From Gates to OS: Design and Implementation of a 64-bit RISC-V IMAC Core with Privilege Modes and Level-1 Caches Capable of Running OS

By

Ahmed Hamdi Ibrahim
Ali Khaled Mohamed
Aly Mohamed Elruby
Basmala Eldabaa Fayez
Hythem Ahmed Shaban
Mohamed Maged Abd-Elhaleem
Mohamed Morsi Abd-Elsamee
Mohamed Mohamed Tariq
Nsreen Nashaat Abdo
Ziad Taha Mohamed Taha

A Thesis Submitted to
Faculty of Engineering, Alexandria University
In Partial Fulfilment of the Requirements
for the Degree of
BACHELOR OF SCIENCE
In
Electronics and Communications Engineering

Supervised By

Prof. Dr. Mohammed Rizq
Prof. Dr. Yasser Hanafy

Mentored By

Si-Vision

ACKNOWLEDGMENTS

We extend our gratitude to our supervisors, Prof. Dr. Mohamed Rizq and Prof. Dr. Yasser Hanafy, for their unwavering guidance and expertise throughout this project. Their mentorship and encouragement have been invaluable. We would also like to thank Eng. Mohamed Nagi and his team from Si-vision, our mentoring company. Their practical advice contributed to our project. Lastly, to our families: your unwavering support sustained us during this academic journey. Thank you for believing in us.

Abstract

The objective of the thesis is to implement a core with instruction set that can run an Operating System such as LINUX and assembly programs that require a core with certain instructions. We chose the RISC-V open-source instruction set due to its variety in instructions and the fact that RISC-V is the future in processors nowadays. That's why we started researching for the needed instruction set and modes to implement in our core and ended by figuring out the needed extensions which are: the basic instruction set for Integer (I), the compressed instruction set extension (C), the multiplication and division set extension (M) and atomic instruction set extension (A) to end up with IMAC core. In addition to the instruction set, we also implemented Privilege modes for our core which are Machine and Supervisor modes which are crucial for the needs of running an Operating System on our core. And for the purpose of Completion of the System, we implemented 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 and added some hardware features to increase the performance of our core as Dynamic Branch Prediction. At the end of our implementation, our core is capable, with some modifications and additional modules, of running OS like LINUX and handling interrupts. As a Conclusion, our core is a 64-bit RISC-V IMAC core with Privilege modes and Level 1 Caches.

Table of Contents

List of Figures

List of Tables

1. Introduction

This chapter introduces the basic knowledge and brief background of topics related to this thesis. It also describes the objective behind this thesis and a map for this documentation to make it easier to navigate through different topics covered by it.

1.1. Motivation

With the intent in mind of implementing a core with an instruction set that enables the running of Operating Systems such as LINUX and assembly programs the selection 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.

Privilege modes (machine mode and supervisor mode) as it provides the necessary framework for security, stability, resource management, interrupt handling, and memory protection. By enforcing different levels of access and control, privilege modes ensure that the OS can manage system resources effectively while protecting the system from potential threats and ensuring reliable operation.

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 and added some hardware features to increase the performance of our core as Dynamic Branch Prediction.

1.2. History

A group of academics began it all at the University of California, Berkley in 2010 as an alternative to ARM and x86 architecture. There are numerous RISC-V variations available for advanced computing and educational purposes. It was made free and open source. The first two generations had an impact on the SPARC processor; this is the fifth generation. The ISA specification was released in 2011 and the RISC-V foundation was established in late 2015 to oversee RISC-V activities.

1.3. Thesis Objectives

The primary objective of this thesis is to design and implement a RISC-V core processor using SystemVerilog that includes a cached memory subsystem and supports privileged modes. The specific goals of this research are as follows:

1. Design a RISC-V Core Processor:
    - Develop a functional RISC-V core processor adhering to the RISC-V ISA specifications.
    - Implement key architectural components including the instruction fetch unit, decode unit, execution unit, memory access unit, and write-back unit.
2. Incorporate Cached Memory:
    - Design and integrate a cached memory subsystem to improve processor performance.
    - Implement both instruction and data caches with appropriate cache coherence and consistency mechanisms.
    - Evaluate different cache architectures and configurations to determine the optimal design for the processor.
3. Support for Privileged Modes:
    - Implement the RISC-V privileged architecture to support multiple privilege levels (user mode, supervisor mode, machine mode).
    - Ensure proper handling of exceptions, interrupts, and traps.
    - Develop and integrate a memory management unit (MMU) to support virtual memory and address translation in privileged modes.
4. Verification and Testing:
    - Develop a comprehensive verification plan to validate the functionality and performance of the designed RISC-V core processor.
    - Use simulation and hardware description language (HDL) testbenches to verify the correctness of the design.
    - Conduct performance analysis and benchmarking to evaluate the efficiency and effectiveness of the cached memory and privileged mode implementations.
5. Documentation and Analysis:
    - Provide detailed documentation of the design process, including architectural decisions, design trade-offs, and implementation details.

- Analyze the results of verification and performance testing to identify areas for improvement and future research.

By achieving these objectives, this thesis aims to contribute to the advancement of RISC-V processor design, particularly in the areas of cached memory integration and privileged mode support, using SystemVerilog as the hardware description language.

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.

Our 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. We covered every stage of the design cycle, which included equivalency verification, placement and routing (PnR), floorplanning, synthesis, and static time analysis (STA).

1.4. What is RISC-V

RISC-V (pronounced "risk-five") is an open-standard instruction set architecture (ISA) based on the established reduced instruction set computing (RISC) principles. Unlike most other ISAs, RISC-V is provided under open-source licenses, which means that anyone can design, manufacture, and sell RISC-V chips and software.

Here are some key points about RISC-V:

1. Open Source: The RISC-V ISA is available under open-source licenses, which encourages innovation and collaboration. It provides freedom from the proprietary licensing constraints imposed by other ISAs.
2. Simplicity and Flexibility: RISC-V is designed to be simple and flexible, making it suitable for a wide range of applications, from small embedded systems to large supercomputers.
3. Modular Design: RISC-V's modular design allows for extensions, meaning that specific features can be added to the base ISA as needed. This modularity helps in creating tailored solutions for different applications.
4. Scalability: RISC-V supports a wide range of applications by allowing the addition of custom instructions and extensions. This makes it highly scalable across various computing domains.
5. Community-Driven: The development of RISC-V is driven by a global community of contributors, which includes academic institutions, research organizations, and commercial enterprises. This community-driven approach helps in rapidly evolving the architecture and addressing a wide array of use cases.
6. Adoption: RISC-V has gained significant traction in the industry, with many companies and organizations developing RISC-V-based processors, tools, and software. It is increasingly being adopted in academia for research and education as well.

7. Comparisons with Other ISAs: While traditional ISAs like x86 and ARM are widely used, they come with licensing fees and restrictions. RISC-V offers a competitive alternative by being free of such constraints, potentially reducing costs and encouraging broader adoption.

RISC-V's open nature and growing ecosystem make it a promising ISA for the future of computing across various sectors.

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 subset variant of the RV32I base (support small microcontrollers).
5. RV128I (future work).

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 and divide extension.
2. “A”: Atomic memory operations (AMO) (read, modify, and write memory).
3. “F”: Single-precision floating-point extension.
4. “D”: Double-precision floating-point extension.
5. “C”: Compressed instruction extension provides narrower 16-bit forms of common instructions.
6. “G”: (“IMAFD”) General-purpose ISA which contains all the above extension.

Comparing the RISC-V base ISA with other ISAs, First, there are only six formats in the base ISA and all instructions are 32 bits long, which simplifies instruction decoding. ARM-32 and particularly x86-32 have numerous formats, which make decoding expensive in low-end implementations and a performance challenge for medium and high-end processor designs. Second, RISC-V instructions offer three register operands, rather than having one field shared for source and destination, as with x86-32. When an operation naturally has three distinct operands, but the ISA provides only a two- operand instruction, the compiler or assembly language programmer must use an extra move instruction to preserve the destination operand. Third, in RISC-V the specifiers of the registers to be read and written are always in the same location in all instructions, which means the register accesses can begin before decoding the instruction. Many other ISAs reuse a field as a source in some instructions and as a destination in others (e.g., ARM-32 and MIPS-32), which forces addition of extra hardware to be placed in a potentially time-critical path to select the proper field. Fourth, the immediate fields in these formats are

always sign extended, and the sign bit is always in the most significant bit of the instruction. This decision means sign extension of the immediate, which may also be on a critical timing path, can proceed before decoding the instruction.

With full software support, including a general-purpose compiler. RISC-V has 32 (or 16 in the embedded variant) integer registers, and, when the floating-point extension is implemented, separate 32 floating-point registers. Except for memory access instructions, instructions address only registers. Like many RISC designs, RISC-V is a load–store architecture: instructions address only registers, with load and store instructions conveying to and from memory. RISC-V segregates math into a minimal set of integer instructions with add, subtract, shift, bitwise logic and comparing-branches. The integer multiplication instructions include signed and unsigned multiply and divide.

2. Implementation of Base Integer Instruction Set (RV64I)

The initial objective of our project was to design a 5-stage pipelined micro-architecture of the integer instructions found in the RV64I Base Integer Instruction Set which is shown in Table 1.

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 1. Base RV64I ISA

2.1. RV64I Design Process and Pipeline Stages

Our microarchitecture is divided into two interacting parts: the datapath and the control unit. The datapath operates on double-words of data. It contains structures such as memories, registers, ALUs, and multiplexers. We are implementing the 64-bit RISC-V (RV64I) architecture, so we use a 64-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

We started the design with the state elements such as memories, program counter, and registers then we added combinational logic blocks between these state elements to compute the new state.

Firstly, we designed single cycle microarchitecture then we applied pipelining and added hazard unit to handle dependencies between simultaneously executing instructions.

Our RV64I microarchitecture is shown in Figure 1.

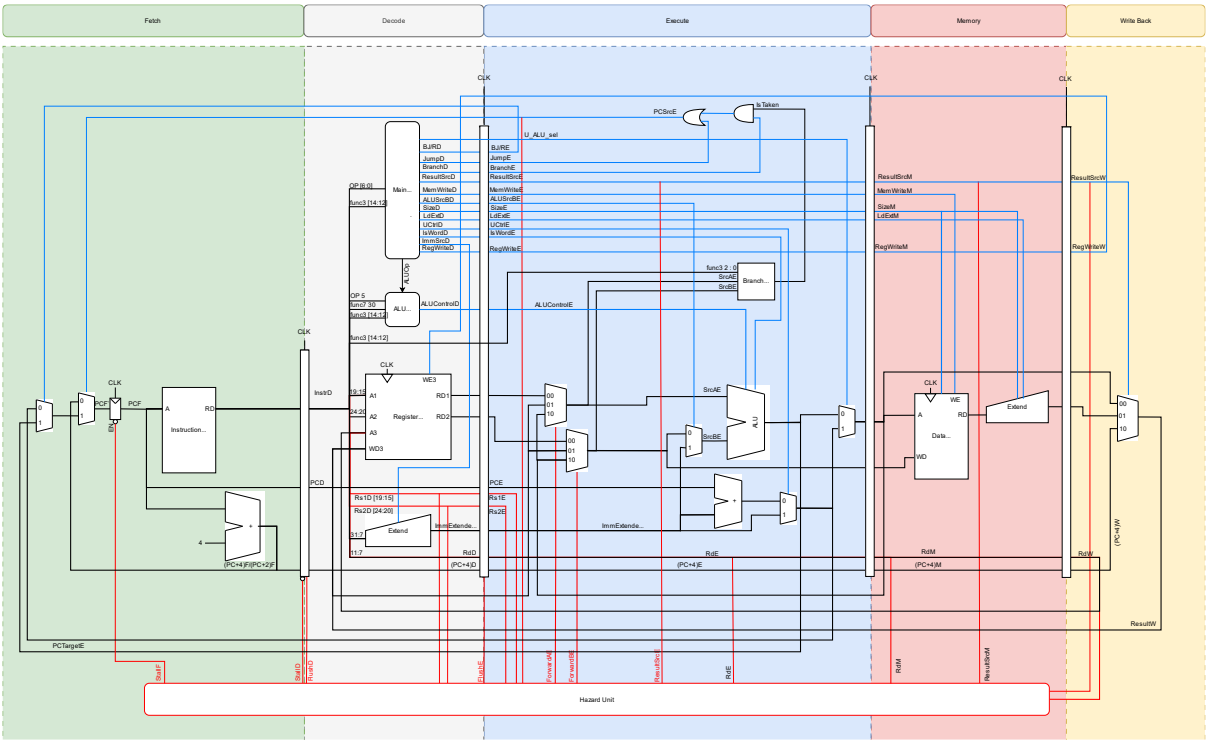

Figure 1. RV64I Microarchitecture

2.1.1. 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.

2.1.1.1. Instruction Cache

Instruction-Cache will be discussed in section 5.

2.1.2. 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.

2.1.2.1. Register File

For RV64I, there are 32 integer registers, from x0 to x31, each of width 64-bit. The usage of each register is as shown in Table 2.

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

Table 2. RISC-V Integer Registers

The block diagram of the register file is shown in Figure 2.

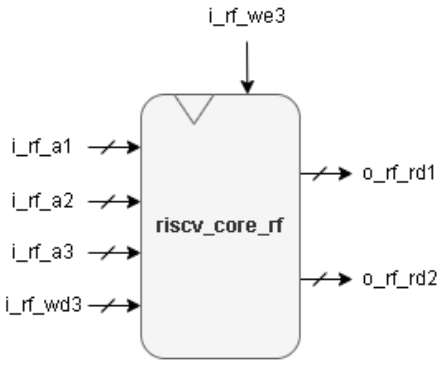

Figure 2. Register File Block Diagram

The block interface of the register file is shown in Table 3.

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	64	Data to be written to rd1
i_rf_we3	Input	1	Write Enable
o_rf_rd1	Output	64	Data read from rs1
o_rf_rd2	Output	64	Data read from rs2

Table 3. Block Interface of Register File

2.1.2.2. 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 3.

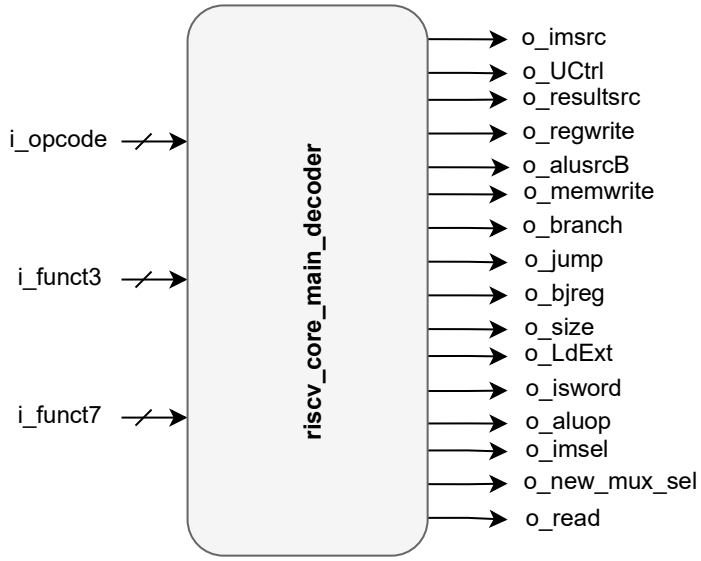

Figure 3. Main Decoder Block Diagram

The block interface of the main decoder is shown in Table 4.

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 4. Block Interface of Main Decoder

The function table of the main decoder can be found in the spreadsheet directed by this link:

https://docs.google.com/spreadsheets/d/1-5eVyOpvU7N59lBgBsmqMkrRyMExImC9KKijjkCBRag/edit?usp=sharing

2.1.2.3. 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 4.

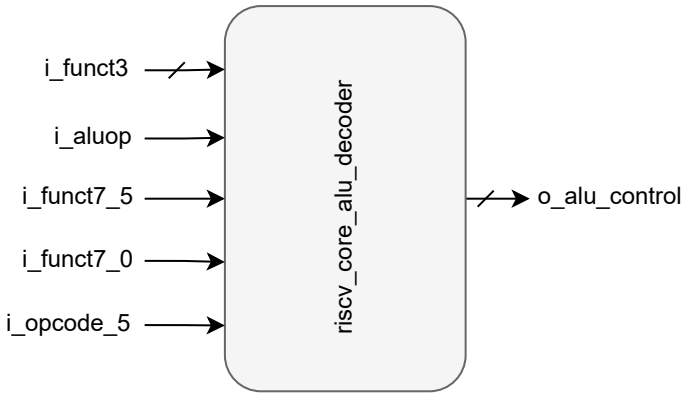

Figure 4. ALU Decoder Block Diagram

The block interface of the ALU decoder is shown in Table 5.

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

Table 5. The Block Interface of ALU Decoder

The function table of the ALU decoder can be found in the spreadsheet directed by this link:

https://docs.google.com/spreadsheets/d/1-5eVyOpvU7N59lBgBsmqMkrRyMExImC9KKijjkCBRag/edit?usp=sharing

2.1.2.4. Immediate Extend

The immediate extend extends immediate values to the appropriate size for processing. Since RV64I is a 64-bit architecture, immediate values within instructions are often shorter than 64 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 64 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 5.

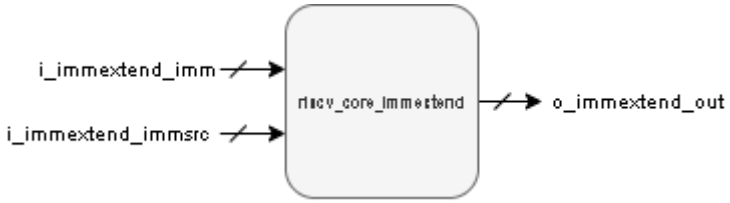

Figure 5. Immediate Extend Block Diagram

The block interface of the immediate extend is shown in Table 6.

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	64	Extended output

Table 6. Block Interface of Immediate Extend

The function table of the immediate extend is shown in Table 7.

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}}}

Table 7. Function Table of Immediate Extend

2.1.3. 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.

2.1.3.1. 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 6.

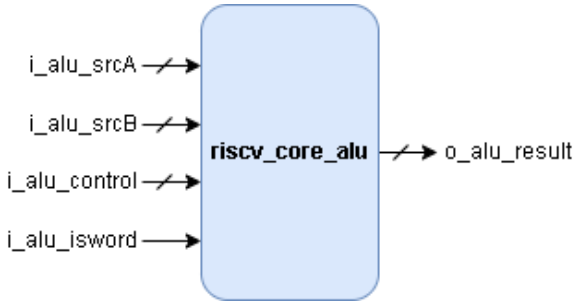

Figure 6. ALU Block Diagram

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

Port Name	Direction	Width	Description
i_alu_srcA	Input	64	Operand 1
i_alu_srcB	Input	64	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	64	ALU result

Table 8. Block Interface of ALU

The function table of the ALU is shown in Table 9.

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)

Table 9. Function Table of ALU

2.1.3.2. Branch Unit

In the RV64I 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 7.

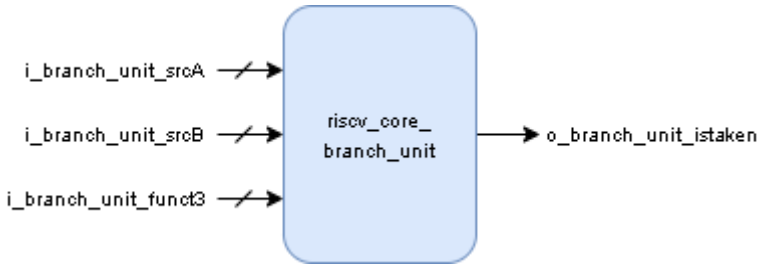

Figure 7. Branch Unit Block Diagram

The block interface of the Branch Unit is shown in Table 10.

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 10. Block Interface of Branch Unit

The function table of the Branch Unit is shown in Table 11.

i_branch_unit_funct3	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)

Table 11. Function Table of Branch Unit

2.1.4. 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 64-bit value from memory into rd.
  - *LW* instruction loads a 32-bit value from memory, then sign-extends to 64-bits before storing in rd.
  - *LH* loads a 16-bit value from memory, then sign-extends to 64-bits before storing in rd.

- *LHU* loads a 16-bit value from memory but then zero extends to 64-bits before storing in rd.
- *LB* and *LBU* are defined analogously for 8-bit values.

- Store Instructions:
  - *SD* instruction stores 64-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.

2.1.4.1. Data Cache

Data Cache will be discussed in section 5.

2.1.4.2. Load Extend

The load extend is responsible for extending the loaded value from memory to 64-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 8.

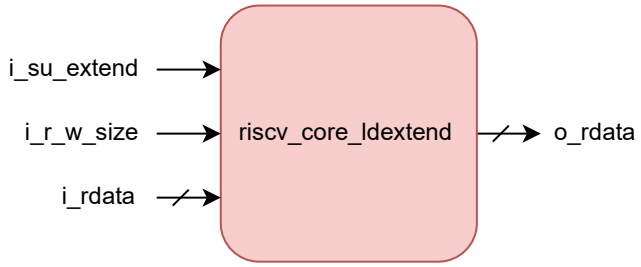

Figure 8. Load Extend Block Diagram

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 12. Block Interface of Load Extend

The function table of the load extend is shown in Table 13

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 13. Function Table of Load Extend

2.1.5. 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 14.

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 14. Function Table of Writeback MUX

2.1.6. Hazard Unit

The hazard unit handles the control and data hazards. A data hazard occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A control hazard occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. we enhanced the pipelined processor with a Hazard Unit that detects hazards and handles them appropriately so that the processor executes the program correctly.

Raw data hazards occur when an instruction depends on a result, of another instruction, that has not yet been written into the register file. Data hazards can be resolved by forwarding if the result is computed soon enough; otherwise, they require stalling the pipeline until the result is available.

Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction must be fetched. Control hazards are solved by stalling the pipeline until the decision is made or by predicting which instruction should be fetched and flushing the pipeline if the prediction is later determined to be wrong. Moving the decision as early as possible minimizes the number of instructions that are flushed on a misprediction.

The block diagram of the hazard unit is shown in Figure 9.

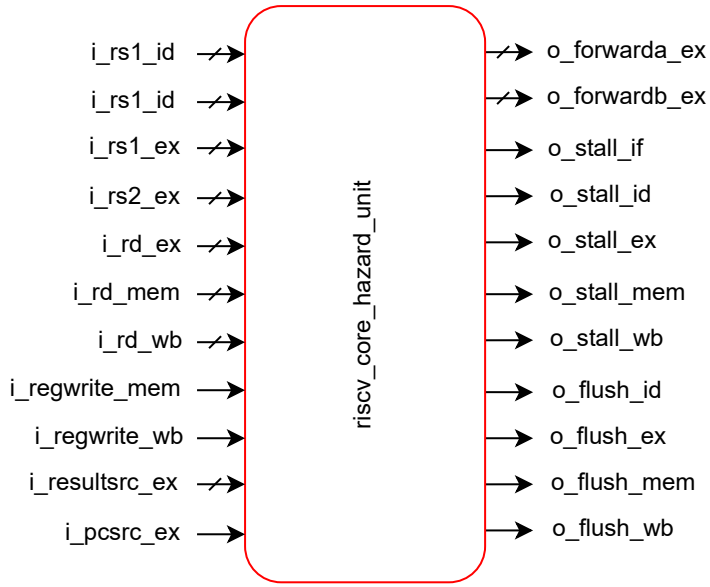

Figure 9. Hazard Unit Block Diagram

The block interface of the hazard unit is shown in Table 15.

Port Name	Direction	Width	Description
i_hazard_unit_rs1_id	Input	5	RV64I 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	RV64I 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	

Table 15. Block Interface of Hazard Unit

3. 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.

RV64M ISA introduces a total of thirteen multiplication and division instructions as shown in Table 16.

Instruction	Format	Opcode	Funct3	Funct7	Description
mul	R	0110011	0x0	0x01	rd = (rs1 * rs2)[63:0]
mulh	R	0110011	0x1	0x01	rd = (rs1 * rs2)[127:64]
mulhsu	R	0110011	0x2	0x01	rd = (rs1 * rs2)[127:64]
mulhu	R	0110011	0x3	0x01	rd = (rs1 * rs2)[127:64]
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)

Table 16. M-Extension for Integer Multiplication and Division Instruction.

Figure 10 shows the microarchitecture of M-Extension.

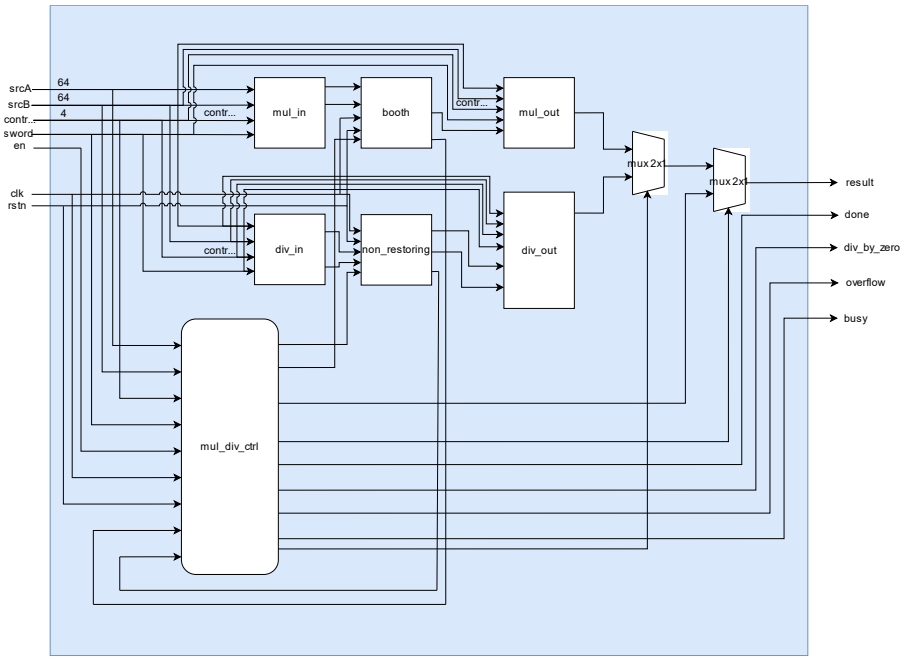

Figure 10. Microarchitecture of M-Extension.

3.1. Multiplier

3.1.1. 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 64 (for 64-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.

3.1.2. Multiplier Design

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

3.1.2.1. 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 11.

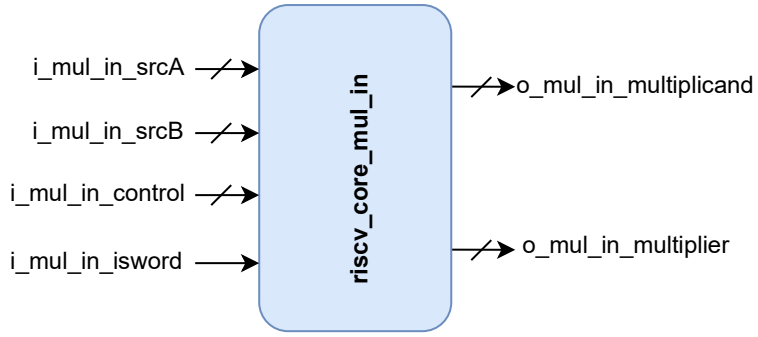

Figure 11. mul_in Block Diagram

The block interface of the mul_in block is shown in Table 17.

Port Name	Direction	Width	Description
i_mul_in_srcA	Input	64	Operand 1
i_mul_in_srcB	Input	64	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	64	Multiplicand
o_mul_in_multiplier	Output	64	Multiplier

Table 17. Block Interface of mul_in Block

3.1.2.2. Booth Block

This block is responsible for performing the multiplication.

The block diagram of the booth block is shown in Figure 12.

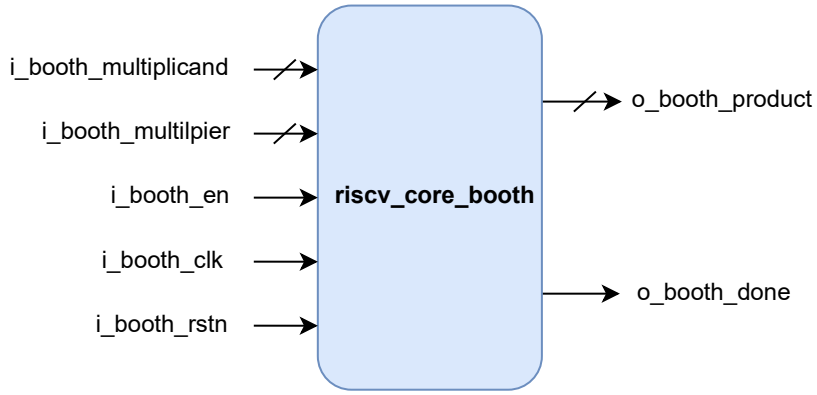

Figure 12. Booth Block Diagram

The block interface of the Booth is shown in Table 18.

Port Name	Direction	Width	Description
i_booth_multiplicand	Input	64	Multiplicand
i_booth_multilpier	Input	64	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

Table 18. Block Interface of Booth Block

3.1.2.3. 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 13.

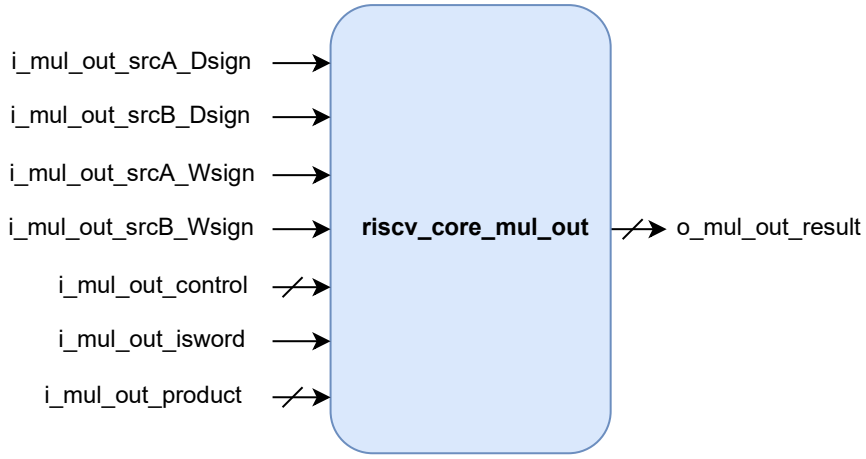

Figure 13. mul_out Block Diagram

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

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	64	Product result of booth block
o_mul_out_result	Output	64	Multiplication result

Table 19. Block Interface of mul_out Block.

