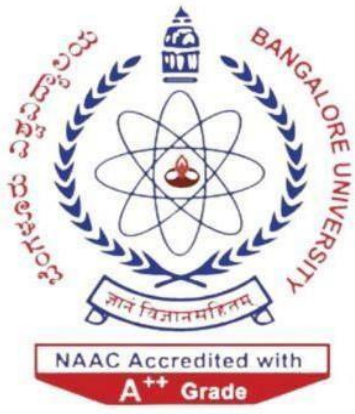
A Project Report

On

5 STAGE PIPELINE RISC V PROCESSOR WITH FAULT TOLERANT ALU

Submitted by

Abhishek A S	Gowtham J S	Gururaj D	Lakshmeesha
U03NM22T043001	U03NM22T043004	U03NM22T043005	U03NM22T043009
VII Semester, ECE	VII Semester, ECE	VII Semester, ECE	VII Semester, ECE



Under the Guidance of

Deepthi

Guest Faculty

Department of Electronics and Communication Engineering

Bangalore University

University Visvesvaraya College of Engineering

K.R. Circle, Bangalore – 560001

Bangalore University

University Visvesvaraya College of Engineering

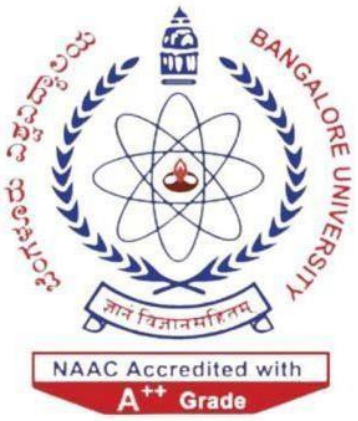
K.R. Circle, Bangalore – 560001

Department of Electronics and Communication Engineering

DECLARATION

We the students of VIII Semester B.Tech. in Electronics and Communication Engineering of University College of Engineering, Bangalore, hereby declare that the project work entitled **"5 STAGE PIPELINE RISC V PROCESSOR WITH FAULT TOLERANT ALU"** has been carried out and submitted in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Electronics and Communication Engineering of Bangalore University, during the academic year 2024 - 2025.

NAME	USN	SIGNATURE
Abhishek A S	U03NM22T043001	
Gowtham J S	U03NM22T043004	
Gururaj D	U03NM22T043005	
Lakshmeesha	U03NM22T043009	



Department of Electronics and Communication Engineering

CERTIFICATE

This is to certify that **Abhishek A S (U03NM22T043001), Gowtham J S (U03NM22T043004), Gururaj D (U03NM22T043005)** and **Lakshmeesha (U03NM22T043009)** have successfully completed the Project work entitled **"5 STAGE PIPELINE RISC V PROCESSOR WITH FAULT TOLERANT ALU"**, in Partial Fulfillment for the Requirement of the **Project (21ECPW705L)** of VII Semester Electronics and Communication Engineering prescribed by the Bangalore University during the Academic Year 2024 - 2025.

Guide:

Deepthi
Department of ECE
UVCE

Chairperson:

Dr. Kiran K
Chairperson
Department of ECE
UVCE

Examiners:

1. ……

2. ...…….

ACKNOWLEDGEMENT

The knowledge and satisfaction that accompanies the successful completion of any task would be incomplete without the mention of the people who made it possible, whose guidance and encouragement crowned our effort with success. We would like to thank and acknowledge the help we have received to carry out this project.

We are grateful to **Dr Subhasish Tripathy**, director, University Visvesvaraya College of Engineering for being kind enough by giving this opportunity and platform to present our final year project.

We would also like to thank **Dr. Kiran K**, Professor and Chairperson of the department of Electronics and Communication Engineering, University Visvesvaraya College of Engineering, for his constant encouragement and insightful discussions.

We would also like to extend our thanks to our Guide, **Deepthi**, Guest Faculty, Department of Electronics and Communication Engineering, University Visvesvaraya College of Engineering, for her guidance and insightful discussions.

We wish to express our immense gratitude for the friendly cooperation shown by all the staff members of the department of Electronics and Communication Engineering, UVCE.

We would also take this opportunity to thank our friends and family for their constant support and help.

ABSTRACT

This project focuses on the design and implementation of a five-stage pipelined RISC-V processor architecture, integrated with a fault-tolerant Arithmetic Logic Unit (ALU) to ensure reliable operation even in the presence of hardware faults. The architecture strictly follows the RV32I instruction set architecture (ISA), which is a 32-bit base ISA designed to support efficient, open, and extensible processor implementations.

The processor is built using a classical five-stage pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage was modularly designed and verified to ensure correct data and control signal flow across the pipeline. Pipelining allows the processor to achieve higher throughput by overlapping the execution of multiple instructions, thus significantly improving performance compared to single-cycle designs. However, pipelining introduces challenges such as data hazards and control hazards, which are mitigated in this design through the implementation of a Hazard Detection Unit (HDU) and a Forwarding Unit. These units ensure smooth execution by managing instruction dependencies and enabling bypassing of data between stages without stalling the pipeline unnecessarily.

A standout feature of this processor is the integration of a fault-tolerant ALU using the Triple Modular Redundancy (TMR) technique. In this setup, three identical ALUs perform the same computation in parallel. Their outputs are compared using majority voting logic to determine the correct result. If any one ALU produces an incorrect output due to a transient or permanent fault, the system continues to operate correctly by selecting the majority result. Additionally, an error_flag signal is raised to indicate the detection of a fault. This mechanism provides robust protection against soft errors caused by radiation or power fluctuations, which are common in environments like space or industrial control systems.

The entire design is implemented in System Verilog and simulated using the Vivado Design Suite, a comprehensive FPGA development tool by Xilinx. Vivado was instrumental in all phases of the project, including writing HDL code, performing behavioral simulations, synthesizing the design, and analyzing waveforms for verification. Simulation waveforms confirmed the proper operation of the pipelined processor and the effectiveness of the TMR-based ALU in maintaining correct outputs despite injected faults.

INTRODUCTION

The evolution of semiconductor technology has been pivotal in transforming the modern digital world. Since the advent of Very Large-Scale Integration (VLSI) in the 1970s, it has become possible to integrate millions of transistors onto a single chip, dramatically improving computational capability, reducing power consumption, and enhancing system reliability. VLSI technology has enabled the development of compact and powerful embedded systems, and its applications have grown extensively in fields such as aerospace, defence, automotive electronics, biomedical devices, and consumer electronics. The increasing demand for intelligent, portable, and real-time systems has further fueled the need for customized processors capable of delivering high performance while maintaining low power and fault resilience.

One of the central components of any digital computing system is the processor, often referred to as the "brain" of the system. It controls the flow of data, executes instructions, and facilitates interaction with peripheral devices. Traditional processor architectures are broadly classified into Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). RISC architectures simplify the instruction set, allowing for faster instruction execution and more efficient pipeline implementation. Among modern RISC-based architectures, RISC-V has gained significant traction due to its open-source nature, modular design, and flexibility for research and industrial use.

Introduced in 2010 by researchers at the University of California, Berkeley, RISC-V was designed from the ground up to be a clean-slate ISA that is extensible, power-efficient, and ideal for modern applications. Unlike proprietary ISAs, RISC-V eliminates licensing costs and restrictions, enabling broader adoption in academia and industry. Its modular architecture allows designers to implement only the required base and optional instruction extensions, offering significant advantages in custom processor design, particularly for embedded systems, AI accelerators, and energy-constrained applications.

One of the most important architectural advancements in processor design is pipelining, which increases instruction throughput by overlapping the execution of multiple instructions. The classical 5-stage RISC pipeline consists of the following stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).

By dividing instruction processing into discrete stages and using pipeline registers, the processor can execute one instruction per clock cycle after an initial latency, thereby significantly improving performance. However, pipelining introduces its own challenges such as data hazards, control hazards, and structural hazards, which must be addressed using techniques like forwarding, stalling, and hazard detection logic.

In safety-critical systems such as those found in aerospace, automotive, and industrial automation, fault tolerance is a key design requirement. As transistor sizes shrink and operating voltages decrease, circuits become increasingly susceptible to soft errors due to radiation, power fluctuations, and manufacturing variability. One of the most effective techniques to enhance fault resilience in critical components is Triple Modular Redundancy (TMR). In TMR, three identical instances of a module (e.g., an ALU) perform the same computation, and a majority voter circuit selects the correct output. This method ensures continued operation even in the presence of single faults, thereby improving reliability and robustness.

This project presents the design and implementation of a RISC-V based 5-stage pipelined processor integrated with a fault-tolerant Arithmetic Logic Unit (ALU). The processor strictly adheres to the RV32I instruction set, with each pipeline stage developed and verified individually before integration. The fault-tolerant ALU is designed using the TMR approach, providing resilience against both transient and permanent faults without compromising on performance. The overall design is written in System Verilog, simulated using Vivado, and validated using custom testbenches and instruction-level test cases.

The objective of this work is twofold: to demonstrate a full implementation of a pipelined RISC-V processor and to enhance its reliability by introducing fault tolerance in the arithmetic core. The integration of pipelining and TMR in a modular RISC-V architecture reflects current trends in reconfigurable, high-performance, and mission-critical computing platforms. This project lays the groundwork for future extensions including superscalar execution, low-power techniques, and AI-specific instruction support, particularly in the context of aerospace applications and AI edge processing.

LITERATURE SURVEY

The RISC-V Instruction Set Architecture (ISA), as defined in the RISC-V User-Level ISA Specification Version 2.2, represents a modern, clean-slate ISA that is freely available under open-source licenses. Its design philosophy emphasizes simplicity, modularity, and extensibility—making it a compelling alternative to legacy proprietary ISAs such as ARM and x86. The specification defines a small but powerful base instruction set, RV32I, which is tailored for 32-bit integer operations and serves as the foundation for more advanced extensions such as multiplication/division (M), atomic operations (A), floating-point (F/D), and compressed instructions (C).

The RV32I base ISA includes 47 well-defined instructions that handle a wide range of operations including arithmetic, logical, load/store, control transfer, and system-level functionality. It features a fixed instruction length of 32 bits, uniform register usage, and a consistent three-operand instruction format, all of which simplify instruction decoding and hardware implementation. The use of 32 general-purpose registers further enhances performance and design regularity.

The specification outlines detailed formats for R-type, I-type, S-type, B-type, U-type, and J-type instructions, including how immediate values are encoded and sign-extended. This consistency enables designers to build decoders and control units with minimal logic overhead. Additionally, the manual specifies well-defined behaviour for exceptions, system calls, and environment transitions, ensuring compatibility across a wide range of platforms.

Single cycle RISC-V micro architecture processor and its FPGA prototype by D. K. Dennis et al 2017 7th International Symposium on Embedded Computing and System Design (ISED), 2017, pp. 1-5, doi: 10.1109/ISED.2017.8303926.

This work describes the creation of a fully synthesizable 32-bit processor using the open-source RISC-V (RV32I) ISA. This CPU was created with low-cost embedded devices in mind. This document also includes a RISC-V development and validation framework, as well as assembling tools and automated test suites. The result is a single-core, in-order, RISC-V processor with low hardware complexity that is not bus-based. The suggested processor is written in Verilog HDL and then prototyped on an FPGA board called the "Spartan 3E

XC3S500E." The maximum functioning frequency was discovered to be 32MHz. Using the Xilinx Power Analyzer, the power is assessed to be 7.9mW.

Advantages: -

- It is based on the latest 32-bit RISC V ISA. Hence consuming less LUTs.

Disadvantages: -

- No hazard avoidance unit.
- Non-pipelined hence less throughput.

Design of an 8-bit five stage pipelined RISC microprocessor for sensor platform application IEEE by R. J. L. Austria, A. L. Sambile, K. M. Villegas and J. N. T. Tabing, TENCON 2017 - 2017 Region 10 10.1109/TENCON.2017.8228209. Conference, 2017, pp. 2110-2115, doi:

This study describes a low-power pipelined 64-bit RISC processor with Floating Point Unit based on FPGA technology. This processor is designed for fixed- and floating-point numerical arithmetic, as well as branch and logical tasks. Because dynamic branch prediction is used to achieve pipelining, it will not flush when a branch instruction is sent. This will improve the efficiency of the instruction stream. Using the clock gating technique in RTL coding, one can lower the dynamic power. Addition, subtraction, multiplication, and division are also implemented in this work using Double Precision floating point arithmetic. Because floating point operations are used in many applications, such as signal processing, graphics, and medicine, this architecture has become vital and increasingly important. The hardware description contains all of the necessary code.

Advantages:-

- It is a pipelined processor hence more throughput.
- It is synthesized using Synopsys tool.

Disadvantages:-

- It is based on a older ISA. Hence more LUT consumption than RISC-V.
- It is a 8bit processor hence more memory latency.

Design and Implementation of 32-bit RISC-V Processor Using Verilog by Manjusha Rao P; Prabha Niranjan; Dileep Kumar M J .

The design and implementation of a 32-bit single-cycle RISC-V processor in Verilog is a sophisticated and elaborate process that aims to create a functioning processor architecture that adheres to the RISC-V instruction set. To execute instructions in a single clock cycle, this research work requires the synthesis of components such as the program counter, register file, arithmetic logic unit (ALU), and memory modules. The Verilog-based implementation includes RISC-V instructions such as arithmetic, memory access, and control flow instructions. The design prioritizes simplicity and clarity, laying the groundwork for educational study and the eventual development of more advanced processing functionality. This emphasizes importance in understanding processor architecture and Verilog-based processor design.

Fault Tolerant ALU System by Ayon Majumdar; Sahil Nayyar; Jitendra Singh Sengar

Majumdar et al. (2012) proposed a fault-tolerant ALU using Triple Modular Redundancy (TMR) to enhance processor reliability. Their Verilog-based design uses three identical ALUs and a voting circuit to mask faults. A disagreement detector identifies faulty modules without interrupting system functionality. Simulation results confirm effective fault masking. The study emphasizes TMR's practicality in critical applications like aerospace and embedded systems.

