DESIGN OF SINGLE CYCLE 32-BIT RISC-V PROCESSOR

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY

in

VLSI AND EMBEDDED SYSTEMS

by

Kartike

(2K24/VLS/24)

Under the Supervision of

Dr. Ajai Gautam

(Associate Professor)

DTU, Delhi

Department of Electronics and Communication Engineering

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi-110042

June, 2026

Department of Electronics and Communication Engineering
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

I, Kartike, Roll No. 2K24/VLS/24 hereby certify that the work which is being presented in the thesis entitled ***'Design of Single Cycle 32-bit RISC V Processor'*** in partial fulfilment of the requirements for the award of the Degree of Master of Technology (VLSI and Embedded Systems), submitted in the Department of Electronics and Communication Engineering, Delhi Technological University, is an authentic record of my own work carried out during the period from January 2025 to May 2026 under the supervision of **Dr. Ajai Kumar Gautam**.

I have not submitted the matter presented in the thesis for the award of any other degree of this or any other Institute.

Candidate's Signature

Kartike
(2k24/VLS/24)

Department of Electronics and Communication Engineering
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Bawana Road, Delhi-110042

CERTIFICATE BY THE SUPERVISOR

This is to certify that the work reported in this thesis entitled *"**Design of Single Cycle 32-bit RISC V Processor**"* has been carried out by Mr. Kartike (2K24/VLS/24) under my supervision.

Supervisor's Signature
Dr. Ajay Gautam
Assosiate Professor
ECE, DTU, Delhi

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Bawana Road, Delhi-110042

PLAGIARISM VERIFICATION

Title of the Thesis: **Design of Single Cycle 32-bit RISC V Processor**

Total Pages: _____

Name of the Scholar: **Kartike**

Supervisor: **Dr. Ajai Gautam**

Department of Electronics and Communication Engineering

This is to report that the above thesis was scanned for similarity detection. The process and outcome is given below:

Software used: **Turnitin** Similarity Index: ____% Total Word Count: _____

Date: _____

Candidate's Signature **Signature of Supervisor(s)**

ACKNOWLEDGEMENT

The successful completion of any task is incomplete and meaningless without giving any due credit to the people who made it possible without which the project would not have been successful and would have existed in theory. First and foremost, I am grateful to Prof. Neeta Pandey, H.O.D., Department of Electronics and Communication Engineering, Delhi Technological University, and all other faculty members of our department for their constant guidance and support, constant motivation and sincere support and gratitude for this project work. I would owe a lot of thanks to our supervisor, Dr. Ajai Gautam, Associate Professor, Department of Electronics and Communication Engineering, Delhi Technological University for igniting and constantly motivating us and guiding us in the idea of a creatively and amazingly performed Major Project 1 in undertaking this endeavour and challenge and also for being there whenever I needed his guidance or assistance.

I would also like to take this moment to show our thanks and gratitude to one and all, who indirectly or directly have given us their hand in this challenging task. I feel happy and joyful and content in expressing our vote of thanks to all those who have helped and guided me in presenting this project work for my Major project. Last, but never least, I thank our well-wishers and parents for always being with us, in every sense and constantly supporting me in every possible sense whenever possible.

Date:
Place: Delhi

Kartike
2K24/VLS/24

ABSTRACT

This project focuses on the design and implementation of single cycle risc v processor. This project focuses on the design, simulation, and synthesis of a Single-Cycle RISC-V Processor using Verilog Hardware Description Language (HDL). The processor is built around the RV32I base integer instruction set, which offers a minimalist yet powerful set of instructions suitable for educational, experimental, and lightweight embedded systems. By implementing the processor in a single-cycle format, all major operations are completed in one clock cycle, simplifying the control logic and providing a clear and effective understanding of the instruction execution process.

The architecture follows a modular and systematic approach, comprising five primary stages of instruction execution: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each module is designed to operate synchronously within a single clock cycle, enabling fast and predictable execution. This approach highlights the fundamental working of a processor pipeline in its most basic form, making it ideal for understanding the internal working of modern processors.

The design has been developed and simulated using Verilog testbenches to verify the correct execution of various RISC-V instructions, including arithmetic, logical, load/store, and branch operations. Simulation results confirm that the processor accurately handles instruction flow and data processing in accordance with the RISC-V ISA specifications.

In addition to the 32-bit single-cycle processor, a 64-bit RISC-V processor architecture was also implemented to extend the computational capability of the design. The 64-bit core supports a wider data path and enhanced processing performance while integrating Level-1 (L1) Instruction Cache and Level-1 Data Cache within the architecture. The inclusion of L1 caches significantly improves memory access efficiency by reducing memory principle latency and enabling faster instruction and data retrieval. The cache subsystem was integrated with the processor datapath and memory interface to optimize overall system performance

Overall, the project demonstrates the complete development flow of RISC-V processor design, including architecture planning, Verilog implementation, simulation, and hardware verification and Synthesis. The work provides a strong understanding of processor internals, cache integration, and RISC-V architecture with ASIC flow implementation.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

In this chapter, a short description of the fundamentals and background of the topics which are part of this thesis is presented. Furthermore, the purpose of writing this thesis is discussed, along with a map of this thesis document.

1.1. Motivation

With the intent in mind of implementing a core with an instruction set of a 64-bit RISC-V IMAC core with privilege modes and level 1 caches was made such that, RISC-V coming from hearts of Berkeley's Parallel Computing lab in 2010 to the commercial adoption including IoT, embedded systems, and high-performance computing in the present and to a future where it would further expand into markets such as automotive, aerospace, and telecommunications while standardize additional extensions and improve compatibility and a perk of it being an open-source hardware as that it would provide higher freedom for the programmers that enables them to have maximum access to the advantages of hardware feature while strive for a software solution.

A 64-bit architecture would be suitable for Enhanced Software Capabilities as many modern software applications including operating systems and databases and it is optimized for taking advantage of the larger address space and improved performance it also supports more advanced security features such as larger cryptographic keys which improves the security of data and communications.

An IMAC core as that would be responsible for:

I: Provides basic instructions for general-purpose computing.

M: Enhances performance for numerical and computational tasks.

A: Enables safe and efficient handling of concurrent processes.

C: Improves memory efficiency and performance through reduced code size.

And finally, A Level 1 Data Caches and Instruction Caches and the memory arbiter with a native interface to connect both caches with the Main Memory of our System .

1.2. History

The idea of this chip was started by some scholars in Berkeley University in 2010 as an alternative to ARM and x86 architectures. RISC-V has many versions which can be used for more advanced computations and also educational purposes. It was developed as open source hardware design. The first two generations were used to develop the SPARC processor, but this processor belongs to the fifth generation. The ISA architecture was launched in 2011, whereas RISC-V foundation was formed by the end of 2015.

1.3. Thesis Objectives

The main purpose is to develop the processor core based on SV & Verilog with cached memory and privileged mode support. The particular aims of this work are the following:

1. Developing RISC-V core:
    - Developing this processor core according to the RISC-V ISA requirements.
    - Implementing key components: instruction fetch unit, decoder, executive unit, memory access unit, write-back unit.
2. Incorporate Cached Memory:
    - Design and implement a cache memory system that helps enhance the performance of the processor.
    - Implement an instruction and a data cache along with cache coherence and cache consistency algorithms.
    - Experiment with various cache implementations to identify the best cache architecture for our processor.
3. Verification and Testing:
    - Come up with a verification plan for checking whether the function and performance of the designed RISC-V core processor are correct.
    - Do verification through simulation and test benches using hardware description language (HDL).
    - Analyze the performance of the implemented cached memory and privileged mode functionalities.
4. Documentation and Analysis:
    - Conduct an analysis of the design process that covers such issues as architecture and design decisions.

- Conduct a review of the results from verification and performance tests.

With such accomplishment, the goal of the dissertation is to contribute towards the development of RISC-V processors through incorporating cache memories and implementing the privileged modes using Verilog and System Verilog Hardware Description Languages.

The rising usage of consumer electronics devices worldwide has led to a massive development in the global semiconductor market. However, it has become more difficult to create quicker, smaller, and more complicated devices due to the difficulties in overcoming the lack of chips and manufacturing constraints, in addition to intense rivalry.

My project's goal is to fully implement the RISC-V core digitally by working through the whole ASIC physical design, from RTL to GDSII Implementation Flow. I covered every stage of the design cycle, which included placement and routing (PnR), floorplanning, synthesis, and static time analysis (STA).

1.4. Overview of RISC-V

RISC-V, which is also called “risk five,” is an ISA that follows the principle of RISC. Unlike other ISAs, the ISA of RISC-V comes with open source licensing. This implies that anyone can manufacture, sell, and design RISC-V chips and software.

Some important things about RISC-V are:

1. Open source: Unlike other Instruction Set Architecture it comes with open-source licensing that promotes innovation and creativity. In contrast to the restrictions of proprietary licenses, the open-source nature of its ISA ensures freedom in innovation.
2. Simple and Flexible: It is simple and flexible, enabling its usage across various platforms, ranging from simple embedded devices to complex supercomputers.
3. Multimodular Approach: With its modular architecture, RISC-V supports extensibility; it enables certain features to be added to the ISA according to requirement.
4. Supporting Wide Applications: As the architecture allows extension through adding custom instructions and features, RISC-V is quite scalable.
5. Community-based Development: RISC-V architecture has been developed based on contributions from a worldwide community that consists of educational institutions, research organizations, and business entities. Such an approach aids in the rapid evolution of the architecture along with addressing various applications.
6. Adoption: There have been numerous instances where RISC-V has gained immense popularity within the industry with several businesses and institutions creating processors and tools based on RISC-V. Moreover, it is gaining acceptance within the academic circles as well.

7. Comparison with Other ISA's: Although x86 and ARM are popular ISAs, they do pose some challenges such as licensing fee and limitations. Hence, RISC-V provides an interesting and competitive approach since there are no such challenges involved.

RISC-V makes an intriguing ISA choice for the future of computing industry.

1.5. RISC-V ISA Overview

The base integer ISAs are very similar to the RISC processors except with no branch delay slots and with support for optional variable length instruction encodings.

1. Base ISAs
2. RV32I
3. RV64I
4. RV32E
5. RV128I

Each base ISA can be extended with one or more optional instruction set extensions to support more general software development. The base integer ISA ("I" extension) contains integer computations, integer loads, integer stores and control flow signals.

Standard extensions:

1. "M" Integer multiply & divide Extension.
2. "F" Single Precision Fl0ating Point Extension.
3. "D" Double Precision Fl0ating Point Extension.
4. "A" Atomic memory operations (AM0) (read, modify, and write memory)
5. "C" Compressed Instruction Extension which offer narrower 16-bit forms of common instructions.
6. "G" ("1MAFD") General-purpose ISA which contains all the above extension.

1.6 Risc V Instruction Set Architecture

In a RISC-V single-cycle processor, all operations are performed directly on registers, RISC-V single-cycle processor, all operations are performed directly on registers, adhering to the principle of a load/store architecture. This means the memory is access exclusively through load and store instructions, and no direct arithmetic or logical operation is performed on memory locations. The processor's core components work together to implement these operations efficiently:

R-type (Register type)

R-These instructions are used for register-to-register arithmetic and logical operations, such as add, sub, sll, slt, and, or, xor.

- **Operands:** 2 source registers, 1 destination register
- **Example:** add x1, x2, x3

- **When to cover:** First, as they represent the core ALU operations without memory branching

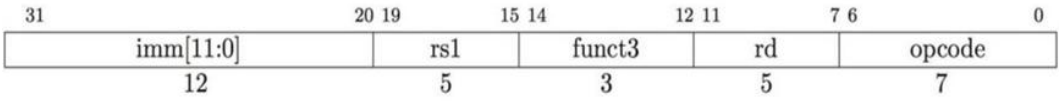
Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments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

Table 1. Registers Type

I-Type (Immediate Type):

Used in operations where the immediate value is needed instantly or used as the instruction load. Common instructions include addi, andi, ori, lw, jalr

- **Operands:** I source and destination register, 1 immediate value
- **Example:** addi. x5, x6, 10 or lw x10, 0(x5)
- **When to cover:** After R-type, as they build on similar datapath but introduce **immediate** decoding and memory read access.

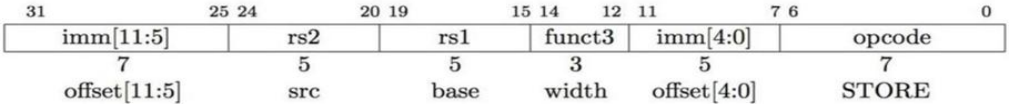


- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, `imm[11:0]`

S-Type (Store Type):

These commands are used to move data from a register to memory. They are necessary in order to be able to perform write-back of data to memory, like sw, sh, sb.

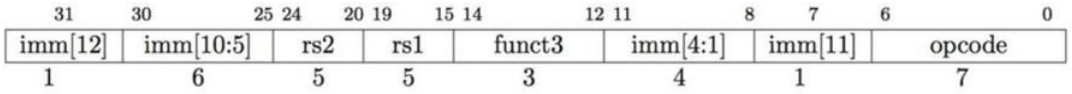
- **Operands:** 1 source register (data), 1 base register (address), 1 immediate offset
- **Example:** sw x10, 0(x5)
- **When to cover:** After I-type, as they complement load instructions and require understanding of memory write operations.



B-Type (Branch Type):

Branch instructions allow conditional changes to control flow, such as beq, bne, blt, bge.

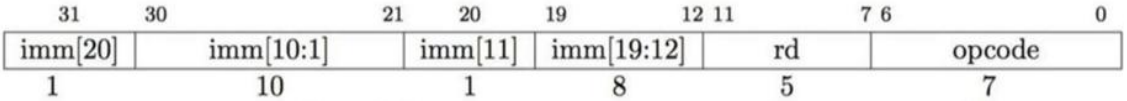
- **Operands:** 2 source registers, 1 signed immediate (branch offset)
- **Example:** beq x1, x2, label
- **When to cover:** After arithmetic and memory operations, because they introduce control hazards and decision-making logic.



J-Type (Jump Type):

Used for unconditional jumps, such as jal. These instructions are important for the implementing function calls and loops.

- **Operands:** 1 destination register, 1 immediate
- **Example:** jal x1, label
- **When to cover:** Last, since they involve PC-relative addressing and control flow redirection.



2. Implementation of Base Integer Instruction Set (RV64I)

The goal of this research project is to design a 5 stage pipelining of the integer instructions that exist in the RV64I Base Integer Instruction Set.