3.2. Divider

3.2.1. 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 14.

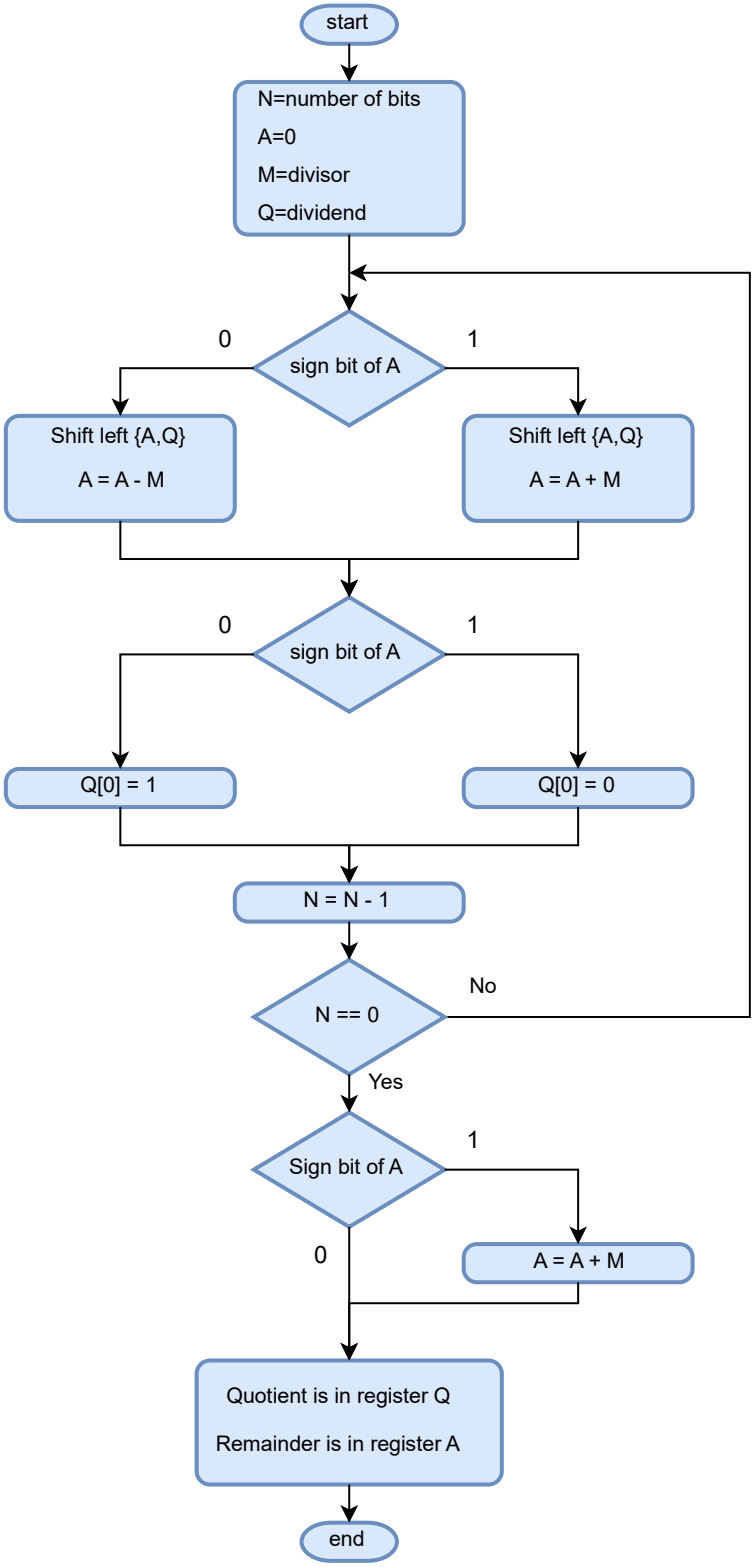

Figure 14. Non-Restoring Flow Chart

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.

3.2.2. Divider Design

3.2.2.1. 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 15.

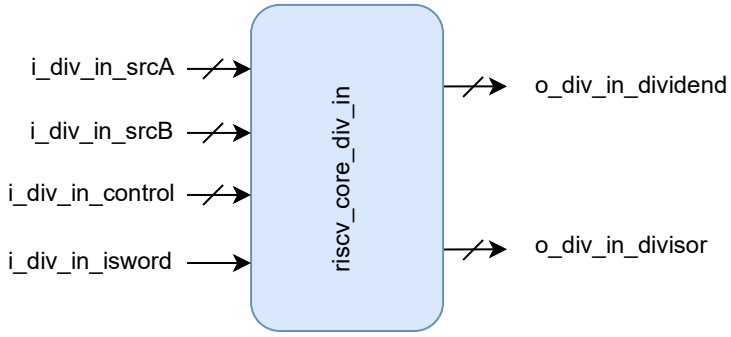

Figure 15. div_in Block Diagram.

The block interface of the div_in is shown in Table 20.

Port Name	Direction	Width	Description
i_div_in_srcA	Input	64	Operand 1

i_div_in_srcB	Input	64	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	64	Dividend
o_div_in_divisor	Output	64	Divisor

Table 20. Block Interface of div_in Block.

3.2.2.2. non_restoring Block

This block is responsible for performing the division.

The block diagram of non-restoring is shown in Figure 16.

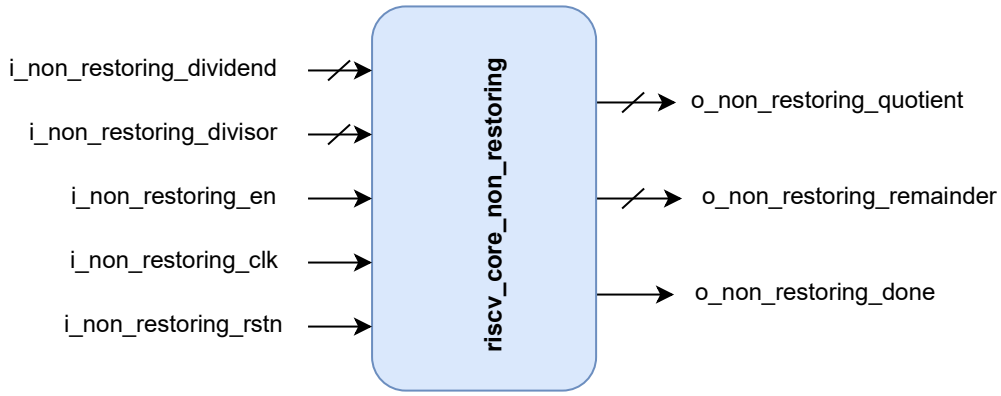

Figure 16. non_restoring Block Diagram

The block interface of the non-restoring is shown in Table 21.

Port Name	Direction	Width	Description
i_non_restoring_dividend	Input	64	Dividend
i_non_restoring_divisor	Input	64	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	64	Quotient
o_non_restoring_remainder	Output	64	Remainder

Table 21. Block Interface of non_restoring Block.

3.2.2.3. 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 17.

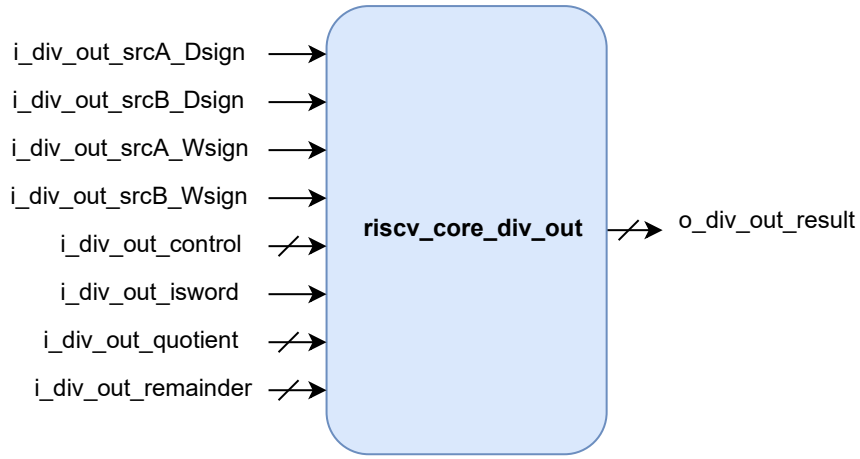

Figure 17. div_out Block Diagram.

The block interface of the div_out is shown in Table 22.

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	64	Quotient result of non-restoring block
i_div_out_remainder	Input	64	Remainder result of non-restoring block
o_div_out_result	Output	64	Result of the operation

Table 22. Block Interface of div_out Block

3.3. 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

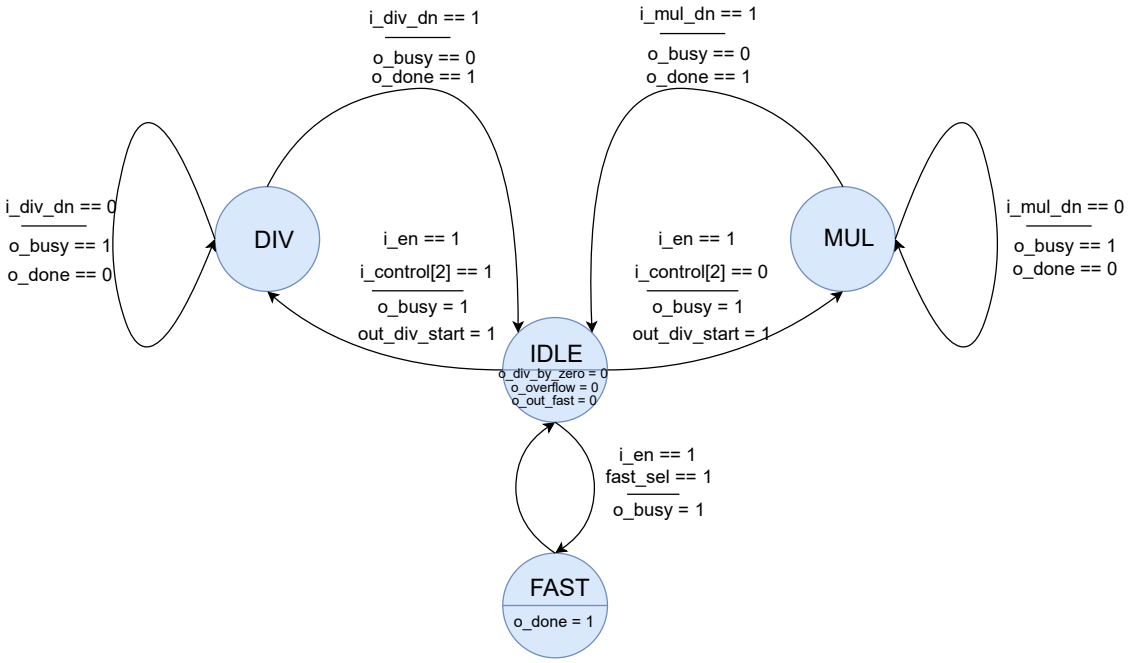

Figure 18. FSM of M-Controller.

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 64 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 23.

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

Table 23. Format of Indication Registers of Fast Operations.

Assuming A is the dividend or multiplicand, and B is divisor or multiplier, then, Fast operations are as shown in Table 24.

Fast_reg	Instr	Operation	Fast_out
7’b1000001	REM	Overflow	0
	DIV		64’h8000_0000_0000_0000
7'b001xxx0	REM(U)	A%0	A
	DIV(U)	A/0	64’hFFFF_FFFF_FFFF_FFFF
	MUL	A*0	0
7'bxx00010	All	(0*B) or (0/B)	0
7'b1000000	REM(U)	A%-1	0
	DIV(U)	A/-1	2’s comp of A
	MUL	A*-1	2’s comp of A
	MULH(SU)(U)		64’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

Table 24. Fast Operations Function Table.

The block diagram of the *mul_div_ctrl* is shown in Figure 19.

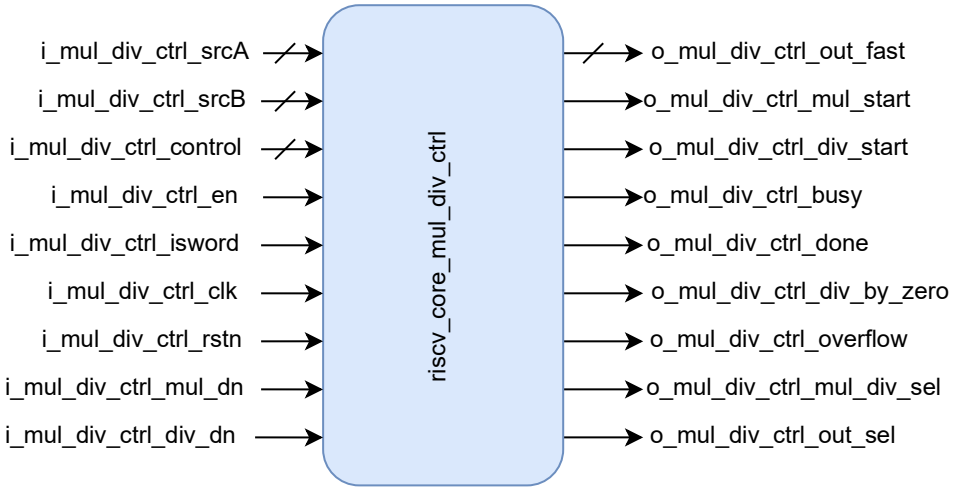

Figure 19. mul_div_ctrl Block Diagram.

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

Port Name	Direction	Width	Description
i_mul_div_ctrl_srcA	Input	64	Operand 1
i_mul_div_ctrl_srcB	Input	64	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	64	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

Table 25. Block Interface of mul_div_ctrl Block.

4. 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 RV64I. The C extension defines 9 new instruction formats, each of them fitting on 16-bit. These instruction formats are shown in Table 26.

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 16-bit RVC instruction formats.

4.1. 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.

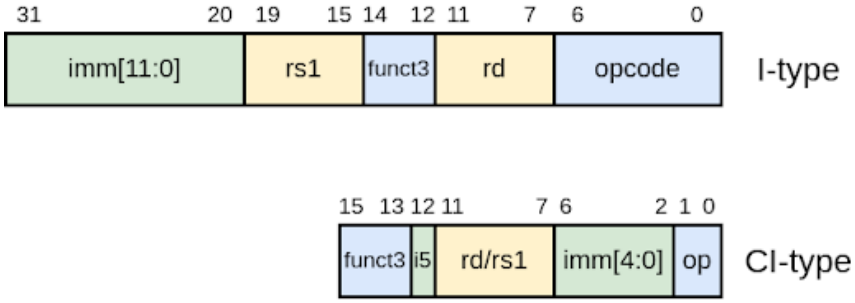

Figure 20. Comparison Between I-type Format and CI-type Format.

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.

4.2. 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.

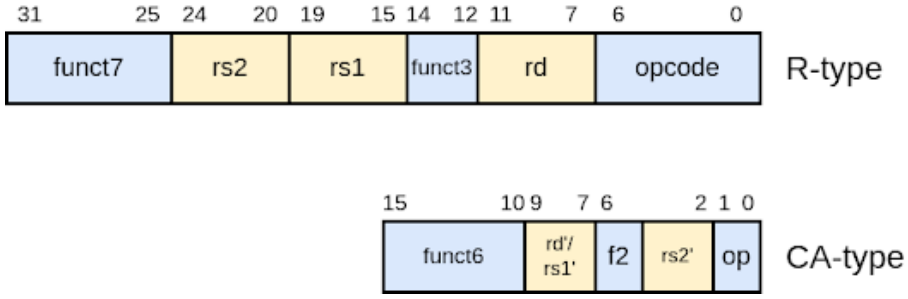

Figure 21. Comparison Between R-type format and in CA-type format.

Table 27 (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.

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

Table 27. Registers specified by the three-bit rs1 ′ , rs2 ′ , and rd ′ fields of the CIW, CL, CS, CA, and CB formats.

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.

4.3. Design of Compressed Decoder

The block diagram of the compressed decoder is shown in Figure 22.

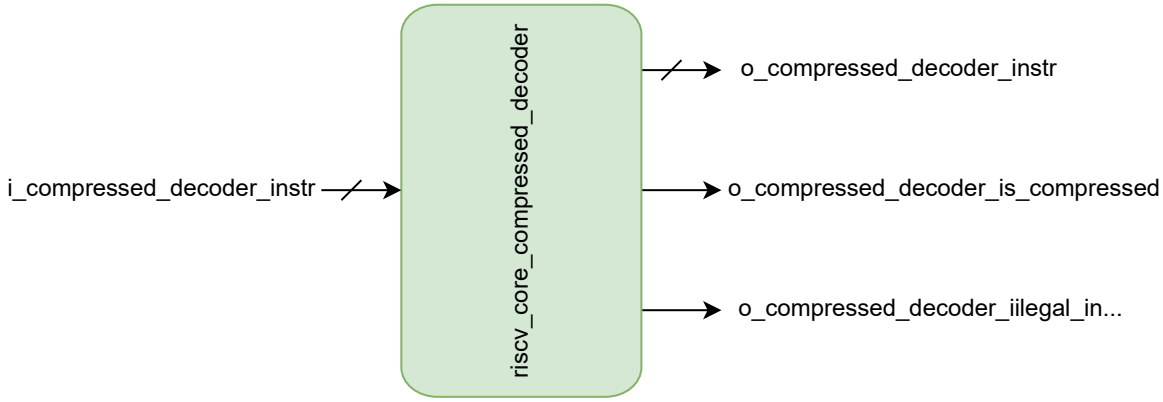

Figure 22. Compressed Decoder Block Diagram

The block interface of the compressed decoder is shown in Table 28.

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 28. Block Interface of Compressed Decoder.

5. Implementation of A-Extension for Atomic Instructions, Memory Hierarchy, and Dynamic Branch Prediction

5.1. Literature Review

5.1.1. Introduction

Cache Memory is extremely quick memory placed between central processor & Main Memory. Cache memory plays an important role when deciding the performance of multi-core systems. Processor speed is increasing at a very fast rate, so a very high-speed cache memory is needed for matching the speed of the processor. Cache systems are on-chip Cache memory part accustomed store knowledge. Cache is a buffer between a central processor and its main memory. Cache memory is employed to synchronize the info transfer rate between central processor and main memory. As cache memory nearer to the microchip, it is faster than the RAM and main memory. The advantage of storing knowledge in cache, as compared to RAM, is that it has faster retrieval times, but it has the disadvantage of on-chip energy consumption. In this review we are going to take a closer look at the memory Hierarchy and Cache Design principles.

5.1.2. Memory Hierarchy

A modem memory system, often referred to as a memory hierarchy, incorporates various storage technologies to create a whole that approximates each of the five memory idealisms. Figure 23 illustrates five typical components in a modem memory hierarchy and plots each on approximate axes that indicate their relative latency and capacity (increasing on the y axis) and bandwidth and cost per bit (increasing on the x axis). Some important attributes of each of these components are summarized in Figure 24.

- **Magnetic disks** provide the most cost-efficient storage and the largest capacities of any memory technology today, costing less than one-ten-millionth of a cent per bit (i.e., roughly $1 per gigabyte of storage), while providing hundreds of gigabytes of storage in a 3.5-inch (in.) standard form factor. However, this tremendous capacity and low cost comes at the expense of limited effective bandwidth (in the tens of megabytes per second for a single disk) and extremely long latency (roughly 10 ms per random access). On the other hand, magnetic storage technologies are non-volatile and maintain their state even when power is turned off.
- **Main memory** based on standard DRAM technology is much more expensive at approximately two hundred-thousandths of a cent per bit (i.e., roughly $200 per gigabyte of storage) but provides much higher bandwidth (several

gigabytes per second even in a low-cost commodity personal computer) and much lower latency (averaging less than 100 ns in a modern design).

- **Cache Memory:** On-chip and off-chip cache memories, both secondary (L2) and primary (LI), utilize static random-access memory (SRAM) technology that pays a much higher area cost per storage cell than DRAM technology, resulting in much lower storage density per unit of chip area and driving the cost much higher. Of course, the latency of SRAM-based storage is much lower as low as a few hundred picoseconds for small LI caches or several nanoseconds for larger L2 caches. The bandwidth provided by such caches is tremendous, in some cases exceeding 100 Gbytes/s. The cost is much harder to estimate, since high-speed custom cache SRAM is available at commodity prices only when integrated with high-performance processors. However, ignoring nonrecurring expenses and considering only the $50 estimated manufacturing cost of a modern x86 processor chip like the Pentium 4 that incorporates 512K bytes of cache and ignoring the cost of the processor core itself, we can arrive at an estimated cost per bit of one hundredth of a cent per bit (i.e., roughly $100,000 per gigabyte).
- **Register File:** Finally, the fastest, smallest, and most expensive element in a modern memory hierarchy is the register file. The register file is responsible for supplying operands to the execution units of a processor at very low latency usually a few hundred picoseconds, corresponding to a single processor cycle and at very high bandwidth, to satisfy multiple execution units in parallel. Register file bandwidth can approach 200 Gbytes/s in a modern eight-issue processor like the IBM PowerPC 970, that operates at 2 GHz and needs to read two and write one 8-byte operand for each of the eight issue slots in each cycle. Estimating the cost per bit in the register file is virtually impossible without detailed knowledge of a particular design and its yield characteristics; suffice it to say that it is likely several orders of magnitude higher than our estimate of $100,000 per gigabyte for on chip cache memory.

These memory hierarchy components are attached to the processor in a hierarchical fashion to provide an overall storage system that approximates the five idealisms infinite capacity, infinite bandwidth, zero latency, persistence, and zero cost as closely as possible. Proper design of an effective memory hierarchy requires careful analysis of the characteristics of the processor, the programs and operating system running on that processor, and a thorough understanding of the capabilities and costs of each component in the memory hierarchy.

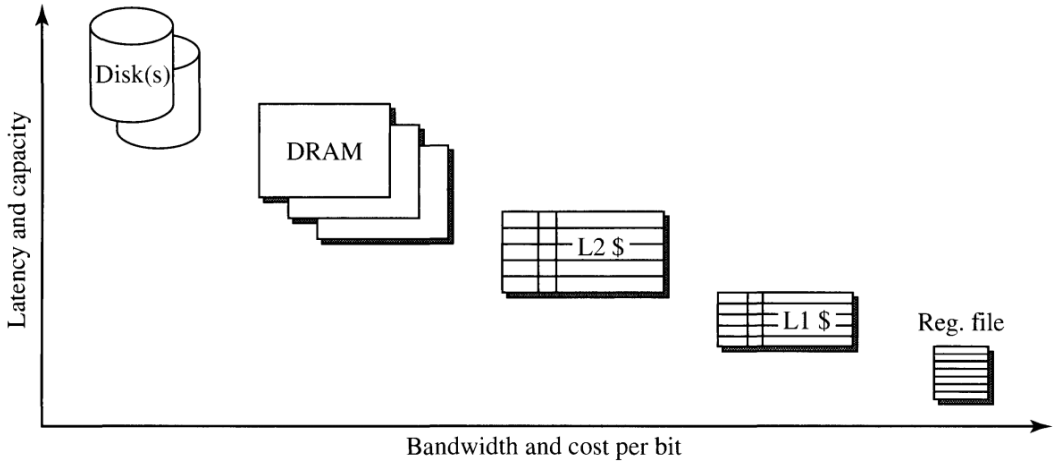

Figure 23. A Plot of 5 Typical Components in a Modern Memory Hierarchy.

Component	Technology	Bandwidth	Latency	Cost per Bit ($)	Cost per Gigabyte ($)
Disk drive	Magnetic field	10+ Mbytes/s	10 ms	$< 1 \times 10^{-9}$	< 1
Main memory	DRAM	2+ Gbytes/s	50+ ns	$< 2 \times 10^{-7}$	< 200
On-chip L2 cache	SRAM	10+ Gbytes/s	2+ ns	$< 1 \times 10^{-4}$	< 100K
On-chip L1 cache	SRAM	50+ Gbytes/s	300+ ps	$> 1 \times 10^{-4}$	> 100K
Register file	Multiported SRAM	200+ Gbytes/s	300+ ps	$> 1 \times 10^{-2}$ (?)	> 10 Mbytes (?)

Figure 24. Important Attributes of 5 Typical Components in a Modern Memory Hierarchy.

5.1.3. Locality Principles

Over the decades that the computer has been used and studied, computer engineers have measured program behaviour and have made the following observations:

- The memory-access patterns of programs are not random.
- Accesses tend to repeat themselves in time and/or be near each other in the memory address space.

The phenomenon of exhibiting predictable, non-random memory-access behaviour is called locality of reference. The behaviour is so named2 because we have observed that a program's memory references tend to be localized in time and space.

5.1.3.1. Temporal Locality

Temporal locality is the tendency of programs to use data items over and again during their execution. This is the founding principle behind caches and gives a clear guide to an appropriate data management heuristic. If a program uses an instruction or data variable, it is probably a good idea to keep that instruction or data item nearby in case the program wants it again in the near future.

5.1.3.2. Spatial Locality

Spatial locality arises because of the tendency of programmers and compilers to cluster related objects together in the memory space. As a result of this, and

because of the human-like way that programs tend to work on related data items one right after the other, memory references within a narrow range of time tend to be clustered together in the memory space. The classic example of this behavior is array processing, in which the elements of an array are processed one right after the other: elements i and i+1 is adjacent in the memory space, and element i+1 is usually processed immediately after element i.

5.1.4. Caches

A memory hierarchy can consist of multiple levels, but data are copied between only two adjacent levels at a time, so we can focus our attention on just two levels. The upper level—the one closer to the processor is smaller and faster than the lower level, since the upper level uses technology that is more expensive.

A cache stores chunks of data (called cache blocks or cache lines) that come from the backing store. A consequence of this block-oriented arrangement is that we logically divide the backing store into equal, cache-block-sized chunks. A useful side effect of this is the provision for a simple and effective means to identify a particular block: the block ID. A subset of the address, the block ID is illustrated in Figure 25, which shows a 32-bit byte main memory address as a bit vector divided into two components: the cache block ID and the byte's offset within that cache block. Note that the "byte in block" component is given as 4 bits. This example, therefore, represents a cache block of size 16 bytes. If a cache were to contain disk blocks, with several disk blocks to a cache block, then this arrangement would be the same, except that the 32-bit physical memory address would become a disk block number.

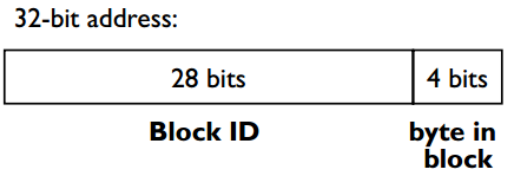

Figure 25. Block ID

Because a cache is typically much smaller than the backing store, there is a good possibility that any requested datum is not in the cache. Therefore, some mechanism must indicate whether any particular datum is present in the cache or not. The cache tags fill this purpose. The tags, a set of block IDs, comprise a list of valid entries in the cache, with one tag per data entry. The basic structure is illustrated in **Figure 4**, which shows a cache as a set of cache entries [Smith 1982]. This can take any form: a solid-state array (e.g., an SRAM implementation) or, if the cache is implemented in software, a binary search tree or even a short linked list.

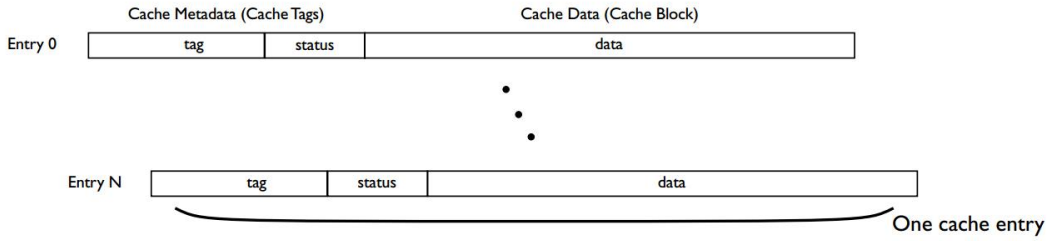

Figure 26. Basic Structure of a Cache.

5.1.4.1. Cache Organization and Design

Cache level in a cache hierarchy must be designed in a way that matches the requirements for bandwidth and latency at that level. Since the upper levels of the hierarchy must operate at speeds comparable to the processor core, they must be implemented using fast hardware techniques, necessarily limiting their complexity. Lower in the cache hierarchy, where latency is not as critical, more sophisticated schemes are attractive, and even software techniques are widely deployed. However, at all levels, there must be efficient policies and mechanisms in place for locating a particular piece or block of data, for evicting existing blocks to make room for newer ones, and for reliably handling updates to any block that the processor writes.

5.1.4.1.1. Locating a Block

Each level must implement a mechanism that enables low latency lookups to check whether a particular block is cache-resident or not. There are two attributes that determine the process for locating a block; the first is the size of the block, and the second is the organization of the blocks within the cache. Block size (sometimes referred to as line size) describes the granularity at which the cache operates. Each block is a contiguous series of bytes in memory and begins on a naturally aligned boundary. For example, in a cache with 16-byte blocks, each block would contain 16 bytes, and the first byte in each block would be aligned to 16-byte boundaries in the address space, implying that the low-order 4 bits of the address of the first byte would always be zero (i.e., Ob ••• 0000). The smallest usable block size is the natural word size of the processor (i.e., 4 bytes for a 32-bit machine, or 8 bytes for a 64-bit machine), since each access will require the cache to supply at least that many bytes, and splitting a single access over multiple blocks would introduce unacceptable overhead into the access path. In practice, applications with abundant spatial locality will benefit from larger blocks, as a reference to any word within a block will place the entire block into the cache. Hence, spatially local references that fall within the boundaries of that block can now be satisfied as hits in the block that was installed in the cache in response to the first reference to that block.

Whenever the block size is greater than 1 byte, the low-order bits of an address must be used to find the byte or word being accessed within the block. As stated above, the low-order bits for the first byte in the block must always be zero, corresponding to a naturally aligned block in memory. However, if a byte other than the first byte needs to be accessed, the low-order bits must be used as a block offset to index into the block to find the right byte. The number of bits needed for the block offset is the log2 of the block size, so that enough bits are available to span all the bytes in the block. For example, if the block size is 64 bytes, log2(64) = 6 low-order bits are used as the block offset. The remaining higher-order bits are then used to locate the appropriate block in the cache memory.