Design a 5-stage pipeline RISC-V CPU and optimise its ALU by Lifu Deng Glasgow College, University of Electronic Science and Technology of China, Chengdu, China

Lifu Deng (2023) designed a 5-stage pipelined RISC-V processor using Verilog and implemented it in Vivado 2022.2, focusing on both pipeline integration and ALU optimization. The processor supports 37 RV32I base instructions and one multiplication extension, making it compatible with standard RISC-V toolchains. The ALU was enhanced using carry look-ahead adders (CLAs), barrel shifters, and a Booth-Wallace multiplier for better performance. Resource usage was analysed, showing reduced LUT occupancy due to efficient shifter design. The CPU's performance was verified using testbenches and waveform analysis. The study reinforces the practical value of RISC-V in SoC and embedded systems, showing its adaptability and open-source strength. Vivado's synthesis tools confirmed RTL correctness and system stability. The project demonstrates how combining pipelining and ALU optimization significantly boosts instruction throughput. This research is a key reference for RISC-V-based CPU design for IoT, embedded, and high-performance applications.

RISCV INSTRUCTION SET ARCHITECTURE

The RISC-V (RV32I) instruction set has a fixed length of 32 bits, which must be aligned to 32 bit boundaries. It is designed to form a sufficient compile target and support modern OS environments. It was constructed in a way that it reduces the hardware needed for minimum implementation. It follows a little-endian format, where the lowest address contains the LSB part of the particular word. The format used in this project is RV32I which is v2.0 of RISC V. Which is an optimized ISA for creating RISC machines. This ISA can support almost all modern operations and features. It has 32 general purpose registers reg0 to reg31 and reg0 is hardwired to the constant 0. There is one additional user accessible register known as the program counter that holds the address of the next instruction to execute. The program counter is of 32 bits in length, at the positive edge of the clock the program counter is incremented. The number of the increment value is dependent on the instruction memory. In this project the instruction memory is word addressable so the program counter is incremented by one. RISC V was mainly chosen because it can be easily pipelined and consumes less hardware and power. Since it is mainly software oriented, we need techniques like loop unrolling and compiler scheduling to optimize the run time of this processor. There are six instruction formats in the RV32I instruction set: R-type, U-type, I-type, B type, J-type. and S-type shown in Figure 3.2. All of the types are explained in the following section

R-type RV32I Instruction Format

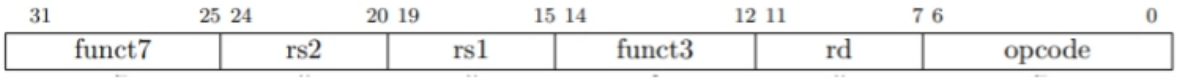
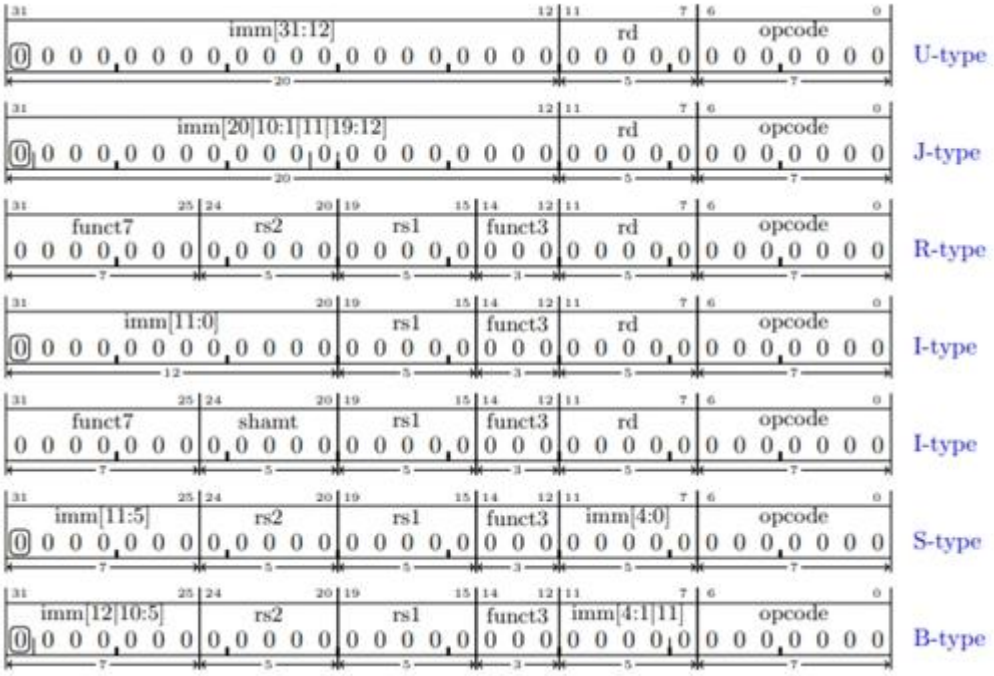


Figure 3.1 depicts the Register-type RV32I ISA V 2.0. It has a total of six fields. Opcode width is 7 bits, which is used to indicate the type of instruction. Source registers (rs1, rs2) and destination register (rd) are indicated by five-bit fields. The function field is of a total of 10 bits, which is used for identifying the type of operation to be performed. The instructions which are supported by this format are add, sub, sltu, sll, xor, and, sra, srl, or, and slt.



3.1.2 I-type RV32I Instruction Format

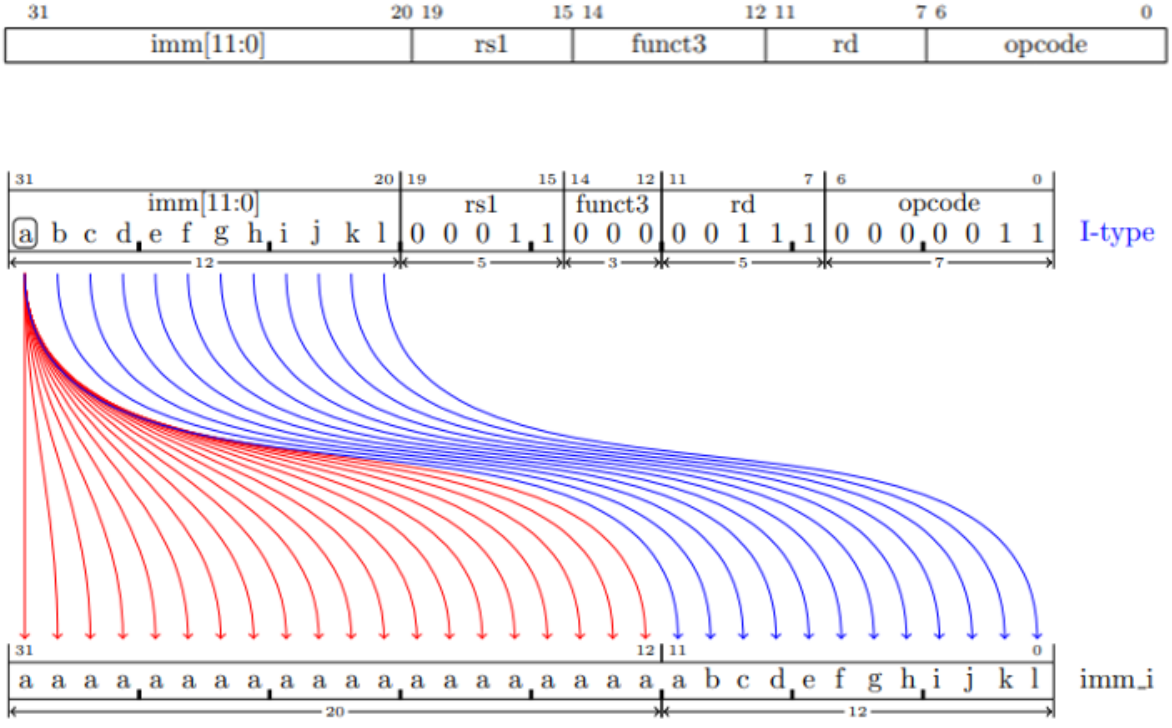


Figure 2.3 depicts the Register-type RV32I ISA V 2.0. Similar to R-type, Opcode width is of 7 bits. Source registers (rs1) and destination register (rd) are indicated by five bit fields. The function field of 3 bits, is used for identifying the type of operation to be performed. It has a separate 12 bit field for holding the immediate operand, used for immediate data operations.

The instructions which are supported by this format are jalr, lhu, lw, lb, lbu, lh, srai, srli, slli, slti, addi, andi, ori, xori and sltiu. Figure 2.4 shows the decoding logic of I type Instruction.

3.1.3 S-type RV32I Instruction Format

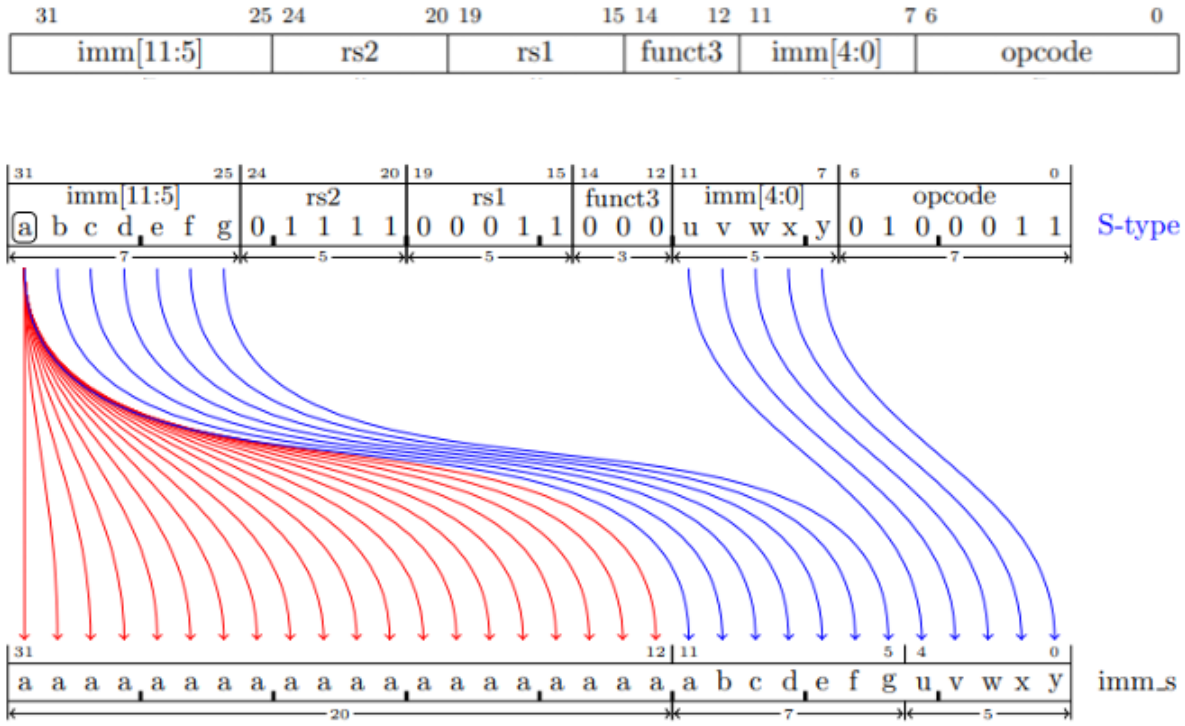
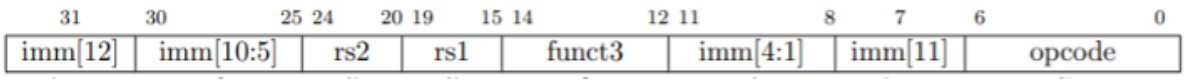


Figure 3.5 depicts the Store-type RV32I ISA V 2.0. Similar to R-type, Opcode width is 7 bits. Source registers (rs1 and rs2) are indicated by five bit fields. Function field is of 3 bits, which is used to indicate the size of the data need to be stored. It has a separate 7+5=12 bit field space for holding the immediate operand. This immediate operand is added with rs1 to calculate the address in which the value rs2 needed to be stored. The instructions which are supported by this format are sw, sb and sh. Figure 2.6 shows the decoding logic of s type Instruction. Figure 3.6 shows the decoding logic of S type Instruction.

