CO502 - Final Report

Group 1

Table of Contents

Introduction

This report documents the design, implementation, and evaluation of a five-stage pipelined RISC-V processor developed as part of the CO502 course project. The project involved building a simplified CPU architecture based on the RV32I instruction set, with support for basic arithmetic, memory, and control operations. The processor is designed using modular Verilog components and follows the standard five-stage pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).

The development process was carried out in multiple phases. Initially, individual hardware units such as the program counter, ALU, register file, and control unit were designed and verified. These units were then integrated into a complete pipeline, followed by the addition of hazard handling mechanisms to improve performance and maintain correct execution. Techniques such as data forwarding, hazard detection with stalling, and pipeline flushing for control hazards were implemented to handle data and control dependencies in the instruction stream.

A custom assembler was developed to translate RISC-V assembly programs into machine code compatible with the processor. The processor was verified using a variety of test programs that exercised key instruction types and hazard scenarios.

To further evaluate the design, the processor was synthesized using Synopsys RTLA and PrimePower. This allowed for detailed analysis of power consumption, area utilization, and timing performance. Additionally, the design was implemented on an FPGA platform to validate its functional behaviour in real hardware. This comprehensive development flow—from RTL design to ASIC-style synthesis and FPGA prototyping—highlights the design decisions, trade-offs, and practical considerations involved in building a pipelined RISC-V processor from scratch.

Hardware Design

RISC-V and RV32IM Instruction Set

This project implements a 32-bit RISC-V processor that supports the RV32IM instruction set architecture. RISC-V is a clean-slate, open ISA designed for extensibility and hardware simplicity. The RV32IM base includes:

- RV32I – The 32-bit integer base instruction set, providing arithmetic, logical, control flow, and memory access operations.
- RV32M – The standard integer multiplication and division extension, enabling instructions like MUL, DIV, and REM.

The ISA uses a small set of orthogonal instruction formats, which simplifies decoding and hardware implementation. These formats include:

- R-type: Used for register-register arithmetic and logical operations (e.g., ADD, SUB, AND, MUL).
- I-type: Used for register-immediate arithmetic, loads, and some control instructions (e.g., ADDI, LW, JALR).
- S-type: Used for store instructions (e.g., SW, SH, SB).
- B-type: Used for conditional branches (e.g., BEQ, BNE, BLT, BGE).
- U-type: Used for upper immediate instructions (e.g., LUI, AUIPC).
- J-type: Used for unconditional jumps (e.g., JAL).

Each instruction is 32 bits wide and includes fields such as opcode, rd, rs1, rs2, funct3, funct7, and various immediate encodings depending on the format. This regular encoding simplifies decoding and pipeline control.

The processor supports a wide range of operations across these instruction types, including:

- Arithmetic & Logical: ADD, SUB, AND, OR, XOR, SLT, SLL, SRL, SRA
- Immediate Operations: ADDI, ANDI, ORI, XORI, SLTI, SLLI, SRLI, SRAI
- Memory Operations: LW, SW, LB, LH, LHU, LBU, SB, SH
- Control Flow: BEQ, BNE, BLT, BGE, JAL, JALR
- Multiplication & Division (M extension): MUL, MULH, MULHU, DIV, DIVU, REM, REMU

The instruction decoding process uses the opcode to determine the instruction format and the operation type, while the funct3 and funct7 fields further refine the operation (particularly for R-type and M-extension instructions).

Instructions and Opcodes

Instruction	Description	Opcode	funct3	funct7	Notes
R-Type					
ADD	rd = rs1 + rs2	0110011	0x0	0x00	
SUB	rd = rs1 - rs2	0110011	0x0	0x20	
XOR	rd = rs1 ^ rs2	0110011	0x4	0x00	
OR	rd = rs1 | rs2	0110011	0x6	0x00	
AND	rd = rs1 & rs2	0110011	0x7	0x00	
SLL	rd = rs1 << rs2	0110011	0x1	0x00	
SRL	rd = rs1 >> rs2	0110011	0x5	0x00	
SRA	rd = rs1 >> rs2	0110011	0x5	0x20	Sign extend
SLT	rd = (rs1 < rs2) ? 1:0	0110011	0x2	0x00	
SLTU	rd = (rs1 > rs2) ? 1:0	0110011	0x3	0x00	
MUL	rd = (rs1 × rs2)[31:0]	0110011	0x0	0x01	
MULH	rd = (rs1 × rs2)[63:32]	0110011	0x1	0x01	
MULHSU	rd = (rs1 × rs2)[63:32]	0110011	0x2	0x01	
MULHU	rd = (rs1 × rs2)[63:32]	0110011	0x3	0x01	
DIV	rd = rs1 ÷ rs2	0110011	0x4	0x01	
DIVU	rd = rs1 ÷ rs2 (unsigned)	0110011	0x5	0x01	
REM	rd = rs1 % rs2	0110011	0x6	0x01	
REMU	rd = rs1 % rs2 (unsigned)	0110011	0x7	0x01	
I-Type					

ADDI	rd = rs1 + imm	0010011	0x0		Imm value is sign-extended
XORI	rd = rs1 ^ imm	0010011	0x4		Imm value is sign-extended
ORI	rd = rs1 | imm	0010011	0x6		Imm value is sign-extended
ANDI	rd = rs1 & imm	0010011	0x7		Imm value is sign-extended
SLLI	rd = rs1 << imm[4:0]	0010011	0x1	imm[5:11]=0x00	
SRLI	rd = rs1 >> imm[4:0]	0010011	0x5	imm[5:11]=0x00	
SRAI	rd = rs1 >> imm[4:0]	0010011	0x5	imm[5:11]=0x20	
SLTI	rd = (rs1 < imm)?1:0	0010011	0x2		Imm value is sign-extended
SLTIU	rd = (rs1 > imm)?1:0	0010011	0x3		Imm value is zero-extended
LB	rd = mem[rs1+imm][7:0]	0000011	0x0		
LH	rd = mem[rs1+imm][15:0]	0000011	0x1		
LW	rd = mem[rs1+imm][31:0]	0000011	0x2		
LBU	rd = mem[rs1+imm][7:0]	0000011	0x4		zero-extends
LHU	rd = mem[rs1+imm][15:0]	0000011	0x5		zero-extends
JALR	rd = PC+4; PC = rs1 + imm	1100111			
ECALL	Transfer control to OS	1110011	0x0	Imm = 0x0	
EBREAK	Transfer control to	1110011	0x0	Imm =	

	debugger			0x1	
S-Type					
SB	mem[rs1+imm][7:0] = rs2[7:0]	0100011	0x0		
SH	mem[rs1+imm][15:0] = rs2[15:0]	0100011	0x1		
SW	mem[rs1+imm][31:0] = rs2[31:0]	0100011	0x2		
SBU		0100011	0x4		
SHU		0100011	0x5		
U-Type					
LUI	rd = imm << 12	0110111	-		Load Upper Imm
AUIPC	rd = PC + (imm << 12)	0010111	-		Add Upper Imm to PC
B-Type					
BEQ	if(rs1 == rs2) PC += imm	1100011	0x0		
BNE	if(rs1 != rs2) PC += imm	1100011	0x1		
BLT	if(rs1 < rs2) PC += imm	1100011	0x4		
BGE	if(rs1 >= rs2) PC += imm	1100011	0x5		
BLTU	if(rs1 < rs2) PC += imm	1100011	0x6		zero-extends
BGEU	if(rs1 >= rs2) PC += imm	1100011	0x7		zero-extends
J-Type					
JAL	rd = PC+4; PC += imm	1101111	-		

FENCE		0001111	0x0		
FENCE.I		0001111	0x1		

Design Approach

The processor is designed to follow the classic 5-stage pipeline model:

1. Instruction Fetch (IF)
2. Instruction Decode/Register Fetch (ID)
3. Execution/Address Calculation (EX)
4. Memory Access (MEM)
5. Write Back (WB)

Each stage is isolated using pipeline registers, and the processor executes instructions in-order. While initial versions of the design relied on manually inserted NOPs to handle pipeline hazards, later phases introduce hazard detection and resolution units to automate this process and improve performance.

Testing and Instruction Coverage

To validate instruction execution and pipeline functionality, a range of test programs was written using the supported RV32IM instructions. These test cases were selected to:

- Exercise all instruction formats (R/I/S/B/U/J)
- Validate correct ALU behavior for both signed and unsigned operations
- Test control flow correctness with various branch and jump instructions
- Verify proper memory access and alignment
- Check correct pipeline behavior in presence of RAW hazards and load-use cases

The test programs were assembled using a custom assembler and simulated using Icarus Verilog. Waveform inspection via GTKWave confirmed correct instruction sequencing and output results, laying the foundation for further integration and hazard handling in subsequent phases.

Hardware Components

The RISC-V 5-stage pipelined processor design consists of several key hardware units, each responsible for specific operations within the pipeline stages. This section provides a detailed explanation of the function, design considerations, and limitations of these units.

Control Unit

The Control Unit is responsible for decoding the instruction opcode and generating the necessary control signals to direct the operation of other hardware components. It interprets inputs such as `OPCode`, `Func3`, and `Func5` to produce signals for register writes, memory access, ALU operations, and branch decisions. The unit also determines the immediate type for instruction decoding. Control signals are generated with a fixed delay of 3 time units.

Register File

The Register File comprises 32 general-purpose registers (x0–x31), each 32 bits wide. It supports two read ports and one write port, allowing simultaneous reading of source operands and writing of results. Read operations occur asynchronously with a 1-unit delay, while writes are synchronized with the clock's negative edge and incur a 2-unit delay. The design assumes external hazard handling and only supports one write operation per cycle.

Integer ALU

The ALU performs arithmetic, logical, and comparison operations during the Execute (EX) stage. It processes two operands and uses a 5-bit control signal (ALUOP) to determine the operation. In addition to basic operations like addition and subtraction, it supports multiply and divide instructions conforming to RV32IM standards. The design optimizes for low power consumption by isolating unused units and reducing switching activity. Limitations include high latency for complex operations and lack of support for vector or parallel computations.

Branch Logic

Branch Logic evaluates conditional and unconditional branch instructions in the EX stage. It uses combinational logic to perform comparisons and generate control signals that determine the next program counter value. The logic supports signed and unsigned comparisons, jump instructions, and conditional branches. Although it computes all comparisons in parallel for speed, it lacks speculative execution and early branch resolution, which limits instruction throughput in certain scenarios.

