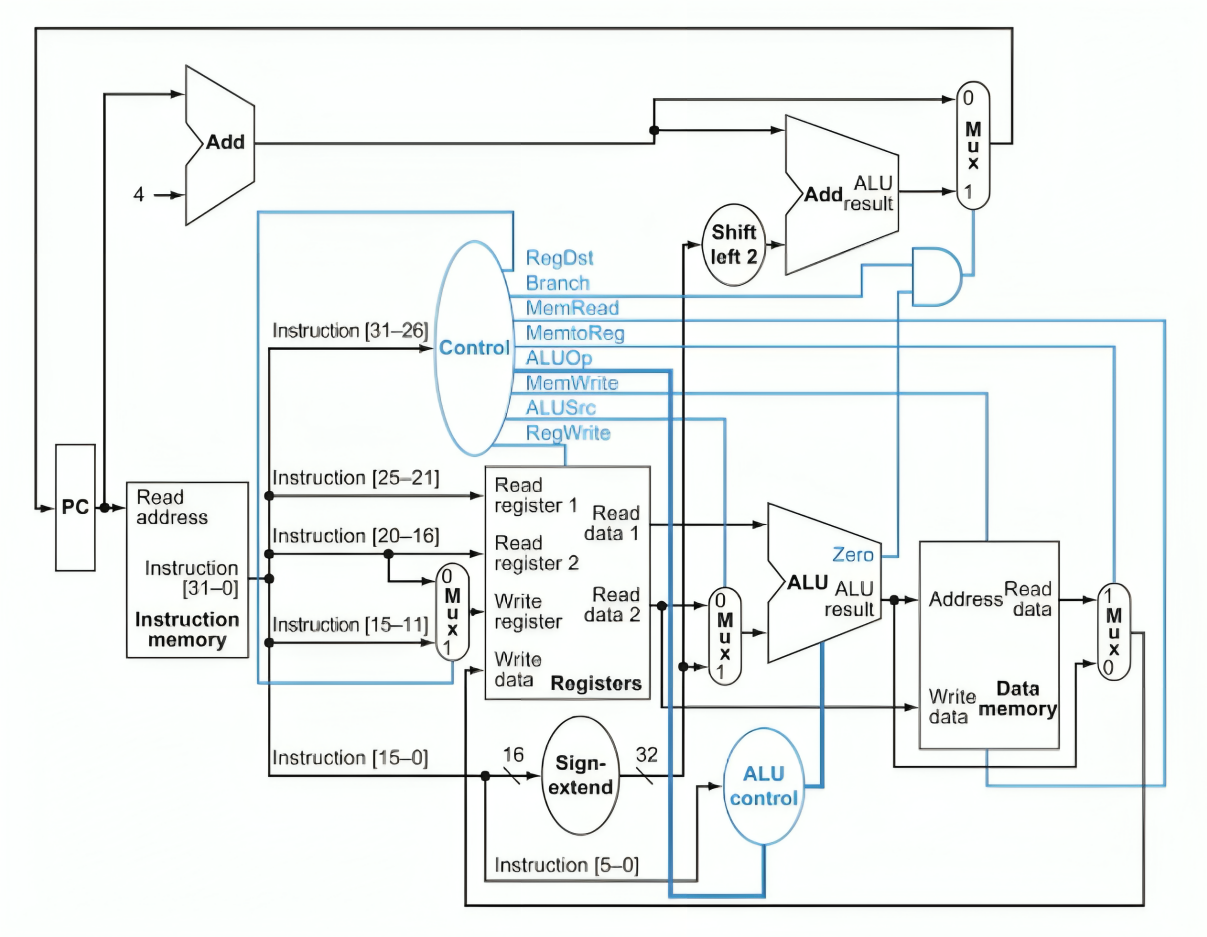

Figure 4: RISC-V 32I Architecture [Cai24]

2.1.1 Program Counter (PC)

The Program Counter (PC) holds the address of the next instruction to be executed. It is updated after each instruction fetch to point to the next instruction, which could be sequential or a result of a branch/jump instruction.