Instruction	Format	Opcode	Funct3	Funct7	Description
add	R	0110011	0x0	0x00	rd = rs1 + rs2
sub	R	0110011	0x0	0x20	rd = rs1 - rs2
xor	R	0110011	0x4	0x00	rd = rs1 ^ rs2
or	R	0110011	0x6	0x00	rd = rs1 | rs2
and	R	0110011	0x7	0x00	rd = rs1 & rs2
sll	R	0110011	0x1	0x00	rd = rs1 << rs2[5:0]
srl	R	0110011	0x5	0x00	rd = rs1 >> rs2[5:0]
sra	R	0110011	0x5	0x20	rd = rs1 >>> rs2[5:0]
slt	R	0110011	0x2	0x00	rd = (rs1 < rs2)? 1:0
sltu	R	0110011	0x3	0x00	rd = (rs1 < rs2)? 1:0
addw	R	0111011	0x0	0x00	rd = rs1[31:0] + rs2[31:0] (sign extend)
subw	R	0111011	0x0	0x20	rd = rs1[31:0] - rs2[31:0] (sign extend)
sllw	R	0111011	0x1	0x00	rd = rs1[31:0] << rs2 (sign extend)
srlw	R	0111011	0x5	0x00	rd = rs1[31:0] >> rs2 (sign extend)
sraw	R	0111011	0x5	0x20	rd = rs1[31:0] >>> rs2 (sign extend)
addi	I	0010011	0x0	imm[11:0]	rd = rs1 + imm
xori	I	0010011	0x4	imm[11:0]	rd = rs1 ^ imm
ori	I	0010011	0x6	imm[11:0]	rd = rs1 | imm
andi	I	0010011	0x7	imm[11:0]	rd = rs1 & imm
slli	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[5:0]
srli	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[5:0]
srai	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >>> imm[5:0]
slti	I	0010011	0x2	imm[11:0]	rd = (rs1 < rs2)? 1:0
sltiu	I	0010011	0x3	imm[11:0]	rd = (rs1 < rs2)? 1:0
addiw	I	0011011	0x0	imm[11:0]	rd = rs1[31:0] + imm (sign extend)
slliw	I	0011011	0x1	imm[5:11]=0x00	rd = rs1[31:0] << imm[4:0] (sign extend)
srliw	I	0011011	0x5	imm[5:11]=0x00	rd = rs1[31:0] >> imm[4:0] (sign extend)
sraiw	I	0011011	0x5	imm[5:11]=0x20	rd = rs1[31:0] >>> imm[4:0] (sign extend)
lb	I	0000011	0x0		rd = M[rs1+imm][7:0]
lh	I	0000011	0x1		rd = M[rs1+imm][15:0]
lw	I	0000011	0x2		rd = M[rs1+imm][31:0]
lbu	I	0000011	0x4		rd = M[rs1+imm][7:0]
lhu	I	0000011	0x5		rd = M[rs1+imm][15:0]
lwu	I	0000011	0x6		rd = M[rs1+imm][31:0]
ld	I	0000011	0x3		rd = M[rs1+imm]
sb	S	0100011	0x0		M[rs1+imm][7:0] = rs2[7:0]

sh	S	0100011	0x1		M[rs1+imm][15:0] = rs2[15:0]
sw	S	0100011	0x2		M[rs1+imm][31:0] = rs2[31:0]
sd	S	0100011	0x3		M[rs1+imm] = rs2
beq	B	1100011	0x0		if(rs1 == rs2) PC += imm
bne	B	1100011	0x1		if(rs1 != rs2) PC += imm
blt	B	1100011	0x4		if(rs1 < rs2) PC += imm
bge	B	1100011	0x5		if(rs1 >= rs2) PC += imm
bltu	B	1100011	0x6		if(rs1 < rs2) PC += imm
bgeu	B	1100011	0x7		if(rs1 >= rs2) PC += imm
jal	J	1101111			rd = PC+4; PC += imm
jalr	I	1100111	0x0		rd = PC+4; PC = rs1 + imm
lui	U	0110111			rd = imm << 12
auipc	U	0010111			rd = PC + (imm << 12)

Table 2. Base RV64I ISA

2.1. RV64I Design Process and Pipeline Stages

The processor's microarchitecture can be classified into two parts which are very closely linked together; namely, the datapath and the control unit. The datapath unit takes care of all computations on the data, and in the current implementation, it works on 64 bits because we have designed it based on the RV64I instruction set of RISC-V.

The design process starts from specifying the basic storage elements like program counter, register file, and memory blocks. Having specified these storage elements, combinational logic is then incorporated between these to find out how the data flow happens and the next state of the system can be determined.

Firstly, we designed single cycle microarchitecture then we applied pipelining and added L1 Cache for performance boost.

RISC V microarchitecture is shown in Figure 1.

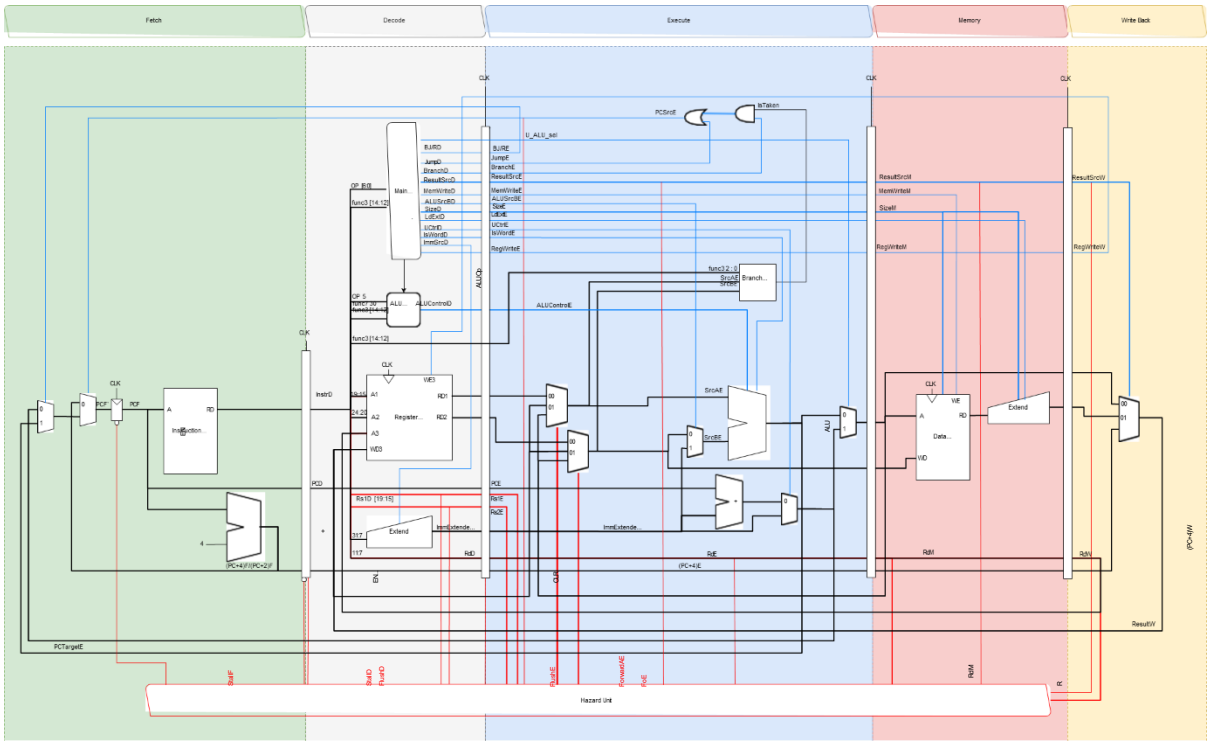


Figure 1. RISC V Microarchitecture

2.1.1. Instruction Fetch Stage

Instruction Fetch (IF) obtains the next instruction to be executed from memory. This is achieved through the use of the program counter register, which holds the address of the currently executing instruction. The next instruction is then obtained from the instruction memory using the above address, following which it can be forwarded to the next stage of the pipeline.

For fetching of the next instruction, it is necessary to modify the address present in the program counter such that it reflects the next address of the instruction to be executed. Usually, this is done by adding an increment of 4 bytes to the address held in the program counter, since each instruction takes up 4 bytes. At other times, the next address in the program counter depends on the type of control flow instruction being executed.

2.1.2. Instruction Decode Stage

Analysis of the instruction, which is already obtained in the ID phase, is done in this phase. All the required signals for controlling the datapath circuitry can be derived using the opcode and the rest of the fields in the instruction. This phase also comprises the fetching of operands from the register file.

Using this decoded information, the required control signals are then obtained to tell the execution unit what operation needs to be performed.

2.1.2.1. Register File

32 integer registers are there from X0 to X31, width 64-bit each. The usage of each register is as shown below.

Name	Register Number	Usage
zero	x0	Constant value 0
tp	x4	Thread pointer
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global 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
t3-6	x28-31	Temporary registers
s2-11	x18-27	Saved registers

Table 3. Integer Registers

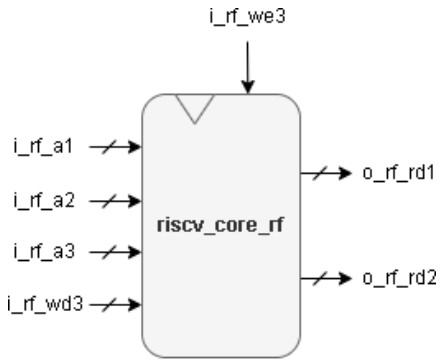


Figure 2. Block Diagram of Register File

Port Name	Direction	Description
i_rf_clk	Input	Clock
i_rf_a1	Input	Address of rs1 to be read from
i_rf_a2	Input	Address of rs2 to be read from
i_rf_a3	Input	Address of rd1 to be written to
i_rf_wd3	Input	Data to be written to rd1(64)
i_rf_we3	Input	Write Enable
o_rf_rd1	Output	Data read from rs1(64)
o_rf_rd2	Output	Data read from rs2(64)

Table 4. Register File Block Interface

2.1.2.2. Main Decoder

The main decoder is a vital element in the processor pipeline that decodes the instruction that has been fetched and provides control signals for execution. Different components of the instruction are analyzed, including opcode, register numbers, immediate values, and destination registers. These include:

- ALU operation code is determined by the main decoder depending on the type of instruction.
- Appropriate multiplexers are activated using output signals from the main decoder.
- In case of branches, the decision to take the branch is made using comparators in case of condition codes.
- Output signals are provided to determine whether to write back the result to the register file.
- In case of loads and stores, it provides control signals for memory access.

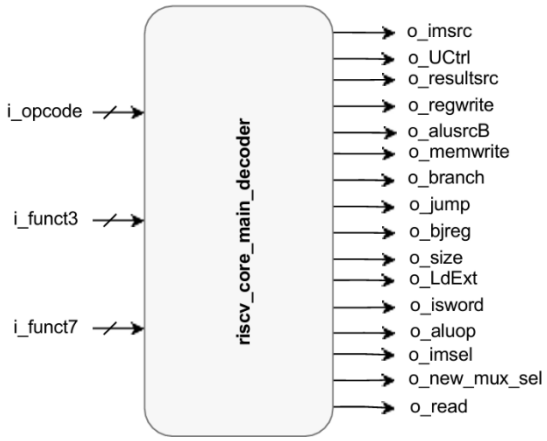


Figure 3. Main Decoder Block Diagram

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.

Table 5. Block Interface of Main Decoder

2.1.3 Execute Stage

The execution stage is the phase in the pipeline in which the instruction's execution happens. At the execute stage, the signals created during decoding are given to the input operands, which could have been fetched from the registers or immediately from the instruction itself. In this regard, the role of the Arithmetic Logic Unit (ALU) is quite important as it performs the required operation.

2.1.3.1 ALU (Arithmetic & Logical Unit)

The ALU is the essential part of the data path, which is used to perform operations such as arithmetic and logic calculations. The ALU receives two input operands and, depending on the control signals, gives the output.

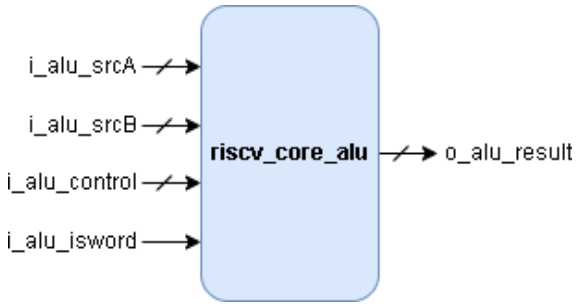


Figure 4. ALU Block Diagram

2.1.3.2 ALU Decoder

The ALU decoder is responsible for deciding what kind of operation needs to be done by the ALU. It examines particular fields within the instruction, such as funct3, some selected fields in the opcode, and funct7.

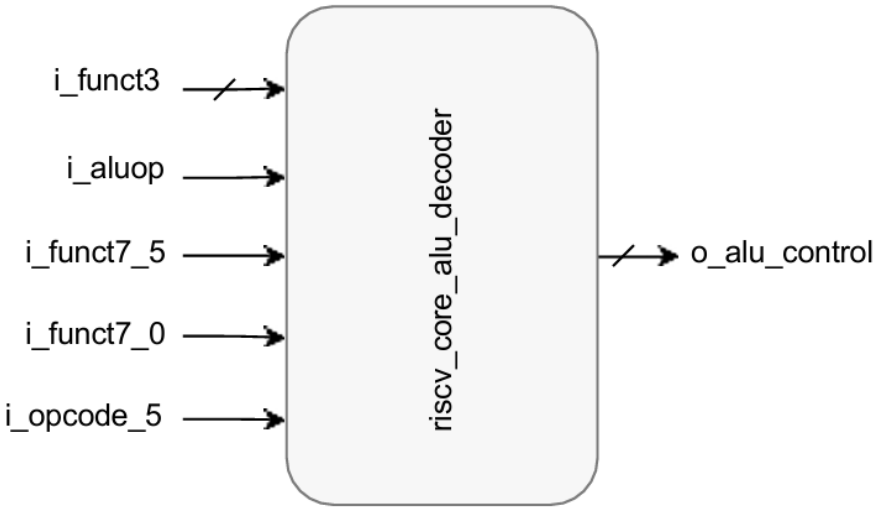


Figure 5. Block Diagram for ALU Decoder

Port Name	Direction	Description
i_funct3	Input	funct3 bits of the instruction
i_aluop	Input	Enables the ALU decoder
i_funct7_5	Input	Bit number 5 of funct7 part of the instruction
i_funct7_0	Input	Bit number 0 of funct7 part of the instruction
i_opcode_5	Input	Bit number 5 of opcode part of the instruction
o_alu_control	Output	Specifies the ALU operation

Table 6. ALU Decoder Block Interface

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

Table 7. Block Interface of ALU

2.1.3.3 Branch Unit

Branch Unit is where the control instructions such as branch instruction and unconditional jumps are processed. The Branch Unit compares whether the branch instruction is satisfied and produces an output on the result.

When there is an execution of the branch instruction, the next value of the Program Counter is produced through the target address, which is produced using the ALU. The role played by the Branch Unit in determining how to execute cannot be downplayed.