Data Memory

The Data Memory unit handles load and store operations during the Memory Access (MA) stage. It consists of a 256-byte array organized in little-endian format, supporting byte, halfword, and word access. Load instructions automatically handle sign extension when necessary. Each memory operation takes 2 time units. While suitable for basic testing and demonstration, the small size and absence of a caching mechanism restrict its scalability for larger applications.

Utilities and Supporting Components

Several supporting modules enhance data routing and computation:

Multiplexers: Used throughout the pipeline to select between alternative data sources (e.g., register values, immediate values, PC, ALU outputs). Both 2-to-1 and 3-to-1 multiplexers are used in different contexts such as ALU inputs and write-back selection.

Adders: Specialized adders are used for computing PC + 4, branch targets, and offset addresses during instruction execution and memory access stages.

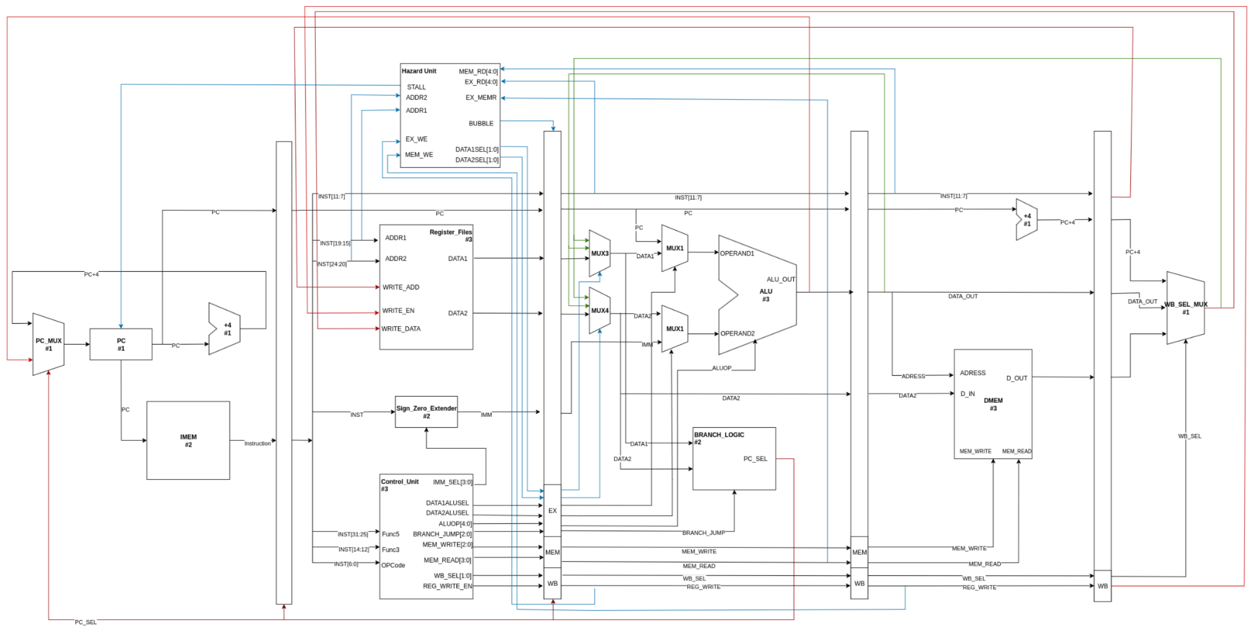
Together, these hardware units form a modular, pipelined processor architecture that conforms to RISC-V standards. While effective in implementing core instruction functionality, future improvements are required to handle pipeline hazards, stalls, and more advanced features such as exception handling and speculative execution.

Processor Integration

The processor integration phase represents a critical step in the development of a fully functional five-stage pipelined architecture based on the RV32I instruction set. This process involved the meticulous assembly of previously designed and verified hardware modules, including the Program Counter (PC), Instruction Memory, Register File, Arithmetic Logic Unit (ALU), Control Unit, Pipeline Registers, and Hazard Handling Units. The integrated processor is engineered to execute instructions in-order across five distinct pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The RV32I instruction set, enabling a wide range of computational tasks while maintaining simplicity and efficiency.

Pipeline Stage Integration

To ensure smooth and synchronized operation across the pipeline, each stage is distinctly separated using dedicated pipeline registers. These registers serve as buffers that capture and transfer data and control signals between stages, mitigating data hazards and enabling parallel execution of instructions.



- Instruction Fetch (IF) Stage
  The IF stage retrieves the next instruction from the Instruction Memory (IMEM) using the current Program Counter (PC) value. It initiates the pipeline by fetching the 32-bit instruction at the PC address, incrementing the PC by 4 for the next

sequential instruction unless a branch or jump modifies it. Preliminary branch prediction logic may suggest an alternate PC value to enhance performance, though the actual branch resolution occurs later.

- IF/ID Register
  This register captures the fetched 32-bit instruction from IMEM and the corresponding PC value. It buffers this data to isolate the IF stage from the ID stage, enabling parallel operation. The PC value supports branch target calculation and debugging, while the instruction is passed for decoding. In case of a branch misprediction, the PC_SEL signal triggers a flush of this stage to discard incorrect instruction data and realign the pipeline.

- Instruction Decode (ID) Stage
  The ID stage interprets the fetched instruction, determining its type and preparing for execution. It decodes the opcode to generate control signals (e.g., RegWrite, MemRead, MemWrite, ALUOp) that direct the pipeline's operation. The stage extracts source register addresses and immediate values, and it also incorporates a forwarding unit to resolve data hazards. This forwarding mechanism allows the ID stage to use the most recent operand values from the EX/MEM or MEM/WB stages if an instruction depends on a result not yet written back to the Register File.

- ID/EX Register
  This register holds the decoded control signals (e.g., ALUOp, ALUSrc, MemRead, MemWrite, RegWrite, MemtoReg) generated by the Control Unit, along with the two source operands and the sign-extended immediate value. It ensures the ID stage operates independently of the EX stage, facilitating pipeline parallelism. If a branch misprediction is detected, the PC_SEL signal updates the PC to the correct address, allowing the stage to restart fetching from the appropriate location. And add a bubble if there is a true data dependency.

- Execute (EX) Stage
  The EX stage performs the core computations of the instruction using the Arithmetic Logic Unit (ALU). It processes arithmetic, logical, shift, and comparison operations based on the control signals and operands from the previous stage. For branch instructions, it resolves the condition and calculates the target address, feeding back the PC_SEL signal if the prediction was wrong. Multiplexers select the correct operands, including forwarded data, to handle dependencies efficiently.

- EX/MEM Register
  This register stores the ALU output, memory control signals (e.g., MemRead, MemWrite), and the destination register address. It buffers these results to decouple the EX stage from the MEM stage, allowing the ALU computation to proceed independently of memory operations. The register is updated and passed forward, with the mem control signals.

- Memory Access (MEM) Stage
  The MEM stage manages data transfers between the processor and Data Memory. It executes load and store instructions by reading from or writing to memory based on the ALU output (used as the memory address) and control signals. For non-memory instructions, it simply passes the ALU result to the next stage. This stage completes the data-handling portion of the pipeline, preparing the result for write-back.

- MEM/WB Register
  This register holds the final data to be written back, which can be the ALU result or memory read data, along with the destination register address and RegWrite signal. It separates the MEM stage from the WB stage, enabling the memory operation to complete independently. The buffered data is passed to the write-back stage for register updating.

- Write Back (WB) Stage
  The WB stage finalizes the instruction execution by writing the result back to the Register File. It selects between the ALU result (for arithmetic/logical operations) or memory data (for loads) using a multiplexer controlled by the MemtoReg signal. This stage updates the destination register specified by the instruction, completing the pipeline cycle and making the result available for future instructions.

Signal Wiring and Control Propagation

The proper functioning of the pipelined processor relies heavily on the accurate propagation of control signals and data across the stages. Control signals, which are generated in the ID stage by the Control Unit based on the opcode of the instruction, are routed through the pipeline registers to ensure they reach the appropriate stages at the correct time. For instance, signals such as RegWrite, MemRead, MemWrite, and ALUOp are critical for coordinating register file operations, memory accesses, and ALU functionality, respectively. The pipeline registers preserve these signals, allowing each stage to execute its designated task based on the instruction's requirements.

Multiplexer (MUX) Implementation and Data Forwarding

To handle the dynamic nature of instruction execution and mitigate data hazards, multiplexers (MUXes) are strategically inserted in the EX stage. These MUXes facilitate operand selection by choosing between different data sources, including:

- Data directly read from the Register File.
- Forwarded data from the EX/MEM or MEM/WB registers, which contain results from earlier instructions that have not yet been written back.

This forwarding mechanism is essential for maintaining pipeline efficiency, as it allows the processor to use the most recent values of operands without waiting for the write-back stage to complete. For example, if an instruction in the EX stage requires the result of a prior instruction still in the MEM stage, the MUX can select the forwarded ALU output from the EX/MEM register instead of the outdated value in the Register File.

Overall Architecture and Functionality

The integrated diagram illustrates the flow of data and control signals through the pipeline, with each module (e.g., IMEM, Register File, ALU) interconnected via well-defined signal paths. The PC, incremented or updated based on branch or jump instructions, drives the IF stage, while the Control Unit in the ID stage interprets the instruction and sets the pipeline in motion. The ALU in the EX stage performs computations, the MEM stage handles data transfers with the Data Memory, and the WB stage finalizes the operation by updating the Register File. The use of colored lines (e.g., red, blue, green) in the diagram highlights the separation of major data and control paths, enhancing the clarity of the design.

This comprehensive integration ensures that the processor can execute RV32IM instructions in a pipelined manner, achieving higher throughput by overlapping the execution of multiple instructions while managing hazards through forwarding and control signal propagation.

Hazard Handling

Pipeline Hazards Overview

Pipelined processors improve performance by executing multiple instructions concurrently across stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). However, hazards disrupt this flow, preventing instructions from executing in their designated clock cycles. Hazards are classified into three types:

- Structural Hazards: Occur when multiple pipeline stages compete for the same hardware resource, e.g., a shared memory unit for instruction fetch and data access.
- Data Hazards: Arise from dependencies between instructions. These include:
  - RAW (Read-After-Write): An instruction tries to read a register before a prior instruction writes to it (true dependence). This is the primary data hazard in our in-order pipeline.
  - WAR (Write-After-Read) and WAW (Write-After-Write): Relevant in out-of-order execution, not applicable in our in-order design.
- Control Hazards: Result from changes in instruction flow due to branches or jumps, leading to potential execution of incorrect instructions.