3.1.4 B-type RV32I Instruction Format



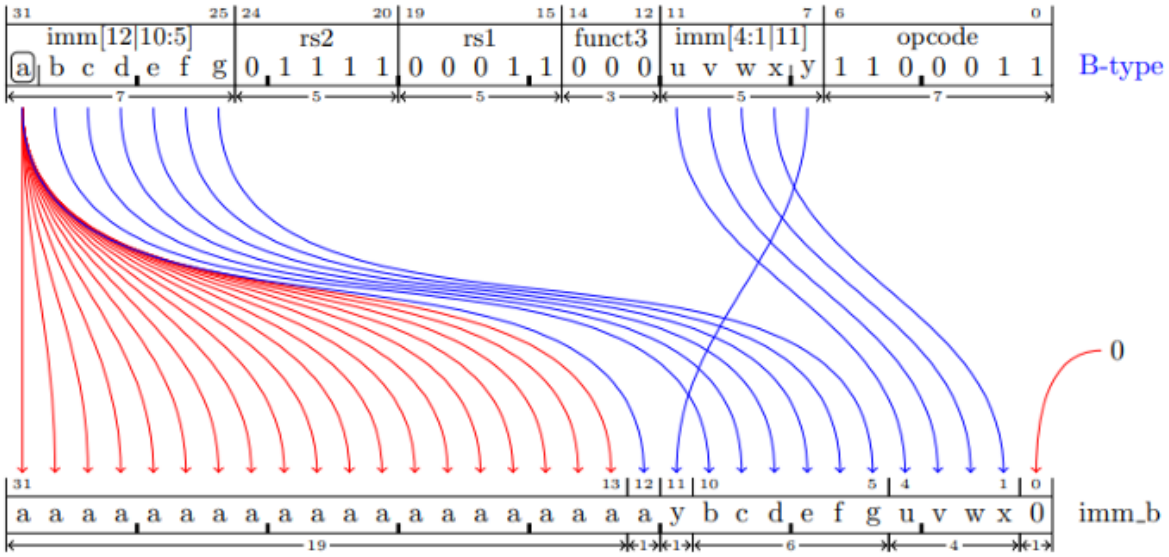
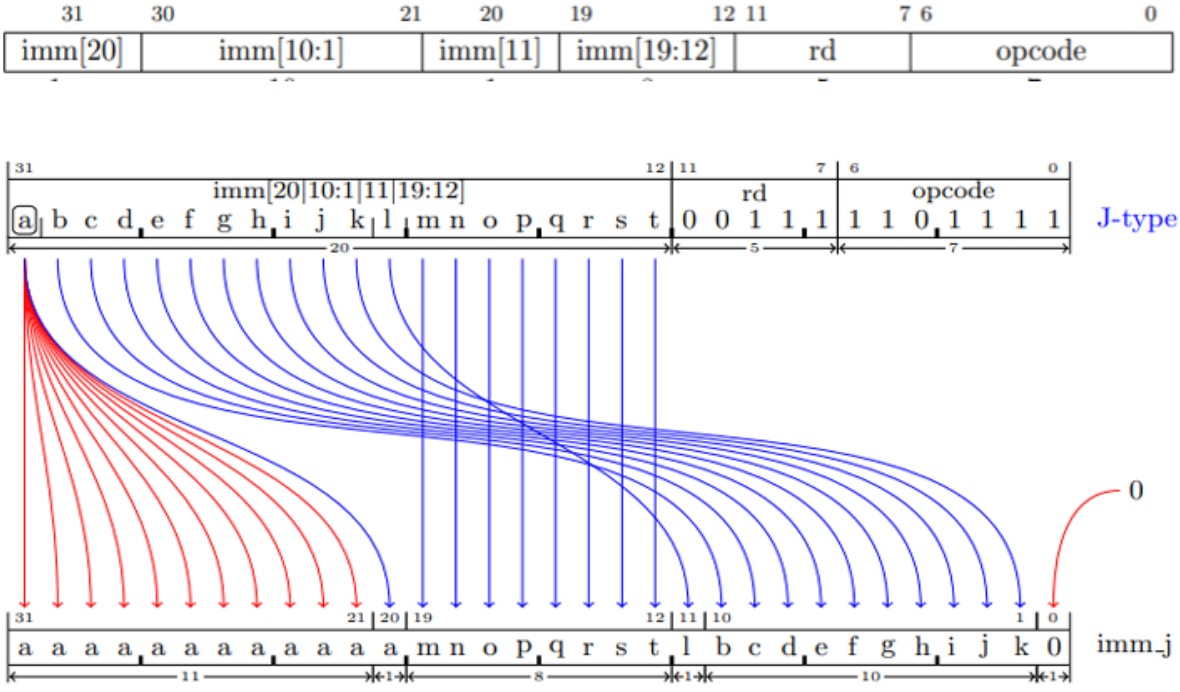


Figure 3.7 shows the Branch-type RV32I ISA V 2.0. Similar to other instructions, the Opcode width is of 7 bits. Source registers (rs1 and rs2) are indicated by five bit fields which are used for comparison for branching. The function field is of 3 bits, which is used to indicate the type of condition that need to be checked for branching. It has a separate 7+5=12 bit field space for holding the immediate operand, which is added to the program counter if a branch is taken. The instructions which are supported by this format are bne, bltu, blt, bgeu, bge and beq. Figure 3.8 shows the decoding logic of B type Instruction

3.1.5 U-type & J-type RV32I Instruction Format



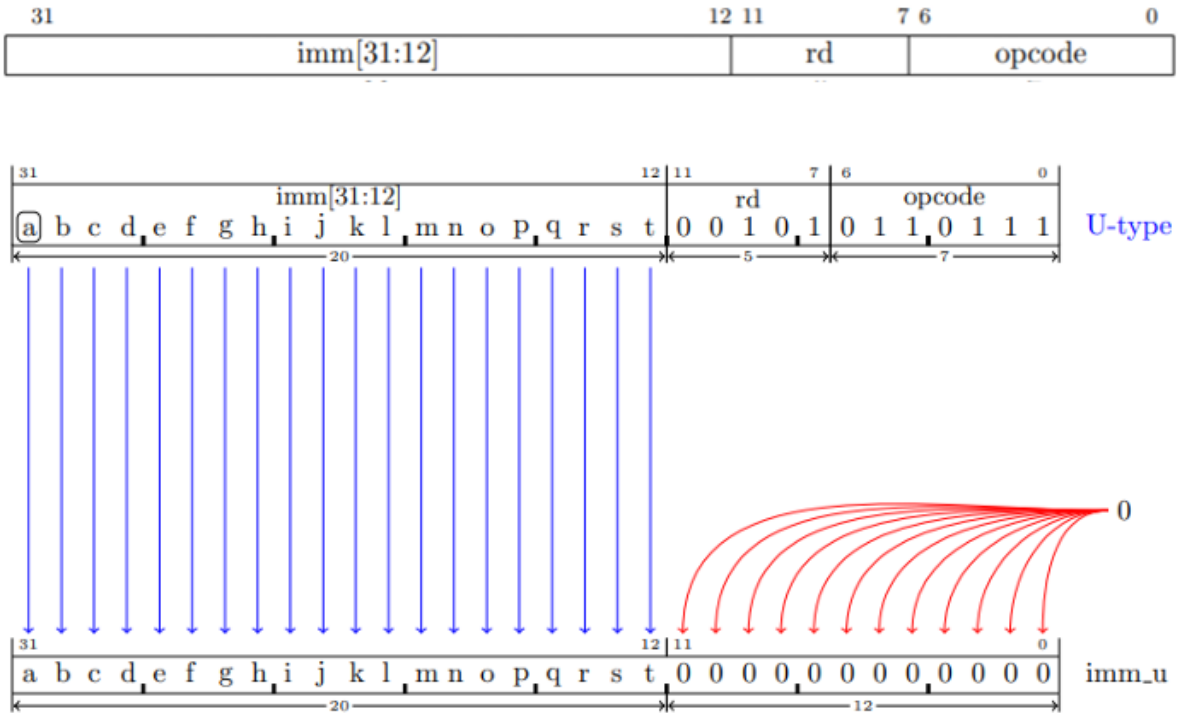


Figure 3.9 & 3.11 shows the U-type and J-type RV32I ISA V 2.0 which are similar to each other. It has a total of two fields. Opcode width is 7 bits, which is used to indicate the type of instruction format. The destination register (rd) is indicated by a five bit field. It has a 20 bit field for holding the immediate operand, used for immediate data operations. For J type the immediate data is rearranged before branching. The instructions which are supported by this format are jal, lui and auipc. Figure 3.10 & 3.12 shows the decoding logic of J & U type Instruction.

3.2.1 Arithmetic Operations

Instruction	Type	Example	Meaning
Add	R	`add rd, rs1, rs2`	`R[rd] = R[rs1] + R[rs2]`
Subtract	R	`sub rd, rs1, rs2`	`R[rd] = R[rs1] - R[rs2]`
Add immediate	I	`addi rd, rs1, imm12`	`R[rd] = R[rs1] + SignExt(imm12)`
Set less than	R	`slt rd, rs1, rs2`	`R[rd] = (R[rs1] < R[rs2])? 1 : 0`
Set less than immediate	I	`slti rd, rs1, imm12`	`R[rd] = (R[rs1] < SignExt(imm12))? 1 : 0`
Set less than unsigned	R	`sltu rd, rs1, rs2`	`R[rd] = (R[rs1] <u R[rs2])? 1 : 0`
Set less than immediate unsigned	I	`sltiu rd, rs1, imm12`	`R[rd] = (R[rs1] <u SignExt(imm12))? 1 : 0`
Load upper immediate	U	`lui rd, imm20`	`R[rd] = SignExt(imm20 << 12)`
Add upper immediate to PC	U	`auipc rd, imm20`	`R[rd] = PC + SignExt(imm20 << 12)`

Table 3.1 shows the list of arithmetic operations supported by the processor. Arithmetic operations are carried out by the ALU in the execution stage of the processor. These operations are carried out on two source operands and the result is written back to the register file at the memory write back stage. The immediate data are sign extended to 32 bits, thus all operations are carried out with respect to 32 bits. All operations are done such that register 1 will always take the left hand side and the register 2 or the immediate data on the right hand side.

3.2.2 Logical Operations

Instruction	Type	Example	Meaning
AND	R	`and rd, rs1, rs2`	`R[rd] = R[rs1] & R[rs2]`
OR	R	`or rd, rs1, rs2`	`R[rd] = R[rs1] | R[rs2]`
XOR	R	`xor rd, rs1, rs2`	`R[rd] = R[rs1] ^ R[rs2]`
AND immediate	I	`andi rd, rs1, imm12`	`R[rd] = R[rs1] & SignExt(imm12)`
OR immediate	I	`ori rd, rs1, imm12`	`R[rd] = R[rs1] | SignExt(imm12)`
XOR immediate	I	`xori rd, rs1, imm12`	`R[rd] = R[rs1] ^ SignExt(imm12)`
Shift left logical	R	`sll rd, rs1, rs2`	`R[rd] = R[rs1] << R[rs2]`
Shift right logical	R	`srl rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(logical)*
Shift right arithmetic	R	`sra rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(arithmetic)*
Shift left logical immediate	I	`slli rd, rs1, shamt`	`R[rd] = R[rs1] << shamt`
Shift right logical imm.	I	`srli rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(logical)*
Shift right arithmetic immediate	I	`srai rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(arithmetic)*

Table 3.2 shows the list of Logical operations supported by the processor. Logical operations are carried out by the ALU in the execution stage of the processor. These operations are carried out on two source operands and the result is written back to the register file at the memory write back stage. The immediate data are sign extended to 32 bits, thus all operations are carried out with respect to 32 bits. All operations are done such that register 1 will always take the left hand side and the register 2 or the immediate data on the right hand side.

3.2.3 Data Transfer Operations

Instruction	Type	Example	Meaning
Load doubleword	I	`ld rd, imm12(rs1)`	`R[rd] = Mem8[R[rs1] + SignExt(imm12)]`
Load word	I	`lw rd, imm12(rs1)`	`R[rd] = SignExt(Mem4[R[rs1] + SignExt(imm12)])`
Load halfword	I	`lh rd, imm12(rs1)`	`R[rd] = SignExt(Mem2[R[rs1] + SignExt(imm12)])`
Load byte	I	`lb rd, imm12(rs1)`	`R[rd] = SignExt(Mem1[R[rs1] + SignExt(imm12)])`
Load word unsigned	I	`lwu rd, imm12(rs1)`	`R[rd] = ZeroExt(Mem4[R[rs1] + SignExt(imm12)])`
Load halfword unsigned	I	`lhu rd, imm12(rs1)`	`R[rd] = ZeroExt(Mem2[R[rs1] + SignExt(imm12)])`
Load byte unsigned	I	`lbu rd, imm12(rs1)`	`R[rd] = ZeroExt(Mem1[R[rs1] + SignExt(imm12)])`
Store doubleword	S	`sd rs2, imm12(rs1)`	`Mem8[R[rs1] + SignExt(imm12)] = R[rs2]`
Store word	S	`sw rs2, imm12(rs1)`	`Mem4[R[rs1] + SignExt(imm12)] = R[rs2](31:0)`
Store halfword	S	`sh rs2, imm12(rs1)`	`Mem2[R[rs1] + SignExt(imm12)] = R[rs2](15:0)`
Store byte	S	`sb rs2, imm12(rs1)`	`Mem1[R[rs1] + SignExt(imm12)] = R[rs2](7:0)`

Table 3.3 shows the list of Data transfer operations supported by the processor. Address calculation part is carried out by the ALU in the execution stage of the processor. These operations are carried out on two source operands and the result is written back to the next stages for memory access. The immediate data are sign extended to 32 bits, thus all operations are carried out with respect to 32 bits. All operations are done such that register 1 will always take the left hand side and the register 2 or the immediate data on the right hand side. Load operations are carried out on the write back stage and the store operations are carried on the memory access stage.

3.2.4 Control Transfer Instructions

Instruction	Type	Example	Meaning
Branch equal	SB	`beq rs1, rs2, imm12`	`if (R[rs1] == R[rs2]) pc = pc + SignExt(imm12 << 1)`
Branch not equal	SB	`bne rs1, rs2, imm12`	`if (R[rs1] != R[rs2]) pc = pc + SignExt(imm12 << 1)`
Branch greater than or equal	SB	`bge rs1, rs2, imm12`	`if (R[rs1] >= R[rs2]) pc = pc + SignExt(imm12 << 1)`
Branch greater than or equal unsigned	SB	`bgeu rs1, rs2, imm12`	`if (R[rs1] >=u R[rs2]) pc = pc + SignExt(imm12 << 1)`
Branch less than	SB	`blt rs1, rs2, imm12`	`if (R[rs1] < R[rs2]) pc = pc + SignExt(imm12 << 1)`
Branch less than unsigned	SB	`bltu rs1, rs2, imm12`	`if (R[rs1] <u R[rs2]) pc = pc + SignExt(imm12 << 1)`
Jump and link	UJ	`jal rd, imm20`	`R[rd] = PC + 4` `PC = PC + SignExt(imm20 << 1)`
Jump and link register	I	`jalr rd, imm12(rs1)`	`R[rd] = PC + 4` `PC = (R[rs1] + SignExt(imm12)) & (~1)`

Table 3.4 shows the list of Control Transfer operations supported by the processor. The condition for branching is checked and carried out by the ALU in the execution stage of the processor. These operations are carried out on two source operands and the result is used for setting the taken branch flag. The immediate data are sign extended to 32 bits, thus all operations are carried out with respect to 32 bits. All operations are done such that register 1 will always take the left hand side and the register 2 or the immediate data on the right hand side. The taken branch flag does not allow the following two instructions to make changes to the memory and register file of the processor.

5-STAGE PIPELINE

With traditional single cycle data path, instructions are executed in a single clock cycle. The next instruction will need to wait for the previous instruction to be complete before it can be executed. Moreover, the execution time for each instruction is different. There are instructions that takes longer time to execute than the other. This might lead to wasted clock cycle and reduced performance of the processor. However, this issue can be solved by using pipelining. Pipelining is a technique used in computer architecture to increase the overall performance of a processor. It involves breaking down the execution of a task into smaller stages and allowing these stages to overlap in time. With this, multiple instructions can be executed simultaneously and improve the throughput of the system. The most common used pipeline technique in modern processor design is 5-Stage pipeline technique. As suggested by its name, the instruction execution cycle was divided into 5 different stages. Each stage performs a specific operation on the input data and passes the result to the next stage. The output of the first stage become the input of the second stage, and so on. Besides, every stage operates on a different part of the data, which allow multiple instructions to be in different stages of execution at the same time. The 5 stages of a typical pipeline are: fetch, decode, execute, memory, and writeback. The details for each stage were discussed below.