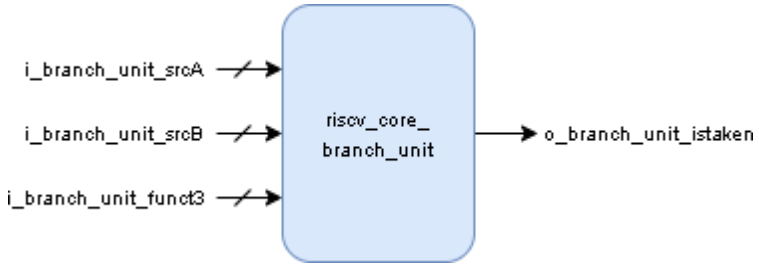


Figure 6. Block Diagram of Branch Unit

Port Name	Direction	Width	Description
i_btanch_unit_srcA	Input	64	Operand 1
i_branch_unit_srcB	Input	64	Operand 2
i_branch_unit_funct3	Input	3	Specifies branch operation
o_branhc_unit_istaken	Output	1	Flag specifies if the branch is taken or not

Table 8. Block Interface of Branch Unit

2.1.4 Memory Stage

The memory stage plays an integral role in the processing pipeline where it carries out all instructions that have data movement from and to the processor and memory. Instructions that are related to data movement include loading and storing of data from the memory.

In RISC-V instruction set architecture, loading and storing operations transfer data between the registers and memory addresses. The loading operation follows the I-format, while the storing operations use the S-format. Memory addresses for such operations are derived through adding the contents of rs1 to the sign extended 12-bit offset..

Loading operations read data from the memory into the destination register rd, while storing operations write the data from register rs2 to the memory address.

- Load Instructions:
  - The LD instruction stores a 64-bit value stored in memory into rd.
  - The LW instruction stores a 32-bit value stored in memory into rd where this value is first extended by sign-extension into a 64-bit word.
  - LH stores a 16-bit value into rd using sign-extension to expand it into a 64-bit word.
  - And LHU stores a 16-bit value into rd using zero-extension into a 64-bit word.
  - LB instructions are similarly defined for 8-bit values.

- Store Instructions:
    - The SD instruction stores a 64-bit value contained in register rs2 into memory.
    - The SW instruction stores a 32-bit value contained in the lower half of register rs2 into memory.
    - The SH instruction stores a 16-bit value contained in the lower half of register rs2 into memory.
    - SB instruction st0res an 8-bit value contained in the half of register rs2 into memory..

2.1.4.1 Load Extend

This unit adjusts the data fetched from memory to match the 64-bit register width. Depending on the type of load instruction, it either sign-extends or zero-extends the value to ensure correct representation before writing it into the register file.

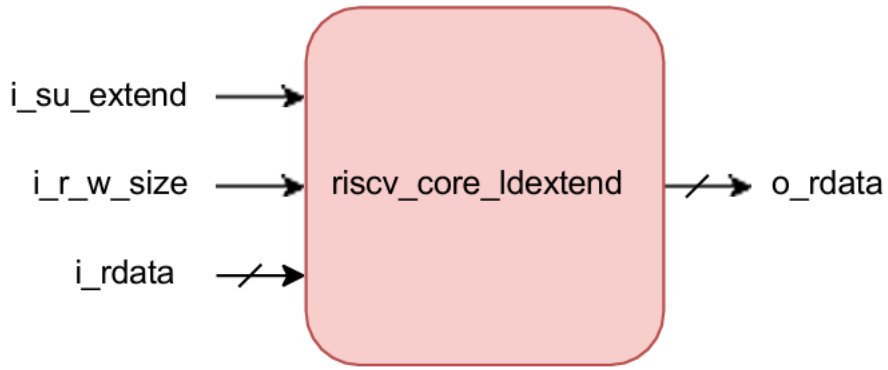


Figure 7. Block Diagram of Load Extend

The block interface of the load extend is shown in Table 12.

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	64	Data loaded from memory without extend.
o_rdata	Output	64	Data loaded from memory after sign/zero extend.

Table 9. Load Extend Block Interface

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

Table 10. Function Table of Load Extend

2.1.4.2 Data Cache

Data Cache will be discussed in section 4

2.1.5 Writeback Stage

This particular phase involves writing back the final result into the register file after executing the instruction. By now in the pipeline, the data is either coming from the ALU operation or from the memory itself and needs to be written into the destination register.

This phase mainly involves the use of a multiplexer that determines the appropriate source for writing back to the register file. This is decided by the signal generated by the main decoder depending upon the nature of the instruction executed.

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

Table 11. Function Table of Writeback MUX

2.1.6 Hazard Unit

The hazard unit is designed to handle data hazards and control hazards in the pipeline to ensure proper functioning of the program. Hazard unit keeps on checking the instruction dependability and control transfer at all times and reacts accordingly to the hazard found in the process.

Data hazard occurs when an instruction tries to access a value which has not been updated in the register yet by a previous instruction. This dependency is often termed as RAW, and such cases can usually be taken care of using data forwarding. However, when data forwarding does not work, a stall is introduced into the pipeline.

These control hazards arise due to uncertainty in the sequence of program execution, especially if the processor has not decided on the result of the instruction that will be taken for branch/jump execution. This hazard can be solved by suspending instruction fetch till this uncertainty is sorted out or through the process of branch prediction whereby the CPU predicts which instruction will follow. In the event that the predicted instruction is wrong, the wrongly fetched instruction is flushed from the pipeline.

A reduction in the time required to resolve the branch helps limit the number of instructions that have to be flushed.

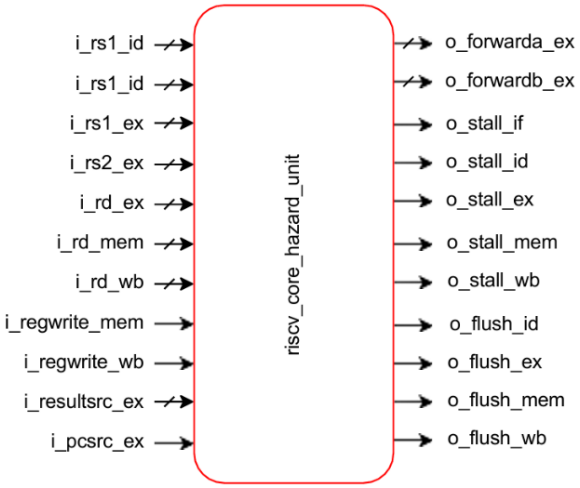


Figure 9. Hazard Unit Block Diagram

Port Name	Direction	Description
i_hazard_unit_rs1_id	Input	RV64I Signals
i_hazard_unit_rs2_id	Input	
i_hazard_unit_rs1_ex	Input	
i_hazard_unit_rs2_ex	Input	
i_hazard_unit_rd_ex	Input	
i_hazard_unit_mbusy	Input	
o_hazard_unit_forwarda_ex	Output	Forwarding Signals
o_hazard_unit_forwardb_ex	Output	
o_hazard_unit_stall_if	Output	Stall Signals
o_hazard_unit_stall_id	Output	
o_hazard_unit_stall_ex	Output	
o_hazard_unit_stall_mem	Output	
o_hazard_unit_stall_wb	Output	
o_hazard_unit_flush_id	Output	Flush Signals
o_hazard_unit_flush_ex	Output	
o_hazard_unit_flush_mem	Output	
o_hazard_unit_flush_wb	Output	

Table 12. Block Interface of Hazard Unit

3. Implementation of Compressed Instructions

The C extension for RISC-V provides small encodings for a chosen set of base instructions, thus offering code compression. Generally it is be useful for both high-performance machine and power-efficient devices since code size reduction allows better utilization of memory and instruction fetching..

This extension is not independent; rather, it is built atop the RV32I and RV64I instruction sets. The C extension adds 9 different types of instructions that fit into 16 bits.

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

Table 26. Compressed instruction formats for 16-bit RISC

3.1 Compressed Operations with Immediate

Format CI is called the Compressed Instruction format because it contains compressed instructions with immediate values. Unlike the normal I-format, which is a 32-bit instruction format, this compressed instruction format uses 6-bits for the immediate field compared to 12-bits in the I-format. The other point to remember is that the same register is used as the source as well as destination.

One observation worth making here is that the register number is a 5-bit field.

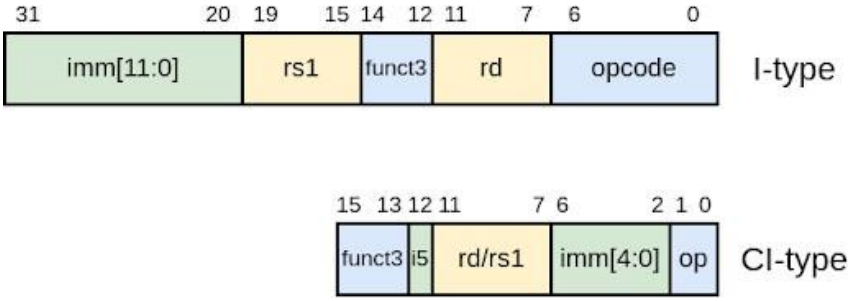


Figure 10. Comparison Between I-type and CI-type Format.

Despite these reductions, the register field remains 5 bits wide, meaning that all general-purpose registers can still be accessed. However, due to the smaller immediate range and reuse of registers, instructions like c.addi offer less flexibility compared to their full-length counterparts.

3.2 Arithmetic Instructions for Compressed Design

Another form of compression is the CA-type compression, which is illustrated graphically below.

The CA-type compressed format is employed for performing arithmetic operations in compressed form. Here, one source register is merged with the destination register, and the register fields are coded using just 3 bits. The 3-bit codes do not represent any actual register number but instead address some common subset of frequently utilized registers.

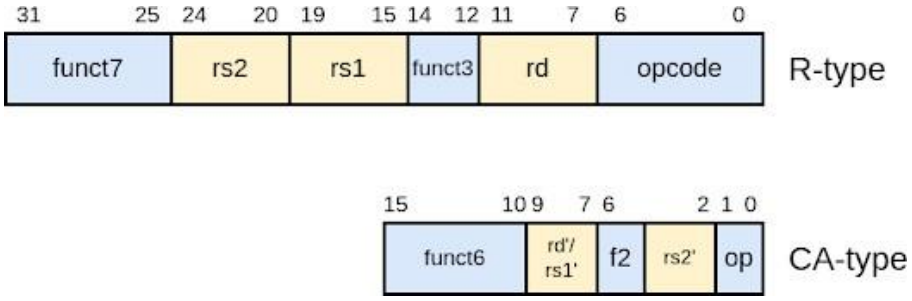


Figure 11. Comparison Between R-type and in CA-type format.

Because of this mapping, the registers in compressed instructions are denoted with a prime notation (rd′, rs1′, rs2′) to distinguish them from standard register fields. The selected subset of registers reflects common usage patterns, helping maintain efficiency despite reduced encoding space.

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

Overall, compressed instructions in the RVC extension trade off some expressiveness for compactness. They support a limited range of operations and operands compared to full-length instructions but are designed to cover the most commonly used cases. Although they utilize more opcode space, their frequent use leads to an overall reduction in program size, improving system efficiency.

3.3 Compressed Decoder Design

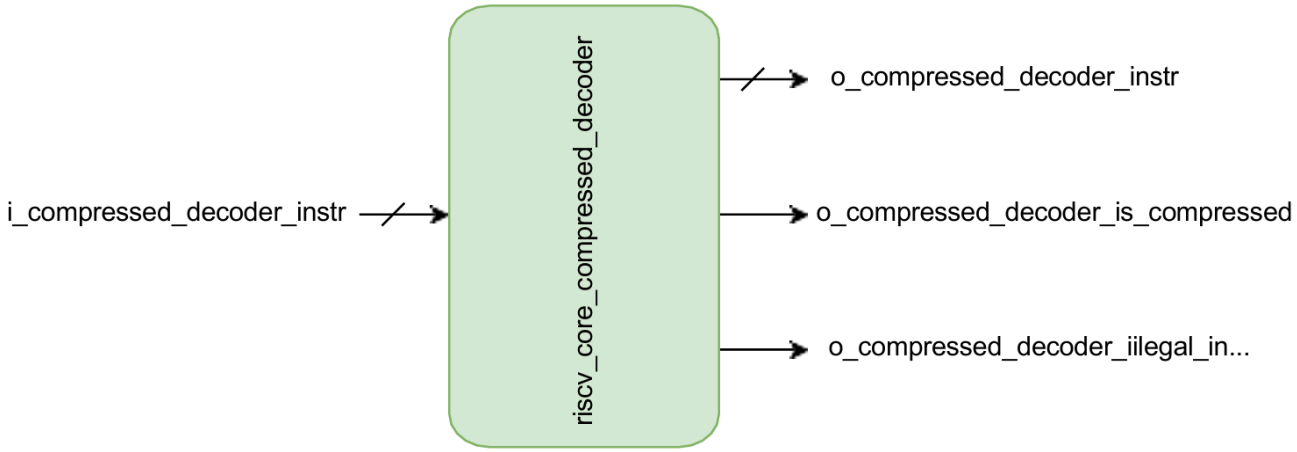


Figure 12. Compressed Decoder Block Diagram

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.

Table 14. Block Interface of Compressed Decoder.

4 Implementation of Memory Hierarchy

4.1 Literature Review

4.1.1 Introduction

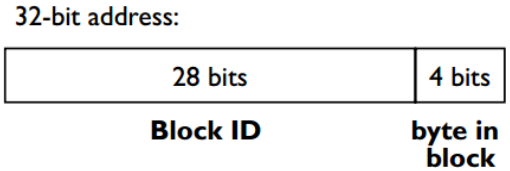
Cache memory can be defined as a very fast storage medium which sits between the main memory and the processor in order to enhance system performance by minimizing data access time. Cache memory has gained much importance especially in multi-core processors since it allows higher performance of the CPU compared to the main memory. This performance difference is bridged using very fast caches that are placed on the chip.

The Cache keeps a copy of frequently used data and instructions in temporary storage to allow for a much faster access by the processor than accessing the data from the main memory. Serving as an intermediary between the CPU and main memory, the cache contributes towards effective transfer of information, preventing unnecessary delays in processing. As cache memory is located near to the processor, it provides a faster access compared to regular memory.
A key advantage of using cache memory is its fast access capability which makes it very efficient in terms of enhancing system performance. Nonetheless, using cache memory requires increased power consumption and space. The current paper reviews the subject of memory hierarchy, addressing the basic fundamentals behind cache memory.

4.1.2 Caches

While there could be many levels in a memory hierarchy, the transfer of data takes place between the adjacent levels alone. As such, the analysis can be done taking into consideration only two adjacent levels. The higher level will have limited capacity as compared to the lower level, but provides for faster access because it uses more sophisticated technology than the lower level. The concept of cache works on the principle that data is fetched in blocks or chunks, which are referred to as cache lines. These lines are accessed by the cache from a lower level memory or backing store. In other words, while accessing the data from the lower level memory, it is segmented in the form of blocks that are distinguished by the block.

As an example, in a memory address of 32 bits, we can divide these bits into two sections, one containing the block number and the other containing the offset for that particular byte of that block. Now, if the offset contains 4 bits, then the size of the block is 16. This idea of addressing can be applied to other types of storage mechanisms as well, such as a disk cache.