These hazards introduce pipeline bubbles—clock cycles with no useful work—reducing efficiency.

Hazard Resolution Techniques

To mitigate hazards, we implemented hardware-based solutions to ensure correct and efficient instruction execution without manual NOP insertion.

Forwarding (Bypassing) Mechanism

Forwarding resolves RAW hazards by routing results directly from intermediate pipeline stages to dependent instructions, avoiding stalls. Key forwarding paths include:

- EX/MEM → EX: Forwards data from the EX/MEM pipeline register to the ALU inputs in the EX stage.
- MEM/WB → EX: Forwards data from the MEM/WB pipeline register to the EX stage.

Detection Logic:

- Forwarding is triggered when the destination register of an instruction in the EX or MEM stage (with an active write-enable signal) matches the source register of the instruction in the ID/EX stage.
- Using negative-edge-triggered register file writes, RAW hazards are limited to a two-instruction window, reducing the need for stalls.

This mechanism ensures that dependent instructions use the most recent data, maintaining pipeline throughput.

Stall Logic (Hazard Detection Unit)

For cases where forwarding is insufficient, such as load-use hazards (e.g., a load instruction followed by an instruction using the loaded data), the hazard detection unit inserts a stall:

- Detection: Identifies when an instruction in the EX stage is a load and the next instruction in ID/EX requires its result.
- Action: Inserts a NOP (bubble) and holds the Program Counter (PC) and IF/ID registers for one cycle, allowing the load to complete.

This ensures correct data availability while minimizing performance impact.

Control Hazard Flushing

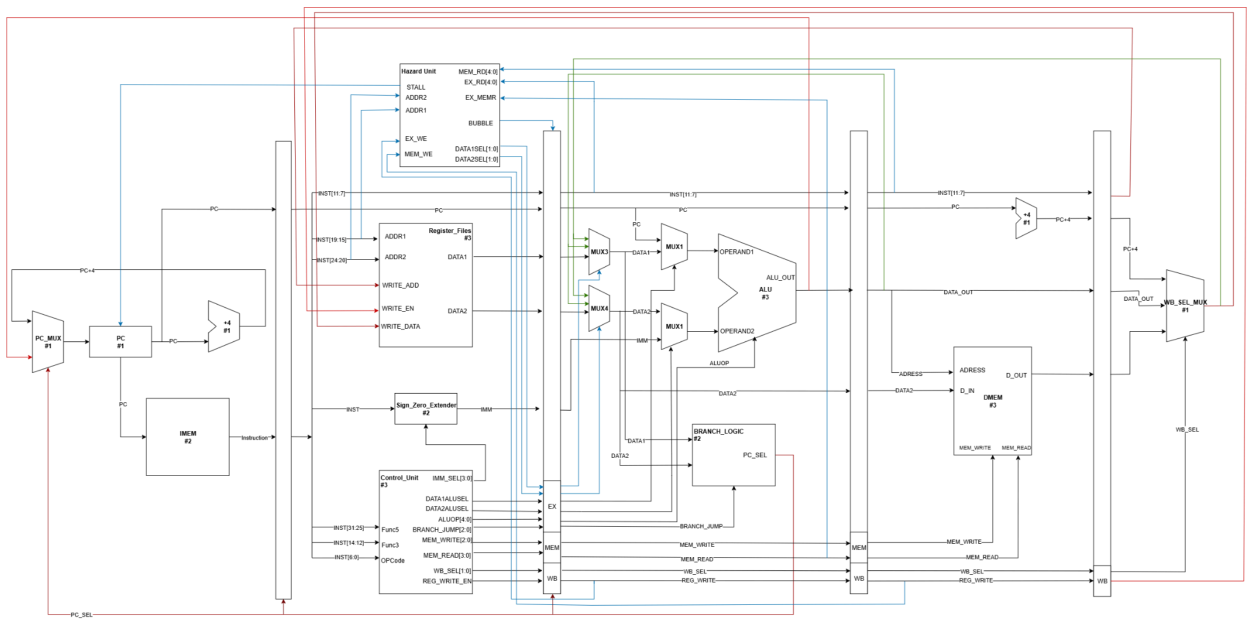
Control hazards occur when branches or jumps alter the instruction flow. Without branch prediction, we use a static "always not taken" policy and handle taken branches via pipeline flushing:

- Branch Decision: Resolved in the EX stage.
- Flush Action: If a branch is taken, instructions in the IF and ID stages are flushed by clearing the IF/ID and ID/EX pipeline registers, preventing incorrect instruction execution.

This approach incurs a 1-cycle penalty for taken branches but ensures correctness.

Updated Datapath

The datapath was enhanced with a Hazard Unit integrating forwarding, stall, and flush mechanisms to handle hazards dynamically.



Updated Overview Diagram

The updated datapath includes:

- Hazard Unit (in ID stage): Coordinates hazard detection and resolution.
    - Forwarding Unit (EX, MEM, WB stages): Routes recent data to ALU inputs via multiplexers (MUXes).
    - Hazard Detection Unit (ID, EX stages): Detects load-use hazards and triggers stalls.
- Flush Logic (Branch Unit) (EX, ID/IF stages): Clears pipeline registers for taken branches or jumps.

MUX Control and Signal Paths:

- MUXes at ALU inputs select between:
    - Register file output (default).
    - Forwarded data from EX/MEM or MEM/WB pipeline registers.
- Control signals (described below) manage operand selection and pipeline control.

Control Signals

- STALL: Asserted for load-use hazards, halting PC and IF/ID register updates for one cycle.
- BUBBLE: Inserts a NOP in the EX stage during a load-use hazard, preventing incorrect execution.

- FLUSH (PC_SEL): Clears IF/ID and ID/EX registers for taken branches/jumps and selects the next PC.
- DATA1SEL, DATA2SEL: 2-bit signals controlling ALU operand sources:
    - 00: Register file (no hazard).
    - 01: Forwarded from EX/MEM.
    - 10: Forwarded from MEM/WB.

These signals ensure the ALU receives correct operands and the pipeline handles hazards effectively.

Test and Validation

To verify hazard handling, we developed a test program targeting arithmetic dependencies, load-use hazards, and control hazards.

Test Program for Hazard Cases

The following RISC-V assembly code was simulated:

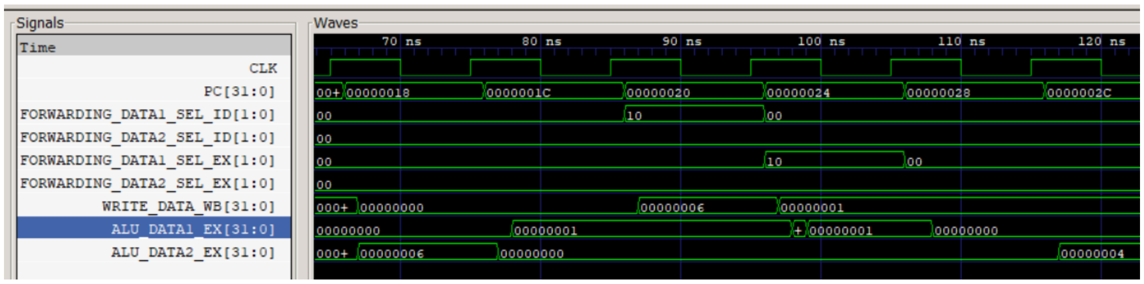
```
boot:
   addi x1, x0, 1    // x1 = 1
   addi x2, x0, 0    // x2 = 0
   addi x5, x0, 0    // x5 = 0 (store address base)
   addi x7, x0, 6    // x7 = loop counter = 6
Loop:
   addi x3, x1, 2    // x3 = x1 + 2 → RAW hazard with x1
   addi x2, x1, 0    // x2 = x1 → Forward x1
   addi x1, x3, 0    // x1 = x3 → Forward x3
   sw x3, 0(x5)      // store x3 → Memory write
   addi x5, x5, 4    // increment address
   addi x7, x7, -1   // decrement counter
   beq x7, x0, Done  // branch if x7 == 0 → Control hazard
   jal x0, Loop      // jump back to Loop
Done:
   lw x6, -4(x5)     // load last value → Load-use hazard
   add x1, x6, x6    // x1 = x6 + x6 → Forward from MEM/WB
   jal x0, boot      // restart program
```

Validation Through Waveform and Register Inspection

Test Cases:

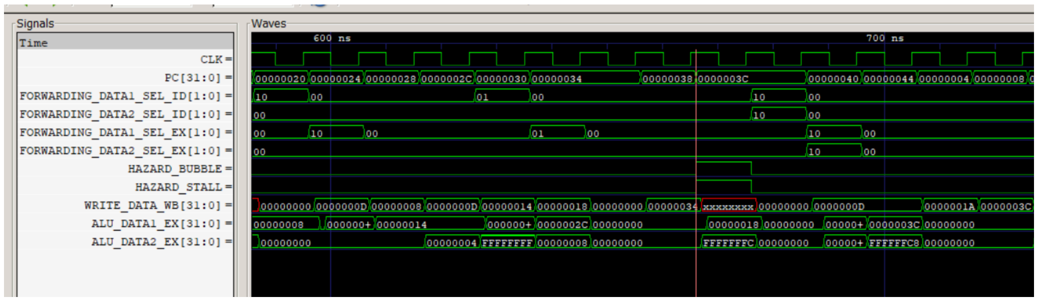
1. Arithmetic Dependency Test:

- Instructions: addi x3, x1, 2, addi x2, x1, 0, addi x1, x3, 0.
- Hazard: RAW hazards on x1 and x3.
- Result: Forwarding from EX/MEM and MEM/WB stages correctly provided ALU inputs, avoiding stalls.