The second attribute that determines how blocks are located, cache organization, determines how blocks are arranged in a cache that contains multiple blocks. Figure 27 illustrates three fundamental approaches for organizing a cache that directly affect the complexity of the lookup process:

a) direct mapped
b) fully associative
c) set-associative.

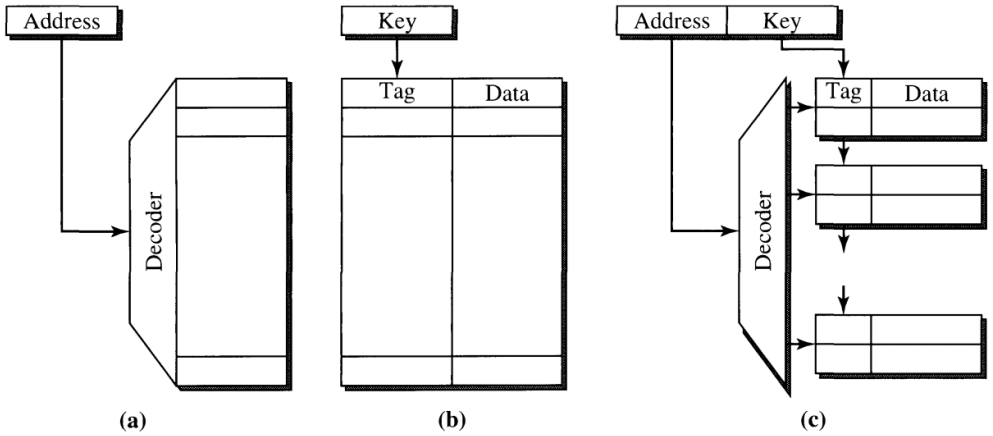

Figure 27. The Three Fundamental Approaches for Organizing a Cache.

The simplest approach, direct mapped, forces a many-to-one mapping between addresses and the available storage locations in the cache. In other words, a particular address can reside only in a single location in the cache; that location is usually determined by extracting n bits from the address and using those n bits as a direct index into one of 2n possible locations in the cache.

Of course, since there is a many-to-one mapping, each location must also store a tag that contains the remaining address bits corresponding to the block of data stored at that location. On each lookup, the hardware must read the tag and compare it with the address bits of the reference being performed to determine whether a hit or miss has occurred.

In the degenerate case where a direct-mapped memory contains enough storage locations for every address block (i.e., the n index bits include all bits of the address), no tag is needed, as the mapping between addresses and storage locations is now one-to-one instead of many-to-one. The register file inside the processor is an example of such a memory; it need not be tagged since all the address bits (all bits of the register identifier) are used as the index into the register file.

The second approach, fully associative, allows an any-to-any mapping between addresses and the available storage locations in the cache. In this organization, any memory address can reside anywhere in the cache, and all locations must be searched to find the right one; hence, no index bits are extracted from the address to determine the storage location. Again, each entry must be tagged with the address it is currently holding, and all these tags are compared with the address of the current reference. Whichever entry matches is then used to supply the data; if no entry matches, a miss has occurred.

The final approach, set-associative, is a compromise between the other two. Here a many-to-few mapping exists between addresses and storage locations. On each lookup, a subset of address bits is used to generate an index, just as in the direct-mapped case. However, this index now corresponds to a set of entries, usually two to eight, that are searched in parallel for a matching tag. In practice, this approach is much more efficient from a hardware implementation perspective, since it requires fewer address comparators than a fully associative cache, but due to its flexible mapping policy behaves similarly to a fully associative cache. Hill and Smith [1989] present a classic evaluation of associativity in caches.

5.1.4.1.2. Evicting Blocks

Since each level in the cache has finite capacity, there must be a policy and mechanism for removing or evicting current occupants to make room for blocks corresponding to more recent references. The replacement policy of the cache determines the algorithm used to identify a candidate for eviction. In a direct-mapped cache, this is a trivial problem, since there is only a single potential candidate, as only a single entry in the cache can be used to store the new block, and the current occupant of that entry must be evicted to free up the entry.

In fully associative and set-associative caches, however, there is a choice to be made, since the new block can be placed in any one of several entries, and the current occupants of all those entries are candidates for eviction. There are three common policies that are implemented in modern cache designs: first in, first out (FIFO), least recently used (LRU), and random.

The FIFO policy simply keeps track of the insertion order of the candidates and evicts the entry that has resided in the cache for the longest amount of

time. The mechanism that implements this policy is straightforward, since the candidate eviction set (all blocks in a fully associative cache, or all blocks in a single set in a set-associative cache) can be managed as a circular queue. The circular queue has a single pointer to the oldest entry which is used to identify the eviction candidate, and the pointer is incremented whenever a new entry is placed in the queue. This results in a single update for every miss in the cache. However, the FIFO policy does not always match the temporal locality characteristics inherent in a program's reference stream, since some memory locations are accessed continually throughout the execution (e.g., commonly referenced global variables). Such references would experience frequent misses under a FIFO policy, since the blocks used to satisfy them would be evicted at regular intervals, as soon as every other block in the candidate eviction set had been evicted.

The LRU policy attempts to mitigate this problem by keeping an ordered list that tracks the recent references to each of the blocks that form an eviction set. Every time a block is referenced as a hit or a miss, it is placed on the head of this ordered list, while the other blocks in the set are pushed down the list. Whenever a block needs to be evicted, the one on the tail of the list is chosen, since it has been referenced least recently (hence the name least recently used). Empirically, this policy has been found to work quite well, but is challenging to implement, as it requires storing an ordered list in hardware and updating that list, not just on every cache miss, but on every hit as well. Quite often, a practical hardware mechanism will only implement an approximate LRU policy, rather than an exact LRU policy, due to such implementation challenges. An instance of an approximate algorithm is the not-most-recently-used (NMRU) policy, where the history mechanism must remember which block was referenced most recently and victimize one of the other blocks, choosing randomly if there is more than one other block to choose from. In the case of a two-way associative cache, LRU and NMRU are equivalent, but for higher degrees of associativity, NMRU is less exact but simpler to implement, since the history list needs only a single element (the most recently referenced block).

The final policy we consider is random replacement. As the name implies, under this policy a block from the candidate eviction set is chosen at random. While this may sound risky, empirical studies have shown that random replacement is only slightly worse than true LRU and still significantly better than FIFO. Clearly, implementing a true random policy would be very difficult, so practical mechanisms usually employ some reasonable pseudo-random approximation for choosing a block for eviction from the candidate set.

5.1.4.1.3. Handling Updates to a Block

The presence of a cache in the memory sub-system implies the existence of more than one copy of a block of memory in the system. Even with a single level of cache, a block that is currently cached also has a copy still stored in the main memory. As long as blocks are only read, and never written, this is not a problem, since all copies of the block have exactly the same contents. However, when the processor writes to a block, some mechanism must exist for updating all copies of the block, in order to guarantee that the effects of the write persist beyond the time that the block resides in the cache. There are two approaches for handling this problem: write-through caches and writeback caches.

A *write-through* cache, as the name implies, simply propagates each write through the cache and on to the next level. This approach is attractive due to its simplicity, since correctness is easily maintained and there is never any ambiguity about which copy of a particular block is the current one. However, its main drawback is the amount of bandwidth required to support it. Typical programs contain about 15% of the writings, meaning that about one in six instructions updates a block in memory. Providing adequate bandwidth to the lowest level of the memory hierarchy to write through at this rate is practically impossible, given the current and continually increasing disparity in frequency between processors and main memories. Hence, write-through policies are rarely if ever used throughout all levels of a cache hierarchy.

A *write-through* cache must also decide whether or not to fetch and allocate space for a block that has experienced a miss due to a write. A write-allocate policy implies fetching such a block and installing it in the cache, while a write-no-allocate policy would avoid the fetch and would fetch and install blocks only on read misses. The main advantage of a write-no-allocate policy occurs when streaming writes overwrite most or all of an entire block before any unwritten part of the block is read. In this scenario, a useless fetch of data from the next level is avoided (the fetched data is useless since it is overwritten before it is read).

A *writeback* cache, in contrast, delays updating the other copies of the block until it has to in order to maintain correctness. In a writeback cache hierarchy, an implicit priority order is used to find the most up-to-date copy of a block, and only that copy is updated. This priority order corresponds to the levels of the cache hierarchy and the order in which they are searched by the processor when attempting to satisfy a reference. In other words, if a block is found in the highest level of cache, that copy is updated, while copies in lower levels are allowed to become stale, since the update is not propagated to them. If a block is only found in a lower level, it is promoted to the top level of cache and is updated there, once again leaving behind stale copies in lower levels of the hierarchy. The updated copy in a writeback cache is also marked with a

dirty bit or flag to indicate that it has been updated and that stale copies exist at lower levels of the hierarchy. Ultimately, when a dirty block is evicted to make room for other blocks, it is written back to the next level in the hierarchy, to make sure that the update to the block persists. The copy in the next level now becomes the most up-to-date copy and must also have its dirty bit set, in order to ensure that the block will get written back to the next level when it gets evicted.

Writeback caches are almost universally deployed, since they require much less write bandwidth. Care must be taken to design these caches correctly, so that no updates are ever dropped due to losing track of a dirty cache line. However, despite the apparent drawbacks of write-through caches, several modern processors, including the IBM Power4 [Tendler etal., 2001] and Sun UltraSPARC III [Lauterbach and Horel, 1999], do use a write-through policy for the first level of cache. In such schemes, the hierarchy propagates all writes to the second-level cache, which is also on the processor chip. Since the next level of cache is on the chip, it is relatively easy to provide adequate bandwidth for the write-through traffic, while the design of the first-level cache is simplified, since it no longer needs to serve as the sole repository for the most up-to-date copy of a cache block and never needs to initiate writebacks when dirty blocks are evicted from it. However, to avoid excessive off-chip bandwidth consumption due to write-throughs, the second-level cache maintains dirty bits to implement a writeback policy.

Figure 28 summarizes the main parameters—block size, block organization, replacement policy, write policy, and write-allocation policy—that can be used to describe a typical cache design.

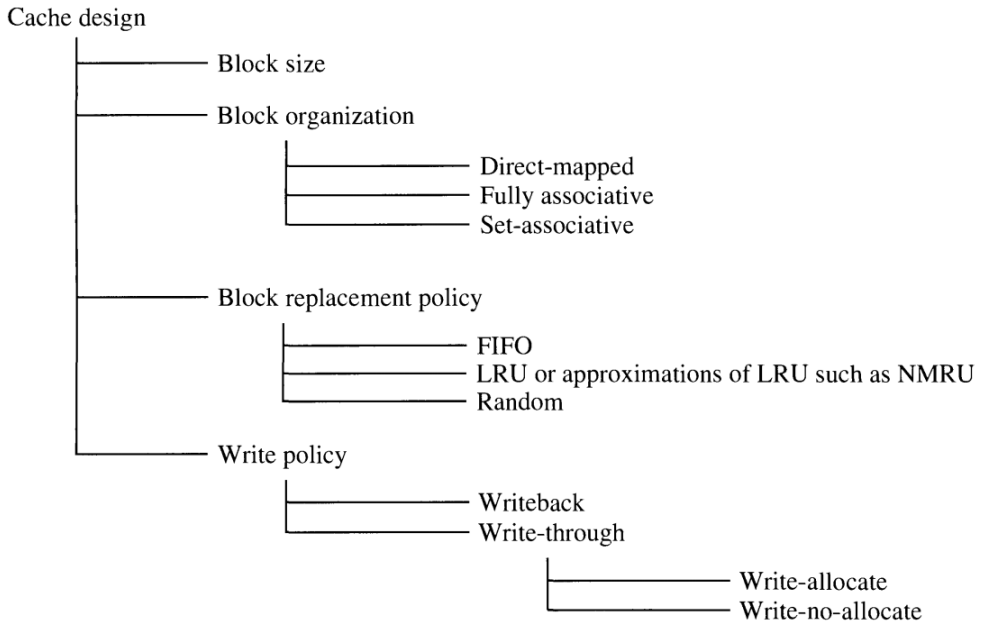

Figure 28. The main Parameters that can be used to describe a typical cache design.

5.2. Data Cache and A-Extension Design

5.2.1. Data Cache Design Without Atomic Instructions

The microarchitecture of the data cache is shown in Figure 29.

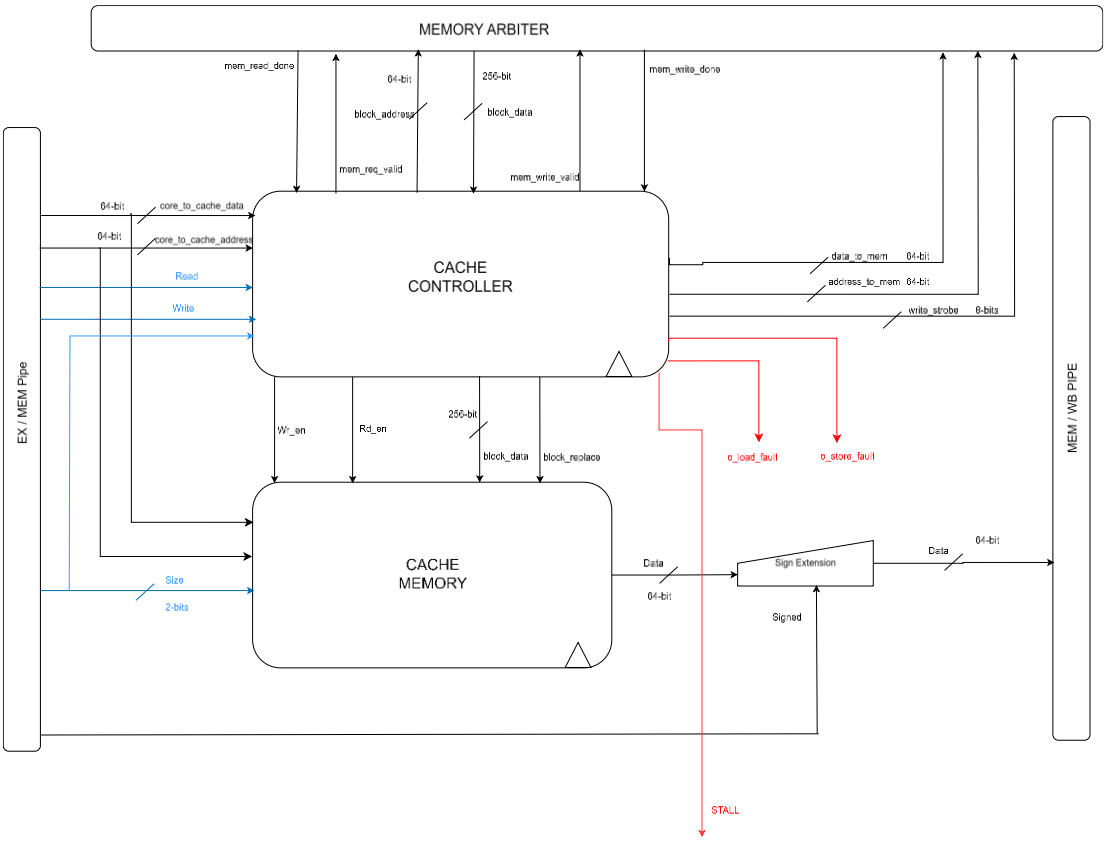

Figure 29. Cache Microarchitecture.

The specifications of our chosen implementation of cache are summarized in Table 29.

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 29. Our 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.

5.2.1.1. Cache Memory

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

5.2.1.2. 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.

The signals description of the cache is shown Table 30.

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

				it in the block determined by core_to_cache_address INDEX bits
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 30. 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.

The cache states are shown in Figure 30.

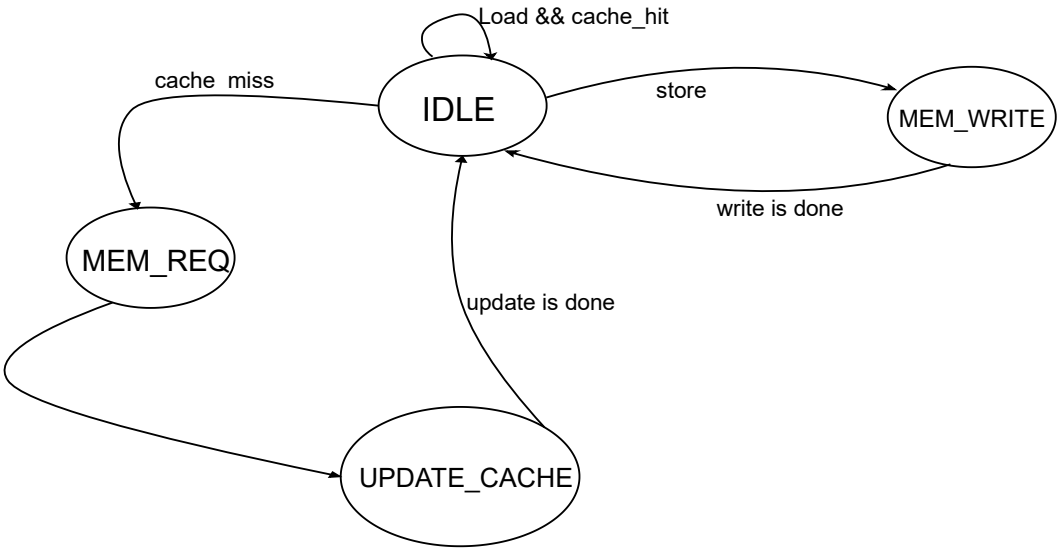

Figure 30. Cache States.

5.2.2. A-Extension for Atomic Instructions

The standard atomic-instruction extension, named “A”, contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space. The two forms of atomic instruction provided are load-reserved/store-conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency orderings including unordered, acquire, release, and sequentially consistent semantics. These instructions allow RISC-V to support the RCsc memory consistency model.

The supported instructions are shown in Table 31.

Instruction	Format	Opcode	Funct3	Funct5	Description
lr.w	R	0101111	010	00010	x[rd] = LoadReserved32(M[x[rs1]])
sc.w	R	0101111	010	00011	x[rd] = StoreConditional32(M[x[rs1]], x[rs2])
amoswap.w	R	0101111	010	00001	x[rd] = AMO32(M[x[rs1]] SWAP x[rs2])
amoadd.w	R	0101111	010	00000	x[rd] = AMO32(M[x[rs1]] + x[rs2])
amoand.w	R	0101111	010	01100	x[rd] = AMO32(M[x[rs1]] & x[rs2])
amoor.w	R	0101111	010	01000	x[rd] = AMO32(M[x[rs1]] | x[rs2])
amoxor.w	R	0101111	010	00100	x[rd] = AMO32(M[x[rs1]] ^ x[rs2])
amomax.w	R	0101111	010	10100	x[rd] = AMO32(M[x[rs1]] MAX x[rs2])
amomin.w	R	0101111	010	10000	x[rd] = AMO32(M[x[rs1]] MIN x[rs2])
amomaxu.w	R	0101111	010	11100	x[rd] = AMO32(M[x[rs1]] MAXU x[rs2])
amominu.w	R	0101111	010	11000	x[rd] = AMO32(M[x[rs1]] MINU x[rs2])
lr.d	R	0101111	011	00010	x[rd] = LoadReserved64(M[x[rs1]])

sc.d	R	0101111	011	00011	x[rd] = StoreConditional64(M[x[rs1]], x[rs2])
amoswap.d	R	0101111	011	00001	x[rd] = AMO64(M[x[rs1]] SWAP x[rs2])
amoadd.d	R	0101111	011	00000	x[rd] = AMO64(M[x[rs1]] + x[rs2])
amoand.d	R	0101111	011	01100	x[rd] = AMO64(M[x[rs1]] & x[rs2])
amoor.d	R	0101111	011	01000	x[rd] = AMO64(M[x[rs1]] | x[rs2])
amoxor.d	R	0101111	011	00100	x[rd] = AMO64(M[x[rs1]] ^ x[rs2])
amomax.d	R	0101111	011	10100	x[rd] = AMO64(M[x[rs1]] MAX x[rs2])
amomin.d	R	0101111	011	10000	x[rd] = AMO64(M[x[rs1]] MIN x[rs2])
amomaxu.d	R	0101111	011	11100	x[rd] = AMO64(M[x[rs1]] MAXU x[rs2])
amominu.d	R	0101111	011	11000	x[rd] = AMO64(M[x[rs1]] MINU x[rs2])

Table 31. A-Extension for Atomic Instructions.

5.2.2.1. Load-Reserved/Store-Conditional Instructions

Complex atomic memory operations on a single memory word or doubleword are performed with the load-reserved (LR) and store-conditional (SC) instructions. LR.W loads a word from the address in rs1, places the sign-extended value in rd, and registers a reservation set—a set of bytes that subsumes the bytes in the addressed word. SC.W conditionally writes a word in rs2 to the address in rs1: the SC.W succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the SC.W succeeds, the instruction writes the word in rs2 to memory, and it writes zero to rd. If the SC.W fails, the instruction does not write to memory, and it writes a nonzero value to rd. Regardless of success or failure, executing an SC.W instruction invalidates any reservation held by this hart. LR.D and SC.D act analogously on doublewords and are only available on RV64. For RV64, LR.W and SC.W sign-extend the value placed in rd.

5.2.2.2. Atomic Memory Operations

The atomic memory operation (AMO) instructions perform read-modify-write operations for multiprocessor synchronization and are encoded with an R-type instruction format. These AMO instructions atomically load a data value from the address in rs1, place the value into register rd, apply a binary operator to the loaded value and the original value in rs2, then store the result back to the address in rs1. AMOs can either operate on 64-bit (RV64 only) or 32-bit words in memory.

For RV64, 32-bit AMOs always sign-extend the value placed in rd.

5.2.3. Data Cache Design with Atomic Instructions

The microarchitecture of the cache with atomic is shown in Figure 31.

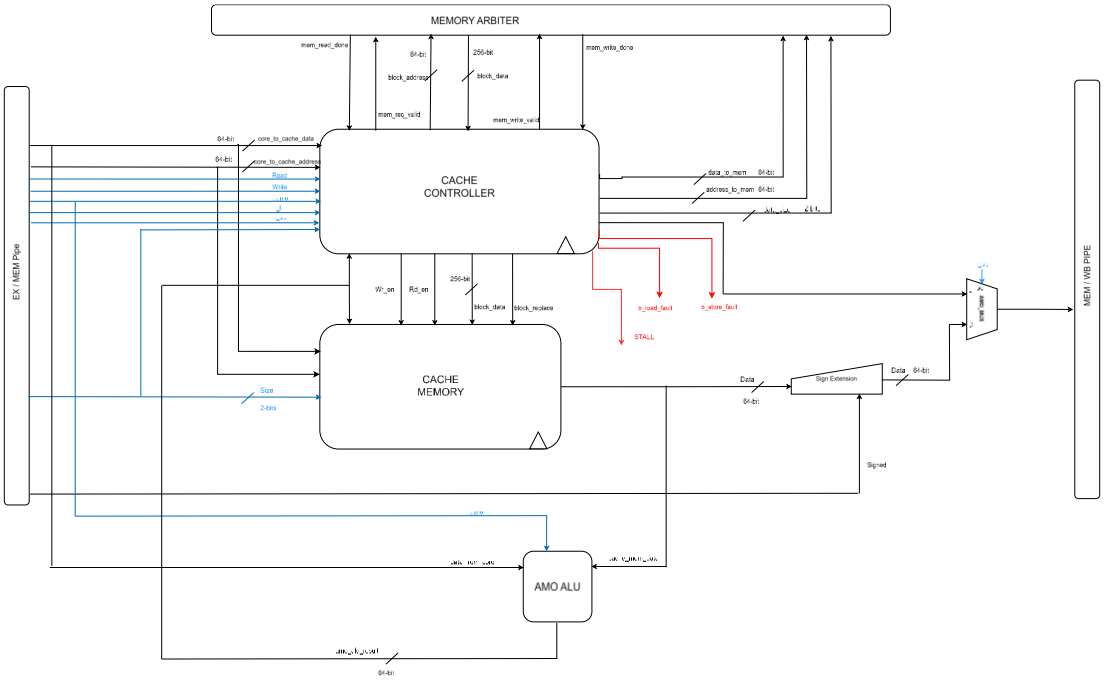

Figure 31. Cache with A-Extension Microarchitecture.

5.2.3.1. *Added Modules*

- **AMO ALU:**
  ALU to serve the AMO atomic instructions.
- **AMO MUX:**
  Mux to choose between the store conditional result and the data from cache memory.

5.2.3.2. *Added Memories*

We added for the LR/SC instructions a memory to store the address of memory to be locked by LR instruction and only stored in it by SC instruction.

5.2.3.3. *Added Signals*

The signals added to support A-extension is shown in Table 32.

Signal	Width	From	To	Description
i_amo	1-bit	core	Cache Controller	Control signal from core indicating the type of AMO instruction
i_lr	1-bit	core	Cache Controller	Control signal from core indicating the type of instruction is LOAD RESERVED (Lr)
i_sc	1-bit	core	Cache Controller	Control signal from core indicating the type of instruction is STORE CONDITIONAL (Sc)
o_sc_result	64-bit	core	AMO MUX	Result of Sc instruction indicating whether the Sc is valid or not
amo_alu_result	64-bit	AMO MUX	Cache Controller& Cache Memory	Result of AMO ALU to be written inside the cache

Table 32. Cache Additional Signals for A-Extension.

5.2.3.4. Modified Cache States

There is only one added state to support atomic instructions which is:

- **AMO_OP:**
  When there is a valid AMO instruction, IDLE goes to AMO_OP state where AMO operation is done and then goes to MEM_WRITE state.

The modified cache states to support A-extension are shown in Figure 32.

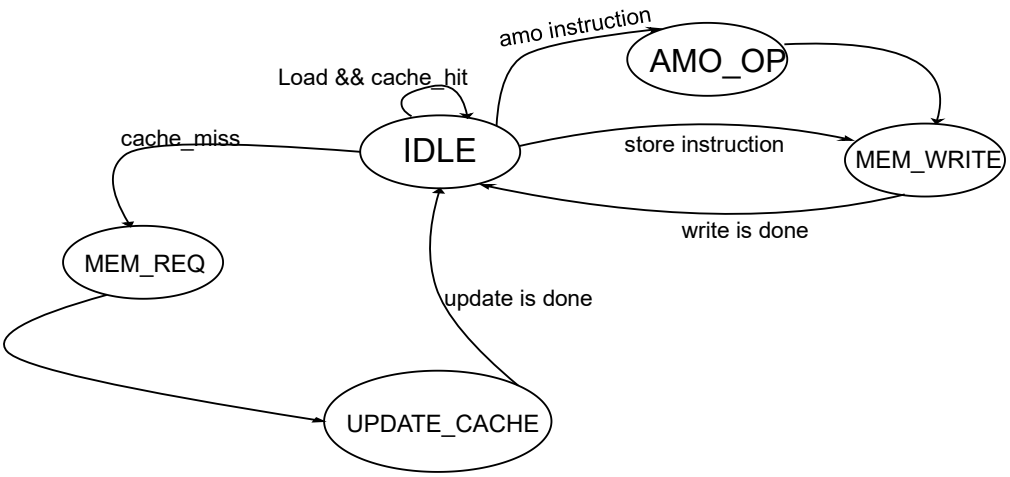

Figure 32. Modified Data Cache States.

5.3. 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.

The microarchitecture of the instruction cache is shown in Figure 33.

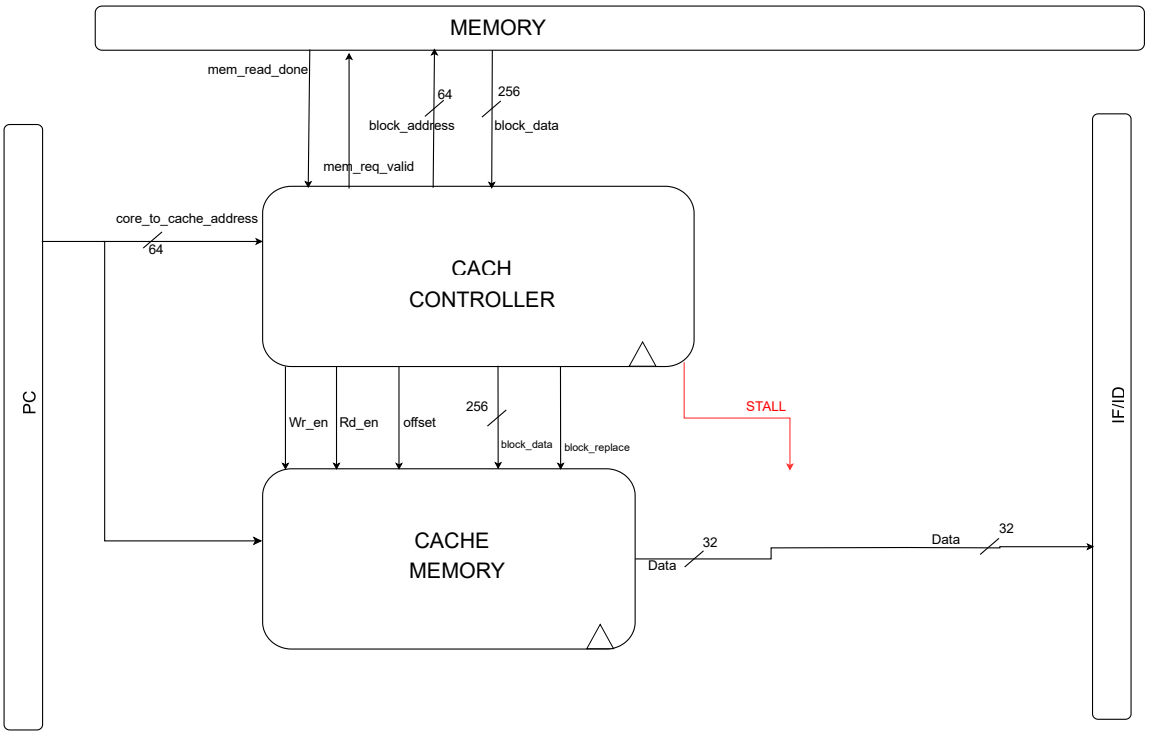

Figure 33. Instruction Cache Microarchitecture.

5.3.1. 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.

5.3.2. Cache States

The instruction cache states are shown in Figure 34.

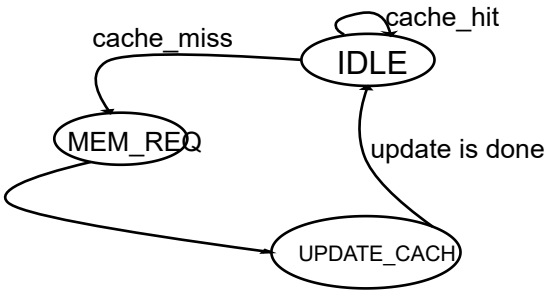

Figure 34. Instruction Cache States.