Since cache memory is much smaller than the backing store, not all requested data will be present in it at all times. To determine whether a required block exists in the cache, a tagging mechanism is used. Cache tags store the identifiers of the blocks currently held in the cache, with each tag corresponding to a specific cache entry. These entries can be implemented using various hardware or software structures, such as SRAM arrays, trees, or linked lists, depending on the system design.

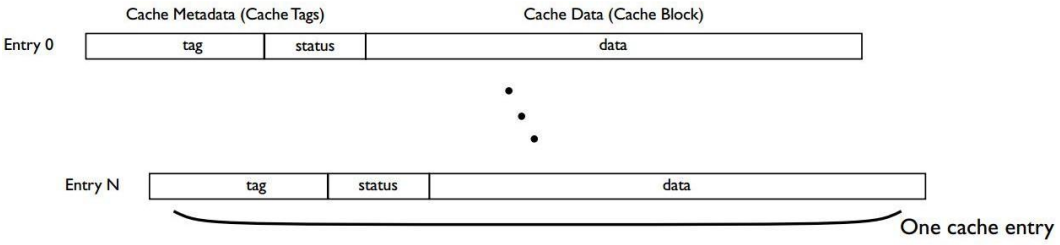


Figure 13. Basic Structure of a Cache.

4.2 Data Cache and Design

4.2.1 Data Cache Design Without Atomic Instructions

d

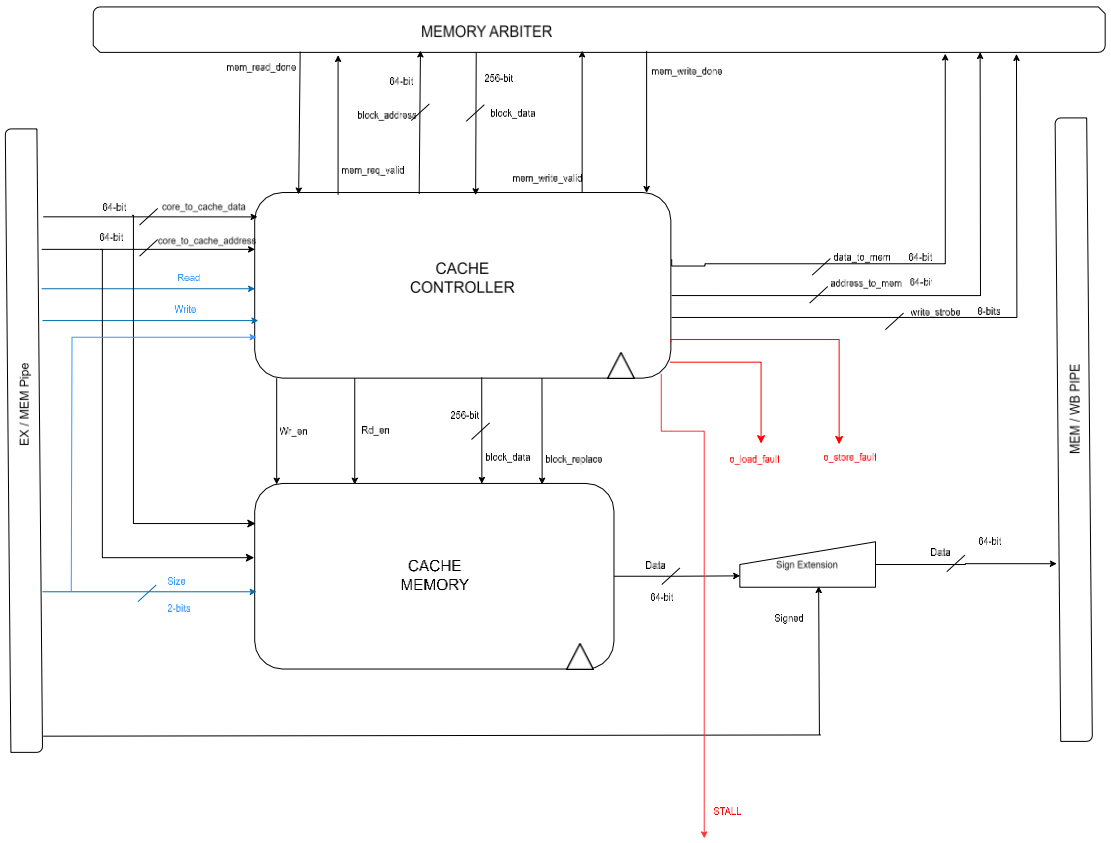


Figure 14. Cache Microarchitecture.

Specification	Chosen one
Block Placement	Direct Mapping
Block Identification	Using Tag and Index
Block Replacement	Block is replaced in the place of the missed one.
Write Strategy	Write-Through and Write Allocate
Cache Size	4 KiB
Block Size	4 Doublewords (2^5 Bytes)

Table 15. Cache Specifications.

[63:12]	[11:5]	[4:3]	[2:0]
TAG	INDEX	BLOCK_OFFSET	BYTE_OFFSET

- BYTE_OFFSET : Since each Doubleword (DW) is 8 bytes so we need these 3-bits for the byte offset in each DW.
- BLOCK_OFFSET: Since we have 4 DW in each block so we need 2-bits to identify the needed DW.
- INDEX: Since we have 4KiB and each block has 4 DW and each DW has 8 bytes then we have 128 (2^7) $Blocks$ so we need for the INDEX 7-bits to identify the needed block.
- TAG : Each block has its own TAG bits which are the rest of the 64-bit address to uniquely identify the block and check if it is a miss or hit.

4.2.1.1 Cache Memory

It is the block that contains the data memory inside it and responsible for storing or load data from that memory.

4.2.1.2 Cache Controller

It is the block responsible for dealing with store and load instructions indicating whether they are cache miss or hit since it has the tag memory of the cache inside it, if there is fault or not and sending the appropriate signals to both cache memory and memory arbiter when needed.

Signal	Width	From	To	Description
core_to_cache_data	64-bit	core	Cache Controller & Cache Memory	Data coming from core to be written inside the cache if it is a store instruction
core_to_cache_address	64-bit	core	Cache Controller & Cache Memory	Address coming from core to which the data is to be written to (if store) or the address of the data to be read from cache (if load)
read	1-bit	core	Cache Controller	Control signal from core indicating that the current instruction is a load instruction
write	1-bit	core	Cache Controller	Control signal from core indicating that the current instruction is a store instruction
size	2-bit	core	Cache Controller & Cache Memory	Control signal indicating the size of the data to be written or read from cache
Wr_en	1-bit	Cache Controller	Cache Memory	Control signal from cache controller to cache memory indicating cache hit and tells cache memory to STORE the data present in core_to_cache_data in the address present in core_to_cache_address
Rd_en	1-bit	Cache Controller	Cache Memory	Control signal from cache controller to cache memory indicating cache hit and tells cache memory to LOAD the data in the address present in core_to_cache_address
block_data	256-bit	Cache Controller	Cache Memory	Cache Line (Block) coming from main memory to cache due to cache MISS
block_replace	1-bit	Cache Controller	Cache Memory	Control signal from cache controller to cache memory indicating that block_data contains the new cache line and tells cache memory to store

mem_req_valid	1-bit	Cache Controller	Memory Arbiter	Control signal indicating cache MISS and that a block of data defined by block_address is requested
block_address	64-bit	Cache Controller	Memory Arbiter	Address of block of data needed by cache from main memory due to cache MISS
block_data	256-bit	Memory Arbiter	Cache Controller	Block of data from Main Memory serving the cache MISS state
mem_read_done	1-bit	Memory Arbiter	Cache Controller	Control signal indicating that cache request is done and that needed block is present on block data bus
mem_write_valid	1-bit	Cache Controller	Memory Arbiter	Control signal indicating cache wants to write inside main memory (since it is write-through policy cache)
data_to_mem	64-bit	Cache Controller	Memory Arbiter	Data to be written inside main memory from cache
address_to_mem	64-bit	Cache Controller	Memory Arbiter	Address of data to be written inside main memory from cache
write_strobe	8-bits	Cache Controller	Memory Arbiter	Control signals indicating which bytes of double word should be written inside the main memory
mem_write_done	1-bit	Memory Arbiter	Cache Controller	Control signal indicating that cache write data has been written inside main memory
o_store_fault	1-bit	Cache Controller	Core (Hazard Unit)	Control signal indicating that store instruction is fault
o_load_fault	1-bit	Cache Controller	Core (Hazard Unit)	Control signal indicating that load instruction is fault
Stall	1-bit	Cache Controller	Core (Hazard Unit)	Control signal indicating that core should be stalled due to cache
signed	1-bit	core	Sign Extension	Control signal indicating if data to be sent to core as signed or unsigned

Table 16. Signals Description of Cache.

The cache consists of 4 states which are:

- **IDLE:**
  State reached upon reset. When there is a load and cache hit, it is handled in the IDLE state. When there is a cache miss a stall is risen and goes to MEM_REQ state. When there is a store instruction with cache hit, a stall is risen and goes to MEM_WRITE state.
- **MEM_REQ:**
  When reached, it loads the data of the block needed and waits till ACK from memory is returned which is mem_read_done and then goes to UPDATA_CACHE state.
- **UPDATE_CACHE:**
  State where cache content is updated to handle the cache miss and then goes back to IDLE state.
- **MEM_WRITE:**

When reached, it loads the data to be written on main mem and waits till ACK from memory is returned which is mem_write_done and then goes back to IDLE state.

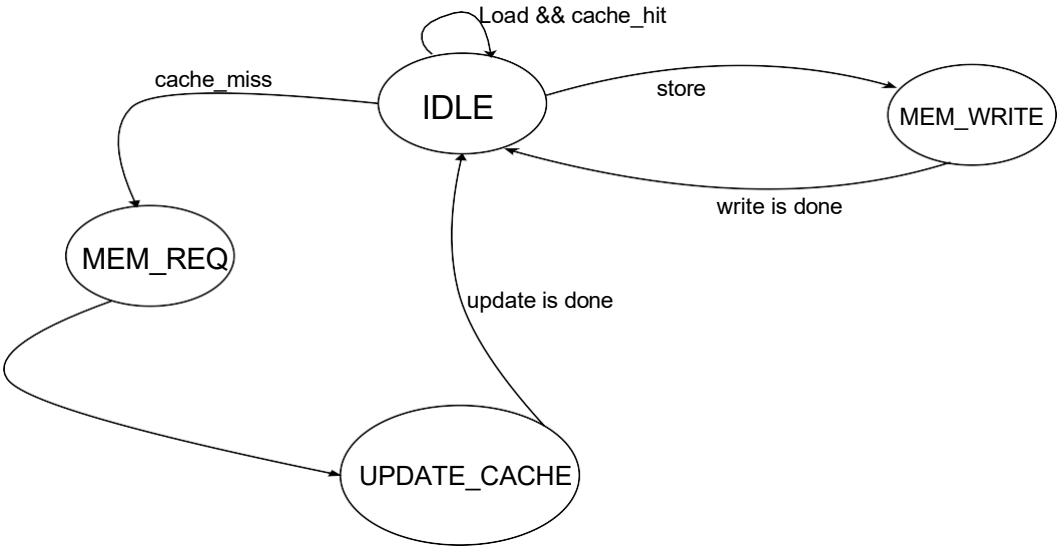


Figure 15. Cache States.

4.3 Instruction Cache

Its Implementation is the same as Data Cache but with no writing mechanism and it is always reading from the address provided as long as it is a cache hit.

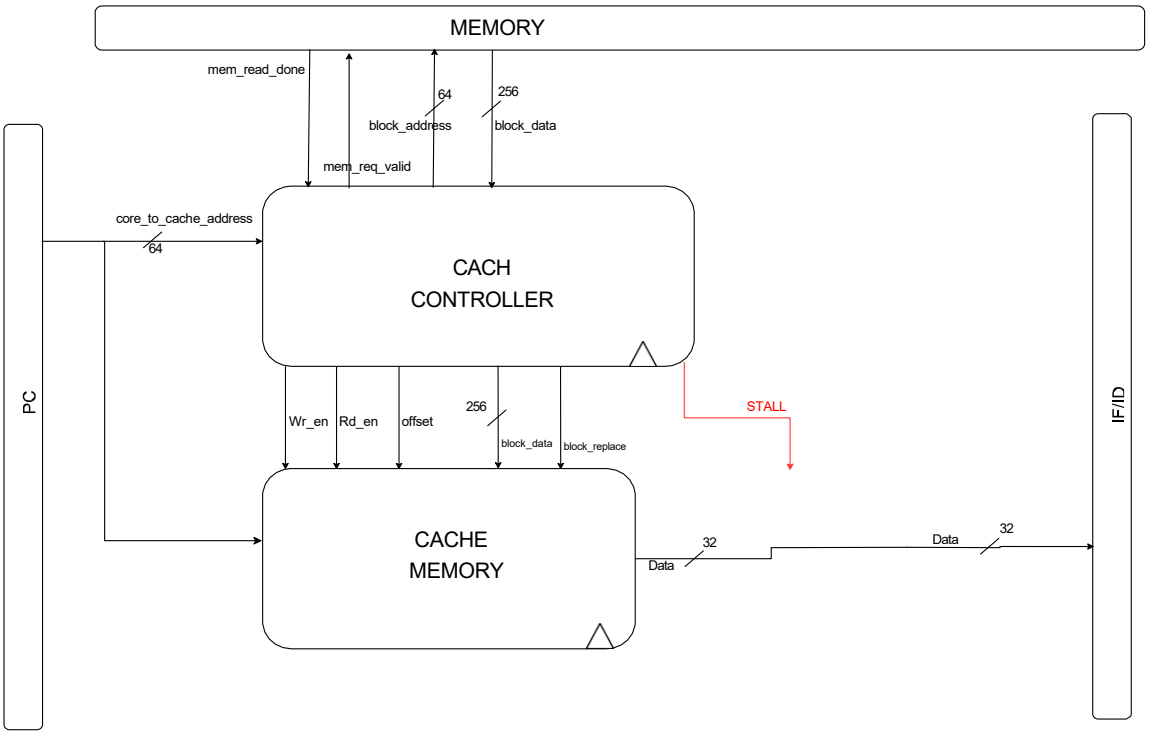


Figure 16. Instruction Cache Microarchitecture.

4.3.1 Added Signals

OFFSET signal: Since we have Compressed instructions, we jump 2 locations when we encounter such an instruction. This may cause an instruction to be present in TWO different cache lines. That can happen when 2 bytes are in the last 2 bytes of a line and the other 2 bytes are at the beginning of the next cache line. To solve this problem, we added the OFFSET signal, where the controller checks for two cases for cache miss, one is the known which is the whole block is missing , and the other one when the next address which MAY contain the rest of the instruction is also missing, the cache requests the needed blocks and raises the OFFSET signal to the memory indicating that the next instruction to be loaded is present in two cache lines.

4.3.2 Cache States

The instruction cache states are shown in Figure 34.