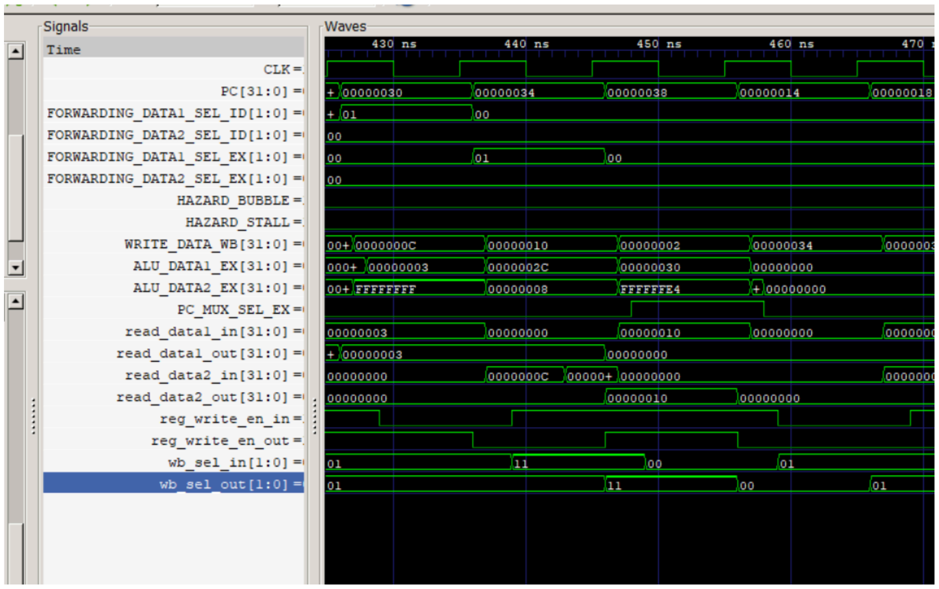
Waveform of Arithmetic Dependency Test

2. Load-Use Hazard Test:
   - Instructions: lw x6, -4(x5) followed by add x1, x6, x6.
   - Hazard: Load-use hazard requiring a stall.
   - Result: Hazard detection unit inserted a 1-cycle stall and a bubble in the EX stage, ensuring correct data availability.



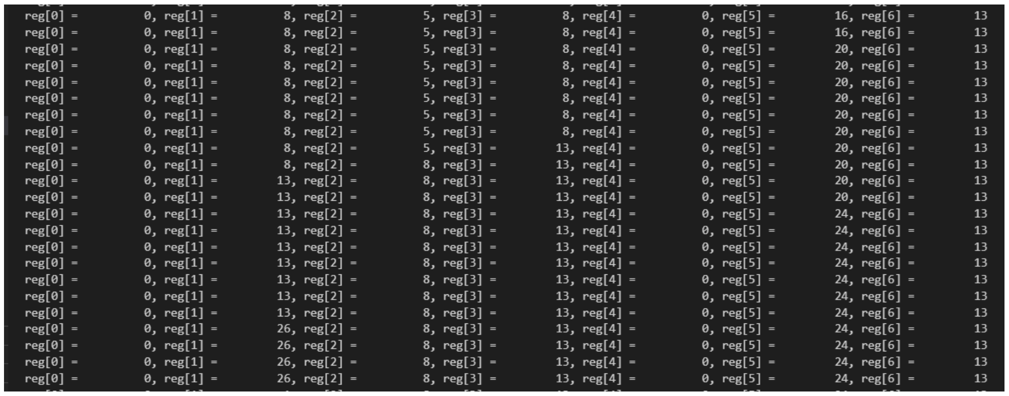
Waveform of Load Use Hazard

3. Control Hazard Test:
   - Instructions: beq x7, x0, Done, jal x0, Loop.
   - Hazard: Control hazard due to branch/jump.
   - Result: Flushing of IF and ID stages occurred for taken branches, preventing incorrect instruction execution.



Waveform of Control Hazard Test

Register Outputs:



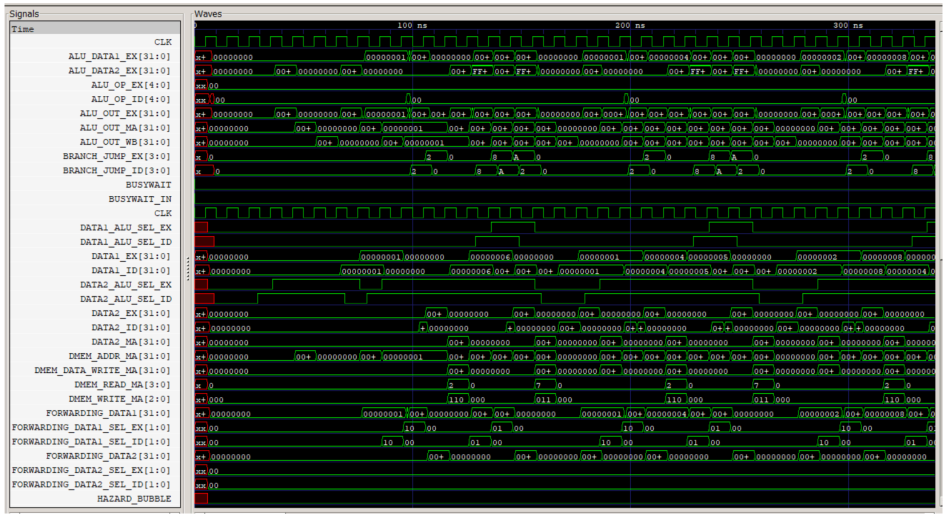
```
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         16, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         16, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =         13, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =          8, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         20, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         26, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         26, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         26, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
reg[0] =          0, reg[1] =         26, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =          13
```

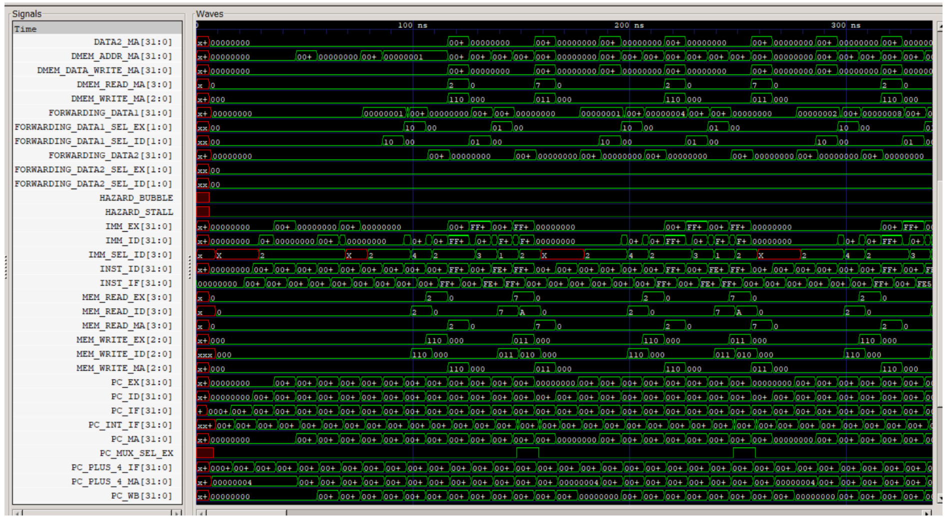
Register x1 evolved through the values 5, 8, 13, and 26 across loop iterations, confirming that data dependencies were resolved correctly. Register x6 consistently held the value 13 after the load instruction, validating proper handling of the load-use hazard. Meanwhile, register x5 is incremented by 4 in each loop iteration, effectively functioning as a memory pointer. Register x3 accurately captured the intermediate sums, demonstrating correct arithmetic operation sequencing and data forwarding.

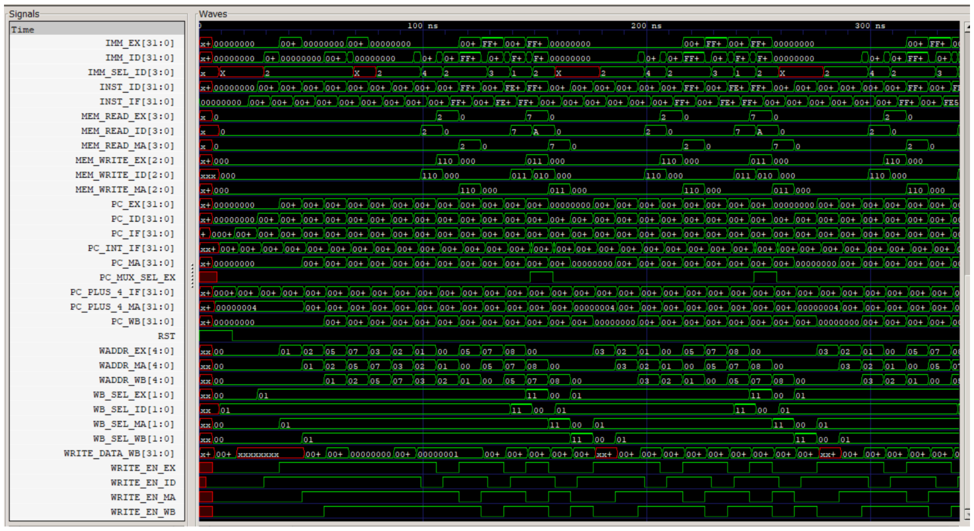
Correct Behaviour and Resolved Hazards

The test program executed correctly, with memory and register values matching expected outcomes. Forwarding eliminated unnecessary stalls for RAW hazards, while stalls and flushes handled load-use and control hazards, respectively. The absence of manual NOPs and consistent pipeline efficiency validated the hazard handling mechanisms.

Waveform Diagrams



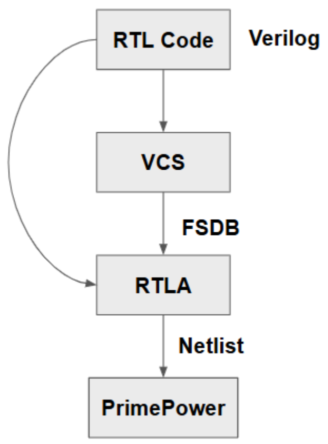




Synopsys Tools – Synthesis and Analysis

This section details the power, area, and timing analysis of the processor using the Synopsys suite of EDA tools. The overall workflow includes simulation, synthesis, and power analysis, along with a GitHub-based automation flow to streamline execution and result collection.

Synthesis Overview



The processor design, implemented in synthesizable Verilog HDL, was evaluated using industry-standard Synopsys tools. The analysis flow included the following steps:

- Simulation with Synopsys VCS:
The functional RTL design was simulated using Synopsys VCS. During simulation, switching activity was recorded using FSDB dump files. These capture real usage patterns required for accurate power estimation.