2.1.2 Adders

Adders are used within the architecture for several purposes, including incrementing the PC and performing arithmetic operations. The PC adder specifically adds a constant value (typically 4 for 32-bit instruction sets) to the current PC to fetch the next instruction.

2.1.3 Instruction Memory

Instruction memory stores the program instructions. It is accessed by the PC to fetch the current instruction for decoding and execution. The instruction memory is typically read-only during program execution.

2.1.4 Data Memory

Data memory is used to store and retrieve data required during program execution. It can be accessed for both read and write operations. Data memory interactions occur via load and store instructions.

2.1.5 Register File

The register file is a small, fast storage area consisting of a set of registers. The RISC-V 32I architecture includes 32 general-purpose registers (x0 to x31), each 32 bits wide. The register file supports simultaneous reading from two registers and writing to one register in a single cycle. Each register serves specific purposes, as outlined below:

- x0: The constant value 0
- x1: Return address
- x2: Stack pointer
- x3: Global pointer
- x4: Thread pointer
- x5 – x7, x28 – x31: Temporaries
- x8: Frame pointer
- x9, x18 – x27: Saved registers
- x10 – x11: Function arguments/results
- x12 – x17: Function arguments

2.1.6 Arithmetic Logic Unit (ALU)

The ALU performs arithmetic and logical operations. It takes two input operands and produces an output based on the operation specified by the ALU control unit. Operations include addition, subtraction, AND, OR, XOR, and comparisons.

2.1.7 ALU Control

The ALU control unit generates the necessary control signals to dictate the specific operation the ALU should perform. It interprets the operation code from the instruction and sets the ALU operation accordingly.

2.1.8 Control Unit

The control unit is responsible for generating control signals that guide the flow of data within the processor. It decodes the fetched instruction and determines the sequence of operations required to execute it, including signal generation for the ALU, register file, memory, and multiplexers.

2.1.9 Shifter

The shifter is a component within the ALU responsible for performing bitwise shift operations (left shift, right shift). It shifts the bits of the operand as specified by the instruction.

2.1.10 Multiplexers (Muxes)

Multiplexers are used to select between different data sources based on control signals. Common uses include selecting ALU input operands, choosing between register data and immediate values, and deciding between different branch/jump targets.

2.1.11 Flags

Flags are status indicators set by the ALU to reflect the result of an operation. Common flags include zero (Z), carry (C), negative (N), and overflow (V). These flags are used for conditional branching and other decision-making processes.

2.1.12 Bit Distribution

The RISC-V 32I architecture includes several instruction formats: R-type, I-type, S-type, B-type, U-type, and J-type. Each format has a specific bit distribution, with fields like opcode, register addresses, immediate values, and function codes.

Figure 3 illustrates the bit distribution for these instruction types.

2.2 TMR Majority Voting

Triple Modular Redundancy (TMR) is a fault-tolerant design methodology used to enhance the reliability of critical system components by mitigating the effects of hardware faults. In the context of the RISC-V 32I architecture, TMR can be effectively applied to data storage elements such as the Program Counter (PC), Register File, Instruction Memory, and Data Memory to ensure the integrity and correctness of stored data.

Implementation of TMR in Data Storage

1. Triplication of Components: Each critical data storage component is replicated three times. This includes triplicating the PC, Register File, Instruction Memory, and Data Memory.

2. Voting Mechanism: A majority voter is used to compare the outputs from the three replicated components and determine the correct value.

Majority Voting Process

- Input Signals: Identical input signals are sent to all three replicated components.

- Processing: Each component processes the input independently, whether it is incrementing the PC, accessing the Register File, or fetching data from memory.

- Voting: The outputs from the three components are compared by the majority voter. The voting logic is based on the formula: $Output = AB + AC + BC$, where

A, B, and C are the outputs of the three replicated components. The truth table for this logic is as follows:

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- Logic Gate Design: The logic gate design for the majority voter can be implemented as shown in Figure 5. This design ensures that the output is determined by the majority value of the inputs A, B, and C.