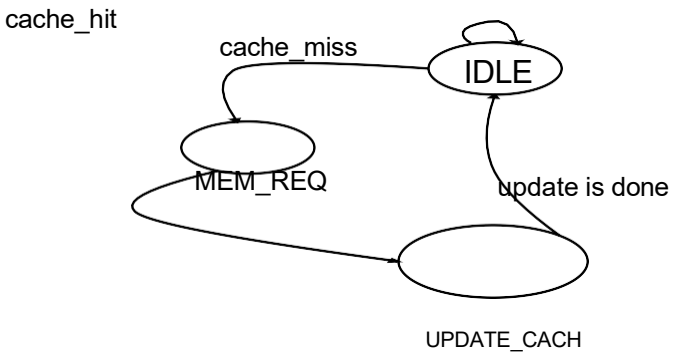


Figure 17. Instruction Cache States.

4.4 Main Memory & Memory Arbiter

- Main Memory
  We implemented the main memory on the Block Ram (BRAM) of our FPGA. This memory contains both the instructions and data of our core and that by partitioning the memory. The instruction memory starts at address 0x000 and the data memory starts at address 0x001000000.

- Memory Arbiter
  Due to lack of time and to complete our system, we implemented a native interface memory arbiter with no AXI interface. This arbiter serves both the instruction and data caches at cache misses and when data cache wants to write on main memory.

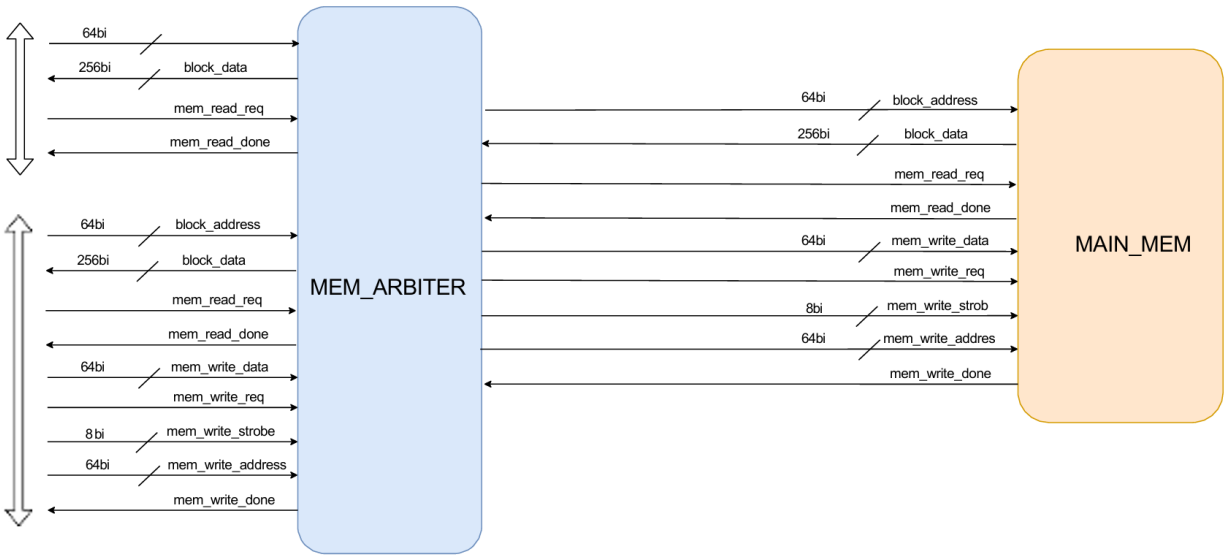


Figure 18. Main Memory and Memory Arbiter.

5. RV64IMAC with L1 Cache

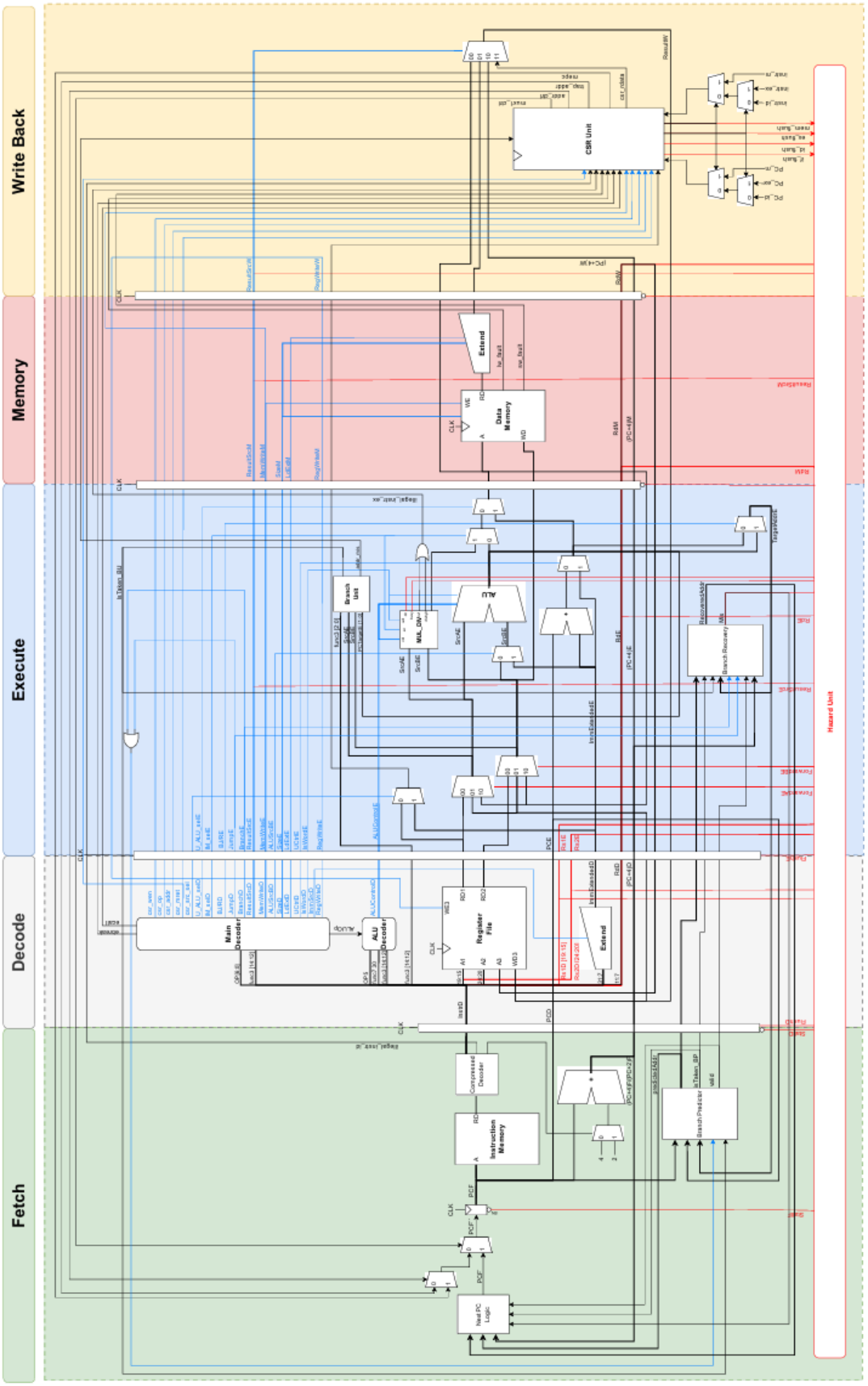


Figure 19. RISC V64 IMAC Architecture

6. ASIC Implementation

Physical Design Flow

Asic design flow refers to a set of procedures through which a customized chip design for an intended application is designed and manufactured. This section gives an outline of the steps that characterize Asic design flow process. Asic design flow is outlined below.

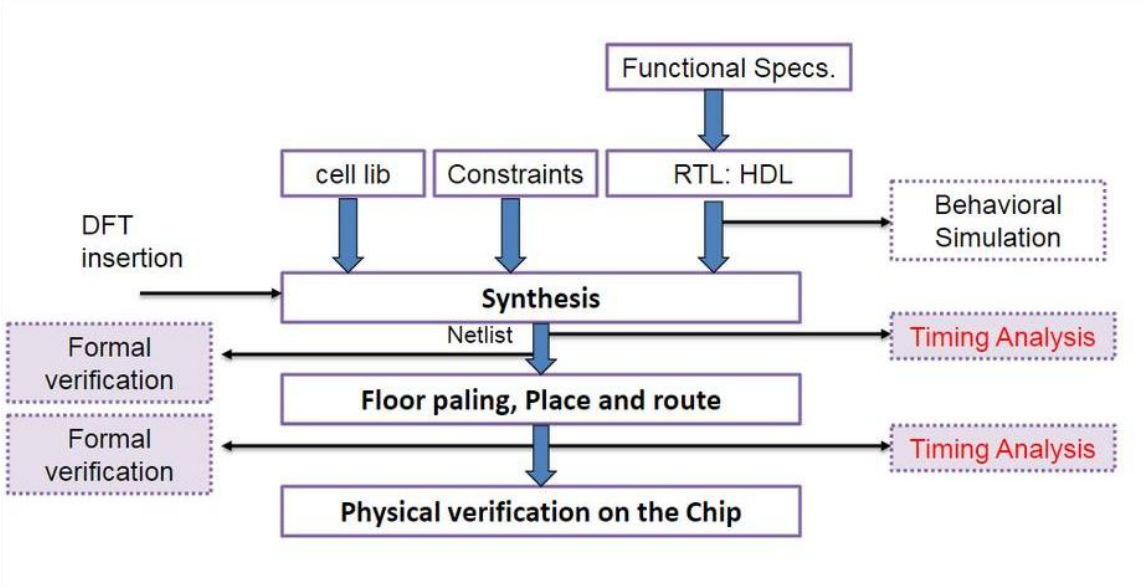


Figure 20. ASIC Design Flow.

Specification and Requirement Analysis

- Define the functionality, performance, p0wer, and area requirements of the ASIC.
- Identify the target technology node and process.

Architectural Design

- Develop the high-level architecture.
- Create block diagrams and define interfaces and protocols.
- Perform high-level simulations to validate the architecture.

RTL Design

- Write RTL code using hardware description 1anguages (HDLs) like Verilog or VHDL.
- Simulate the RTL code to ensure it meets the functional requirements.

Functional Verification

- Create testbenches and use simulation tools to verify the functionality of the RTL code.
- Convert the RTL code into a gate-level netlist using synthesis tools.
- Optimize the design for timing, area, and power constraints.
- Perform static timing analysis (STA) to ensure timing requirements are met.

Design for Testability (DFT)

- Insert test structures such as scan chains, Built-1n Self-Test (BIST), and b0undary scan.
- Generate test vectors to facilitate manufacturing testing.

Floor planning and Partitioning

- Plan the physical layout of the ASIC.
- Define the placement of major blocks and input/output (I/O) pads.
- Ensure efficient use of space and routing resources.

Placement and Routing

- Place the standard cells and macros within the defined floorplan.
- Route the interconnections between cells and blocks.
- Optimize the placement and routing for performance, power, and area.

Power Analysis and Optimization

- Analyze the power consumption of the design.
- Implement power optimization techniques such as clock gating, multi-Vt cells, and power gating.

Timing Analysis and Sign-off

- Perform detailed static timing analysis t0 ensure all timing constraints are met.
- Check for setup, hold, and clock skew issues.
- Perform signal integrity checks and noise analysis.

Physical Verification

- Perform Design Rule Check (DRC) to validate that the physical design meets the design rules for the target technology
- Also do the LVS (Layout Versus Schematic) checks.

Tape-out

- Prepare the final GDSII or OASIS file for manufacturing.
- Submit the design to the foundry for fabrication.

Fabrication & Packaging

- The foundry fabricates the ASIC using the design files submitted to it
- This takes many processes such as photolithography, etching, doping, and metallization.

7. SOFTWARE REQUIREMENT

This work involves the design, simulation, and verification of the designed RISC-V processor.
The implementation workflow made use of several industry-standard EDA tools for synthesis, simulation, waveform analysis, and FPGA-based implementation.
Below is a summary of the tools utilized:

1. Vivado Design Suite (by Xilinx)

Purposes: RTL Development, synthesis and FPGA implementation
Uses:
Design entry using Verilog HDL
Integrating IP & block design (If any)
Function simulation and waveforms observation
Bit stream generation for FPGA configuration
Target Device: Xilinx FPGA (e.g., Basys 3/Nexys A7)

2. Synopsys VCS (Verilog Compiler Simulator)

Purpose: Simulation of RTL accurately and fastly
Uses:
Compiling verilog files
RISC-V Testbench Simulation Timing Accurately
Integration with Verdi for Wave dump in .fsdb format

3. Synopsys Verdi

Purposes: Observation of wave forms and debugging
Uses:
Analyzing RTL behaviour through .fsdb files
Debugging Control Signal, ALU Output, Memory Interfacing and PC increment

4. Synopsys Design Compiler (DC)

Purposes: RTL logic synthesis to generate gate level netlist
Uses:
Synthesis of RTL using Verilog
Constraint driven synthesis using SDC
Generation of Area Power & Time report

8. SIMULATION IMPLEMENTATION AND WAVEFORMS

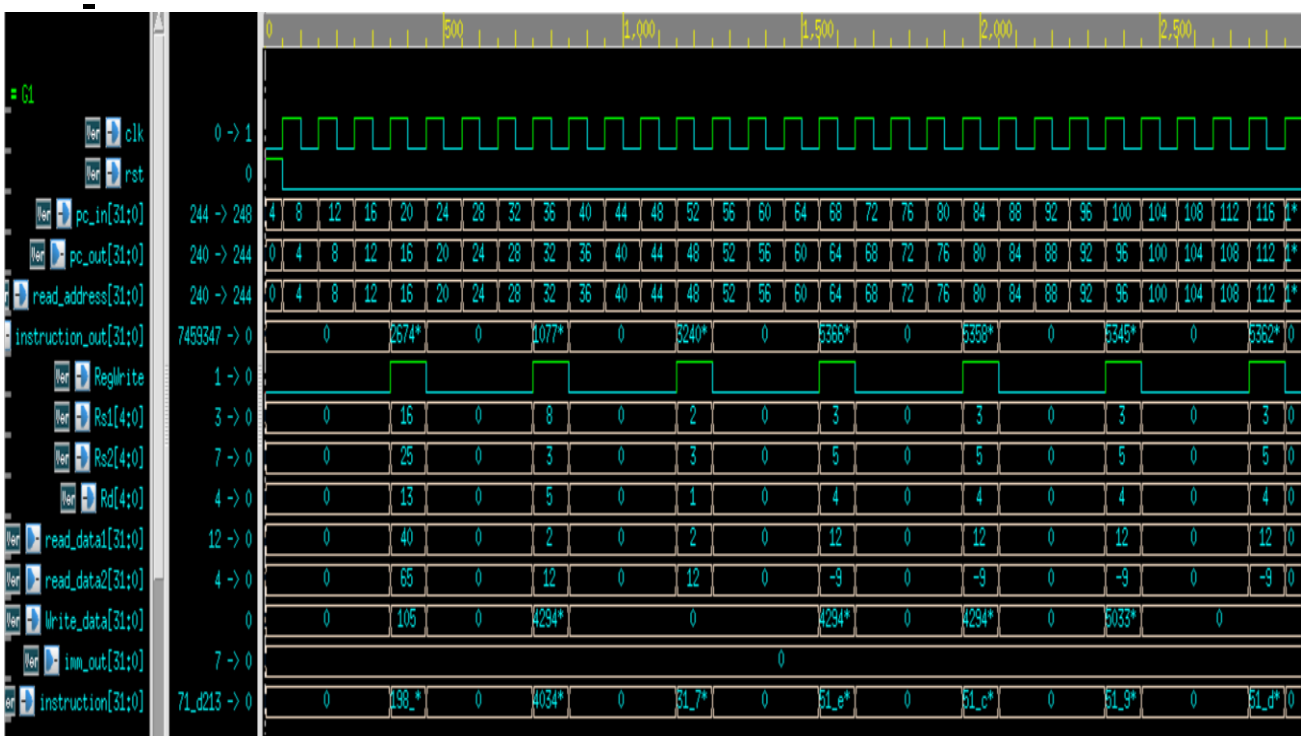


Figure 21 Simulation Waveform 1

This is the testing stage. Time is represented by the x-axis, while each row represents the behavior of a certain signal or data line with respect to each clock cycle.
Some examples include the following:

- The clock signal is the heart of the whole process.