1. Fetch

In this stage, the processor fetches the instruction from memory. Then, the instruction is loaded into the instruction cache, which is used to store the recently used instructions.

2. Decode

In this stage, the processor decodes the fetched instruction to determine the operation it needs to perform. The instruction is analyzed to determine the type of operation, registers, and the locations of any operands.

3. Execute

In this stage, the processor performs actual operation specified by the instruction. This involves arithmetic or logical operations, such as addition and subtraction. Besides, it also involves memory access or branching to a different part of the program.

4. Memory

In this stage, the processor access memory to read or write the data obtained from previous stage. However, this stage is optional. Some instruction such as “add” does not involve memory access.

5. Writeback

In this stage, the results of the operation are written back to the appropriate register in the register file.

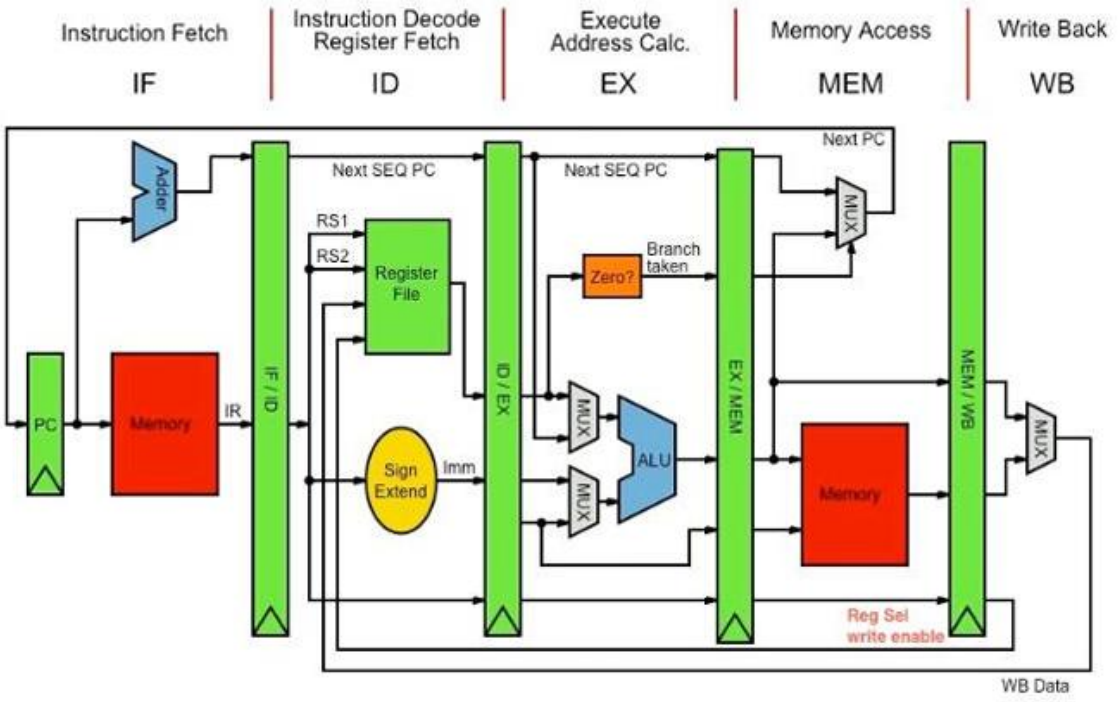
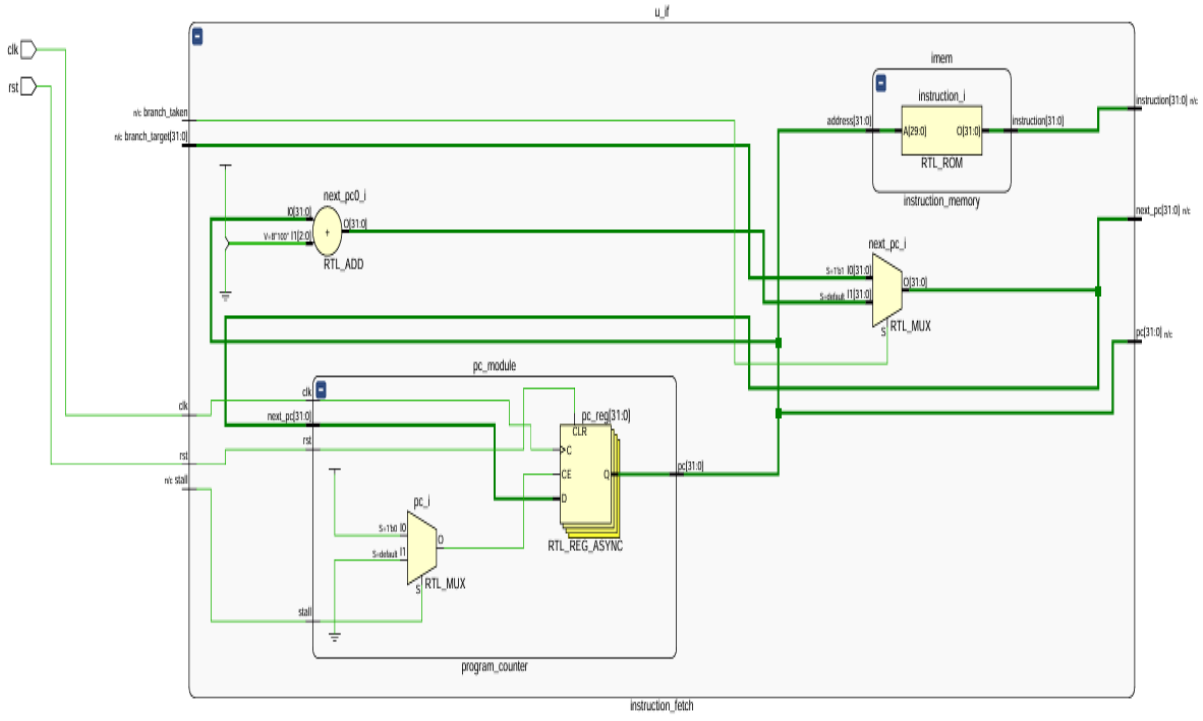


Figure : 5 Stage pipeline

DESIGN METHODOLOGY

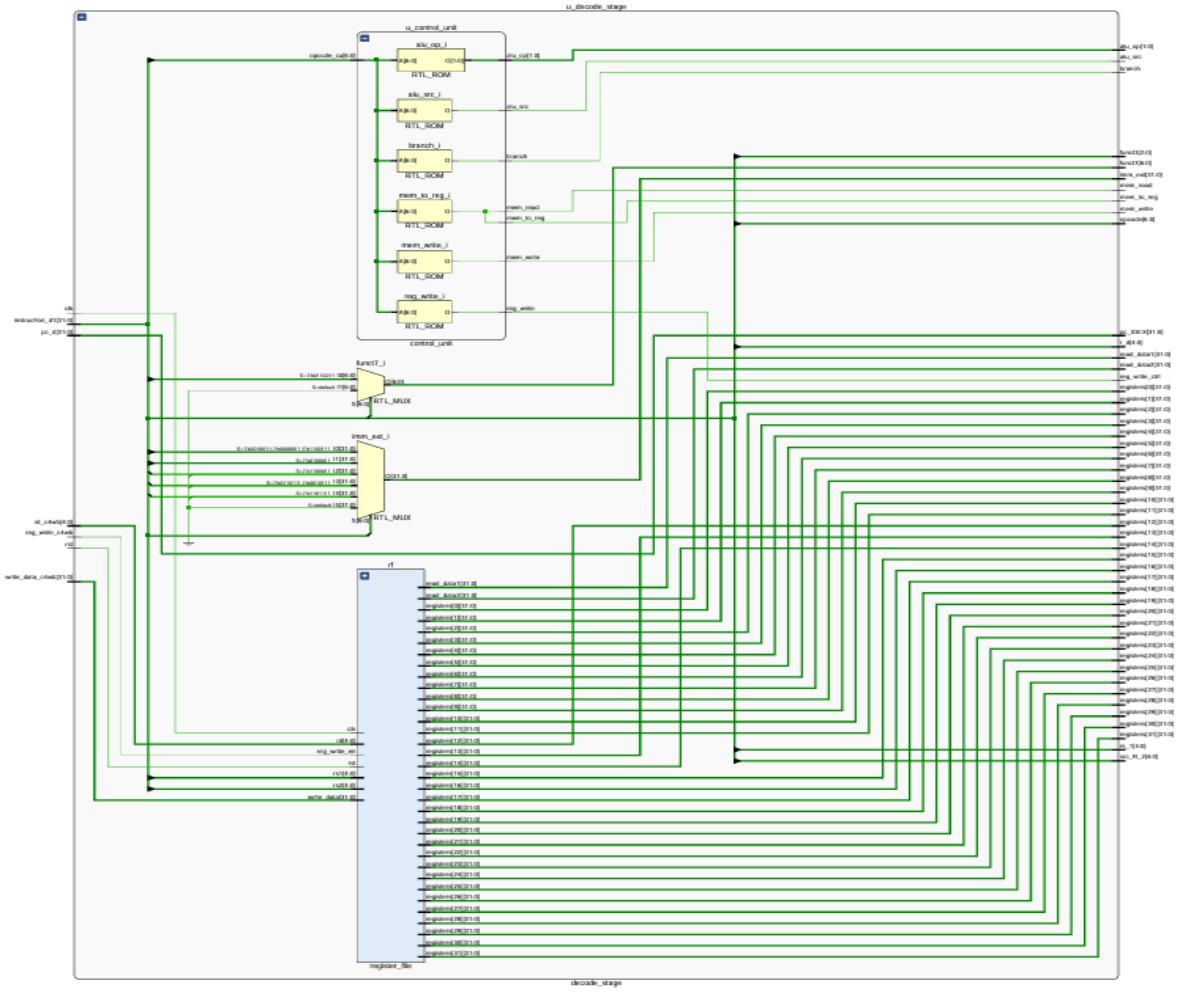
Instruction Fetch

The **Instruction Fetch (IF)** stage is a fundamental component of a pipelined CPU architecture responsible for retrieving the next instruction to be executed. This stage includes several interconnected modules that work together to determine the correct program counter (PC) value and to fetch the corresponding instruction from memory. The main components include the **program counter module (pc_module)**, which generates the current PC based on inputs such as reset (rst), clock (clk), and control signals like stall. It uses a **register (pc_reg[31:0])** to hold the current PC value and a **multiplexer (pc_i)** to select between default and updated PC values. The **instruction memory (imem)** fetches the instruction located at the current PC, facilitated by a ROM block labeled instruction_i. The address and output data paths are managed through buses like instruction[31:0] and address[31:0]. Another key module is the **next PC generator (next_pc_i)**, which calculates the next PC value using a multiplexer and an adder (next_pc0_i) depending on whether a branch is taken. The control signal branch_taken determines whether the next instruction should be sequential or redirected to a branch target. This design ensures accurate instruction fetching, supporting efficient pipeline execution by dynamically selecting the correct instruction address and seamlessly handling branching logic.



Decode

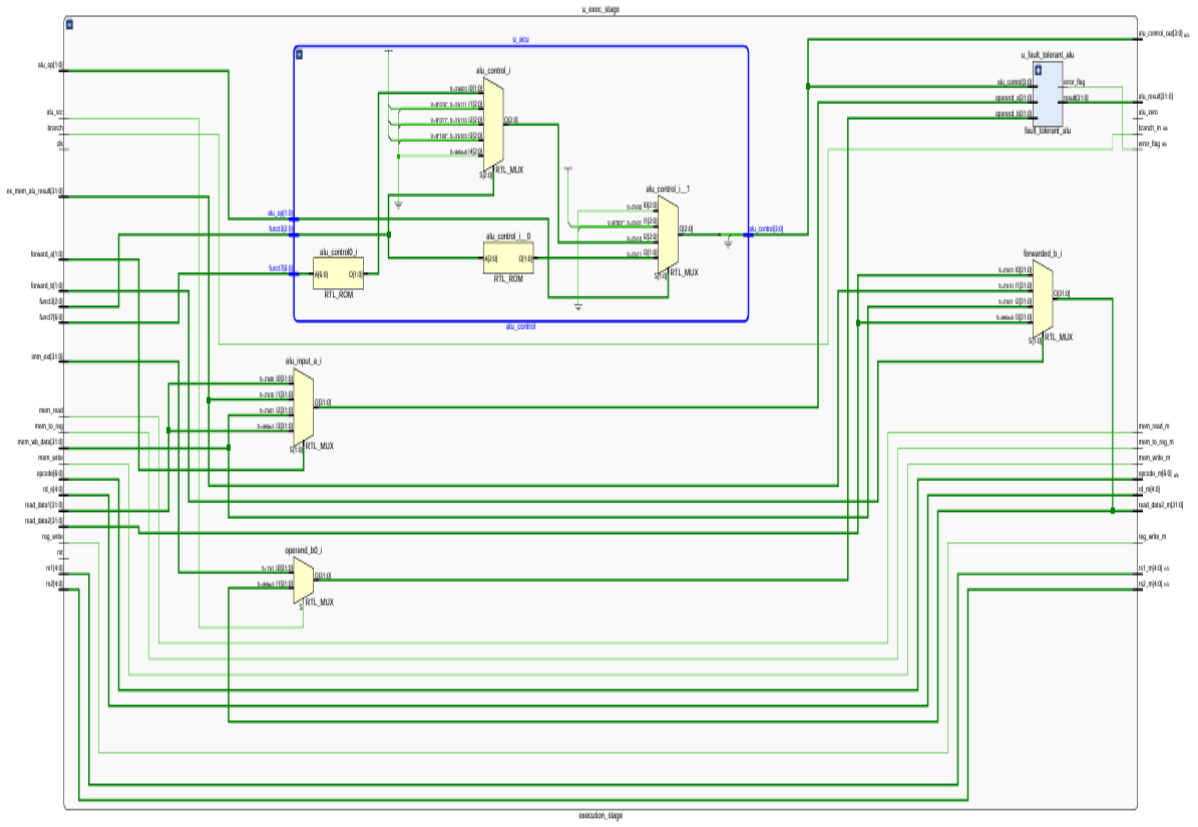
The decode stage represents a critical part of a CPU's data path, responsible for interpreting and processing the fetched instruction. The decode stage includes multiple components such as the control unit, register file, and several multiplexers (MUX) and read-only memories (ROMs) that help generate control signals based on the opcode and function codes of the instruction. It takes the instruction and extracts key parts like the opcode, source and destination register addresses (rs1, rs2, and rd), immediate values, and function codes (funct3, funct7). These values are used to generate control signals such as alu_src, mem_read, mem_write, branch, and reg_write, which guide how the instruction is executed in subsequent stages. The register file within this stage reads the source registers and writes to the destination register based on these control signals. Additionally, components like RTL_MUX and RTL_ROM help in selecting the correct paths and values depending on the instruction type (R-type, I-type, etc.). This setup ensures that the correct data and signals are passed forward for execution, memory access, and write-back stages in the pipeline.