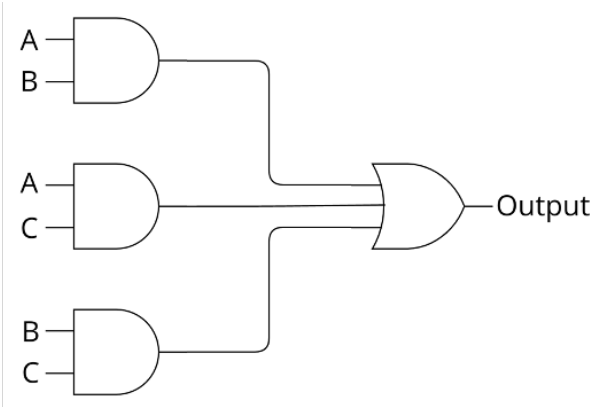

Figure 5: TMR Majority Voter

2.3 Project Implementation

The implementation of the RISC-V 32I architecture with Triple Modular Redundancy (TMR) for critical data storage components involves several steps. This section outlines the process of designing, simulating, and testing the system to ensure reliability and fault tolerance.

2.3.1 RISC-V Implementation

All RISC-V Modules are implemented and each of these components is triplicated to form the basis for TMR:

- Program Counter (PC)
- Register File
- Instruction Memory
- Data Memory

2.3.2 TMR Voter Implementation

Listing 1 shows the implementation of the TMR in Verilog code

```
module majority_voter (
    input wire A,
    input wire B,
    input wire C,
    output wire Output
);
assign Output = (A & B) | (A & C) | (B & C);
endmodule
```

Listing 1: "TMR Voter Implementation" in Verilog

2.3.3 Tools Used

Vivado: The design, synthesis, implementation, and simulation of the TMR-enhanced RISC-V 32I architecture were conducted using Xilinx Vivado. Vivado provides comprehensive tools for FPGA design, including simulation, verification, and performance analysis. Alpha-Data ADM-PCIE-7V3 was chosen in the Vivado simulation configurations as the simulation board since it is used mostly in the High-Performance Computing (HPC) systems

2.3.4 Reports

- Outcome Report

  The performance of the TMR-enhanced RISC-V 32I architecture was evaluated based on several key metrics: timing, power consumption, and temperature.

  - Timing Report:
    - The timing report includes results for total delay, net delay, and logic delay of the longest path.
    - The report provides results for applying the voter on each of the modules (PC, Register File, Instruction Memory, Data Memory) individually, and then on all modules together.
    - Timing analysis was conducted with the opt_design option enabled and disabled to compare the impact of design optimization.
  - Power Consumption:
    - Power analysis was performed to measure the power consumption of the system with TMR implemented.
    - The power report includes static and dynamic power components.
  - Temperature:
    - Thermal analysis was conducted to ensure the system operates within safe temperature limits.

- Collected Results Data
  - Timing Analysis:
    - Total Delay: The total delay measured the overall time taken for data to propagate through the system.
    - Net Delay: The net delay included the propagation delay through the interconnecting wires.
    - Logic Delay: The logic delay represented the time taken by the combinational logic (e.g., the majority voter).
  - Timing results were collected under two conditions:
    - Without opt_design:
    - Individual module application: Reported delays for each module with TMR.
    - Full system application: Reported delays for the entire RISC-V 32I architecture with TMR.
    - With opt_design:
    - Individual module application: Reported delays for each module with TMR, showing improvements due to optimization.
    - Full system application: Reported delays for the entire RISC-V 32I architecture with TMR, showing overall performance improvement.
  - Power Consumption:
    - The power report indicated an increase in power consumption due to the triplication of components and the addition of the majority voter.
    - The optimized design (opt_design enabled) showed a slight reduction in power consumption compared to the non-optimized design.
  - Temperature:
    - The thermal report confirmed that the system operated within safe temperature limits under both conditions.
    - The temperature slightly increased with the implementation of TMR due to higher power consumption but remained within acceptable bounds.

3 Results

1. Figure 6 shows the longest path logic delay when each of the voters is applied alone on RISC-V in two cases: opt_design enabled and opt_design disabled. It can be seen that the Instruction memory voter is responsible for the huge increase in the logic delay. When opt_design was disabled the overall logic delay increased by 121.4%, while when opt_design was enabled, the overall delay decreased by 0.05%.