- Instruction out is a representation of the current instruction being executed.

- Register signals such as Rs1, Rs2, as well as memory read/write signals represent data flows as the processor processes the instructions.

This type of waveform enables one to verify the proper execution of decoding, instruction execution, and writeback operations. This is an essential verification tool that helps detect any problems in the processor's operations.
It allows you to ensure that every part of the processor is functioning properly at every stage.

Some Operations performed on RISC-V using Verdi

I_Mem[4] = 32'h019806B3; // **ADD x13, x16, x25**

Registers[16] = 40;

Registers[25] = 65;

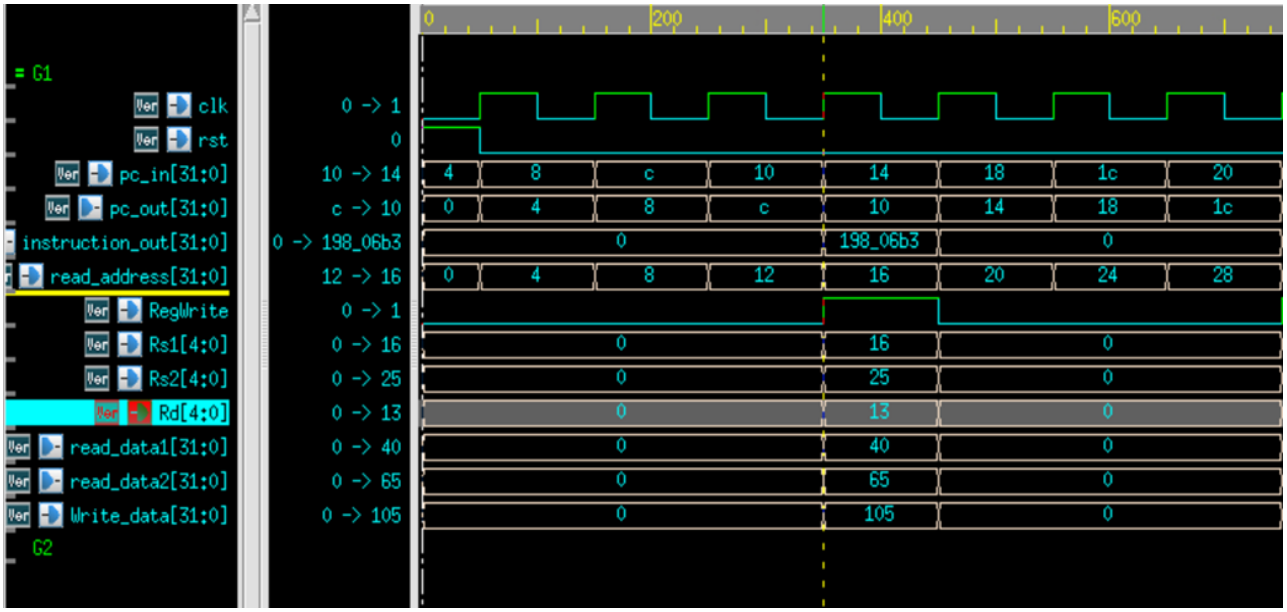


Figure 22 Simulation Waveform 2

I_Mem[8] = 32'h403402B3; // **SUB x5, x8, x3**

Registers[8] = 2;

Registers[3] = 12;

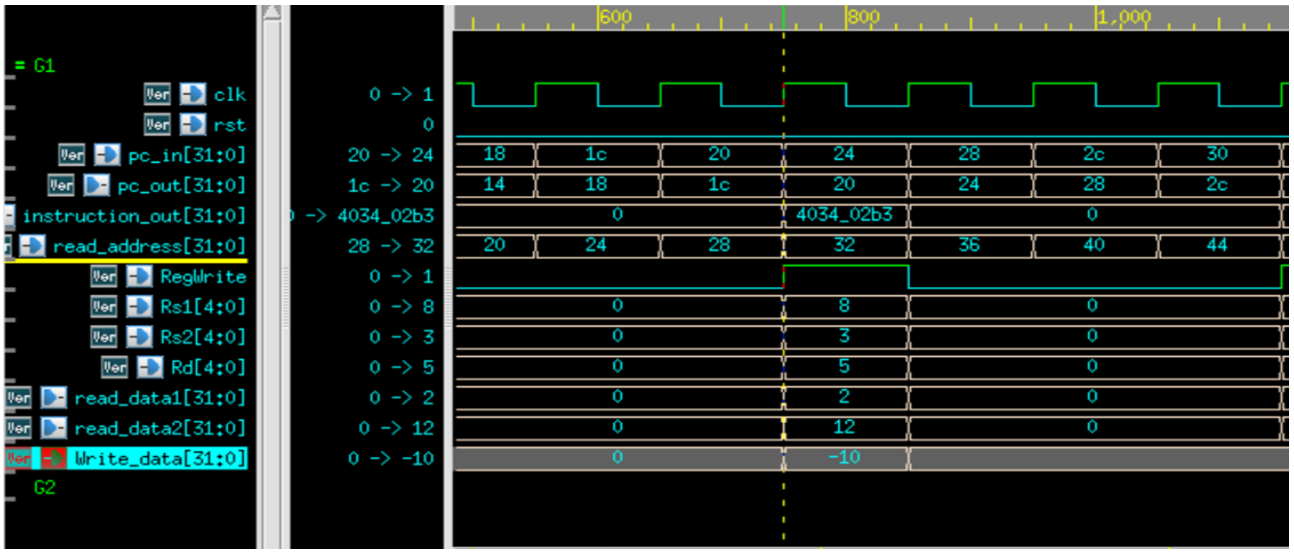


Figure 3Simulation Waveform 3

I_Mem[40] = 32'h002A8B13; // **ADDI x22, x21, 2**

Registers[8] = 80;
Registers[3] = 2;

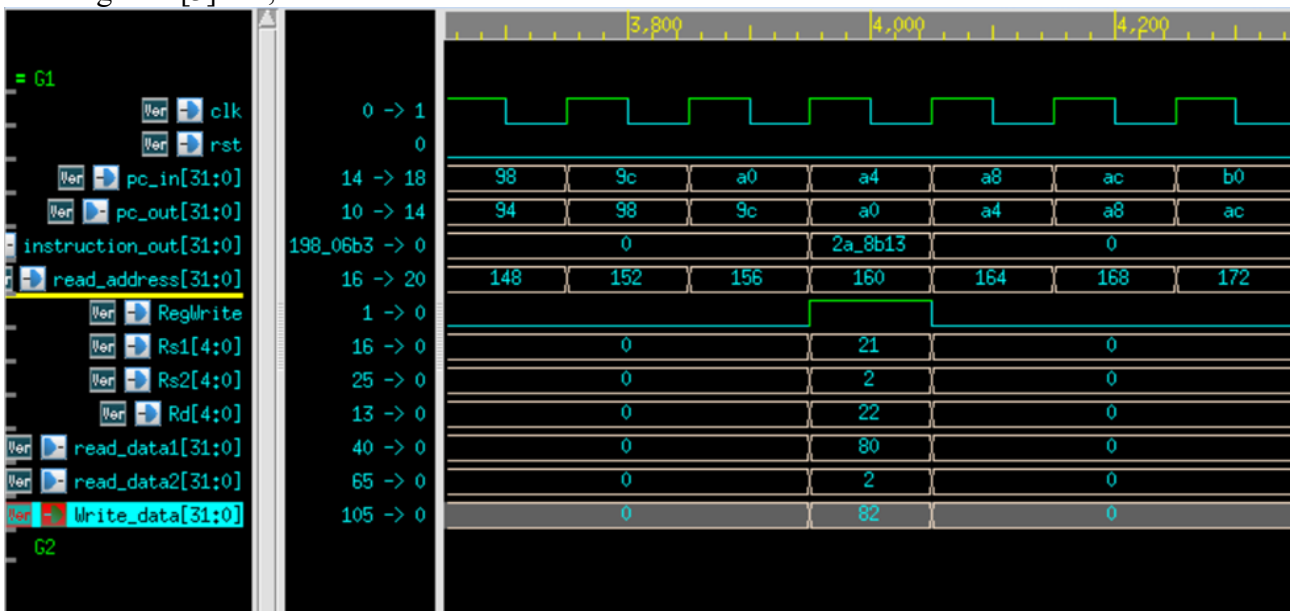


Figure 24 Simulation Waveform 4

I_Mem[44] = 32'h00346493; // **ORI x9, x8, 3**

Registers[8] = 10;
Registers[3] = 11;

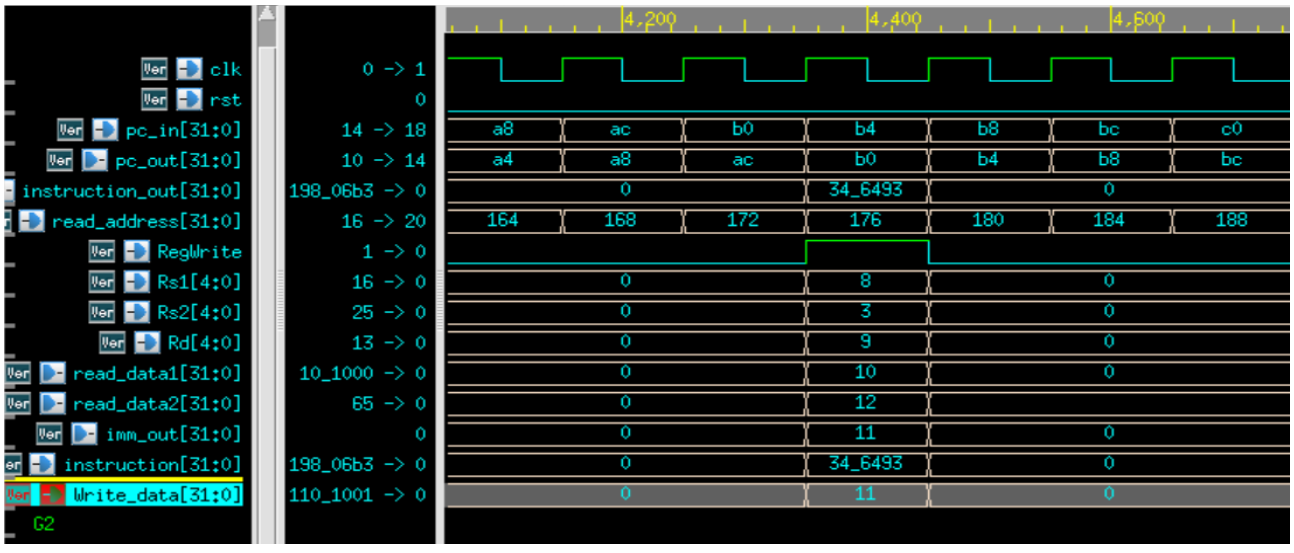


Figure 25 Simulation Waveform 5

9. RESULTS

Single Cycle 32-Bit RISC V (32 nm)

Report Cell

```
Total 20 cells
```

7094 overall cells are present.

Report Area

35860 µm²

```
Number of ports:                          386
Number of nets:                          7373
Number of cells:                         7094
Number of combinational cells:           4947
Number of sequential cells:              2145
Number of macros/black boxes:               0
Number of buf/inv:                        575
Number of references:                      34

Combinational area:              12126.226875
Buf/Inv area:                      913.139418
Noncombinational area:           14123.290827
Macro/Black Box area:                0.000000
Net Interconnect area:            9611.096582

Total cell area:                 26249.517702
Total area:                      35860.614284
```

Report_Timing

```
  clock clk (rise edge)                          1.00       1.00
  clock network delay (ideal)                    0.00       1.00
  clock uncertainty                             -0.30       0.70
  pc/pc_out_reg[31]/CLK (DFFX1_RVT)              0.00       0.70 r
  library setup time                            -0.10       0.60
  data required time                                       0.60
  ---------------------------------------------------------------
  data required time                                       0.60
  data arrival time                                       -2.65
  ---------------------------------------------------------------
  slack (VIOLATED)                                        -2.05
```

Report_Power

```
  Cell Internal Power  =   7.8724 mW   (99%)
  Net Switching Power  =  91.6947 uW    (1%)
                         ---------
Total Dynamic Power    =   7.9641 mW  (100%)

Cell Leakage Power     =  60.7370 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network   7.6665e+03           0.0000              0.0000          7.6665e+03  (  95.54%)  i
register          13.1465           4.8328           3.9829e+07            57.6775  (   0.72%)
sequential         5.3501           1.8135           5.5944e+05             7.7231  (   0.10%)
combinational    187.4500          85.0486           2.0349e+07           292.8462  (   3.65%)
--------------------------------------------------------------------------------------------------
Total          7.8725e+03 uW       91.6950 uW       6.0737e+07 pW       8.0248e+03 uW
```

Pipelined 32-Bit RISC V (32 nm)

Report Cell

```
Total 144 cells
```

31460 overall cells are present.