5.4. 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.

The block diagram of memory arbiter and main memory is shown in Figure 35.

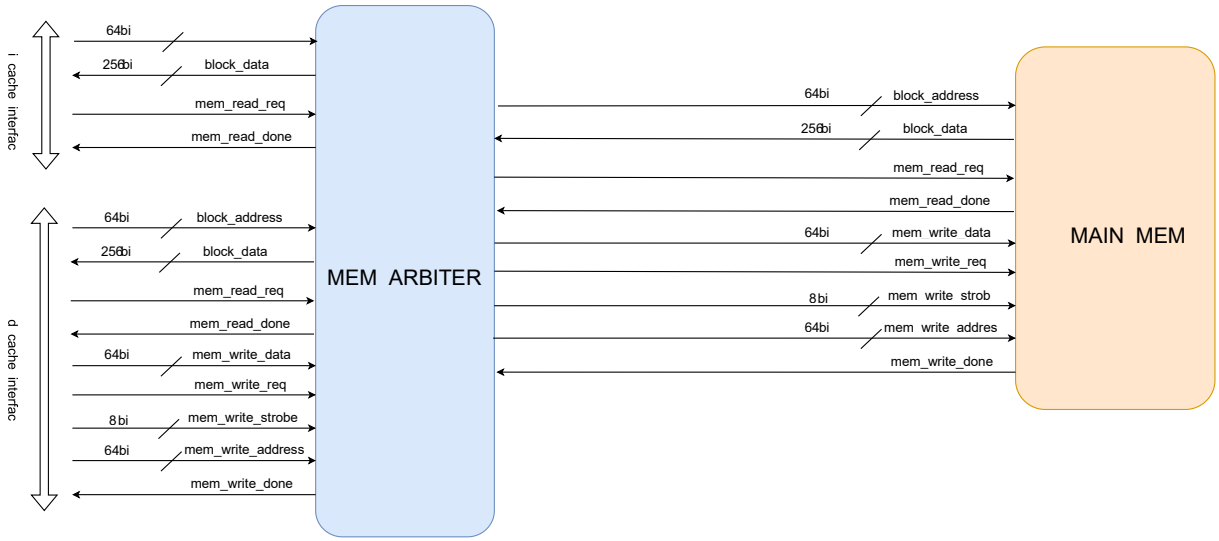

Figure 35. Main Memory and Memory Arbiter.

5.5. Branch Prediction

5.6. Branch Penalties limit the performance of the core and increases CPI beyond 1. Branch prediction with high accuracy can reduce branch penalties.

What is Control Transfer?

Control Transfer instructions cause the program flow to change to another address in the instruction memory. This means we have two things to determine to execute these instructions:

- Decision (Taken or Not Taken)
- Target Address

5.6.1. Static Prediction

First type of prediction is static prediction by always take the same decision for all control flow instructions. Overall probability a branch is taken is ~60-70% with the following distribution according to the direction of branch.

Figure 36 shows how branch prediction works by modelling conditional branches and their probabilities.

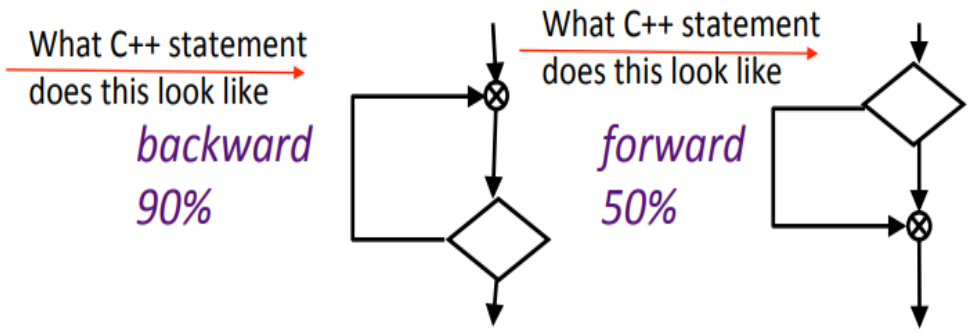

Figure 36. Conditional Branches and their Probabilities.

5.6.1.1. Static Prediction Accuracy

The accuracy of Static Predictor ranges from 50 to 80 %, in addition to at least 1 bubble is introduced either the prediction is right or wrong.

5.6.2. Dynamic Prediction

Dynamic Prediction makes use of the actual branch decisions to update itself and dynamically perform better.

5.6.2.1. Introducing Branch History Table (BHT)

It is a table that contains all previous control transfer instructions according to their corresponding PC.

For each prediction, BHT updates its next decision by comparing its prediction with the actual right decision produced from the Branch Unit.

It uses a Two-Bit-Counter to update its decision every time the same branch instruction is encountered.

Two-Bit-Counter:

It is an FSM that updates the decision of each branch instruction according to past behaviour. If the current state is strong taken or weak taken, then the decision is to take the branch. Otherwise, it is not taken. According to the output of the branch unit in the EX-stage to that branch instruction, the state is updated.

The FSM of Two-Bit-Counter is shown in Figure 37.

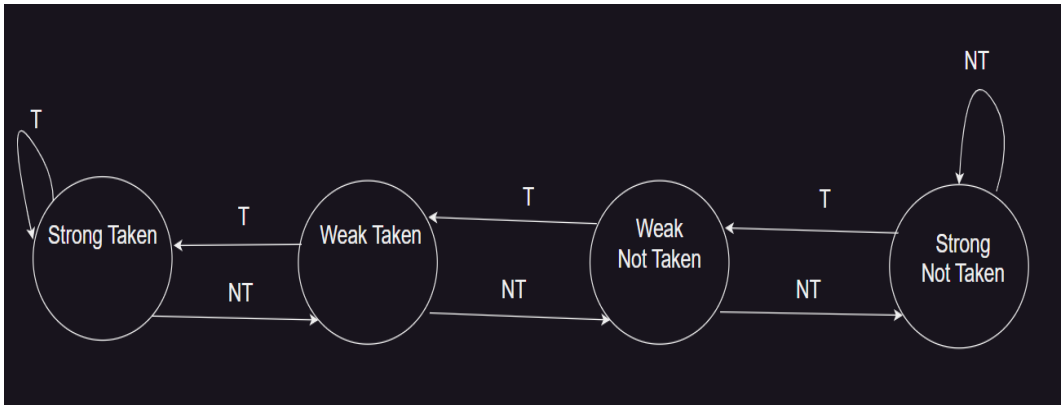

Figure 37. Two-Bit-Counter

How BHT Works?

Assume a Loop with 10 iterations, first time is mis predicted. The values of Valid and Two-Bit-Saturation is as shown in Table 33.

Valid	Two-Bit-Saturation
0	00
0	00
0	00
0	00

Table 33. Valid and Two-Bit-Saturation initially.

After 1st Branch encounter with TAKEN decision. Also, second time is mis predicted. The values of Valid and Two-Bit-Saturation is as shown in Table 34.

Valid	Two-Bit-Saturation
1	01
0	00
0	00
0	00

Table 34. Valid and Two-Bit-Saturation after first TAKEN.

After 2nd Branch encounter with TAKEN decision. Next Prediction will be "Taken" for the rest of the Loop Iterations. The values of Valid and Two-Bit-Saturation is as shown in Table 35.

Valid	Two-Bit-Saturation
1	10
0	00
0	00
0	00

Table 35. Valid and Two-Bit-Saturation after second TAKEN.

After 3rd Branch encounter with TAKEN decision became Strongly Taken. The values of Valid and Two-Bit-Saturation is as shown in Table 36.

Valid	Two-Bit-Saturation

1	11
0	00
0	00
0	00

Table 36. Valid and Two-Bit-Saturation after third TAKEN.

After first NOT TAKEN decision at the end of the Loop. If the Loop is initialized again, it will be predicted right from the first time! The values of Valid and Two-Bit-Saturation is as shown in Table 37.

Valid	Two-Bit-Saturation
1	10
0	00
0	00
0	00

Table 37. Valid and Two-Bit-Saturation after first NOT-TAKEN.

Decision Prediction is done! Now we have solved half of the problem, but we want to know where to go if the branch is taken.

5.6.2.2. Introducing Branch Target Buffer (BTB)

From the past behaviour of the branch instruction, we can also determine the Target address to branch to.

Also, for JALR instruction the Target address can be changed every few times the instruction is encountered.

BTB saves the Target Address of the branch instructions according to the PC of it.

How BTB Works?

Initially, the values of Valid and Target-Address is as shown in Table 38.

Valid	Target-Address
1	32'h0000_0000
0	32'h0000_0000
0	32'h0000_0000
0	32'h0000_0000

Table 38. Valid and Target-Address Initially

After 1st branch encounter, the values of Valid and Target-Address is as shown in Table 39.

Valid	Target-Address
1	32'h0008_0F00
0	32'h0000_0000
0	32'h0000_0000
0	32'h0000_0000

Table 39. Valid and Target-Address after first branch encounter.

5.6.2.3. Dynamic Prediction Accuracy

By using BHT and BTB, prediction accuracy can range from 80 to 95% which will increase our performance, decrease bubbles and help in limiting the CPI to 1.

5.6.3. Branch Prediction Implementation

5.6.3.1. Branch Predictor

It is a module that predicts branch instruction as shown previously.

The block diagram of Branch Predictor is shown in Figure 38.

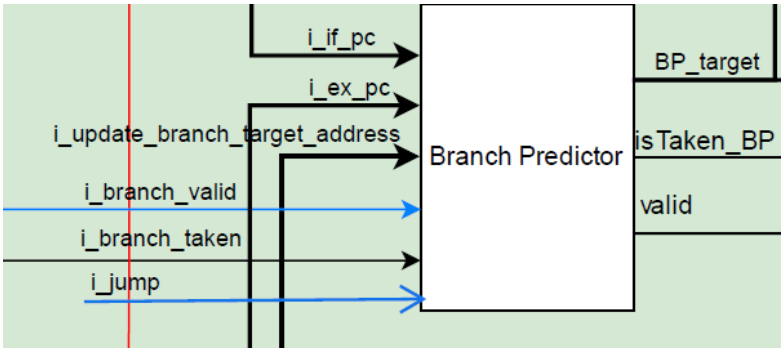

Figure 38. Branch Predictor Block Diagram.

The block interface of Branch Predictor is shown in Table 40.

Port Name	Direction	Width	Description
i_if_pc	Input	64	PC address from the IF stage
i_ex_pc	Input	64	PC address of branch instruction from EX stage to update branch predictor
i_update_branch_target	Input	64	Branch target address to update the branch predictor
i_valid_branch	Input	1	Signal indicating that there is a valid branch update
i_valid_branch_taken	Input	1	Signal indicating the branch decision
i_jump	Input	1	Signal to indicate that the PC EX is of a jump instruction
o_branch_target	Output	64	Branch target address predicted by the branch predictor
o_branch_taken	Output	1	Branch decision predicted by the branch predictor
o_branch_valid	Output	1	Signal indicating that it is a valid branch prediction

Table 40. Block Interface of Branch Predictor.

5.6.3.2. Branch Recovery

It is a module that checks the prediction of branch if taken and recovers the program flow from any misprediction.

The block diagram of Branch Recovery is shown in Figure 39.

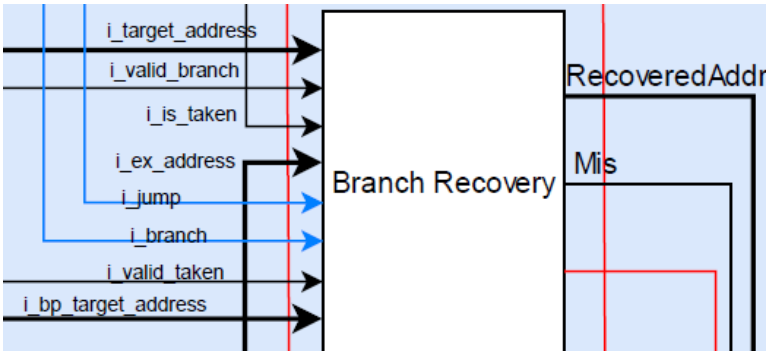

Figure 39. Branch Recovery Block Diagram.

The block interface of Branch Recovery is shown in Table 41.

Port Name	Direction	Width	Description
i_branch	Input	1	Signal indicating branch instruction from main decoder
i_jump	Input	1	Signal indicating jump instruction from main decoder
i_valid_branch	Input	1	Signal indicating valid branch prediction
i_branch_taken	Input	1	Signal indicating branch prediction is taken
i_target_address	Input	64	Branch target address predicted by the branch predictor
i_ex_address	Input	64	Target address from EX stage
i_pc_plus_offset	Input	64	PC plus 2 or 4 if the recovery indicated no taken branch
i_is_taken	Input	1	Signal from the branch unit in EX stage indicating the correct result of branch
o_mis_prediction	Output	1	Signal indicating a mis prediction
o_recovery_address	Output	1	Recovery address to go to for correct program flow

Table 41. Block Interface of Branch Recovery.

5.6.3.3. Next PC Logic

It is a module that chooses the next logic pc from addresses from branch prediction, recovery module or pc plus offset if no prediction or misprediction.

The block diagram of Next PC Logic is shown in Figure 40.

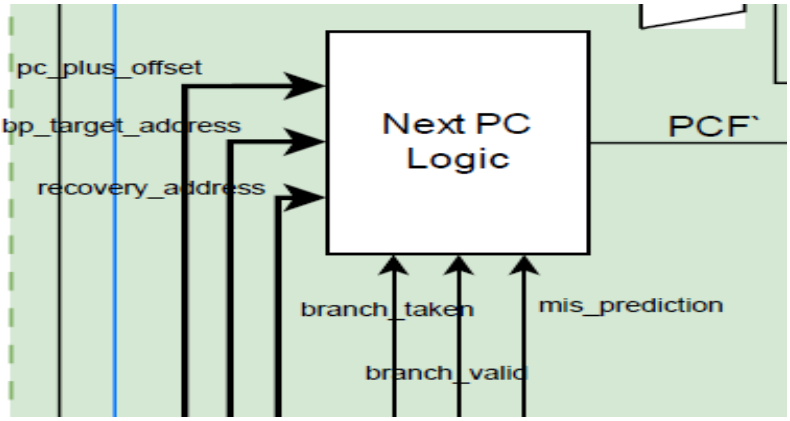

Figure 40. Next PC Logic Block Diagram.

The block interface of Next PC Logic is shown in Table 42.

Port Name	Direction	Width	Description
i_mis_prediction	Input	1-bit	Signal Indicating a misprediction from branch recovery module
i_valid_branch	Input	1-bit	Signal indicating valid branch prediction
i_branch_taken	Input	1-bit	Signal indicating branch prediction is taken
i_bp_target_address	Input	64-bit	Branch target address predicted by the branch predictor
i_recovery_address	Input	64-bit	Recovery Target address to be taken at misprediction
i_pc_plus_offset	Input	64-bit	PC plus 2 or 4 if no prediction or misprediction
next_pc	output	64-bit	Next PC to the instruction cache

Table 42. Block Interface of Next PC Logic.

6. Implementation of Privileged ISA

6.1. Introduction

In this chapter, we will discuss the implementation of the privileged architecture. First, we need the privileged architecture to manage shared resources like memory and IO devices. We also need to protect these resources as there are different entities that are trying to talk to them at the same time.

There are several levels of privilege, each level has a mode of operation, and at any time the core will be running in a certain mode, this mode has a set of associated registers and instructions and a certain access to the underlaying hardware. The privileged architecture provides a clear separation between these modes to prevent any interference between them.

Privileged architecture is responsible for handling interrupts and exceptions. It also provides system calls from lower to higher privilege level for operations that require elevated privileges.

6.2. Privilege Architecture

6.2.1. Software Stack

RISC-V can support multiple of software stacks but since we are targeting to run a LINUX-OS we chose the implementation stack shown in Figure 41.

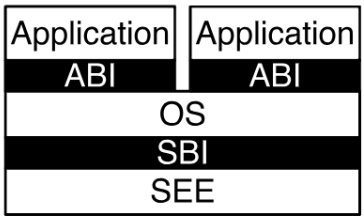

Figure 41. Software Implementation Stack.

This configuration shows that a conventional operating system can support multiprogrammed execution of multiple application. Each application communicates over an ABI with the OS, which provides the AEE. Just as applications interface with an AEE via an ABI, RISC-V operating systems interface with a supervisor execution environment (SEE) via a supervisor binary interface (SBI). A SBI comprises the supervisor-level ISA with a set of SBI function calls.

6.2.2. Privilege Levels

At any time, the core is running at some privilege level encoded as a mode in one or more CSRs (control and status registers). Three RISC-V privilege levels are currently defined as shown in Table 43.

Level	Encoding	Name	Abbreviation

0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Table 43. RISC-V Privilege Levels.

In our core we have implemented the machine mode and the supervisor mode. The machine mode is the highest privilege mode and the most mandatory mode, and it has access to all underlying Hardware and it's inherently trusted. Supervisor mode is intended for operating system usage. Supervisor mode (S-mode) can provide isolation between a supervisor-level operating system and the SEE.

A hart is normally running in a certain mode until some trap forces a switch to a trap handler, which usually runs in the machine mode by default as it's the highest privilege mode. The hart will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction.

Traps that increase privilege level are termed vertical traps, while traps that remain at the same privilege level are termed horizontal traps. The RISC-V privileged architecture provides flexible routing of traps to different privilege layers.

6.2.3. CSR unit

The block diagram of the CSR unit is shown in

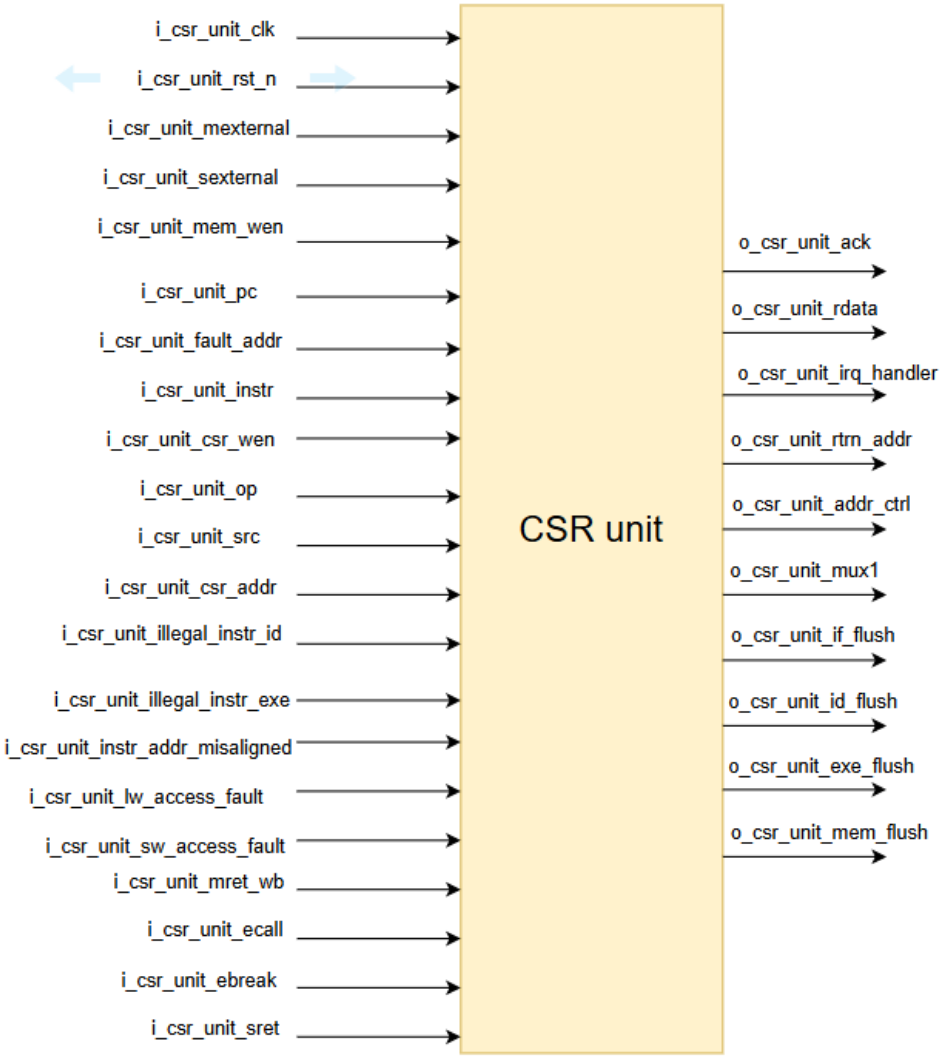

Figure 42. CSR Unit Block Diagram.

o_csr_unit_mem_flush is used to flush the write backstage, but this naming convention is used to indicate that the instruction that caused the exception was in the memory stage, the same naming convention applies for the other flush signals.

In case of interrupts all stages are flushed, and the address of the instruction in memory stage is saved in either mepc or sepc.

The block interface of the CSR unit is shown in Table 44.

Port name	Direction	Width	Description
i_csr_unit_clk	Input	1	Clock signal
i_csr_unit_rst_n	Input	1	Reset signal
i_csr_unit_mexternal	Input	1	Machine external interrupt
i_csr_unit_sexternal	Input	1	Supervisor external interrupt
i_csr_unit_mem_wen	Input	1	Memory write enable signal
i_csr_unit_pc	Input	64	Program counter

i_csr_unit_fault_addr	Input	64	Fault address that caused exception from memory stage
i_csr_unit_instr	Input	32	Instruction that was interrupted or caused the exception
i_csr_unit_csr_wen	Input	1	CSR write enable signal
i_csr_unit_op	Input	2	Operation of the CSR instruction
i_csr_unit_src	Input	64	64-bit source data
i_csr_unit_csr_addr	Input	12	Address of the CSR
i_csr_unit_illegal_instr_id	Input	1	Illegal instruction exception from decode stage
i_csr_unit_illegal_instr_exe	Input	1	Illegal instruction exception from execute stage
i_csr_unit_instr_addr_misaligned	Input	1	Instruction address misaligned exception
i_csr_unit_lw_access_fault	Input	1	Load access fault exception
i_csr_unit_sw_access_fault	Input	1	Store access fault exception
i_csr_unit_mret_wb	Input	1	Machine return instruction
i_csr_unit_ecall	Input	1	Environment call instruction
i_csr_unit_ebreak	Input	1	Environment break instruction
i_csr_unit_sret	Input	1	Supervisor return instruction
o_csr_unit_ack	Output	1	Acknowledgment signal
o_csr_unit_rdata	Output	64	Data read from CSR
o_csr_unit_irq_handler	Output	64	Trap handler address
o_csr_unit_rtrn_addr	Output	64	Address to be returned to after handling the trap
o_csr_unit_addr_ctrl	Output	1	Selects between the trap handler address and return address
o_csr_unit_mux1	Output	1	Mux1 selector signal
o_csr_unit_if_flush	Output	1	Decode stage flush signal
o_csr_unit_id_flush	Output	1	Execute stage flush signal
o_csr_unit_exe_flush	Output	1	Memory stage flush signal
o_csr_unit_mem_flush	Output	1	Write back stage flush signal

Table 44. Block Interface of CSR Unit.

6.3. Machine level CSRs

This section describes the implemented machine-level control and status registers (CSRs), and the meaning of each field in each CSR.

Machine ISA register *misa*

The *misa* CSR is a read-write register reporting the ISA supported by the hart. In our implementation we made it read only to prevent any modification to the specified fields. The misa register is shown in Figure 43.

```
//misa register
assign misa = {
    2'b10,                          //MXL=2  XLEN = 64
    36'b0,                          //reserved
    26'b00000001000001000100000101  //RV-IMAC with machine and supervisor modes
};
```

Figure 43. misa Register.

The most significant two bits indicates the native base integer ISA width which is 64 bits in our implementation. The MXL field may be writable in implementations that support multiple base ISAs. The least significant 26 bits encodes the implemented extensions which is IMAC and m-mode and s-mode in our case as shown in Figure 43.

Machine Vendor ID Register *mvendorid*

The *mvendorid* CSR is a 32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core. This register is hardwired to zero because it's a non-commercial implementation.

Machine Architecture ID Register *marched*

The *marchid* CSR is an MXLEN-bit read-only register encoding the base microarchitecture of the hart.

Machine Implementation ID Register *mimpid*

The *mimpid* CSR provides a unique encoding of the version of the processor implementation.

Hart ID Register *mhartid*

The *mhartid* CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code.

The previous mentioned three registers *mvendorid*, *mimpid* and *mhartid* are all hardwired to zero as our implementation is single-core single-hart processor.

Machine Status Registers *mstatus*

The *mstatus* register is an MXLEN-bit read/write register. The *mstatus* register keeps track of and controls the hart's current operating state. The *mstatus* register is shown in Figure 44.

```
assign mstatus = {
51'b0,
mstatus_mpp,
2'b0,
mstatus_spp,
mstatus_mpie,
1'b0,
mstatus_spie,
1'b0,
mstatus_mie,
1'b0,
mstatus_sie,
1'b0
};
```

Figure 44. mstatus Register

As shown in figure 5, *mstatus_mpp* is a two bits fields that holds the previous privilege mode before entering a trap in machine mode. If the core is running in the machine mode it can handle machine level traps and supervisor level traps.

Machine mode traps can only be handled at machine level. When running in supervisor mode, supervisor level traps are handled in the machine level by default unless a specific trap is set to be delegated to the supervisor level.

mstatus_spp is one bit that holds the previous privilege mode before entering a trap in supervisor mode. It can hold 1 in case a trap is delegated to supervisor mode.

mstatus_mpie holds the value of *mstatus_mie* prior to the trap, same for *mstatus_spie* which holds the value of *mstatus_sie* prior to the trap.

Global interrupt-enable bits, *mstatus_mie* and *mstatus_sie*, are provided for M-mode and S-mode respectively.

When the core is executing in machine mode, interrupts are globally enabled when mstatus_mie=1 and globally disabled when mstatus_mie=0. Interrupts from machine to supervisor mode are always globally disabled regardless of the setting of *mstatus_sie* bit.

When the core is running supervisor mode, Interrupts for machine mode are always globally enabled regardless of the setting of the global *mstatus_mie*.

Machine Trap-Vector Base-Address Register *mtvec*

The *mtvec* register is an MXLEN-bit read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

When the least significant two bits(mode) is 0 that indicates that the mode is direct and all traps into machine mode cause the PC to be set to the address in the base.

When the mode field holds 1 that means the mode is vectored and all exceptions into machine mode cause PC to be set the address in the base whereas interrupts cause the pc to be set to the address in the BASE field plus four times the interrupt cause number.

Machine Interrupt Registers *mip* and *mie*

The *mip* register is an MXLEN-bit read/write register containing information on pending interrupts, while *mie* is the corresponding MXLEN-bit read/write register containing interrupt enable bits. Interrupt cause number i corresponds with bit i in both *mie* and *mip*.

An interrupt i to machine mode is taken if all the following is true:

a. The current privilege mode is m-mode and mstatus_mie is set to 1 or the current privilege mode is s-mode
b. Bit i is set in both mip and mie
c. Bit i is not set in mideleg

```
assign mip = {
52'b0,
mip_meip,
1'b0,
ip_seip,
1'b0,
mip_mtip,
1'b0,
ip_stip,
5'b0
};
```

Figure 45. mip Register.

```
assign mie = {
mie_reserved,
4'b0,
mie_meie,
1'b0,
mie_seie,
1'b0,
mie_mtie,
1'b0,
mie_stie,
5'b0
};
```

Figure 46. mie Register.

As shown in Figure 45 and Figure 46, Bits *mip_meip* and *mie_meie* are the interrupt-pending and interrupt-enable bits for machine-level external interrupts. *mip_meip* is read-only and is set when there's an external interrupt coming from outside the core.

Bits *mip_mtip* and *mie_mtie* are the interrupt-pending and interrupt-enable bits for machine-level timer interrupts. *mip_mtip* is read-only in *mip* and is set when the internal counter exceeds the value stored in *mtimecmp* register.

bits *ip_seip* and *mie_seie* are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. *ip_seip* is read-only and is set when there's an external interrupt coming from outside the core.

bits *ip_ stip* and *mie_stie* are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. *ip_stip* is read-only in *ip* and is set when the internal counter exceeds the value stored in *stimecmp* register.

In our implementation, there's no machine-level interprocessor interrupts as we have one hart so there's no software interrupts and bit *mip_msip*, *mie_msie*, *mip_ssip* and *mie_ssie* are all hardwired to zero.

Interrupts to M-mode take priority over any interrupts to supervisor mode.

Multiple simultaneous interrupts destined for M-mode are handled in the following decreasing priority order: machine external interrupts, machine timer interrupts, supervisor external interrupts, supervisor timer interrupts.

Machine Trap Delegation Registers *medeleg* and *mideleg*

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the MRET instruction. To increase the performance in our implementation we made the *medeleg* and *mideleg* registers writable so exceptions and interrupts in supervisor level can be processed directly in their level.

The interrupt cause number is the index for both *medeleg* and *mideleg*

Setting a bit in *medeleg* or *mideleg* will delegate the corresponding trap to the supervisor mode. For example, if the supervisor timer interrupt (STI) is delegated to S-mode by setting mideleg[5], STIs will not be taken when executing in M-mode. By contrast, if mideleg[5] is clear, STIs can be taken in any mode and regardless of current mode will transfer control to M-mode.

Traps never transition from a more-privileged mode to a less-privileged mode. For example, if M-mode has delegated illegal instruction exceptions to S-mode, and M-mode software later executes an illegal instruction, the trap is taken in M-mode, rather than being delegated to S-mode.

Machine Scratch Register *mscratch*

The *mscratch* register is an MXLEN-bit read/write register dedicated for use by machine mode. It provides a temporary storage location that can be used by the machine-mode software.

Machine Exception Program Counter *mepc*

mepc is an MXLEN-bit read/write register and it holds all valid addresses. When a trap is taken into M-mode, *mepc* is written with the address of the instruction that was interrupted or that encountered the exception.

Machine Cause Register *mcause*

The *mcause* register is an MXLEN-bit read-write register. When a trap is taken into M-mode, *mcause* is written with a code indicating the event that caused the trap. Otherwise, *mcause* is never written by the implementation.

The most significant bit of *mcause* register is The Interrupt bit and it is set if the trap was caused by an interrupt.

Interrupts always takes priority over exceptions.