Execution Stage

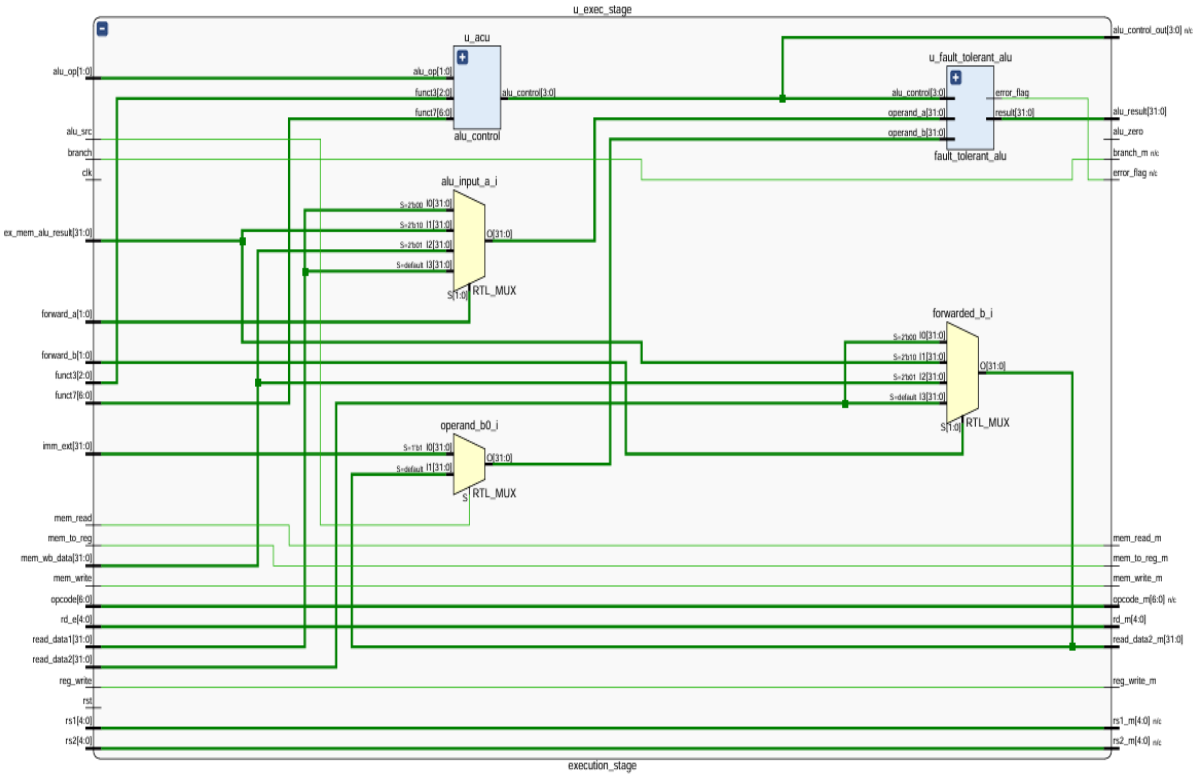
Arithmetic Control Unit (ACU)

The below schematic shows the design of a CPU's execution stage, mainly focusing on the Arithmetic Control Unit (ACU) and how it fits into the overall data path. This part of the CPU is responsible for carrying out arithmetic and logic operations when instructions are executed. The ACU takes instruction fields like opcode, funct3, and funct7 and uses them to generate control signals that tell the Arithmetic Logic Unit (ALU) what operation to perform, such as addition or subtraction. Inside the ACU, there are smaller modules like multiplexers and ROMs that help select and map the right control signals. The ALU gets its input values through different multiplexers, which can choose between forwarded data, immediate values, or values from registers. To make the system more reliable, the design also includes a fault-tolerant ALU that checks for errors and raises a flag if something goes wrong during a calculation. There are also control signals that manage how data moves between the CPU and memory, such as reading from memory, writing to memory, or writing back results to registers. Overall, the design is built using modular components at the Register Transfer Level (RTL), making it organized, reliable, and suitable for running standard CPU instructions efficiently.



The schematic represents the **execution stage schematic of a CPU**, highlighting the core components responsible for performing arithmetic and logic operations as part of a pipelined architecture.

The schematic in execu.pdf provides an overview of the execution stage in a pipelined CPU, with a focus on the Arithmetic Logic Unit (ALU), its control unit (ACU), and supporting data paths. The execution stage (u_exec_stage) integrates components that receive decoded instructions and perform the required calculations. It includes multiple multiplexers such as alu_input_a_i, operand_b0_i, and forwarded_b_i, which are used to select the appropriate operands from various sources like registers, immediate values, or forwarded data. The control signals for the ALU operations are generated by the Arithmetic Control Unit (u_acu), which takes instruction fields like opcode, funct3, and funct7 to determine what operation the ALU should perform. The actual arithmetic or logic processing is handled by a fault-tolerant ALU module (u_fault_tolerant_alu), which is designed to enhance reliability by detecting errors and raising an error_flag if a fault occurs during execution. This stage also includes control signals such as alu_src, branch, mem_read, mem_write, mem_to_reg, and reg_write, which help manage data movement and control flow in the processor. The design is modular and implemented at the RTL level, showing how instructions are executed safely and efficiently as part of a pipelined processor system.

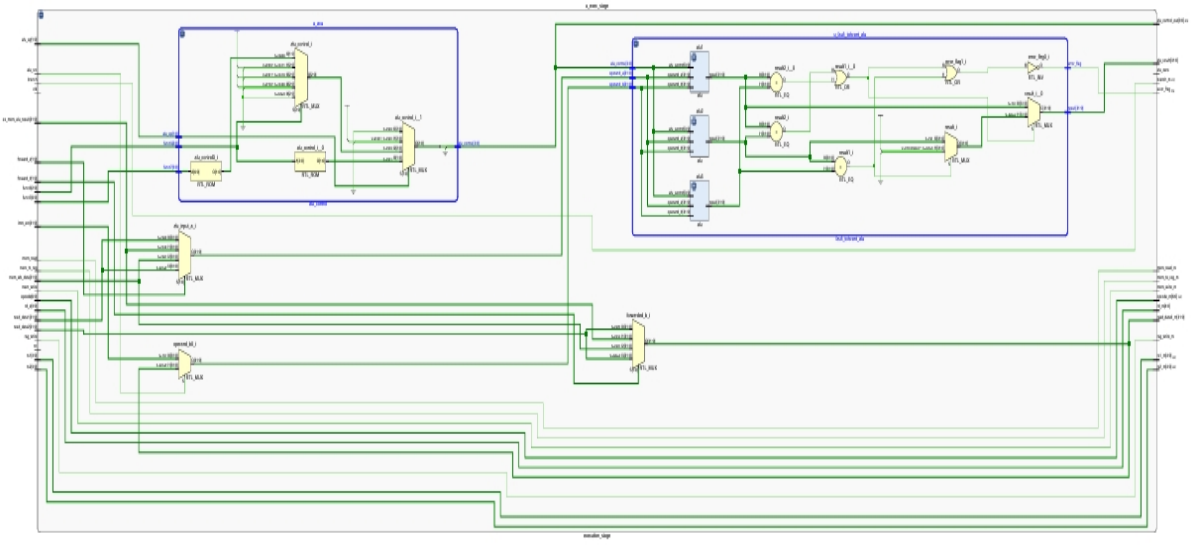


ACU with Fault Tolerant ALU subsystem:

The detailed schematic of the **Execution Stage** of a pipelined CPU design, emphasizing both the **Arithmetic Control Unit (ACU)** and a **Fault-Tolerant ALU subsystem**.

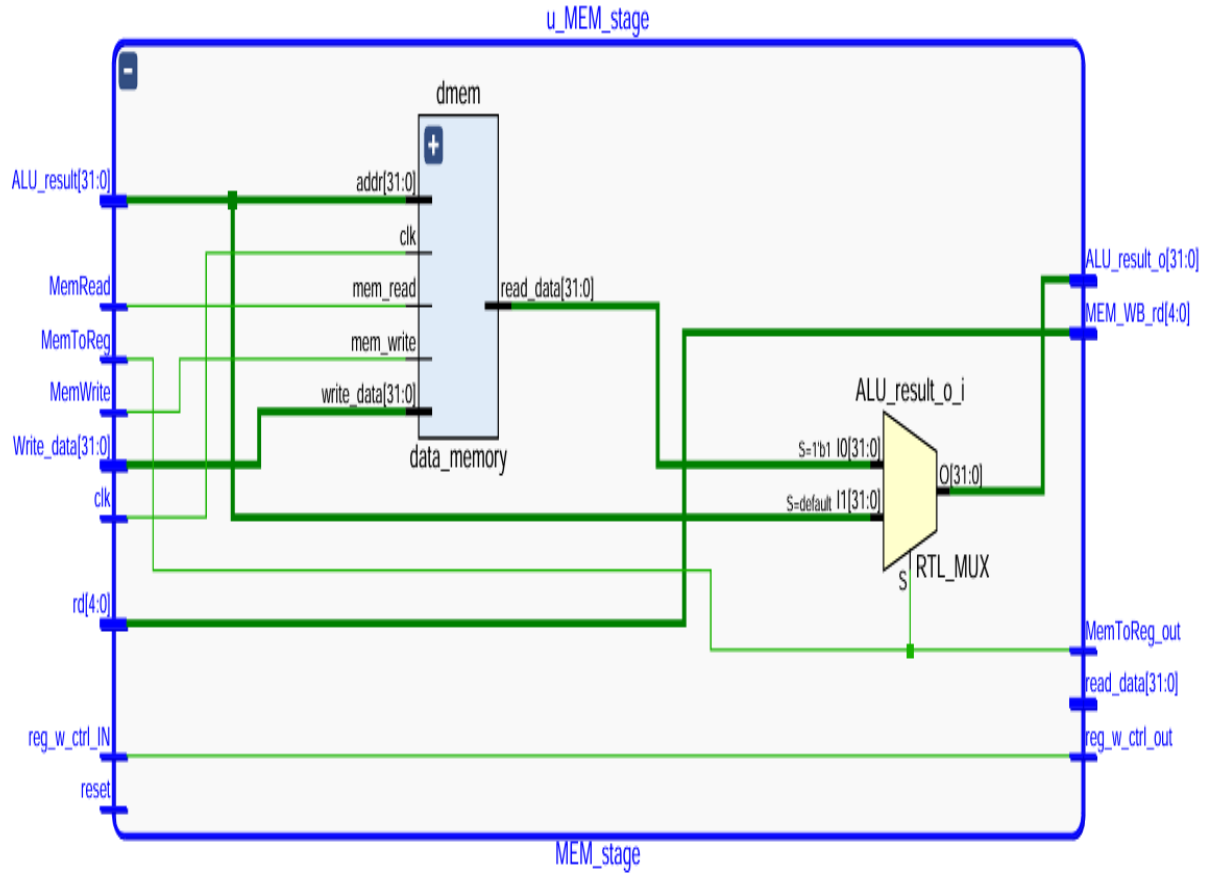
The schematic describes the internal structure of the CPU's execution stage, focusing on robust arithmetic processing and control logic. The execution stage (u_exec_stage) is responsible for executing instructions by performing arithmetic and logical operations. It includes several multiplexers (alu_input_a_i, forwarded_b_i, operand_b0_i) to manage data routing and operand selection from various sources like register values, forwarded data, or immediate values. The **Arithmetic Control Unit (ACU)** (u_acu) plays a central role in determining which operation the ALU should execute, using control signals derived from instruction fields like opcode, funct3, and funct7. The ACU generates ALU control outputs through a mix of multiplexers (alu_control_i, alu_control_i__1) and ROM blocks (alu_control0_i, alu_control_i__0), providing flexible control decoding.

A significant part of the design is the **Fault-Tolerant ALU** (u_fault_tolerant_alu), which ensures reliability by running the same operation in parallel on three ALUs (alu1, alu2, alu3). Their results are compared using equality checkers (result1_i, result2_i) and logic units such as OR (result1_i__0, error_flag1_i) and inversion logic (error_flag0_i) to generate an error_flag if a mismatch is detected. If errors occur, a result multiplexer (result_i, result_i__0) can output a default or safe value (e.g., DEADBEEF), maintaining stable system behavior. These fault detection mechanisms are important for use in critical systems where high reliability is required.