Report Area

114323 µm²

```
Number of ports:                        7692
Number of nets:                        37452
Number of cells:                       31460
Number of combinational cells:         27767
Number of sequential cells:             3602
Number of macros/black boxes:              0
Number of buf/inv:                      4223
Number of references:                     30

Combinational area:              60356.404861
Buf/Inv area:                     6885.777661
Noncombinational area:           23463.591375
Macro/Black Box area:                0.000000
Net Interconnect area:           30503.829453

Total cell area:                 83819.996236
Total area:                     114323.825689
```

Report_Timing

```
clock clk (rise edge)                                  1.00       1.00
clock network delay (ideal)                            0.00       1.00
clock uncertainty                                     -0.30       0.70
datamem/mem_reg[11][4]/CLK (DFFX1_RVT)                 0.00       0.70 r
library setup time                                    -0.09       0.61

data required time                                                0.61
--------------------------------------------------------------------------
data required time                                                0.61
data arrival time                                                -4.90
--------------------------------------------------------------------------
slack (VIOLATED)                                                 -4.28
```

Report_Power

```
  Cell Internal Power  =  12.3014 mW   (100%)
  Net Switching Power  =  24.9743 uW     (0%)
                         ---------
Total Dynamic Power    =  12.3264 mW   (100%)

Cell Leakage Power     = 133.8181 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network  1.2264e+04           0.0000              0.0000         1.2264e+04  (  98.42%)  i
register          10.9355           1.4775          6.2850e+07            75.2378  (   0.60%)
sequential         0.0000           0.0000          4.4758e+06             4.4758  (   0.04%)
combinational     26.5609          23.4966          6.6492e+07           116.5495  (   0.94%)
--------------------------------------------------------------------------------------------------
Total          1.2302e+04 uW       24.9742 uW       1.3382e+08 pW      1.2460e+04 uW
```

Pipelined 64 Bit RISC-V with Level-1 Caches (28 nm)

Report Cell

```
@genus:root: 27> sizeof_collection [get_cells -hier]
200001
```

200001 overall cells are present.

Report Area

348974 µm²

```
@genus:root: 28> report_area
------------------------------------------------------------------------------------------------------------------------------------------------------
Generated by:           Genus(TM) Synthesis Solution 25.12-s067_1
Generated on:           May 26 2026  02:24:32 pm
Module:                 riscv_core_top
Operating conditions:   ss28_0.86V_0.00V_m0.85V_0.85V_0.86V_0.00V_150C_SigCmax_2ey
Operating conditions:   ss28_0.86V_0.00V_m1.00V_1.00V_0.86V_0.00V_140C_SigCmax_2ey
Operating conditions:   ss28_1.10V_0.00V_m1.10V_1.10V_1.10V_0.00V_150C_SigCmax_2ey
Interconnect mode:      global
Area mode:              physical library
------------------------------------------------------------------------------------------------------------------------------------------------------

          Instance                                        Module                          Cell-Count  Cell-Area   Net-Area   Total-Area   %
Hierarchical Area
------------------------------------------------------------------------------------------------------------------------------------------------------
riscv_core_top                  NA                                                        200018  241589.570  107384.846  348974.416  100.00
  u_main_mem                    main_mem_MEM_DEPTH12_DATA_WIDTH64_ADDR_WIDTH64              3       4.896       0.243       5.139    0.00
  u_riscv_core_alu_decoder      riscv_core_alu_decoder                                     15       7.711       4.086      11.797    0.00
  u_riscv_core_dcache_top       riscv_core_dcache_top_BLOCK_OFFSET2_INDEX_WIDTH7_TAG_WIDTH52_CORE_DATA_WIDTH64_
                                WIDTH64_AXI_DATA_WIDTH256                                183778  219465.648   98520.345  317985.993   91.12
  u_amo_alu                     riscv_core_amo_alu                                        711     528.034     232.206     760.240    0.22
  u_dcache_memory               riscv_core_dcache_memory_TAG_WIDTH52                    172039  197229.362   91884.616  289113.979   82.82
  u_riscv_core_main_decoder     riscv_core_main_decoder_top                                81      41.126      17.650      58.776    0.02
  u_ub                          riscv_core_main_decoder                                    59      28.886      12.461      41.347    0.01
  u_riscv_core_mux2x1_stg1      riscv_core_mux2x1_XLEN64_2                                 21      11.628       4.702      16.330    0.00
  u_riscv_core_pcsr             riscv_core_pcsr                                            65      55.325       7.687      63.012    0.02
  u_riscv_core_pipe_alu_result_mem_wb  riscv_core_pipe_W_PIPE_BUS64_13                      1       0.612       0.000       0.612    0.00
  u_riscv_core_pipe_alucontrol_id_ex   riscv_core_pipe_W_PIPE_BUS4                         131     228.766      31.258     260.023    0.07
...
------------------------------------------------------------------------------------------------------------------------------------------------------
Total                                                                                   200018  241589.570  107384.846  348974.416  100.00
@genus:root: 29>
```

Report Timing

```
No setup violations found.

Hold violations
-------------------------------------------------------------
          Total    reg->reg    in->reg    reg->out    in->out
-------------------------------------------------------------
WNS      -0.055      -0.055      0.000       0.000      0.000
TNS    -917.438    -917.438      0.000       0.000      0.000
NUM       34159       34159          0           0          0
-------------------------------------------------------------
```

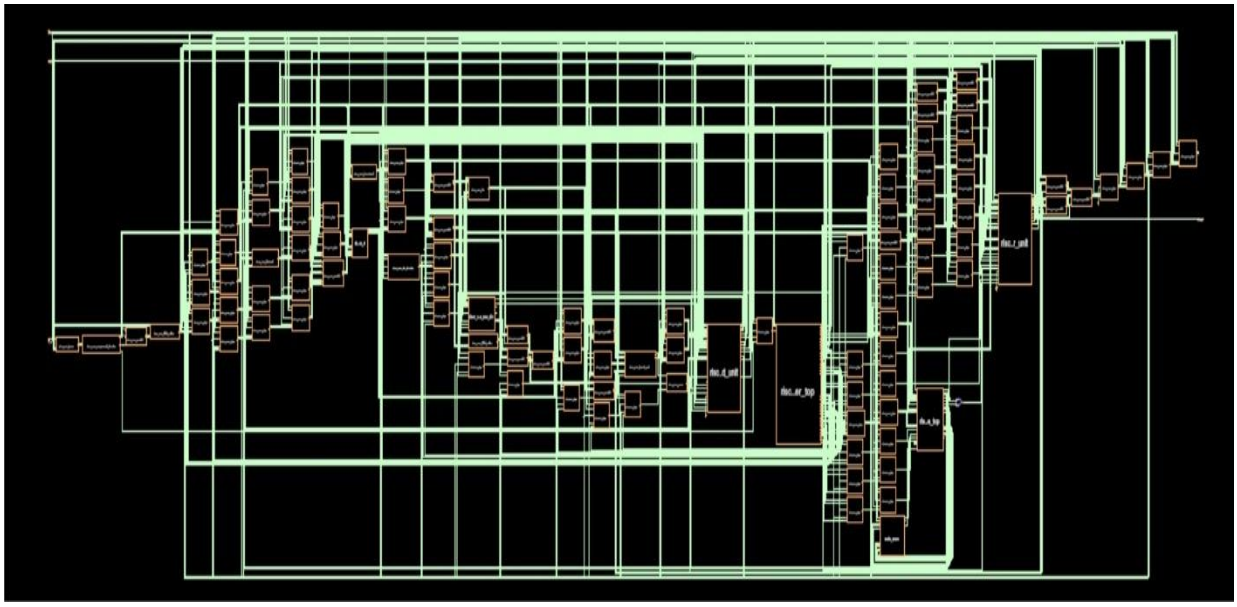


Figure 26 Schematics of 64-bit RISC V Pipelined.

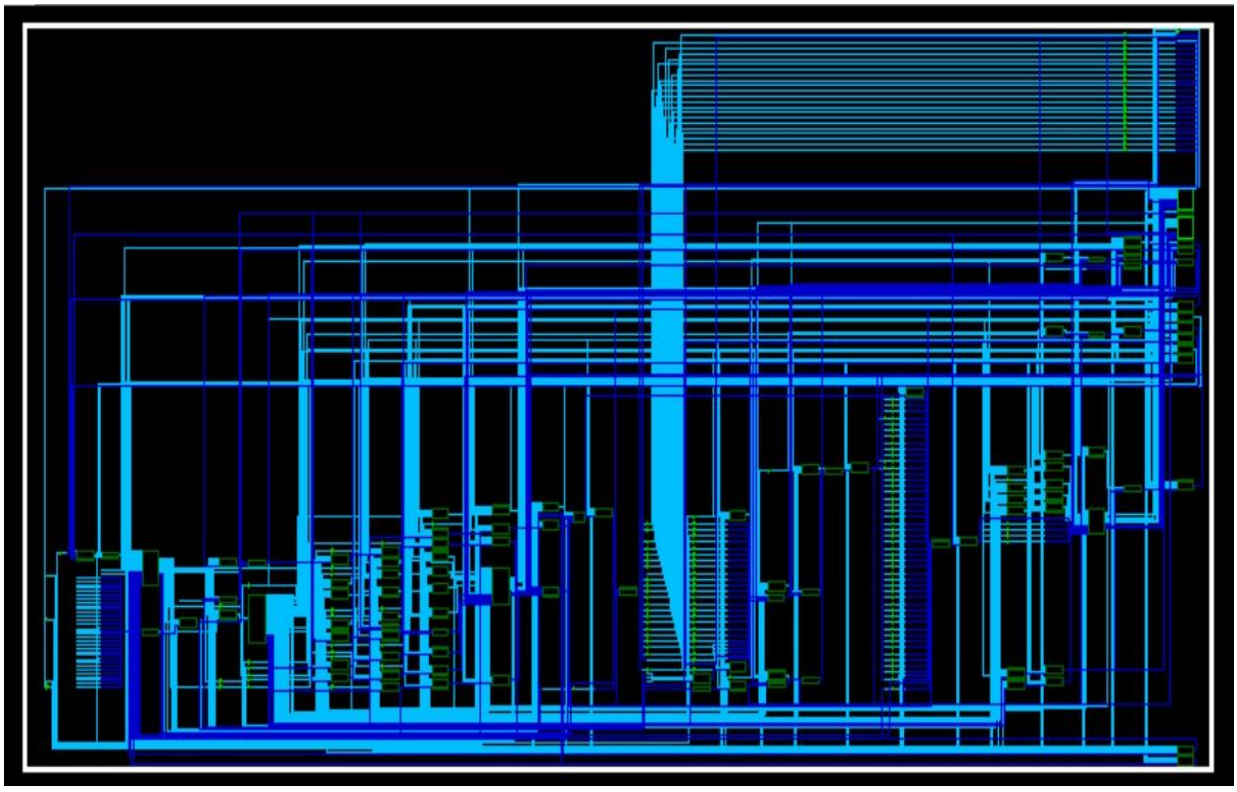


Figure 27. Synthesized 64-bit Pipelined RISC V (**28 nm**)

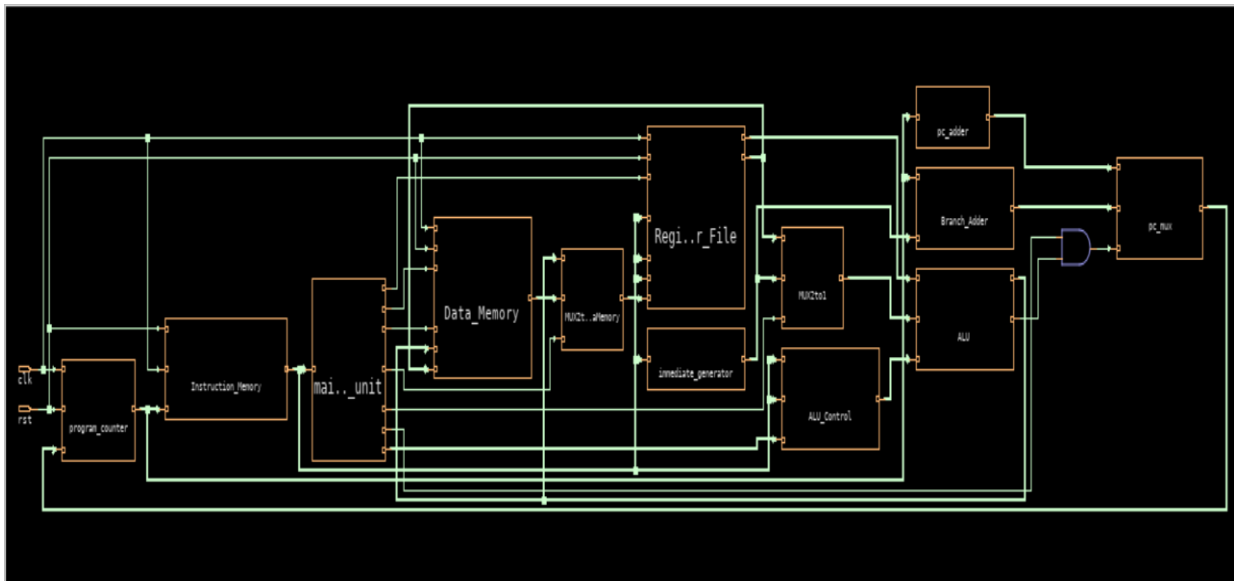


Figure 28. . Schematics of 32-bit Single Cycle RISC V

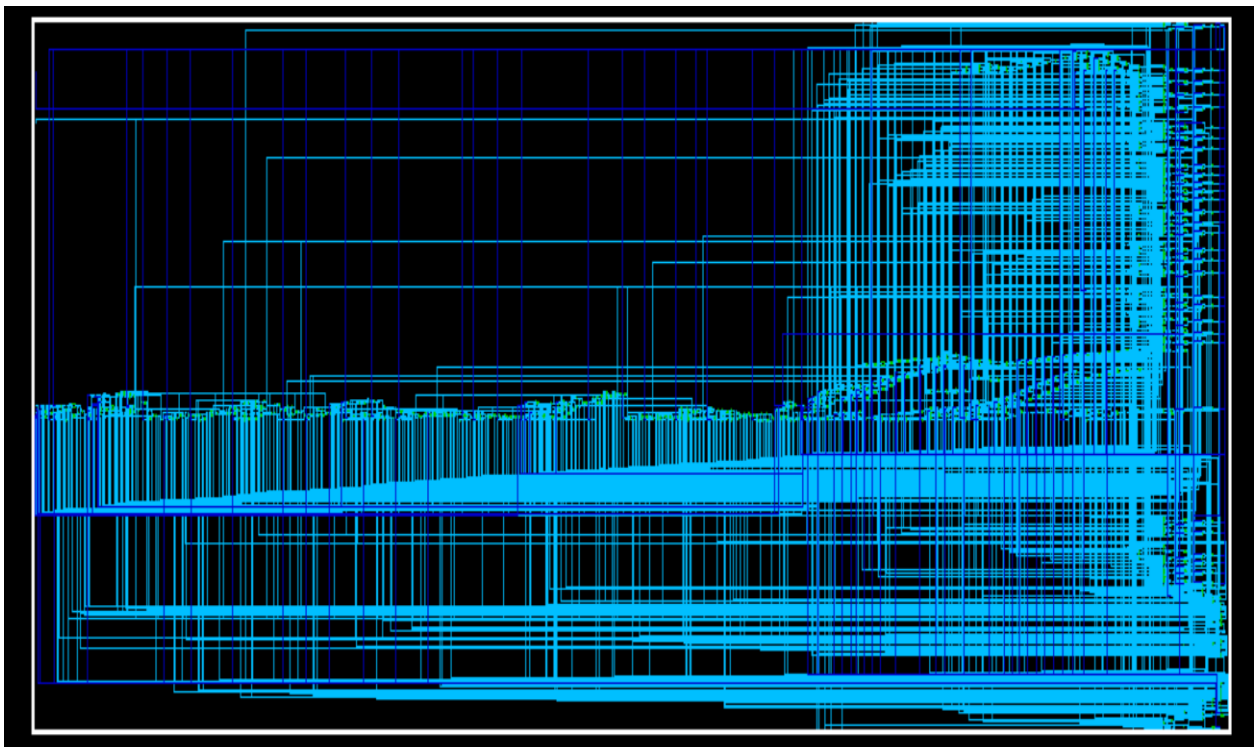


Figure 29. Synthesized 32-bit Single Cycle RISC V (**32nm**)

.

