HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY OF INTERNATIONAL EDUCATION

MAJOR OF COMPUTER ENGINEERING TECHNOLOGY

††††

COURSE: SENIOR PROJECT 2

TOPIC: DESIGN AND IMPLEMENTATION OF AN RISC-V 32 BIT CORE 5 PIPELINE WITH C AND M EXTENDSION (RV32IMC)

STUDENTS:

Đỗ Mạnh Dũng - 21119301

Trần Long - 21119311

Đặng Trung Nghĩa - 21119313

LECTURER: Ph.D Phạm Văn Khoa

Ho Chi Minh, January'12, 2025

ABSTRACTION

The 32-bit RISC-V architecture, particularly the RV32IMC variant, represents a milestone in open-source processor design, offering scalability, modularity, and efficiency. This project focuses on researching, designing, and implementing a processor based on the RV32IMC architecture to leverage its reduced instruction set computing (RISC) principles while supporting integer arithmetic (I), multiplication and division (M), and compressed instructions (C). The RV32IMC architecture is well-suited for modern applications requiring a balance between computational power and energy efficiency, such as embedded systems, IoT devices, and low-power computing environments.

The project involves a detailed exploration of the RISC-V instruction set, pipeline design, and critical components such as the hazard unit, control unit, and ALU. Furthermore, the RV32IMC's compressed instruction set extension is analyzed and implemented to optimize memory usage and improve execution efficiency. By adopting the RISC-V open standard, the project promotes innovation, portability, and compatibility, contributing to the advancement of flexible processor designs.

The primary objective of this project is to design a processor that adheres to the RV32IMC specifications, validate its functionality through simulation, and evaluate its performance in practical scenarios. The project outcomes are expected to pave the way for integrating RISC-V-based solutions in real-world applications, demonstrating the potential of the RV32IMC as a versatile and high-performing architecture in modern computing.

ACKNOWLEDGMENTS

We would like to express my deepest gratitude to Ph.D Pham Van Khoa, our esteemed lecturer and guide, for his invaluable support, guidance, and encouragement throughout the course of this project. His profound knowledge, insightful advice, and unwavering patience have been instrumental in shaping the direction and execution of this research.

This project, focused on the design and implementation of the 32-bit RISC-V architecture (RV32IMC), would not have been possible without his mentorship and the opportunities he provided to explore and innovate. Ph.D Khoa's dedication to fostering a culture of academic excellence and his commitment to advancing the field of processor design have been a constant source of inspiration.

We are also grateful for the resources, insights, and constructive feedback he shared, which played a crucial role in overcoming challenges and achieving the objectives of this project. His guidance has not only contributed to the successful completion of this research but also enriched our understanding of the principles of reduced instruction set computing and modern processor architectures.

Thank you, Ph.D Pham Van Khoa, for your belief in our abilities and for guiding we on this journey of learning and discovery.

TABLE OF CONTENT

LIST OF FIGURE

LIST OF TABLE

1. Introduction

1.1. Introduction

In modern computing, processor architecture plays a critical role in shaping the performance, energy efficiency, and adaptability of systems across diverse applications. The RISC-V Instruction Set Architecture (ISA) has emerged as a revolutionary standard in this field, offering an open-source, modular, and extensible platform for processor design. Unlike proprietary ISAs, RISC-V enables developers and researchers to customize architectures to meet specific requirements without being constrained by licensing costs or vendor lock-ins. This flexibility and scalability make RISC-V a compelling choice for building processors tailored to high-performance, low-power, and cost-sensitive applications.

This project aims to design and implement a 5-stage pipelined processor based on the RV32IMC variant of the RISC-V ISA. The RV32IMC architecture is a 32-bit implementation that includes the base integer instruction set (RV32I), the "M" extension for integer multiplication and division, and the "C" extension for compressed instructions. The inclusion of the "M" extension facilitates efficient execution of computationally intensive arithmetic operations, while the "C" extension reduces code size and improves instruction cache utilization by introducing compressed 16-bit instructions. Together, these features enable the development of high-performance processors optimized for embedded systems and resource-constrained environments.

The 5-stage pipelined architecture adopted in this project ensures efficient instruction processing by dividing execution into five sequential stages: Fetch, Decode, Execute, Memory, and Writeback. This approach allows multiple instructions to be processed concurrently, significantly enhancing throughput. The pipeline design is complemented by advanced mechanisms, including hazard detection units, forwarding logic, and branch prediction, to mitigate the impact of data, control, and structural hazards. These features ensure that the processor delivers high performance while maintaining functional correctness.

The project's overarching objective is to build a functional RV32IMC processor capable of executing a wide range of instructions efficiently, with a particular focus on real-world applicability. The design process includes simulation and verification stages to validate compliance with the RISC-V specification, functionality under various workloads, and performance metrics. The modular nature of the RV32IMC processor also positions it as a versatile platform for educational and research purposes, enabling future extensions and optimizations.

By leveraging the power of the RV32IMC architecture, this project highlights the potential of RISC-V to address the needs of modern computing systems. The combination of open-source principles, modular design, and advanced features like compressed instructions makes the RV32IMC an ideal choice for building efficient, scalable, and adaptable processors that can cater to diverse applications ranging from embedded devices to high-performance systems.

1.2. Target of project

The primary target of this project is to design, implement, and verify a fully functional RISC-V processor based on the 5-stage pipelined architecture. This processor aims to achieve efficient instruction execution while addressing common challenges such as data hazards, control hazards, and structural hazards. By incorporating modular design principles and utilizing the extensibility of the RISC-V instruction set architecture (ISA), the project seeks to provide a processor that is not only functional but also scalable for future development. The processor will support standard RISC-V base instructions (RV32I) along with relevant extensions, enabling compatibility with a wide range of applications, from embedded systems to educational purposes. Additionally, this project aims to integrate practical I/O interfaces, ensuring the processor is ready for real-world implementation. With a focus on optimization and compliance with open standards, the project ultimately targets the creation of a reliable and cost-effective processor that exemplifies the flexibility and potential of the RISC-V architecture.

1.3. Structure

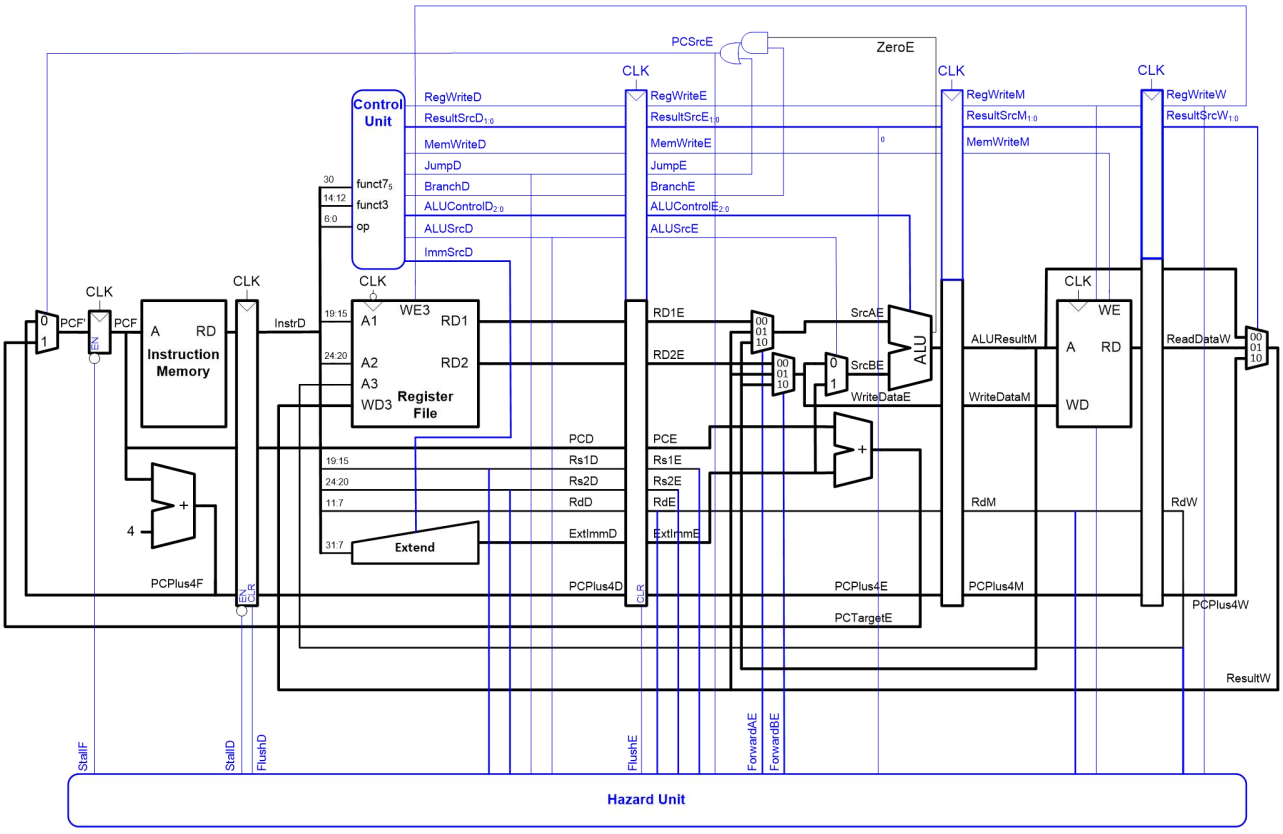

Figure 1: RV32I architecture

2. RICS-V architectures

2.1. Overview of RICS-V architecture

The chip serves as the backbone of the information industry, with the instruction set architecture (ISA) as its essential technology. For decades, proprietary ISAs like x86 and ARM have dominated the mainstream market, solidifying their positions due to well-established ecosystems and performance. However, both x86 and ARM carry high licensing fees and restrict broader industry access, creating barriers to widespread innovation. This gap in accessibility has led to the emergence of RISC-V, an open-source ISA that is rapidly transforming the chip landscape. Due to its cost-free licensing, flexibility, and scalability, RISC-V has become a powerful driver of global chip innovation. Unlike proprietary ISAs, RISC-V is BSD-licensed, royalty-free, and designed with extensibility in mind. Its modular structure supports 32-, 64-, and 128-bit integer bases, with optional extensions like floating-point operations. This modularity offers simplicity and flexibility: implementing RISC-V cores is generally easier, as minimal cores are about half the size of equivalent ARM cores, making RISC-V highly efficient and accessible. This advantage, combined with its adaptability, has led to strong support within the semiconductor industry. This open ISA with a growing community of developers makes RISC-V well-suited for diverse applications, from embedded systems to high-performance processors. RISC-V's standout feature is its extensibility: developers can selectively implement optional extensions from the base ISA and even create custom extensions tailored to specific needs. This flexibility allows developers to optimize RISC-V processors for specialized applications, boosting performance in targeted areas. By offering a royalty-free, open-source ISA, RISC-V unlocks new possibilities for processor innovation. Its modular structure and support for domain-specific customization are ideal for embedded processors, while its flexibility and scalability continue to drive its popularity and adoption across the chip industry. Through these unique advantages, RISC-V paves the way for a new era of accessible, high-performance processor design.

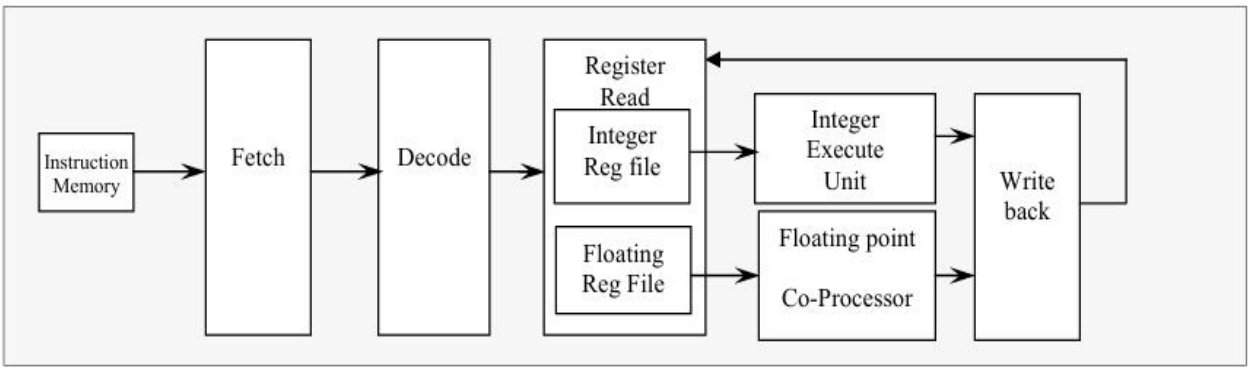

Figure 2: Top level for RISC-V Architecture

2.2. Features of RISC-V ISA

2.2.1. Key features of RISC-V ISA

a. Modularity and Extensibility:

- RISC-V allows for modularity, meaning extensions can be added as needed (M extension for multiplication/division, F/D extensions for floating-point operations).
- This flexibility enables the ALU to handle a broader range of operations without needing a complete redesign, ideal for applications requiring both basic and complex operations (IoT and AI).

b. *Open-Source Nature*:

- RISC-V is open-source, meaning it is royalty-free and allows for customization without licensing fees.
- This promotes widespread adoption and development, particularly valuable for educational, research, and startup environments where cost and flexibility are critical.

c. *Compressed Instructions (C Extension)*:

- The C extension in RISC-V reduces code size by using 16-bit instructions for frequently used operations, improving memory efficiency, which is especially beneficial in embedded systems.
- A smaller instruction size means faster fetching and lower memory requirements, which improves overall performance in resource-constrained environments.

d. *Future-Proof and Scalable*:

- With support for both 32-bit and higher architectures (64-bit, 128-bit), RISC-V can be scaled up, making it easier to transition from smaller ALU designs to more complex, high-performance processors.
- This scalability allows RISC-V to be used in various applications, from microcontrollers to high-performance computing (HPC).

e. *Industry and Community Support*:

- The open-source community and industry leaders, like SiFive and NVIDIA, actively contribute to the RISC-V ecosystem, which accelerates toolchain support and brings cutting-edge optimizations.
- Extensive support enables access to optimized compilers, simulators, and software libraries, making RISC-V more accessible than some proprietary ISAs.

2.2.2. Limitations of RISC-V ISA

a. *Complexity in Design for Specific Applications:*

- The modular nature of RISC-V can add complexity, as adding specific extensions (F/D for floating-point) increases the ALU's design complexity and may require specialized knowledge.
- A basic ISA may be simpler to implement for small, low-power applications that only require basic integer operations without the overhead of modular extensions.

b. *Toolchain and Software Ecosystem*:

- Although RISC-V's ecosystem is growing, it is still less mature compared to established ISAs like ARM and x86. Limited software compatibility and optimization might affect performance in legacy applications.

- For highly optimized or proprietary software, additional adaptation or toolchain customization may be necessary.

c. *Hardware Resource Usage*:

- Implementing certain extensions (like V for vector processing or F for floating-point operations) in RISC-V can consume significant hardware resources, such as logic gates and power.
- For applications that only need basic operations, a simplified ISA may be more efficient and cost-effective, particularly in low-power environments.

d. *Limited Specialized Support*:

- Advanced optimizations found in proprietary ISAs, like ARM's DSP and big.LITTLE architecture, offer finely-tuned power-performance balances that may be missing in RISC-V.
- Without such specialized optimizations, RISC-V may fall short in applications requiring tight control over power-performance trade-offs, such as in mobile processors.

e. *Learning Curve and Integration*:

- As an open ISA, RISC-V may present challenges for teams unfamiliar with modular architectures, especially when designing complex ALUs or adding extensions.
- Learning to utilize open-source RISC-V toolchains and simulators might require additional time compared to established proprietary ISAs with comprehensive documentation and support.

2.3. RICS-V Architecture classification

2.3.1. RV32I (Base Integer Instruction Set)

- Overview: RV32I is the basic instruction set for RISC-V, designed for integer operations in a 32-bit processor. It provides all essential arithmetic, logical, and control instructions needed for a fully functional processor.
- Instruction Set:
  - Arithmetic: ADD, SUB, ADDI
  - Logic: AND, OR, XOR,NOT
  - Shift: SLL, SRL, SRA
  - Control Flow: BEQ, BNE, BLT, BGE
  - Use Case: Used widely in embedded systems and microcontrollers, as it provides the essential operations for general-purpose processing

2.3.2. RV32IM (Integer with Multiplication and Division)

- Overview: RV32IM extends RV32I by adding multiplication and division instructions, enabling more complex arithmetic functions.
- Instruction Set:
  - Multiplication: MUL, MULH, MULHSU
  - Division: DIV, DIVU, REM, REMU
- Use Case: Ideal for applications requiring integer-based math, such as signal processing and control systems.

2.3.3. RV32F (Single-Precision Floating Point)

- **Overview:** The RV32F extension adds support for single-precision floating-point operations based on IEEE-754 standards.
- **Instruction Set:**

- Arithmetic: FADD.S, FSUB.S, FMUL.S, FDIV.S
- Comparison: FEQ.S, FLT.S, FLE.S
- Conversion: FCVT.S.W (integer to single-precision float) and FCVT.W.S (single-precision float to integer)

- **Use Case:** Commonly used in digital signal processing, audio processing, and real-time IoT applications that need moderate floating-point precision.

2.3.4. RV32E (Embedded Integer Instruction Set)

- **Overview:** RV32E is a streamlined version of RV32I with a reduced register set (16 registers instead of 32) designed for embedded systems with strict power and area constraints.
- **Instruction Set:**
  - Same as RV32I, but tailored for smaller, low-power processors.
- **Use Case:** Used in low-power microcontrollers and simple IoT devices where minimizing hardware complexity and power consumption is crucial.

2.3.5. RV32C (Compressed Instruction Set)

- **Overview:** The RV32C extension adds compressed instructions to reduce code size, making it suitable for memory-constrained environments. It provides shortened versions of common instructions.
- **Instruction Set:**
  - Compressed versions of instructions like C.ADD, C.LW, C.SW, C.J, C.BEQZ, which are compact versions of full instructions.
- **Use Case:** Widely used in embedded systems and microcontrollers to reduce memory footprint and improve code density, enabling more efficient use of limited storage.

2.4. Summary

The table below highlights the modular and scalable design of the RISC-V architecture by breaking down its instruction subsets. The base instruction sets, RV32I and RV64I, provide foundational support for 32-bit and 64-bit systems with essential instructions for integer arithmetic, memory access, and branching, making them suitable for a wide range of applications. Optional extensions, such as RV{32,64}M for integer multiplication and division, RV{32,64}A for atomic operations, RV{32,64}F/D for floating-point calculations, and RV32V for vector operations, offer flexibility to customize processors for specific needs, from embedded systems to high-performance computing. The inclusion of RV{32,64}C compressed instructions optimizes code density, further enhancing its suitability for memory-constrained environments. With a total of 268 instructions, including 59 pseudo instructions that simplify programming, the architecture strikes a balance between simplicity and functionality. The modular and extensible design ensures RISC-V's applicability across various domains, enabling developers to optimize performance and efficiency while maintaining scalability for applications ranging from IoT devices to data-intensive workloads [8].

Table 1: Specified Instructions of RISC-V

Instruction subset	Description	Number of instructions
RV32I	Base integer arithmetic, memory access, branches	37
RV32I	Access to control registers	6
RV32I	System instructions	9
RV64I	Base integer arithmetic, branches	12
RV{32.64}M	Integer multiplication and division	8+5
RV{32,64}A	Atomic instructions	11+11
RV{32,64}F	Floating-point instructions	26+4
RV{32,64}D	Floating-point instructions	26+4
RV{32,64}C	Compressed instructions	35+7
RV32V	Vector instructions	8
Pseudo	Pseudo instructions	59
Totally		268

Conclusion: Through the above lists and comparisons of popular RISC-V architectures. At the same time, our team's long-term intention is to develop a 32-bit ALU into a 32-bit processor that can be applied to specific purpose designs. So our team has decided to choose the RV32I architecture to conduct research and apply to 32-bit ALUs and, if feasible, expand to the "extension architecture" to better facilitate the study of more complex designs in the future.