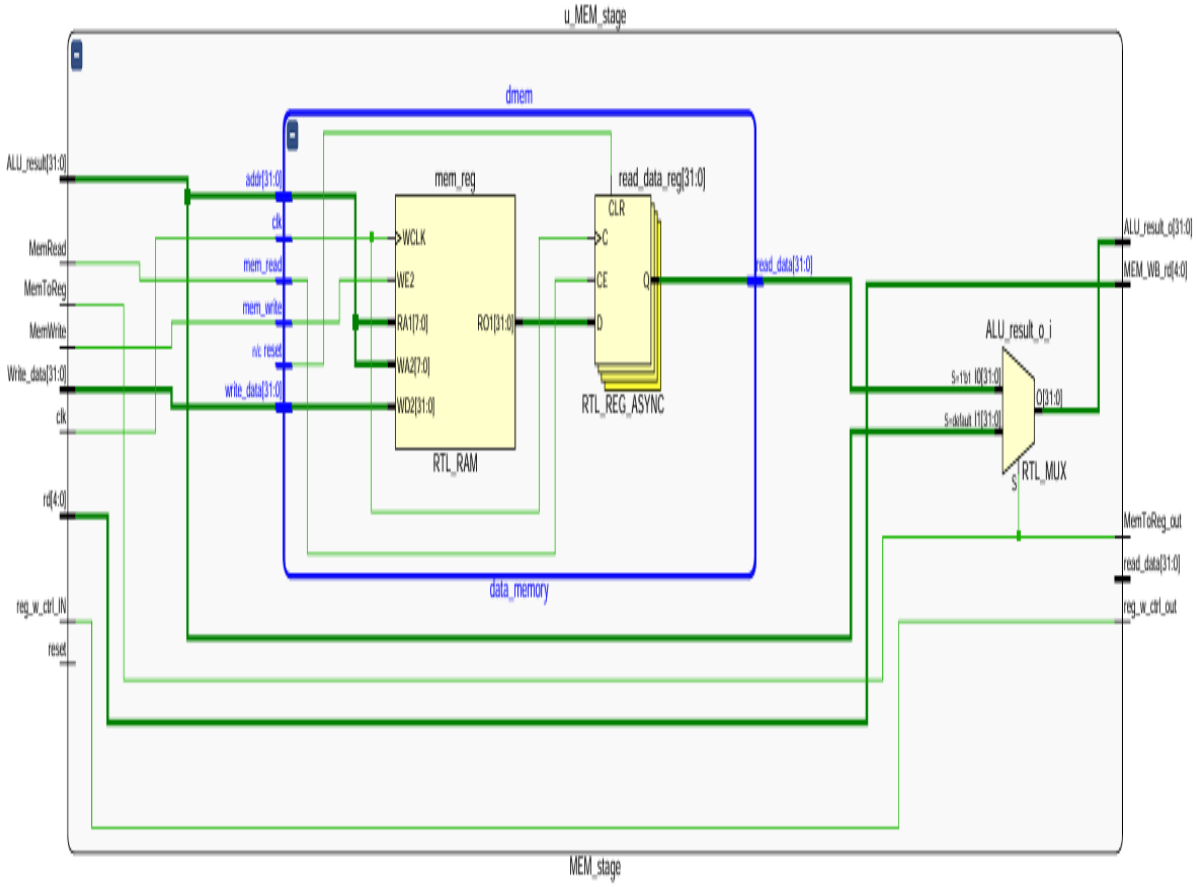


Memory Access

The **Memory Access (MEM) stage** is a critical phase in a pipelined CPU where interactions with data memory occur, based on the results computed by the Arithmetic Logic Unit (ALU) in previous stages. In this stage, the control signals MemRead, MemWrite, and MemToReg determine whether a memory read or write operation is performed and whether the data to be written back to the register file comes from memory or the ALU result. The **ALU result (ALU_result[31:0])** is used as the address for memory access, and based on the MemRead or MemWrite signal, the **data memory module (dmem)** either provides data to the Read_data bus or stores the input from the Write_data[31:0] bus. A **multiplexer (ALU_result_o_i)** decides whether the data forwarded to the next stage comes from the memory read or directly from the ALU output, depending on MemToReg. This output is routed to the write-back stage for updating the destination register (rd[4:0]). The entire operation is synchronized with the clk signal and controlled through input and output control signals like reg_w_ctrl_IN and reg_w_ctrl_out. This design enables precise coordination between arithmetic processing and memory interaction, which is essential for executing load/store instructions and maintaining data flow integrity in the pipeline.

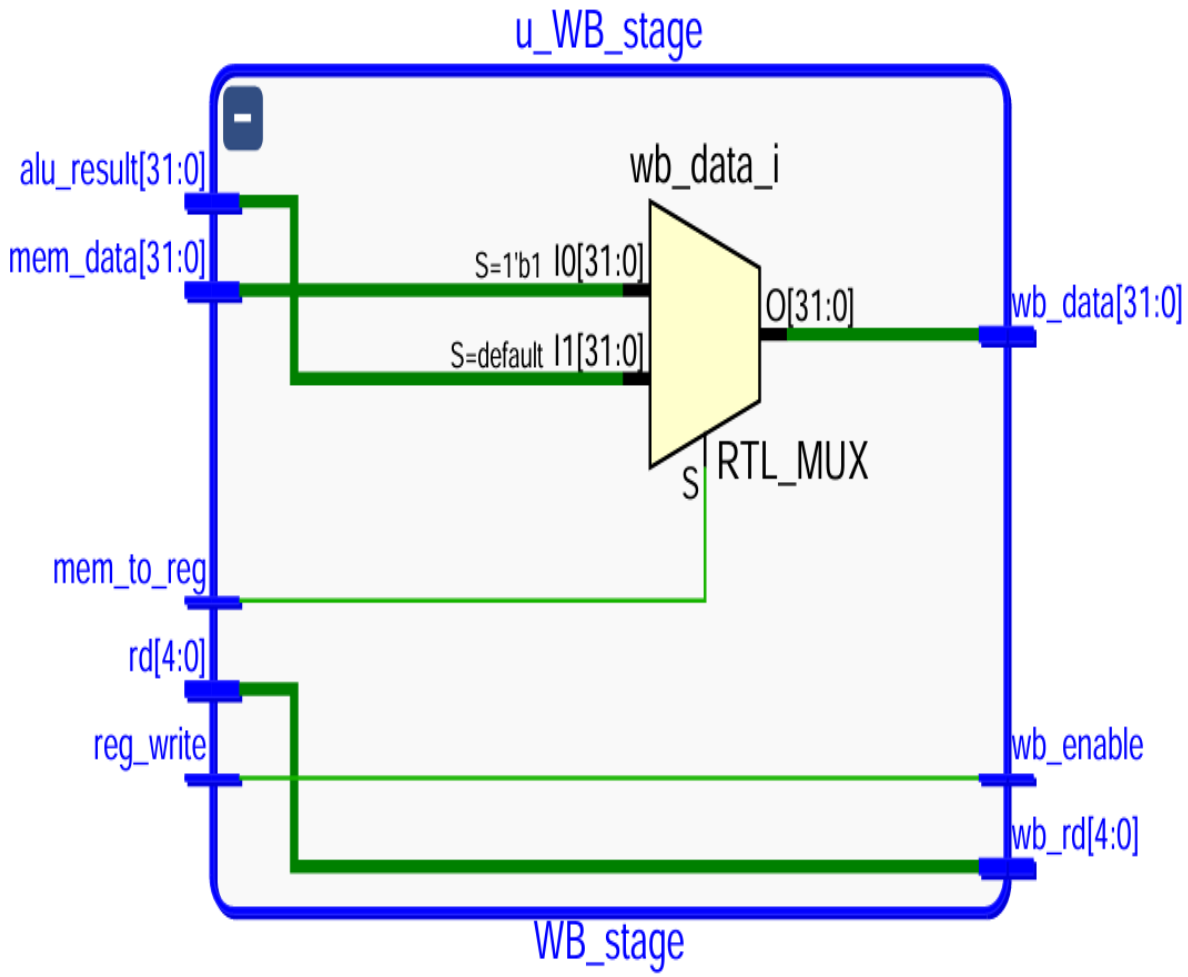


The **Memory Access (MEM) stage** of a pipelined processor handles data transfer between the processor and memory during instruction execution. It uses several key control signals, including MemRead, MemWrite, and MemToReg, to determine the type of memory operation required. When a read operation is needed (MemRead is high), the memory module fetches data from the address specified by the ALU result and stores it in a **read data register** (read_data_reg[31:0]) for stability and synchronization. When writing to memory (MemWrite is high), data is sent to memory using the write_data[31:0] bus. The memory unit (dmem) includes a register array (mem_reg) that represents the actual memory storage, and it is controlled via clock and reset signals. A multiplexer (ALU_result_o_i) determines whether the value forwarded to the next stage is from memory or directly from the ALU result, depending on the MemToReg signal. Additionally, control signals like reg_w_ctrl_IN and reg_w_ctrl_out coordinate with the Write Back (WB) stage to ensure the correct data is written to the appropriate register (rd[4:0]). This entire process is driven by the system clock (clk) and synchronized with pipeline control logic to maintain correct instruction execution flow. This stage is vital for executing load and store instructions and ensures proper data exchange with memory during program execution.



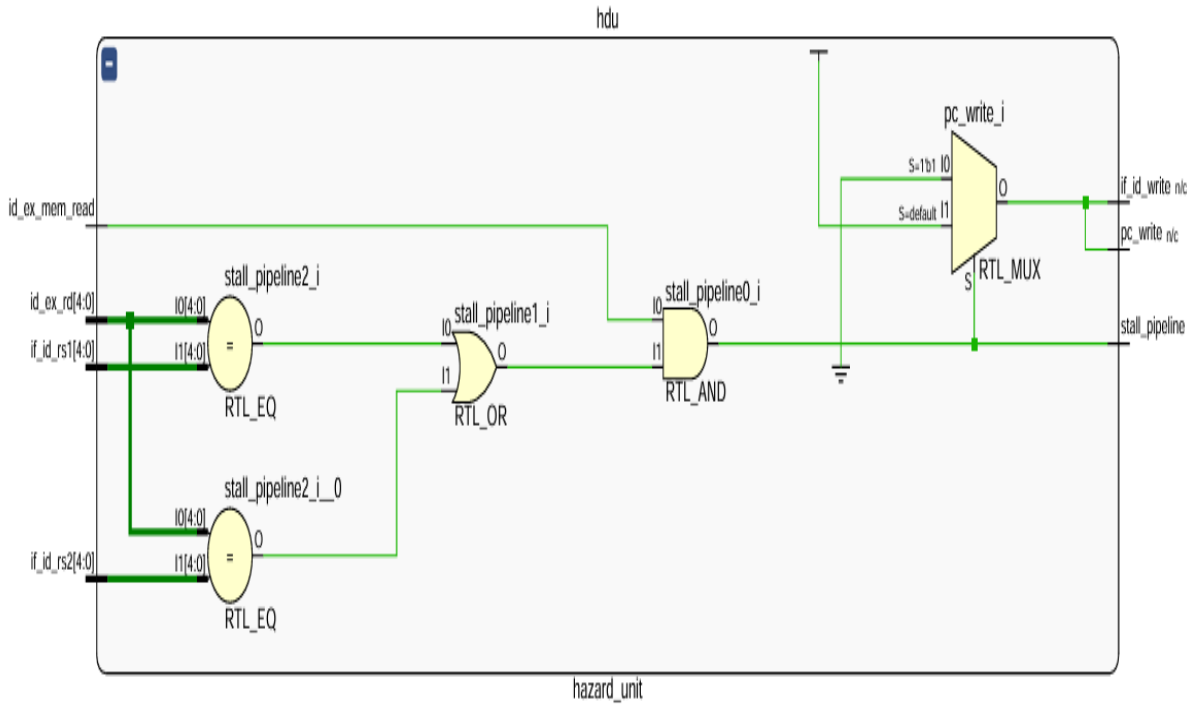
Write Back (WB) Stage

The **Write Back (WB) stage** is the final step in the CPU pipeline, where the results of instruction execution are written back into the register file. This stage determines whether the data to be written comes from the memory or directly from the ALU result, based on the mem_to_reg control signal. A multiplexer (wb_data_i) is used to select the correct data source—either the output of the memory (mem_data[31:0]) or the result from the ALU (alu_result[31:0]). The selected data is then sent to the destination register specified by the rd[4:0] signal. The control signal reg_write enables the writing process, ensuring that the data is stored only when necessary. The final write-back data is output as wb_data[31:0] along with the corresponding register address wb_rd[4:0]. This stage ensures that the results of operations like arithmetic computations or memory loads are saved correctly, allowing subsequent instructions to access the updated register values. As a result, the WB stage plays a crucial role in maintaining data consistency and completing the instruction lifecycle in a pipelined processor.



Hazard Detection Unit (HDU)

The Hazard Detection Unit (HDU) is an essential part of a pipelined processor that helps maintain correct instruction execution by detecting data hazards, specifically when an instruction depends on the result of a previous one that hasn't completed yet. In this project, the HDU checks for situations where the instruction currently in the decode stage (IF/ID) uses registers (rs1 or rs2) that are being written by the instruction in the execute stage (ID/EX) which is performing a memory read (mem_read). If such a dependency is found, and the destination register (rd) matches either of the source registers, the HDU triggers a stall. This stall is handled by preventing updates to the program counter (pc_write_i) and the instruction fetch/decode register (if_id_write), effectively pausing the pipeline for one cycle to give the previous instruction time to complete. The HDU uses simple logic gates like multiplexers, AND, OR, and equality checkers to implement this functionality. Overall, this unit helps savoid incorrect data being used in calculations by ensuring that instructions are executed in a safe order without overwriting or reading incomplete results.