2. Figure 7 shows the longest path net delay when each of the voters is applied alone on RISC-V in two cases: opt_design enabled and opt_design disabled. When opt_design was disabled the overall logic delay increased by 38.4%, while when opt_design was enabled, the overall delay increased by 63.9%.

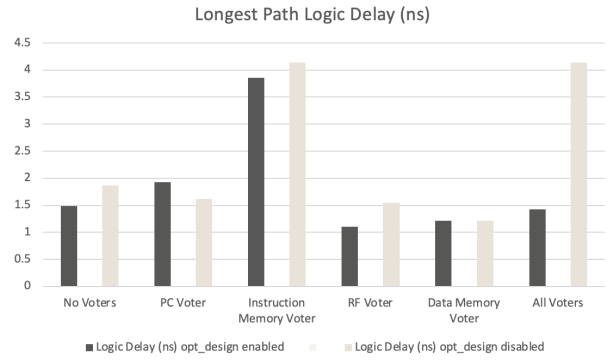

Figure 6: Longest Path Logic Delay

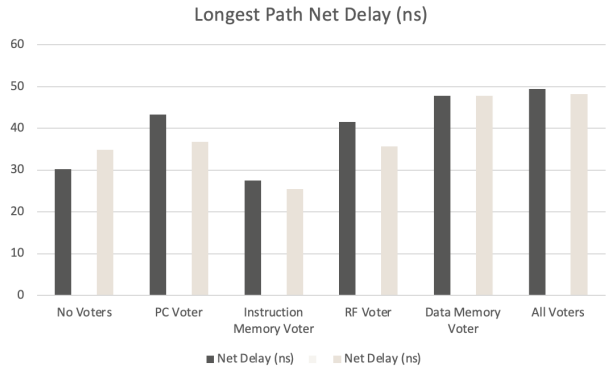

Figure 7: Longest Path Net Delay

3. Figure 8 shows the longest path total delay when each of the voters is applied alone on RISC-V in two cases: opt_design enabled and opt_design disabled. When opt_design was disabled the overall logic delay increased by 42.7%, while when opt_design was enabled, the overall delay increased by 60.6%.

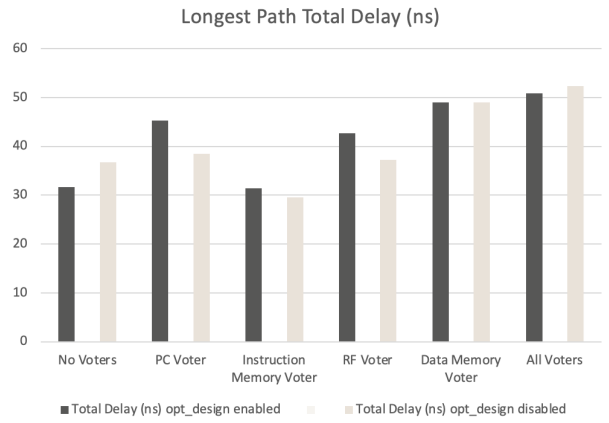

Figure 8: Longest Path Total Delay

4. Figure 9 shows the power consumption when each of the voters is applied alone on RISC-V and all of them combined. Figure 9 indicates an increase in the overall power consumption with 17.1%.

5. Figure 10 shows all RISC-V Voters' on-chip power distribution. 95% of the on-chip

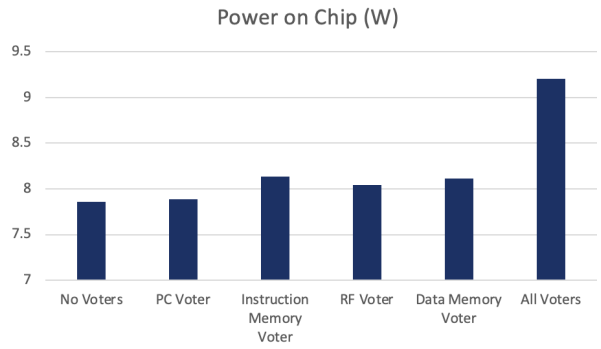

Figure 9: Power Consumption

power was dynamic and 5% was static. The dynamic power was 8.732 W: 35% for Signals and 65% for Logic.