Machine Trap Value Register *mtval*

The *mtval* register is an MXLEN-bit read-write register. When a trap is taken into M-mode, *mtval* is either set to zero or written with exception-specific information to assist software in handling the trap.

In our implementation *mtval* holds the faulting address coming from the cache if there's a load access misaligned or store access misaligned otherwise it's hardwired to zero.

Hardware Performance Monitor

The **cycle** and **time** CSRs count the number of clock cycles executed by the processor core.

6.4. Supervisor level CSRs

This section describes the RISC-V supervisor-level architecture and the implemented CSRs. The intent from implementing the supervisor mode was to run an operating system but due to the lack of time and the memory limitations this was not possible, so we decided to run a program that tests all the implemented extensions.

Supervisor Status Register *sstatus*

The *sstatus* register is an SXLEN-bit read/write register.

```
assign sstatus = {
55'b0,
sstatus_spp,
2'b0,
sstatus_spie,
3'b0,
sstatus_sie,
1'b0
};
```

Figure 47. sstatus Register.

As shown in Figure 47, *sstatus_spp* is the same as *mstatus_spp* in *mstatus* register, it encodes the previous privilege mode before entering a trap in supervisor mode. It can hold 1 in case a core is running in the supervisor mode and the trap is delegated to the supervisor level. Otherwise, it's always 0.

sstatus_spie holds the value of *sstatus_sie* prior to the trap. *sstatus_sie* is the global interrupt bit for s-mode.

Interrupts can't be taken from machine mode into supervisor mode regardless of the setting of the global *sstatus_sie*.

Supervisor Interrupt Registers (*sip* and *sie*)

The sip register is an SXLEN-bit read/write register containing information on pending interrupts, while *sie* is the corresponding SXLEN-bit read/write register containing interrupt enable bits.

An interrupt i will trap to S-mode if both of the following are true:

a. either the current privilege mode is S or the *sstatus_sie* bit in the *sstatus* register is set.
b. bit i is set in both *sip* and *sie*.

sie_seie and *sie_stie* are interrupt-enable bits for supervisor-level external interrupts and supervisor-level timer interrupts respectively.

ip_seip and *ip_stip* are the interrupt-pending bits for supervisor-level external interrupts and supervisor-level timer interrupts respectively.

```
assign sip = {
54'b0,
ip_seip,
3'b0,
ip_stip,
5'b0
};

assign sie = {
sie_reserved,
6'b0,
sie_seie,
3'b0,
sie_stie,
5'b0
};
```

Figure 48. sip and sie CSRs

Supervisor Trap Vector Base Address Register *stvec*

The *stvec* register is an SXLEN-bit read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

Supervisor Exception Program Counter *sepc*

sepc is an SXLEN-bit read/write register and it holds all valid addresses. When a trap is taken into S-mode, *sepc* is written with the address of the instruction that was interrupted or that encountered the exception.

Supervisor Cause Register *scause*

The *scause* register is an SXLEN-bit read-write register. When a trap is taken into S-mode, *scause* is written with a code indicating the event that caused the trap. Otherwise, *scause* is never written by the implementation.

Supervisor Trap Value Register *stval*

The *stval* register is an SXLEN-bit read-write register. When a trap is taken into S-mode, *stval* is either set to zero or written with exception-specific information to assist software in handling the trap.

Supervisor Scratch Register *sscratch*

The *sscratch* register is an SXLEN-bit read/write register dedicated for use by supervisor mode. It provides a temporary storage location that can be used by the supervisor-mode software.

Supervisor Address Translation and Protection Register *satp*

The *satp* register controls supervisor-mode address translation and protection. It's hardwired to zero since our implementation doesn't support virtualization and address translation. We consider all addresses to be physical addresses.

6.5. CSR instructions

The set of instructions that can be used to access control and status registers is defined in the Zicsr extension, any instruction of those can atomically read-modify-write a single CSR, whose specifier is encoded in the most significant 12 bits of the instruction.

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

Figure 49. CSR Instructions.

CSRRW (atomic read/write CSR) instruction swaps the values in the CSR and integer registers. It reads the value of the CSR then writes it to the destination register. The initial value in the source register is written to the CSR.

CSRRS (atomic read and Set Bits in CSR) instruction reads the value of the CSR then writes it to the destination register. The initial value in the source register acts as bit mask that specifies the bits to be set in the CSR through an OR operation with the specified CSR then store the result in the CSR.

CSRRC (atomic read and clear Bits in CSR) instruction reads the value of the CSR then writes it to the destination register. The initial value in the source register acts as bit mask

that specifies the bits to be cleared in the CSR by ANDing the CSR with the inverted value stored in the source register then store the result in the CSR.

The CSRRWI, CSRRSI and CSRRCI are similar to CSRRW, CSRRS, and CSRRC respectively, except that the source here is 5-bit unsigned immediate.

6.6. Trap handling

In this section, we describe how the core handles the exceptions and interrupts.

6.6.1. Interrupts and Exceptions

Table 45 shows the priority order of interrupts in a decreasing manner. Interrupts always take priority over exceptions.

Priority	Code	Description
Highest	11	Machine external interrupt
	7	Machine timer interrupt
	9	Supervisor external interrupt
Lowest	5	Supervisor timer interrupt

Table 45. Decreasing priority order of interrupts.

Table 46 shows the priority order of exceptions in a decreasing manner.

Priority	Code	Description
Highest	2	Illegal instruction
	0	Instruction address misaligned
	11	Environment call
	3	Environment break
	7	Store access fault
Lowest	5	Load access fault

Table 46. Decreasing priority order of exceptions.

Illegal instruction, environment call and environment break exceptions are detected in the decode stage. Instruction address misaligned is detected in the execute stage. Load/store access fault exception is detected in the memory stage.

6.6.2. Trap Setup

The ASM chart shown in Figure 50 realizes the task of handling traps.

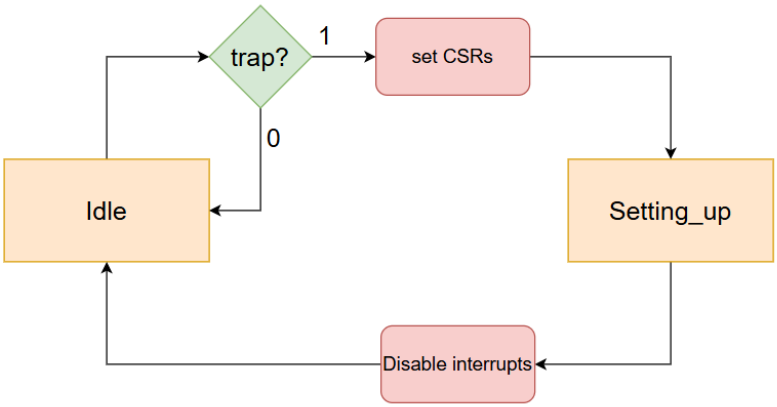

Figure 50. ASM chart of traps

Ideally the core is in the **idle** state, when a trap occurs:

1. Flush the pipeline
2. Check the current operating mode
   a. If it's m-mode:
      1) Set mstatus_mpp to 2'11
      2) Set the values of mepc, mtval, mtinst, and mcause CSRs
      3) Move to **setting_up** state
   b. If it's s-mode and the trap is delegated:
      1) Set mstatus_spp and sstatus_spp to 1'b1
      2) Set the values of sepc, stval and scause CSRs
      3) Move to **setting_up** state
   c. If it's s-mode and the trap is not delegated:
      1) Set mstatus_mpp to 2'b01
      2) Set the values of mepc, mtval, mtinst, and mcause CSRs
      3) Move to **setting_up** state.
3. Disable interrupts globally
4. Set PC to the address of the handler

   Return to the **idle** state

6.6.3. Trap Return

To return after handling a trap, there are separate trap return instructions per privilege level, MRET and SRET.

Once xret instruction is met at the end of the trap handler the following steps happen:

1. Flush the pipeline
2. Return to the previous privilege mode
3. xRET sets the pc to the value stored in the xepc register.
4. Set xie to the value stored in xpie, set xpie to 1

7. RV64IMAC with Privilege Modes

Our final microarchitecture is shown in Figure 51.

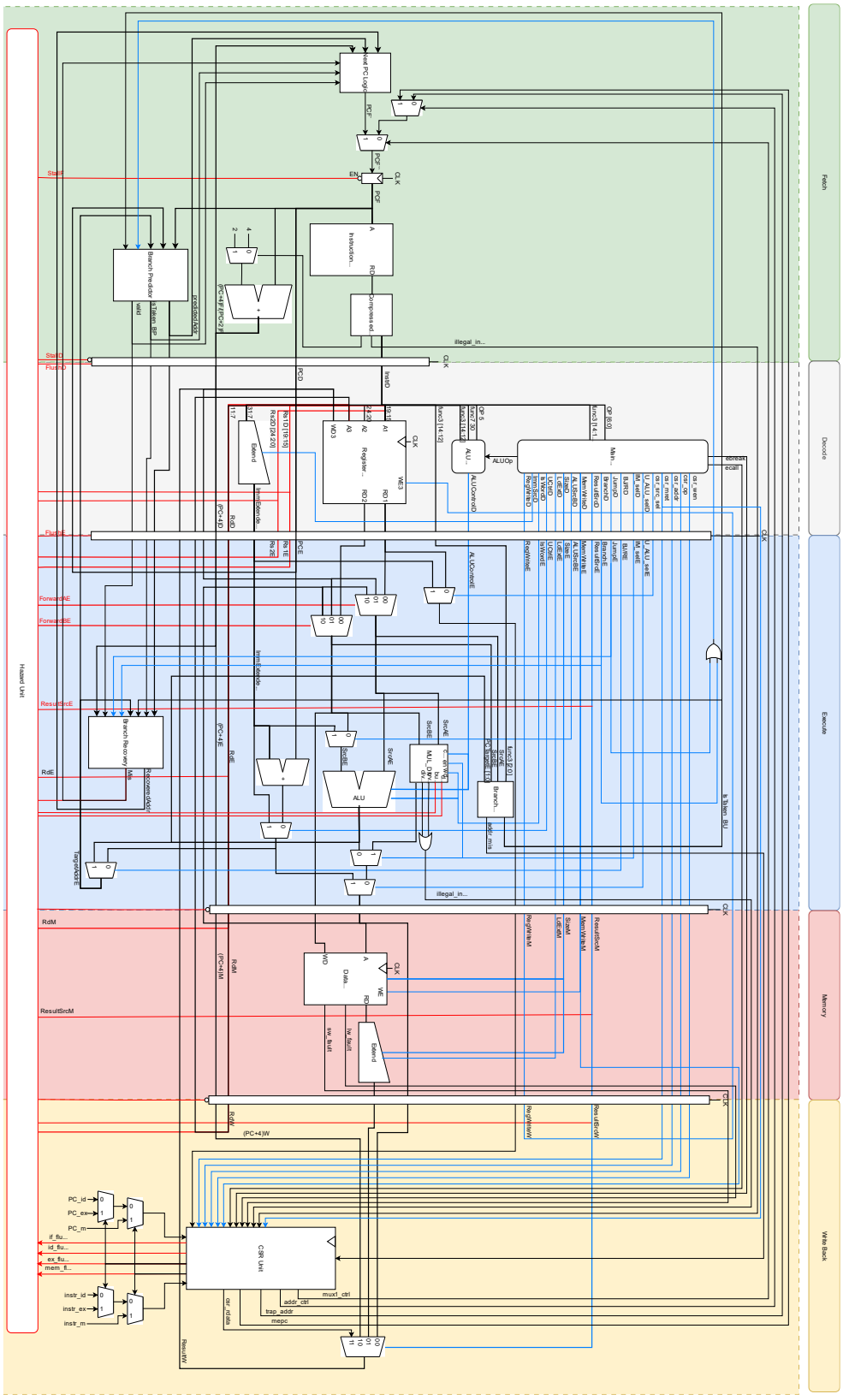

Figure 51. RV64IMAC Microarchitecture

8. Testing and Verification

In this chapter, we will discuss the verification process of the core project. First, we start with a direct test bench to make sure everything is connected well before checking corners and special scenarios in random high-coverage tests. The verification process is time-consuming, so we started the verification process in parallel with designing each stage and each extension.

Before we start you must know this is a graduation project and if we had enough time, we would make our UVM environment, but we have time limitations and the dv environment is very good at generating high coverage stimulus for our project.

Figure 52 shows our verification plan.

A	B	C	D	E	F
CHECKED	feature	feature describtion	Vrefication Goal	Pass or Fail	Test file .txt
☑	ADDI	addi rd, rs1, imm[11:0] rd = rs1 + Sext(imm[11:0]) Arithmetic overflow is ignored	proof of the instruction being decoded, executed	☑	I Type.txt
☑	SLTI	slti rd, rs1, imm[11:0] rd = (rs1 < Sext(imm[11:0]) ? 1 : 0 Both imm and rs1 treated as signed numbers	proof of the instruction being decoded, executed	☑	I Type.txt
☑	SLTIU	sltui rd, rs1, imm[11:0] rd = (rs1 < Sext(imm[11:0]) ? 1 : 0 Both imm and rs1 treated as unsigned numbers	proof of the instruction being decoded, executed	☑	I Type.txt
☑	ANDI	andi rd, rs1, imm[11:0] rd = rs1 & Sext(imm[11:0]) Note: this is a bitwise, not logical operation	proof of the instruction being decoded, executed	☑	I Type.txt
☑	ORI	ori rd, rs1, imm[11:0] rd = rs1 | Sext(imm[11:0]) Note: this is a bitwise, not logical operation	proof of the instruction being decoded, executed	☑	I Type.txt
☑	XORI	xori rd, rs1, imm[11:0] rd = rs1 ^ Sext(imm[11:0]) Note: this is a bitwise, not logical operation	proof of the instruction being decoded, executed	☑	I Type.txt
☑	SLLI	slli rd, rs, imm[4:0] rd = rs << imm[4:0] Zeros are shirfted into lower bits	proof of the instruction being decoded, executed	☑	I Type.txt
☑	SRLI	srli rd, rs, imm[4:0] rd = rs >> imm[4:0] Zeros are shirfted into upper bits	proof of the instruction being decoded, executed	☑	I Type.txt
☑	SRAI	srai rd, rs, imm[4:0] rd = rs >> imm[4:0] original sign bit is copied into the vacated upper bits	proof of the instruction being decoded, executed	☑	I Type.txt
☑	LUI	lui rd, imm[19:0]			

Figure 52. Part of Our Verification Plan

8.1. Direct Testbench Verification

Firstly, we implemented the base ISA for 64-bit RISC-V, so we started to verify the top module for the base I only then we add the other extensions one by one.

We started with a direct test bench to ensure that everything was connected well and to start debugging the top module for every instruction in base I.

For the direct test bench, we have done all the simulations on *ripes.me* to see the expected output, we considered *Ripes* simulator as a golden reference for us.

We have set the sources and destination to be the same for all to check one location only (we didn't consider forwarding at this stage).

8.1.1. Assumptions

We assumed two points for our direct testbench which are:

1. All Registers are initialized by the equation RF[i] = 2 x i
2. All Data Memory locations are initialized by the equation Mem[i]= i

8.1.2. Instruction Testing

8.1.2.1. R-Type

R-Type instruction testing is shown in Figure 53.

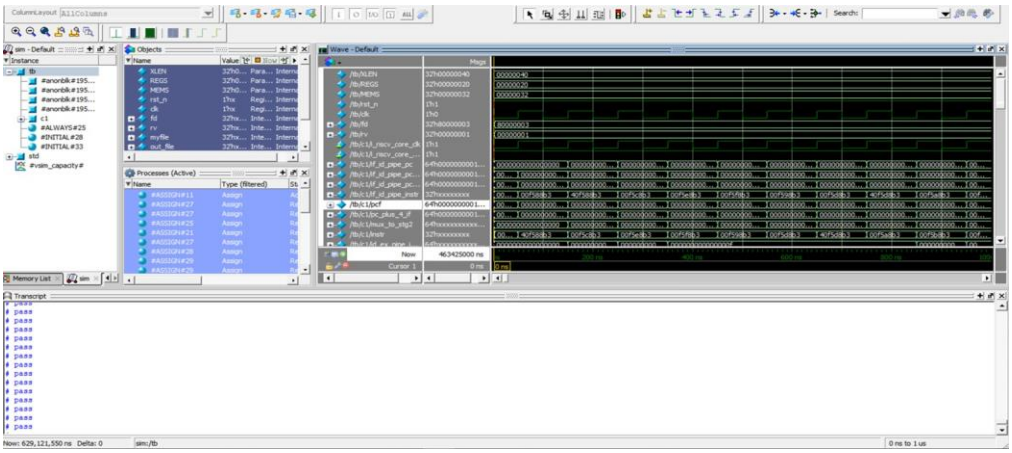

Figure 53. R-type Instruction Testing.

Expected output of R-type testing is shown in Figure 54.

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

Figure 54. Expected Output of R-Type Testing.

8.1.2.2. I-Type

I-Type instruction testing is shown in Figure 55.

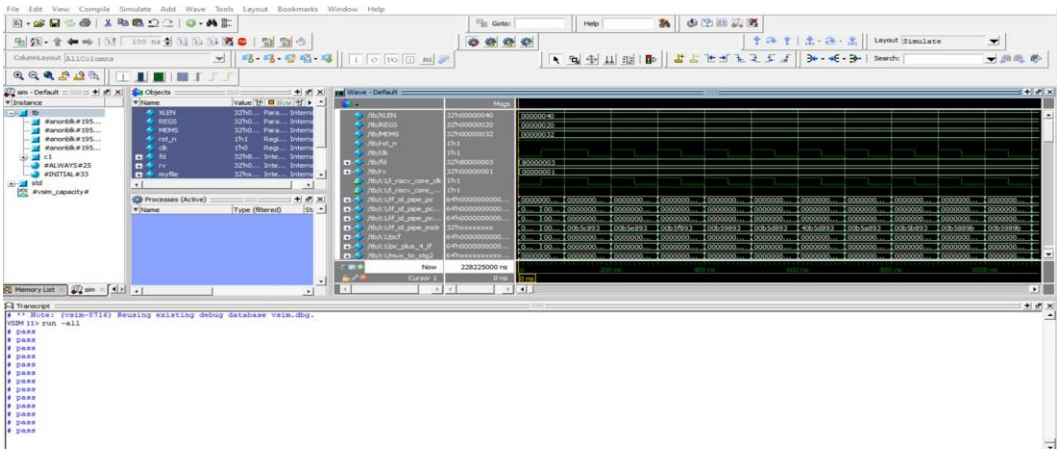

Figure 55. I-Type Instruction Testing.

Expected output of I-type testing is shown in Figure 56.

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

Figure 56. Expected Output of I-Type.

8.1.2.3. Load, Store, and U-Type

Load, Store, and U-Type instruction testing is shown in Figure 57.

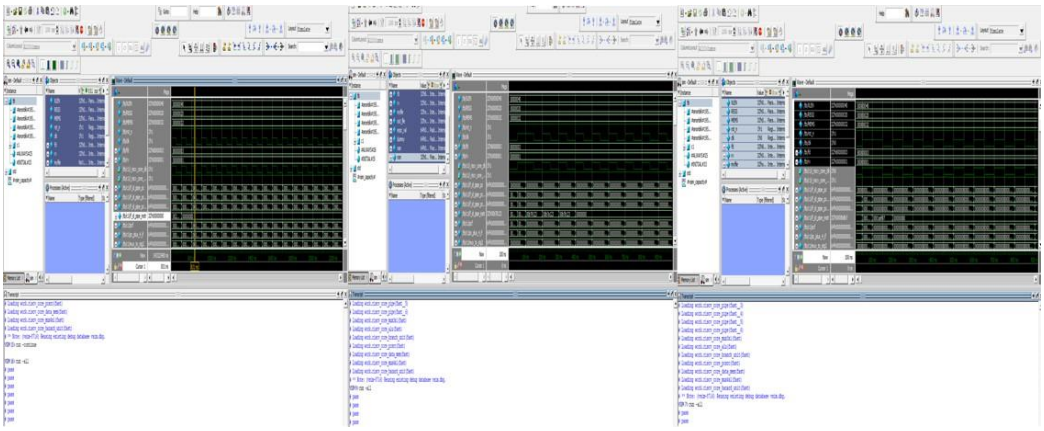

Figure 57. Load, Store, and U-type Testing.

Expected output of Load, Store, and U-type testing is shown in Figure 58.

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
U type			
lui x17 11	0x0000b8b7	x17 = immm<<<12	x17 = 0x000000000000b000
auipc x17 11	0x0000b897	x17 = pc + (imm<<<12)	x17 = pc + immm<<<12

Figure 58. Expected Output of Load, Store, and U-Type.

8.1.2.4. Jump and Branch

Expected output of jump and branch testing is shown in Figure 59.

Jump			
jal x17 56	0x038008ef	x17 = PC+4 nxt_pc = pc+56	x17 = PC+4 nxt_pc = PC+56
jalr x17,x11,11	0x00b588e7	x17 = PC+4 nxt_pc =0x16+0b	x17 = PC+4 nxt_pc = 0x00000021
Branch			
beq x11, x15,11	0x00f58563	0x16 != 0x1e	PC = PC+12 istaken=0
bne x11, x15,11	0x00f59563	0x16 != 0x1e	PC = PC+11(imm) istaken = 1
blt x11, x15,11	0x00f5c563	0x16 < 0x1e	PC = PC+ 11(imm) istaken = 1
bge x11, x15,11	0x00f5d563	0x16 !>= 0x1e	PC = PC+12 istaken = 0
bltu x11, x15,11	0x00f5e563	0x16 < 0x1e	PC = PC +11(imm) istaken=1
bgeu x11, x15,11	0x00f5f563	0x16 !>= 0x1e	PC = PC+12 istaken=0

Figure 59. Expected Output of Jump and Branch.

8.2. Random Testbench Verification

RISC-V DV-environment random instruction generator has been used.

8.2.1. Setting up the Environment

8.2.1.1. Prerequisites

1. Linux environment.
2. UVM simulator.
3. RISC-V simulator ISS.
4. RISC-V toolchain.
5. elf to hex tool.

8.2.1.2. Testing Environment and Instruction Generator

1. Riscv-dv by google and chipallinece.
2. Riscv-tests by imperas and microchip.

8.2.1.3. Linux Environment

There are many ways to implement a Linux environment in Windows OS like:

- WSL but there are some limitations in installing the RTL simulator in WSL that you can’t see any GUI
- virtual machine This is the best choice for Linux amateurs

- mysys2 same as WSL

8.2.1.4. UVM Simulator

- The best choice is to go with a free simulator like Verilator to avoid license issues.
- A paid simulator like Questa may be better and more familiar to us.

8.2.1.5. RISC-V Simulator ISSS

RISC-V simulator ISS is shown in Table 47.

Spike has been used in our project.

	Spike	Whisper	Sail*	riscvOVPsimPlus**
Opensource	Yes	Yes	Yes	No, free
ISA Support	Full ISA support Constantly updated	No Only supports machine mode privileges	Full ISA support	Full ISA support +Drafts
Docs	No proper documentation for testing	No proper documentation	Detailed documentation	An example of how to use with testing is provided
Support for UVM testing	Require code modifications	Yes	Yes	Yes
Used by	Ariane, Ibex, OpenHW Group, And more	Western Digital SweRV core lineup	NA	Supported by OpenHW group UVM

Table 47. RISC-V Simulator ISSS.

8.2.1.6. RISC-V Toolchain

- You can generate your toolchain based on your micro-arch, but it won't work with all simulators.
- You can use the generic gnu toolchain.
- You can use the pre-compiled RISC-V toolchain from Si-five.

8.2.1.7. elf to hex tool

elf to hex is a tool from Si-five that converts elf files to hex and bin files.

8.2.2. RISC-V Test Repo

8.2.2.1. Assembly File

This repo contains an assembly file for each instruction to cover all its use cases.

8.2.2.2. Disassembly

Compiling and disassembly to see the instructions as shown in Figure 60 and Figure 61.

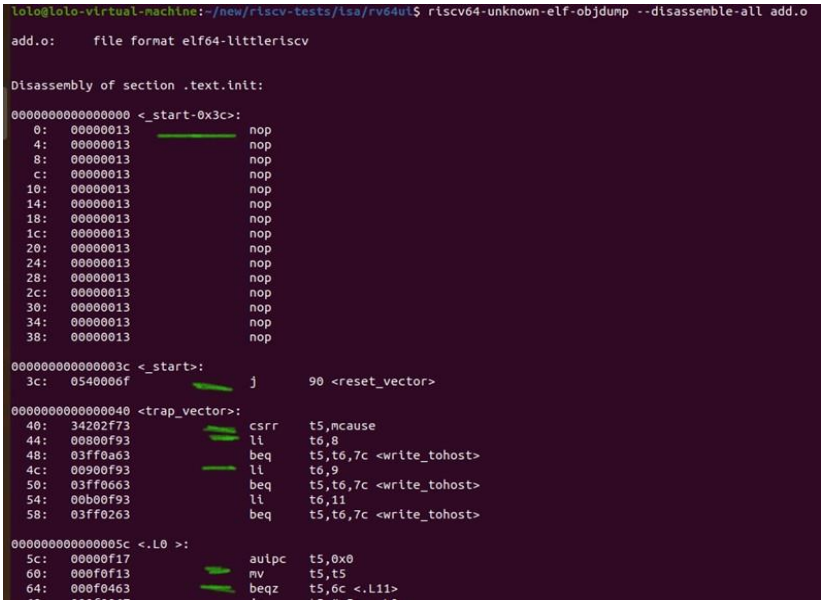

Figure 60. Disassembly.

Figure 61. Disassembly.

8.2.2.3. Conversion to Hex

The compiled file was converted to a hex using the elf-to-hex tool as shown in Figure 62.

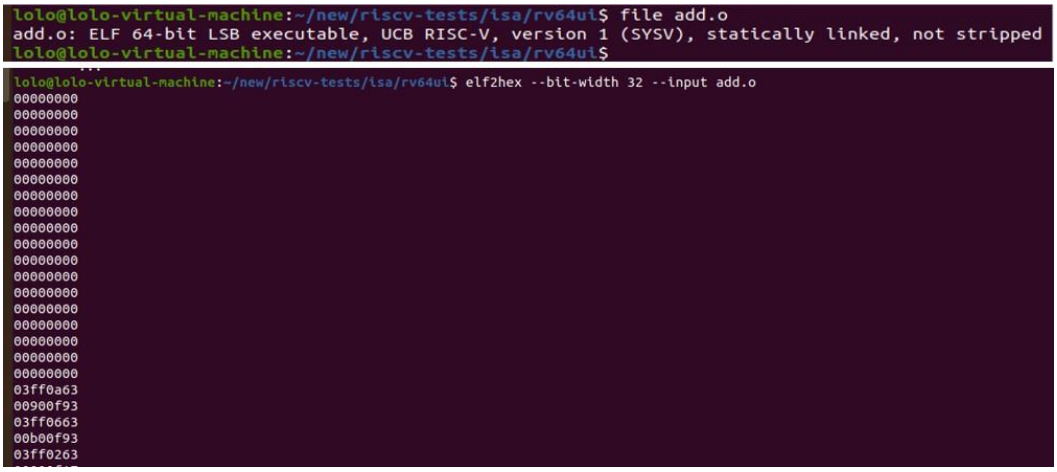

Figure 62. Conversion to Hex.

8.2.3. RISC-V DV

It is a testing environment that can generate high-coverage tests for each extension. Figure 63 shows the flow of RISC-V DV.

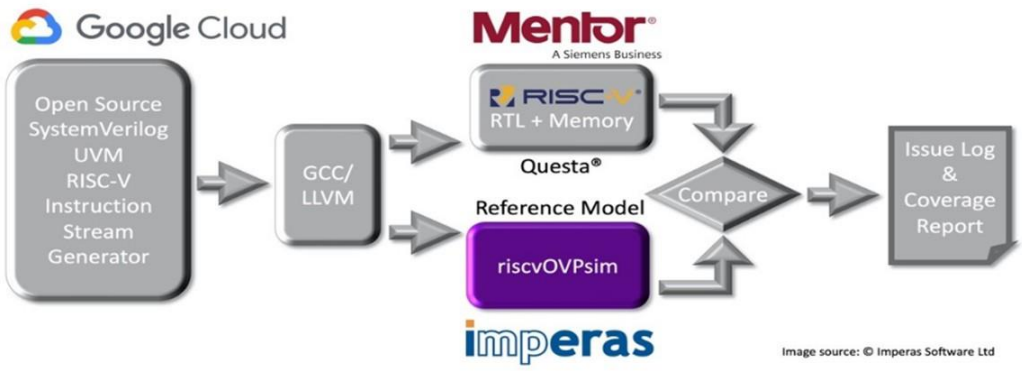

Figure 63. The Flow of the RISC-V DV

The output files are shown in Figure 64 and Figure 65.

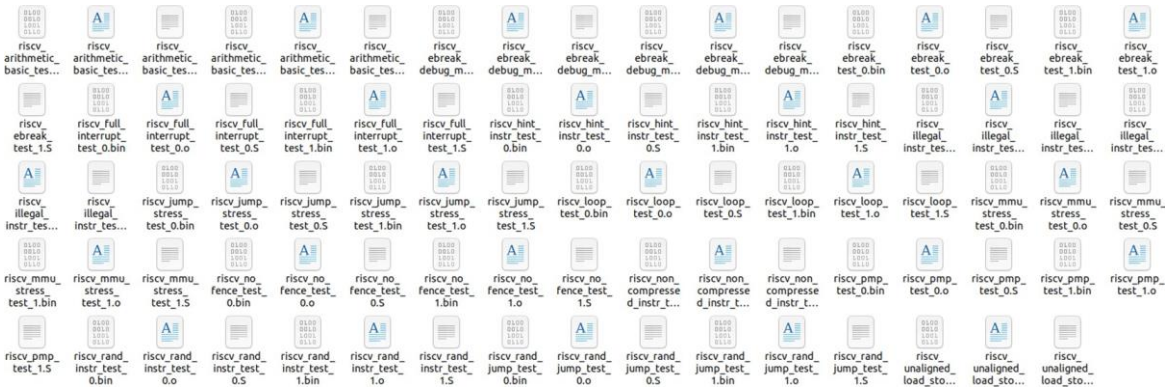

Figure 64. Output Files from DV.

```
1 .include "user_define.h"
2 .globl _start
3 .section .text
4 _start:
5                   .include "user_init.s"
6                   csrr x5, 0xf14
7                   li x6, 0
8                   beq x5, x6, 0f
9
10 0: la x13, h0_start
11 jalr x0, x13, 0
12 h0_start:
13                  li x10, 0x40001104
14                  csrw 0x301, x10
15 kernel_sp:
16                  la x5, kernel_stack_end
17
18 trap_vec_init:
19                  la x10, mtvec_handler
20                  ori x10, x10, 1
21                  csrw 0x305, x10 # MTVEC
22
23 mepc_setup:
24                  la x10, init
25                  csrw 0x341, x10
26
27 custom_csr_setup:
28                  nop
29
30 init_machine_mode:
31                  li x10, 0x1800
32                  csrw 0x300, x10 # MSTATUS
33                  li x10, 0x0
34                  csrw 0x304, x10 # MIE
35                  mret
36 init:
37                  li x0, 0xa3ccd224
```

Figure 65. Output Files from DV.