3. RV32I Base ISA

3.1. Overview of RV32I

The RV32I is the base integer instruction set for the RISC-V architecture, designed to operate on 32-bit data widths. As the foundational instruction set, RV32I provides all essential instructions for arithmetic, logic, memory access, and control flow needed for general-purpose computing. This simplicity and modularity make RV32I ideal for small, efficient processors, especially in embedded and IoT applications [4].

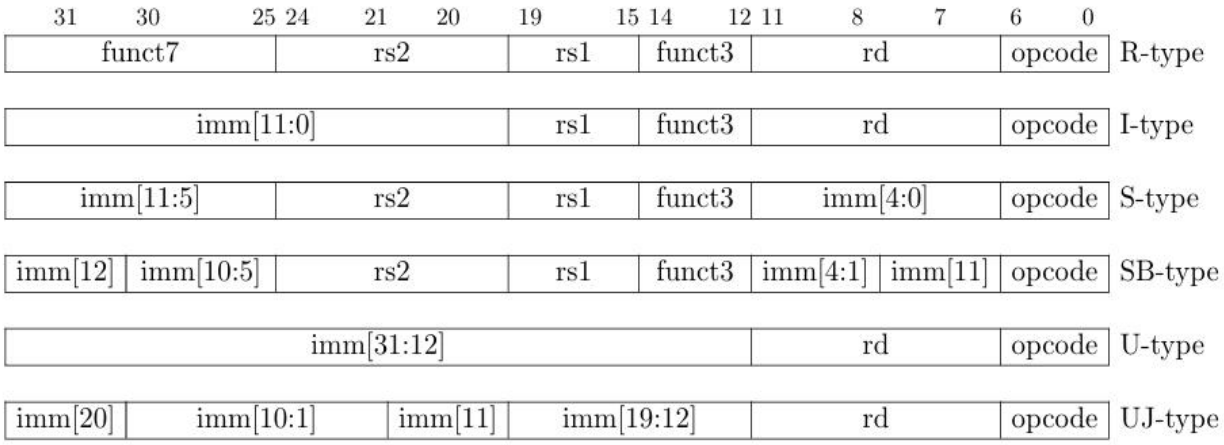

Figure 3: RV32I Instruction format

This figure illustrates the six instruction formats utilized in the RISC-V architecture: R-type, I-type, S-type, SB-type, U-type, and UJ-type. Each format is optimized for specific operations, demonstrating the flexibility and efficiency of the instruction set. The R-type format is used for register-to-register operations and includes fields for two source registers (rs1, rs2), a destination register (rd), and function codes (funct3, funct7) to specify the operation, along with an opcode. The I-type format supports immediate values for operations such as arithmetic or memory access, incorporating a 12-bit immediate field (imm[11:0]) alongside rs1, rd, funct3, and opcode fields. The S-type format is used for memory store instructions, where the 12-bit immediate is split across imm[11:5] and imm[4:0]. Similarly, the SB-type format is used for conditional branch instructions, with the immediate field organized differently to facilitate efficient branching. The U-type format supports large immediate values, using a 20-bit imm[31:12] field, and is ideal for instructions requiring absolute address calculations. Lastly, the UJ-type format, designed for jump instructions, includes a 20-bit immediate field (imm[20:1, 11, 19:12]) to allow for large jump offsets. This modular and structured approach ensures that RISC-V maintains simplicity while accommodating various instruction requirements efficiently [8].

The table below is the the list of computational instructions of RV32I ISA

Table 2: Listing of RV32I computational instructions.

Instruction		Format	Meaning
`add`	`rd, rs1, rs2`	R	Add registers
`sub`	`rd, rs1, rs2`	R	Subtract registers
`sll`	`rd, rs1, rs2`	R	Shift left logical by register
`srl`	`rd, rs1, rs2`	R	Shift right logical by register
`sra`	`rd, rs1, rs2`	R	Shift right arithmetic by register
`and`	`rd, rs1, rs2`	R	Bitwise AND with register
`or`	`rd, rs1, rs2`	R	Bitwise OR with register
`xor`	`rd, rs1, rs2`	R	Bitwise XOR with register
`slt`	`rd, rs1, rs2`	R	Set if less than register, 2's complement
`sltu`	`rd, rs1, rs2`	R	Set if less than register, unsigned
`addi`	`rd, rs1, imm[11:0]`	I	Add immediate
`slli`	`rd, rs1, shamt[4:0]`	I	Shift left logical by immediate
`srli`	`rd, rs1, shamt[4:0]`	I	Shift right logical by immediate
`srai`	`rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate
`andi`	`rd, rs1, imm[11:0]`	I	Bitwise AND with immediate
`ori`	`rd, rs1, imm[11:0]`	I	Bitwise OR with immediate
`xori`	`rd, rs1, imm[11:0]`	I	Bitwise XOR with immediate
`slti`	`rd, rs1, imm[11:0]`	I	Set if less than immediate, 2's complement
`sltiu`	`rd, rs1, imm[11:0]`	I	Set if less than immediate, unsigned
`lui`	`rd, imm[31:12]`	U	Load upper immediate
`auipc`	`rd, imm[31:12]`	U	Add upper immediate to `pc`

3.2. Features of RV32I

3.2.1. Instruction Categories

RV32I covers the following main categories of instructions:

❖ Arithmetic and Logical Operations

- Basic Arithmetic: ADD, SUB, ADDI (immediate addition), LUI (load upper immediate).
- Logical Operations: AND, OR, XOR, and NOT.
- Shift Operations: SLL (shift left logical), SRL (shift right logical), SRA (shift right arithmetic).
- Comparison: SLT and SLTU (set less than, signed and unsigned) for conditional operations based on register values.

❖ Memory Access Operations

- Load and Store: LW (load word), SW (store word).
- Byte and Halfword Support: LB, LBU (load byte, load byte unsigned), LH, LHU (load halfword, load halfword unsigned), SB (store byte), and SH (store halfword).
- These instructions enable efficient data transfer between registers and memory, supporting byte, halfword, and word (4 bytes) sizes.

❖ Control Flow Instructions

- Conditional Branches: BEQ (branch if equal), BNE (branch if not equal), BLT (branch if less than), BGE (branch if greater than or equal).

- Unconditional Jumps: JAL (jump and link), JALR (jump and link register).
- These instructions provide essential control flow, enabling loops, conditional execution, and function calls.

3.2.2. Instruction Formats in RV32I

RV32I uses a few main instruction formats to standardize encoding and simplify decoding:

- R-Type: Used for register-to-register arithmetic and logic operations (ADD, SUB).
- I-Type: Used for immediate arithmetic and load instructions (ADDI, LW).
- S-Type: Used for store instructions, specifying a base register and offset for memory storage (SW).
- B-Type: Used for conditional branching (BEQ, BNE).
- U-Type: Used for upper immediate operations with 20-bit immediates (LUI).
- J-Type: Used for jumps with immediate offsets (JAL).

3.2.3. Registers in RV32I

Register Set: RV32I uses 32 general-purpose registers (x0 to x31), each 32 bits wide:

- x0 (Zero Register): Hardwired to zero, always reads as 0 and discards any value written to it, helping simplify certain instructions.
- x1 (Return Address): Commonly used in jumps and procedure calls to hold the return address.
- x2 (Stack Pointer): Often used as the stack pointer in function calls and recursive algorithms.
- x31 to x3 serve various purposes based on the application or calling conventions.

3.2.4. RV32I Pipeline Compatibility

RV32I is designed to work efficiently with a **5-stage pipeline**:

- Fetch: The instruction is fetched from memory.
- Decode: Decodes the opcode and fetches operands.
- Execute: Executes the operation (arithmetic).
- Memory Access: For loads or stores, data is accessed in memory.
- Write-back: Writes the result back to the destination register.

This structure allows RV32I to achieve good performance while remaining power-efficient and straightforward to implement.

3.2.5. Applications of RV32I

- Embedded Systems: RV32I is widely used in IoT and microcontroller applications because of its simplicity, low power consumption, and reduced hardware requirements.

- Education and Research: As a foundational ISA, RV32I is used in academia for teaching and exploring computer architecture fundamentals.
- Basic Control Systems: The small, efficient RV32I processors are suitable for control systems in robotics, industrial automation, and real-time processing [4].

3.2.6. Advantages of RV32I

- Simplicity and Efficiency: With a minimal instruction set, RV32I is easy to implement and optimize for various environments.
- Modularity: RV32I can be used as a standalone core or as the basis for more complex processors with additional extensions (RV32IM for integer multiplication).
- Open-Source Flexibility: As part of the open RISC-V ISA, RV32I offers a royalty-free, extensible platform for experimentation and innovation

3.2.7. FPGA on RISC-V.

3.2.7.1. AI Application (CNN).

Convolutional Neural Networks (CNNs) require massive parallelism due to the high-precision, floating-point arithmetic operations they perform. So, demand of processing power in them is significantly higher than what a standard CPU can offer. This has traditionally made CNNs more suited for running on a Graphics Processing Unit (GPU). However, GPUs consume much more power than CPUs, rendering the former impractical for implementing CNNs in Edge AI (Artificial Intelligence), where restraining power consumption is paramount [1]. On the other hand, FPGAs (Field Programmable Gate Arrays) are more suited for AI computing at the edge as they consume much lesser power than GPUs and even CPUs. Additionally, GPUs and CPUs are not suited for real-time AI applications, which require both high throughput and low latency at the same time and FPGAs excel at all these requirements. [2] A comparison of deploying a CNN for MI classification across different computing platforms, namely, CPU-, embedded GPU-, and FPGA-based. Specifically, it delivers an impressive reduction of up to 89% in power consumption compared to the CPU and 71% compared to the GPU and up to a 98% reduction in memory footprint for model inference, albeit at the cost of a 39% increase in inference time compared to the GPU. Both the embedded GPU and FPGAoutperform the CPU in terms of inference time.

3.2.7.2. FPGAs for Edge AI

GPUs are time-efficient at AI training which happens offline. Although training time is not as critical as actual inferencing, GPUs remain the best platform for achieving a minimum training time. However, GPUs are power-hungry, rendering them unsuitable for AI inference at the edge where power consumption is a major limitation. In FPGAs, different inputs are pipelined through massively parallel reconfigurable logic fabric and produce outputs concurrently leading to high bit-level parallelism. Moreover, no waiting is required for all the weights to be loaded (in contrast to GPU batches), resulting in low latency. Pipelined designs guarantee high throughput alongside the FPGA's capabilities of customizable data precisions as per the design requirements. Furthermore, with customizable memory hierarchies on the FPGA, the design can utilize faster on-chip memories. As a result, FPGAs can achieve high throughput with low latency at the same

time. This makes FPGAs the most suitable hardware platform for real-time inferencing of most practical AI/DL solutions.[1]

3.2.7.3. FPGA versus GPU/GPU

The FPGA exhibited a remarkable 89% reduction in power consumption compared to the CPU and a 71% reduction compared to the embedded GPU. Moreover, transitioning to the FPGA enabled a reduction in memory footprint of approximately 98%. However, the FPGA incurred an inference time 39% higher than the GPU, attributed to the specific implementation of the FPG-AI framework used for compressing and optimizing the CNN, which lacks optimization for convolutional filters with large widths. Nevertheless, the FPGA's inference time could be reduced by modifying the underlying FPG-AI framework microarchitecture, potentially widening the gap between GPU- and FPGA-based deployment.[2]

Table 3: Reesult of aggregation among the different plattform deployments

	CPU	**FPGA**	**Jetson Nano**
Power Consumption (W)	13.22	1.47	5
Inference Time (ms)	121.57	19.97	15.85
FOM (Wxs)	1.61	$29.36x10^{-3}$	$79.25x10^{-3}$
Memory footprint (Mb)	476	6.37	415
Test accuracy	81%	77%	81%

3.2.7.4. FPGA Framework For Convolutional Neural Networks.

Recent works have pushed the performance of GPU implementations of CNNs to significantly improve their classification and training times. With these improvements, many frameworks have become avail able for implementing CNNs on both CPUs and GPUs, with no support for FPGA implementations To Present a modified version of the popular CNN framework Caffe, with FPGA support. This allows for classification using CNN models and specialized FPGA implementations with the flexibility of reprogramming the device when necessary, seamless memory transactions between host and device, simple-to-use test benches, and the ability to create pipelined layer implementation [3].

Present an adaptation of the Caffe CNN framework with support for the Xilinx FPGA environment. This adaptation allows us to launch CNN classification on CPU-FPGA-based systems.Describing a modification to the Winograd convolution algorithm to further reduce DSP utilization for FPGA based implementations.Results show that the architecture achieves approximately 50 GFLOPS across the 3×3 convolution layers of the benchmark suite, while using 83.2% of the FPGA environment (Xilinx Virtex 7 XC7VX690T-2).

Table 4: CPU, GPU/ FPGA 3x3 Convolution Benchmark Result.

Network	CPU Run Time (ms)	GPU Run Time (ms)	FPGA Run Time (ms)	CPU GFLOPS	GPU GFLOPS	FPGA GFLOPS
AlexNet (64 Images)	492	261.2	1,010	94	177.1	45.8
VGG A (32 Images)	4,310	745.14 [a]	8,713	111.2	642.9	55
Overfeat (64 Images)	2,030	387.1	4,781	139.2	730.2	59.1
GoogleNet (64 Images)	1,506	209.66 [b]	2,937	81.8	587.8	42
Geometric Average	1,595.6	354.51	3,333.9	104.46	470.17	50

3.3. RISC-V Pipeline Architecture

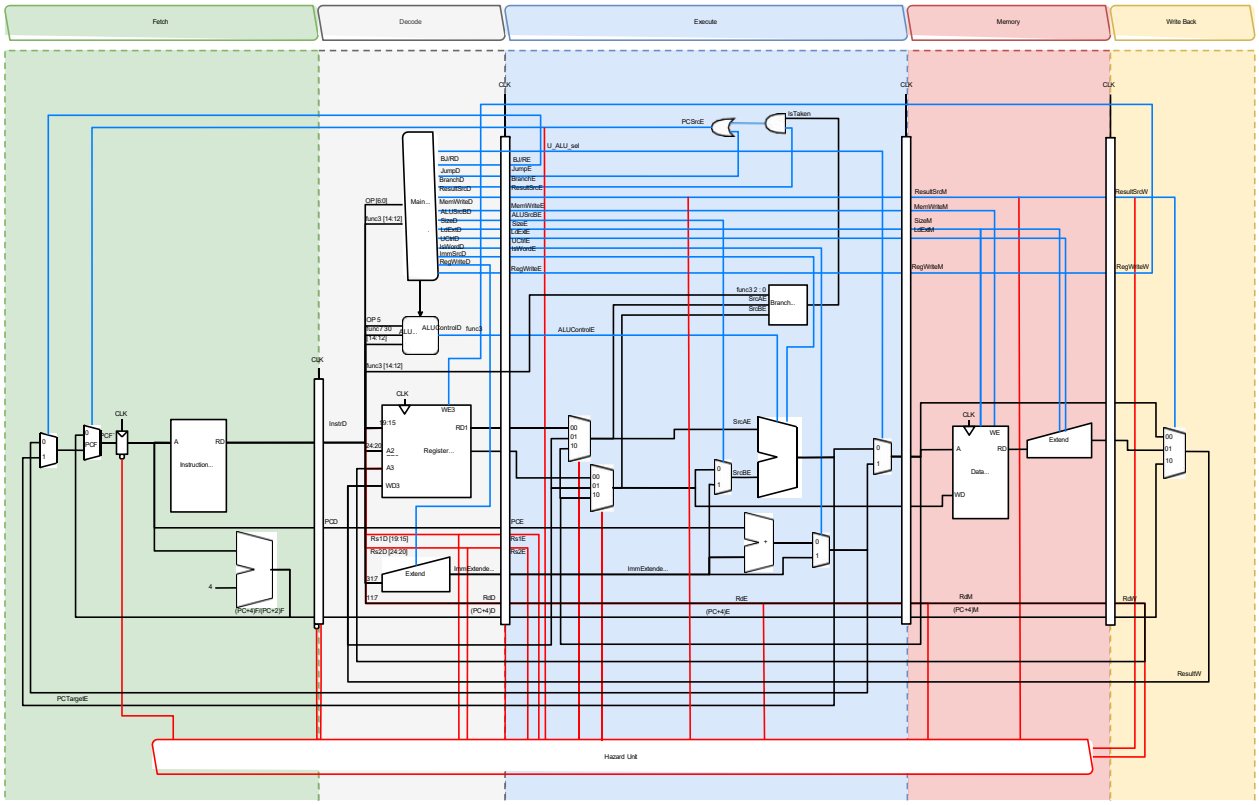

Figure 4: RV32I Pipeline Architecture

The RV32I pipeline is divided into 5 main stages:

1. **Fetch (IF)**: Fetches instructions from instruction memory.
2. **Decode (ID)**: Decodes the instruction and reads values from registers.
3. **Execute (EX)**: Performs arithmetic and logical operations.
4. **Memory Access (MEM)**: Accesses memory for LOAD/STORE instructions.
5. **Writeback (WB)**: Writes the result back to the destination register.

3.3.1. Instruction Fetch Stage

The instruction fetch stage is responsible for retrieving the next instruction to be executed from memory. This stage involves accessing the instruction memory using the program counter (PC) which holds the address of the current instruction. The fetched instruction is then sent to the subsequent stages of the pipeline for decoding and execution. The PC is typically updated to point to the next instruction in sequence, which could involve incrementing it by the size of an instruction which is 4 or updating it based on branch or jump instructions. This is determined by the two multiplexers in IF-stage whose select signals, *PCSrcE* and *BJ/RE*, are decoded by the main decoder in ID-stage and the branch unit in EX-stage.

3.3.2. Instruction Decode Stage

The instruction decode stage involves interpreting the fetched instruction's opcode and other fields to determine the operation to be performed. This stage decodes the instruction into control signals that will orchestrate how the datapath should process the instruction. It also involves reading operands from the register file if the instruction requires them. The outcome of this stage is a set of signals that guide the subsequent execution stage on how to perform the instruction's specified operation.

3.3.2.1. Register File

For RV32I, there are 32 integer registers, from x0 to x31, each of width 32-bit. The usage of each register is as shown in Table 5.