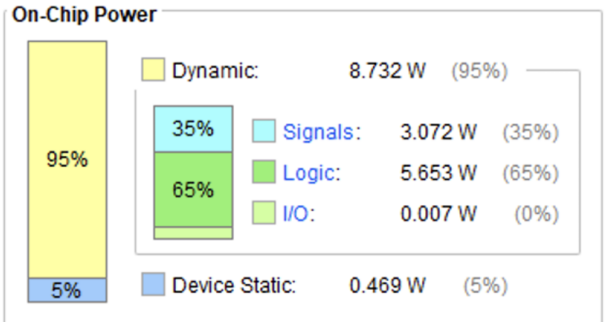

Figure 10: All RISC-V Voters Power Distribution

6. Figure 11 shows the Junction Temperature when each of the voters is applied alone on RISC-V and all of them combined. Figure 11 indicates an increase in the overall Junction Temperature with 5.3%.

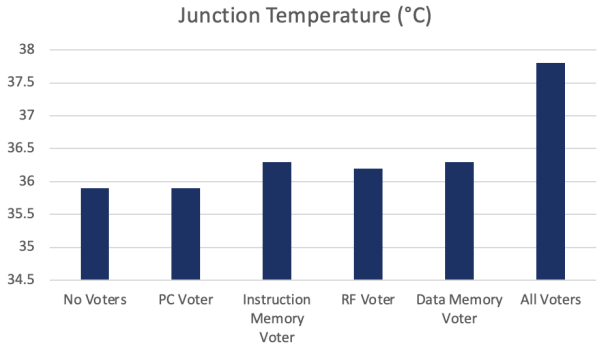

Figure 11: Junction Temperature

4 Discussion

The implementation of Triple Modular Redundancy (TMR) in the RISC-V 32I architecture demonstrated significant impacts on various performance metrics, including logic

delay, net delay, total delay, power consumption, and temperature. The results obtained from the simulation using Vivado provide insights into the trade-offs between enhanced fault tolerance and performance overhead.

Longest Path Logic Delay

Figure 6 illustrates the longest path logic delay when each of the voters is applied individually to the RISC-V architecture, both with and without the opt_design optimization enabled. The most notable increase in logic delay is attributed to the voter applied to the Instruction Memory. When opt_design was disabled, the logic delay surged by 121.4%, indicating a significant performance hit due to the complexity introduced by the voter. However, when opt_design was enabled, the logic delay was reduced slightly by 0.05%, suggesting that design optimization can effectively mitigate some of the performance penalties associated with TMR.

Longest Path Net Delay

Figure 7 shows the longest path net delay under similar conditions. The net delay increased by 38.4% without opt_design, reflecting the additional routing and interconnect complexities introduced by the triplicated components and majority voter. With opt_design enabled, the net delay further increased by 63.9%, highlighting that while optimization improves logic performance, it may inadvertently increase routing delays due to the compact placement and increased fan-out of signals.

Longest Path Total Delay

The total delay, which encompasses both logic and net delays, is depicted in Figure 8. Without opt_design, the total delay increased by 42.7%, whereas with opt_design enabled, the increase was more pronounced at 60.6%. This indicates that while optimization can balance some aspects of performance, the overall complexity added by TMR still results in a noticeable increase in total delay.

Power Consumption

Figure 9 presents the power consumption data for the RISC-V architecture with each voter applied individually and all voters combined. The introduction of TMR led to a 17.1% increase in overall power consumption. This rise is expected due to the triplication of components and the additional logic required for the majority voting mechanism. Notably, dynamic power, which accounts for 95% of the total power, was significantly higher than static power, as shown in Figure 10. The dynamic power, primarily consumed by signals and logic, reached 8.732 W, with 35% attributed to signals and 65% to logic operations.