8.2.4. Testing Steps for Each Extension

1. Edit the test parameters in Yamal file to choose the extension , instructions and scenarios you want to cover it.

2. Generating the random test code assembly from riscv-dv environment.
3. Compiling the assembly file with linker script using the RISC-V toolchain.
4. Running ISS with the elf file and the log that is the change in registers after each instruction.
5. Converting elf file to hex file using elf-to-hex tool.
6. Load our instruction memory with the hex file and start the simulation.
7. Compare the core results with the ISS results.
8. Debug and trace if there's a mismatch.
9. Report the scenario that make that error to design team.

We followed these steps for each extension even with privileged extensions but in atomic extension, the riscv-dv doesn't support it so we go with direct tests again.

8.3. Testing the Core with Caches

- **Testing the core after adding caches:** Regression tests have been done with random tests for store and load and direct tests that test cache-miss scenarios.
- **Run a program:** An assembly-written program compiled and converted to hex to send data on the UART bus to test the capability of the system to run full programs.

8.4. Compilation Process

8.4.1. The Compiler

The Compiler features are shown in Figure 66.

Figure 66. Compiler Features.

8.4.2. ABI Library

The Application Binary Interface (ABI) is shown in Figure 67.

Figure 67. Application Binary Interface for Integers.

The compiling command is shown in Figure 68.

```
[redacted]:~/workspace/newdv/riscv-dv-master/out_2024-04-16/asm_test$ riscv64-unknown-elf-gcc
-march=rv64ima -mabi=lp64 -Wl,-T,link.ld,-Bstatic,--strip-debug -ffreestanding -nostdlib -o output_file input
_file
```

Figure 68. Compiling Command.

The disassembly command is shown in Figure 69.

```
[redacted]:~/workspace/newdv/riscv-dv-master/out_2024-04-16/asm_test$ riscv64-unknown-elf-objd
ump --disassemble-all elfFile
```

Figure 69. Disassembly Command.

The command used for conversion to hex is shown in Figure 70.

```
[redacted]:~/workspace/newdv/riscv-dv-master/out_2024-04-16/asm_test$ elf2hex --bit-width 64 -
-output ls.hex --input tst.elf
```

Figure 70. Conversio to Hex Command.

Then you can load the Hex to your design and the Elf file to ISS then compare the results and debug.

9. FPGA Implementation and Application

9.1. FPGA Implementation

9.1.1. Constrains

The constraints file (.xdc) plays a crucial role in mapping the Ball-Grid-Array (BGA) pins on the actual Xilinx FPGA to more human-readable names based on their hard-wired purpose.

The .xdc file specifies constraints for your FPGA design. It defines critical hardware settings that affect how the FPGA interfaces with other components. Constraints include pin assignments, clock frequencies, input/output delays, and reset pins.

Our constrains are shown in Figure 71.

```
15  set_property CFGBVS GND [current_design]
16  set_property CONFIG_VOLTAGE 1.8 [current_design]
17  set_property PACKAGE_PIN G10      [get_ports "sysclk_125_clk_p"]
18  set_property IOSTANDARD  LVDS     [get_ports "sysclk_125_clk_p"]
19  # #
20  set_property PACKAGE_PIN F10      [get_ports "sysclk_125_clk_n"]
21  set_property IOSTANDARD  LVDS     [get_ports "sysclk_125_clk_n"]
22  ############################################################################ ACTIVE-HIGH RESET
23  set_property PACKAGE_PIN AE10     [get_ports "reset"]
24  set_property IOSTANDARD  LVCMOS18 [get_ports "reset"]
25
26  ############################################################################ ACTIVE-LOW RESET
27  set_property PACKAGE_PIN AN16      [get_ports "i_riscv_core_rst_n"]
28  set_property IOSTANDARD  LVCMOS12  [get_ports "i_riscv_core_rst_n"]
29
30  #leds
31  set_property PACKAGE_PIN P20      [get_ports "for_leds[0]"]
32  set_property PACKAGE_PIN P21      [get_ports "for_leds[1]"]
33  set_property PACKAGE_PIN N22      [get_ports "for_leds[2]"]
34  set_property PACKAGE_PIN M22      [get_ports "for_leds[3]"]
35  set_property PACKAGE_PIN R23      [get_ports "for_leds[4]"]
36  set_property PACKAGE_PIN P23       [get_ports "for_leds[5]"]
37  set_property PACKAGE_PIN H23       [get_ports "for_leds[6]"]
38  #set_property PACKAGE_PIN DS7       [get_ports "for_leds[7]"]
39
40  set_property IOSTANDARD  LVCMOS18       [get_ports "for_leds[0]"]
41  set_property IOSTANDARD  LVCMOS18       [get_ports "for_leds[1]"]
42  set_property IOSTANDARD  LVCMOS18       [get_ports "for_leds[2]"]
43  set_property IOSTANDARD  LVCMOS18       [get_ports "for_leds[3]"]
44  set_property IOSTANDARD  LVCMOS18       [get_ports "for_leds[4]"]
45  set_property IOSTANDARD  LVCMOS18       [get_ports "for_leds[5]"]
46  set_property IOSTANDARD  LVCMOS18       [get_ports "for_leds[6]"]
```

Figure 71. Constrains File.

These constraints define the operating voltage, clock source and its frequency and the physical pins and connect it to a certain port.

9.1.2. Synthesis Reports

9.1.2.1. Timing Report

The timing report for our synthesized core is shown in Figure 72.

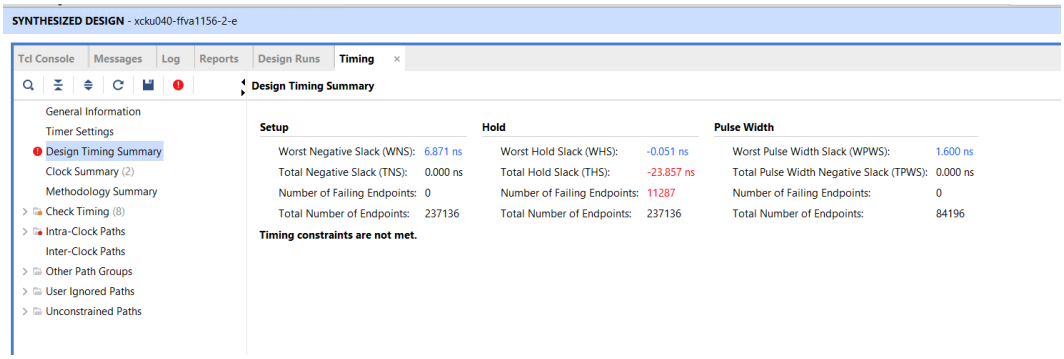

Figure 72. Synthesis Timing Report.

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.

9.1.2.2. Power Report

The power report is shown in Figure 73.

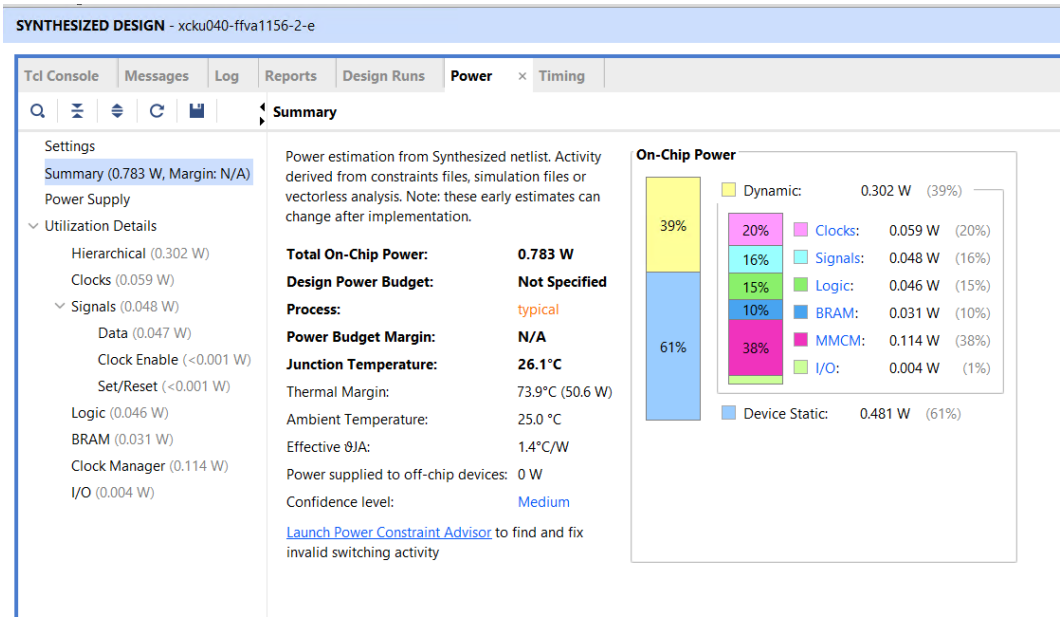

Figure 73. Synthesis Power Report.

9.1.2.3. Utilization Report

The utilization report is shown in

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

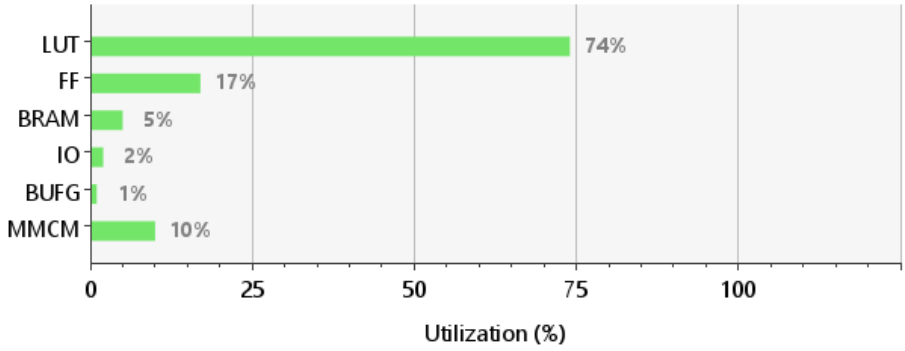

Figure 74. Synthesis Utilization Report.

9.1.3. Implementation Reports

9.1.3.1. Timing Report

The timing report is shown in Figure 75.

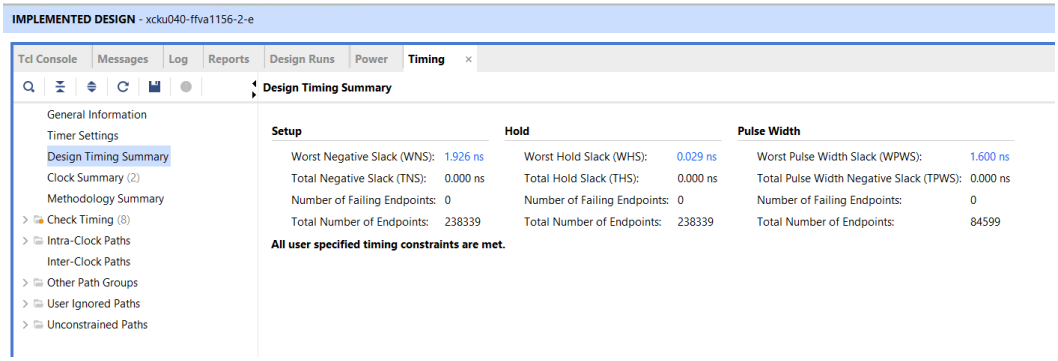

Figure 75. Implementation Timing Report.

9.1.3.2. Power Report

The power report is shown in Figure 76.

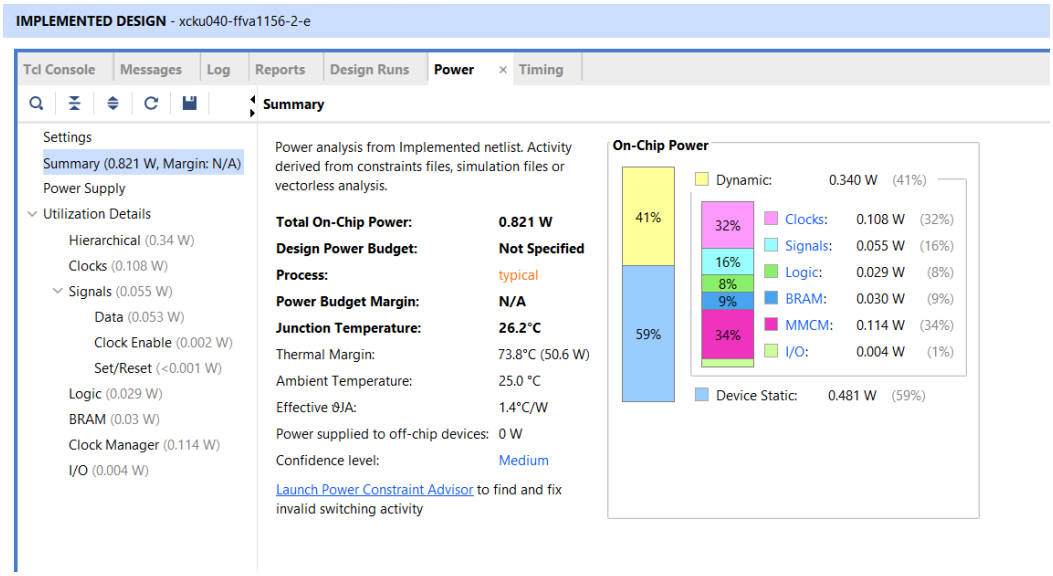

Figure 76. Implementation Power Report.

9.1.3.3. Utilization Report

The utilization report is shown in Figure 77.

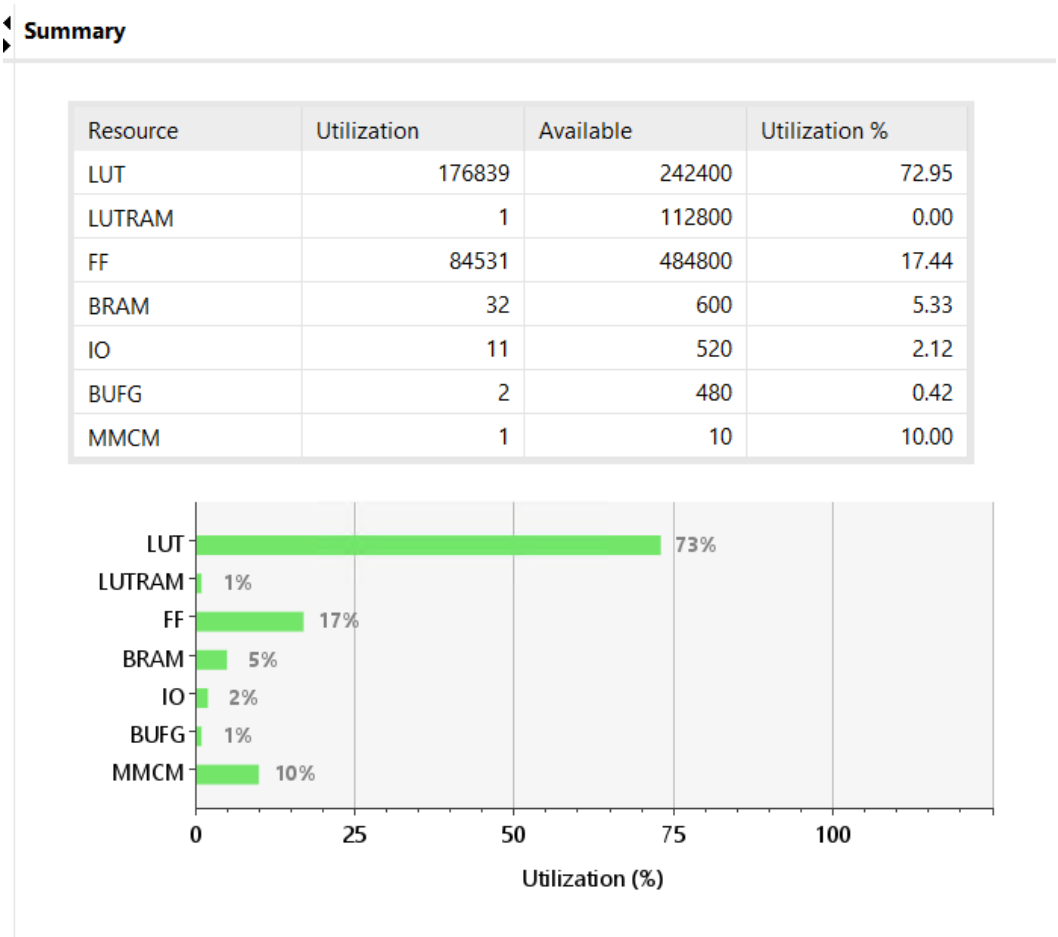

Figure 77. Implementation Utilization Report.

9.1.3.4. Noise Margin Report

The noise-margin report is shown in Figure 78.

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 78. Noise Margin Report.

9.2. Application

9.2.1. Application 1: Sending Characters using UART

The Block Diagram of our system is shown in Figure 79.

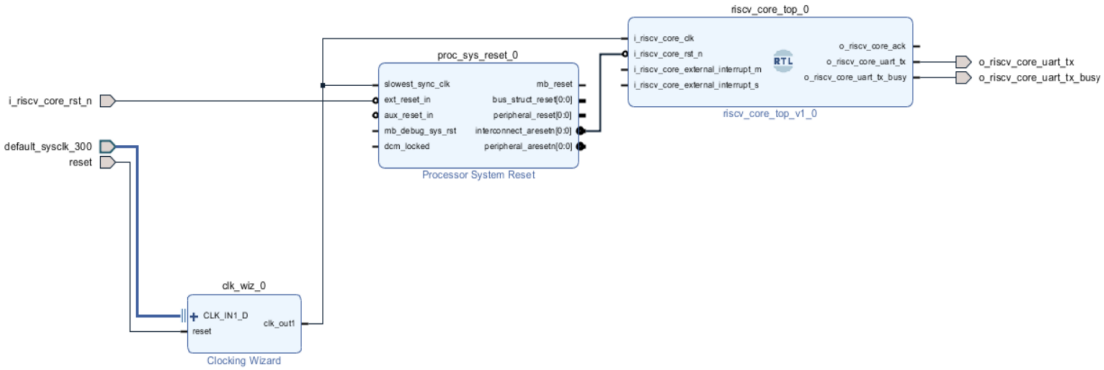

Figure 79. System Block Diagram for App 1.

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 80. 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.

9.2.2. Application 2: LEDs Testing

The Block Diagram of our system is shown in Figure 81.

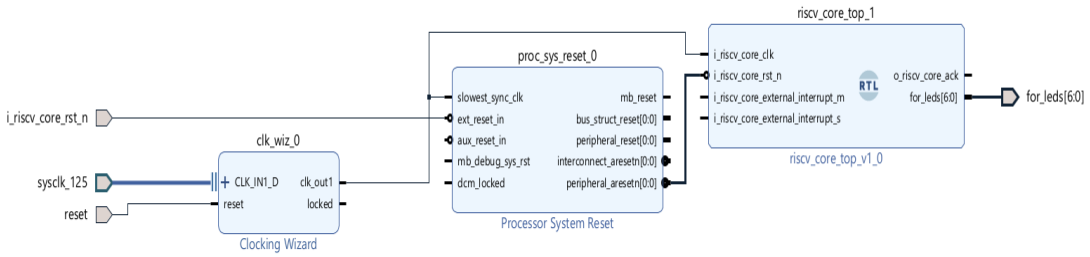

Figure 81. System Block Diagram for App 2.

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:
           ori x9 ,x9 ,0x4
           li x6 , 0x0
           j Delay_2
FAIL_c:
           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
```

10. ASIC Implementation

10.1. ASIC Physical Design Flow

ASIC (Application-Specific Integrated Circuit) design flow involves a series of steps to design and fabricate a custom chip tailored for a specific application. In this section high-level overview of the typical ASIC design flow is explained. The ASIC flow is shown in Figure 82.

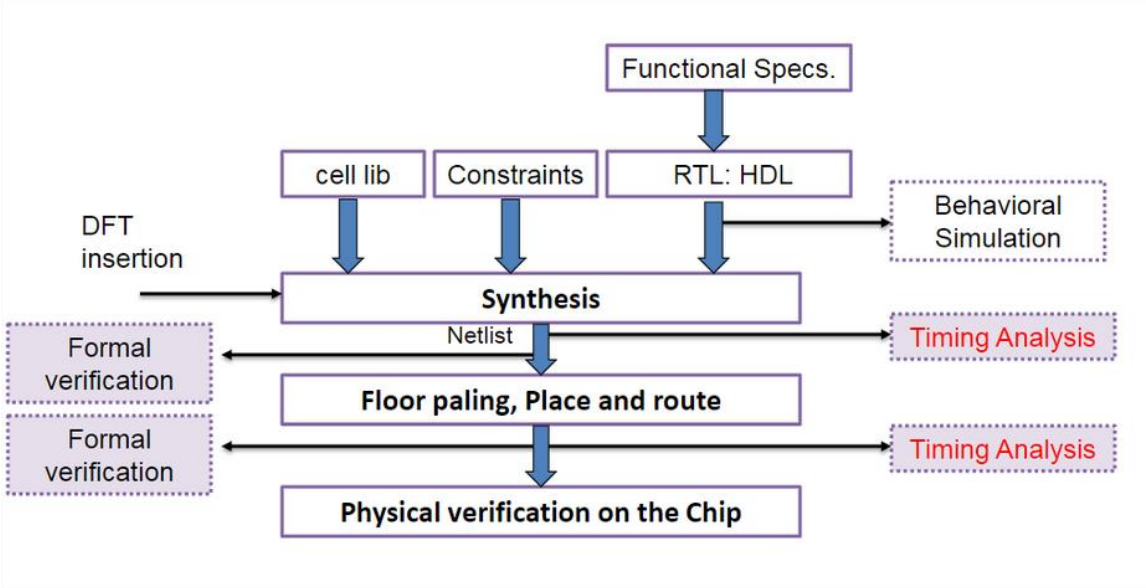

Figure 82. ASIC Design Flow.

Specification and Requirement Analysis

- Define the functionality, performance, power, 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 Register-Transfer Level (RTL) code using hardware description languages (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.
- Use techniques such as formal verification and coverage analysis to ensure thorough testing.

Synthesis

- 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-In Self-Test (BIST) circuits, and boundary 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 to ensure all timing constraints are met.
- Check for setup, hold, and clock skew issues.
- Perform signal integrity checks and noise analysis.

Physical Verification

- Verify the physical design against the design rules of the target process (Design Rule Check, DRC).
- Perform Layout Versus Schematic (LVS) checks to ensure the layout matches the schematic.

Tape-out

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

Fabrication

- The foundry manufactures the ASIC using the submitted design files.
- This involves multiple steps including photolithography, etching, doping, and metallization.

Packaging and Testing

- Package the fabricated die into an appropriate package.
- Perform initial testing of the packaged chips to check for manufacturing defects.
- Conduct functional and parametric testing to ensure the ASIC meets all specifications.

Validation and Debugging

- Test the final ASIC in a real environment to validate its functionality and performance.
- Debug any issues and make necessary design modifications if required.

Production and Deployment

- Once validated, start the mass production of the ASIC.
- Deploy the ASIC in the target application.

10.2. Project Implementation

10.2.1. Synthesis

10.2.1.1. Synthesis Steps

1. **HDL Design Input:** The process begins with an HDL description of the circuit. This description includes both the behavioural (what the circuit does) and structural (how the circuit is organized) aspects of the design.
2. **Design Constraints:** Along with the HDL code, design constraints are provided. These constraints specify requirements such as timing (e.g., clock frequency, setup and hold times), area (e.g., maximum silicon area), and power consumption.
3. **Logic Synthesis:** The HDL code is compiled by a synthesis tool (such as Synopsys Design Compiler, Cadence Genus, or Mentor Graphics Precision) to produce a gate-level netlist. This involves several sub-steps:
    a. Elaboration: The HDL code is analyzed and unrolled into a more primitive representation.
    b. Optimization: Various optimizations are applied to improve the performance, area, and power of the design. This can include techniques such as logic minimization, gate resizing, and technology mapping.
    c. Mapping to Technology Library: The design is mapped to the standard cells available in the target technology library (provided by the semiconductor foundry).
4. **Verification:** The synthesized netlist is verified to ensure that it meets the design specifications and constraints. This can involve:
    a. Functional Verification: Ensuring the synthesized netlist behaves as intended.

b. Formal Verification: Ensuring the equivalence between the original HDL and the synthesized netlist.
c. Static Timing Analysis (STA): Checking that all timing constraints are met.

5. **Generation of Reports:** The synthesis tool generates various reports that detail the results of the synthesis process. These reports include information on timing, area, power, and any violations of constraints.
6. **Netlist Handoff:** The final gate-level netlist, along with the constraint files and reports, is handed off to the next phase in the ASIC design flow, which is typically placement and routing (P&R).

10.2.1.2. Library Definitions

The following TCL script part shows the library definitions:

```
############################ Define Top Module ############################

set top_module riscv_core_top

###################### Define Working Library Directory ######################

define_design_lib work -path ./work

############################## Formality Setup File ##########################

set_svf $top_module.svf

################## Design Compiler Library Files #setup ####################

puts "############################################"
puts "#      #setting Design Libraries          #"
puts "############################################"

#Add the path of the libraries and RTL files to the search_path variable

set PROJECT_PATH /home/IC/RISCV_GP/RV64IMAC
set LIB_PATH     /home/IC/tsmc_fb_cl013g_sc/aci/sc-m

lappend search_path $LIB_PATH/synopsys
lappend search_path $PROJECT_PATH/rtl

set SSLIB "scmetro_tsmc_cl013g_rvt_ss_1p08v_125c.db"
set TTLIB "scmetro_tsmc_cl013g_rvt_tt_1p2v_25c.db"
set FFLIB "scmetro_tsmc_cl013g_rvt_ff_1p32v_m40c.db"

## Standard Cell libraries
set target_library [list $SSLIB $TTLIB $FFLIB]

## Standard Cell & Hard Macros libraries
```

```
set link_library [list * $SSLIB $TTLIB $FFLIB]
```

Target Libraries contain the cells used to generate the netlist. DC Explorer selects functionally correct gates from the target libraries to build a circuit during mapping.

The target libraries that are used to map a design become the local link libraries for the design.

DC saves this information in the design's *local_link_library* attribute.

To specify the target libraries, use the *target_library* variable.

search_path variable specifies a list of directory paths that the tool uses to find logic libraries and other files when you specify a plain file name without a path. It also sets the paths where DC Explorer can continue the search for unresolved references after it searches the link libraries.

10.2.1.3. Reading Designs

We read our RTL files using *analyze* and *elaborate* commands.

- The *analyze* command:
  - Reads an HDL source file and performs HDL syntax checking and report errors.
  - Creates HDL library objects in an intermediate format (.syn) and stores them in work directory.
- The *elaborate* command:
  - Synthesize the design into a technology independent representation “GTECH” from intermediate format (.syn) generated from *analyze* command.
  - Perform link automatically.

The following TCL script part shows how we read the design:

```
######################### Reading RTL Files ################################

puts "###########################################"
puts "#             Reading RTL Files           #"
puts "###########################################"

set file_format sverilog

#IF
analyze -format $file_format riscv_core_compressed_decoder.sv
analyze -format $file_format {riscv_core_icache_controller.sv \
        riscv_core_icache_memory.sv \
        riscv_core_icache_top.sv}
```

```
analyze -format $file_format riscv_core_64bit_adder.sv
#ID
analyze -format $file_format riscv_core_rf.sv
analyze -format $file_format {riscv_core_main_decoder.sv \
        riscv_core_csr_control_signals.sv \
        riscv_core_main_decoder_top.sv}
analyze -format $file_format riscv_core_alu_decoder.sv
analyze -format $file_format riscv_core_immextend.sv
#EX
analyze -format $file_format {riscv_core_mul_in.sv \
        riscv_core_booth.sv \
        riscv_core_mul_out.sv \
        riscv_core_div_in.sv \
        riscv_core_non_restoring.sv \
        riscv_core_div_out.sv \
        riscv_core_mul_div_ctrl.sv \
        riscv_core_mul_div.sv}
analyze -format $file_format riscv_core_alu.sv
analyze -format $file_format riscv_core_branch_unit.sv
analyze -format $file_format riscv_core_pcsrc.sv
#MEM
analyze -format $file_format {riscv_core_dcache_controller.sv \
        riscv_core_dcache_memory.sv \
        riscv_core_amo_alu.sv \
        riscv_core_dcache_top.sv}
analyze -format $file_format riscv_core_ldextend.sv
#WB
analyze -format $file_format csr_regfile.sv
#HZ
analyze -format $file_format riscv_core_hazard_unit.sv
#MUXs
analyze -format $file_format riscv_core_mux2x1.sv
analyze -format $file_format riscv_core_mux_2to1.sv
analyze -format $file_format riscv_core_mux3x1.sv
analyze -format $file_format riscv_core_mux4x1.sv
#PIPE
analyze -format $file_format riscv_core_pipe.sv
#UART_TX
analyze -format $file_format uartTrans.sv
#TOP
analyze -format $file_format riscv_core_top.sv

elaborate -lib WORK riscv_core_top
```

10.2.1.4. Design Compiler Constrains

The following TCL script part shows design constrains:

```
################################################################################
          ##############################################################
                ##### Section 0 : DC Variables #####
          ##############################################################
################################################################################

# Prevent assign statements in the generated netlist (must be applied before
compile command)
set_fix_multiple_port_nets -all -buffer_constants -feedthroughs

################################################################################
          ##############################################################
                ##### Section 1 : Clock Definition #####
          ##############################################################
################################################################################