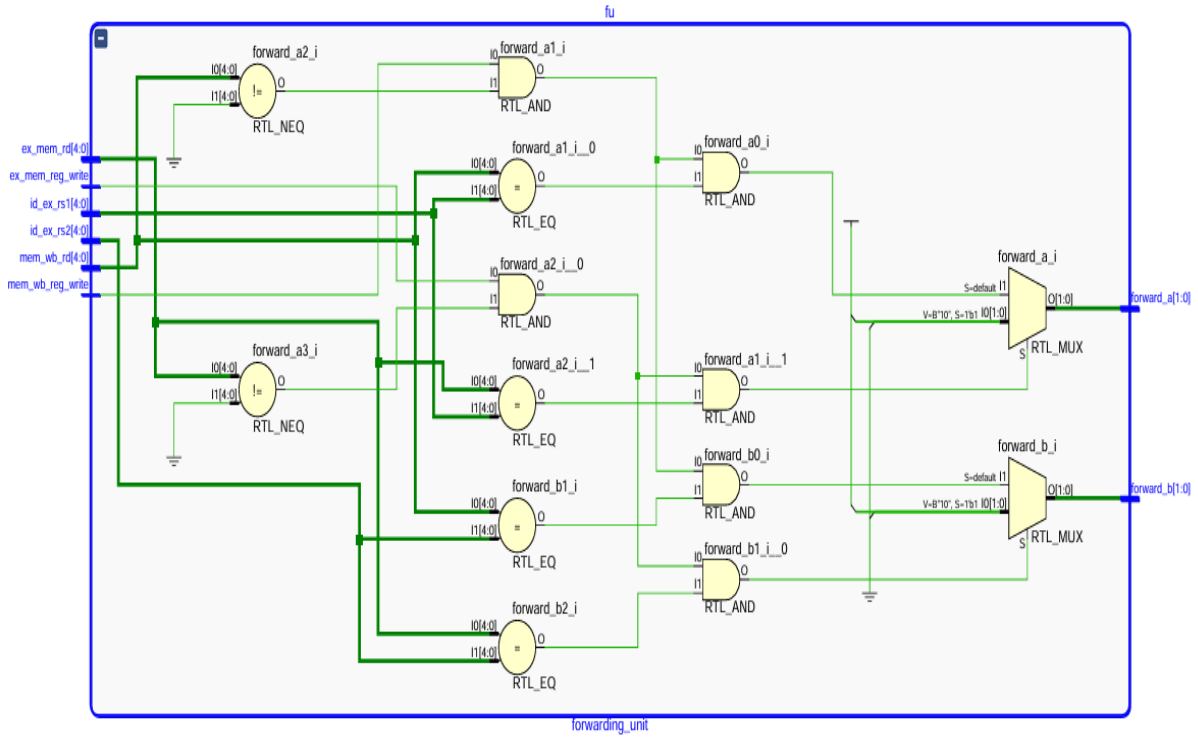


This typically happens when a load instruction in the execute stage (ID/EX) is followed by an instruction in the decode stage (IF/ID) that needs the same register. The HDU monitors key signals such as id_ex_mem_read, id_ex_rd, if_id_rs1, and if_id_rs2 to detect such cases. If a match is found, meaning the destination register of the load instruction is the same as one of the source registers of the following instruction, the HDU activates control signals like stall_pipeline, pc_write_i, and if_id_write to pause the pipeline for one cycle. This prevents the incorrect use of incomplete data. The logic is implemented using standard digital components like multiplexers, AND, OR, and equality checkers. The register file reads source operands based on rs1 and rs2 and provides outputs like read_data1 and read_data2, while the control unit generates control signals like alu_op, mem_read, and reg_write based on the opcode. Together, these modules work in sync to decode instructions, detect hazards, and maintain smooth and correct pipeline operation.

Forwarding Unit

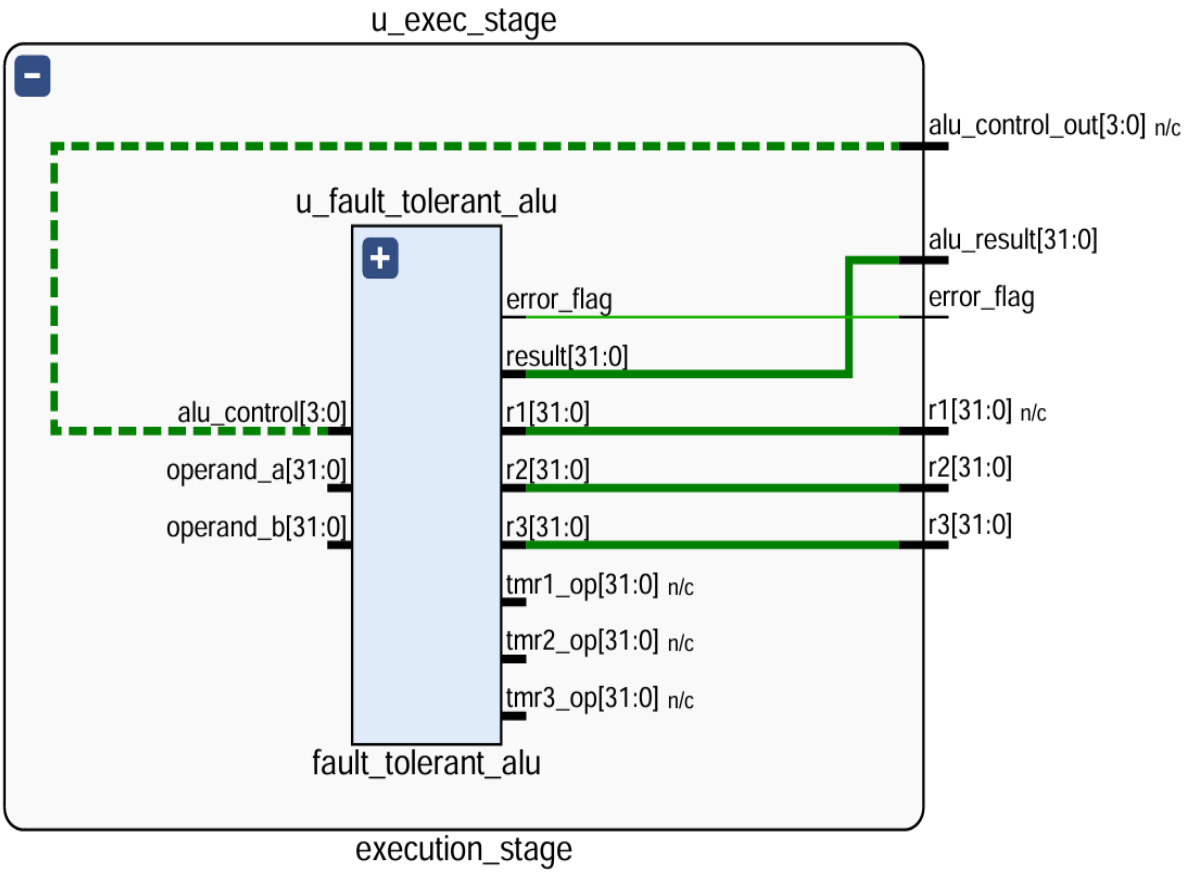
The schematic of the **Forwarding Unit** in a pipelined CPU architecture. This unit is crucial for addressing data hazards that occur when instructions depend on the results of previous instructions that have not yet completed their execution.

The design of the **Forwarding Unit** within a pipelined CPU, responsible for resolving data hazards by enabling the bypassing of data between pipeline stages. The unit compares register destination addresses from later pipeline stages (EX/MEM.rd, MEM/WB.rd) with the source registers of the current instruction in the ID/EX stage (id_ex_rs1, id_ex_rs2). If a match is detected and the corresponding register write signals (ex_mem_reg_write, mem_wb_reg_write) are enabled, the forwarding unit outputs control signals (forward_a, forward_b) to select the correct data path via multiplexers. This allows the CPU to use the most recent value of a register directly, without waiting for it to be written back to the register file, thereby preventing stalls. Internally, the unit employs a combination of logic gates such as AND, EQ (equality), and NEQ (not equal) to perform comparisons and control signal generation. Multiplexers (forward_a_i, forward_b_i) select the appropriate forwarding path based on these comparisons. This design helps maintain high instruction throughput in a pipelined processor by ensuring data dependencies are resolved efficiently and without unnecessary delays.

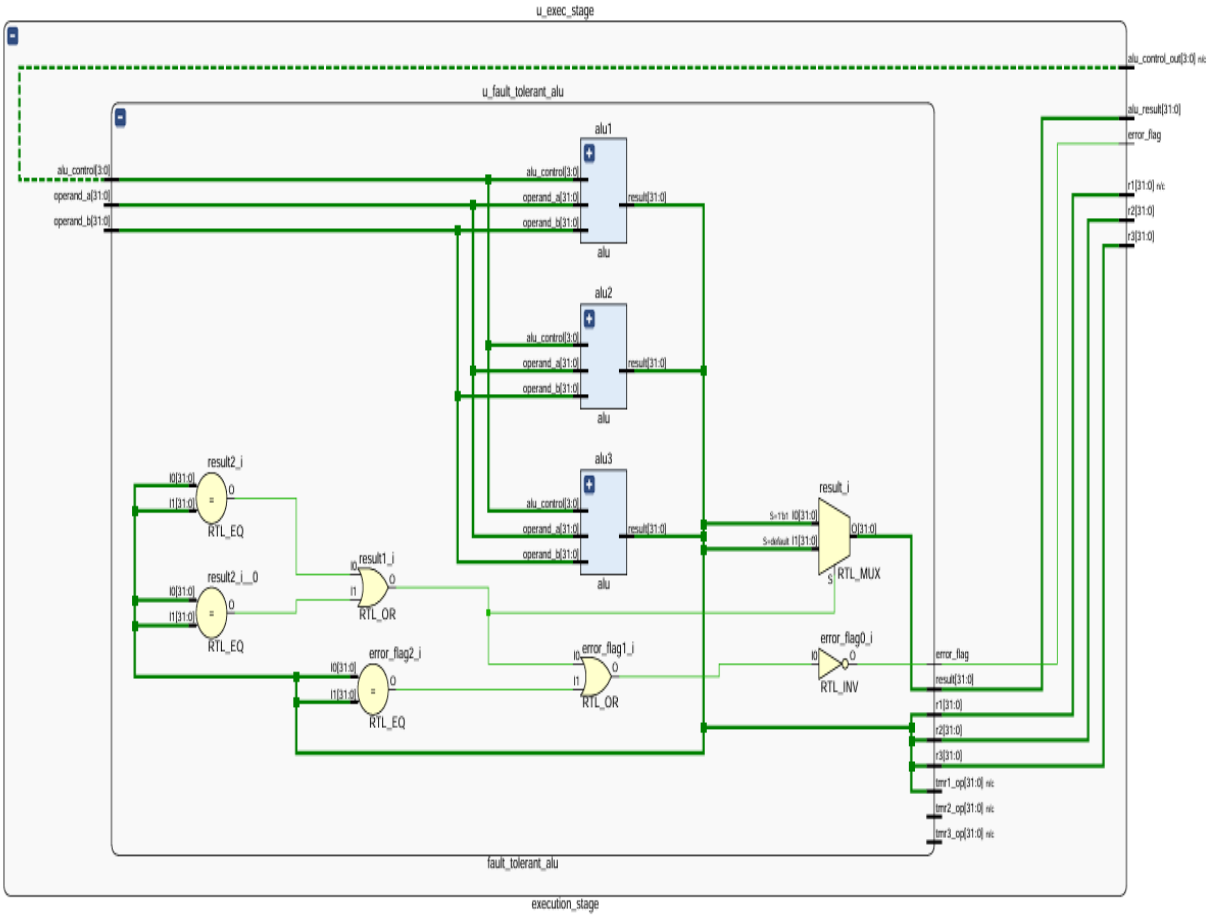


Fault Tolerant ALU

The execution stage of the CPU design includes a fault-tolerant ALU (Arithmetic Logic Unit), which is a crucial component responsible for performing arithmetic and logical operations. In this design, the ALU is enhanced with fault-tolerant mechanisms to ensure reliability and accuracy even in the presence of faults or errors. The system uses three redundant operation paths—tmr1_op, tmr2_op, and tmr3_op—which likely implement a technique such as Triple Modular Redundancy (TMR). Each path computes the same operation independently, and the results are compared to detect discrepancies. If one result differs from the others, the majority vote logic selects the correct output, enhancing error resilience. The final output is provided as alu_result[31:0], and any detected inconsistencies raise an error_flag signal. Inputs to the ALU include two 32-bit operands (operand_a and operand_b) and a 4-bit control signal (alu_control) that specifies the operation to be performed. This setup ensures that the CPU continues to operate correctly and reliably, even when a fault occurs in one of the ALU units.



A Fault-Tolerant Arithmetic Logic Unit (ALU) has been implemented as part of the CPU's execution stage to ensure reliable computation even in the presence of hardware faults. The ALU is designed using a technique called Triple Modular Redundancy (TMR), where three separate ALU units—alu1, alu2, and alu3—perform the same operation in parallel using the same inputs (operand_a, operand_b, and alu_control). Their outputs are then compared using logic components such as equality checkers (RTL_EQ) and majority voting through multiplexers. If one of the ALUs produces an incorrect result due to a fault, the system can still determine the correct result by choosing the value that matches at least two of the three outputs. An error_flag is raised when a discrepancy is detected among the ALU outputs, helping to identify faults in the system. This setup ensures that even if one ALU unit fails, the CPU continues to function correctly, making the design robust and suitable for critical or high-reliability applications.



Full Top Module CPU

A simple pipelined processor was implemented with key components including the Instruction Fetch (IF) block, Instruction Decode (ID) stage, Hazard Detection Unit (HDU), and Memory Access (MEM) stage—all working together to ensure correct and efficient execution of instructions. The IF block, driven by clock and reset signals, fetches instructions from memory to begin the pipeline process. These instructions are then sent to the ID stage, where the instruction is decoded using a control unit and a register file: the control unit generates control signals like alu_op, mem_read, mem_write, and reg_write, while the register file reads operands from registers based on fields rs1 and rs2. To ensure that instructions don't interfere with each other, especially when one depends on the result of a previous instruction, the HDU checks if the current instruction uses registers that are yet to be updated by a preceding load instruction. If such a data hazard is detected, the HDU activates control signals to temporarily pause (stall) the pipeline, preventing incorrect execution. After decoding, the instruction proceeds to the MEM stage, where memory operations are performed using control signals like MemRead and MemWrite. Depending on whether the result should come from memory or the ALU, a multiplexer selects the correct output to be written back. All these components coordinate seamlessly to keep the pipeline moving while handling hazards, decoding instructions, and managing memory access in a controlled and synchronized manner.

The CPU pipeline design illustrated in the schematic consists of several interconnected stages that work together to execute instructions efficiently and in a synchronized manner. The stages include **Instruction Fetch (IF)**, **Instruction Decode (ID)**, **Execution (EX)**, **Memory Access (MEM)**, and **Write Back (WB)**. In the IF stage, the next instruction is retrieved from the instruction memory using the current value of the program counter (PC). This value is updated based on control logic, including branch handling, and then passed to the decode stage through a pipeline register. In the ID stage, the instruction is decoded into its components such as opcode, source and destination registers, function codes (funct3, funct7), and immediate values. Control signals are generated, and the register file is accessed to retrieve operand values. The EX stage performs the actual computation using a fault-tolerant ALU, which compares results from multiple ALU instances to detect and correct errors. It also handles forwarding and hazard detection to manage data dependencies. In the MEM stage, based on control signals, data is either read from or written to the memory using the ALU result as the address. A multiplexer determines whether the value forwarded to WB comes from memory or

the ALU. Finally, in the WB stage, the selected data is written back into the destination register. The design includes robust control units, hazard detection, and forwarding units to ensure correct and optimized execution of instructions in a pipelined manner, minimizing stalls and improving overall CPU performance.