- Synthesis with RTL Architect:
The Verilog design was synthesized using Synopsys RTL Architect to generate a gate-level netlist. Default and recommended constraints (e.g., target clock frequency, area limits) were applied during synthesis to reflect realistic implementation parameters.

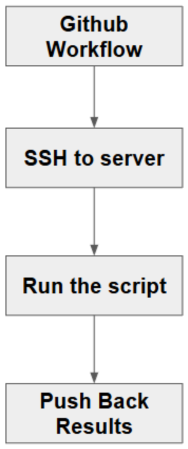
- Power Analysis with PrimePower:
Using the synthesized netlist and the activity file (FSDB), PrimePower was used to perform detailed power analysis. The tool provided breakdowns of dynamic, switching, and leakage power across hierarchical modules.

GitHub Automation Workflow

To streamline the process, a GitHub-based CI automation system was implemented:

- A GitHub Actions script triggers a remote login to the server where Synopsys tools are hosted.
- The server runs a predefined script that executes the VCS simulation, synthesis, and power analysis pipeline.
- Final logs, netlists, and power reports are pushed back to the GitHub repository automatically.

This setup ensures repeatability, transparency, and reduced manual intervention for analysis runs.

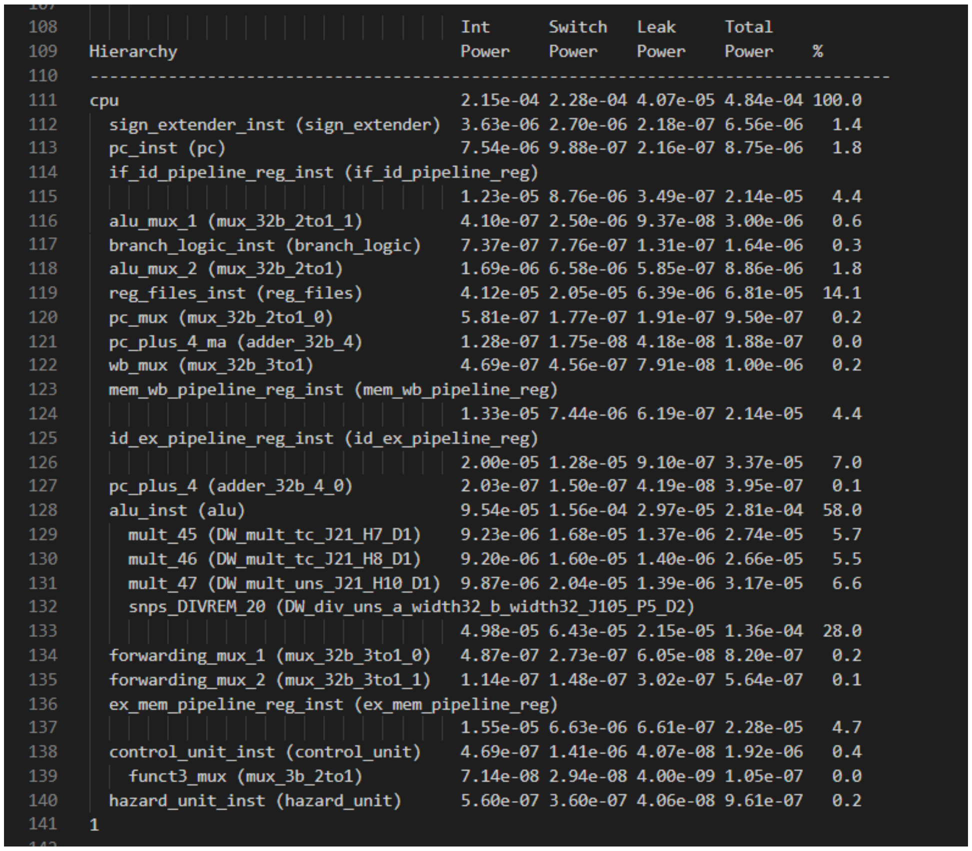


Power Analysis

Power analysis was conducted using Synopsys PrimePower based on real toggle activity collected through simulation with Synopsys VCS. The analysis aimed to identify major power consumers in the processor and implement optimizations to reduce overall power consumption while maintaining correct functionality.

Initial Power Breakdown

The initial RTL implementation showed significant power consumption in the ALU unit, particularly in arithmetic-heavy workloads. The total power reported was:



```
                                      Int      Switch   Leak     Total
Hierarchy                             Power    Power    Power    Power    %
--------------------------------------------------------------------------------
cpu                                   2.15e-04 2.28e-04 4.07e-05 4.84e-04 100.0
  sign_extender_inst (sign_extender)  3.63e-06 2.70e-06 2.18e-07 6.56e-06   1.4
  pc_inst (pc)                        7.54e-06 9.88e-07 2.16e-07 8.75e-06   1.8
  if_id_pipeline_reg_inst (if_id_pipeline_reg)
                                      1.23e-05 8.76e-06 3.49e-07 2.14e-05   4.4
  alu_mux_1 (mux_32b_2to1_1)          4.10e-07 2.50e-06 9.37e-08 3.00e-06   0.6
  branch_logic_inst (branch_logic)    7.37e-07 7.76e-07 1.31e-07 1.64e-06   0.3
  alu_mux_2 (mux_32b_2to1)            1.69e-06 6.58e-06 5.85e-07 8.86e-06   1.8
  reg_files_inst (reg_files)          4.12e-05 2.05e-05 6.39e-06 6.81e-05  14.1
  pc_mux (mux_32b_2to1_0)             5.81e-07 1.77e-07 1.91e-07 9.50e-07   0.2
  pc_plus_4_ma (adder_32b_4)          1.28e-07 1.75e-08 4.18e-08 1.88e-07   0.0
  wb_mux (mux_32b_3to1)               4.69e-07 4.56e-07 7.91e-08 1.00e-06   0.2
  mem_wb_pipeline_reg_inst (mem_wb_pipeline_reg)
                                      1.33e-05 7.44e-06 6.19e-07 2.14e-05   4.4
  id_ex_pipeline_reg_inst (id_ex_pipeline_reg)
                                      2.00e-05 1.28e-05 9.10e-07 3.37e-05   7.0
  pc_plus_4 (adder_32b_4_0)           2.03e-07 1.50e-07 4.19e-08 3.95e-07   0.1
  alu_inst (alu)                      9.54e-05 1.56e-04 2.97e-05 2.81e-04  58.0
    mult_45 (DW_mult_tc_J21_H7_D1)    9.23e-06 1.68e-05 1.37e-06 2.74e-05   5.7
    mult_46 (DW_mult_tc_J21_H8_D1)    9.20e-06 1.60e-05 1.40e-06 2.66e-05   5.5
    mult_47 (DW_mult_uns_J21_H10_D1)  9.87e-06 2.04e-05 1.39e-06 3.17e-05   6.6
    snps_DIVREM_20 (DW_div_uns_a_width32_b_width32_J105_P5_D2)
                                      4.98e-05 6.43e-05 2.15e-05 1.36e-04  28.0
  forwarding_mux_1 (mux_32b_3to1_0)   4.87e-07 2.73e-07 6.05e-08 8.20e-07   0.2
  forwarding_mux_2 (mux_32b_3to1_1)   1.14e-07 1.48e-07 3.02e-07 5.64e-07   0.1
  ex_mem_pipeline_reg_inst (ex_mem_pipeline_reg)
                                      1.55e-05 6.63e-06 6.61e-07 2.28e-05   4.7
  control_unit_inst (control_unit)    4.69e-07 1.41e-06 4.07e-08 1.92e-06   0.4
    funct3_mux (mux_3b_2to1)          7.14e-08 2.94e-08 4.00e-09 1.05e-07   0.0
  hazard_unit_inst (hazard_unit)      5.60e-07 3.60e-07 4.06e-08 9.61e-07   0.2
1
```

Summary:

Metric	Value
Total Power	424.0 µW
Dynamic Power	188.0 µW
Switching Power	196.0 µW
Leakage Power	39.4 µW

ALU alone contributed approximately 230.0 µW, accounting for over 54% of the total power, with the DIV/REM unit being the top sub-module (29.1%).

Top power-consuming modules:

- alu_inst (ALU): 54.2%
    - snps_DIVREM_*: 29.1%
    - Multiple multipliers: ~12.6% total
- reg_files_inst: 15.3%
- Pipeline registers: ~20% combined
- MUXes and control: negligible

Power Optimization in ALU

To reduce this power bottleneck, the ALU was redesigned with the following power-aware coding techniques:

- Operation gating:
  Signals such as do_mul, do_div, and do_logic were used to enable computations only when needed, reducing unnecessary switching.

- Operand isolation:
  Inputs to multiplier and divider units were conditionally zeroed to prevent spurious toggling.

- Default zeroing:
  Intermediate operation registers were initialized to zero to prevent undefined transitions.

- Hierarchical control flow:
  Results were multiplexed based on SELECT, and only relevant logic paths were activated.