Table 5. RISC-V Integer Registers

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary registers
s0/fp	x8	Saved register/Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments/Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporary registers

The block diagram of the register file is shown in *figure 5.*

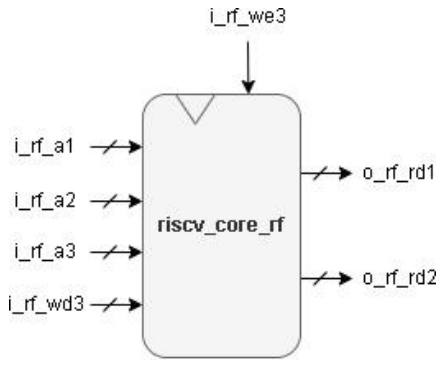

Figure 5. Register File Block Diagram

The block interface of the register file is shown in *Table 6.*

Table 6. Block Interface of Register File

Port Name	Direction	Width	Description
i_rf_clk	Input	1	Clock
i_rf_a1	Input	5	Address of rs1 to be read from
i_rf_a2	Input	5	Address of rs2 to be read from
i_rf_a3	Input	5	Address of rd1 to be written to
i_rf_wd3	Input	32	Data to be written to rd1

i_rf_we3	Input	1	Write Enable
o_rf_rd1	Output	32	Data read from rs1
o_rf_rd2	Output	32	Data read from rs2

3.3.2.2. Main Decoder

The main decoder plays a crucial role in the instruction execution pipeline. It receives the instruction fetched from instruction memory and decodes it to determine the necessary control signals for subsequent stages. It extracts relevant fields from the instruction, such as the opcode, source registers, immediate values, and destination registers. Based on the opcode and other information, it generates control signals for various components within the processor. The control signals generated are as follows:

- The main decoder determines the ALU operation (addition, subtraction, logical operations, etc.) required by the instruction.
- For load/store instructions, it sets control signals to access memory (load data from or store data to memory).
- If the instruction is a branch, the decoder decides whether to take the branch based on comparison results.
- It specifies whether the result should be written back to a register.
- The main decoder activates the appropriate multiplexers (MUXes) in subsequent stages.

The block diagram of the main decoder is shown in *Figure 6*.

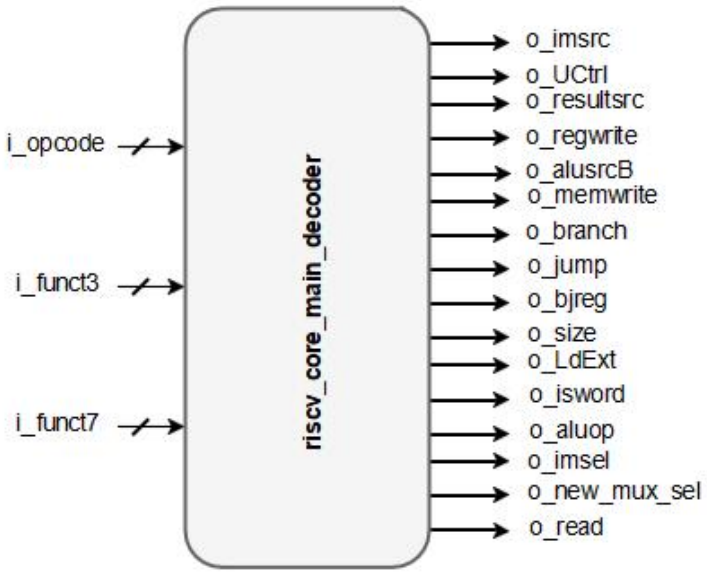

Figure 6. Main Decoder Block Diagram

The block interface of the main decoder is shown in *Table 7*.

Table 7. Block Interface of Main Decoder

Port Name	Direction	Width	Description
i_opcode	Input	7	Opcode bits of the instruction
i_funct3	Input	3	funct3 bits of the instruction
i_funct7	Input	7	funct7 bits of the instruction
o_imsrc	Output	3	Specifies extend operation on the immediate
o_UCtrl	Output	1	Selector for mux to select U-type instructions
o_resultsrc	Output	2	Selector for writeback mux
o_regwrite	Output	1	Write enable to register file

o_alusrcB	Output	1	Selector for a mux to select between data read from register rd2 and the extended immediate
o_memwrite	Output	1	Write enable for data cache
o_branch	Output	1	Indicates a branch instruction
o_jump	Output	1	Indicates a jump instruction
o_bjreg	Output	1	Selector for a mux to select next PC for *JALR* instruction
o_size	Output	2	Determines the size of data to be read from or written to data memory.
o_LdExt	Output	1	Determines whether to sign or zero extend the data read from data memory.
o_isword	Output	1	Determines whether the instruction is a word instruction or not.
o_aluop	Output	1	Enables the ALU decoder for any ALU operation
o_imsel	Output	1	Selector for a mux to select between ALU result or Mul/Div result.
o_new_mux_sel	Output	1	Selector for a mux to select between ALU/MUL/DIVdatapath or U instructions datapath.

3.3.2.3. ALU Decoder

The ALU decoder is responsible for interpreting the funct3, bit number 5 of the opcode, and bits number 0 and 5 of the funct7 of an instruction and determining the specific operation to be performed by the ALU. It generates control signals that guide the ALU's behaviour during execution.

The block diagram of the ALU decoder is shown in *Figure 7*.

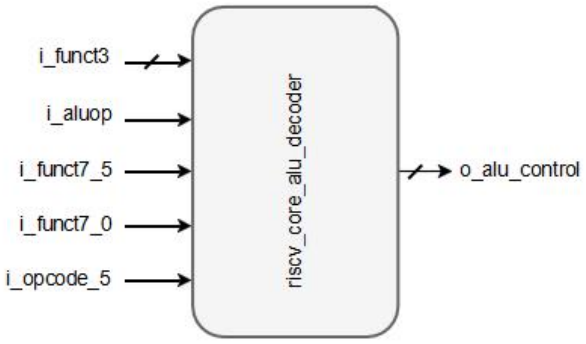

Figure 7. ALU Decoder Block Diagram

The block interface of the ALU decoder is shown in *Table 8*.

Table 8. The Block Interface of ALU Decoder

Port Name	Direction	Width	Description
i_funct3	Input	3	funct3 bits of the instruction
i_aluop	Input	1	Enables the ALU decoder
i_funct7_5	Input	1	Bit number 5 of funct7 part of the instruction
i_funct7_0	Input	1	Bit number 0 of funct7 part of the instruction
i_opcode_5	Input	1	Bit number 5 of opcode part of the instruction

o_alu_control	Output	4	Specifies the ALU operation

3.3.2.4. Immediate Extend

The immediate extend extends immediate values to the appropriate size for processing. Since RV32I is a 32-bit architecture, immediate values within instructions are often shorter than 32 bits and need to be extended to the full width before they can be used in arithmetic operations. The immediate extend block takes these shorter immediates and sign-extends them to 32 bits, preserving the value's sign and magnitude for correct computation in the datapath.

The block diagram of the immediate extend is shown in *Figure 8*.

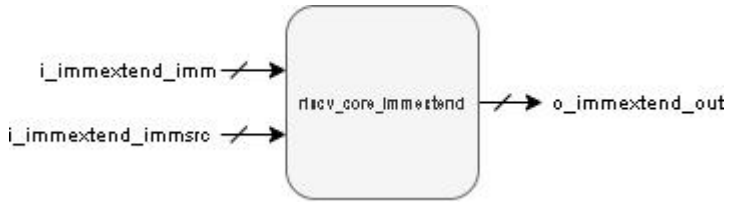

Figure 8. Immediate Extend Block Diagram

The block interface of the immediate extend is shown in *Table 9*.

Table 9. Block Interface of Immediate Extend

Port Name	Direction	Width	Description
i_immextend_imm	Input	25	Instr[31:20]
			Instr[31:25], Instr[11:7]
			Instr[31:12]
i_immextend_immsrc	Input	3	Type of extend according to instruction type
o_immextend_out	Output	32	Extended output

The function table of the immediate extend is shown in *Table 10*.

Table 10. Function Table of Immediate Extend

Format	i_immextend_immsrc	o_immextend_out
I	3'b000	{{52{instr[31]}},instr[31:20]}
S	3'b001	{{52{Instr[31]}}, Instr[31:25], Instr[11:7]}
B	3'b010	{{52{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}
J	3'b011	{{44{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}
R	3'b100	{32{instr[31]} ,instr[31:12], {12{1'b0}}}

3.3.3. Execute Stage

The Execute Stage is where the actual computation of the instruction takes place. This stage uses the control signals from the decode stage and the operands from the register file or immediate extend block to perform the specified operation. The arithmetic logic unit (ALU) within the datapath is typically involved in this process, carrying out arithmetic and logical operations. The result of the ALU's computation may then be used for updating the program counter, written back to a register, or used in memory access operations in subsequent stages.

3.3.3.1. ALU (Arithmetic and Logical Unit)

The Arithmetic Logic Unit (ALU) is a critical component of the datapath that performs arithmetic and logical operations. It takes two operands as input and produces one output based on the operation code (opcode) it receives from the control unit. The ALU can perform a variety of operations such as addition, subtraction, and bitwise operations like AND, OR, XOR, and shift operations.

The block diagram of the ALU is shown in *Figure 9*.

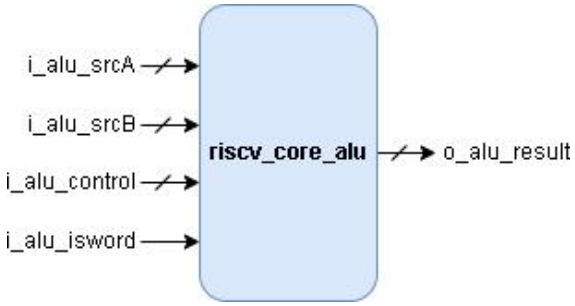

Figure 9. ALU Block Diagram

The block interface of the ALU is shown in *Table 11*.

Table 11. Block Interface of ALU

Port Name	Direction	Width	Description
i_alu_srcA	Input	32	Operand 1
i_alu_srcB	Input	32	Operand 2
i_alu_control	Input	4	Specifies ALU operation
i_alu_isword	Input	1	Specifies whether the instruction is word or not
o_alu_result	Output	32	ALU result

The function table of the ALU is shown in *Table 12*.

Table 12. Function Table of ALU

i_alu_isword	i_alu_control	Operation
1'b0	4'b0000	o_alu_result = i_alu_srcA + i_alu_srcB
1'b0	4'b0001	o_alu_result = i_alu_srcA - i_alu_srcB
1'b0	4'b0110	o_alu_result = i_alu_srcA ^ i_alu_srcB
1'b0	4'b0011	o_alu_result = i_alu_srcA | i_alu_srcB
1'b0	4'b0010	o_alu_result = i_alu_srcA & i_alu_srcB
1'b0	4'b0100	o_alu_result = i_alu_srcA << i_alu_srcB[5:0]
1'b0	4'b0111	o_alu_result = i_alu_srcA >> i_alu_srcB[5:0]
1'b0	4'b1111	o_alu_result = i_alu_srcA >>> i_alu_srcB[5:0]
1'b0	4'b0101	o_alu_result = i_alu_srcA < i_alu_srcB
1'b0	4'b1000	o_alu_result = i_alu_srcA(U) < i_alu_srcB(U)
1'b1	4'b0000	o_alu_result = i_alu_srcA[31:0] + i_alu_srcB[31:0] (sign ext)
1'b1	4'b0001	o_alu_result = i_alu_srcA[31:0] - i_alu_srcB[31:0] (sign ext)

1'b1	4'b0100	o_alu_result = i_alu_srcA[31:0] << i_alu_srcB[4:0] (sign ext)
1'b1	4'b0111	o_alu_result = i_alu_srcA[31:0] >> i_alu_srcB[4:0] (sign ext)
1'b1	4'b1111	o_alu_result = i_alu_srcA[31:0] >>> i_alu_srcB[4:0] (sign ext)

3.3.3.2. Branch Unit

In the RV32I microarchitecture, the Branch unit is responsible for handling branch instructions, such as conditional and unconditional jumps. It evaluates the branch condition based on the comparison operations it performs and determines whether to take the branch or not by the flag, *istaken*, it outputs. If a branch is taken, the new program counter (PC) value is calculated in ALU based on the branch target address. This unit plays a crucial role in controlling the flow of execution within the processor.

The block diagram of the Branch Unit is shown in *Figure 10*.

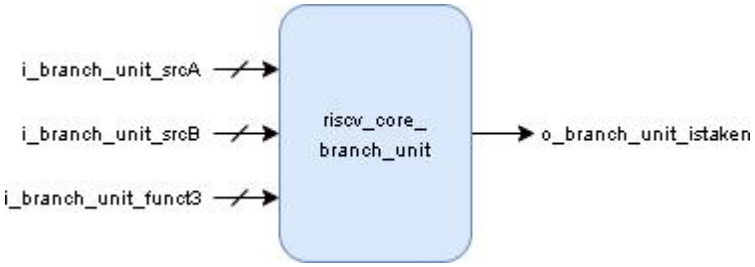

Figure 10. Branch Unit Block Diagram

The block interface of the Branch Unit is shown in table *13*.

Table 13. Block Interface of Branch Unit

Port Name	Direction	Width	Description
i_btanch_unit_srcA	Input	32	Operand 1
i_branch_unit_srcB	Input	32	Operand 2
i_branch_unit_funct 3	Input	3	Specifies branch operation
o_branhc_unit_ista ken	Output	1	Flag specifies if the branch is taken or not

The function table of the Branch Unit is shown in *Table 14*.

Table 14. Function Table of Branch Unit

i_branch_unit_fun ct3	Instr.	Operation
3'b000	BEQ	istaken = i_branch_unit_srcA == i_branch_unit_srcB
3'b001	BNE	istaken = i_branch_unit_srcA != i_branch_unit_srcB
3'b100	BLT	istaken = i_branch_unit_srcA < i_branch_unit_srcB
3'b101	BGE	istaken = i_branch_unit_srcA >= i_branch_unit_srcB
3'b110	BLTU	istaken = i_branch_unit_srcA(U) < i_branch_unit_srcB(U)
3'b111	BGEU	Istaken = i_branch_unit_srcA(U) >= i_branch_unit_srcB(U)

3.3.4. Memory Stage

The memory stage is a crucial part of the processor pipeline. It handles memory- related operations, such as loading data from memory (load instructions) and storing data to memory (store instructions). Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective address is obtained by adding register rs1 to the sign- extended 12-bit offset. Loads copy a value from memory to register rd. Stores copy the value in register rs2 to memory.

❖ Load Instructions:

- *LD* instruction loads a 32-bit value from memory into rd.
- *LW* instruction loads a 32-bit value from memory, then sign-extends to 32-bits before storing in rd.
- *LH* loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd.
- *LHU* loads a 16-bit value from memory but then zero extends to 32- bits before storing in rd.
- *LB and LBU are defined analogously for 8-bit values.*

❖ Store Instructions:

- *SD* instruction stores 32-bit value from register rs2 to memory.
- *SW* instruction stores 32-bit value from the low bits of register rs2 to memory.
- *SH* instruction stores 16-bit value from the low bits of register rs2 to memory.
- *SB* instruction stores 8-bit value from the low bits of register rs2 to memory.

3.3.4.1. Load Extend

The load extend is responsible for extending the loaded value from memory to 32-bit to be stored in register file according to the size of the load instruction executed.

The block diagram of the load extend is shown in Figure 11.

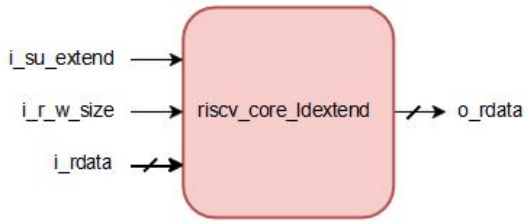

Figure 11. Load Extend Block Diagram

The block interface of the load extend is shown in *Table 15*.

Table 15. Block Interface of Load Extend

Port Name	Direction	Width	Description
i_su_extend	Input	1	Determines whether the instruction is signed or unsigned.
i_r_w_size	Input	2	Determines the size of the load instruction e.g., LD, LW, LH, LB.
i_rdata	Input	32	Data loaded from memory without extend.
o_rdata	Output	32	Data loaded from memory after sign/zero extend.

The function table of the load extend is shown in *Table 16*

Table 16. Function Table of Load Extend

Instruction	i_su_extend	i_r_w_size	o_rdata
LB	1'b0	2'b00	{{56{i_rdata[7]}}, i_rdata[7:0]}
LH	1'b0	2'b01	{{48{i_rdata[15]}}, i_rdata[15:0]}
LW	1'b0	2'b10	{{32{i_rdata[31]}}, i_rdata[31:0]}
LD	1'b0	2'b11	i_rdata
LBU	1'b1	2'b00	{{56{1'b0}}, i_rdata[7:0]}
LHU	1'b1	2'b01	{{48{1'b0}}, i_rdata[15:0]}
LWU	1'b1	2'b10	{{32{1'b0}}, i_rdata[31:0]}
LD	1'b1	2'b11	i_rdata

3.3.5. Writeback Stage

The WB stage handles the result of an executed instruction. It writes the computed value back to the destination register. It is just a mux that selects the data to be written to the register file according to the instruction being executed. The selection is based on the control signal *resultsrc* which is computed by the main decoder.

The function table of WB mux is shown in *Table 17*.

Table 17. Function Table of Writeback MUX

resultsrc	Operation
2'b00	Writes back the data computed from EX stage
2'b01	Writes back the data loaded from data memory
2'b10	Writes back PC+4 for JAL & JALR instructions

3.3.6. Hazard Unit

The Hazard Unit manages both control and data hazards in the pipelined processor. A data hazard happens when an instruction tries to read a register that hasn't been updated by a preceding instruction. A control hazard arises when the next instruction to be fetched hasn't been determined before the fetch takes place. To ensure the processor runs the program correctly, we enhanced it with a Hazard Unit that identifies and appropriately handles hazards.

Raw data hazards occur when an instruction depends on the result of a previous instruction that hasn't yet been written to the register file. These hazards can be resolved through forwarding if the result is computed in time, otherwise, the pipeline must stall until the result is available.

Control hazards happen when the decision on the next instruction to fetch hasn't been made by the time it needs to be fetched. These hazards can be addressed by stalling the pipeline until the decision is made or by predicting the next instruction and then flushing the pipeline if the prediction is wrong. The earlier the decision is made, the fewer instructions need to be flushed in case of a misprediction..

The block diagram of the hazard unit is shown in *Figure 12*.

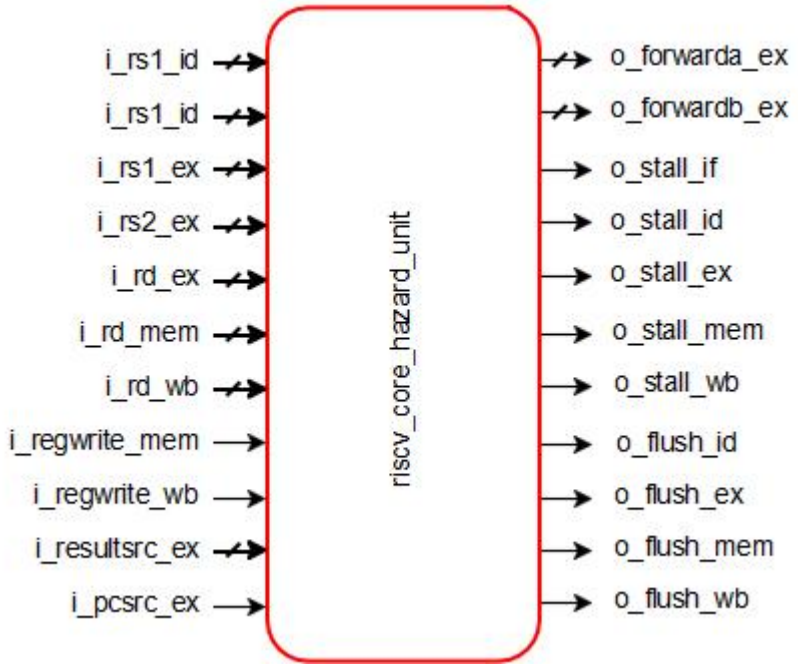

Figure 12. Hazard Unit Block Diagram

The block interface of the hazard unit is shown in *Table 18.*

Table 18. Block Interface of Hazard Unit

Port Name	Direction	Width	Description
i_hazard_unit_rs1_id	Input	5	RV32I Signals
i_hazard_unit_rs2_id	Input	5	
i_hazard_unit_rs1_ex	Input	5	
i_hazard_unit_rs2_ex	Input	5	
i_hazard_unit_rd_ex	Input	5	RV32I Signals
i_hazard_unit_mbusy	Input	1	
o_hazard_unit_forwarda_ex	Output	2	Forwarding Signals
o_hazard_unit_forwardb_ex	Output	2	
o_hazard_unit_stall_if	Output	1	Stall Signals
o_hazard_unit_stall_id	Output	1	
o_hazard_unit_stall_ex	Output	1	
o_hazard_unit_stall_mem	Output	1	
o_hazard_unit_stall_wb	Output	1	
o_hazard_unit_flush_id	Output	1	Flush Signals
o_hazard_unit_flush_ex	Output	1	
o_hazard_unit_flush_mem	Output	1	
o_hazard_unit_flush_wb	Output	1	

4. Implementation of M-Extension for Integer Multiplication and Division

This chapter outlines the standard integer multiplication and division instruction extension, known as "M," which includes instructions for multiplying or dividing values stored in two integer registers.