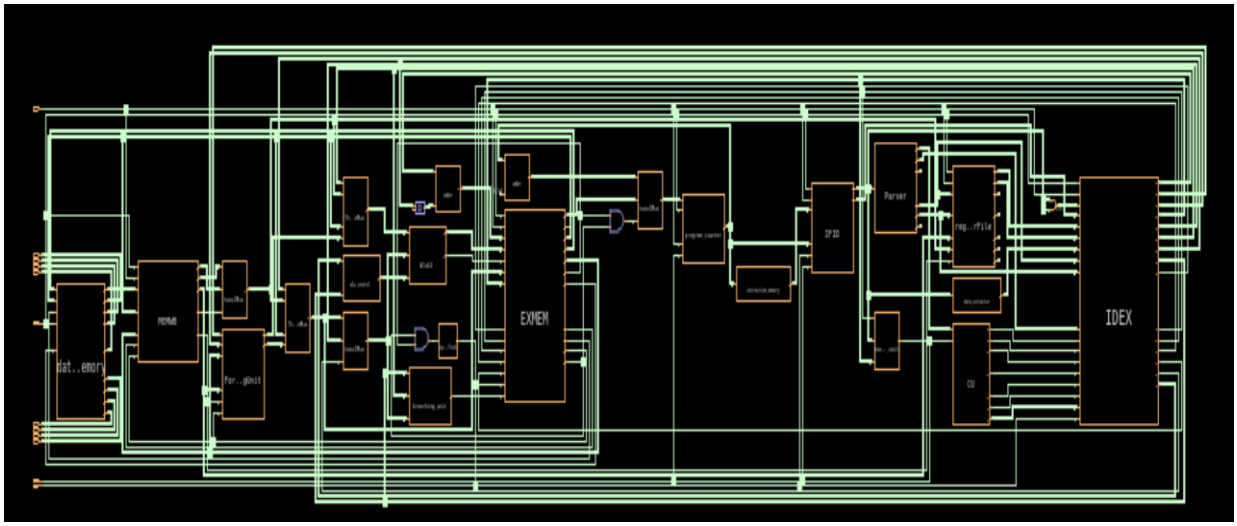


Figure 30. Schematics of 32-bit Pipelined RISC V.

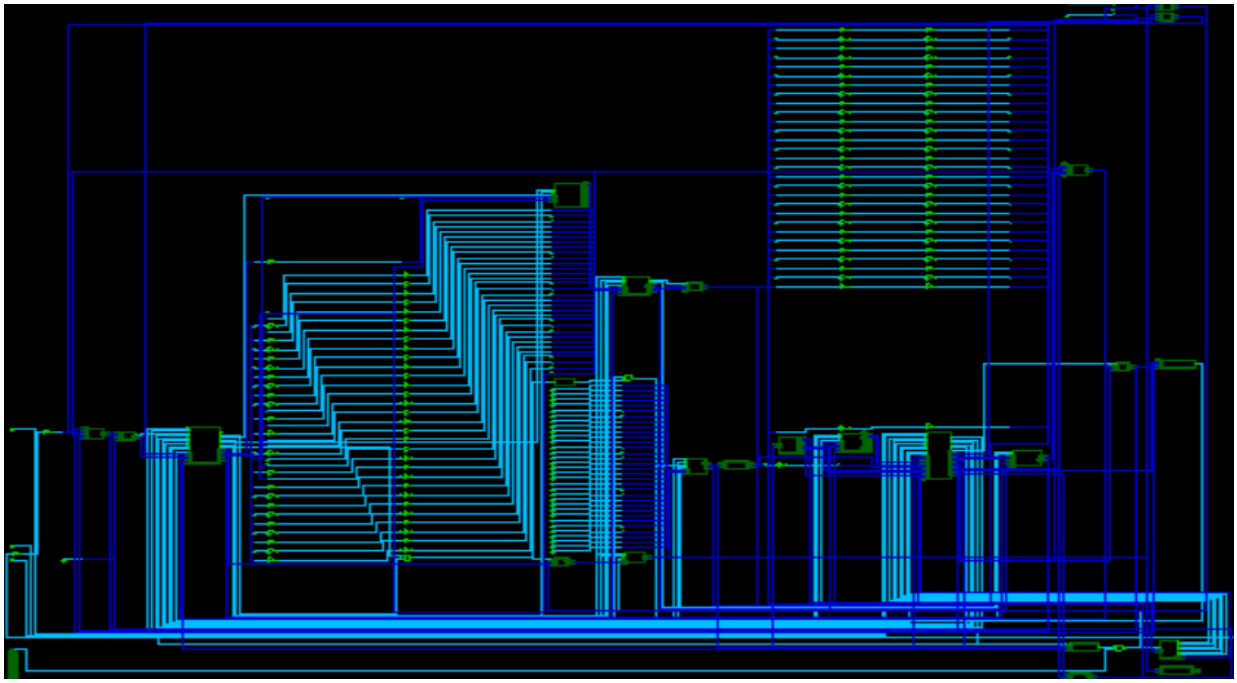


Figure 31. Synthesized 32-bit Pipelined RISC V (**32nm**)

10. CONCLUSION

In this work, we have been able to design and implement an extremely minimal yet functional RISC-V processor based on core concepts of theory. The minimalistic approach allowed us to create a very simple and easy-to-understand processor that can be effectively used for learning purposes.

The simplified instruction set was associated with minimized logic circuits that allowed us to make efficient use of hardware and perfectly aligned with our goal of designing a single-cycle processor. Notwithstanding, this minimalistic approach made the processor deliver promising results in Dhrystone tests.

The testing and verification processes have proven that the designed R1SC-V core is capable in implementing the required functionality. The successful design of the presented project lays down the grounds for the further continuation of research in which more enhancements, performance improvements, and inclusion in bigger system designs would be done.

Apart from design optimizations, the full RTL-to-GDSII flow of the presented core will be accomplished using Synopsys IC Compiler II (ICC2) on 32nm technology, which will cover all the necessary steps like placement and routing, finally resulting in the GDSII layout. Such development would increase the practicality of the design and gain invaluable experience in the field of digital backend design.

This project has made a considerable contribution to the study of computer architecture as it illustrates the potential of using RISC-V cores for developing a sophisticated system. To conclude, the output of this project clearly demonstrates that RISC-V is a feasible option in modern computing. Along with this I have also implemented the Physical Design Flow.

REFRENCES

1. P. Nair V. M. and L. V., "Design and Implementation of 5-Stage Pipelined RISC-V Processor on FPGA," *2024 28th International Symposium on VLSI Design and Test (VDAT)*, 2024.

2. L. Poli, S. Saha, X. Zhai, and K. D. McDonald-Maier, "Design and Implementation of a RISC V Processor on FPGA," *Proceedings of the 2021 17th International Conference on Mobility, Sensing and Networking (MSN)*, Exeter, United Kingdom, Dec. 2021

3. Digital Design and Computer Architecture, Book by Harris and Harris

4. Synopsys Tool Documentation (VCS, Verdi, Spyglass, Design Compiler)

5. YouTube RISC-V Instruction by Transistors to AI

6. RISC-V Instruction Set Manual – Volume I: User-Level ISA

7. R.-V. International, “The RISC-V Instruction Set Manual Volume II: Privileged Architecture,” 2019. [Online]

8. Ganguly, A., Chakraborty, A., Ambekar, A. A. and Mondal, H. K. "Power, Performance, and Area Optimisation of the RISC-V Processor" 2024 IEEE International Symposium on Smart Electronic Systems (iSES) 2024

9. K. P., P. S. and E. Prabhu. "A Power-Efficient Core Micro-Architecture Based on RISC-V Instruction Set Architecture" 2024

10. Li, Y., Yu, Z. and Ji, W. "Design and Performance Optimization of a RISC-V Processor with Five-stage Pipeline" 2025 IEEE 6th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT) 2025

11. Navik, A. P., Tiwari, S. K., Anand, V., Yadav, B., Park, J. and Sung, H. J. "RISC-V: Redefining the Future of Computing Architecture, Innovations, and Beyond" 2025

12. Patil, C. M., G. A, S. R, Yadav, R., S, A. H. and S. D., "RTL to GDSII: Design and Physical Implementation of a 32-bit RISC-V Floating-Point Co-Processor Using Verilog and Cadence Tools" 2025 5th International Conference on Intelligent Technologies (CONIT) 2025

10%	8%	5%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Delhi Technological University Student Paper	2%
2	dspace.dtu.ac.in:8080 Internet Source	2%
3	gist.github.com Internet Source	1%
4	researchr.org Internet Source	1%
5	fdocuments.in Internet Source	<1%
6	www.ieo.es Internet Source	<1%
7	www.fastercapital.com Internet Source	<1%
8	Jacobs, Sean. "Design of a 32-Bit 4-Phase Asynchronous RISC-V Processor", Rochester Institute of Technology Publication	<1%
9	www.mdpi.com Internet Source	<1%
10	Bhagdikar, Nikhil. "Design and Optimization of a CGRA Functional Unit", Stanford University, 2025 Publication	<1%

12	ijmrset.com Internet Source	<1%
13	www.ijirset.com Internet Source	<1%
14	Shidong Shen, Shijie Chen, Yicheng Liu, Lijun Zhang, Fu Song, Zhilin Wu. " RVFormal: Formal verification of RISC-V processor Chisel designs ", Journal of Systems Architecture, 2026 Publication	<1%
15	Thakar, Bhavin. "Secure RISCV Design for Side-Channel Evaluation Platform", The University of North Carolina at Charlotte, 2021 Publication	<1%
16	"ICT: Applications and Social Interfaces", Springer Science and Business Media LLC, 2025 Publication	<1%
17	Sarah L. Harris, David Harris. "Microarchitecture", Elsevier BV, 2022 Publication	<1%
18	J. Perez Acle, R. Cantoro, E. Sanchez, M. Sonza Reorda, G. Squillero. "Observability solutions for in-field functional test of processor-based systems: A survey and quantitative test case evaluation", Microprocessors and Microsystems, 2016 Publication	<1%
19	pure.bit.edu.cn Internet Source	<1%

KARTIKE

Email-id:kartikeverma2001@gmail.com
Mobile No.: **8400753685**
LinkedIn Profile: https://www.linkedin.com/in/kartike-v/

EDUCATION

Year	Degree	Institute	CGPA/%
2024-2026	M.Tech(VLSI & Emb Sys)	Delhi Technological University (DTU), New Delhi	7.80 CGPA
2020-2024	B.Tech(ECE)	JSS Academy of Technical Education, Noida	7.70 CGPA
2019	12th CBSE	Mahatma Hansraj Modern School, Jhansi	86.60 %
2017	10th CBSE	St Joseph's Eng Med School, Mahoba	9.2 CGPA

PROJECTS

- **Synthesized P5E an automotive SoC (STMicroelectronics)** *Oct 2025*

  Tool used: **Genus, Innovus, Conformal**
  - Run the **synthesis** flow at block level using Genus for **ADC**, **NPU** blocks along with low power optimization in the synthesis.
  - Executed LEC and Handoff Checks to ensure functional equivalence and get hands in TCL Scripting for automation of synthesis and STA runs.

- **Designed a 32-bit RISC V Pipelined Processor** *Jan 2025*

  Tool used: **Synopsys (VCS, Verdi, Spyglass, Design Compiler, ICC2)**
  - Designed a 32-bit RISC-V Processor using verilog supporting a subset of the RV32I ISA.
  - Implemented register, ALU, instruction fetch, decode, execute, and memory access stages in a single-cycle architecture.
  - Implemented the design in Verilog, synthesized in design compiler and verified using Verdi and VCS with testbench and done the **PNR** using **ICC2**

- **Physical Implementation of Hierarchical 8-bit RCA Adder Design** *Oct 2024*

  Tool used: **Synopsys (Cadence Virtuoso, Synopsys ICC2)**
  - Designed 8-bit Ripple Carry Adder using cadence virtuoso.
  - Performed simulations to verify functionality PVT corner analysis to ensure reliability under various conditions.
  - Utilized Synopsys tools to complete the **RTL-to-GDSII flow**, including **Design Compiler** for synthesis, **IC Compiler II** for place and route, and Prime Time for timing analysis at 32nm technology node.

- **Implementation of Synchronous and Asynchronous FIFO in Verilog** *Sep 2024*
  - Developed and analyzed Asynchronous and Synchronous FIFO designs in Verilog, focusing on efficiency and detailed examination of design aspects.
  - Conducted in-depth analysis of various modules in the asynchronous FIFO design, demonstrating expertise in creating efficient solutions for data transfer.

TECHNICAL SKILLS

- **Coding Languages**: C++
- **Operating System:** Linux
- **HDL Language:** Verilog
- **EDA Tools**: Synopsys (Custom Compiler, Verdi, VCS, Spyglass, ICC2, Design Compiler, Prime Time), Cadence Virtuoso, Vivado, LT Spice, Genus, Conformal, Cerebrus
- **Scripting Languages**: Shell, Tcl
- **Modules**: Digital IC Design, Analog IC Design, Physical Design, Low Power VLSI , STA, Computer Architecture (RISC V), Semiconductor Fabrication

INTERNSHIP AND WORK EXPERIENCE

- **STMicroelectronics (July 2025 – present)**
  - **Synthesis and STA intern** where i get hands-on experience in **RTL-to-gate synthesis and STA**, including SDC constraint development, multi-corner multi-mode timing analysis, setup/hold debug, ECO timing fixes, logic equivalence check (LEC), and synthesis-to-PD handoff validation,Cerebrus, I/O delays, false paths, and multicycle paths for accurate timing analysis.

- Worked on **power-aware design and flow automation**, covering low-power optimization (clock gating, UPF), TCL scripting, and gained good understanding of architecture flow of automotive designs.

- **VLSI EXPERT (July 2024 - Feb 2024)**
  - **Physical Design Trainee** which helped in understanding from basic MOSFET operation, CMOS inverter to Complex gates Design and operation and familiarity with process corners and mismatch variations, temperature variations and corner analysis and **RTL to GDS** flow with familiarity in automating and file handling in Linux OS.

EXTRA CURRICULAR ACTIVITIES AND ACHIEVEMENTS

- Selected for **Indo-Taiwan** International Workshop on Semiconductor Fabrication
- Ranked at 60th position among top 100 team in **DIR-V** Challenge
- Qualified GATE 2024, 2025