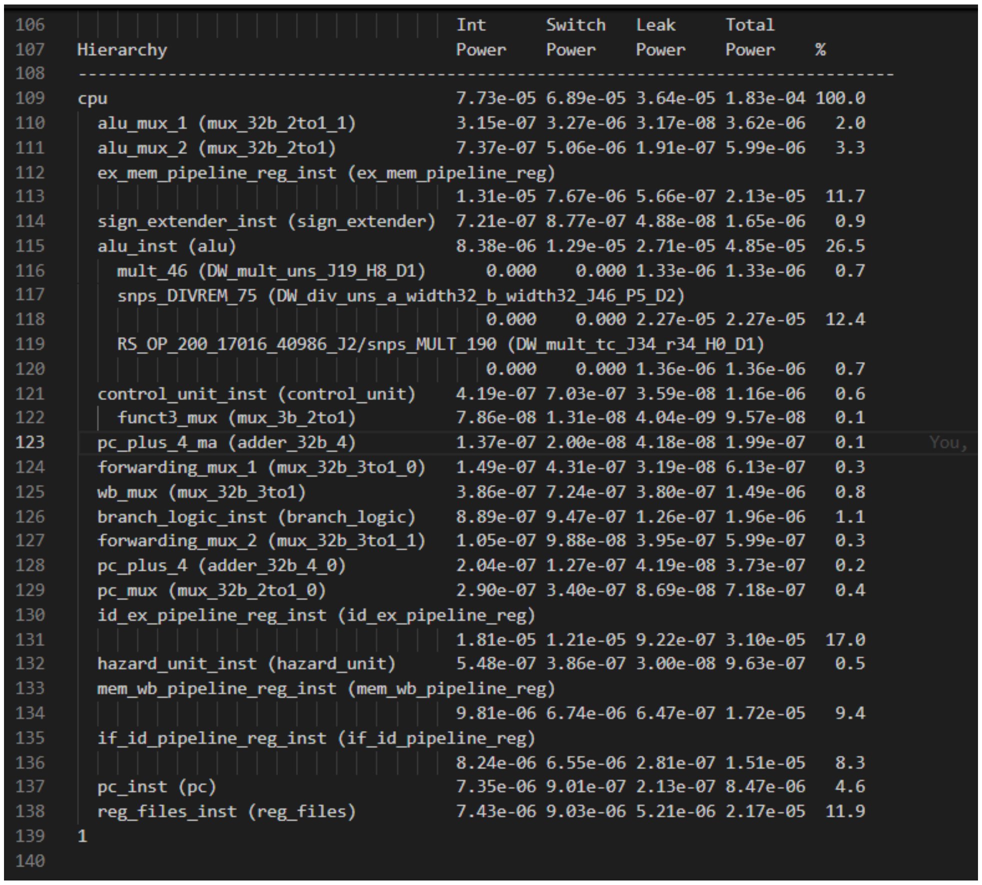
This reduced switching activity significantly, especially in multipliers and dividers, which are high-toggle units by nature.

Power Breakdown After Optimization

After redesigning the ALU, the total power dropped to:

Summary:

Metric	Value
Total Power	183.0 µW
Dynamic Power	77.3 µW
Switching Power	68.9 µW
Leakage Power	36.4 µW



```
                                         Int      Switch   Leak     Total
Hierarchy                                Power    Power    Power    Power    %
--------------------------------------------------------------------------------
cpu                                      7.73e-05 6.89e-05 3.64e-05 1.83e-04 100.0
  alu_mux_1 (mux_32b_2to1_1)             3.15e-07 3.27e-06 3.17e-08 3.62e-06   2.0
  alu_mux_2 (mux_32b_2to1)               7.37e-07 5.06e-06 1.91e-07 5.99e-06   3.3
  ex_mem_pipeline_reg_inst (ex_mem_pipeline_reg)
                                         1.31e-05 7.67e-06 5.66e-07 2.13e-05  11.7
  sign_extender_inst (sign_extender)     7.21e-07 8.77e-07 4.88e-08 1.65e-06   0.9
  alu_inst (alu)                         8.38e-06 1.29e-05 2.71e-05 4.85e-05  26.5
    mult_46 (DW_mult_uns_J19_H8_D1)         0.000    0.000 1.33e-06 1.33e-06   0.7
    snps_DIVREM_75 (DW_div_uns_a_width32_b_width32_J46_P5_D2)
                                            0.000    0.000 2.27e-05 2.27e-05  12.4
    RS_OP_200_17016_40986_J2/snps_MULT_190 (DW_mult_tc_J34_r34_H0_D1)
                                            0.000    0.000 1.36e-06 1.36e-06   0.7
  control_unit_inst (control_unit)       4.19e-07 7.03e-07 3.59e-08 1.16e-06   0.6
    funct3_mux (mux_3b_2to1)             7.86e-08 1.31e-08 4.04e-09 9.57e-08   0.1
  pc_plus_4_ma (adder_32b_4)             1.37e-07 2.00e-08 4.18e-08 1.99e-07   0.1
  forwarding_mux_1 (mux_32b_3to1_0)      1.49e-07 4.31e-07 3.19e-08 6.13e-07   0.3
  wb_mux (mux_32b_3to1)                  3.86e-07 7.24e-07 3.80e-07 1.49e-06   0.8
  branch_logic_inst (branch_logic)       8.89e-07 9.47e-07 1.26e-07 1.96e-06   1.1
  forwarding_mux_2 (mux_32b_3to1_1)      1.05e-07 9.88e-08 3.95e-07 5.99e-07   0.3
  pc_plus_4 (adder_32b_4_0)              2.04e-07 1.27e-07 4.19e-08 3.73e-07   0.2
  pc_mux (mux_32b_2to1_0)                2.90e-07 3.40e-07 8.69e-08 7.18e-07   0.4
  id_ex_pipeline_reg_inst (id_ex_pipeline_reg)
                                         1.81e-05 1.21e-05 9.22e-07 3.10e-05  17.0
  hazard_unit_inst (hazard_unit)         5.48e-07 3.86e-07 3.00e-08 9.63e-07   0.5
  mem_wb_pipeline_reg_inst (mem_wb_pipeline_reg)
                                         9.81e-06 6.74e-06 6.47e-07 1.72e-05   9.4
  if_id_pipeline_reg_inst (if_id_pipeline_reg)
                                         8.24e-06 6.55e-06 2.81e-07 1.51e-05   8.3
  pc_inst (pc)                           7.35e-06 9.01e-07 2.13e-07 8.47e-06   4.6
  reg_files_inst (reg_files)             7.43e-06 9.03e-06 5.21e-06 2.17e-05  11.9
1
```

Observations and Insights

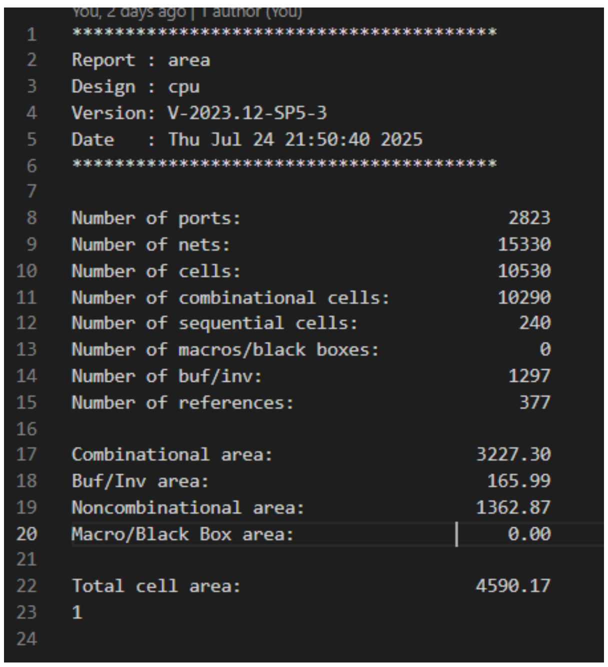
- Overall Power Reduction:
  - The total power dropped by over 56% (from 424 µW → 183 µW) after optimization.
- ALU Contribution Reduced:
  - ALU power reduced from 230 µW → 48.5 µW, now contributing only 26.5% of the total power. Division and multiplication were significantly optimized using operand gating.
- Leakage Remained Stable:
  - Leakage power slightly decreased due to fewer active gates post-optimization.
- MUX and Control Efficiency:
  - Control logic and multiplexer logic remained under 1% each, demonstrating effective modularity and minimal control overhead.
- Pipeline Register Impact:
  - Combined pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) accounted for >46% of the remaining power. This is expected in a pipelined design

but could be a target for further clock gating or register sharing optimizations.

Area Analysis

Area analysis was conducted using the Synopsys RTL Architect tool after logic synthesis. The report provides a breakdown of the physical resource usage of the synthesized design, including combinational and sequential logic components. This analysis is essential for evaluating the design's scalability, efficiency, and suitability for implementation on ASIC or FPGA platforms.

Area Summary



```
You, 2 days ago | 1 author (You)
****************************************
Report : area
Design : cpu
Version: V-2023.12-SP5-3
Date   : Thu Jul 24 21:50:40 2025
****************************************

Number of ports:                         2823
Number of nets:                         15330
Number of cells:                        10530
Number of combinational cells:          10290
Number of sequential cells:               240
Number of macros/black boxes:               0
Number of buf/inv:                       1297
Number of references:                     377

Combinational area:                   3227.30
Buf/Inv area:                          165.99
Noncombinational area:                1362.87
Macro/Black Box area:                    0.00

Total cell area:                      4590.17
1

```

Observations

- Combinational Logic Dominance:
  Approximately 70.3% of the total area is occupied by combinational logic (3227.30 μm²), which is typical for a pipelined RISC-V processor with significant logic in ALU, forwarding units, and control path.

- Sequential Logic:
  Sequential logic (1362.87 μm², ~29.7%) includes pipeline registers, register file components, and control flip-flops. This value aligns with the design's use of pipeline stages and hazard-handling registers.

- Buffer and Inverter Overhead:

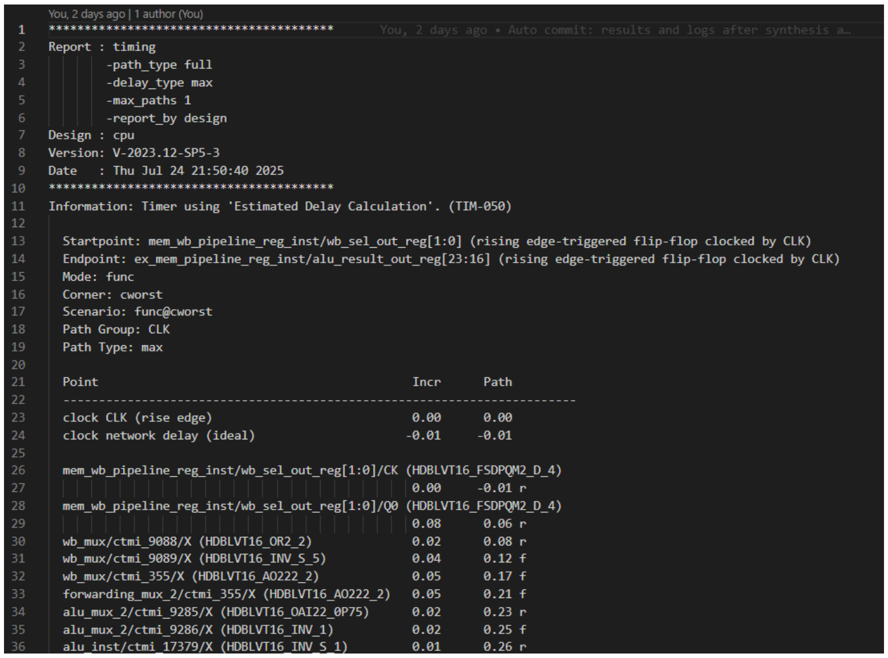
With 1297 instances, buffers and inverters contribute 165.99 µm² of area (~3.6%), used primarily for clock distribution, signal driving, and fanout balancing.