RV32M ISA introduces a total of thirteen multiplication and division instructions as shown in *Table 19*.

Table 19. M-Extension for Integer Multiplication and Division Instruction.

Instruction	Format	Opcode	Funct3	Funct7	Description
mul	R	0110011	0x0	0x01	rd = (rs1 * rs2)[63:0]
mulh	R	0110011	0x1	0x01	rd = (rs1 * rs2)[127:32]
mulhsu	R	0110011	0x2	0x01	rd = (rs1 * rs2)[127:32]
mulhu	R	0110011	0x3	0x01	rd = (rs1 * rs2)[127:32]
div	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	R	0110011	0x7	0x01	rd = rs1 % rs2
mulw	R	0111011	0x0	0x01	rd = rs1[31:0] * rs2[31:0] (sign extend)
divw	R	0111011	0x4	0x01	rd = rs1[31:0] / rs2[31:0] (sign extend)
divuw	R	0111011	0x5	0x01	rd = rs1[31:0] / rs2[31:0] (sign extend)
remw	R	0111011	0x6	0x01	rd = rs1[31:0] % rs2[31:0] (sign extend)
remuw	R	0111011	0x7	0x01	rd = rs1[31:0] % rs2[31:0] (sign extend)

Figure 13 shows the microarchitecture of M-Extension.

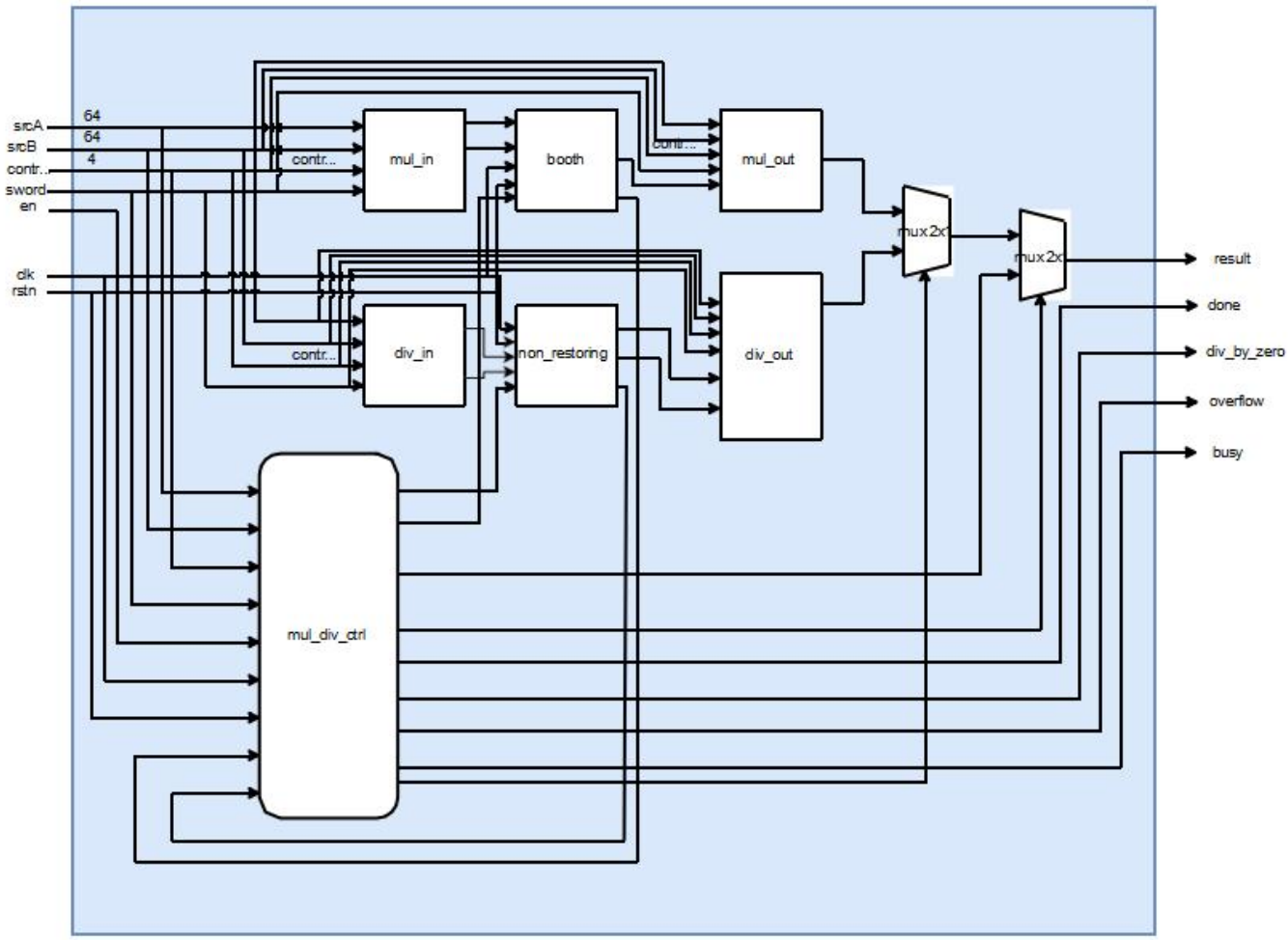

Figure 13. Microarchitecture of M-Extension.

4.1. Multiplier

4.1.1. Multiplier Algorithm

At first the choice of using an optimizing algorithm to implement the multiplication process was Booth's Algorithm since Booth's Algorithm is an efficient technique used for performing binary multiplication. It reduces the number of additional operations performed, thereby optimizing the power. But since Booth's Algorithm is used for singed operations and there are unsigned instructions such as *mulhsu* and *mulhu*, then we modified this algorithm to overcome this problem.

Steps of the algorithm:

1. Initialization
- Load the multiplier.
- Clear the accumulator and carry registers.
- Set the counter to 32 (for 32-bit multiplication).

2. Iterative Multiplication in MUL State:
- Check the LSB of the multiplier.
- If 1, add the multiplicand to the accumulator, if 0, do nothing
- Perform an arithmetic right shift on the combined registers {carry, accumulator, and multiplier}.
- Decrement counter.

3. Result

- If the counter reaches 0, the result is in the combined accumulator and multiplier.

4.1.2. Multiplier Design

In this section the main block of the multiplier will be illustrated:

4.1.2.1. mul_in Block

Since the multiplication algorithm is unsigned, we need to adjust the two inputs srcA and srcB to get the proper multiplicand and multiplier which must be positive to perform unsigned multiplication, so each instruction needs a proper adjustment.

- *mul* and *mulh*: if either one of the two inputs is negative, get its positive value using two's complement.
- *mulhsu*: if the multiplicand is negative, get its positive value using two's complement.
- *mulhu*: no adjustment needed.
- *mulw*: take only the first 32-bit since it's a word operation and get the positive value if there is a negative input.

The block diagram of mul_in block is shown in *Figure 14*.

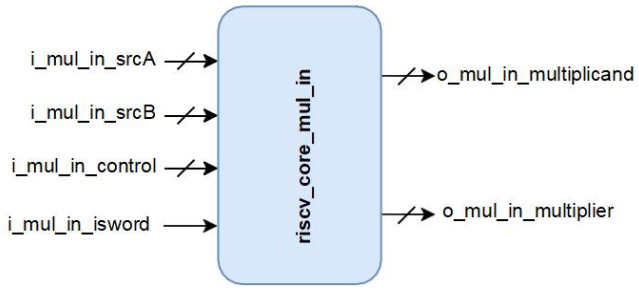

Figure 14. mul_in Block Diagram

The block interface of the mul_in block is shown in *Table 20.*

Table 20. Block Interface of mul_in Block

Port Name	Direction	Width	Description
i_mul_in_srcA	Input	32	Operand 1
i_mul_in_srcB	Input	32	Operand 2
i_mul_in_control	Input	2	Control bits to specify the instruction
i_mul_in_isword	Input	1	Specifies the word instructions
o_mul_in_multiplicand	Output	32	Multiplicand
o_mul_in_multiplier	Output	32	Multiplier

4.1.2.2. Booth Block

This block is responsible for performing the multiplication. The block diagram of the booth block is shown in *Figure 15*.

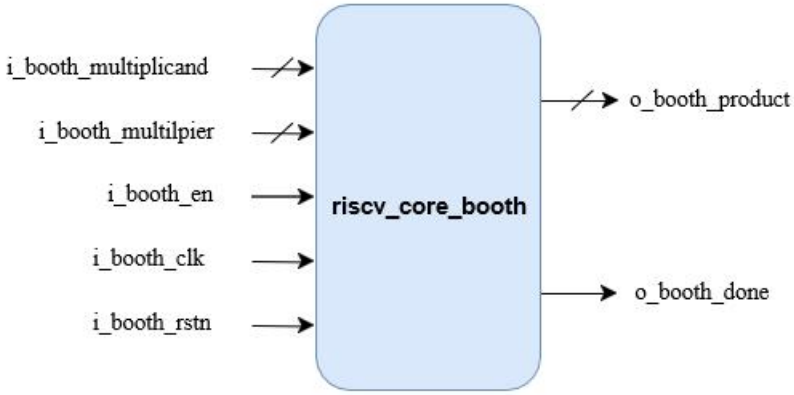

Figure 15. Booth Block Diagram

The block interface of the Booth is shown in *Table 21*.

Table 21. Block Interface of Booth Block

Port Name	Direction	Width	Description
i_booth_multiplicand	Input	32	Multiplicand
i_booth_multilpier	Input	32	Multiplier
i_booth_en	Input	1	Enable
i_booth_clk	Input	1	Clock
i_booth_rstn	Input	1	Reset
o_booth_done	Output	1	Done flag
o_booth_product	Output	128	Product result

4.1.2.3. mul_out Block

Since the previous block perform unsigned multiplication, this block is responsible for adjusting the sign of the product result of the signed instructions (Mul, Mulh, Mulhsu and Mulw).

If the two input operands have different sign, then get the two's compliment of the product result.

The block diagram of mul_out block is shown in *Figure 16*.

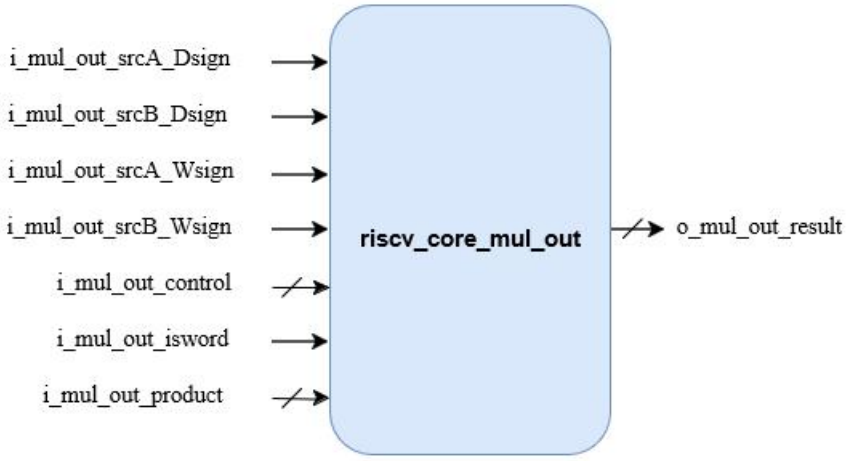

Figure 16. mul_out Block Diagram

The block interface of the mul_out is shown in *Table 19*.

Table 19. Block Interface of mul_out Block.

Port Name	Direction	Width	Description
i_mul_out_srcA_Dsign	Input	1	Operand 1 sign
i_mul_out_srcB_Dsign	Input	1	Operand 2 sign
i_mul_out_srcA_Wsign	Input	1	Word of Operand 1 (first 32-bit) sign
i_mul_out_srcB_Wsign	Input	1	Word of Operand 1 (first 32-bit) sign
i_mul_out_control	Input	2	Control bits to specify the instruction
i_mul_out_isword	Input	1	Specifies the word instructions
i_mul_out_product	Input	32	Product result of booth block
o_mul_out_result	Output	32	Multiplication result

4.2. Divider

4.2.1. Divider Algorithm

Our divider is implemented based on the non-restoring algorithm for unsigned division. The non-restoring division algorithm is a method used in digital arithmetic to divide two binary numbers. It is particularly efficient for hardware implementations because it avoids the need for restoring the partial remainder to a positive value after each subtraction operation. The non-restoring division algorithm reduces the number of arithmetic operations needed compared to other division methods, making it faster for hardware implementation.

The non-restoring algorithm flow chart is shown in *Figure 17.*

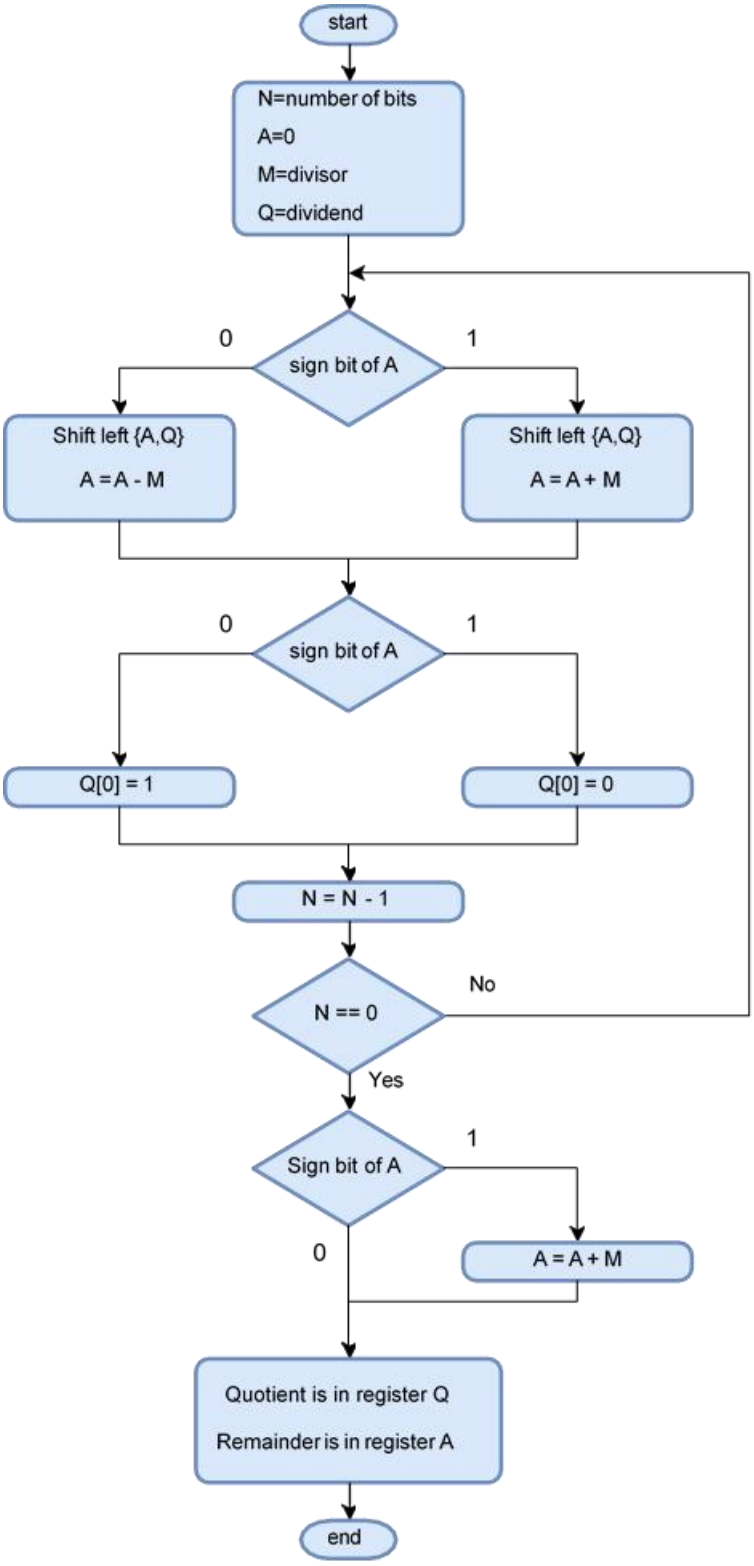

Figure 17. Non-Restoring Flow Chart

Steps of the algorithm:

1. In this step, the corresponding values are initialized in the registers as follows: register A is set to 0, register M holds the Divisor, register Q contains the Dividend, and N specifies the number of bits in the Dividend.
2. In this step, we will check the sign bit of A.
3. If the bit in register A is 1, shift the combined value of AQ to the left and perform the operation A = A + M. If the bit is 0, shift the combined value of AQ to the left and perform the operation A = A - M. In the case of 0, this involves adding the 2's complement of M to register A, with the result stored in A.
4. check the sign bit of A again.
5. If the bit in register A is 1, then Q[0] will be set to 0. If the bit is 0, then Q[0] will be set to 1. Here, Q[0] refers to the least significant bit of Q.
6. Then, the value of N, which serves as a counter, will be decremented.
7. If the value of N = 0, then we will go to the next step. Otherwise, we must again go to step 2.
8. If the sign bit of register A is 1, we will execute A = A + M.
9. In this step, register A holds the remainder, and register Q contains the quotient.

4.2.2. Divider Design

4.2.2.1. div_in Block

Same as the multiplier we need to adjust the inputs of the operation, Since the non-restoring algorithm is unsigned, we need to adjust the two inputs srcA and srcB to get the proper dividend and divisor which must be positive, and each instruction needs a proper adjustment.

- *div*, *divw*, *rem* and *remw*: if either one of the two inputs is negative, get its positive value using two's complement.
- *divu*, *divuw*, *remu* and *remuw*: no adjustment needed.

Note: for any word instruction get the first 32-bit only.

The block diagram of div_in is shown in *Figure 18*.

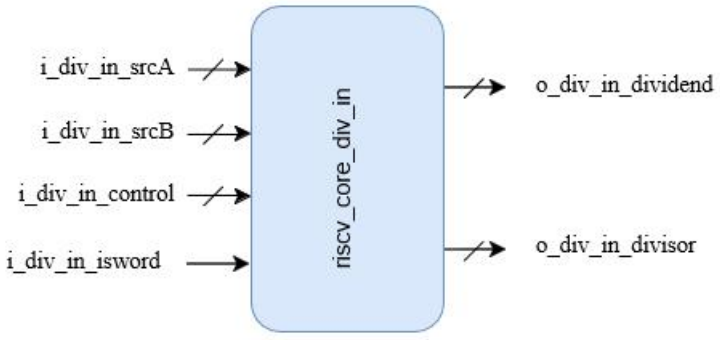

Figure 18. div_in Block Diagram.

The block interface of the div_in is shown in *Table 23*.