The overall CPU architecture implemented in this project follows a classic five-stage pipelined design, which includes the stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). This structure allows multiple instructions to be processed simultaneously at different stages, increasing the throughput and efficiency of the processor. Each stage is modularized with its own functional unit, such as the instruction_fetch unit for fetching instructions from memory, the decode_stage for interpreting instruction fields, and the execution_stage which includes a fault-tolerant ALU to perform arithmetic or logic operations reliably. The MEM_stage handles memory read and write operations, while the WB_stage writes the final result back to the register file. Additional components like the forwarding unit and hazard detection unit are integrated to manage data hazards and control hazards, preventing pipeline stalls and maintaining instruction flow. Registers such as ID_EX, EX_MEM, and MEM_WB are used as pipeline buffers between stages to hold intermediate data and control signals. The design also includes mechanisms for branch prediction and error detection to enhance performance and fault tolerance. This modular and pipelined CPU design achieves efficient instruction execution while maintaining data integrity and system reliability.

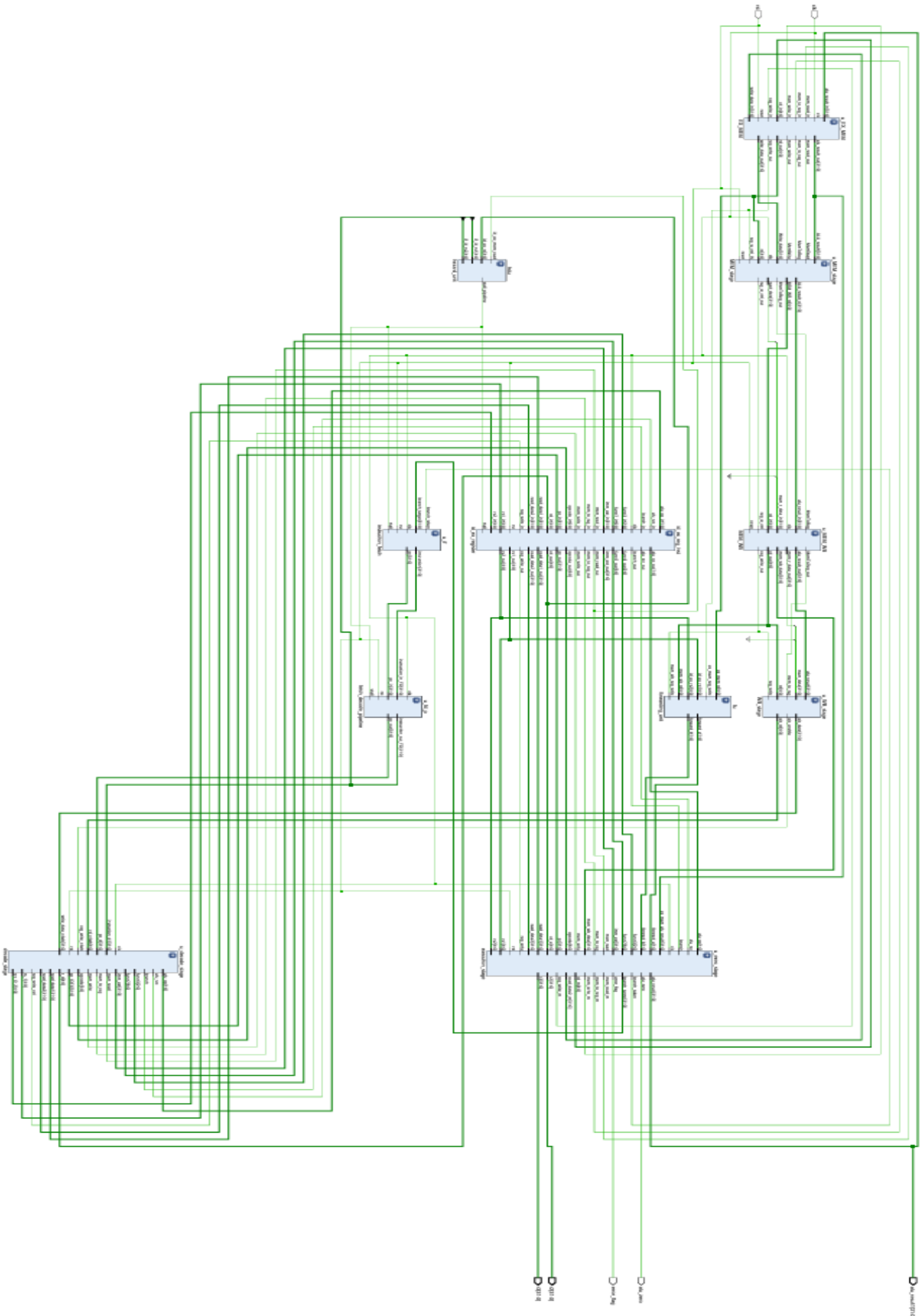


Fig. **Top-Level Schematic of 5-Stage Pipelined RISC-V Processor**

SOFTWARE USED

Vivado Design Suite

Vivado Design Suite, developed by Xilinx (now part of AMD), is an advanced software tool used for digital design and verification of systems implemented on Field Programmable Gate Arrays (FPGAs) and System on Chips (SoCs). In this project, Vivado played a central role in the design, simulation, synthesis, and implementation of a pipelined CPU architecture featuring a fault-tolerant Arithmetic Logic Unit (ALU). The software provided a comprehensive development environment to manage each stage of the hardware design process with precision and efficiency.

One of the key advantages of Vivado is its integration of multiple design tools into a single platform. This includes the Vivado Integrated Design Environment (IDE) for writing and managing HDL code, a simulator for behavioral and timing simulations, a synthesizer to convert HDL code into gate-level netlists, and an implementation engine that maps the design onto an FPGA. Vivado also supports IP integration, debugging, and timing analysis, all of which were crucial to building and validating the fault-tolerant processor in this project.

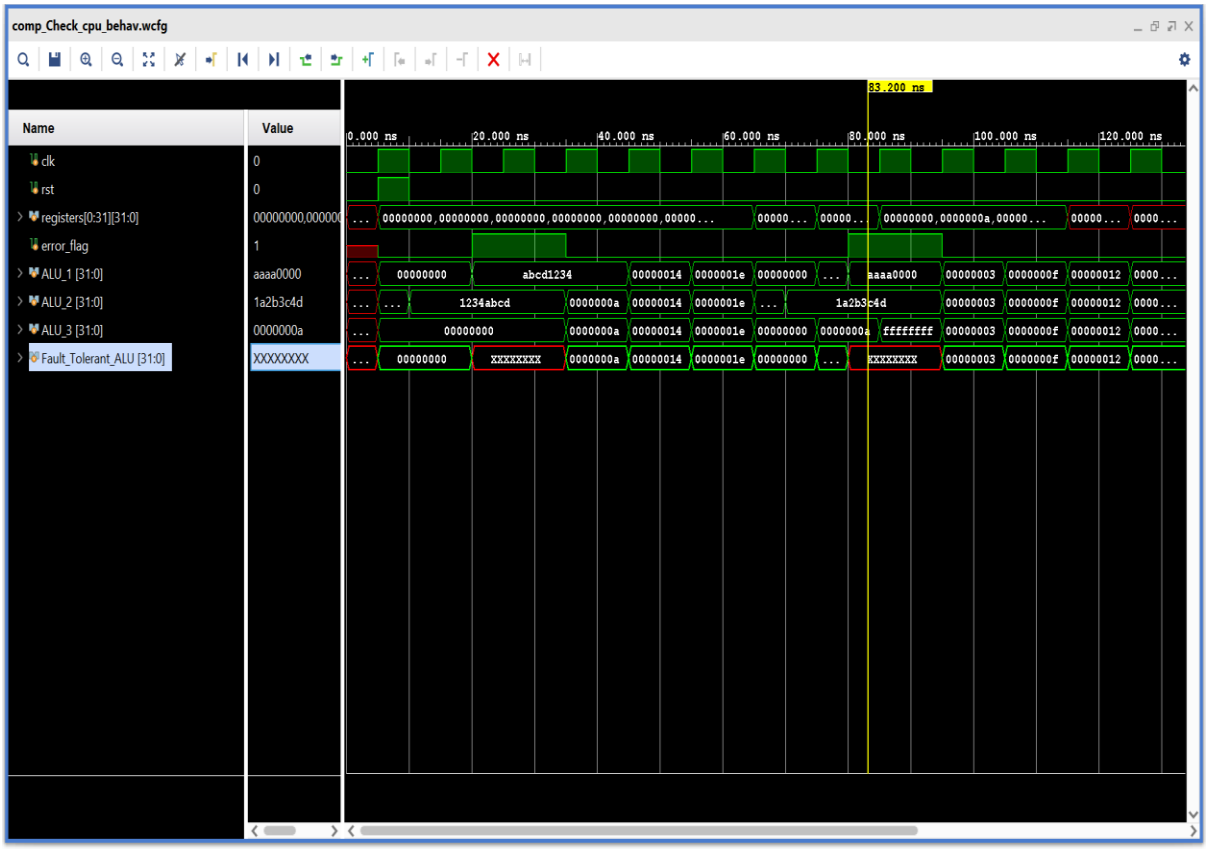
For this project, the Verilog HDL modules were written and tested using Vivado's built-in simulation tools. Waveform analysis, as shown in the simulation results, helped verify the correct operation of the fault-tolerant ALU by displaying the outputs of three ALU instances and ensuring that the majority-voting logic correctly handled any faults. Additionally, Vivado allowed for real-time debugging using signal probes and testbenches, which accelerated the development and validation cycle.

Furthermore, Vivado’s IP Integrator and support for hierarchical block design made it easier to manage the complexity of the CPU’s multi-stage pipeline structure. The software enabled seamless interconnection of modules such as the decode stage, execution stage, memory stage, and write-back stage. It also facilitated the incorporation of additional control units like the hazard detection and forwarding units.

In conclusion, Vivado Design Suite provided a powerful and user-friendly environment that significantly streamlined the development and testing of the CPU architecture in this project. Its robust toolchain, intuitive interface, and advanced simulation capabilities made it an essential component in ensuring the design’s correctness, performance, and fault tolerance.

SIMULATION AND RESULTS

The waveform simulation displayed in the figure demonstrates the behavior of the fault-tolerant ALU during runtime. The signals for three ALU instances—ALU_1, ALU_2, and ALU_3—are shown processing the same inputs to produce outputs. At specific points in time, the outputs of the individual ALUs differ, indicating a fault in one or more of them. For instance, around the 83.2 ns mark, ALU_3 outputs ffffffff, which clearly differs from the correct outputs 00000003 produced by both ALU_1 and ALU_2. As a result, the final output of the Fault_Tolerant_ALU is resolved using majority voting logic and correctly displays 00000003, while also triggering the error_flag signal to indicate the presence of a fault. This confirms that the fault-tolerant ALU is functioning as intended—it detects discrepancies, flags errors, and ensures reliable output by selecting the majority result. This kind of setup is crucial for systems where accuracy and reliability are essential, such as aerospace or medical devices.



CONCLUSION

This project successfully demonstrates the design and implementation of a RISC-V based five-stage pipelined CPU architecture, enhanced with a fault-tolerant Arithmetic Logic Unit (ALU). Through the integration of key components such as the instruction fetch, decode, execute, memory access, and write-back stages, the processor achieves efficient instruction execution by leveraging pipelining to increase throughput. A significant highlight of the design is the incorporation of Triple Modular Redundancy (TMR) in the ALU, which ensures fault tolerance by comparing the outputs of three identical ALU instances and using majority voting logic to deliver a reliable result, even in the presence of a hardware fault. This feature greatly enhances the processor's reliability, making it suitable for critical applications in aerospace, automotive, and embedded systems.

The use of Vivado Design Suite played a crucial role in the development cycle, providing powerful simulation, synthesis, and debugging capabilities. Functional verification through waveform analysis confirmed the accuracy of the ALU's fault detection and correction mechanisms. Additionally, supporting units such as the Hazard Detection Unit and Forwarding Unit were implemented to resolve data and control hazards, ensuring correct pipeline operation and minimizing performance loss due to instruction dependencies.

Overall, this project not only reinforces the practical value of RISC-V as a flexible and extensible ISA but also highlights the effectiveness of combining pipelining and fault tolerance in modern CPU design. The modular and scalable nature of the design opens up opportunities for future enhancements, including support for custom instruction sets, multi-core extensions, low-power optimizations, and domain-specific accelerators.

REFERENCES

1. https://ieeexplore.ieee.org/document/6391685
2. https://ieeexplore.ieee.org/document/10750638
3. Single cycle RISC-V micro architecture processor and its FPGA prototype by D. K. Dennis et al 2017 7th International Symposium on Embedded Computing and System Design (ISED), 2017, pp. 1-5, doi: 10.1109/ISED.2017.8303926.
4. Design of an 8-bit five stage pipelined RISC microprocessor for sensor platform application IEEE by R. J. L. Austria, A. L. Sambile, K. M. Villegas and J. N. T. Tabing, TENCON 2017 - 2017 Region 10 10.1109/TENCON.2017.8228209. Conference, 2017, pp. 2110-2115, doi:
5. Design and Implementation of 32-bit RISC-V Processor Using Verilog by Manjusha Rao P; Prabha Niranjan; Dileep Kumar M J .
6. Fault Tolerant ALU System by Ayon Majumdar; Sahil Nayyar; Jitendra Singh Sengar.
7. Design a 5-stage pipeline RISC-V CPU and optimise its ALU by Lifu Deng Glasgow College, University of Electronic Science and Technology of China, Chengdu, China.
8. S. P. Pitpurkar et al., 2015
9. Mohit N. Topiwala et al., 2014
10. Shawkat S. Khairullah, 2022
11. Shofiqul Islam et al., 2006
12. Animesh Kulshreshtha et al., 2021
13. Sarah M. Al-sudany et al., 2021