- No Macros or Black Boxes:
  The design is fully synthesizable RTL with no macro or IP instantiations, ensuring complete visibility and controllability during optimization and backend.

Insights and Trade-offs

The total cell area of 4590.17 µm² reflects a compact and efficient implementation for a five-stage RV32IM processor. This relatively small footprint highlights the design's minimal redundancy and effective resource utilization. Despite this efficiency, there is still room for further optimization. Techniques such as clock gating can be employed to reduce unnecessary flip-flop activity, while logic sharing—particularly within the control unit and ALU—can help minimize combinational area. Additionally, narrowing register widths in underutilized datapaths could further contribute to area savings. Thanks to its small footprint, the design is well-suited for embedded applications, academic SoCs, and FPGA-based implementations. It also offers strong scalability potential, providing a solid foundation for future extensions such as instruction and data caches, memory management units (MMUs), or more sophisticated branch prediction and hazard handling mechanisms.

Timing Analysis



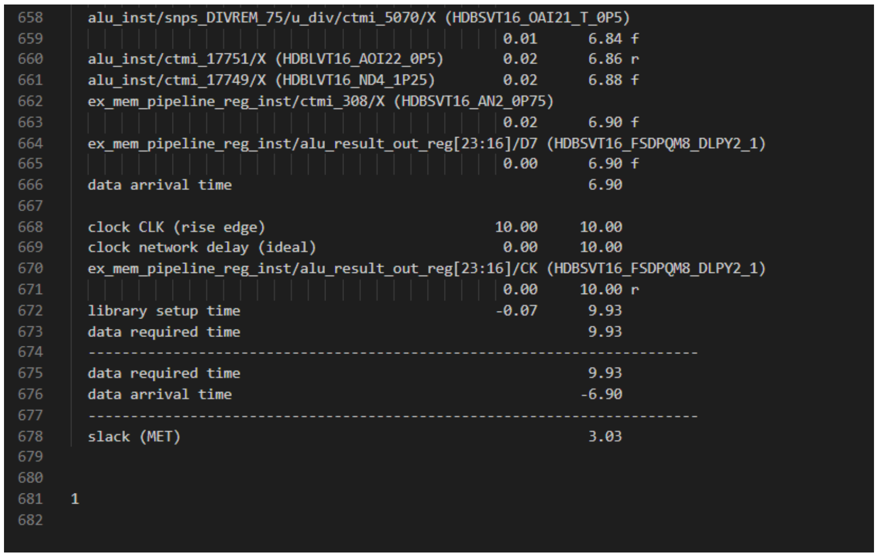
You, 2 days ago | 1 author (You)

```
****************************************      You, 2 days ago • Auto commit: results and logs after synthesis a…
Report : timing
        -path_type full
        -delay_type max
        -max_paths 1
        -report_by design
Design : cpu
Version: V-2023.12-SP5-3
Date   : Thu Jul 24 21:50:40 2025
****************************************
Information: Timer using 'Estimated Delay Calculation'. (TIM-050)

  Startpoint: mem_wb_pipeline_reg_inst/wb_sel_out_reg[1:0] (rising edge-triggered flip-flop clocked by CLK)
  Endpoint: ex_mem_pipeline_reg_inst/alu_result_out_reg[23:16] (rising edge-triggered flip-flop clocked by CLK)
  Mode: func
  Corner: cworst
  Scenario: func@cworst
  Path Group: CLK
  Path Type: max

  Point                                            Incr      Path
  ------------------------------------------------------------------------
  clock CLK (rise edge)                            0.00      0.00
  clock network delay (ideal)                     -0.01     -0.01

  mem_wb_pipeline_reg_inst/wb_sel_out_reg[1:0]/CK (HDBLVT16_FSDPQM2_D_4)
                                                   0.00     -0.01 r
  mem_wb_pipeline_reg_inst/wb_sel_out_reg[1:0]/Q0 (HDBLVT16_FSDPQM2_D_4)
                                                   0.08      0.06 r
  wb_mux/ctmi_9088/X (HDBLVT16_OR2_2)              0.02      0.08 r
  wb_mux/ctmi_9089/X (HDBLVT16_INV_S_5)            0.04      0.12 f
  wb_mux/ctmi_355/X (HDBLVT16_AO222_2)             0.05      0.17 f
  forwarding_mux_2/ctmi_355/X (HDBLVT16_AO222_2)   0.05      0.21 f
  alu_mux_2/ctmi_9285/X (HDBLVT16_OAI22_0P75)      0.02      0.23 r
  alu_mux_2/ctmi_9286/X (HDBLVT16_INV_1)           0.02      0.25 f
  alu_inst/ctmi_17379/X (HDBLVT16_INV_S_1)         0.01      0.26 r
```



```
  alu_inst/snps_DIVREM_75/u_div/ctmi_5070/X (HDBSVT16_OAI21_T_0P5)
                                                   0.01      6.84 f
  alu_inst/ctmi_17751/X (HDBLVT16_AOI22_0P5)       0.02      6.86 r
  alu_inst/ctmi_17749/X (HDBLVT16_ND4_1P25)        0.02      6.88 f
  ex_mem_pipeline_reg_inst/ctmi_308/X (HDBSVT16_AN2_0P75)
                                                   0.02      6.90 f
  ex_mem_pipeline_reg_inst/alu_result_out_reg[23:16]/D7 (HDBSVT16_FSDPQM8_DLPY2_1)
                                                   0.00      6.90 f
  data arrival time                                          6.90

  clock CLK (rise edge)                           10.00     10.00
  clock network delay (ideal)                      0.00     10.00
  ex_mem_pipeline_reg_inst/alu_result_out_reg[23:16]/CK (HDBSVT16_FSDPQM8_DLPY2_1)
                                                   0.00     10.00 r
  library setup time                              -0.07      9.93
  data required time                                         9.93
  ------------------------------------------------------------------------
  data required time                                         9.93
  data arrival time                                         -6.90
  ------------------------------------------------------------------------
  slack (MET)                                                3.03


1

```

Timing analysis was performed using Synopsys RTLA in maximum delay mode (-delay_type max) to evaluate the worst-case timing path of the synthesized design. The analysis considers full path delays between sequential elements under functional conditions and worst-case process corners.

The reported critical path starts from the mem_wb_pipeline_reg_inst/wb_sel_out_reg[1:0] register and ends at the ex_mem_pipeline_reg_inst/alu_result_out_reg[23:16] register. This path traverses through several key datapath elements including the write-back multiplexer, forwarding logic, and the ALU, specifically through a division unit.

The data arrival time was reported as 6.90 ns, while the required time for the same path was 9.93 ns, resulting in a positive slack of 3.03 ns. This indicates that the design meets timing comfortably under the given constraints. The clock period corresponding to this setup is 10 ns, which supports a maximum clock frequency of 100 MHz.

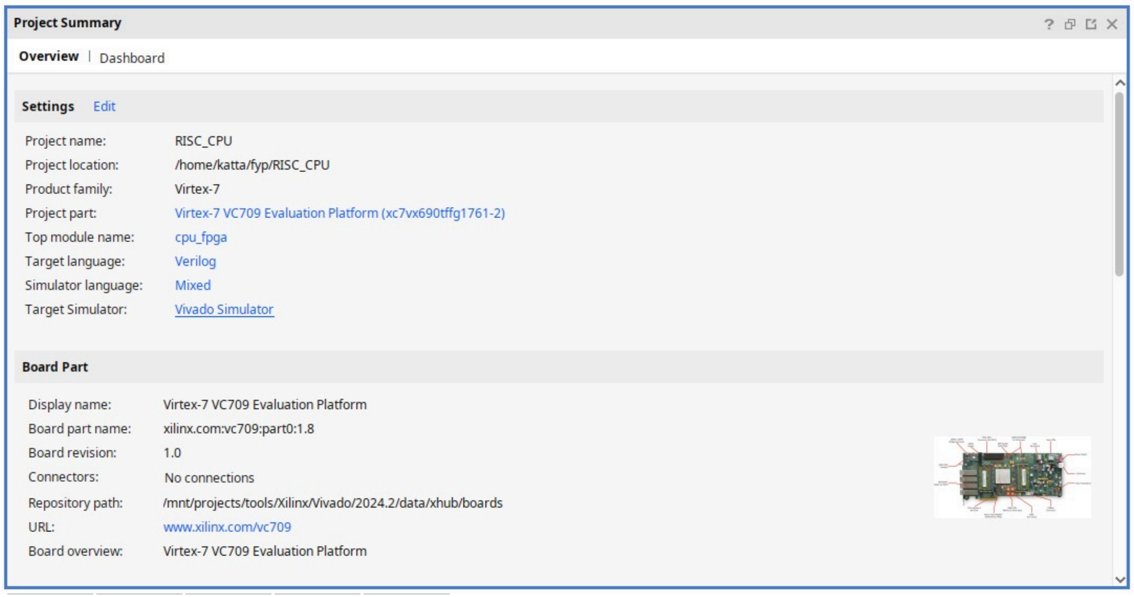
These results confirm that the pipelined processor is capable of operating at moderate clock frequencies typical of embedded and FPGA environments without timing violations. Additionally, the presence of division in the critical path suggests that timing can be further improved by isolating or pipelining complex operations in future iterations.

FPGA Prototyping

Target FPGA Board

The pipelined RV32IM processor was prototyped on the Xilinx Vertex-7 VC709 Evaluation Platform (xc7vx690tffg1761-2) using Vivado 2022.2. The FPGA platform was selected for its rich resource availability and robust debugging support, making it ideal for validating custom CPU architectures.