Table 23. Block Interface of div_in Block

Port Name	Direction	Width	Description
i_div_in_srcA	Input	32	Operand 1
i_div_in_srcB	Input	32	Operand 2
i_div_in_control	Input	2	Control bits to specify the instruction
i_div_in_isword	Input	1	Specifies the word instructions

o_div_in_dividend	Output	32	Dividend
o_div_in_divisor	Output	32	Divisor

4.2.2.2. non_restoring Block

This block is responsible for performing the division.

The block diagram of non-restoring is shown in *Figure 19*.

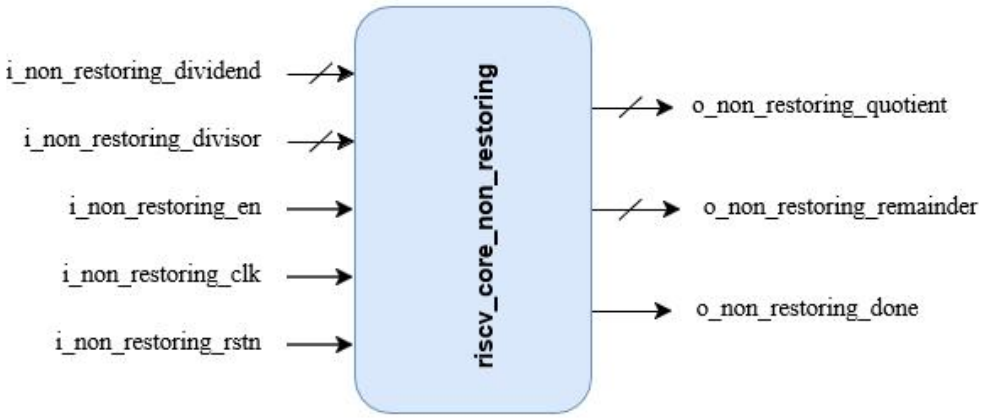

Figure 19. non_restoring Block Diagram

The block interface of the non-restoring is shown in *Table 24*.

Table 24. Block Interface of non_restoring Block.

Port Name	Direction	Width	Description
i_non_restoring_dividend	Input	32	Dividend
i_non_restoring_divisor	Input	32	Divisor
i_non_restoring_en	Input	1	Enable
i_non_restoring_clk	Input	1	Clock
i_non_restoring_rstn	Input	1	Reset
o_non_restoring_done	Output	1	Done flag
o_non_restoring_quotient	Output	32	Quotient
o_non_restoring_remainder	Output	32	Remainder

4.2.2.3. div_out Block

Since the previous block perform unsigned division, this block is responsible for adjusting the sign of the product result of the signed instructions (*div, divw, rem and remw*).

If the two input operands have different sign, then get the two's compliment of the product result. Assign the quotient or the remainder to the output of the block according to the instruction (division or remainder). The block diagram of div_out is shown in *Figure 20*.

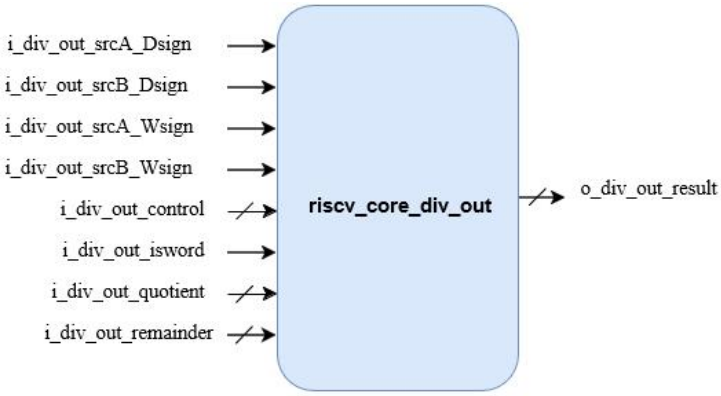

Figure 20. div_out Block Diagram.

The block interface of the div_out is shown in *Table 25.*

Table 25. Block Interface of div_out Block

Port Name	Direction	Width	Description
i_div_out_srcA_Dsign	Input	1	Operand 1 sign
i_div_out_srcB_Dsign	Input	1	Operand 2 sign
i_div_out_srcA_Wsign	Input	1	Word of Operand 1 (first 32-bit) sign
i_div_out_srcB_Wsign	Input	1	Word of Operand 1 (first 32-bit) sign
i_div_out_control	Input	2	Control bits to specify the instruction
i_div_out_isword	Input	1	Specifies the word instructions
i_div_out_quotient	Input	32	Quotient result of non-restoring block
i_div_out_remainder	Input	32	Remainder result of non-restoring block
o_div_out_result	Output	32	Result of the operation

4.3. M-Controller

Its primary role is to ensure efficient and accurate execution of M-Extension instructions by coordinating the necessary control signals and handling data flow within the processor. The controller optimizes performance by implementing fast paths for simple operations (fast operations). This contributes to the overall performance and functionality of the RISC-V core, enhancing its ability to handle computationally intensive tasks.

Figure 18 represents the finite state machine of the controller block which have four states:

- **IDLE**
- **MUL**: Performs Multiplication
- **DIV**: Performs Division
- **FAST**: Performs Fast Operations

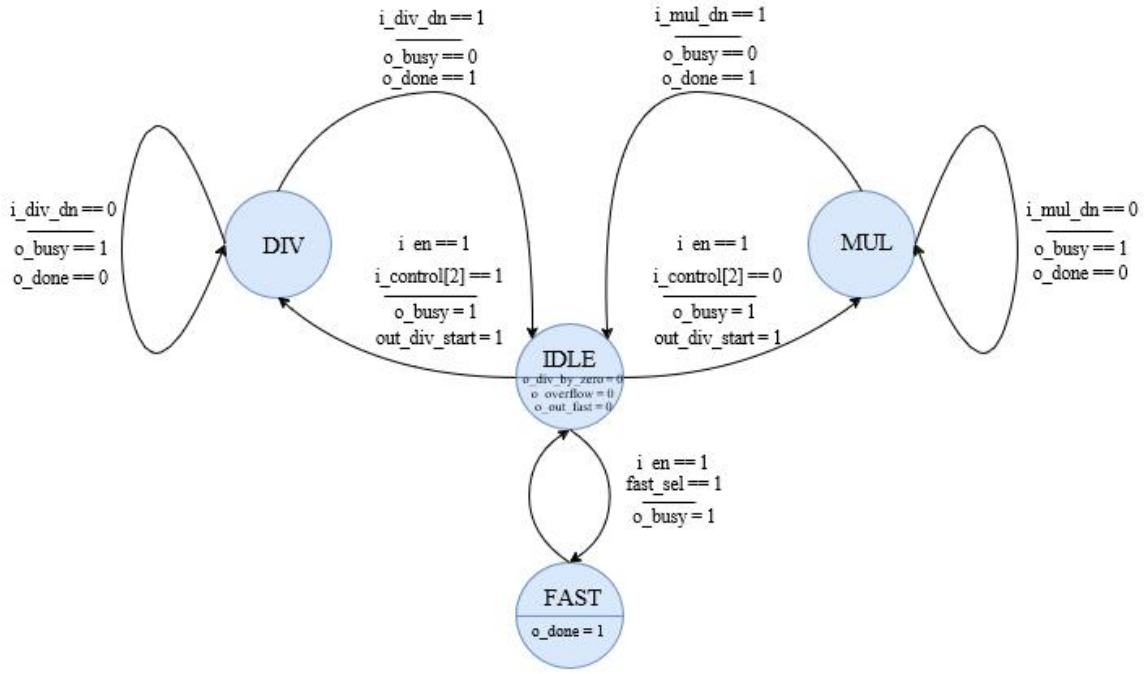

Figure 21. FSM of M-Controller.

We added Another feature to the implementation of the M-Extension, which is fast operations, fast operations mean preforming the basic operations during only one clock cycle and not waiting 32 clock cycles.

We created two registers, one for double word instructions and the other for word instructions. Each one is 7-bit wide. These two registers indicate whether one of the operands is -1, 0, 1, or indicate an overflow value. The format of these two registers is shown in *Table 26*.

Table 26. Format of Indication Registers of Fast Operations.

Fast DW reg	B == -1	B == 1	B == 0	A == -1	A == 1	A == 0	overflow
Fast W reg	B == -1	B == 1	B == 0	A == -1	A == 1	A == 0	overflow

Assuming A is the dividend or multiplicand, and B is divisor or multiplier, then, Fast operations are as shown in *Table 27*.

Table 27. Fast Operations Function Table.

Fast_reg	Instr	Operation	Fast_out
7’b1000001	REM	Overflow	0
	DIV		32’h8000_0000_0000_0000
7'b001xxx0	REM(U)	A%0	A
	DIV(U)	A/0	32’hFFFF_FFFF_FFFF_FFFF
	MUL	A*0	0
7'bxx00010	All	(0*B) or (0/B)	0
	REM(U)	A%-1	0

7'b1000000	DIV(U)	A/-1	2’s comp of A
	MUL	A*-1	2’s comp of A
	MULH(SU)(U)		32’hFFFF_FFFF_FFFF_FFFF
7’b0100000	REM(U)	A%1	0
	DIV(U)	A/1	A
	MUL	A*1	A
	MULH(SU)(U)		not fast operation
7'b1001000	REM(U)	-1%-1	0
	DIV(U)	-1/-1	1
	MUL	-1*-1	1
	MULH		0
	MULHSU(U)		not fast operation
7'b0100100	REM(U)	1%1	0
	DIV(U)	1/1	1
	MUL	1*1	1
	MULH(SU)(U)		0
7'b1000100	REM(U)	1%-1	0
	DIV(U)	1/-1	-1
	MUL	1*-1	-1
	MULH		-1
	MULHSU(U)		not fast operation
7'b0101000	REM(U)	-1%1	0
	DIV(U)	-1/1	not fast operation
	MUL	-1*1	-1
	MULH		0
	MULHSU(U)		-1
7'b0000100	REM(U)	1%B	1
	DIV(U)	1/B	0
	MUL	1*B	B
	MULH(SU)(U)		0

The block diagram of the *mul_div_ctrl* is shown in *Figure 22*.

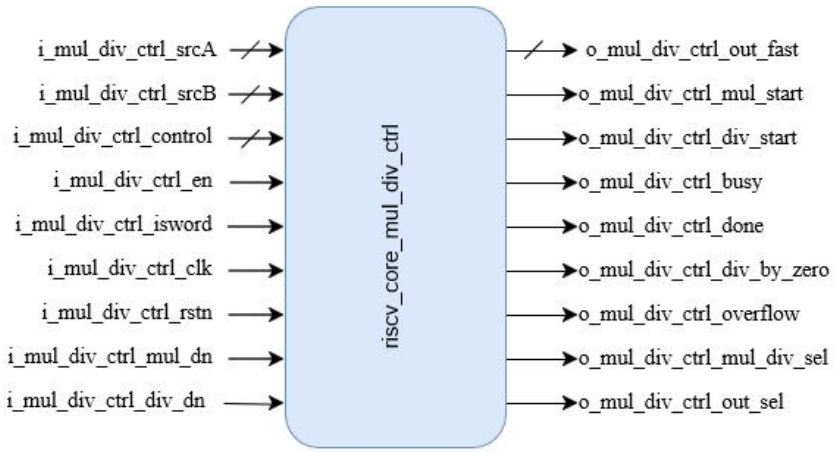

Figure 22. mul_div_ctrl Block Diagram.

The block interface of the *mul_div_ctrl* is shown in *Table 28*.

Table 28. Block Interface of mul_div_ctrl Block.

Port Name	Direction	Width	Description
i_mul_div_ctrl_srcA	Input	32	Operand 1
i_mul_div_ctrl_srcB	Input	32	Operand 2
i_mul_div_ctrl_control	Input	2	Control bits to specify the instruction
i_mul_div_ctrl_en	Input	1	Enable
i_mul_div_ctrl_isword	Input	1	Specifies the word instructions
i_mul_div_ctrl_clk	Input	1	Clock
i_mul_div_ctrl_rstn	Input	1	Reset
i_mul_div_ctrl_mul_dn	Input	1	Done flag for multiplication
i_mul_div_ctrl_div_dn	Input	1	Done flag for division
o_mul_div_ctrl_out_fast	Output	32	Fast operations result
o_mul_div_ctrl_mul_start	Output	1	Start signal for multiplication
o_mul_div_ctrl_div_start	Output	1	Start signal for division
o_mul_div_ctrl_busy	Output	1	Busy flag
o_mul_div_ctrl_done	Output	1	Done flag
o_mul_div_ctrl_div_by_zero	Output	1	Divide by zero flag
o_mul_div_ctrl_overflow	Output	1	Overflow flag
o_mul_div_ctrl_mul_div_sel	Output	1	Selects between Mul_out result and Div_out result according to instruction
o_mul_div_ctrl_out_sel	Output	1	Selects between Mul_div_result and fast_result according to operation

5. Implementation of C-Extension for Compressed Instructions

The C extension provides a reduced encoding for a subset of the base instructions. The subset was selected to cover the most used instructions in their most used form. The reduced encoding allows for code size improvements which benefit both low power and high-performance implementations.

It is not a standalone extension but is built on top of RV32I or RV32I. The C extension defines 9 new instruction formats, each of them fitting on 16-bit. These instruction formats are shown in *Table 29.*

Table 29. Compressed 16-bit RVC instruction formats.

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm	rs1′	imm		rd′	op
CS	Store	funct3	imm	rs1′	imm		rs2′	op
CA	Arithmetic	funct6		rd′/rs1′		funct2	rs2′	op
CB	Branch	funct3	offset	rs1′		offset		op
CJ	Jump	funct3	jump target					op

5.1. CI-type: Compressed Operations with Immediate

The diagram shown in Figure 20 illustrates the differences between the encoding of *addi* in the base ISA (top, 32-bit wide) and in the C extension (bottom, 16-bit wide): the immediate has been reduced from 12 to 6 bits, the same register index is used for both source and destination and the instruction operation is encoded on 5 bits rather than 10.

One thing to note is that the register index is still 5-bit wide, all the registers can be addressed in this opcode.

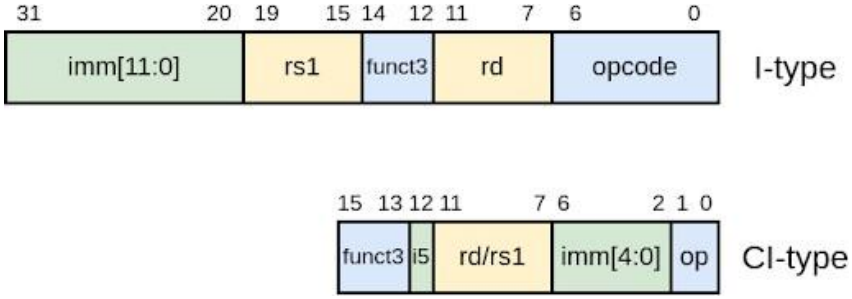

Figure 23. Comparison Between I-type Format and CI-type Format.

Obviously *c.addi* is less expressive than *addi*: smaller immediate range, source and destination have to be the same register. This affects all compressed instructions: they constitute a subset of their expanded counterparts, hopefully the most useful one.

5.2. CA-type: Compressed Arithmetic Instructions

Another compressed format is the CA-type (compressed arithmetic) illustrated by the diagram shown in Figure 21.

In the CA-type, one of the source register indices **rs1** is fused with destination **rd**, and the register indices are now 3-bit wide. Furthermore, they do not encode directly a register index, the indirection is presented in the following table (copied from the RVC standard): for example, 0b100 encodes **x12** in compressed instructions working on general purpose/integer registers. This is why the register indices are labelled with **rd'**, **rs1'**, and **rs2'** (rather than rd, rs1, and rs2). The encoded registers were selected because they are the most frequently used ones.

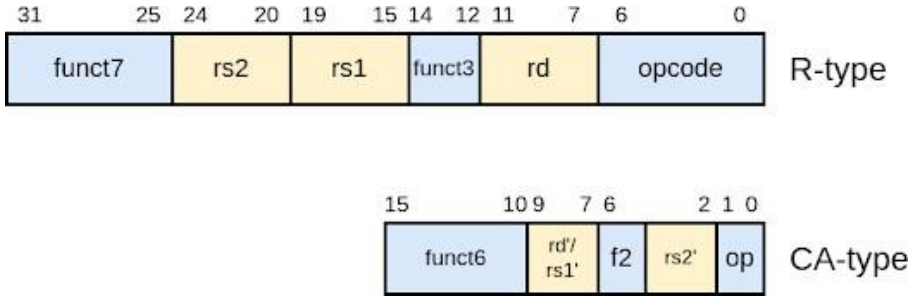

Figure 24. Comparison Between R-type format and in CA-type format.

Table 30 (copied from RISC-V unprivileged specification, C extension instruction formats) provides the mapping between the 8 possible values of encoded register indices in the CA-type and the actual general purpose or floating-point registers, alongside the ABI register names.

Table 30. Registers specified by the three-bit rs1 ' , rs2 ' , and rd ' fields of the CIW, CL, CS, CA, and CB formats.

RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

Both previous examples of RV-C encoding illustrate that compressed instructions are less expressive than their base extension counterparts and use larger parts of the opcode space. This is the price to pay to ensure shorter code and is balanced by the fact that they are more heavily used than the extended versions providing an overall code size reduction.

5.3. Design of Compressed Decoder

The block diagram of the compressed decoder is shown in *Figure 25*.

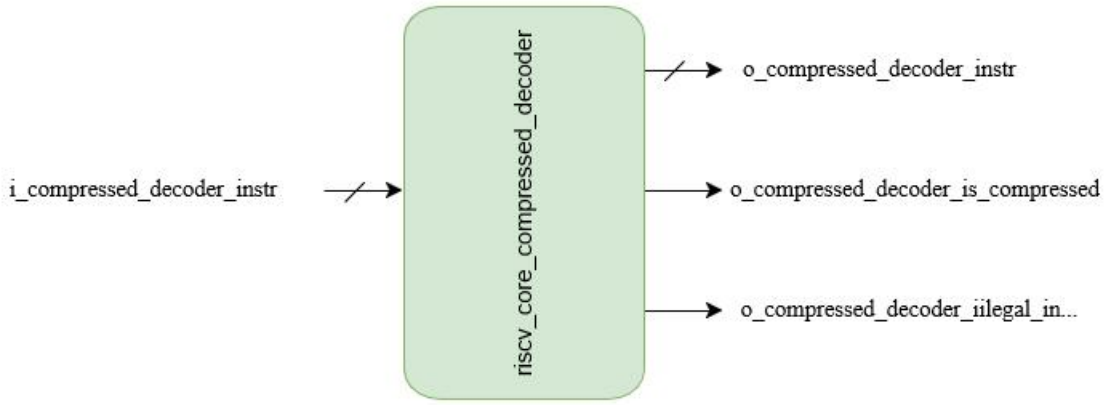

Figure 25. Compressed Decoder Block Diagram

The block interface of the compressed decoder is shown in *Table 31*.

Table 31. Block Interface of Compressed Decoder.

Port Name	Width	Description
i_compressed_decoder_instr	32	Compressed Instruction Input.
o_compressed_decoder_instr	32	Compressed Instruction Output.
o_compressed_decoder_is_compressed	1	Flag indicates that the input instruction is compressed.
o_compressed_decoder_illegal_instr	1	Flag indicates that the input instruction is illegal for hazard.