#1. Master Clocks
#REF_CLK
set REF_CLK_NAME i_riscv_core_clk
set REF_CLK_PER 10
set REF_CLK_SETUP_SKEW 0.2
set REF_CLK_HOLD_SKEW 0.1

create_clock -name $REF_CLK_NAME -period $REF_CLK_PER -waveform "0 [expr
$REF_CLK_PER/2]" [get_ports $REF_CLK_NAME]
set_clock_uncertainty -setup $REF_CLK_SETUP_SKEW [get_clocks $REF_CLK_NAME]
set_clock_uncertainty -hold $REF_CLK_HOLD_SKEW  [get_clocks $REF_CLK_NAME]


################################################################################
          ##############################################################
            #### Section 2 : Clocks Relationship ####
          ##############################################################
################################################################################


################################################################################
          ##############################################################
            #### Section 3 : set input/output delay on ports ####
          ##############################################################
################################################################################

set in_delay  [expr 0.2*$REF_CLK_PER]
set out_delay [expr 0.2*$REF_CLK_PER]

#Constrain Input Paths

#Global
```

```
set_input_delay $in_delay -clock $REF_CLK_NAME [get_port
i_riscv_core_external_interrupt_m]
set_input_delay $in_delay -clock $REF_CLK_NAME [get_port
i_riscv_core_external_interrupt_s]
#D$
set_input_delay $in_delay -clock $REF_CLK_NAME [get_port mem_read_done]
set_input_delay $in_delay -clock $REF_CLK_NAME [get_port i_mem_write_done]
set_input_delay $in_delay -clock $REF_CLK_NAME [get_port
i_block_from_axi_data_cache]
#I$
set_input_delay $in_delay -clock $REF_CLK_NAME [get_port i_mem_done]
set_input_delay $in_delay -clock $REF_CLK_NAME [get_port
i_block_from_axi_i_cache]
#UART_TX
set_input_delay $in_delay -clock $REF_CLK_NAME [get_port uart_ready]

#Constrain Output Paths

#Global
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port o_riscv_core_ack]
#D$
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port mem_read_address]
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port o_mem_write_data]
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port
o_mem_write_address]
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port mem_read_req]
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port o_mem_write_valid]
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port o_mem_write_strobe]
#I$
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port
o_addr_from_control_to_axi]
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port o_mem_req]
#UART_TX
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port uart_out_data]
set_output_delay $out_delay -clock $REF_CLK_NAME [get_port uart_valid]

##############################################################################
           ############################################################
                  #### Section 4 : Driving cells ####
           ############################################################
##############################################################################

set_driving_cell -library scmetro_tsmc_cl013g_rvt_ss_1p08v_125c -lib_cell
BUFX2M -pin Y [get_port i_riscv_core_external_interrupt_m]
set_driving_cell -library scmetro_tsmc_cl013g_rvt_ss_1p08v_125c -lib_cell
BUFX2M -pin Y [get_port i_riscv_core_external_interrupt_s]
set_driving_cell -library scmetro_tsmc_cl013g_rvt_ss_1p08v_125c -lib_cell
BUFX2M -pin Y [get_port mem_read_done]
```

```
set_driving_cell -library scmetro_tsmc_cl013g_rvt_ss_1p08v_125c -lib_cell
BUFX2M -pin Y [get_port i_mem_write_done]
set_driving_cell -library scmetro_tsmc_cl013g_rvt_ss_1p08v_125c -lib_cell
BUFX2M -pin Y [get_port i_block_from_axi_data_cache]
set_driving_cell -library scmetro_tsmc_cl013g_rvt_ss_1p08v_125c -lib_cell
BUFX2M -pin Y [get_port i_mem_done]
set_driving_cell -library scmetro_tsmc_cl013g_rvt_ss_1p08v_125c -lib_cell
BUFX2M -pin Y [get_port i_block_from_axi_i_cache]
set_driving_cell -library scmetro_tsmc_cl013g_rvt_ss_1p08v_125c -lib_cell
BUFX2M -pin Y [get_port uart_ready]

################################################################################
           ############################################################
                  #### Section 5 : Output load ####
           ############################################################
################################################################################

set_load 0.5 [get_port o_riscv_core_ack]
set_load 0.5 [get_port mem_read_address]
set_load 0.5 [get_port o_mem_write_data]
set_load 0.5 [get_port o_mem_write_address]
set_load 0.5 [get_port mem_read_req]
set_load 0.5 [get_port o_mem_write_valid]
set_load 0.5 [get_port o_mem_write_strobe]
set_load 0.5 [get_port o_addr_from_control_to_axi]
set_load 0.5 [get_port o_mem_req]
set_load 0.5 [get_port uart_out_data]
set_load 0.5 [get_port uart_valid]

################################################################################
           ############################################################
                 #### Section 6 : Operating Condition ####
           ############################################################
################################################################################

# Define the Worst Library for Max(#setup) analysis
# Define the Best Library for Min(hold) analysis

set_operating_conditions -min_library "scmetro_tsmc_cl013g_rvt_ff_1p32v_m40c"
-min "scmetro_tsmc_cl013g_rvt_ff_1p32v_m40c" -max_library
"scmetro_tsmc_cl013g_rvt_ss_1p08v_125c" -max
"scmetro_tsmc_cl013g_rvt_ss_1p08v_125c"
```

- **Section 0: DC Variables**

This section sets a Design Compiler variable to avoid using assign statements in the generated netlist. It ensures that multiple port nets are handled correctly by buffering constants and feedthroughs.

- **Section 1: Clock Definition**
  This section defines the master clock, clock uncertainties, and transitions. Master Clock Definitions: It sets the clock name, period, setup skew, and hold skew, then creates the clock and sets the clock uncertainties.
- **Section 2: Clocks Relationship**
  This section is a placeholder for defining the relationships between different clocks.
- **Section 3: Set Input/Output Delay on Ports**
  This section sets input and output delays for different ports based on the clock period. The delays are set to 20% of the clock period.
- **Section 4: Driving Cells**
  This section specifies the driving cells for various ports, defining which cells are used to drive the signals for these ports.
- **Section 5: Output Load**
  This section sets the output load for different ports. The load is set to 0.5 for each port.
- **Section 6: Operating Condition**
  This section defines the operating conditions for the synthesis process. It specifies the libraries for the worst and best-case scenarios for timing analysis.

10.2.1.5. Design Compiler Analysis

The following TCL script part shows DC analysis:

```
###################### Defining toplevel ###################################

current_design $top_module

################### Liniking All The Design Parts ########################
puts "################################################"
puts "######## Liniking All The Design Parts ########"
puts "################################################"

link

################### Liniking All The Design Parts ########################
puts "################################################"
puts "######## checking design consistency ##########"
puts "################################################"

check_design >> reports/check_design.rpt

################### Define Design Constraints ########################
```

```
puts "################################################"
puts "############ Design Constraints #### ##########"
puts "################################################"

source -echo ./cons.tcl

###################### Mapping and optimization ######################
puts "################################################"
puts "########## Mapping & Optimization #############"
puts "################################################"


compile -map_effort medium

#################### Close Formality Setup file ##########################

set_svf -off

#############################################################################
# Write out files
#############################################################################
write_file -format verilog -hierarchy -output netlists/$top_module.ddc
write_file -format verilog -hierarchy -output netlists/$top_module.v
write_sdf  sdf/$top_module.sdf
write_sdc  -nosplit sdc/$top_module.sdc

####################### reporting ########################################

report_area -hierarchy > reports/area.rpt
report_power -hierarchy > reports/power.rpt
report_timing -delay_type min -max_paths 20 > reports/hold.rpt
report_timing -delay_type max -max_paths 20 > reports/setup.rpt
report_clock -attributes > reports/clocks.rpt
report_constraint -all_violators -nosplit > reports/constraints.rpt
```

10.2.1.6. Design Schematic

To view design schematic, use the command *gui_start.*

Figure 83 shows the schematic of the design.

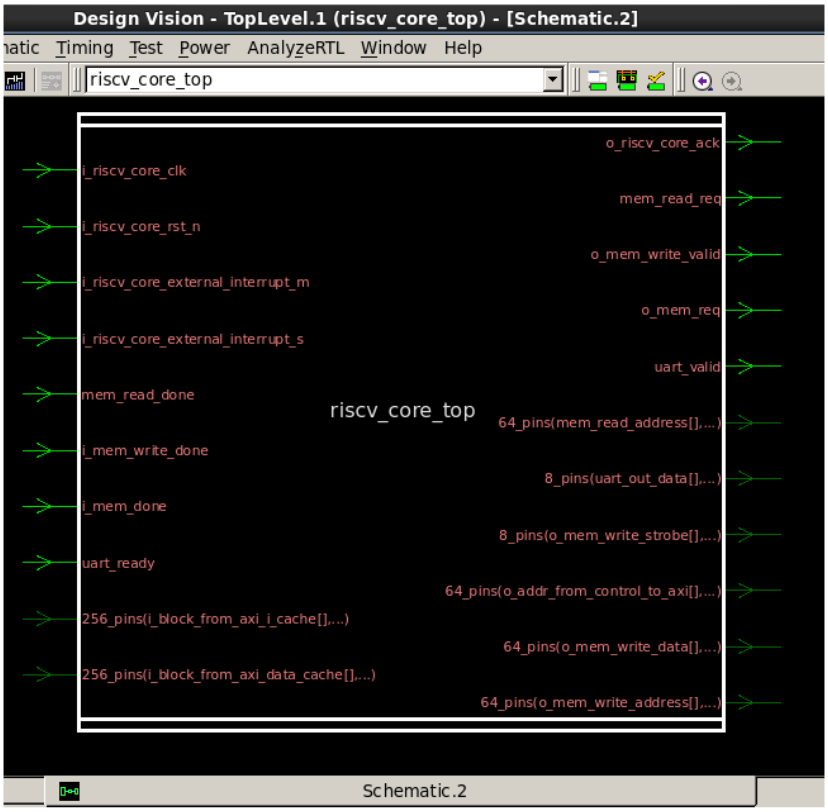

Figure 83. Design Schematic.

10.2.1.7. Generated Reports

- **Setup Report**

  The setup report is shown in Figure 84.

```
20  Endpoint: u_riscv_core_pipe_rf_rd1_id_ex/o_pipe_out_reg[19]
21            (rising edge-triggered flip-flop clocked by i_riscv_core_clk)
22  Path Group: i_riscv_core_clk
23  Path Type: max
24
25  Point                                                   Incr       Path
26  -----------------------------------------------------------------------
27  clock i_riscv_core_clk' (rise edge)                     5.00       5.00
28  clock network delay (ideal)                             0.00       5.00
29  u_riscv_core_rf/rf_reg[0][19]/clocked_on (**SEQGEN**)
30                                                          0.00 #     5.00 r
31  u_riscv_core_rf/rf_reg[0][19]/Q (**SEQGEN**)            0.00       5.00 r
32  u_riscv_core_rf/C4318/Z_44 (*MUX_OP_32_5_64)            0.00       5.00 r
33  u_riscv_core_rf/C4316/Z_19 (*SELECT_OP_2.64_2.1_64)     0.00       5.00 r
34  u_riscv_core_rf/o_rf_rd1[19] (riscv_core_rf)            0.00       5.00 r
35  u_riscv_core_pipe_rf_rd1_id_ex/i_pipe_in[19] (riscv_core_pipe_W_PIPE_BUS64)
36                                                          0.00       5.00 r
37  u_riscv_core_pipe_rf_rd1_id_ex/C343/Z_19 (*SELECT_OP_2.64_2.1_64)
38                                                          0.00       5.00 r
39  u_riscv_core_pipe_rf_rd1_id_ex/o_pipe_out_reg[19]/next_state (**SEQGEN**)
40                                                          0.00       5.00 r
41  data arrival time                                                  5.00
42
43  clock i_riscv_core_clk (rise edge)                     10.00      10.00
44  clock network delay (ideal)                             0.00      10.00
45  clock uncertainty                                      -0.20       9.80
46  u_riscv_core_pipe_rf_rd1_id_ex/o_pipe_out_reg[19]/clocked_on (**SEQGEN**)
47                                                          0.00       9.80 r
48  library setup time                                      0.00       9.80
49  data required time                                                 9.80
50  -----------------------------------------------------------------------
51  data required time                                                 9.80
52  data arrival time                                                 -5.00
53  -----------------------------------------------------------------------
54  slack (MET)                                                        4.80
55
```

Figure 84. Setup Report.

- **Hold Report**

  The hold report is shown in Figure 85.

hold.rpt

```
 26  ---------------------------------------------------------------------------
 27  clock i_riscv_core_clk (rise edge)                      0.00       0.00
 28  clock network delay (ideal)                             0.00       0.00
 29  u_riscv_core_csr_unit/counter_reg[19]/clocked_on (**SEQGEN**)
 30                                                          0.00 #     0.00 r
 31  u_riscv_core_csr_unit/counter_reg[19]/Q (**SEQGEN**)
 32                                                          0.00       0.00 r
 33  u_riscv_core_csr_unit/add_927/A_19 (*ADD_UNS_OP_64_1_64)
 34                                                          0.00       0.00 r
 35  u_riscv_core_csr_unit/add_927/*cell*550/A[19] (DW01_inc_width64)
 36                                                          0.00       0.00 r
 37  ...
 38  u_riscv_core_csr_unit/add_927/*cell*550/SUM[19] (DW01_inc_width64)
 39                                                          0.00       0.00 r
 40  u_riscv_core_csr_unit/add_927/Z_19 (*ADD_UNS_OP_64_1_64)
 41                                                          0.00       0.00 r
 42  u_riscv_core_csr_unit/counter_reg[19]/next_state (**SEQGEN**)
 43                                                          0.00       0.00 r
 44  data arrival time                                                      0.00
 45
 46  clock i_riscv_core_clk (rise edge)                      0.00       0.00
 47  clock network delay (ideal)                             0.00       0.00
 48  clock uncertainty                                       0.10       0.10
 49  u_riscv_core_csr_unit/counter_reg[19]/clocked_on (**SEQGEN**)
 50                                                          0.00       0.10 r
 51  library hold time                                       0.00       0.10
 52  data required time                                                     0.10
 53  ---------------------------------------------------------------------------
 54  data required time                                                     0.10
 55  data arrival time                                                      0.00
 56  ---------------------------------------------------------------------------
 57  slack (VIOLATED)                                                     -0.10
 58
```

Figure 85. Hold Report.

- **Power Report**
  The power report is shown in Figure 86.

```
 25 Global Operating Voltage = 1.08
 26 Power-specific unit information :
 27     Voltage Units = 1V
 28     Capacitance Units = 1.000000pf
 29     Time Units = 1ns
 30     Dynamic Power Units = 1mW    (derived from V,C,T units)
 31     Leakage Power Units = 1pW
 32
 33
 34 --------------------------------------------------------------------------------
 35                                      Switch   Int      Leak     Total
 36 Hierarchy                            Power    Power    Power    Power    %
 37 --------------------------------------------------------------------------------
 38 riscv_core_top                        134.142    0.000    0.000  134.142 100.0
 39   u_riscv_core_hazard_unit (riscv_core_hazard_unit)
 40                                         0.000    0.000    0.000    0.000   0.0
 41     C74 (*SELECT_OP_3.2_3.1_2)          0.000    0.000    0.000    0.000   0.0
 42     C73 (*SELECT_OP_3.2_3.1_2)          0.000    0.000    0.000    0.000   0.0
 43     eq_101_3 (*EQ_UNS_OP_5_5_1)         0.000    0.000    0.000    0.000   0.0
 44     eq_101_2 (*EQ_UNS_OP_5_5_1)         0.000    0.000    0.000    0.000   0.0
 45     eq_86 (*EQ_UNS_OP_5_5_1)            0.000    0.000    0.000    0.000   0.0
 46     eq_82 (*EQ_UNS_OP_5_5_1)            0.000    0.000    0.000    0.000   0.0
 47     eq_72 (*EQ_UNS_OP_5_5_1)            0.000    0.000    0.000    0.000   0.0
 48     eq_68 (*EQ_UNS_OP_5_5_1)            0.000    0.000    0.000    0.000   0.0
 49   u_riscv_core_mux2x1_stg2_instr_csr (riscv_core_mux2x1_XLEN32)
 50                                         0.000    0.000    0.000    0.000   0.0
```

Figure 86. Power Report.

- **Area Report**
  The area report is shown in Figure 87.

```
 6 ****************************************
 7 Report : area
 8 Design : riscv_core_top
 9 Version: K-2015.06
10 Date   : Sat Jun 22 06:50:05 2024
11 ****************************************
12
13 Library(s) Used:
14
15     scmetro_tsmc_cl013g_rvt_ss_1p08v_125c (File: /home/IC/tsmc_fb_cl013
16
17 Number of ports:                         7209
18 Number of nets:                         28810
19 Number of cells:                        18974
20 Number of combinational cells:          15095
21 Number of sequential cells:              3842
22 Number of macros/black boxes:               0
23 Number of buf/inv:                       1358
24 Number of references:                      27
25
26 Combinational area:             208579.490227
27 Buf/Inv area:                     4982.147931
28 Noncombinational area:           89004.410942
29 Macro/Black Box area:                0.000000
30 Net Interconnect area:      undefined  (No wire load specified)
31
32 Total cell area:                297583.901170
33 Total area:                 undefined
34
35 Hierarchical area distribution
36 ------------------------------
37
```

Figure 87. Area Report.

10.2.2. Formal Verification

Formal verification is an alternative to verification through simulation.

As designs become larger and more complex and require more simulation vectors, regression testing with traditional simulation tools becomes a bottleneck in the design flow.

A 100% coverage Equivalence checkers prove or disprove that one design representation is logically equivalent to another. In other words, two circuits exhibit the same exact behaviour under all conditions despite different representations.

The purpose of Formality is to detect unexpected differences that might have been introduced into a design during development.

It uses a formal verification comparison engine to prove or disprove the equivalence of two given designs and presents any differences for follow on detailed analysis.

Figure 88 shows Equivalence Checking Verification process.

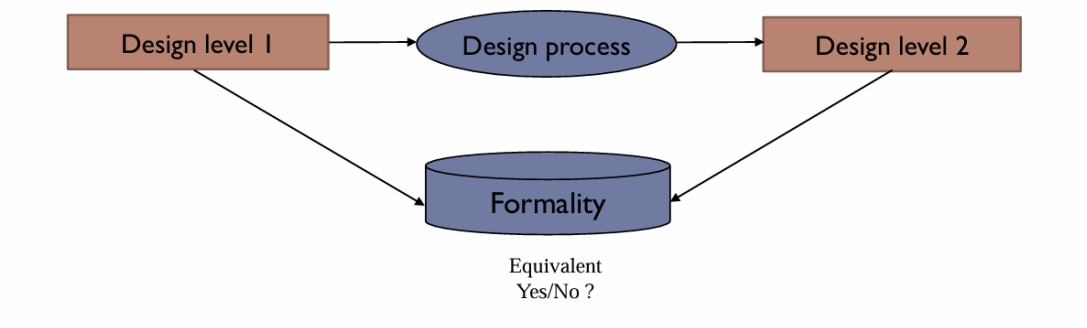

Figure 88. Equivalence Checking Verification Process.

Equivalence checking is a four-phase process:

- Reading and elaborating language descriptions into logical representations.
- Setting Up Designs to Preempt Differences.
- Mapping of corresponding compare points between pairs of designs (Matching).
- Comparison of logic cones that drive the compare points (Verification).

Formality Flow Overview Start Set is shown in Figure 89.

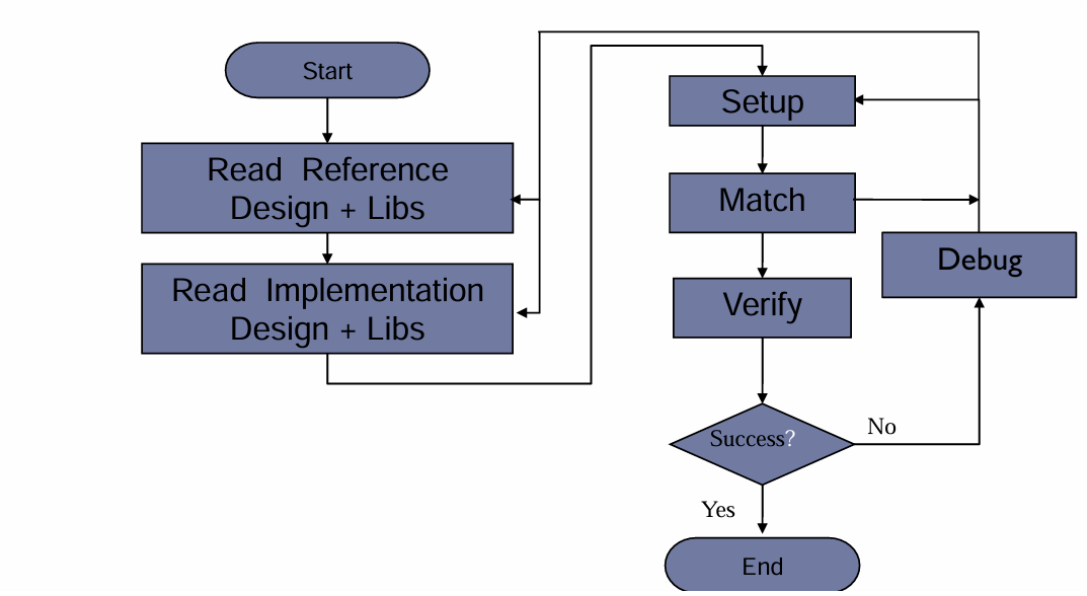

Figure 89. Formality Flow.

- **Load Automated Setup File:**
  Before specifying the reference and implementation designs, an automated setup file (.svf) can be optionally loaded into Formality. The automated setup file helps Formality process design changes caused by other tools used in the design flow. Formality uses this file to assist the compare point matching and verification process. For each automated setup file that is loaded, Formality processes the content and stores the information for use during the name-based compare point matching period.
  Loading automated setup file is shown in Figure 90 and Figure 91.

Figure 90. Load Automated Setup File (a)

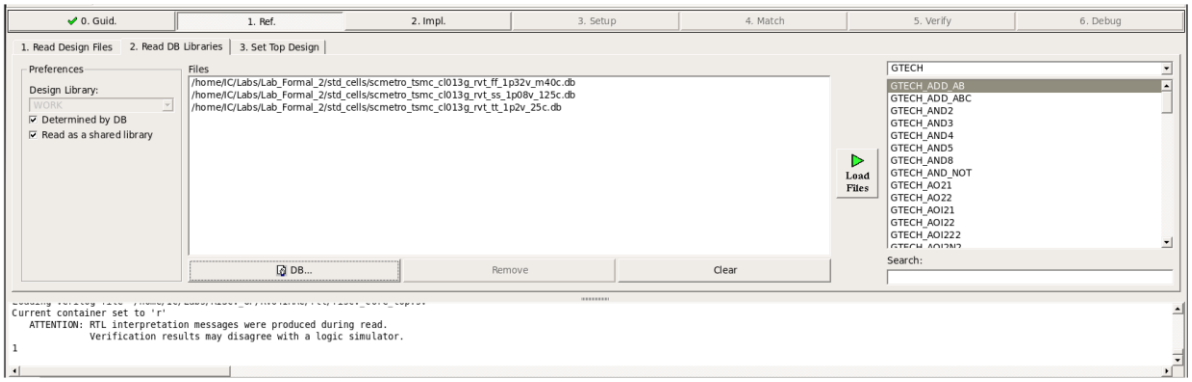

Figure 91. Load Automated Setup File (b)

- **Load Reference and Implementation Design:**
  Loading reference and implementation design is shown in Figure 92, Figure 93, Figure 94 and Figure 95.

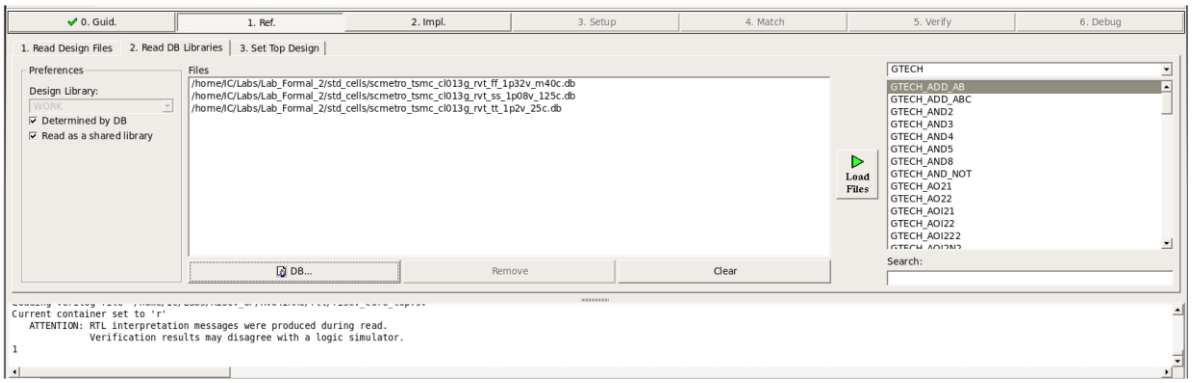

Figure 92. Load Reference and Implementation Design (a)

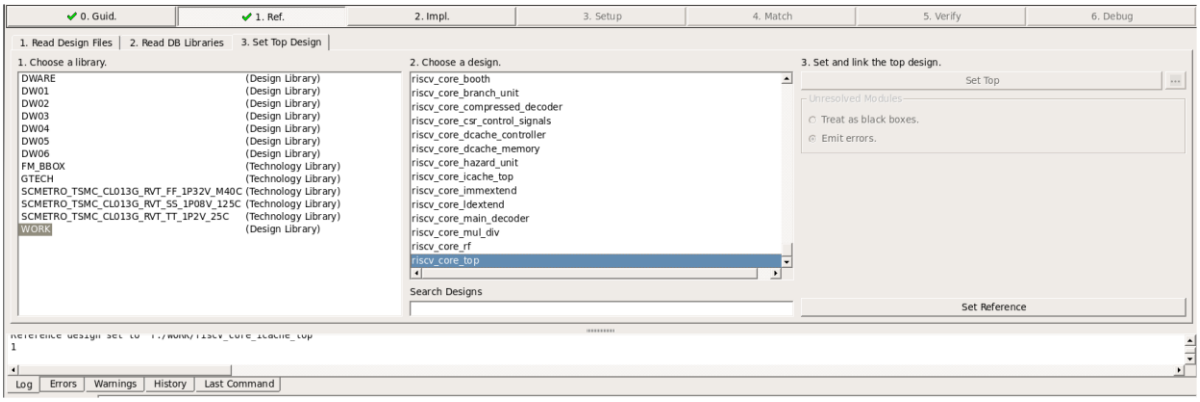

Figure 93. Load Reference and Implementation Design (b)

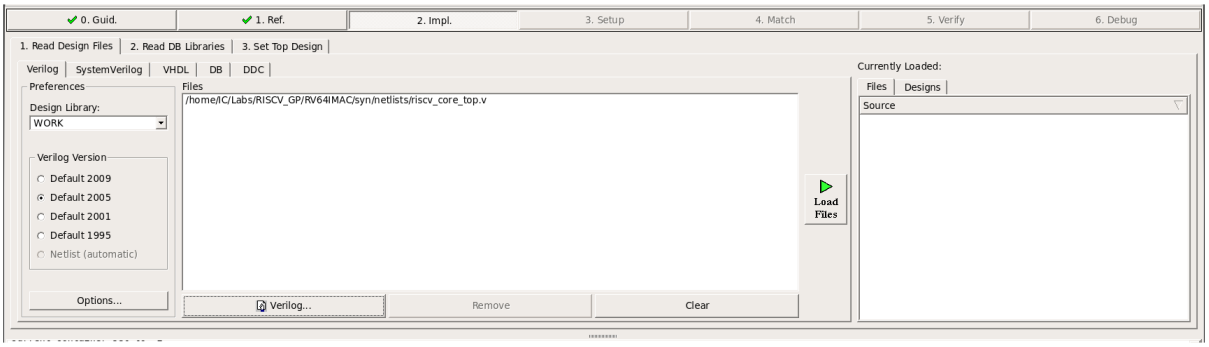

Figure 94. Load Reference and Implementation Design (c)

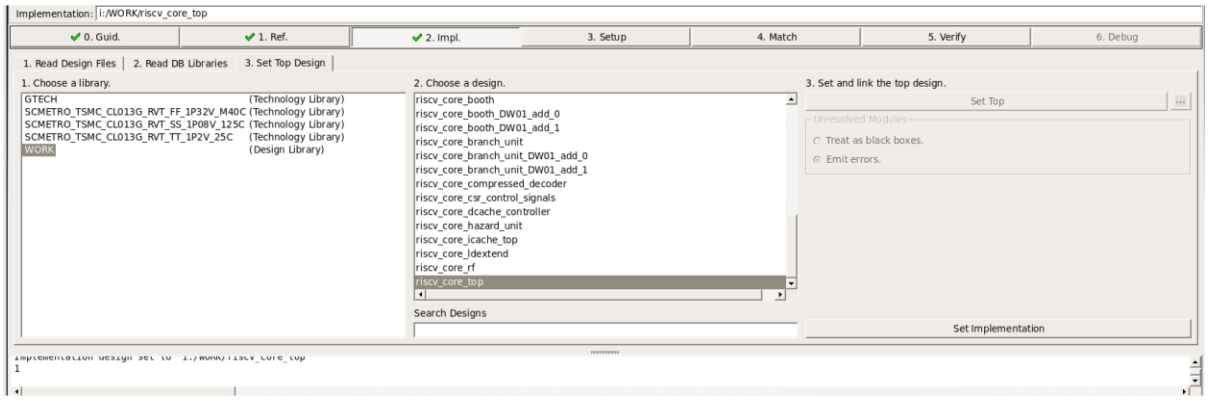

Figure 95. Load Reference and Implementation Design (d)

- **Matching and Verify:**
  Match step is shown in Figure 96.

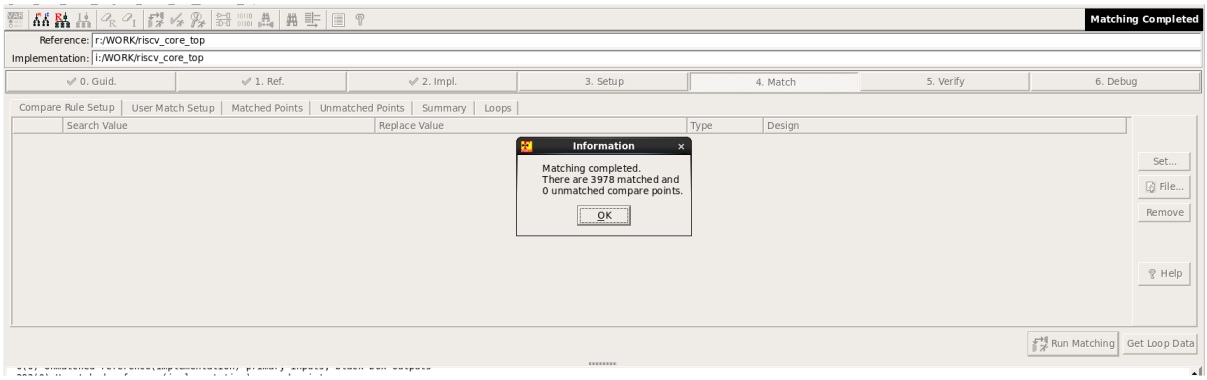

Figure 96. Match.

Verify Process is shown in Figure 97.

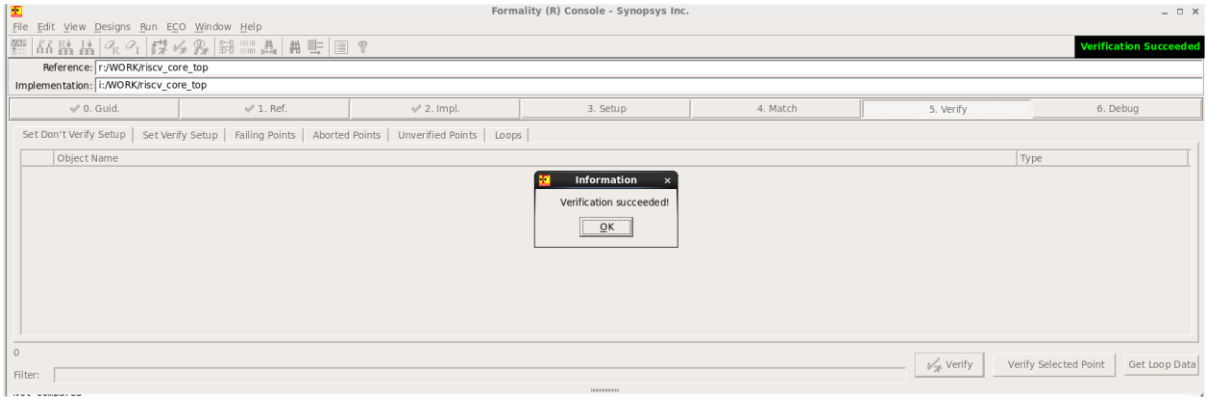

Figure 97. Verify.

10.2.3. DFT Insertion

The Design for Test (DFT) stage involves adding hardware components to the ASIC design to facilitate testing. DFT components include test points, scan chains, and boundary scan cells. DFT components are added to the design to enable efficient testing of the ASIC during production. These components are represented as additional signals added to top module of the design as follows:

```
//DFT
  input  logic                    SI,
  input  logic                    SE,
  input  logic                    scan_clk,
  input  logic                    scan_rst,
  input  logic                    test_mode,
  output logic                    SO,
```

```
//dft_scan_mux

mux2X1 U0_rst_mux(.IN_0(i_riscv_core_rst_n),
                  .IN_1(scan_rst),
                   .SEL(test_mode),
```

```
                  .OUT(scan_fun_rst)
);

mux2X1 U0_clk_mux(.IN_0(i_riscv_core_clk),
                  .IN_1(scan_clk),
                   .SEL(test_mode),
                   .OUT(scan_fun_clk)
);
```

The part of DFT TCL-script is shown below:

```
########################## Define DFT Signals ##########################
set_dft_signal -port [get_ports scan_clk] -type ScanClock -view existing_dft -
timing {30 60}
set_dft_signal -port [get_ports scan_rst] -type Reset -view existing_dft     -
active_state 0
set_dft_signal -port [get_ports test_mode] -type Constant -view existing_dft -
active_state 1
set_dft_signal -port [get_ports test_mode] -type TestMode -view spec         -
active_state 1
set_dft_signal -port [get_ports SE] -type ScanEnable      -view spec         -
active_state 1  -usage scan
set_dft_signal -port [get_ports SI] -type ScanDataIn      -view spec
set_dft_signal -port [get_ports SO] -type ScanDataOut     -view spec


############################## Create Test Protocol ####################
create_test_protocol


####################### Pre-DFT Design Rule Checking ###################
dft_drc -verbose


############################## Preview DFT #################################
preview_dft  -show scan_summary


############################## Insert DFT ##################################
insert_dft


######################### Optimize Logic post DFT ######################

compile -scan -incremental

###################### Design Rule Checking post DFT ###################
```

```
dft_drc -verbose -coverage_estimate
```

The summary report of DFT is shown Figure 98.