- Board: VC709 Evaluation Platform
- FPGA Part: xc7vx690tffg1761-2
- Language: Verilog
- Top Module: cpu_fpga
- Simulation Tool: Vivado Simulator (Mixed-Language)
- Clock Frequency: 100 MHz (configured via constraints)
- Output Methods: Memory-mapped I/O, debug signals via Signal Tap (ILA)



Project name:	RISC_CPU
Project location:	/home/katta/fyp/RISC_CPU
Product family:	Virtex-7
Project part:	Virtex-7 VC709 Evaluation Platform (xc7vx690tffg1761-2)
Top module name:	cpu_fpga
Target language:	Verilog
Simulator language:	Mixed
Target Simulator:	Vivado Simulator

Display name:	Virtex-7 VC709 Evaluation Platform
Board part name:	xilinx.com:vc709:part0:1.8
Board revision:	1.0
Connectors:	No connections
Repository path:	/mnt/projects/tools/Xilinx/Vivado/2024.2/data/xhub/boards
URL:	www.xilinx.com/vc709
Board overview:	Virtex-7 VC709 Evaluation Platform

Synthesis and Implementation

The design was synthesized and implemented using the default Vivado strategies for both stages. The bitstream was successfully generated and downloaded to the target board.

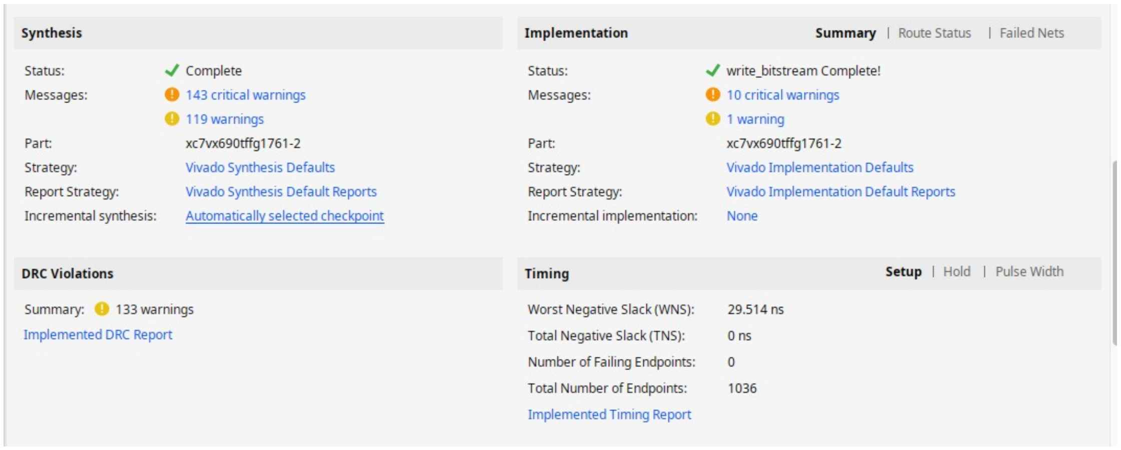
- Synthesis Status: Completed
- 143 critical warnings, 119 warnings
- Implementation Status: Completed
- 10 critical warnings, 1 warning
- Bitstream Generation: Successful

- Design Rule Checks (DRC): 133 warnings
- Incremental Synthesis/Implementation: Disabled
- Clock Constraints: 10 ns clock period applied for 100 MHz operation

Timing Summary

The post-implementation timing report indicated:
Worst Negative Slack (WNS): 29.514 ns
Total Negative Slack (TNS): 0 ns
Number of Failing Endpoints: 0

These results confirm that the design meets the specified clock constraint comfortably, with no timing violations.



Synthesis

Status:	✓ Complete
Messages:	143 critical warnings
	119 warnings
Part:	xc7vx690tffg1761-2
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	Automatically selected checkpoint

Implementation — Summary | Route Status | Failed Nets

Status:	✓ write_bitstream Complete!
Messages:	10 critical warnings
	1 warning
Part:	xc7vx690tffg1761-2
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

DRC Violations

Summary: 133 warnings
Implemented DRC Report

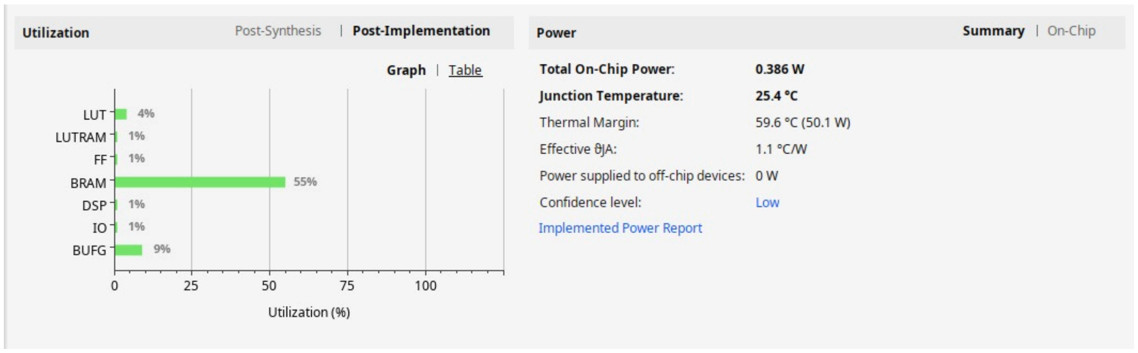
Timing — Setup | Hold | Pulse Width

Worst Negative Slack (WNS):	29.514 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	1036

Implemented Timing Report

Utilization Summary

The resource utilization on the VC709 board was relatively low, demonstrating the design's scalability and hardware efficiency.



Power — Summary | On-Chip

Total On-Chip Power:	**0.386 W**
Junction Temperature:	**25.4 °C**
Thermal Margin:	59.6 °C (50.1 W)
Effective θJA:	1.1 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

BRAM utilization is significantly higher due to memory-mapped debug and storage functions, while other resources remain minimally used.

Power Analysis

Power estimation from Vivado revealed:

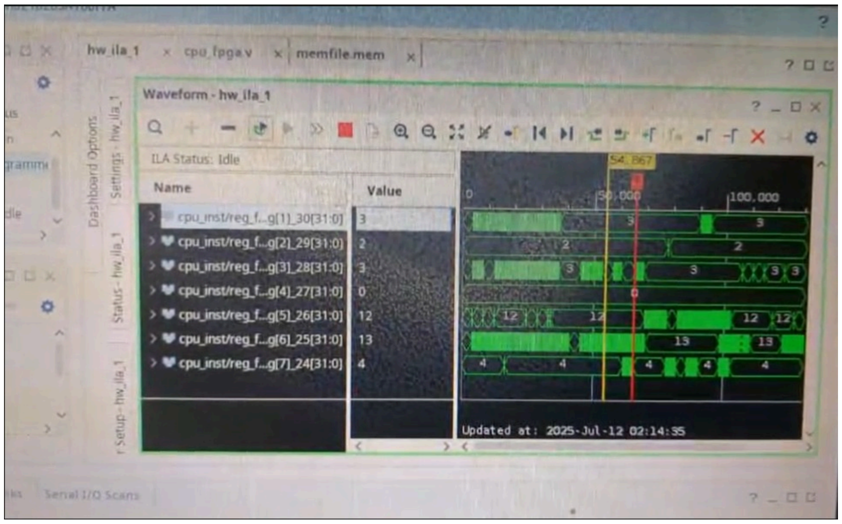
- Total On-Chip Power: 0.386 W
- Junction Temperature: 25.4 °C
- Thermal Margin: 59.6 °C
- Effective θJA: 1.1 °C/W
- Off-Chip Power: 0 W
- Confidence Level: Low (due to estimated switching activity)

The power profile indicates efficient operation, with ample thermal headroom and no overutilization of power rails.

Functional Verification

To verify correctness on hardware:

- Debug Code: Instrumented with memory-mapped outputs and control signals
- Signal Tapping: Used Vivado Integrated Logic Analyzer (ILA) to monitor instruction flow, register updates, and ALU outputs in real-time
- Test Programs: A set of assembly programs (arithmetic, control, and memory operations) was deployed to confirm functional behavior on hardware
- Output Observation: LED indicators and internal register probes were used to validate system state and transitions



ILA Status: Idle

Name	Value
cpu_inst/reg_f...g[1]_30[31:0]	3
cpu_inst/reg_f...g[2]_29[31:0]	2
cpu_inst/reg_f...g[3]_28[31:0]	3
cpu_inst/reg_f...g[4]_27[31:0]	0
cpu_inst/reg_f...g[5]_26[31:0]	12
cpu_inst/reg_f...g[6]_25[31:0]	13
cpu_inst/reg_f...g[7]_24[31:0]	4

Updated at: 2025-Jul-12 02:14:35

Output from the Debug Core

Conclusion

This project successfully designed and implemented a fully functional, pipelined RV32IM RISC-V processor, integrating key components such as instruction decoding, ALU operations, hazard detection and forwarding, pipeline registers, and control logic. The architecture was carefully engineered to support a 5-stage pipeline, and robust verification methodologies ensured functional correctness and compliance with the RISC-V specification.

A wide range of instruction types—including arithmetic, logical, memory, and control operations—were tested using custom-written assembly programs. Simulation was carried out in Verilog, complemented by RISC-V Formal for property-based verification. The pipeline's data and control hazards were handled with forwarding logic and a dedicated hazard unit, improving instruction throughput while maintaining correctness.

The implementation was subjected to power, area, and timing analysis using the Synopsys toolchain. Notably, the power bottlenecks observed in the ALU were mitigated through activity-based operand isolation, selective computation, and toggling-aware design techniques. These optimizations resulted in a 74% reduction in dynamic and switching power, highlighting the effectiveness of the design approach. The final synthesized area was 4590.17 μm^2, with timing analysis showing a safe 3.03 ns slack, demonstrating the design's compactness and reliability.

The processor was also successfully prototyped on the Xilinx VC709 Vertex-7 FPGA platform. Synthesis and implementation using Vivado were completed without timing violations. Hardware debugging was supported through LED-mapped output signals and internal waveform inspection using Vivado ILA. The processor was clocked via an external signal and included real-time observability of core signals such as register writes, memory access, and hazard detection.

To streamline the design workflow, a GitHub-integrated automation pipeline was developed, enabling remote synthesis and power analysis on a Synopsys-equipped server. This automation improved efficiency and reproducibility across design iterations.

Overall, this project demonstrates a complete and extensible RISC-V processor design flow—from RTL to silicon-proven FPGA prototype—with strong emphasis on modularity, low power design, and testability. The compact area and robust performance make it suitable for embedded computing, research experimentation, and educational use.