6. FIELD PROGRAMMABLE GATE-ARRAY (FPGA)

6.1. FPGA BOARD

During the initial research and evaluation process to identify FPGA boards suitable for implementing a RISC-V architecture, we selected FPGA boards: DE1-SoC. We evaluated the board's compatibility with RISC-V, processing power, FPGA resources, and integration features. Based on this analysis, we determined that the DE1-SoC board is the most viable choice for RISC-V implementation. The DE1-SoC's powerful Cyclone V FPGA, combined with its ARM Cortex-A9 processor, provides the ideal balance between hardware acceleration and software flexibility, making it the best option for efficiently executing RISC-V tasks and handling real-time processing.

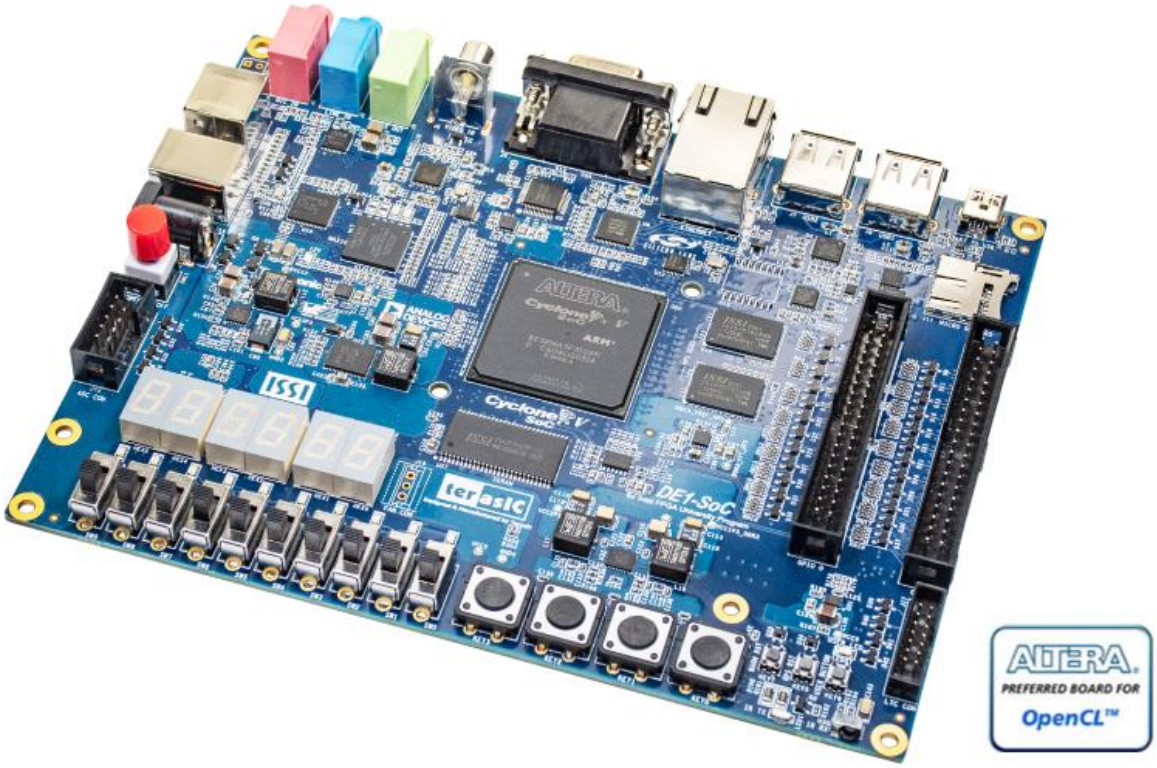

Figure 26: De1SOC Board

6.2. BASIC FEATURES OF DE1SOC

Cyclone V SoC Chip:

- Dual-core ARM Cortex-A9 processor (925 MHz).
- 85,000 logic elements and 112 DSP blocks.
- 4,450 kbit embedded memory (M10K blocks).
- 6 PLLs and integrated ADC with 4 channels (12-bit resolution).

Memory:

- 1GB DDR3 SDRAM on HPS with 32-bit bus.
- 64MB SDRAM on FPGA.

Power Supply:

- Powered via 12V DC adapter or USB Blaster connection for programming.

Ports:

- Gigabit Ethernet, USB 2.0 OTG, and Micro USB-UART bridge.
- HDMI output for video display.
- 40-pin GPIO header, SPI, I2C, and UART support.

Push buttons, LEDs, and switches:

- 4 push buttons, 10 switches, and 10 red LEDs for user interaction.

Expansion Port:

- GPIO header for peripheral integration.
- HPS-FPGA bridges (AXI, lightweight AXI) for high-speed communication.

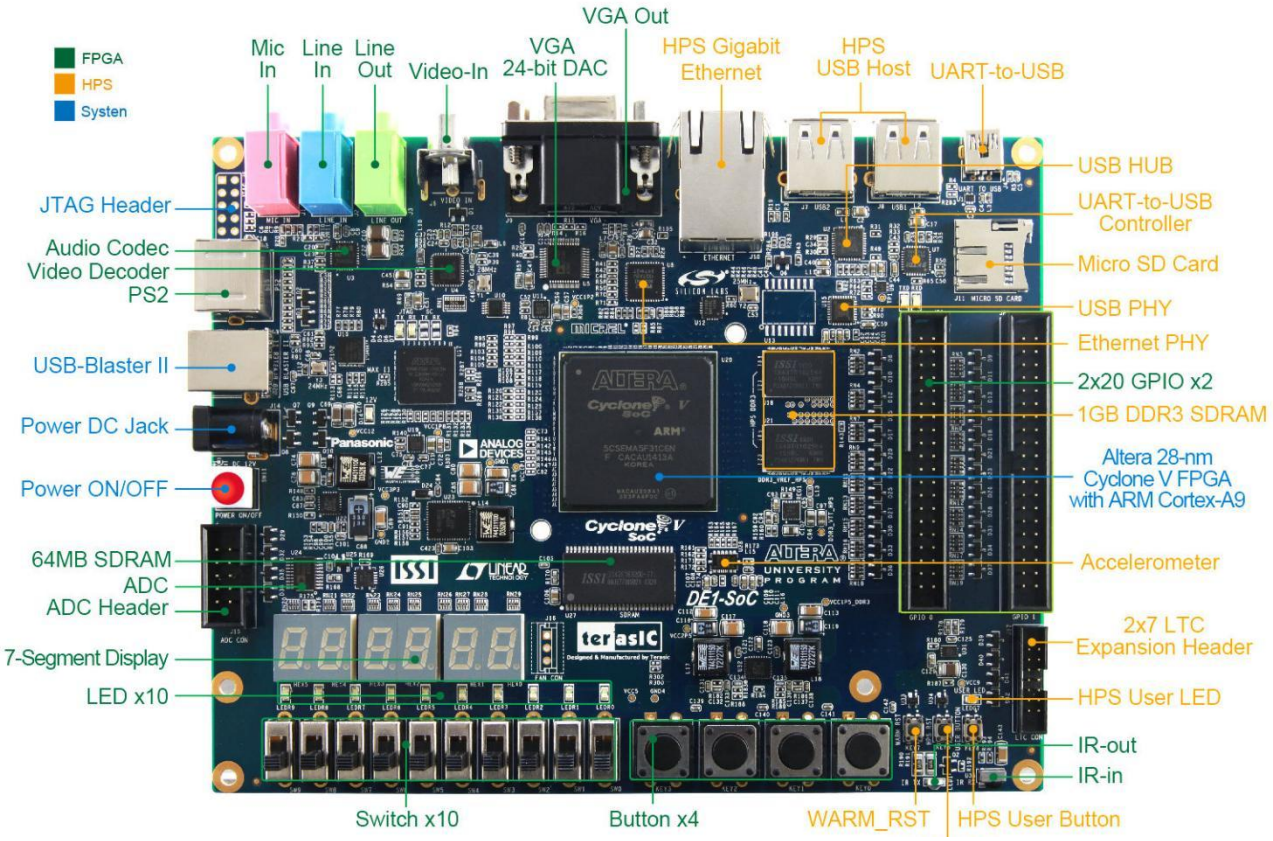

Figure 27: De1SOC Pin diagram

6.3. PROGRAMMABILITY ON DE1SOC

The DE1-SoC integrates an ARM Cortex-A9 dual-core processor (HPS) with Cyclone V FPGA, making it a versatile platform for hybrid software-hardware tasks. The HPS supports high-level languages like C/C++ and Python and can run operating systems like Linux, making it ideal for control logic, task scheduling, and lightweight processing. This flexibility allows the HPS to manage preprocessing, data coordination, and post-processing in applications such as CNN pipelines.

The FPGA fabric excels at accelerating parallel tasks like convolutions and matrix calculations. Developers can program the FPGA using Verilog, VHDL, or Quartus Prime, leveraging its 112 DSP blocks and 4,450 kbit embedded memory for computationally intensive operations. The FPGA can offload heavy workloads from the HPS, enabling efficient execution of CNN layers and other real-time algorithms.

The HPS-FPGA bridges (AXI and lightweight AXI) ensure high-speed communication between the two, allowing seamless data sharing and control. The shared memory architecture further supports efficient integration of software and hardware tasks. While more complex to develop compared to Python-based frameworks like PYNQ, DE1-SoC offers deeper hardware access and customization, making it a robust choice for RISC-V implementations, CNN acceleration, and embedded applications.

7. Waveform

7.1. Fetch Stage

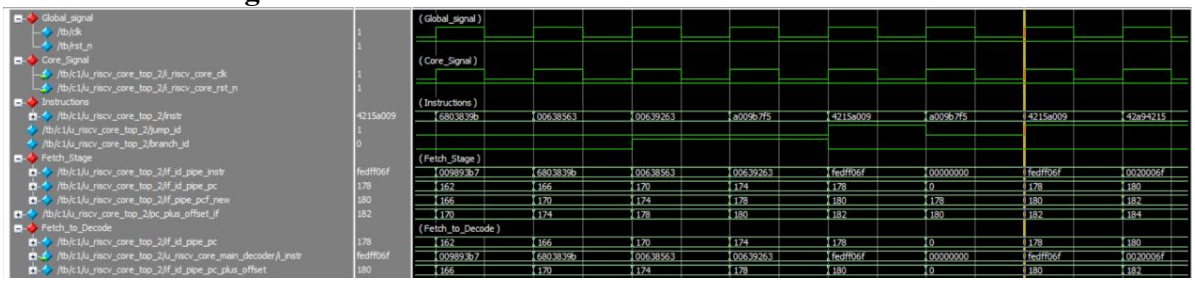

Figure 28: Fetch stage waveform

7.2. Decode Stage

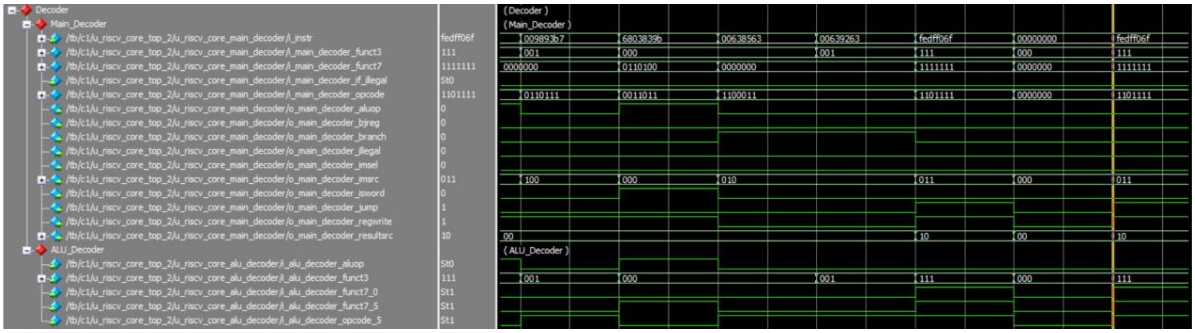

Figure 29: Decode stage waveform

7.3. Decode to Execute

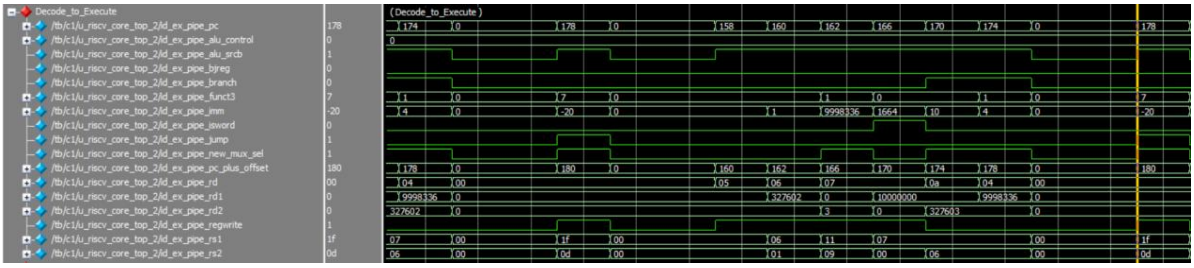

Figure 30: Decode to_Execute stage waveform

7.4. Execute Stage

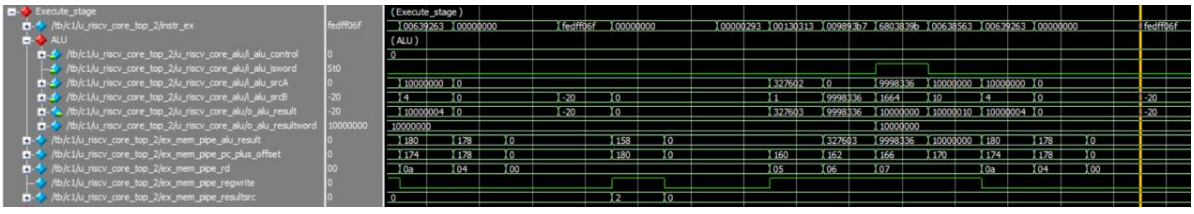

Figure 31: Execute stage waveform

7.5. ALU

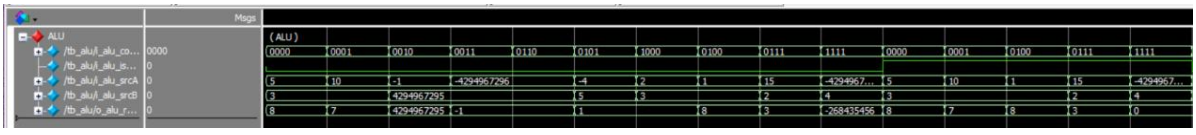

Figure 32: ALU waveform

7.6. Forwarding Block

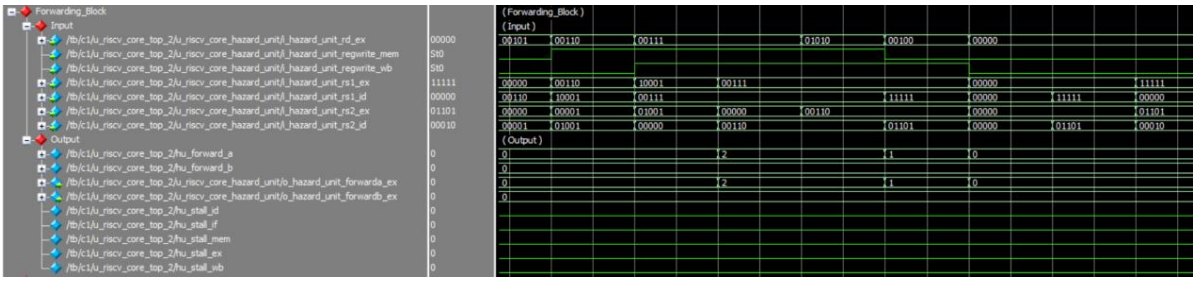

Figure 33: Forwarding Block waveform

7.7. Memory Stage

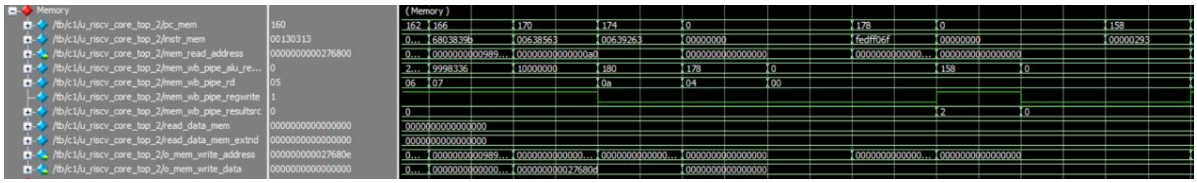

Figure 34: Memory Stage waveform

7.8. Wriback Stage

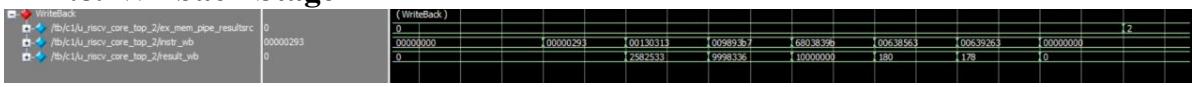

Figure 35: WriteBack Stage waveform

7.9. R-type.

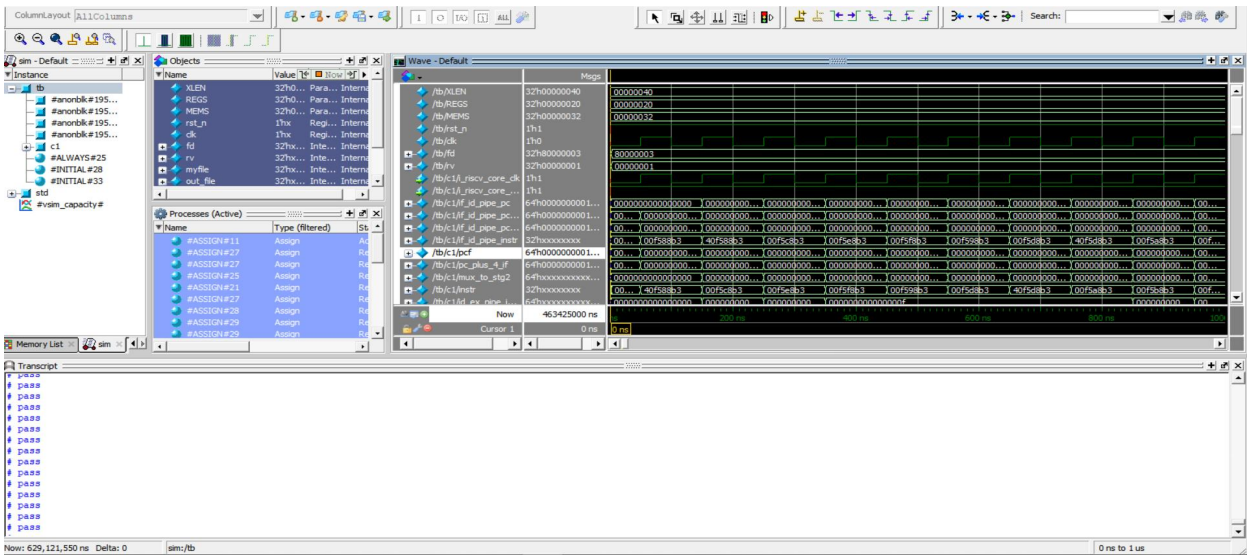

Figure 36: Waveform for R-type

Table 32: R-type instructions

Assembly	INSTR(HEX)	operands(HEX)	excepected
R-TYPE			
add x17 x11 x15	0x00f588b3	16+1E=34	x17 = 0x0000000000000034
sub x17 x11 x15	0x40f588b3	16 - 1E = fff8	x17 = 0xfffffffffffff8
xor x17 x11 x15	0x00f5c8b3	16 ^ 1E = 08	x17 = 0x0000000000000008
or x17 x11 x15	0x00f5e8b3	16 or 1E = 1E	x17 = 0x000000000000001e
and x17 x11 x15	0x00f5f8b3	16 and 1E = 16	x17 = 0x0000000000000016
sll x17 x11 x16	0x00f598b3	16 << 1E =0000000580000000	x17 = 0x0000000580000000
srl x17 x11 x16	0x00f5d8b3	16 >> 1E = 0	x17 = 0x0000000000000000
sra x17 x11 x16	0x40f5d8b3	16 >>>1E = 0	x17 = 0x0000000000000000
slt x17 x11 x16	0x00f5a8b3	16<1E? 1:0	x17 = 0x0000000000000001
sltu x17 x11 x16	0x00f5b8b3	unsigned(16<1E)?1:0	x17 = 0x0000000000000001
addw x17 x11 x15	0x00f588bb	16 + 1E = 34	x17 = 0x0000000000000034
subw x17 x11 x15	0x40f588bb	16 - 1E =fff8	x17 = 0xfffffffffffff8
sllw x17 x11 x15	0x00f598bb	16<<1E (word)	x17 = 0xffffffff80000000
srlw x17 x11 x15	0x00f5d8bb	16>>1E (word)	x17 = 0x0000000000000000
sraw x17 x11 x15	0x40f5d8bb	16>>>1E (word)	x17 = 0x0000000000000000