```
3820 ---------------------------------------------
3821
3822
3823     Uncollapsed Stuck Fault Summary Report
3824 ---------------------------------------------
3825 fault class                     code   #faults
3826 ------------------------------  ----  ---------
3827 Detected                         DT      15259
3828 Possibly detected                PT          0
3829 Undetectable                     UD        198
3830 ATPG untestable                  AU         57
3831 Not detected                     ND          8
3832 ---------------------------------------------
3833 total faults                                 15522
3834 test coverage                               99.58%
3835 ---------------------------------------------
3836  Information: The test coverage above may be inferior
3837               than the real test coverage with customized
3838               protocol and test simulation library.
3839 1
```

Figure 98. DFT Coverage Report.

10.2.4. Placement and Routing (PnR)

The Place and Route (PnR) phase in the ASIC (Application-Specific Integrated Circuit) design flow is a critical step where the physical layout of the circuit is generated from the synthesized netlist. This phase eventually leads to the creation of the final GDSII (GDS2) file, which is used for mask generation and chip fabrication. Here's a detailed overview of the steps involved in the PnR phase, leading to the generation of the GDS file:

1. **Floor planning**
    a. **Purpose**: Define the layout and allocate area for different functional blocks of the ASIC.
    b. **Tasks**:
        i. Define core and die area
        ii. Place macros (large blocks such as memory, IP blocks)
        iii. Create power grid and establish power planning
        iv. Define placement of I/O pins
2. **Placement**
    a. **Purpose**: Place standard cells in the defined floorplan.
    b. **Tasks**:
        i. Initial placement of standard cells
        ii. Legalization to ensure no cells overlap and are placed in legal positions
        iii. Optimization to minimize wirelength and improve timing
3. **Clock Tree Synthesis (CTS)**

a. **Purpose**: Design the clock distribution network to ensure the clock signal reaches all sequential elements with minimal skew and latency.
b. **Tasks**:
    i. Insert clock buffers and clock gates
    ii. Balance the clock tree to minimize skew
    iii. Optimize for clock power and timing

4. **Routing**

    a. **Purpose**: Establish electrical connections between placed cells according to the netlist.
    b. **Tasks**:
        i. Global Routing: Determine routing paths for each net at a coarse level
        ii. Detailed Routing: Precisely connect all pins and vias according to the global routing plan
        iii. Resolve DRC (Design Rule Checking) violations and optimize for signal integrity

5. **Optimization**

    a. **Purpose**: Optimize the design to meet timing, power, and area requirements.
    b. **Tasks**:
        i. Timing Optimization: Ensure all timing paths meet setup and hold constraints
        ii. Power Optimization: Reduce dynamic and static power consumption
        iii. Signal Integrity Optimization: Minimize issues like crosstalk, IR drop, and electromigration

6. **Physical Verification**

    a. **Purpose**: Verify that the physical design meets all manufacturing and performance requirements.
    b. **Tasks**:
        i. DRC: Check for geometrical and spacing rule violations
        ii. LVS (Layout vs. Schematic): Ensure the layout matches the logical design
        iii. Antenna Check: Ensure no excess charge buildup that can damage the gates during manufacturing

7. **Extraction**

    a. **Purpose**: Extract parasitic elements (resistance, capacitance) from the physical layout.
    b. **Tasks**:
        i. Perform parasitic extraction to create an accurate RC model of the interconnect
        ii. Generate extracted netlist for further analysis

The PnR design flow is shown in Figure 99.

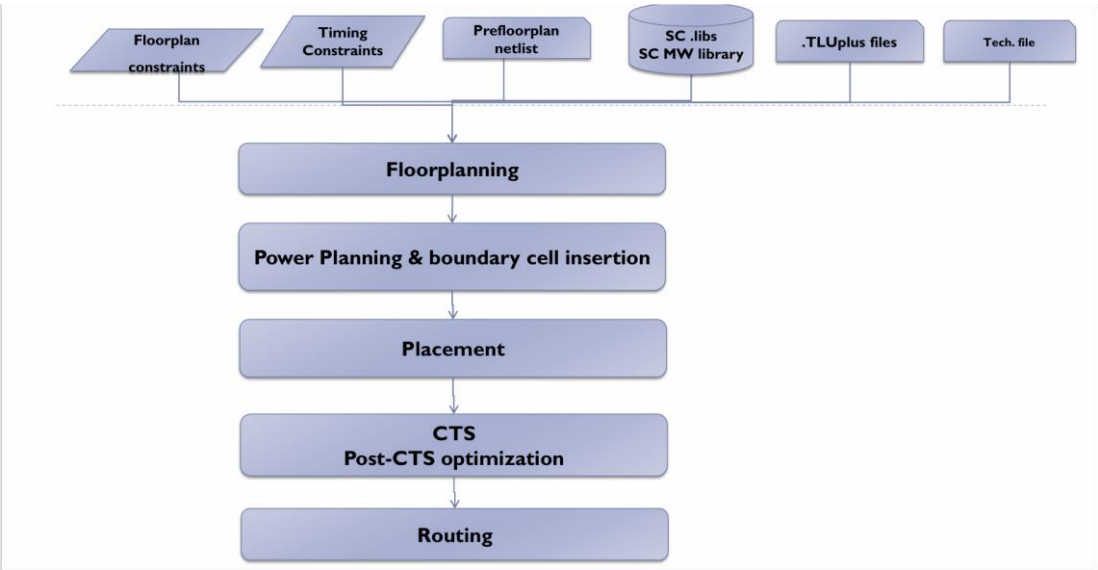

Figure 99. PnR Design Flow.

10.2.4.1. Design Import

The TCL script to import design is shown below:

```
################################## Variables ####################################

set load_fp 0

################## Define Top Module & Gate Level Netlist #######################
set top_module risc_core_top
set gate_level_netlist "/home/dft/risc_core_top.v"

setUIVar rda_Input ui_topcell $top_module
setUIVar rda_Input ui_netlist $gate_level_netlist

###################### Define Tech Libraries Timing Views #######################
set FFLIB "/home/std_cells/libs/scmetro_tsmc_cl013g_rvt_ff_1p32v_m40c.lib"
set SSLIB "/home/std_cells/libs/scmetro_tsmc_cl013g_rvt_ss_1p08v_125c.lib"
set TTLIB "/home/std_cells/libs/scmetro_tsmc_cl013g_rvt_tt_1p2v_25c.lib"

setUIVar rda_Input ui_timelib,min $FFLIB
setUIVar rda_Input ui_timelib,max $SSLIB
setUIVar rda_Input ui_timelib $TTLIB

###################### Define Tech Libraries LEF Views ##########################
set TECH_LEF    "/home/std_cells/lef/tsmc13fsg_7lm_tech.lef"
set MACRO_LEF   "/home/std_cells/lef/tsmc13_m_macros.lef"
set DESIGN_LEF "/home/pnr/import/ALU.lef" #only ex for hard macro

if {$load_fp == 0} {

setUIVar rda_Input ui_leffile [list $TECH_LEF $MACRO_LEF $DESIGN_LEF]
```

```
} else {

setUIVar rda_Input ui_leffile [list $TECH_LEF $MACRO_LEF]

}
######################### Define Capacitance Table ##############################
set CAP_TABLE_FILE  "/home/std_cells/captables/tsmc13fsg.capTbl"

setUIVar rda_Input ui_captbl_file $CAP_TABLE_FILE

########################## Define Constraints File ##############################
set SDC_FILE "/home/dft/risc_core_top.sdc"

setUIVar rda_Input ui_timingcon_file $SDC_FILE

############################## Define PG Pins  ##################################
set gnd_net_name VSS
set pwr_net_name VDD

setUIVar rda_Input ui_pwrnet $pwr_net_name
setUIVar rda_Input ui_gndnet $gnd_net_name

commitConfig

################ Define Multi Mode Multi Corner Analysis  ####################

source ./import/MMMC.tcl
```

10.2.4.2. Floor Planning

Floorplannning is deciding the major design objects size and placement.

The TCL script for floorplanning is shown below:

```
################################## Variables ####################################

set load_fp 0
set top_module riscv_core_top

########### Define Aspect Ratio (Length/Width) of Digital Macro  ############

if {$load_fp == 0} {

floorPlan -d 120.13 100.13 3.0 3.0 3.0 3.0

} else {
```

```
loadFPlan ./$top_module.fp

}
```

The Floorplan is shown in Figure 100.

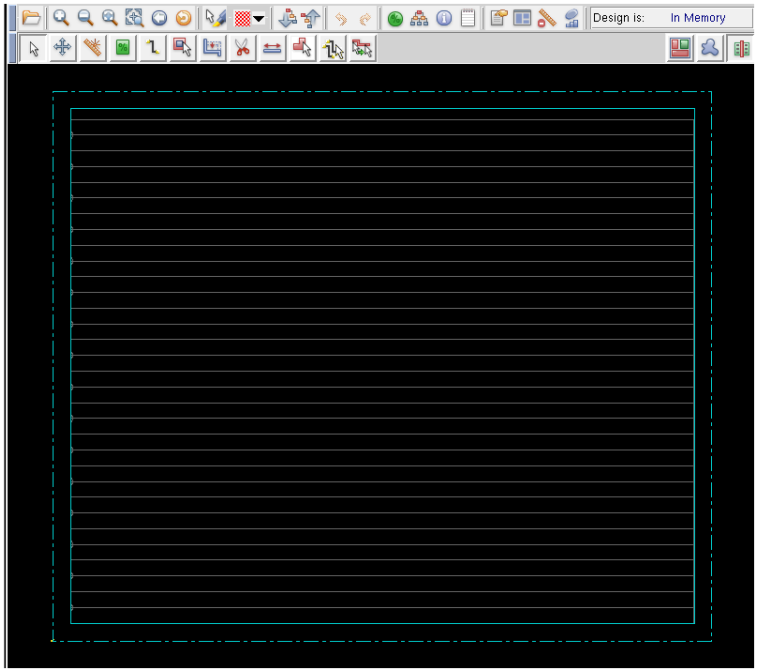

Figure 100. Floorplan.

10.2.4.3. Power Planning

Power planning is deciding how we will deliver power to the design's standard cells.

Steps of power planning is shown in Figure 101 and Figure 102.

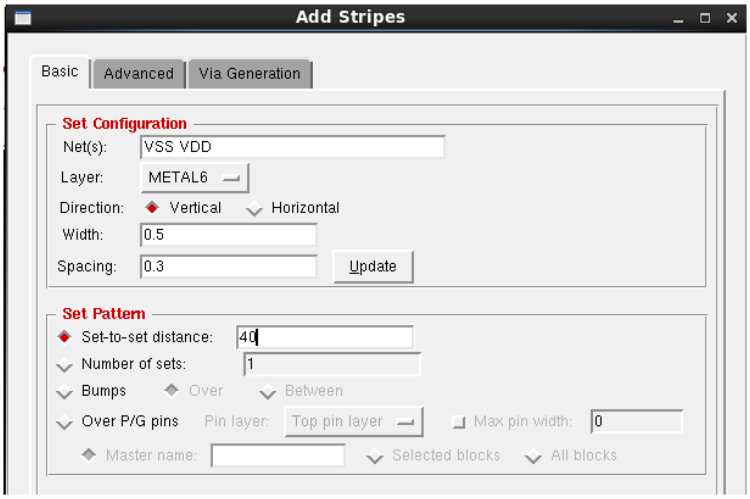

Figure 101. Power Planning (a)

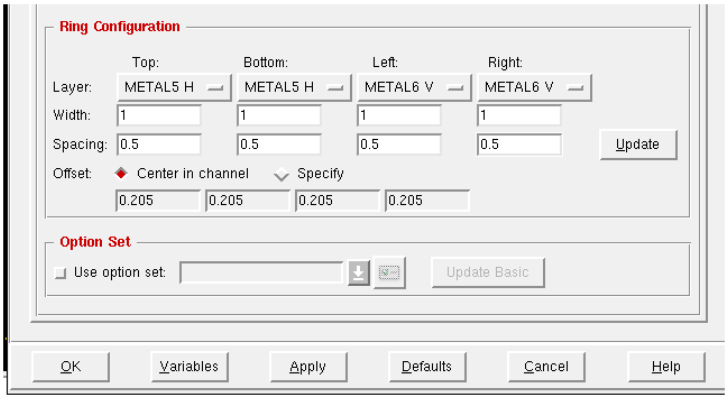

Figure 102. Power Planning (b)

The power plan is shown in Figure 103.

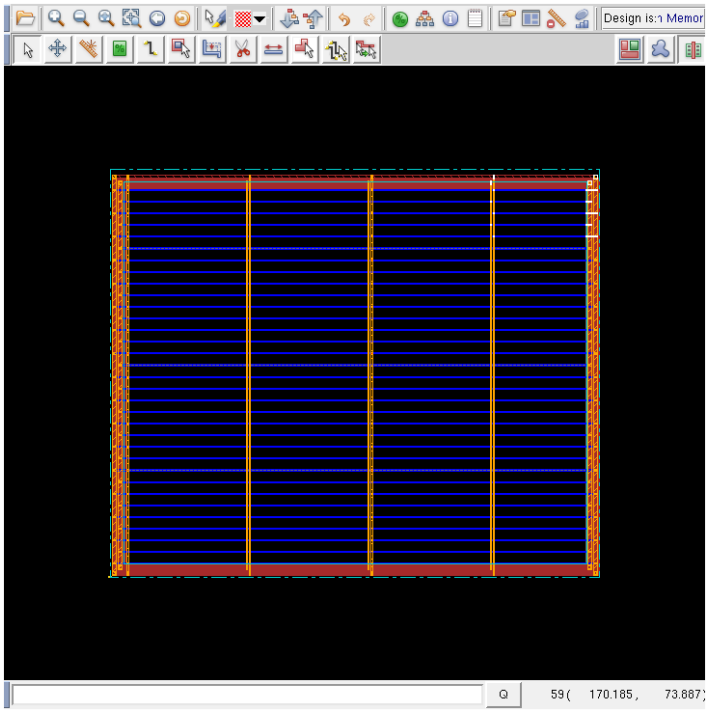

Figure 103. Power Planning.

10.2.4.4. Placement

Placement is the stage of the design flow, during which each instance (standard cell) is given an exact location.

The TCL script for placement is shown below:

```
######### Placement Standard Cells - TIE High/Low -- connect VDD/VSS ##########
placeDesign -inPlaceOpt -prePlaceOpt
addTieHiLo -cell TIELOM -prefix LTIE
addTieHiLo -cell TIEHIM -prefix HTIE
globalNetConnect VDD -type pgpin -pin VDD -inst *
globalNetConnect VSS -type pgpin -pin VSS -inst *
```

The chip after placement is shown in Figure 104.

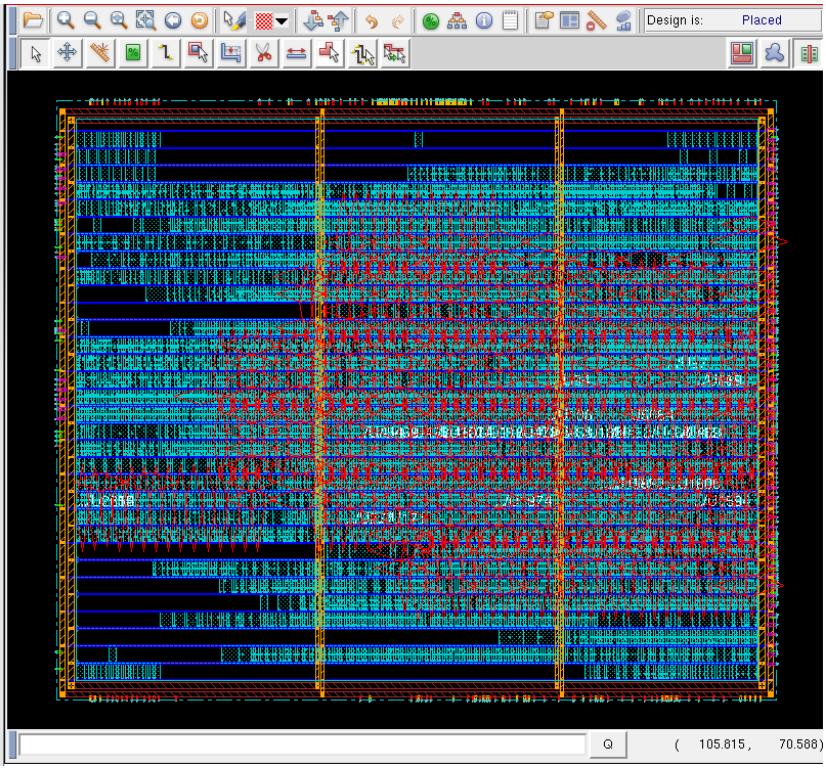

Figure 104. Placement.

10.2.4.5. Clock Tree Synthesis (CTS)

Analyze Timing pre-CTS for setup is shown in Figure 105.

WNS (ns):	48.120	98.783	64.137	48.120	N/A	N/A
TNS (ns):	0.000	0.000	0.000	0.000	N/A	N/A
Violating Paths:	0	0	0	0	N/A	N/A
All Paths:	195	38	118	39	N/A	N/A

DRVs	Real		Total	
	Nr nets(terms)	Worst Vio	Nr nets(terms)	
max_cap	40 (40)	-71.316	40 (40)	
max_tran	40 (40)	-44.067	40 (40)	
max_fanout	0 (0)	0	0 (0)	

Density: 84.838%

Figure 105. Setup Analysis Pre-CTS.

Analyze Timing post-CTS for setup is shown in Figure 106.

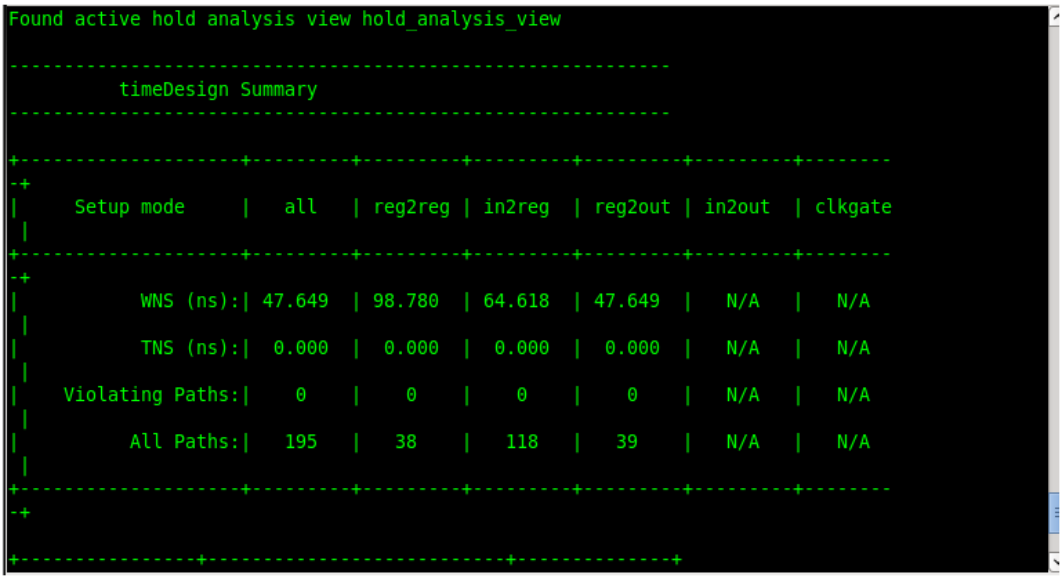

Figure 106. Setup Analysis Post-CTS.

Analyze Timing post-CTS for hold is shown in Figure 107.

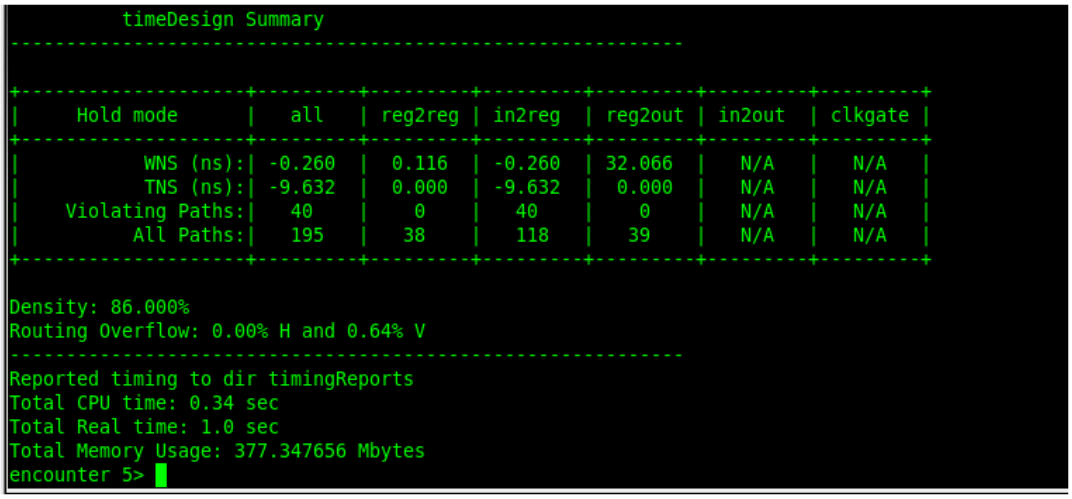

Figure 107. Hold Analysis Post-CTS

10.2.4.6. Routing

It creates physical connections to all clock and signal pins through metal interconnects Routed paths must meet setup and hold timing, max cap/trans, and clock skew requirements Metal traces must meet physical DRC requirements.

The TCL script for routing is shown below:

```
################################ Variables ###################################
set route_0_optimze_1 0

############ Global/Detialed Routing #############
set max_m_layer 6

if {$route_0_optimze_1 == 0} {
```

```
setNanoRouteMode -quiet -routeTopRoutingLayer $max_m_layer
setNanoRouteMode -quiet -routeBottomRoutingLayer default
setNanoRouteMode -quiet -drouteEndIteration default
setNanoRouteMode -quiet -routeWithTimingDriven false
setNanoRouteMode -quiet -routeWithSiDriven false
setNanoRouteMode -quiet -routeWithSiDriven true
setNanoRouteMode -quiet -routeSiEffort max

routeDesign -globalDetail -viaOpt -wireOpt

} else {

refinePlace -preserveRouting
setNanoRouteMode -routeWithEco true
setNanoRouteMode -droutePostRouteSwapVia true
globalDetailRoute

}
```

The chip finish is shown in Figure 108.

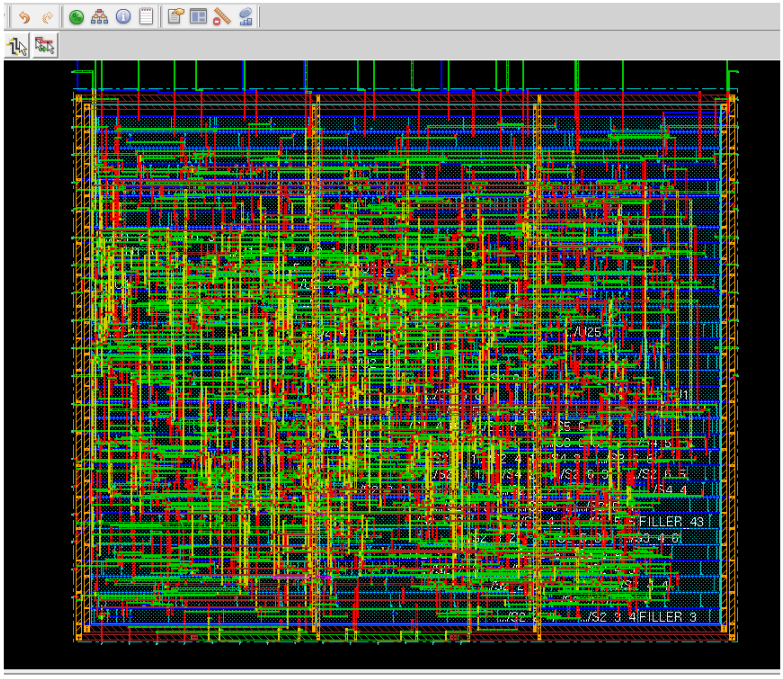

Figure 108. Chip Finish.

11. Conclusion

In conclusion, we have demonstrated the development of a RISC-V core with the IMAC 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 IMAC 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.

You can direct to our GitHub repository from here:

12. Future Work

As our target is to implement a high performance RV64IMAC 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.

References

[1] D. M. &. H. S. L. Harris, Digital Design and Computer Architecture, Morgan Kaufmann, 2013.

[2] “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA,” 2019. [Online].

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

[4] D. M. H. Neil H. E. Weste, CMOS VLSI Design: A Circuits and Systems Perspective, 4th edition ed., Pearson, 2015.

[5] J. L. H. David A. Patterson, Computer Organization and Design RISC-V Eddition: The Hardware/Software Interface, 2. edition, Ed., 2011 .

[6] I. Synopsys, “Design Compiler Guide,” 2011. [Online].

[7] I. Synopsys, “Synopsys Timing Constraints and Optimization User Guide,” 2010. [Online].

[8] A. J. Smith, “Cache memories,” *ACM Computing Surveys (CSUR),* vol. 14, pp. 473-530, 1982.

[9] Y. P. a. o. Chandra, “A Literature Survey on Cache,” *IJARIIE,* vol. 5, pp. 142-147, 2019.

[10] J. P. a. L. M. H. Shen, Modern processor design: fundamentals of superscalar, Waveland Press, 2013.

[11] B. a. W. D. a. N. S. Jacob, Memory systems: cache, DRAM, disk, Morgan Kaufmann, 2010.

[12] OpenHW, “COREV-DV,” OpenHW, 2021. [Online]. Available: https://docs.openhwgroup.org/projects/core-v-verif/en/latest/corev_dv.html. [Accessed 2024].

[13] C. Alliance, “RISCV-DV,” CHIPS Alliance, 2021. [Online]. Available: https://github.com/chipsalliance/riscv-dv. [Accessed 2024].

[14] “Ripes.me,” [Online]. Available: https://ripes.me/. [Accessed 2024].

[15] “Modular Multi-Ported SRAM-based Memories," in Proceedings of the 2014.