Junction Temperature

The thermal impact of TMR is illustrated in Figure 11, which shows a 5.3% increase in junction temperature when all voters are applied. This increase is a direct consequence of the higher power consumption and increased activity within the chip. Although the temperature rise is moderate, it emphasizes the need for efficient thermal management in systems employing TMR to prevent overheating and ensure reliable operation.

5 Conclusion

The implementation of Triple Modular Redundancy (TMR) in the single-cycle RISC-V 32I architecture has proven effective in enhancing fault tolerance and system reliability. By addressing bit flip errors in critical data storage components such as the Program Counter (PC), Register File, Instruction Memory, and Data Memory, TMR ensures the

system maintains accurate operation even in the presence of faults. This reliability is crucial for high-availability systems where data integrity is paramount.

Our simulation and analysis using Xilinx Vivado revealed that TMR introduces significant impacts on various performance metrics. The longest path logic delay showed a substantial increase, especially when the voter was applied to the Instruction Memory, with a 121.4% rise observed when opt_design was disabled. However, enabling design optimization slightly mitigated this delay, reducing it by 0.05%. This indicates that optimization techniques can be effective in managing the complexities introduced by TMR, although the trade-offs remain.

The net and total delays were also affected, with notable increases due to the added complexity of triplicated components and majority voting logic. The net delay increased by 63.9% with opt_design enabled, while the total delay saw a 60.6% rise. These results highlight the balance between enhanced reliability and the performance overheads introduced by TMR. Power consumption analysis indicated a 17.1% increase, primarily due to dynamic power consumed by signals and logic. Additionally, the thermal impact was evident, with a 5.3% increase in junction temperature, underscoring the importance of efficient thermal management in TMR systems.

The study confirms that while TMR significantly enhances fault tolerance, it does come with performance and power penalties. Design optimization techniques can partially alleviate these issues, balancing the trade-offs between reliability and efficiency. This makes TMR a viable solution for high-reliability applications where data integrity is critical.

Future work could focus on implementing TMR in a pipelined version of the RISC-V architecture. A pipelined RISC-V processor would potentially offer better performance by overlapping the execution of multiple instructions, thereby reducing the performance overhead associated with TMR in a single-cycle design. Exploring advanced optimization techniques and alternative fault-tolerant strategies in a pipelined architecture could further improve the balance between reliability and performance. This study paves the way for the adoption of robust fault-tolerant systems in open-source architectures like RISC-V, making them suitable for critical applications requiring high levels of data integrity and system reliability.

References

[Ali23] W. Ali. "Exploring Instruction Set Architectural Variations: x86, ARM, and RISC-V in Compute-Intensive Applications". In: *Eng OA* 1.3 (2023), pp. 157–162.

[Blo24] Paessler Blog. *RISC-V vs. ARM: Who Wins?* Online. 2024. URL: https://blog.paessler.com/risc-v-vs-arm-who-wins.

[Cai24] American University in Cairo. *Course Catalog: CSCI 4336/4536 - RISC-V Architecture*. Online. 2024. URL: https://catalog.aucegypt.edu/preview_course_nopop.php?catoid=36&coid=82235.

[Fou24] RISC-V Foundation. *Specifications*. Online. 2024. URL: https://riscv.org/technical/specifications/.

[HP18] John L. Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Sixth Edition. Morgan Kaufmann, 2018.

[Jou24] Tech Journeyman. *ARM vs. RISC-V*. Online. 2024. URL: https://techjourneyman.com/blog/arm-vs-risc-v/.

[Mak24] MakeUseOf. *RISC-V vs. ARM: What is the Difference?* Online. 2024. URL: https://www.makeuseof.com/risc-vs-arm-what-is-the-difference/#:~:text=The%20ARM%20ISA%20allows%20Arm,chips%20without%20paying%20license%20fees..

[PH17] David Patterson and John L. Hennessy. *Computer Organization and Design: RISC-V Edition*. Morgan Kaufmann, 2017.

[Rad24] Radiolab. *Bit Flip*. Podcast. 2024. URL: https://radiolab.org/podcast/bit-flip.

A Code samples

RISC-V Processor Module

```

(*DONT_TOUCH = "yes"*)

module processor(input clk, rst, output [31:0] RDout);
    wire [31:0] PCout,rData1, aluMUX_out, immRes, data_out, alu_out,
    inst, PCin, rData2;
    wire [32:0] branch_sum, PC_counter;
    wire [1:0] jump;
    wire [2:0] ALUop;
    wire regWrite, memRead, MemToReg, memWrite,Branch,ALUsrc,
    output_branch, terminate, carryFlag, zeroFlag, overflowFlag,
    signFlag;
    wire [3:0] ALUsel;
    wire [31:0]addMUX1_out, MUX_out, dmemMUX_out;

//wire [31:0] w1,w2,w3,w4,w5;

PC_TMR pc_tmr (clk,rst, PCin, PCout);
//PC pc(clk, rst, 1,PCin, PCout);

InstMem instmem(PCout[15:0], inst);
Reg_File RF( inst[`IR_rs1], inst[`IR_rs2], inst[`IR_rd], clk,
rst, regWrite , RDout, rData1, rData2);
prv32_ALU ALU( rData1,aluMUX_out , inst[`IR_shamt],        alu_out,
carryFlag, zeroFlag, overflowFlag, signFlag, ALUsel);
DataMem datamem(clk, memRead, inst[`IR_funct3],memWrite,
alu_out [7:0], rData2[31:0],data_out [31:0]);


rv32_ImmGen Imm(inst,immRes);
Rd_control RD(inst[`IR_opcode],immRes, PC_counter[31:0],
branch_sum[31:0], dmemMUX_out, RDout);
Control_unit CU(inst[`IR_opcode], terminate, regWrite,
memRead, MemToReg, memWrite,Branch,ALUsrc,ALUop,jump );
ALU_controlUnit ALU_CU( ALUop, inst[`IR_funct3],inst[30],
ALUsel);
Branch_control BC( inst[`IR_funct3], Branch,zeroFlag,
overflowFlag, carryFlag, signFlag, output_branch);


Adder PC_add(PCout,4,0, PC_counter);
Adder branch_add(PCout,immRes,0, branch_sum);
MUX32 PC_MUX (PCout,MUX_out, terminate,PCin);
MUX32 ALU_MUX (immRes,rData2, ALUsrc,aluMUX_out);
```

```
MUX32 dMem_MUX (data_out,alu_out, MemToReg,dmemMUX_out);
MUX32 adderMUX1 (branch_sum [31:0],PC_counter [31:0],
output_branch,addMUX1_out [31:0]);
MUX32_3 MUX(addMUX1_out [31:0],branch_sum [31:0],
alu_out[31:0], jump [1:0], MUX_out [31:0]);

endmodule
```

RISC-V Control Unit Module

```

(*DONT_TOUCH = "yes"*)

`include "defines.v"

module Control_unit(input [4:0] inst,
output reg terminate,
output reg regWrite,
output reg memRead,
output reg MemToReg,
output reg memWrite,
output reg Branch,
output reg ALUsrc,
output reg [2:0]ALUop,
output reg [1:0]jump );

always @(*)begin
//fetching the inst and outputting the signals
//depending on the instruction
  case(inst)
//incase of branch the branch signals is 1 and it reads
//from the memory
  `OPCODE_Branch:begin
                  Branch =1;
                  regWrite =0;
                  MemToReg=0;
                  memWrite=0;
                  memRead=1;
                  ALUsrc=0;
                  terminate =0;
                  jump = 2'b00;
                  ALUop= 3'b001;
                  end
//in case of load it reads from the memory and we allow
//writing to the register and we use the immediate generator
   `OPCODE_Load :begin
                  Branch =0;
                  regWrite =1;
                  MemToReg=1;
```

```
                    memWrite=0;
                    memRead=1;
                    ALUsrc=1;
                    terminate =0;
                    jump = 2'b00;
                    ALUop= 3'b000;
                    end
// incase of store we allow writing to the memory and
//register and we use the immediate generator

  `OPCODE_Store :begin
                    Branch =0;
                    regWrite =0;
                    MemToReg=1;
                    memWrite=1;
                    memRead=0;
                    ALUsrc=1;
                    terminate =0;
                    jump = 2'b00;
                    ALUop= 3'b000;
                    end
// incase of JALR we allow writing to the register,
//we use the immediate generator and we make the jump 2
//which we assigned to JALR

    `OPCODE_JALR  :begin
                    Branch =0;
                    regWrite =1;
                    MemToReg=0;
                    memWrite=0;
                    memRead=0;
                    ALUsrc=1;
                    terminate =0;
                    jump = 2'b10;
                    ALUop= 3'b000;
                    end
// incase of JAL we allow writing to the register,
//we use the immediate generator and we make the
//jump 1 which we assigned to JAL

`OPCODE_JAL  :begin
                    Branch =0;
                    regWrite =1;
                    MemToReg=0;
                    memWrite=0;
                    memRead=0;
                    ALUsrc=1;
                    terminate =0;
```

```
                    jump = 2'b01;
                    ALUop= 3'b000;
                    end
// incase of Arithmatic I we allow writing to the
//register, we use the immediate generator

`OPCODE_Arith_I  :begin
                    Branch =0;
                    regWrite =1;
                    MemToReg=0;
                    memWrite=0;
                    memRead=0;
                    ALUsrc=1;
                    terminate =0;
                    jump = 2'b00;
                    ALUop= 3'b111;
                    end
// incase of Arithmatic R we allow writing to the register

`OPCODE_Arith_R  :begin
                    Branch =0;
                    regWrite =1;
                    MemToReg=0;
                    memWrite=0;
                    memRead=0;
                    ALUsrc=0;
                    terminate =0;
                    jump = 2'b00;
                    ALUop= 3'b010;
                    end
   // incase of AUIPC we allow writing to the register

 `OPCODE_AUIPC  :begin
                    Branch =0;
                    regWrite =1;
                    MemToReg=0;
                    memWrite=0;
                    memRead=0;
                    ALUsrc=0;
                    terminate =0;
                    jump = 2'b00;
                    ALUop= 3'b000;
                    end
// incase of Load upper immidiate we allow writing to the
//register, we use the immediate generator

 `OPCODE_LUI  :begin
                    Branch =0;
```

```
                    regWrite =1;
                    MemToReg=0;
                    memWrite=0;
                    memRead=0;
                    ALUsrc=1;
                    terminate =0;
                    jump = 2'b00;
                    ALUop= 3'b100;
                    end
// incase of system we make the program use the same
//PC without anychange

`OPCODE_SYSTEM :begin
                    Branch =0;
                    regWrite =0;
                    MemToReg=0;
                    memWrite=0;
                    memRead=0;
                    ALUsrc=0;
                    terminate =1;
                    jump = 2'b00;
                    ALUop= 3'b110;
                    end
    endcase

end

endmodule

```

RISC-V Register File Module

```
(*DONT_TOUCH = "yes"*)

module Reg_File(input[4:0] rs1, rs2, rd, input clk,
rst, regwrite ,input [31:0] writeData, output [31:0]rData1, rData2);

    wire [31:0] registers [0:31];
//    wire [31:0] registers1 [0:31];
//    wire [31:0] registers2 [0:31];
//    wire [31:0] registers3 [0:31];

    wire [31:0] load;
//assigning the load if and only if the regWrite is 1
    assign load = regwrite ?( (rd!=0) ?(1 << rd):(0) ) : 0;
// loop to iterate the 32 bits
 generate
    genvar i;
```

```
17          for (i=0; i<32; i=i+1) begin
18
19       Register register ( clk,  rst,  load[i],writeData, registers[i]);
20  //    Register register1 ( clk,  rst,  load[i],writeData, registers1[i]);
21  //    Register register2 ( clk,  rst,  load[i],writeData, registers2[i]);
22  //    Register register3 ( clk,  rst,  load[i],writeData, registers3[i]);
23  //    TMR t2( registers1[i],registers2[i],registers3[i], registers[i]);
24
25          end
26      endgenerate
27      assign rData1 = registers[rs1];
28      assign rData2 = registers[rs2];
29
30  endmodule
31
32
```