7.10. I-type.

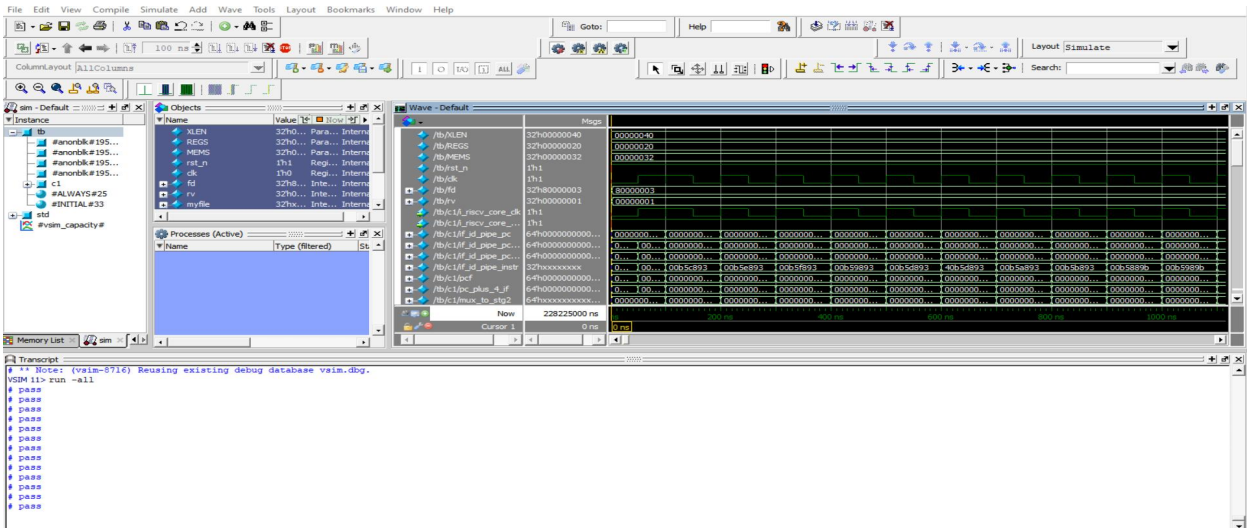

Figure 37: Waveform for I-type

Table 33: Instructions for I-type

I-TYPE			
addi x17,x11,11	0x00b58893	16+b = 21	x17 = 0x0000000000000021
xori x17,x11,11	0x00b5c893	16^b = 1d	x17 = 0x000000000000001d
ori x17,x11,11	0x00b5e893	16|b = 1f	x17 = 0x000000000000001f
andi x17,x11,11	0x00b5f893	16&b = 02	x17 = 0x0000000000000002
slli x17,x11,11	0x00b59893	16<<b = 0x0000b000	x17 = 0x000000000000b000
srli x17,x11,11	0x00b5d893	16>>b = 0X0	x17 = 0x0000000000000000
srai x17,x11,11	0x40b5d893	16>>>b = 0x0	x17 = 0x0000000000000000
slti x17,x11,11	0x00b5a893	16<b = 0	x17 = 0x0000000000000000
sltiu x17,x11,11	0x00b5b893	unsigned(16,b) = 0	x17 = 0x0000000000000000
addiw x17,x11,11	0x00b5889b	16+b = 21	x17 = 0x0000000000000021
slliw x17,x11,11	0x00b5989b	16<<b = 0b	x17 = 0x000000000000b000
srliw x17,x11,11	0x00b5d89b	16>>b = 0	x17 = 0x0000000000000000
sraiw x17,x11,11	0x40b5d89b	16>>>b = 0	x17 = 0x0000000000000000

7.11. S-type.

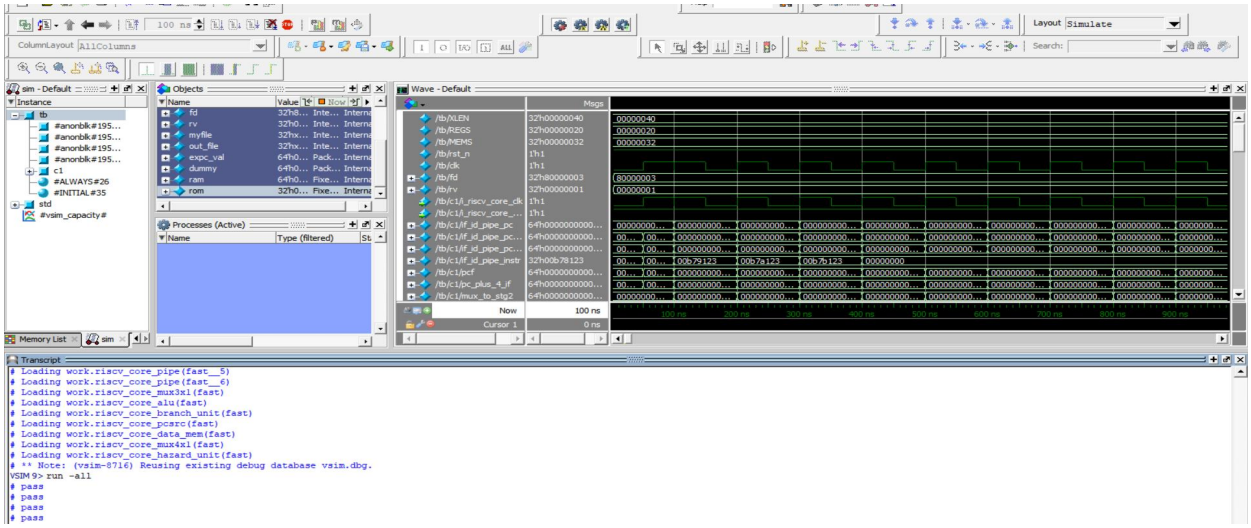

Figure 38: Waveform for S-type

7.12. U-type.

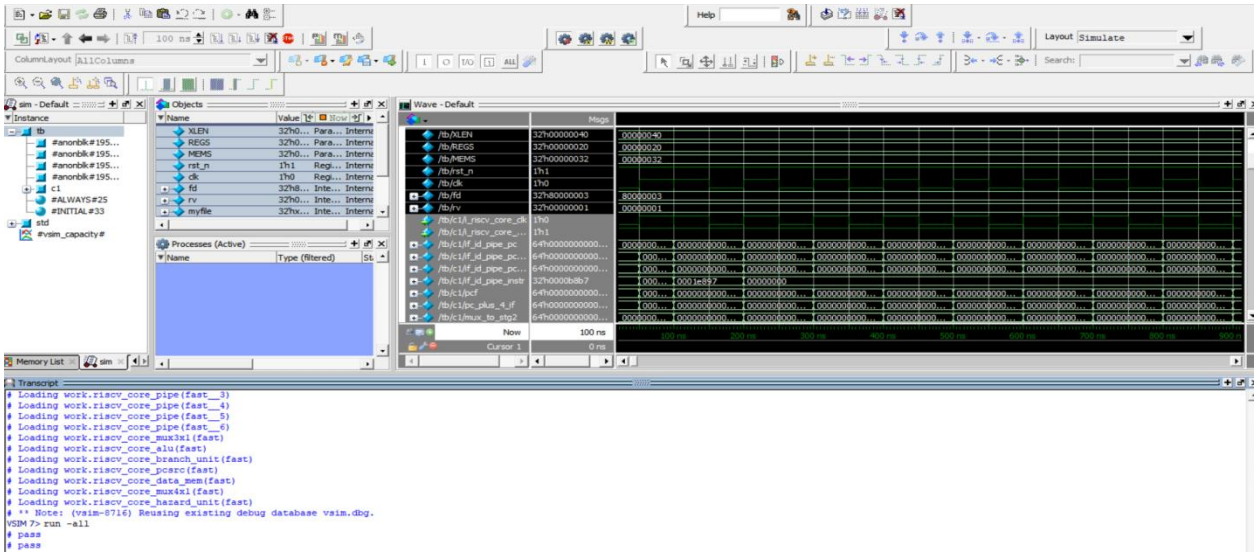

Figure 39: Waveform for U-type

Table 34: Instructions for Store/Load/U-type

Store			
sb x11 2 x15	0x00b78123	mem[x15 + 2] = x11	mem[x20] = 0x16
sh x11 2 x15	0x00b79123	mem[x15 + 2] = x11	mem[x20] = 0x16 mem[x21] = 0x00
sw x11 2 x15	0x00b7a123	mem[x15 + 2] = x11	mem[x20] = 0x16 mem[x21] = 0x00
sd x11 2 x15	0x00b7b123	mem[x15 + 2] = x11	mem[x20] = 0x16 mem[x21] = 0x00
Load			
lb x17 2 x15	0x00278883	x17 = mem[x15+2]	x17 = 0x0000000000000020
lh x17 2 x15	0x00279883	x17 = mem[x15+2:x15+3]	x17 = 0x0000000000002120
lw x17 2 x15	0x0027a883	x17 = mem[x15+2:x15+5]	x17 = 0x0000000023222120
ld x17 2 x15	0x0027b883	x17 = mem[x15+2:x15+9]	x17 = 0x2726252423222120
lbu x17 2 x15	0x0027c883	x17 = mem[x15+2]	x17 = 0x0000000000000020
lhu x17 2 x15	0x0027d883	x17 = mem[x15+2:x15+3]	x17 = 0x0000000000002120
lwu x17 2 x15	0x0027e883	x17 = mem[x15+2:x15+5]	x17 = 0x0000000023222120

8. FPGA Implementation and Application

8.1. FPGA Implementation

8.1.1. Synthesis Reports

8.1.1.1. Timing Report

The timing report for our synthesized core is shown in *Figure 35*.

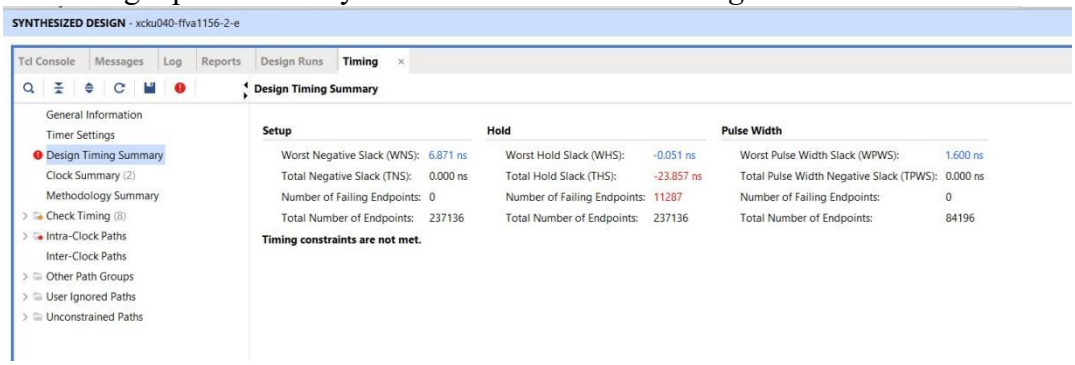

Figure 40. Synthesis Timing Report.

The timing report shows that we have a large setup slack so we can increase the clock frequency.

The hold has negative slack, but it will be fixed after implementation and taking wiring delay into consideration.

8.1.1.2. Power Report

The power report is shown in *Figure 36.*

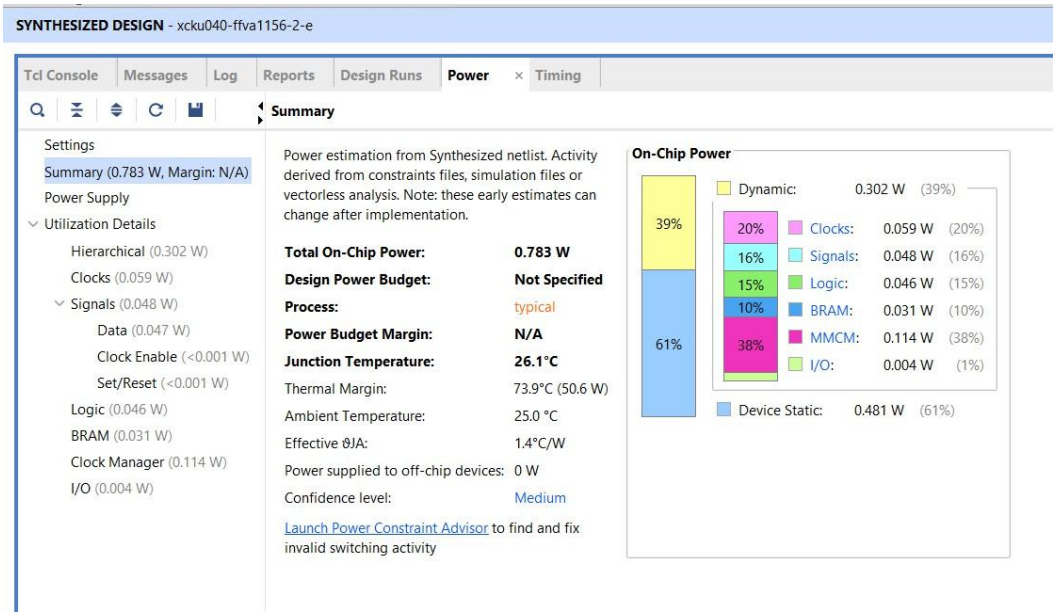

Figure 41. Synthesis Power Report.

8.1.1.3. Utilization Report

The utilization report is shown in *Figure 37*

Summary

Resource	Utilization	Available	Utilization %
LUT	178237	242400	73.53
FF	84128	484800	17.35
BRAM	32	600	5.33
IO	11	520	2.12
BUFG	1	480	0.21
MMCM	1	10	10.00

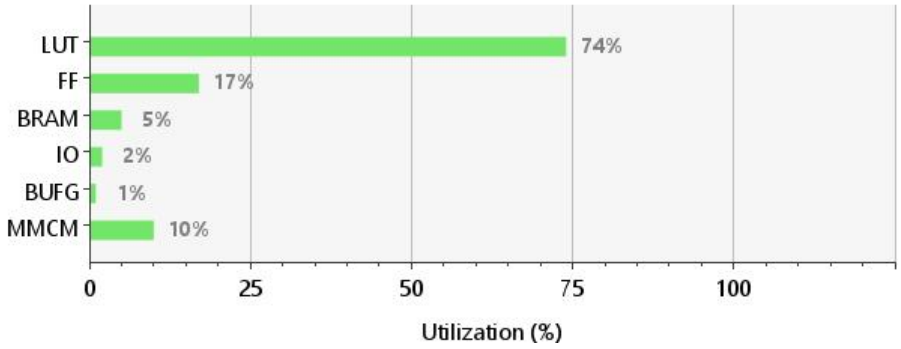

Figure 42. Synthesis Utilization Report.

8.1.2. Implementation Reports

8.1.2.1. Timing Report

The timing report is shown in *Figure 38*.

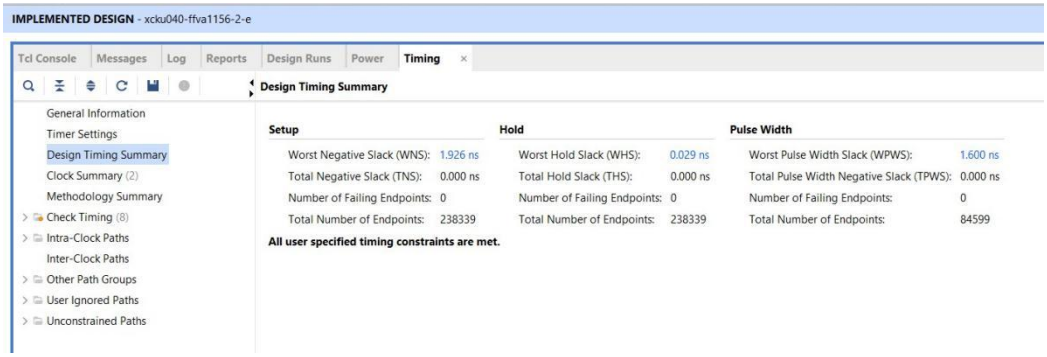

Figure 43. Implementation Timing Report.

8.1.2.2. Power Report

The power report is shown in *Figure 39*.

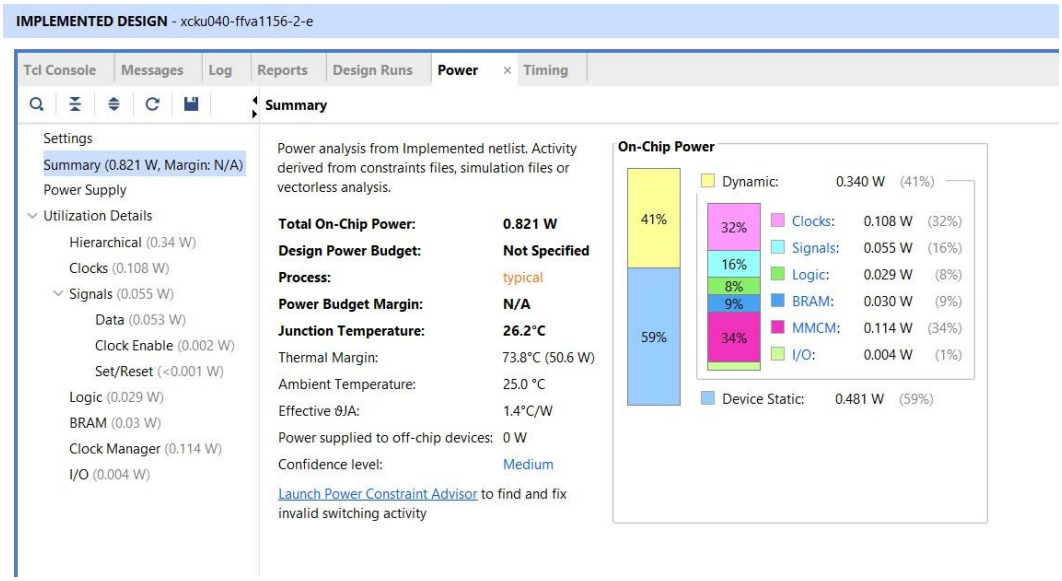

Figure 44. Implementation Power Report.

8.1.2.3. Utilization Report

The utilization report is shown in *Figure 43*.

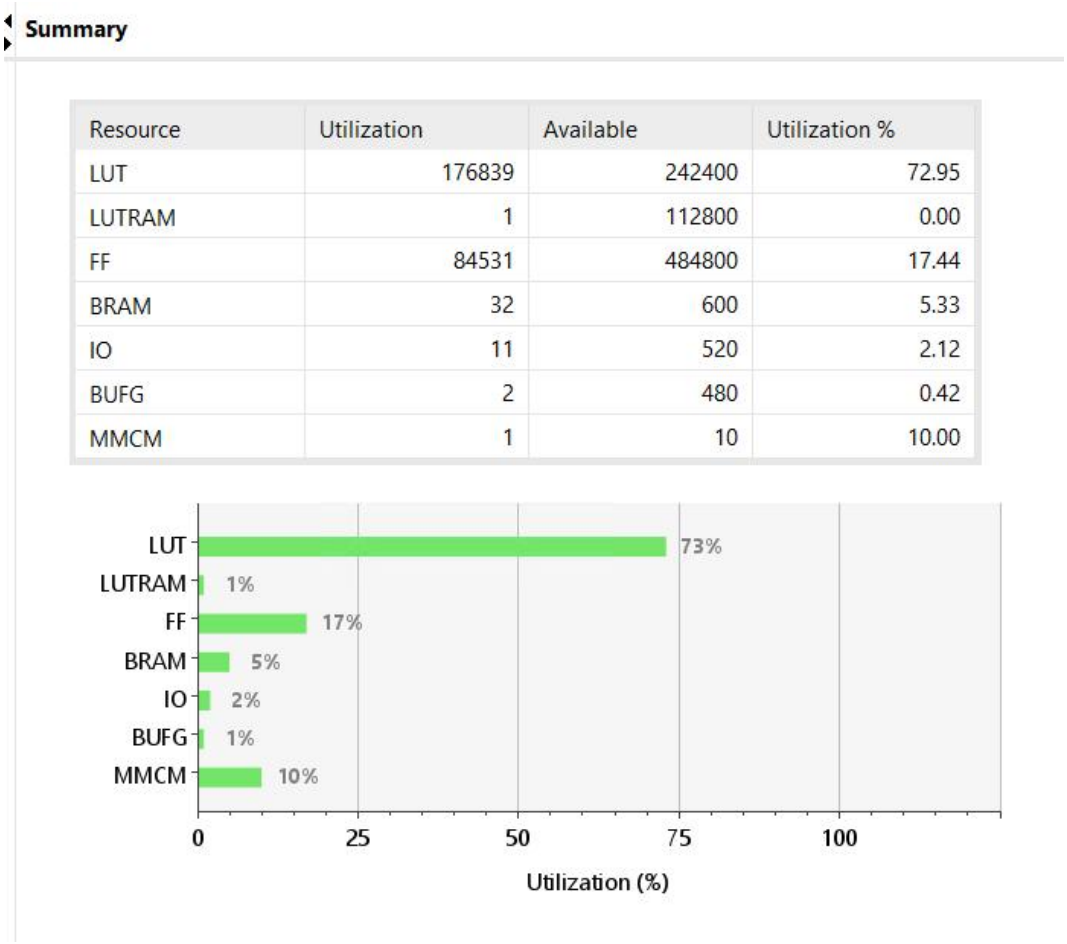

Figure 45. Implementation Utilization Report.

8.1.2.4. Noise Margin Report

The noise-margin report is shown in *Figure 41*.

I/O Bank Details

Name	Port	I/O Std	Vcco	Slew	Drive Strength (...	OUTP...	Remaining Margin...	PRE_EMPH...	LVDS_PRE_EMPH...	Off-Chip Termina...	Notes
I/O Bank 0 (0)											
I/O Bank 44 (0)											
I/O Bank 45 (0)											
I/O Bank 46 (0)											
I/O Bank 47 (0)											
I/O Bank 48 (0)											
I/O Bank 64 (0)											
I/O Bank 65 (7)		LVCMOS18	1.80	SLOW	12					FP_VTT_50	
P20	for_leds[0]	LVCMOS18	1.80	SLOW	12		78.77			FP_VTT_50	
P21	for_leds[1]	LVCMOS18	1.80	SLOW	12		76.35			FP_VTT_50	
N22	for_leds[2]	LVCMOS18	1.80	SLOW	12		62.76			FP_VTT_50	
M22	for_leds[3]	LVCMOS18	1.80	SLOW	12		66.59			FP_VTT_50	
R23	for_leds[4]	LVCMOS18	1.80	SLOW	12		74.49			FP_VTT_50	
P23	for_leds[5]	LVCMOS18	1.80	SLOW	12		74.99			FP_VTT_50	
H23	for_leds[6]	LVCMOS18	1.80	SLOW	12		98.05			FP_VTT_50	
I/O Bank 66 (0)											
I/O Bank 67 (0)											
I/O Bank 68 (0)											

Figure 46. Noise Margin Report.

8.2. Application

8.2.1. Application 1: Sending Characters using UART

The Block Diagram of our system is shown in *Figure 42*.

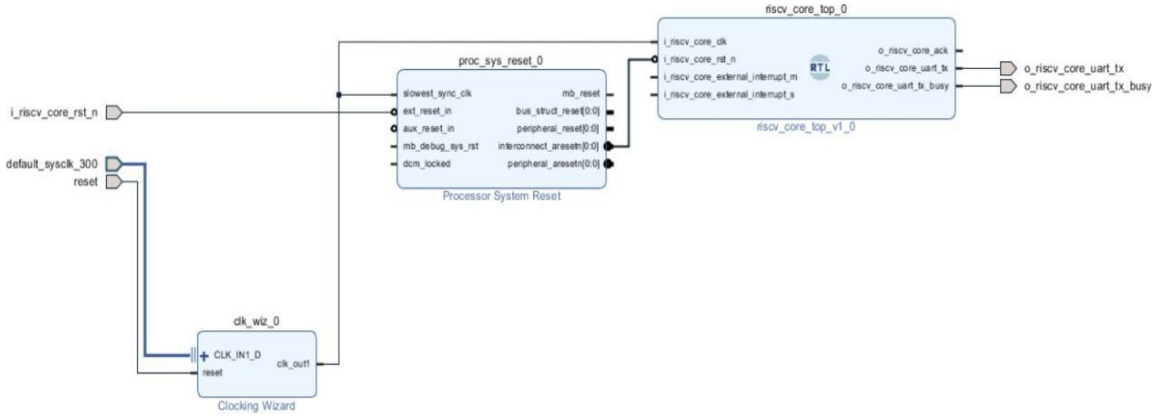

Figure 47. System Block Diagram for App 1.

This application is to send characters from UART Tx port to PC serial port and read the sent characters by TTY.

The program is written with RISC-V assembly then compiled by assembler then converted to hex file then the hex file is loaded into the main memory that connected to caches. This program is shown in Figure 80.

```
.include "user_define.h"
.globl _start
.section .text
_start:
        addi x10,x0 , 0x41
        li x11, 0x10000000
        sb x10, (x11) #'A'
        li x9 , 0x1
        addi x10,x0 ,0x4c
        sb x10,(x11) #'L'
        li x9 , 0x2
        addi x10,x0 ,0x45
        sb x10,(x11) #'E'
        li x9 , 0x3
        addi x10,x0 ,0x58
        sb x10,(x11) #'X'
        li x9 , 0x4
        addi x10,x0 ,0x52
        sb x10,(x11) #'R'
        li x9 , 0x5
        addi x10,x0 ,0x56
        sb x10,(x11) #'V'
        li x9 , 0x6
        addi x10,x0 ,0x41
        sb x10,(x11) #'A'
        li x9 , 0x7
        addi x10,x0 ,0x52
        sb x10,(x11) #'R'
        li x9 , 0x3
        j _start
```

Figure 43. Assembly Program that Sending Characters using UART.

Note: if random characters appear on the screen that means there's a timing mis-synchronization between UART module baud rate and the baud rate on the serial port.

8.2.2. Application 2: LEDs Testing

The Block Diagram of our system is shown in *Figure 44.*

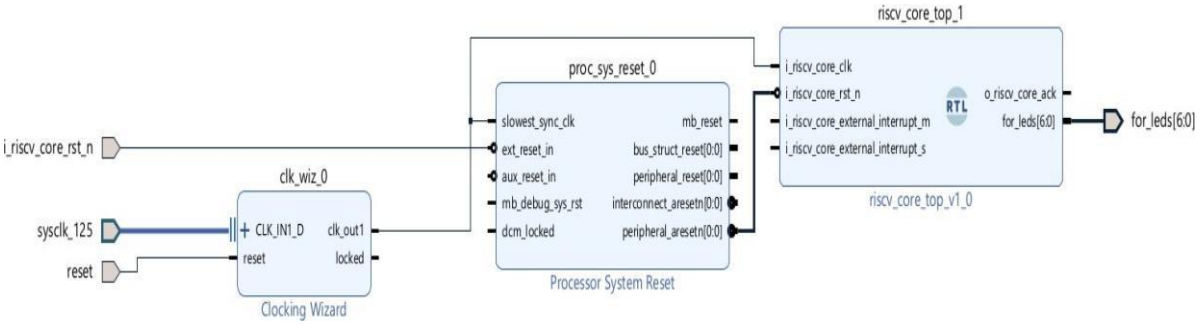

Figure 48. System Block Diagram for App 2.

This application runs a few tests on RISC-V operations while turning a LED on is the indication to the success of specific operations. These operations are as follows:

- Integer (*add*, *or*, *xor* and *slt*).
- Multiplication.
- Compressed.
- Atomic.

The application allows the LEDs to have an approximate of one second delay between the tests to indicate the passing or failing of each operation.

The program is written with RISC-V assembly then compiled by assembler then converted to hex file then the hex file is loaded into the main memory that connected to caches. This program is shown below:

```
.include "user_define.h"
.global _start
.section .text
_start:
li x5, 0x5
addi x6 , x5 , 0x6 li x7, 0xb
beq x7 , x6 , PASSc1 bne x7 , x6 , FAIL
PASSc1:
ori x9 , x9 , 0x1 j _startc2
_startc2:
li x5, 0x5 addi x6, x5 , 0 li x7, 0x5
beq x7 , x6 , PASSc2 bne x7 , x6 , FAIL
PASSc2:
j _startc3
_startc3:
li x5, 0x5
xori x6, x5 , 0x6 li x7, 0x3
beq x7 , x6 , PASSc3 bne x7 , x6 , FAIL
PASSc3:
j _startc4
_startc4:
li x5, 0x5
ori x6 , x5 , 0x6 li x7, 0x7
beq x7 , x6 , PASSc4 bne x7 , x6 , FAIL
PASSc4:
j _startc5
_startc5:
li x5, 0x5
slti x6 , x5 , 0x5 li x7, 0
beq x7 , x6 , PASSc5 bne x7 , x6 , FAIL
PASSc5:
li x6 , 0x0 j Delay
FAIL:
andi x9 , x9 , -2 li x6 , 0x0
j Delay Delay:
li x5, 0 addi x6, x6,0x1 li x7, 0x989680
beq x7 ,x6 ,DONE bne x7 ,x6 ,WAIT
WAIT:
j Delay DONE:
j m_start m_start:
li x4, 0x5 li x5, 0x2
mul x6, x5,x4 li x7 , 0xa
beq x7 , x6 , PASSc51 bne x7 , x6 , FAIL_m
PASSc51:
ori x9 , x9 , 0x2 li x6 , 0x0
j Delay_1 FAIL_m:
andi x9 , x9 ,-3 li x6 , 0x0
j Delay_1 Delay_1:
```

```
li x5, 0
addi x6, x6,0x1 li x7, 0x989680
beq x7 ,x6 ,DONE_1 bne x7 ,x6 ,WAIT_1
WAIT_1:
j Delay_1 DONE_1:
j compress
compress:
c.li x4, 5
c.li x5, 10 c.add x5, x4 c.li x7, 15
beq x7, x5 ,PASSc6 bne x7, x5 ,FAIL_c
PASSc6:
FAIL_c:
ori x9 ,x9 ,0x4 li x6 , 0x0
j Delay_2
andi x9, x9, -5 li x6 , 0x0
j Delay_2
Delay_2:
li x5, 0
addi x6, x6,0x1
li x7, 0x989680
beq x7 ,x6 ,DONE_2
bne x7 ,x6 ,WAIT_2 WAIT_2:
j Delay_2
DONE_2:
j atomic atomic:
addi x4, x0, 5
addi x5, x0, 7
addi x6, x0, 0
addi x7, x0, 8
addi x8, x0, 16
addi x10, x0, 32
addi x11, x0, 40
addi x12, x0, 48
addi x13, x0, 56
addi x14, x0, 64
addi x15, x0, 72
addi x16, x0, 15
addi x17, x0, 31
addi x18, x0, 50
addi x19, x0, 80
addi x20, x0, 99
addi x21, x0, 64
sd x4 , 0(x6)
sd x5 , 0(x7)
sd x16 , 0(x8)
sd x18 , 0(x10)
amoadd.w x23, x5, (x6) # x23 <= 5 , MEM[x6] <= 12 ld
x23 , 0(x6)
li x7 , 12
beq x7, x23, PASSc7 bne x7, x23, FAIL_a
PASSc7:
ori x9,x9,0x8
j FINISH
FAIL_a:
```

```
andi x9, x9 , -9
j FINISH
FINISH:
j FINISH
```

9. Conclusion

In conclusion, we have demonstrated the development of a RISC-V core with the IMC extensions and an integrated cache system capable of running Linux-OS. The project illustrates the potential and versatility of RISC-V architecture and its capability in meeting modern computing environment while maintaining efficient performance.

The implementation of the IMC extensions has enhanced the core capabilities and functionality making it capable of handling high-speed mathematical computations and atomic memory operations. Level-1 cache has been crucial for improving the throughput and reducing the latency of memory access ensuring high efficiency. Privileged architecture has been important also for supporting Linux-OS and handling exceptions and interrupts.

Through testing and verification, the developed RISC-V core has demonstrated the ability to achieve the desired functionality. The successful implementation of this project provides a foundation for future research and development, where more extensions, enhanced performance, and integration into larger systems can be explored.

This project shows a significant contribution to the field of computer architecture by showcasing the capability of utilizing RISC-V cores for complex systems.

In summary, the project's outcomes emphasize the viability of RISC-V as a competitive alternative in the realm of modern computing, encouraging further exploration and adoption in diverse applications.

10. Future Work

As our target is to implement a high performance RV64IMC with privilege modes to make our core capable to run a Linux-based OS, we aim to:

1. Interface with DDR (Double Data Rate) Memory:

- To run a Linux-based OS, interfacing our core with DDR external memory.
- Design an interface that allows our core to read from and write to DDR memory.

2. Implement Virtual Memory and Memory Management Unit (MMU):

- Implementing virtual memory is critical for OS support. It enables processes to have separate address spaces while sharing physical memory.
- Develop an MMU that translates virtual addresses to physical addresses. Explore page tables, TLBs, and efficient address translation techniques.

3. Implement Page Table Walker:

- The Page Table Walker (PTW) efficiently translates virtual addresses to physical addresses.
- Investigate different PTW designs (e.g., multi-level page tables, hashed page tables) and choose the most suitable one for our core.

4. PLIC:

The addition of the PLIC will enhance the core's capability to manage interrupts efficiently, allowing for more efficient handling of multiple interrupt sources. This improvement will be critical for running a Linux-OS, which requires complex interrupt management to ensure smooth operation and. The integration of the PLIC will not only improve the performance and reliability of the system but also expand its applicability in more complex and demanding computing environments.

We also aim to:

1. Enhance our Multiplier and Divider Performance:

Implement a Radix-16 modified Booth multiplication which reduces partial product generation during multiplication.

2. Create our Testing Environment.

Verification of our RISC-V core using a Universal Verification Methodology (UVM) environment will facilitate the creation of reusable and scalable verification environment to achieve comprehensive coverage and achieve correctness.

5. AI Application: CNN for Face Recognition.

- Develop and implement a Convolutional Neural Network (CNN) optimized for face recognition tasks on embedded systems.
- Design and train a CNN model for accurate and efficient face recognition.
- Optimize the model for deployment on resource-constrained systems, such as RISC-V or FPGA-based platforms.
- Face Detection and Recognition: Utilize the CNN model to identify and authenticate faces in real-time.
- Efficiency: Implement quantization and pruning techniques to reduce the computational complexity and memory usage of the CNN.
- Combine the CNN face recognition system with the RISC-V core for real-time applications, such as access control, surveillance, and IoT devices.

- Explore hardware acceleration techniques, such as using FPGA or specialized RISC-V instructions, to enhance the CNN's performance.

11. DISTRIBUTION OF CREDIT

1) **Đỗ Mạnh Dũng – 21119301** (40/100%) - Block diagrams (33%) -Implementation (80%)
   - Block diagram - Find the best Block Diagram that is suitable for the project topic. Then learn and explore the theory to find the methods and tools for code and simulation.
   - Research RV32M architecture (100%); Research RV32I architecture (33%)
   - Implemetation
   - Write report and analysing (33%)

2) **Trần Long - 21119311** (30/100%)
   - Research RV32C architecture (50%)
   - Research RV32I architecture (33%)
   - Write report and analysing (33%)
   - Investigate the architecture on vivado (50%)

3) **Đặng Trung Nghĩa – 21119313** (30/100%)
   - Research RV32C architecture (50%)
   - Research RV32I architecture (33%)
   - Write report and analysing (33%)
   - Investigate the architecture on vivado (50%)

Distribution table

Tasks	Members	Completed(%)
Block diagram	Dũng	100
Research RV32M architecture	Dũng	100
Research RV32C architecture (50%)	Long, Nghĩa	100
Research RV32I architecture	Long, Nghĩa, Dũng	100
Simulation	Dũng	100
Implemetation	Dũng	80
Investigation	Long, Nghĩa	100
Write report and analysing	Dũng, Long, Nghĩa	100

12. REFERENCES

[1] Novel Casestudy and Benchmarking of AlexNet for Edge AI: From CPU and GPU to FPGA Firas Al-Ali Manukau Institute of Technology Auckland, New Zealand Firas.Al-Ali@manukau.ac.nz

[2] Design and Evaluation of CPU-, GPU-, and FPGA-Based Deployment of a CNNforMotor Imagery Classification in Brain-Computer Interfaces Federico Pacini 1,* , Tommaso Pacini 1 1 , Giuseppe Lai 2 , Alessandro Michele Zocco 1 and Luca Fanucci 1

[3] Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks Roberto DiCecco* , Griffin Lacey†, Jasmina Vasiljevic* , Paul Chow* , Graham Taylor† and Shawki Areibi† * University of Toronto, Department of Electrical and Computer Engineering, Ontario, Canada E-mail:{dicecco1, vasiljev, pc}@eecg.toronto.edu †University of Guelph, Ontario, Canada E-mail:{laceyg, gwtaylor, sareibi}@uoguelph.ca

[4] Waterman, A. S. (2016). Design of the RISC-V instruction set architecture (Doctoral dissertation). University of California, Berkeley.

[5] Sawant, T., Korgaonkar, N., Sherlekar, V., Bhise, G., & Thakur, B. (2019). Design and Implementation of 32-Bit RISC Processor Using Verilog HDL. Journal of Emerging Technologies and Innovative Research (JETIR), 6(5), 391-398. Retrieved from JETIR .

[6] Chang, Y., Liu, Y., Peng, C., Guo, J., & Zhao, Y. (2024). Design of a Configurable Five-Stage Pipeline Processor Core Based on RV32IM. Electronics, 13(1), 120.

[7] Poli, L., Saha, S., Mcdonald-Maier, K. D., & Zhai, X. (2021). Design and Implementation of a RISC-V Processor on FPGA. In 2021 17th International Conference on Mobility, Sensing and Networking (MSN) (pp. 161-163). IEEE.

[8] Mikhail Chupilko, Alexander Kamkin, Alexandr S. Protsenko. Open-source Validation Suite for RISC-V. 2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV).