VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS ENGINEERING

------o0o-----

CAPSTONE PROJECT 1

TOPIC

DESIGN AND VERIFICATION OF A 5-STAGE PIPELINED RISC-V PROCESSOR

Instructor: Dr. Trần Hoàng Linh

Student: Nguyễn Duy Ngọc

Student ID: 2251036

Ho Chi Minh City, May 2025

PROJECT SUMMARY

This project presents the complete design, implementation, simulation, and synthesis of a **5-stage pipelined RISC-V RV32I processor**, developed using a modular RTL design approach. The processor adheres strictly to the base RV32I instruction set architecture, supporting all 39 instruction types, and is built with industry-standard design principles from Harris & Harris [1], Patterson & Hennessy [2], and the official RISC-V specifications [3].

The architecture consists of five pipeline stages: Fetch (F), Decode (D), Execute (E), Memory (M), and Writeback (W), with all major pipeline registers and hazard management mechanisms implemented. Submodules such as the ALU, Main Decoder, ALU Decoder, Branch Unit (BRU), Load/Store Unit (LSU), and Environment Unit (EU) were designed and integrated carefully with fully verified control logic. A dedicated **Hazard Unit** handles read-after-write hazards via forwarding, and load-use hazards via pipeline stalls and bubbles. Control hazards are resolved through branch prediction with flush mechanisms.

All control signals were derived from well-structured truth tables, ensuring predictable operation. The **testbench suite**, created using Verilator, includes full verification of each submodule and system-level simulation. The test program includes **over 150 instructions** to exercise all instruction types, with waveform analysis confirming correct timing, hazard resolution, and data integrity. The test starts at cycle 0 and finishes at cycle 160 in the short demonstration, and continues to execute an extensive instruction set afterward.

Post-design, the processor was synthesized using **Quartus Prime**. The synthesized design, including the datapath and control modules, confirmed successful mapping and fitting. Netlist views of all core modules demonstrate correct logic optimization and module interaction.

The project directory structure is organized clearly by module, bench, testcases, simulation logs, and synthesis outputs. This ensures modular testing, easy debugging, and reusability for future upgrades such as ISA extension (e.g., RV32IM or RV64I), support for out-of-order execution, or integration of a basic memory management unit (MMU).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Overview

In the era of open-source hardware and processor democratization, the RISC-V instruction set architecture (ISA) has emerged as a highly influential and flexible standard. Unlike proprietary ISAs, RISC-V enables developers to implement processors tailored for academic research, industrial deployment, or educational purposes without licensing restrictions. This project focuses on the design and verification of a 5-stage pipelined RISC-V processor that supports the base RV32I instruction set. The design is implemented using SystemVerilog, verified using Verilator and GTKWave, and synthesized on Intel Quartus Prime for FPGA deployment.

The 5-stage pipelined architecture is widely used in modern CPU design due to its ability to improve instruction throughput without significantly increasing hardware complexity. This architecture divides instruction execution into five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). However, pipelining introduces challenges such as data hazards, control hazards, and structural hazards, which must be effectively addressed through techniques like forwarding, hazard detection, and stalling.

The objective of this project is to design, simulate, and verify a processor core capable of executing all RV32I instructions accurately and efficiently. By exploring RTL design principles, control logic implementation, and verification workflows, this project not only builds a functional processor but also enhances understanding of low-level hardware architecture.

1.2. Project Objectives

The main goals of the project include designing a functional and synthesizable 5-stage pipelined processor supporting the RV32I ISA and verifying its correctness using testbenches and waveform analysis. Key deliverables include RTL design files, block diagrams, waveform results, control tables, and synthesis reports.

Design Tasks:

Task 1: Theoretical study of pipelining and the RV32I instruction set

Study processor organization based on pipelining principles. Research the full instruction set of RV32I, its categorization (R, I, S, B, U, J types), and operational semantics using the references: “Digital Design and Computer Architecture” by Harris & Harris and “Computer Organization and Design” by Patterson & Hennessy.

Task 2: Control unit and datapath design

Develop separate control modules: main decoder, ALU decoder, branch unit, load – store unit, and environment unit. Analyze instruction formats and determine required control signals. Design the datapath to support instruction execution through all five pipeline stages.

Task 3: Implementation and testbench development

Implement the processor in SystemVerilog. Write testbenches to verify each module individually and then integrate them into a full processor-level test. Feed all RV32I instruction types with descriptive checking messages.

Task 4: RTL verification and simulation

Use Verilator for simulation and GTKWave for waveform tracing. Output internal signals and debug data paths to ensure correct operation.

Task 5: Synthesis and FPGA proof-of-concept

Synthesize the processor using Quartus Prime. Verify the synthesis result and generate netlist views of each component to ensure the design is practically realizable.

The design will be developed incrementally with continuous integration and testing. The overall approach emphasizes modularity, clarity in control signal decoding, and strong verification coverage across all instruction types.

2. DESIGN THEORY AND INTEREST

This section presents the core theoretical foundation and architectural principles underlying the implementation of the pipelined RV32I RISC-V processor. The content is structured around its key components and stages, referencing established computer architecture texts to substantiate each design choice.

2.1. RV32I Instruction Set

The RV32I instruction set is the base integer instruction set of the RISC-V architecture, consisting of 47 core instructions that include arithmetic, logical, control flow, load/store, and system instructions [1]. These instructions follow a load-store architecture and operate on 32 general-purpose registers (x0-x31). The register x0 is hardwired to zero. RV32I supports six instruction formats: R-type, I-type, S-type, B-type, U-type, and J-type (Figure 1).

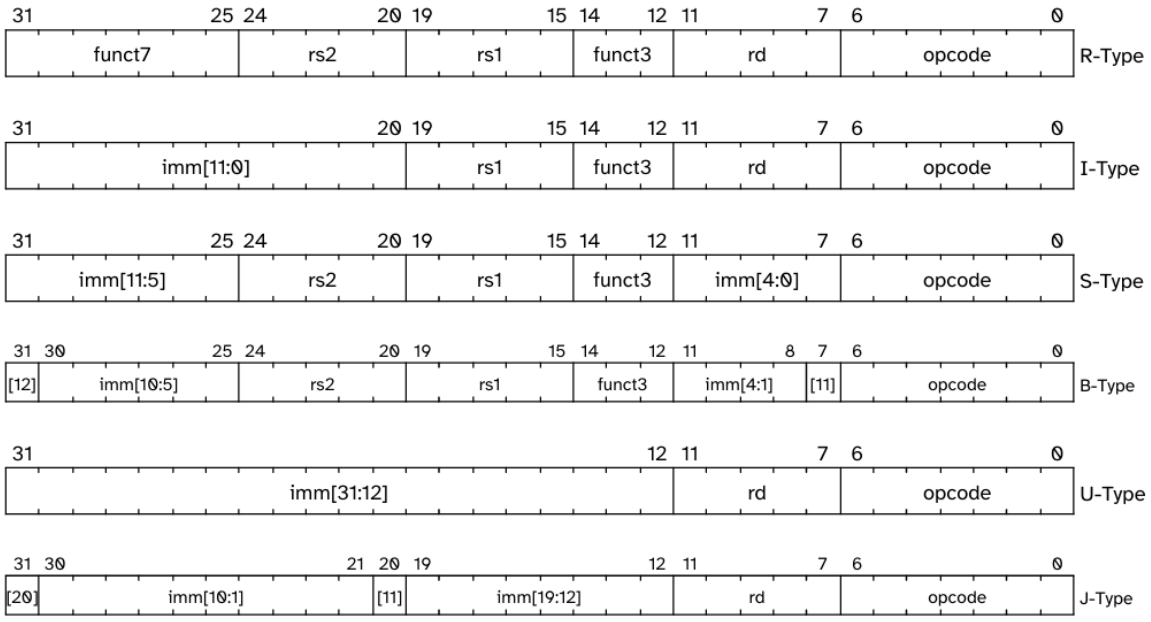

Figure 1. Base RV32I instruction format [3]

2.2. Microarchitectures

In processor design, three principal microarchitectures are often considered: single-cycle, multi-cycle, and pipelined. In a *single-cycle architecture*, each instruction is executed in one clock cycle, leading to inefficient utilization of resources since all steps of the instruction (fetch, decode, execute, memory, writeback) must complete within the cycle's duration. *Multi-cycle architectures* improve on this by reusing functional units across cycles, allowing each instruction to take multiple cycles but reducing wasted hardware.

Pipelined architecture, the focus of this project, achieves a balance between performance and hardware complexity by overlapping instruction execution. Like an assembly line, each instruction is broken into stages with dedicated hardware,

allowing multiple instructions to be in different stages simultaneously. This yields a significant performance boost by increasing instruction throughput without a proportional increase in clock frequency [1].

2.3. Pipeline Stages

To enhance throughput, the processor is divided into five pipeline stages: *Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Writeback (WB)*. Each stage performs a specific subset of operations required to complete an instruction, allowing multiple instructions to be processed simultaneously in a staggered fashion. This pipelining technique enables the processor to complete one instruction per clock cycle under ideal conditions, significantly improving performance over single-cycle designs [1].

In the *Instruction Fetch (IF)* stage, the processor reads an instruction from instruction memory using the current program counter (PC). During the *Instruction Decode (ID)* stage, it decodes the instruction and reads operands from the register file. The *Execute (EX)* stage performs arithmetic or logic operations using the ALU or calculates the effective address for memory instructions. If the instruction involves data memory, the *Memory Access (MEM)* stage performs the read or write. Lastly, in the *Writeback (WB)* stage, the result is written back to the register file, if required [1][2].

As shown in Figure 2(b), each pipeline stage typically completes within a fixed interval — in this project, the longest delay is imposed by memory access (200 ps), thus setting the clock period. Although the instruction latency increases to five cycles (1000 ps), the throughput increases to one instruction per cycle (200 ps per instruction), representing a 5× theoretical improvement compared to a single-cycle design (Figure 2(a)) [1].

To maintain correctness and efficiency, the pipelined architecture uses techniques such as write in first half, read in second half of a cycle for the register file, ensuring that newly written values can be read in the same cycle by subsequent instructions [1].

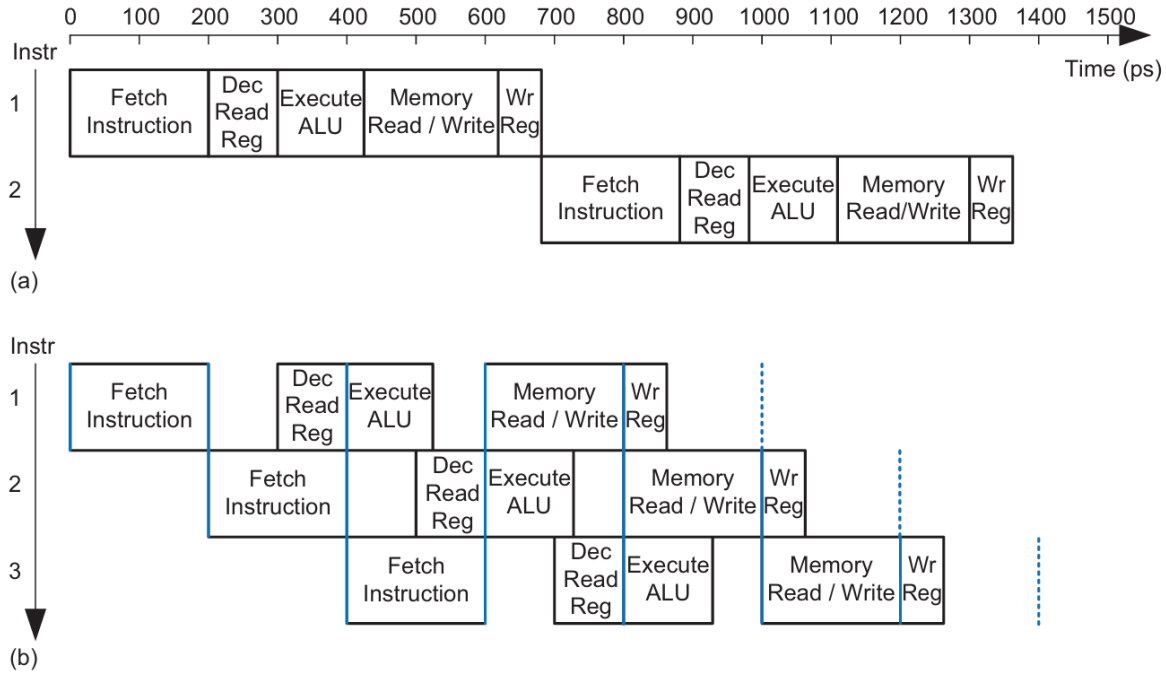

Figure 2. Timing diagram: (a) single-cycle processor and (b) pipelined processor [1]

Figure 3 further illustrates the pipeline in operation. Each instruction progresses through the pipeline with overlapping stages, visualized across cycles. For example, in cycle 4, the ALU is computing the result for an add, while the register file is simultaneously being read for a sub instruction, and another instruction is being fetched from memory [1].

This careful staging of logic helps maximize utilization and throughput while maintaining correctness. In practice, however, data dependencies between instructions may introduce pipeline hazards that require additional logic to resolve — such as forwarding or inserting stalls — which will be discussed in a later section.

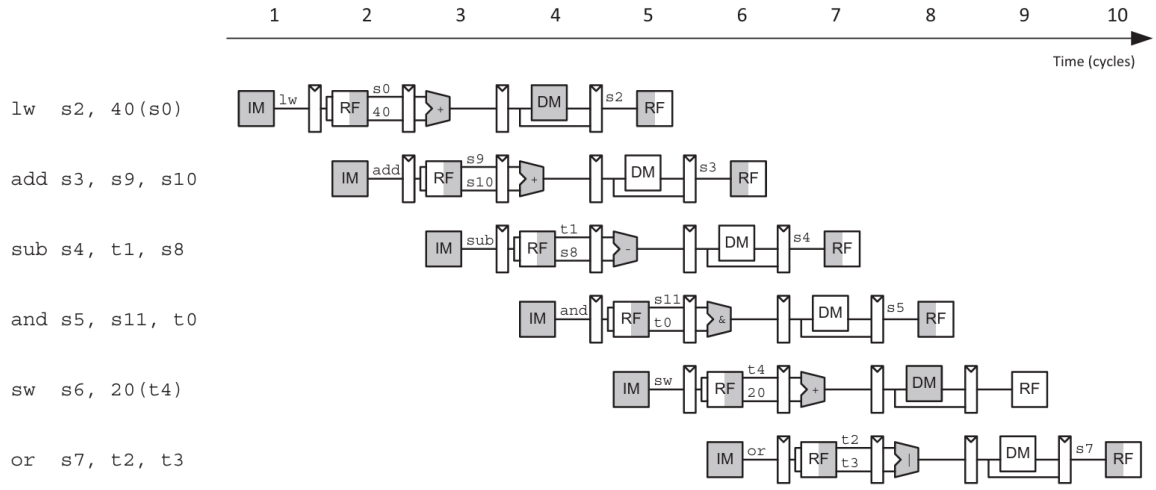

Figure 3. Abstract view of pipeline in operation [1]

2.4. Pipelined Data Path

The pipelined datapath is constructed by dividing the single-cycle processor into five sequential stages separated by pipeline registers. Each stage performs a specific function in the instruction execution cycle. As illustrated in Figure 4, these stages are: Fetch, Decode, Execute, Memory, and Writeback. To support this structure, pipeline registers are inserted between stages, carrying the necessary data and control signals forward in synchrony.

Each signal in the datapath is suffixed to denote the stage it belongs to (e.g., PCF, InstrD, ImmExtE, ResultW). Ensuring these signals advance through the pipeline in lockstep is crucial to preserve instruction semantics.

A particular challenge in pipelined architecture is the interaction with the register file. In this design, registers are read during the Decode stage but written back during the Writeback stage. This creates a feedback loop, requiring careful timing. To support correct operation, the register file writes on the falling edge of the clock and reads on the rising edge, enabling one instruction to write and the next to immediately read the updated value within the same cycle.

An important correction is made from the naive pipeline model shown in Figure 4(b). Initially, the write-back destination register (Rd) is directly taken from the Decode stage (RdD). However, this creates an error: the register destination does not move forward with the instruction, potentially overwriting incorrect registers. For instance, if a lw instruction meant to write to x2, the incorrect design could instead write to x5 due to pipeline misalignment.

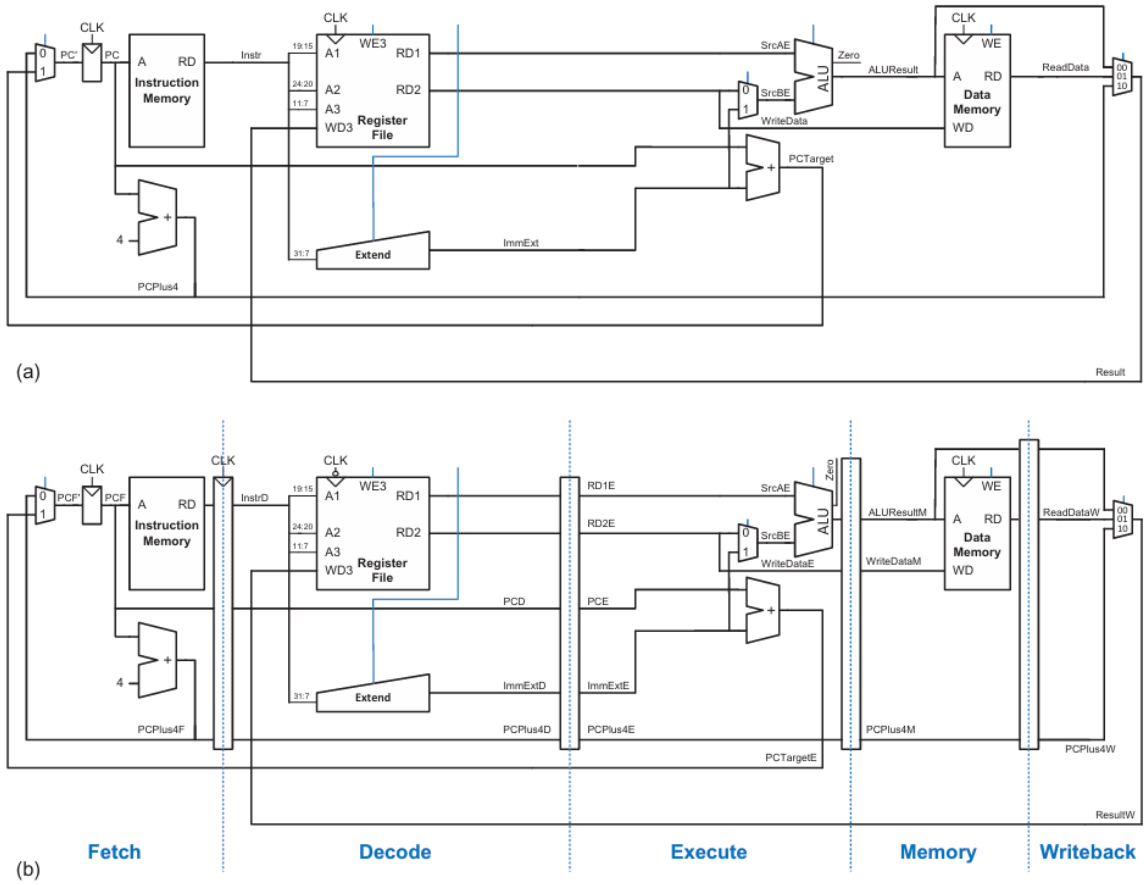

Figure 4. Datapaths: (a) single-cycle and (b) pipelined [1]

Figure 5 corrects this by forwarding the Rd signal through the pipeline registers, producing RdE, RdM, and finally RdW. The Writeback stage then writes the correct ResultW value to the correct register RdW, ensuring functional correctness.

This synchronization principle also applies to control signals like the next program counter (PCF'). Selecting between PCPlus4F and PCTargetE requires special handling to avoid control hazards. These are later addressed using pipeline flushes and control logic [1].

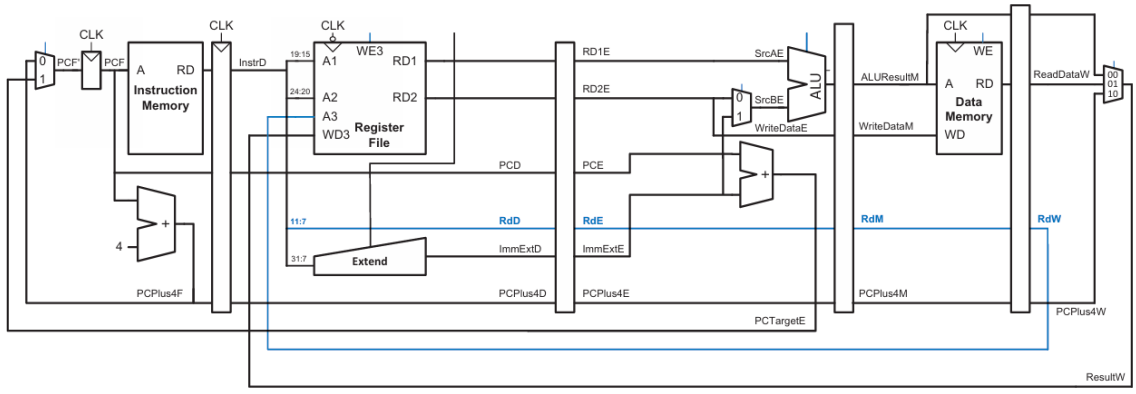

Figure 5. Correct datapath with forwarded data through pipeline registers [1]

2.5. Pipelined Control Unit

The pipelined processor uses the same control signals as the single-cycle processor and, therefore, has the same control unit. The control unit examines the op, funct3, and $funct7_5$ fields of the instruction in the Decode stage to produce the control signals (Figure 6). These control signals must be pipelined along with the data so that they remain synchronized with the instruction. RegWrite must be pipelined into the Writeback stage before it feeds back to the register file, just as Rd was pipelined in Figure 5. In addition to R-type ALU instructions, lw, sw, and beq, this pipelined processor also supports jal and I-type ALU instructions.

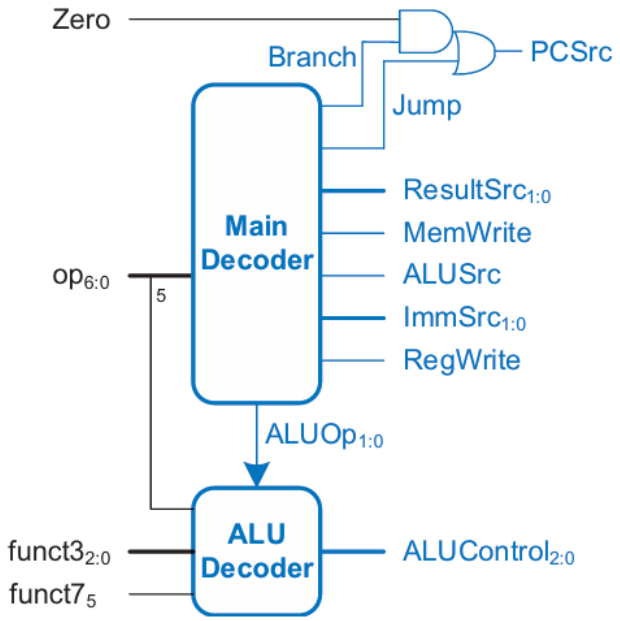

Figure 6. Control unit diagram [1]

2.6. Hazards in Pipelining

In pipelined processors, multiple instructions are processed simultaneously across different stages. While this parallelism enhances throughput, it introduces hazards — situations where one instruction depends on the outcome or control flow of another still in progress. If left unresolved, hazards can compromise functional correctness.

2.6.1. Data Hazards and RAW Dependencies

The most common hazard is a data hazard, especially Read-After-Write (RAW). It occurs when an instruction reads a register that a prior instruction has not yet written. For example, if instruction I1 writes to register x8 and instruction I2 reads x8 before I1 completes, I2 will use outdated data (Figure 7).

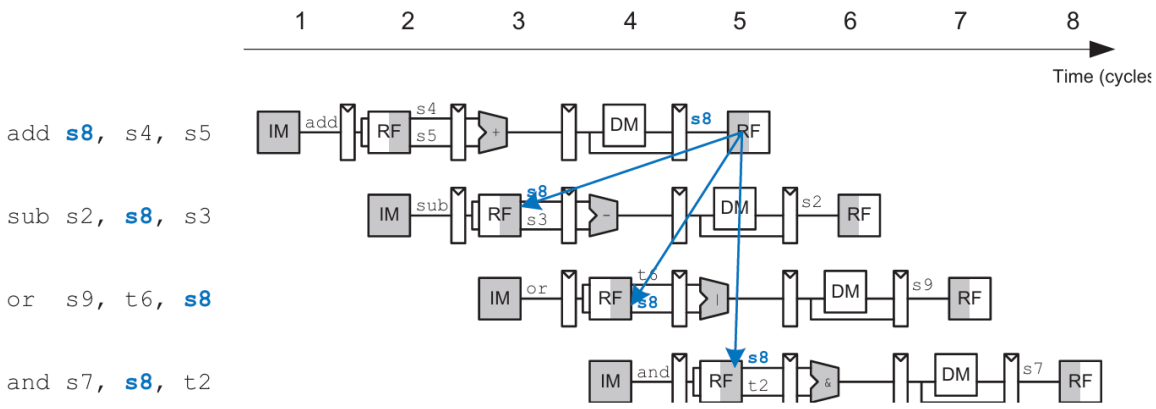

Figure 7. Abstract pipeline diagram illustrating hazards [1]

In a typical 5-stage pipeline (Fetch → Decode → Execute → Memory → Writeback), a RAW hazard can occur when an instruction tries to read a register in the Decode or Execute stage before it has been written back.

To resolve such hazards, two hardware mechanisms are used:

2.6.2. Forwarding (Bypassing)

Forwarding allows the result of a previous instruction to be passed directly to a subsequent instruction without waiting for it to be written back to the register file. The value is forwarded from the Memory or Writeback stages to the Execute stage via dedicated forwarding paths and multiplexers (Figure 8).

A Hazard Unit detects when a source register (Rs1E or Rs2E) matches the destination register (RdM or RdW) of a later-stage instruction and enables the forwarding logic accordingly. Registers like x0, which are always zero, are ignored in forwarding.

The forwarding logic for operand A (ForwardAE) works as:

if ((Rs1E == RdM) && RegWriteM && (Rs1E != 0)) ForwardAE = 10;

else if ((Rs1E == RdW) && RegWriteW && (Rs1E != 0)) ForwardAE = 01;

else ForwardAE = 00;

A similar condition applies for operand B (ForwardBE), using Rs2E.

This method significantly reduces the number of stalls and maintains throughput for most instructions, such as add, or, and, etc.

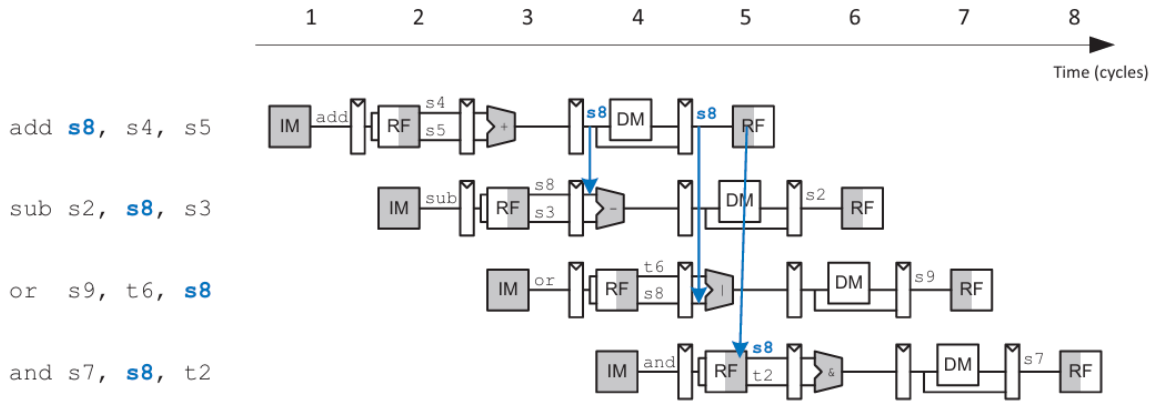

Figure 8. Abstract pipeline diagram illustrating forwarding [1]

2.6.3. Stalling for Load-Use Hazards

Load-Use hazards occur when an instruction loads data from memory (e.g., lw) and the next instruction immediately needs that data. Since memory access is completed at the end of the Memory stage, forwarding cannot help — the data simply is not ready in time for the next cycle (Figure 9).

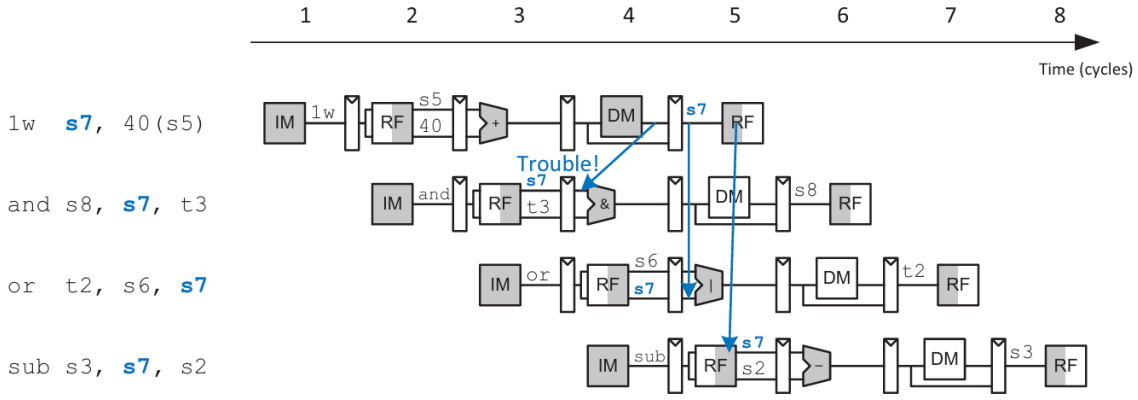

Figure 9. Abstract pipeline diagram illustrating trouble forwarding from load instruction (lw) [1]

In this case, the pipeline must stall. The Hazard Unit temporarily disables the Fetch and Decode pipeline registers (via StallF and StallD) and inserts a bubble (NOP) into the Execute stage by asserting FlushE. This bubble delays the dependent instruction, allowing the memory read to complete and forward the data in the next cycle (Figure 10).

The stall condition is computed as:

lwStall = ResultSrcE && ((Rs1D == RdE) || (Rs2D == RdE));

When lwStall is true: StallF = StallD = FlushE = 1

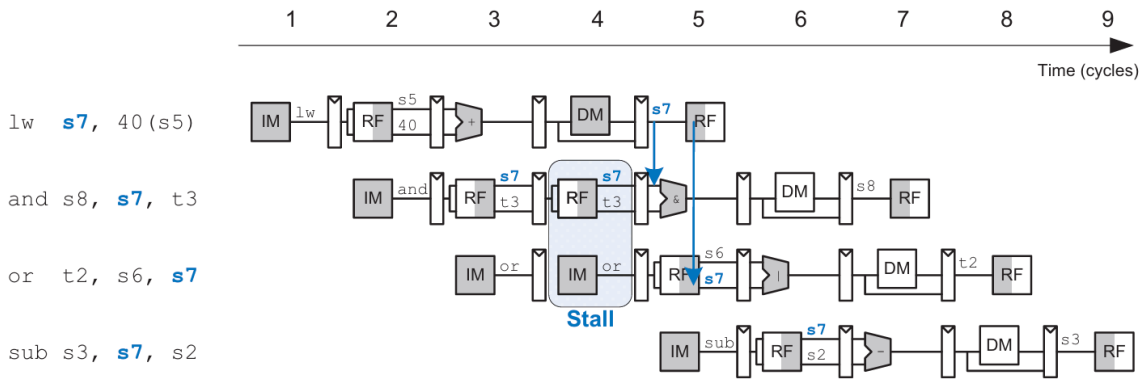

Figure 10. Abstract pipeline diagram illustrating stall to solve hazards [1]

2.6.4. Control Hazards and Branch Misprediction

A control hazard occurs when a branch decision (e.g., from beq) isn't resolved early enough. For example, in your processor, the branch target and decision (PCSrcE) are determined during the Execute stage, two cycles after the branch is fetched.

To deal with this, the pipeline predicts that branches are not taken and fetches the next sequential instruction. If this prediction turns out wrong (i.e., branch taken), two instructions following the branch must be flushed from the Decode and Execute stages:

FlushD = PCSrcE;

FlushE = lwStall || PCSrcE;

This introduces a branch misprediction penalty of two cycles. More sophisticated processors reduce this penalty using branch prediction logic, but in this basic implementation, flushing suffices (Figure 11).

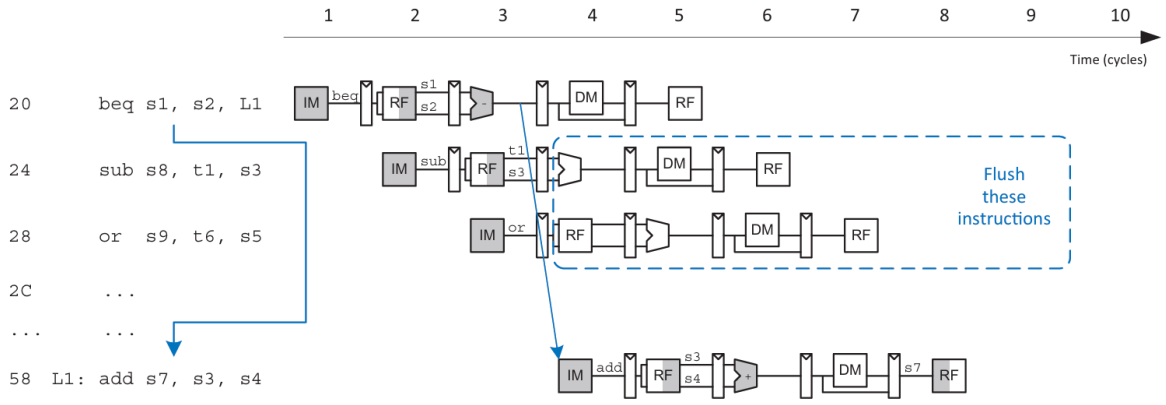

Figure 11. Abstract pipeline diagram illustrating flushing when a branch is taken [1]

Overall, a major design interest lies in how the datapath is structured and how control signals are generated. The datapath must route operands from registers or immediates to the ALU, handle memory operations, and correctly commit results. Special

attention is paid to forwarding units and hazard detection logic, which ensure data integrity during pipelined execution.

The methodology used in this project adopts the following structure:

ISA-Driven Design: The datapath and control logic are constructed directly from the instruction set specification, ensuring compliance with RV32I.

Modular RTL Design: Each pipeline stage is implemented as a separate module, and verification is done at both module and integration levels.

Synthesis-Oriented Coding: All designs are developed using synthesizable SystemVerilog, ensuring that they can be synthesized and deployed to FPGAs.

Simulation-Driven Verification: All modules are tested using self-checking testbenches, and waveform inspection helps identify design bugs.

By combining theoretical instruction set design with practical hardware implementation, this project bridges the gap between software-level instruction execution and hardware-level circuit realization. The resulting processor not only serves as a functional unit but also as a learning tool for understanding digital system design.

3. DESIGN SPECIFICATIONS

3.1. Design Specifications

The project aims to design, simulate, and synthesize a five-stage pipelined RISC-V processor that supports the full RV32I instruction set. Below are the specific and quantifiable design requirements:

- *Instruction Set Architecture (ISA):* RV32I base integer instruction set of RISC-V.
- *Pipeline Stages:* 5 stages: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), Writeback (WB).
- *Instruction Types Supported:*
    - R-type: add, sub, and, or, slt, sll, srl, sra, slt, sltu
    - I-type: addi, andi, ori, xori, slli, srli, srai, slti, sltiu, lb, lh, lw, lbu, lhu
    - S-type: sb, sh, sw
    - B-type: beq, bne, blt, bge, bltu, bgeu
    - U-type: lui, auipc
    - J-type: jal, jalr
    - Environment: ecall, ebreak
- *Instruction Word Length:* 32 bits
- *Register File:* 32 general-purpose registers (x0–x31), each 32 bits wide
- *Memory Interface:*
    - Instruction memory: Read-only
    - Data memory: Read – Write
- *Forwarding Unit:* Supports ALU result forwarding to avoid RAW hazards.
- *Hazard Handling:*
    - Data hazards: Forwarding + Stall
    - Control hazards: Static branch not-taken prediction + pipeline flushing
- *Clock Frequency Target:* Optimized for reduced clock period based on pipelining
- *Synthesis Target:* Altera DE10 – Standard FPGA (using Quartus Prime)

3.2. Design Analysis

To meet the requirements of high performance and full RV32I support, multiple architecture choices were considered:

Option 1: Single-Cycle Architecture

- Each instruction executes in one clock cycle.
- Simplified control logic.
- Disadvantage: The clock cycle must be long enough to accommodate the slowest instruction (e.g., lw). This leads to inefficient use of hardware resources and poor throughput.

Option 2: Multi-Cycle Architecture

- Breaks instruction execution into multiple steps, each mapped to a clock cycle.
- Improves cycle time by splitting work.
- Disadvantage: Still executes one instruction at a time; lower throughput than pipelining. Control logic is more complex.

Option 3: Pipelined Architecture (Chosen)

- Overlaps execution of multiple instructions by dividing the datapath into stages.
- Offers significantly higher instruction throughput and efficient resource utilization.
- Challenge: Must address hazards and ensure correctness using a hazard unit and forwarding logic.

Conclusion:

The pipelined architecture was chosen as the optimal approach due to its superior throughput and relevance to real-world processor design. Despite added complexity in control and hazard resolution, it provides a realistic and efficient processor model.

4. BLOCK DIAGRAMS

4.1. High-Level Block Diagram

The high-level view of the system includes three main components:

- **Instruction Memory (IMEM):** Provides instructions to the processor based on the program counter.
- **Processor Core:** Includes the datapath, control unit, and hazard unit. It is responsible for executing instructions by coordinating operations across the five pipeline stages.
- **Data Memory (DMEM):** Stores and retrieves data as directed by load/store instructions.

Each block interacts through well-defined interfaces. The processor receives instructions from IMEM and performs computations and memory access via DMEM. The hazard unit inside the processor ensures correct sequencing and resolution of data and control hazards.

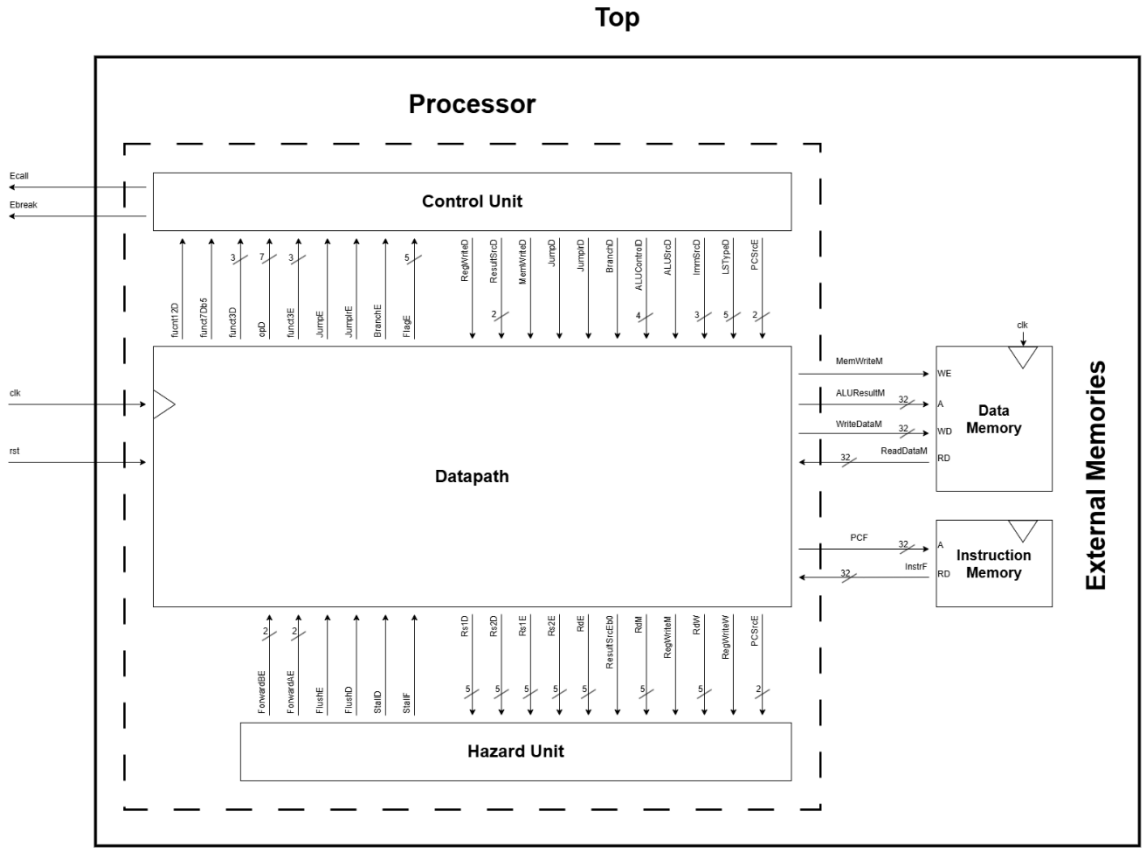

Figure 12. 5-Stage RV32I Pipelined Processor with External Instruction and Data Memory Interfaces

4.2. Detailed Block Diagram

The detailed block diagram expands the processor core into its submodules:

- *Datapath:* Implements the functional units (e.g., ALU, register file, extender) and handles the movement of data between pipeline stages.
- *Control Unit:* Includes multiple sub-decoders:
  - Main Decoder generates high-level control signals.
  - ALU Decoder determines ALU operation.
  - Branch Unit (BRU) handles branch evaluation and prediction.
  - Environment Unit (EU) processes environment instructions.
  - Load – Store Unit (LSU) manages load and store control logic.
- *Hazard Unit:* Detects and resolves hazards via:
  - Forwarding from memory/writeback stages
  - Load-use stalls
  - Flushing instructions on branch misprediction

Each stage is separated by pipeline registers, and each signal is tagged with its pipeline stage suffix (F, D, E, M, W). This systematic tagging ensures correct timing and tracking across the instruction pipeline.

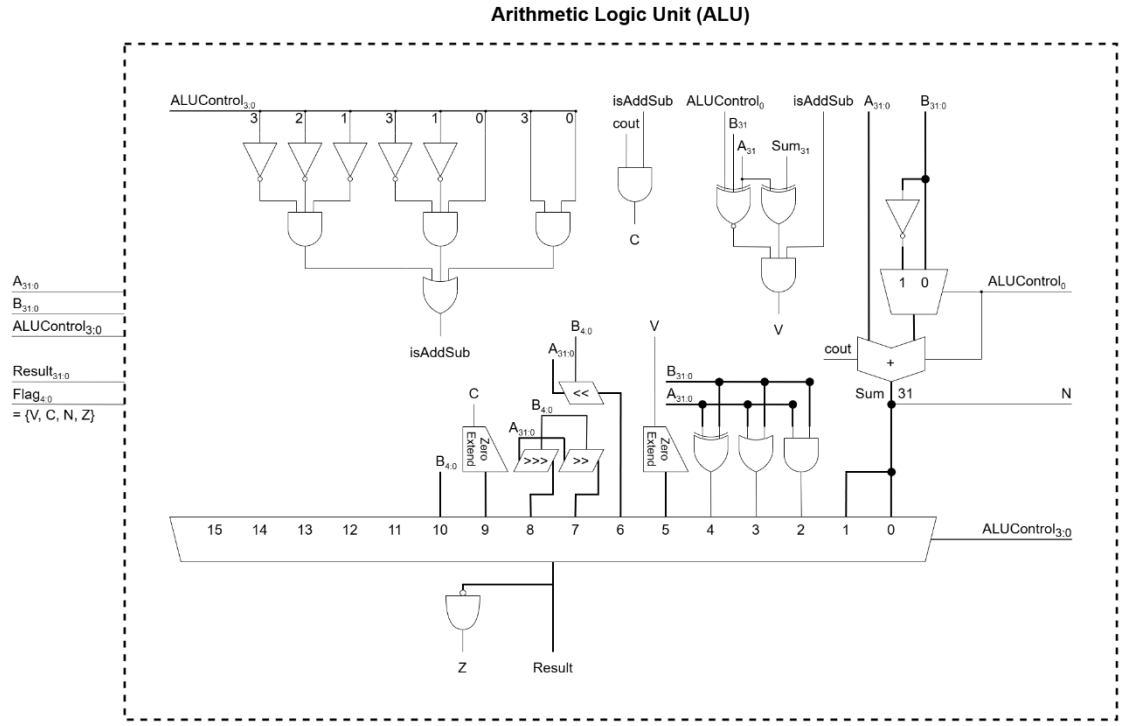

Figure 13. Arithmetic Logic Unit of 5-Stage RV32I Pipelined Processor

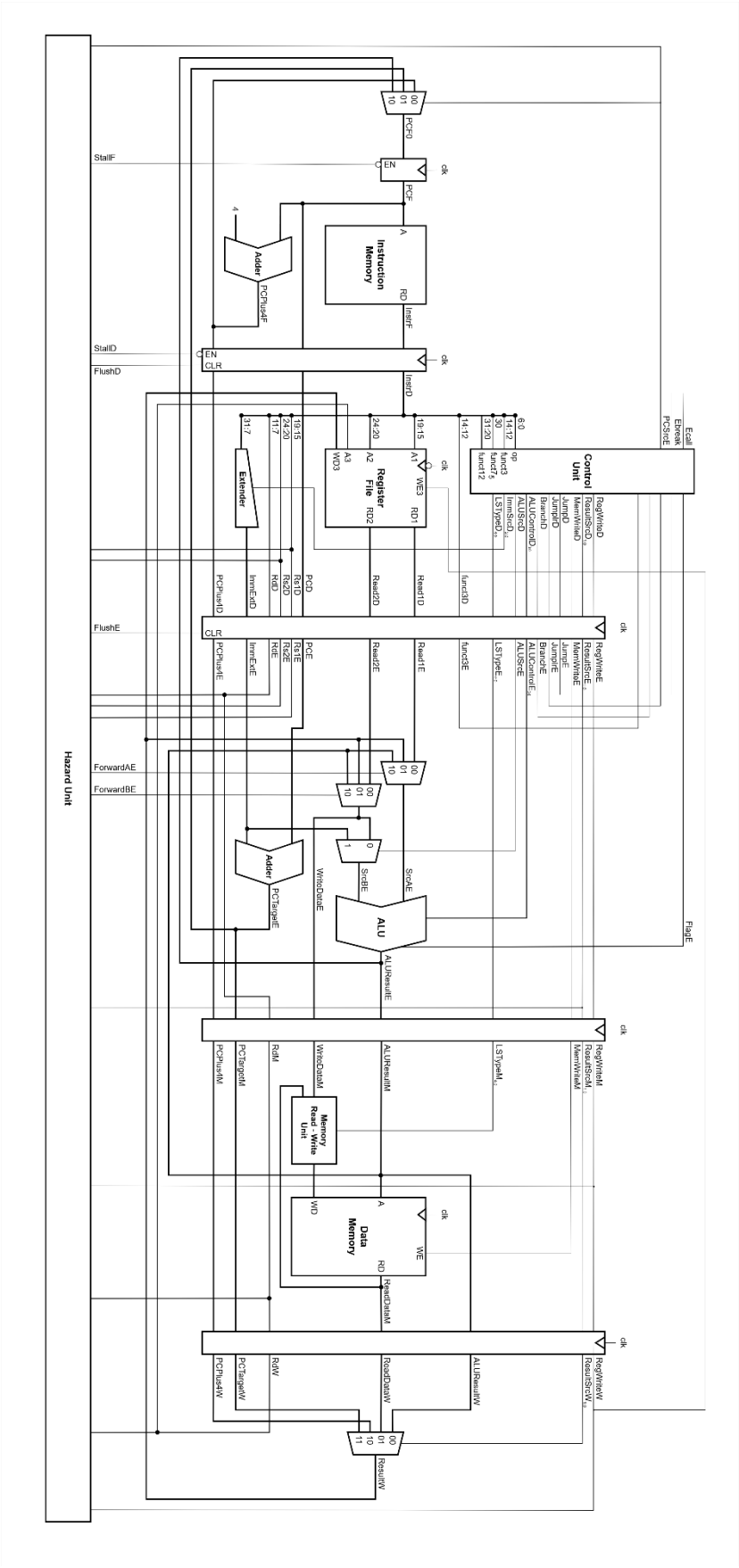

Figure 14. Datapath of 5-Stage RV32I Pipelined Processor with Full Hazard Detection, Forwarding, and Control Logic

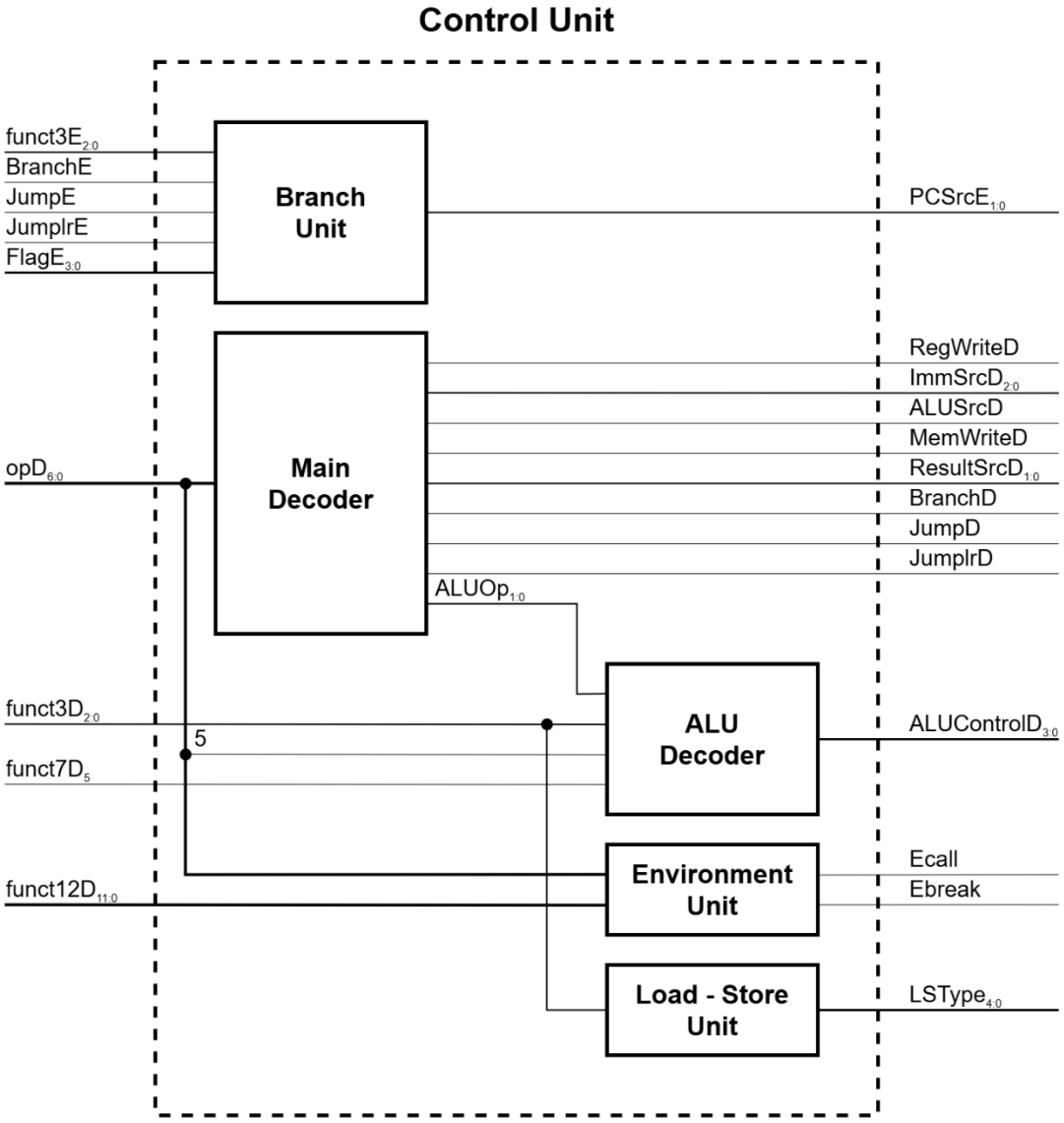

Figure 15. Control Unit of 5-Stage RV32I Pipelined Processor

5. INPUT – OUTPUT DESCRIPTION

The 5-stage pipelined RISC-V processor communicates internally and externally through a well-organized set of signal and bus lines. These signals serve as the backbone of interaction between the processor's submodules (such as datapath, control unit, hazard detection, etc.) and the external memory components. To ensure modularity, clarity, and ease of debugging, each signal is clearly categorized by its role in one of the five pipeline stages: Fetch (F), Decode (D), Execute (E), Memory (M), and Writeback (W), as well as global control signals.

Each signal is characterized by its name, bit width, source, destination, and functional description. Collectively, these signals support the correct flow of instructions and data through the pipeline, ensure synchronization, and manage hazards and control decisions.

In this section, we present a detailed table that categorizes all signal and bus lines used in the processor. The table includes:

- ***Global control signals*** such as the system clock (clk) and reset (rst) that govern synchronous operation.
- ***Fetch stage signals*** used to read the instruction memory and update the program counter.
- ***Decode stage signals*** responsible for decoding instructions and preparing control signals, operands, and immediate values.
- ***Execute stage signals*** that drive the ALU, branch logic, and prepare data for memory access or register writeback.
- ***Memory stage signals*** used for reading from or writing to the data memory.
- ***Writeback stage signals*** that finalize the computation by updating the register file with results from the ALU, memory, or control flow.

These signals are not only essential for functional correctness but also crucial for understanding how each stage of the processor operates independently yet cooperatively. Additionally, signals that support data forwarding, stalls, and pipeline flushes are highlighted, as they play a pivotal role in resolving hazards and maintaining pipeline efficiency.

By clearly documenting the interface and interconnect of the processor, this signal description serves as a foundation for simulation, synthesis, verification, and future extensions of the design.

Table 1. Categorized Signal and Bus Lines

Signal Name	Bit Width	Source	Destination	Description
Global Signals				
clk	1	Global	All synchronous logic	Clock signal
rst	1	Global	All reset logic	Asynchronous reset
Ecall	1	Controller	Global	Transfer control to OS
Ebreak	1	Controller	Global	Transfer control to debugger
Fetch Stage (F)				
pcF0	32	PCmux	Freg	Next PC value
pcF	32	Freg	Instruction Memory	Program Counter (current)
InstrF	32	Instruction Memory	Dreg	Fetched instruction
PCPlus4F	32	PCplus4 adder	Dreg	PC + 4 for sequential flow
StallF	1	Hazard Unit	Freg	Stall PC update
Decode Stage (D)				
InstrD	32	Dreg	Controller	Decoded instruction
funct3D	3	InstrD	Ereg	Branch/ Jump type
Rs1D, Rs2D	5	InstrD	Hazard Unit, Ereg	Source register IDs
RdD	5	InstrD	Ereg	Destination register
Read1D, Read2D	32	Register file	Ereg	Register read data
pcD	32	Dreg	Ereg	PC value in decode
ImmExtD	32	extender	Ereg	Extended immediate
PCPlus4D	32	Dreg	Ereg	PC+4 forwarded
RegWriteD	1	Controller	Ereg	Register write enable
ResultSrcD	2	Controller	Ereg	Result source select
MemWriteD	1	Controller	Ereg	Memory write enable
JumpD, JumplrD, BranchD	1	Controller	Ereg	Jump/Branch control
ALUControlD	4	Controller	Ereg	ALU operation control
ALUSrcD	1	Controller	Ereg	ALU source select
ImmSrcD	3	Controller	extender	Immediate format select
LSTypeD	5	Controller	Ereg	Load/ Store type
StallD	1	Hazard Unit	Dreg	Stall instruction update
FlushD	1	Hazard Unit	Dreg	Clear instruction
Execute Stage (E)				
funct3E	3	Ereg	Controller	Branch/ Jump type

Rs1E, Rs2E	5	Ereg	Hazard Unit	Source register
RdE	5	Ereg	Mreg, Hazard Unit	Destination register
Read1E, Read2E	32	Ereg	Forwarding muxes	Register data
pcE	32	Ereg	PCplusbranch	PC in execute
ImmExtE	32	Ereg	srcBmux2	Extended immediate
PCPlus4E	32	Ereg	Mreg	PC+4 forwarded
SrcAE, SrcBE	32	Muxes	ALU	ALU operands
ALUResultE	32	ALU	Mreg	ALU result
WriteDataE	32	srcBmux1	Mreg	Store data
PCTargetE	32	PCplusbranch	Pcmux, Mreg	Branch/Jump target
RegWriteE	1	Ereg	Mreg	Register write enable
ResultSrcE	2	Ereg	Mreg, Hazard Unit	Result select
MemWriteE	1	Ereg	Mreg	Memory write enable
JumpE, JumplrE, BranchE	1	Ereg	Controller	Jump/Branch control
ALUControlE	4	Ereg	ALU	Select ALU operation
ALUSrcE	1	Ereg	srcBmux2	ALU source B select
LSTypeE	5	Ereg	Mreg	Load/ Store type
FlagE	5	ALU	Controller	ALU flag for branch
PCSrcE	2	Controller	PCmux	Select branch PC
ForwardAE, ForwardBE	2	Hazard Unit	Forwarding muxes	Forwarding control
FlushE	1	Hazard Unit	Ereg	Flush data
Memory Stage (M)				
ALUResultM	32	Mreg	Data Memory/Wreg	Memory address/ALU result
WriteDataM	32	Mreg	Data Memory	Store data
RdM	5	Mreg	Wreg, Hazard Unit	Destination register
PCPlus4M	32	Mreg	Wreg	PC+4 forwarded
PCTargetM	32	PCplusbranch	Wreg	Branch/Jump target
RegWriteM	1	Mreg	Wreg	Register write enable
ResultSrcM	2	Mreg	Wreg	Result select
MemWriteM	1	Mreg	Data Memory	Memory write enable
LSTypeM	5	Mreg	Wreg	Load/ Store type
Writeback Stage (W)				
ALUResultW	32	Wreg	rsltmux	ALU result
ReadDataW	32	Wreg	rsltmux	Memory read data
PCPlus4W	32	Wreg	rsltmux	PC+4 value
PCTargetW	32	PCplusbranch	rsltmux	Branch/Jump target
ResultW	32	rsltmux	reg_file	Write data
RegWriteW	1	Wreg	reg_file	Register write enable
ResultSrcW	2	Wreg	rsltmux	Result select
RdW	5	Wreg	reg_file, Hazard Unit	Destination register

6. FUNCTIONAL DESCRIPTION

This section presents the detailed internal behavior of the 5-stage pipelined RV32I processor through the interaction of its submodules. It describes the functional operations during instruction execution and is supported by control signal decoding and waveform-based simulation results for verification.

6.1. Instruction Execution Overview

The processor consists of five pipeline stages: **Fetch**, **Decode**, **Execute**, **Memory**, and **Writeback**. Each stage is separated by pipeline registers that hold instruction-specific data and control signals.

Fetch Stage:

- The **Program Counter (PC)** selects the address of the instruction to be fetched from the Instruction Memory.
- The instruction is retrieved and passed to the **Decode stage** through the InstrF register.
- Simultaneously, PC + 4 is computed using an adder and forwarded to support control transfer instructions.

Decode Stage:

- The instruction is decoded to extract the opcode, funct3, funct7, rd, rs1, rs2, and immediate fields.
- The **Register File** provides source register values (Read1D, Read2D) based on Rs1D, Rs2D.
- The **Immediate Extender**, controlled by ImmSrc, generates the extended immediate value as shown in Table 2.
- The **Main Decoder** uses the opcode to generate control signals: register write enable, memory write enable, ALU operation selector, result source, and branch/jump indicators (Table 3).
- The **ALU Decoder** further decodes the ALU operation from the ALUop, funct3, and funct7 fields (Table 4).
- Additional decoders include the **Branch Unit (BRU)** and **Environment Unit (EU)**, responsible for computing control flow changes and detecting ecall/ebreak (Tables 6 and 7).

Table 2. Immediate extender truth table

ImmSrc	ImmExt	Type
000	{{20{Instr[31]}}, Instr[31:20]}	**I**
001	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	**S**
010	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}	**B**
011	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}	**J**
100	{Instr[31:12], 12'b0}	**U**

Table 3. Main decoder truth table

op	Reg-Write	Imm-Src	ALU-Src	Mem-Write	Result-Src	Branch	ALU-Op	Jump	Jumplr	Instruction
0110011	1	xxx	0	0	00	0	10	0	0	R-type
0010011	1	000	1	0	00	0	10	0	0	I-type ALU
0000011	1	000	1	0	01	0	00	0	0	I-type load
0100011	0	001	1	1	x0	0	00	0	0	S-type
1100011	0	010	0	0	x0	1	01	0	0	B-type
1101111	1	011	x	0	10	0	xx	1	0	J-type
1100111	1	000	1	0	10	0	10	0	1	I-type jalr
0110111	1	100	1	0	00	0	11	0	0	U-type lui
0010111	1	100	x	0	11	0	xx	0	0	U-type auipc
1110011	0	xxx	x	0	x0	0	xx	0	0	ecall, ebreak

Table 4. Arithmetic Logic Unit (ALU) decoder truth table

ALUOp	funct3	{op_5, $funct7_5$}	ALUControl	Operation	Instruction
00	x	xx	0000	add	I-type load, S-type
01	xxx	xx	0001	subtract	B-type
10	000	00, 01, 10	0000	add	add, addi, jalr
		11	0001	subtract	sub
	001	xx	0110	shift left logical	sll, slli
	010	xx	0101	set less than	slt, slti
	011	xx	1001	set less than (U)	sltu, sltui
	100	xx	0100	xor	xor, xori
	101	x0	0111	shift right logical	srl, srli
		x1	1000	shift right arithmetic	sra, srai
	110	xx	0011	or	or, ori
	111	xx	0010	and	and, andi
11	x	xx	1010	take immediate only	lui

Table 5. Load - Store Unit (LSU) truth table

funct3	LSType	Type	Instruction
000	10000	Byte	lb, sb
001	01000	Half	lh, sh
010	00100	Word	lw, sw
100	00010	Byte (U)	lbu
101	00001	Half (U)	lhu

Table 6. Branch Unit (BRU) truth table

Branch	Jump	Jumplr	funct3	Flag = {V, C, N, Z}	$PCSrc_1$	$PCSrc_0$	Instruction
1	0	0	000	xxxx	0	Z	beq
			001	xxxx	0	~ Z	bne
			100	xxxx	0	N ^ V	blt
			101	xxxx	0	~ (N ^ V)	bge
			110	xxxx	0	~ C	bltu
			111	xxxx	0	C	bgeu
0	1	0	xxx	xxxx	0	1	jal
0	0	1	xxx	xxxx	1	0	jalr

Table 7. Environment Unit (EU) truth table

op	funct12	Ecall	Ebreak	Instruction
1110011	000000000000	1	x	ecall
1110011	000000000001	x	1	ebreak

Execute Stage:

- ALU operands are selected via muxes using ForwardAE and ForwardBE signals, which are determined by the **Hazard Unit** for data hazard resolution.
- The ALU executes the arithmetic or logic operation defined by ALUControl, producing ALUResultE and status flags (Z, N, V, C) used by the **Branch Unit** to compute PCSrcE.
- The **Branch Unit (BRU)** determines control flow redirection based on flags and control signals. The PC is updated via PCSrc if a branch is taken.
- Jump and Link (jal, jalr) instructions write PC + 4 to the destination register and redirect control flow accordingly.

Memory Stage:

- If the instruction is a memory access type (lw, lh, lb, sw, sh, sb), the **Load Store Unit (LSU)** uses funct3 to determine load/store size and type, as defined in Table 5.
- Data is either read from memory (ReadDataW) or written (WriteDataM) based on MemWriteM.

Writeback Stage:

- The final result is selected based on ResultSrcW: ALU result, memory data, PC + 4, or PCTarget.
- The result is written back to the **Register File** at the destination register (RdW), if RegWriteW is asserted.

6.2. Hazard Handling

To resolve **data hazards**, the processor implements:

- **Forwarding** from ALUResultM or ResultW back to the ALU input muxes.
- **Stalling** in the Decode stage when a lw instruction is followed by a dependent instruction, determined by lwStall from the Hazard Unit.
- **Bubbles** are introduced by flushing the Execute stage (FlushE) to delay dependent instructions.

For **control hazards**, the processor assumes **branches are not taken**. When a branch is taken, the **Hazard Unit** asserts FlushD and FlushE to invalidate instructions already fetched or decoded.

Forward to solve data hazards when possible:

if ((Rs1E = = RdM) & RegWriteM) & (Rs1E != 0) then ForwardAE = 10

else if ((Rs1E = = RdW) & RegWriteW) & (Rs1E != 0) then ForwardAE = 01

else ForwardAE = 00

Stall when a load hazard occurs:

lwStall = ResultSrcE0 & ((Rs1D = = RdE) | (Rs2D = = RdE))

StallF = lwStall StallD = lwStall

Flush when a branch is taken or a load introduces a bubble:

FlushD = PCSrcE FlushE = lwStall | PCSrcE

6.3. Simulation Illustration

To verify the correct functionality of the designed 5-stage pipelined processor, a comprehensive test program was executed using the testbench processor_tb.sv. The

test program was written in RV32I machine code and manually fed into the instruction memory module (i_mem.sv). Its goal is to test different instruction types, hazard cases, and memory operations, with the expected final result being the value 0xABCDE02E stored at memory address 0x28C (decimal 652).

The test begins at instruction address 0x00, performing basic arithmetic operations using addi, or, xor, and add. It also includes a sequence of branch instructions (beq, jal) to verify control hazard handling, and load-store instructions (lw, sw) to test the memory interface. Specific instructions intentionally trigger hazards to ensure the forwarding and hazard units respond correctly.

As shown in the simulation result log (sim_rslt.sv), instruction fetching begins at cycle 1 and completes with the infinite loop at address 0x44 by cycle 32, confirming a successful execution of 32 instructions across the pipeline. No erroneous values were written, no incorrect branches taken, and memory was updated precisely as expected.

```
# If successful, it should write the value 0xABCDE02E to address 132 (0x84)
#       RISC-V Assembly         Description               Address   Machine
Code
main:   addi x2, x0, 5          # x2 = 5                  0         00500113
        addi x3, x0, 12         # x3 = 12                 4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3       8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7       C         0023E233
        xor  x5, x3, x4         # x5 = (12 XOR 7) = 11    10        0041C2B3
        add  x5, x5, x4         # x5 = (11 + 7) = 18      14        004282B3
        beq  x5, x7, end        # shouldn't be taken      18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0       1C        0041A233
        beq  x4, x0, around     # should be taken         20        00020463
        addi x5, x0, 0          # shouldn't happen        24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5)  = 1       28        0023A233
        add  x7, x4, x5         # x7 = (1 + 18) = 19      2C        005203B3
        sub  x7, x7, x2         # x7 = (19 - 5) = 14      30        402383B3
        sw   x7, 84(x3)         # [96] = 14               34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 14          38        06002103
        add  x9, x2, x5         # x9 = (14 + 18) = 32     3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44  40        008001EF
        addi x2, x0, 1          # shouldn't happen        44        00100113
end:    add  x2, x2, x9         # x2 = (14 + 32)  = 46    48        00910133
        addi x4, x0, 1          # x4 = 1                  4C        00100213
        lui  x5, 0x80000        # x5 = 0x80000000         50        800002b7
        slt  x6, x5, x4         # x6 = 1                  54        0042a333
wrong:  beq  x6, x0, wrong      # shouldn’t be taken      58        00030063
        lui  x9, 0xABCDE        # x3 = 0xABCDE000         5C        ABCDE4B7
        add  x2, x2, x9         # x2 = 0xABCDE02E         60        00910133
        sw   x2, 0x40(x3)       # mem[132] = 0xABCDE02E   64        0421a023
done:   beq  x2, x2, done       # infinite loop           68        00210063
```

Figure 16. Test program to verify operation including: normal operation, stall operation, flush operation [1]

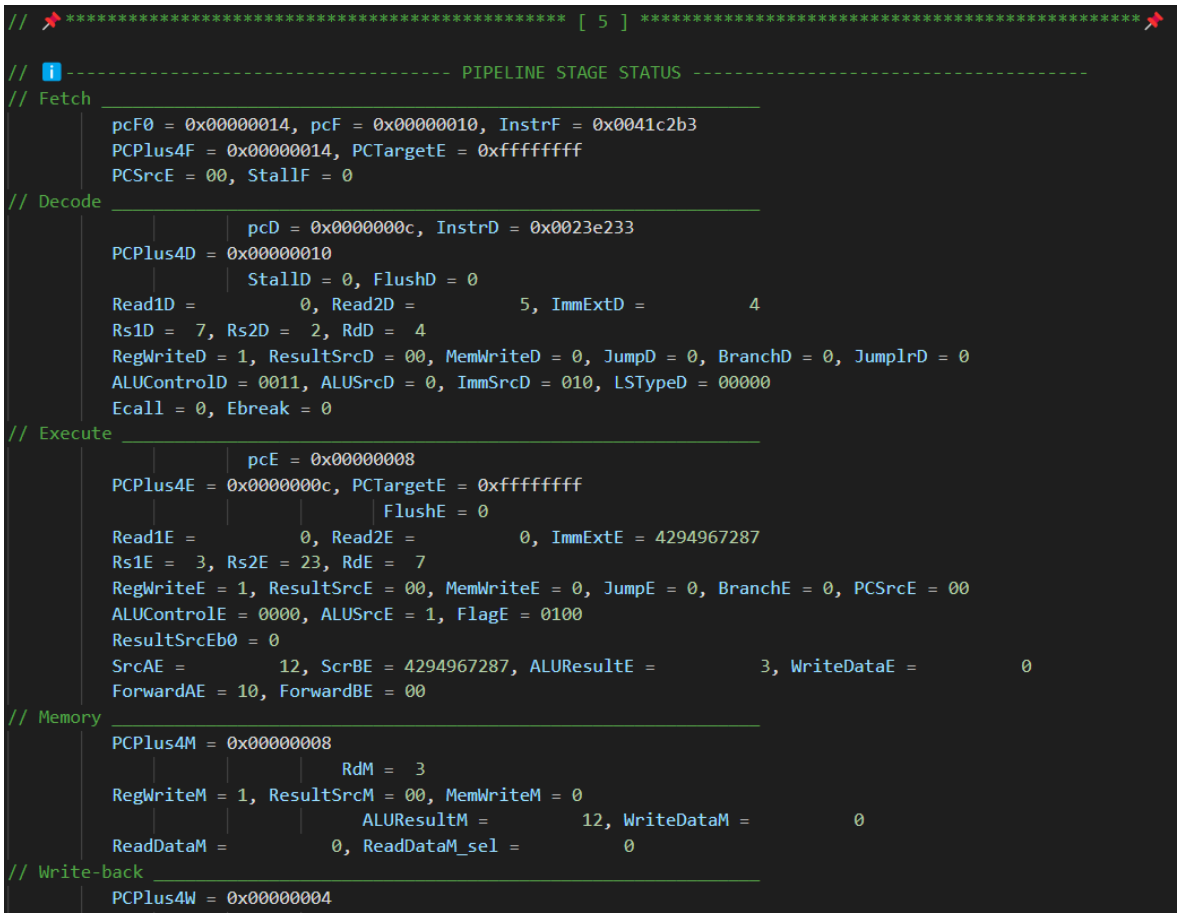

Figure 17. Displayed pipeline stage status from simulation at cycle 5, when the first 5 instructions are at 5 different stages of the processor

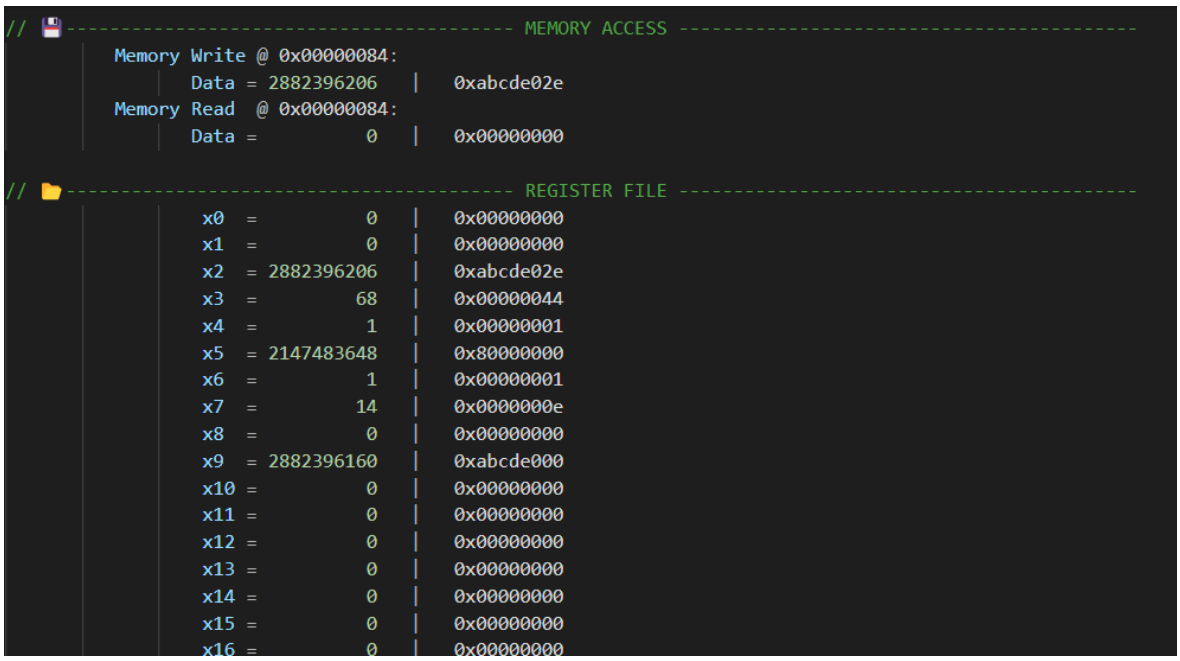

Figure 18. Displayed memory and register file data from simulation at cycle 32, when all instructions have been executed

6.4. Waveform Analysis

The waveform output, captured using GTKWave and enabled through Verilator's VCD dump, provides a deep insight into the real-time behavior of the processor. Critical signals such as pcF, InstrF, register file writebacks, and ALU outputs were monitored.

Some important observations from the waveform:

- **Pipeline Execution**: Each instruction progresses through the pipeline stages IF → ID → EX → MEM → WB in sequential clock cycles, illustrating correct pipeline behavior.
- **Hazard Management**: The processor correctly handles read-after-write (RAW) hazards using forwarding for dependent instructions such as add x7, x4, x5 at cycle 5 and sub x7, x7, x2 at cycle 15. No stalls occurred in those cases, showing the forwarding unit functioned properly at cycle 15 (Figure and Figure).
- **Load-Use Stall**: In the lw x2, 96(x0) followed by add x9, x2, x5, a stall is intentionally inserted as forwarding cannot resolve load-use dependencies at cycle 18. The pipeline bubbles introduced during this stall are clearly visible (Figure).
- **Branch and Jump Behavior**: When beq x5, x7, end is not taken, execution proceeds sequentially at cycle 8 (Figure). Conversely, the beq x4, x0, around branch is taken correctly at cycle 11 (Figure same). The jal x3, end performs an unconditional jump with correct update of x3 to the return address at cycle 21 (Figure).
- **Final Memory Write**: The last store instruction sw x2, 0x40(x3) correctly places the result 0xABCDE02E into address 0x84 at cycle 32. This is confirmed both in memory waveforms and by simulation logs.

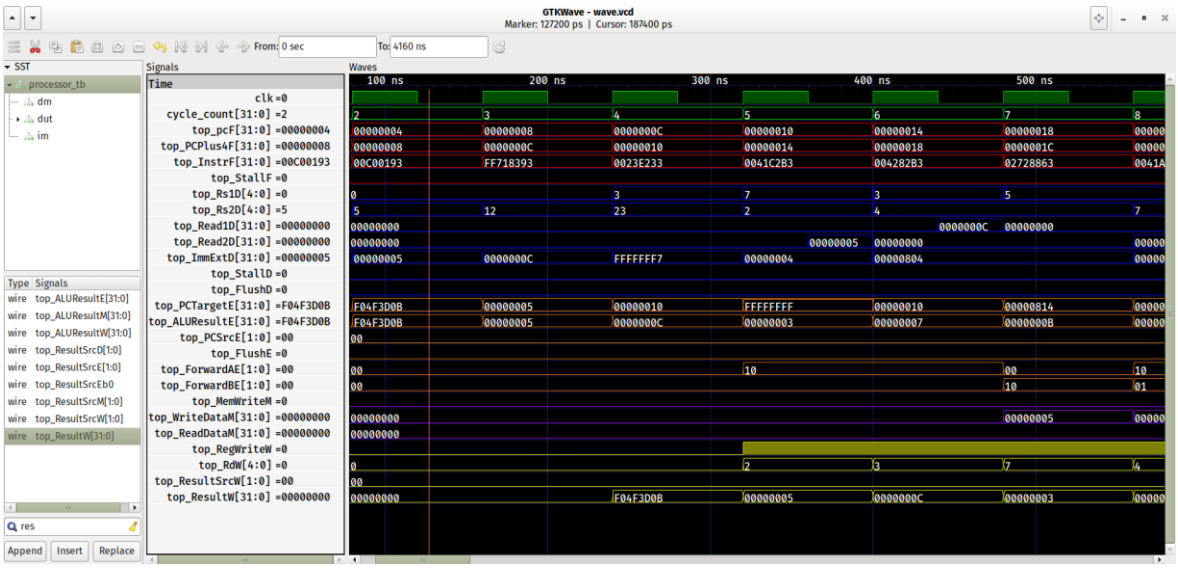

Figure 19. Waveform Analysis – Cycle 5: Data hazard solved by forwarding

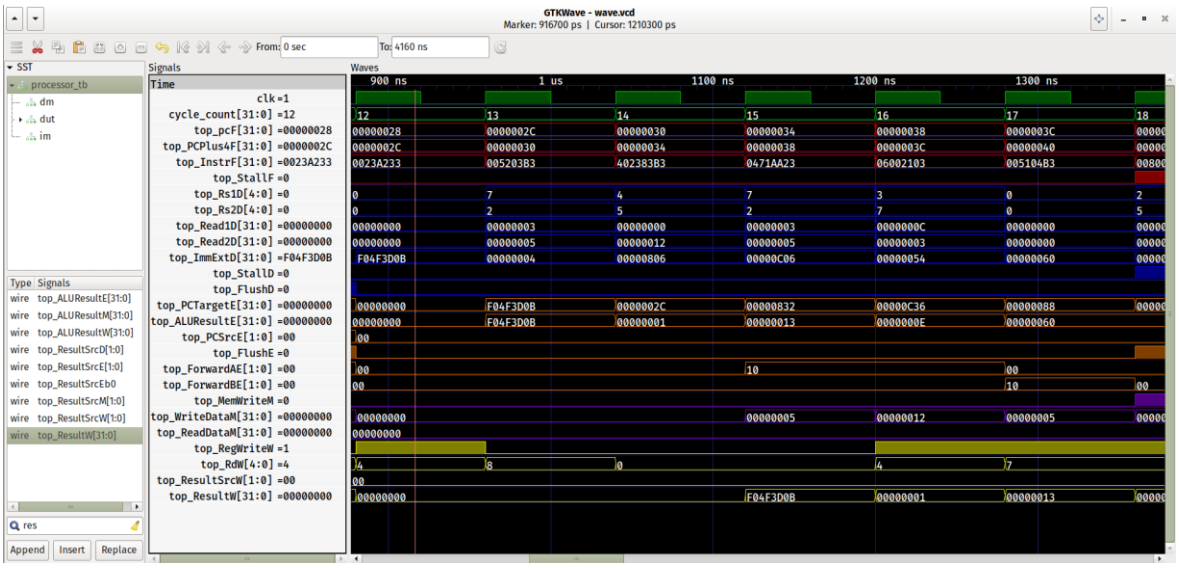

Figure 20. Waveform Analysis – Cycle 15: Data hazard solved by forwarding

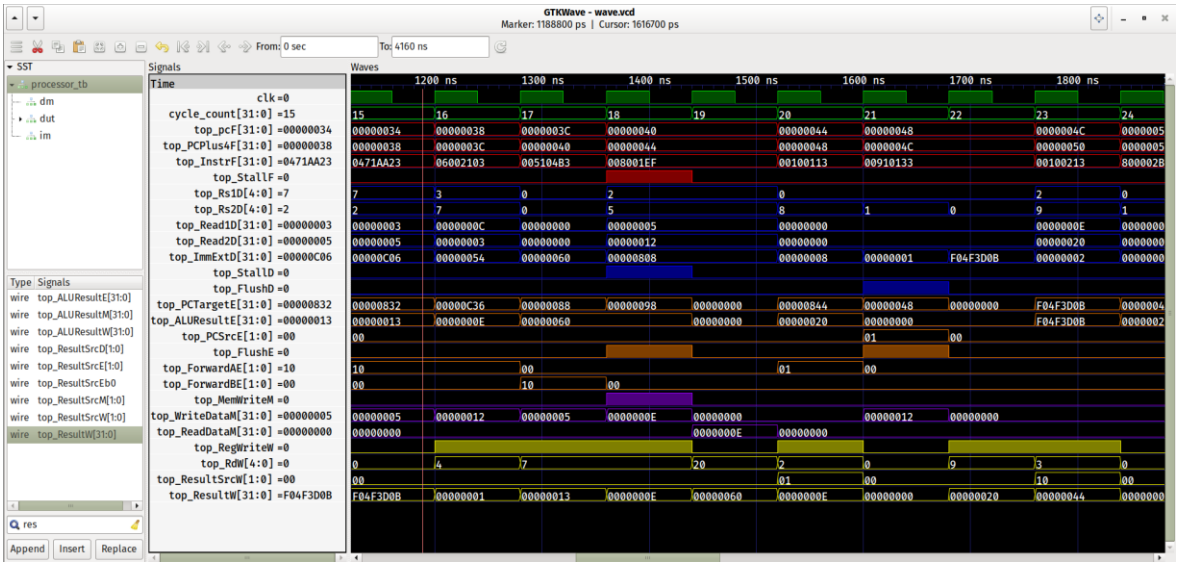

Figure 21. Waveform Analysis – Cycle 18: Load-use stall

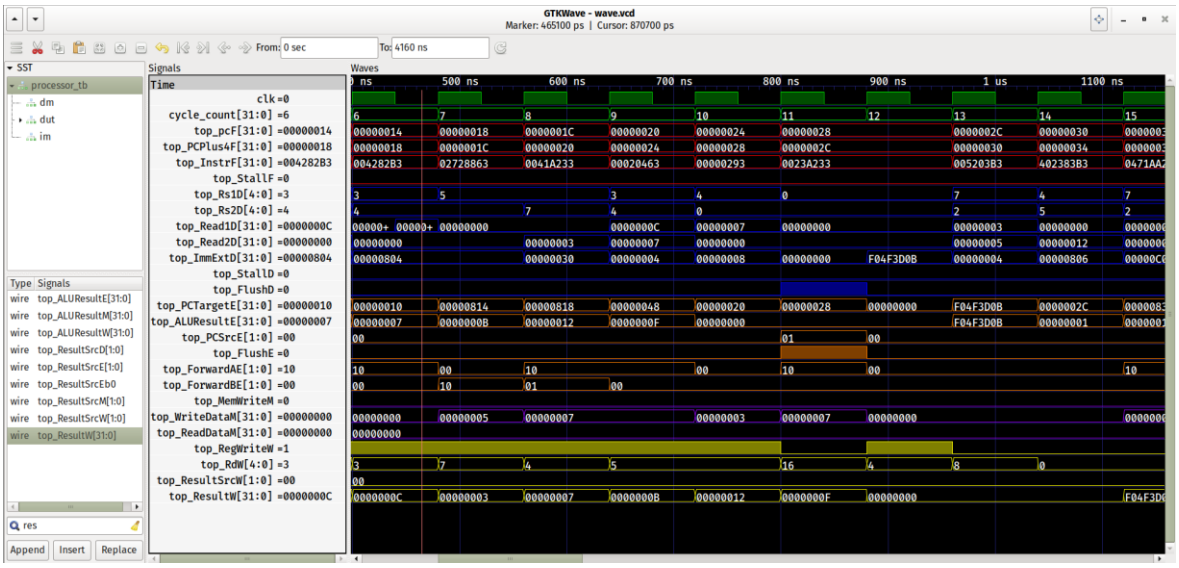

Figure 22. Waveform Analysis – Cycle 8, 11: Branch not taken and branch taken

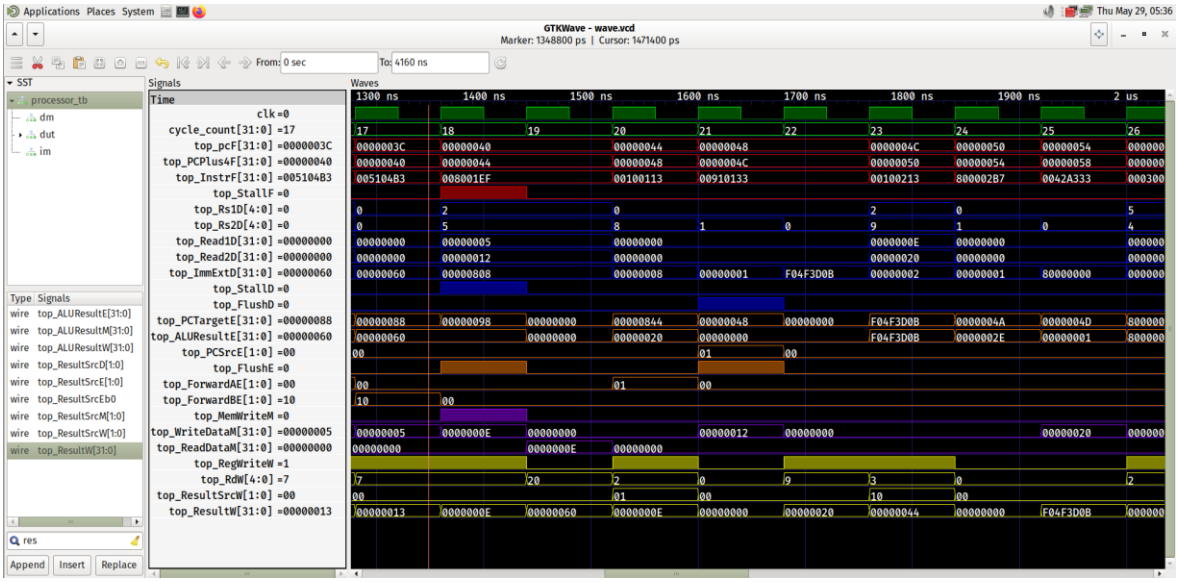

Figure 23. Waveform Analysis – Cycle 21: Jump

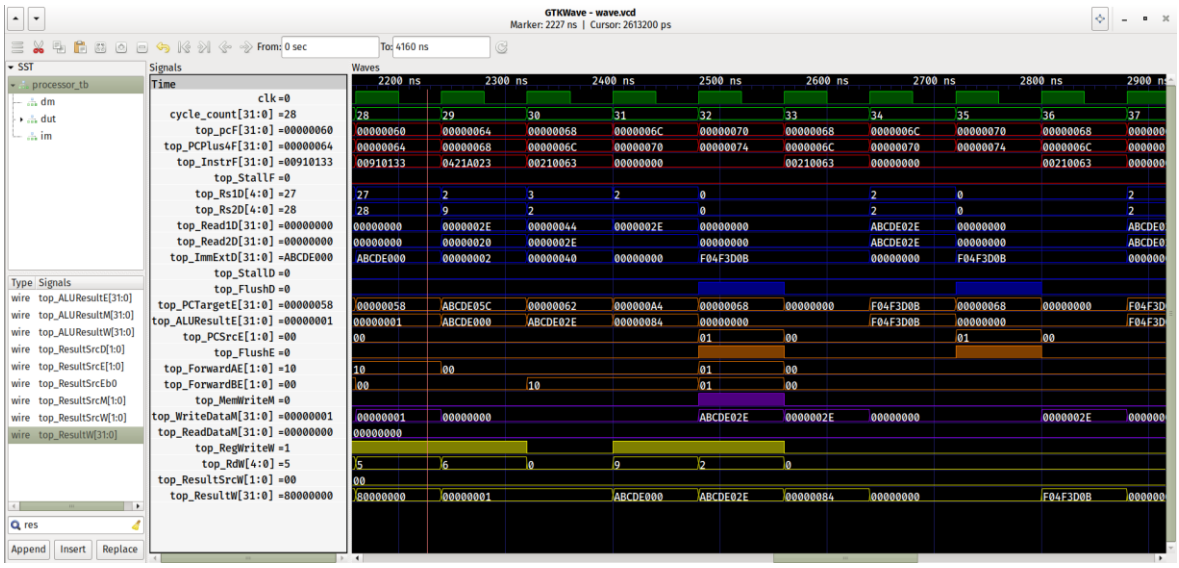

Figure 24. Waveform Analysis – Cycle 32: Memory write

The timing alignment and correct result propagation across pipeline registers such as pcE, pcM, and pcW, along with the expected memory output, demonstrate that the processor correctly implements the RV32I instruction set with robust handling of hazards and memory I/O.

7. RTL SIMULATION RESULTS

7.1. Verification Method

To ensure the functional correctness of the 5-stage pipelined RV32I processor, a comprehensive verification strategy was employed. This involved:

- **Unit-Level Verification**: Individual modules such as the ALU, Register File, Immediate Extender, Control Unit, Hazard Unit, and Pipeline Registers were each tested using dedicated testbenches. These testbenches applied a range of input vectors to validate the expected outputs under various scenarios.
- **Integration Testing**: After unit-level verification, modules were integrated incrementally. Each integration step was accompanied by specific testbenches to verify the correct interaction between modules.
- **System-Level Verification**: The complete processor was subjected to system-level tests using a comprehensive testbench (processor_tb.sv). This testbench simulated realistic instruction sequences to validate the processor's behavior under typical operating conditions.
- **Simulation Tools**: Verilator was utilized as the primary simulation tool due to its efficiency in handling large designs and its capability to generate cycle-accurate models. Waveform analysis was conducted using GTKWave to visualize signal transitions and validate timing relationships.

This multi-tiered verification approach ensured that both individual modules and the integrated processor met the design specifications and operated correctly under various conditions.

7.2. Testbench Setup

The testbench hierarchy was organized as follows:

- **Module-Level Testbenches**: Located in the ../01_bench/ directory, each module had an associated testbench designed to:
    - Apply a comprehensive set of input stimuli.
    - Monitor outputs and compare them against expected results.
    - Report discrepancies for debugging purposes.
- **Top-Level Testbench (processor_tb.sv)**:
    - **Instruction Memory Initialization**: The instruction memory (i_mem.sv) was manually populated with a sequence of RV32I instructions designed to test various processor functionalities, including arithmetic operations, branching, memory access, and hazard scenarios.

- **Clock and Reset Generation**: The testbench generated a clock signal and managed the reset sequence to initialize the processor.
- **Monitoring and Logging**: Key signals such as program counter (pcF), instruction register (InstrF), register write-back data, and memory addresses were monitored. Outputs were logged to sim_rslt.sv for post-simulation analysis.
- **Waveform Generation**: A Value Change Dump (VCD) file was generated during simulation to facilitate waveform analysis using GTKWave.

This structured testbench setup allowed for thorough testing at both module and system levels, ensuring comprehensive coverage of the processor's functionality.

```
// DUT instantiation
processor dut (
    // Debugging ----------
    .top_regfile_addr   (top_regfile_addr),
    .top_regfile_data   (top_regfile_data),
    .top_pcF0           (top_pcF0),
    .top_PCPlus4F       (top_PCPlus4F),
    .top_InstrD         (top_InstrD),
    .top_pcD            (top_pcD),
    .top_ImmExtD        (top_ImmExtD),
    .top_PCPlus4D       (top_PCPlus4D),
    .top_Read1D         (top_Read1D),
    .top_Read2D         (top_Read2D),
    .top_RdD            (top_RdD),
    .top_Read1E         (top_Read1E),
    .top_Read2E         (top_Read2E),
    .top_pcE            (top_pcE),
    .top_ImmExtE        (top_ImmExtE),
    .top_PCPlus4E       (top_PCPlus4E),
    .top_PCTargetE      (top_PCTargetE),
    .top_SrcAE          (top_SrcAE),
    .top_SrcBE          (top_SrcBE),
    .top_ALUResultE     (top_ALUResultE),
    .top_WriteDataE     (top_WriteDataE),
    .top_RegWriteE      (top_RegWriteE),
    .top_MemWriteE      (top_MemWriteE),
    .top_ResultSrcE     (top_ResultSrcE),
    .top_ALUSrcE        (top_ALUSrcE),
    .top_ALUControlE    (top_ALUControlE),
    .top_PCPlus4M       (top_PCPlus4M),
    .top_ResultSrcM     (top_ResultSrcM),
    .top_ReadDataM_sel  (top_ReadDataM_sel),
    .top_ResultSrcW     (top_ResultSrcW),
    .top_ALUResultW     (top_ALUResultW),
    .top_ReadDataW      (top_ReadDataW),
    .top_PCPlus4W       (top_PCPlus4W),
    .top_ResultW        (top_ResultW),
```

Figure 25. Testbench setup – Output top-level signals for debugging purpose

```
d_mem dm (
    .clk    (clk),
    .we     (top_MemWriteM),
    .a      (top_ALUResultM),
    .wd     (top_WriteDataM),
    .rd     (top_ReadDataM)
);

i_mem im (
    .a      (top_pcF),
    .rd     (top_InstrF)
);
```

Figure 26. Testbench setup – Output signals and buses from memories for debugging purpose

```
// Debug monitoring
always @(posedge clk) begin

    $fdisplay(fd, "\n// 📌********************************************** [ %0d ] **********************************************📌", cycle_count);
    // $fdisplay(fd, "// PC = 0x%h", top_pcF);
    // $fdisplay(fd, "// Instr = 0x%h (%s)", top_InstrF, instr_mess);

    // Add pipeline stage monitoring
    $fdisplay(fd, "\n// ℹ️---------------------------------- PIPELINE STAGE STATUS ----------------------------------");

    $fdisplay(fd, "// Fetch ____________________________________________________________");
    $fdisplay(fd, "          pcF0 = 0x%h, pcF = 0x%h, InstrF = 0x%h", top_pcF0, top_pcF, top_InstrF);

    $fdisplay(fd, "          PCPlus4F = 0x%h, PCTargetE = 0x%h", top_PCPlus4F, top_PCTargetE);

    $fdisplay(fd, "          PCSrcE = %b, StallF = %b", top_PCSrcE, top_StallF);

    $fdisplay(fd, "// Decode ____________________________________________________________");
    $fdisplay(fd, "                      pcD = 0x%h, InstrD = 0x%h", top_pcD, top_InstrD);

    $fdisplay(fd, "          PCPlus4D = 0x%h", top_PCPlus4D);

    $fdisplay(fd, "                      StallD = %b, FlushD = %b", top_StallD, top_FlushD);

    $fdisplay(fd, "          Read1D = %d, Read2D = %d, ImmExtD = %d", top_Read1D, top_Read2D, top_ImmExtD);

    $fdisplay(fd, "          Rs1D = %d, Rs2D = %d, RdD = %d", top_Rs1D, top_Rs2D, top_RdD);

    $fdisplay(fd, "          RegWriteD = %b, ResultSrcD = %b, MemWriteD = %b, JumpD = %b, BranchD = %b, JumplrD = %b",
        top_RegWriteD, top_ResultSrcD, top_MemWriteD, top_JumpD, top_BranchD, top_JumplrD);

    $fdisplay(fd, "          ALUControlD = %b, ALUSrcD = %b, ImmSrcD = %b, LSTypeD = %b",
        top_ALUControlD, top_ALUSrcD, top_ImmSrcD, top_LSTypeD);

    $fdisplay(fd, "          Ecall = %b, Ebreak = %b", Ecall, Ebreak);

    $fdisplay(fd, "// Execute ____________________________________________________________");
    $fdisplay(fd, "                      pcE = 0x%h", top_pcE);

    $fdisplay(fd, "          PCPlus4E = 0x%h, PCTargetE = 0x%h", top_PCPlus4E, top_PCTargetE);
```

Figure 27. Testbench setup – Setup display of results and pipeline stage statuses

```
always @(posedge clk) begin

    // Monitor memory operations
    $fdisplay(fd, "\n// 💾---------------------------------------- MEMORY ACCESS ----------------------------------------");
    if (top_MemWriteM)
    begin
                $fdisplay(fd, "          Memory Write @ 0x%h:", top_ALUResultM);
                $fdisplay(fd, "                 Data = %d   |   0x%h", top_WriteDataM, top_WriteDataM);

                // Set Test Pass Flag
                if (top_ALUResultM == exp_addr && top_WriteDataM == exp_dat)    test_passed_flag <= 1;
    end

                $fdisplay(fd, "          Memory Read  @ 0x%h:", top_ALUResultM);
                $fdisplay(fd, "                 Data = %d   |   0x%h", top_ReadDataM_sel, top_ReadDataM_sel);


    // Monitor 32 register values
    $fdisplay(fd, "\n// 📂---------------------------------------- REGISTER FILE ----------------------------------------");
    top_regfile_addr <= 0; #1;
    $fdisplay(fd, "                 x0  = %d   |   0x%h", top_regfile_data, top_regfile_data);
    top_regfile_addr <= 1; #1;
    $fdisplay(fd, "                 x1  = %d   |   0x%h", top_regfile_data, top_regfile_data);
    top_regfile_addr <= 2; #1;
    $fdisplay(fd, "                 x2  = %d   |   0x%h", top_regfile_data, top_regfile_data);
    top_regfile_addr <= 3; #1;
    $fdisplay(fd, "                 x3  = %d   |   0x%h", top_regfile_data, top_regfile_data);
    top_regfile_addr <= 4; #1;
    $fdisplay(fd, "                 x4  = %d   |   0x%h", top_regfile_data, top_regfile_data);
    top_regfile_addr <= 5; #1;
    $fdisplay(fd, "                 x5  = %d   |   0x%h", top_regfile_data, top_regfile_data);
    top_regfile_addr <= 6; #1;
```

Figure 28. Testbench setup – Setup display of data memory and register file access

```
// Test sequence
initial begin
    $dumpfile("wave.vcd");
    $dumpvars(0, processor_tb);

    // Reset Test Pass Flag
    test_passed_flag = 0;

    // Set Scoreboard
    exp_addr = 32'd132;
    exp_dat  = 32'hABCDE02E;

    // Open file for writing
    fd = $fopen("sim_rslt.sv", "w");
    if (fd == 0) begin
        $fdisplay(fd, "Error opening file!");
        $finish;
    end

    // Redirect display output to file
    $fdisplay(fd, "// ================================================================================================================================");
    $fdisplay(fd, "                                                 SIMULATION RESULTS");
    $fdisplay(fd, "// ================================================================================================================================\n");

    // Initialize test
    rst = 1;
    @(posedge clk);

    rst = 0;
    cycle_count = 1;

    // Let it run for enough cycles to complete the program
    // repeat (159) @(posedge clk);  // Adjust number as needed
    repeat (50) @(posedge clk);  // Adjust number as needed


    // Add extra delay for register display to complete
    repeat (32) #1;  // Wait for all 32 register reads to complete
    @(posedge clk);  // One more clock cycle for stability
```

Figure 29. Testbench setup – Initial setup to output results to external file

```
    // repeat (159) @(posedge clk);  // Adjust number as needed
    repeat (50) @(posedge clk);  // Adjust number as needed


    // Add extra delay for register display to complete
    repeat (32) #1;  // Wait for all 32 register reads to complete
    @(posedge clk);  // One more clock cycle for stability

    if (test_passed_flag) begin
        $fdisplay(fd, "\n✅ Test PASSED!");
        $fdisplay(fd, "Final memory verification:");
        $fdisplay(fd, "  Address: 0x%h", exp_addr);
        $fdisplay(fd, "  Data:    0x%h", exp_dat);
    end else begin
        $fdisplay(fd, "\n❌ Test FAILED!");
        $fdisplay(fd, "Expected memory result:");
        $fdisplay(fd, "  Address: 0x%h", exp_addr);
        $fdisplay(fd, "  Data:    0x%h", exp_dat);
    end

    $fdisplay(fd, "\nTest Statistics:");
    $fdisplay(fd, "Total cycles: %0d", cycle_count);

    $fclose(fd);

    $finish;
end

// Cycle counter
always @(posedge clk, posedge rst) begin
    if (rst) begin
        cycle_count <= 0;
    end
    else begin
        cycle_count <= cycle_count + 1;
    end
end
```

Figure 30. Testbench setup – Check match cases between output results and expected results

7.3. Simulation Result

The simulation consisted of two main testing segments:

Segment 1: Core Operation Demonstration (Cycles 0 to 32)

- A handcrafted test program of 32 instructions, including arithmetic, logic, branching, jump, and memory operations, was executed.
- The final result (0xABCDE02E) was written correctly to memory address 0x84, validating arithmetic correctness, control flow accuracy, and memory operation functionality.
- Hazards such as RAW (read-after-write), control hazards, and load-use hazards were effectively handled via forwarding, flushing, and stalls.

- The program flow and signal states were validated through GTKWave to confirm correct stage-by-stage execution.

Segment 2: Full RV32I Instruction Coverage (Cycle 0 to 160)

- An additional 129 instructions were inserted after the main program to test all **39 instruction types of the RV32I base ISA** (Figure 31).
- **Expected behaviors**:
    - Each instruction was executed in isolation with dedicated operands and registers (from x10 to x31) to avoid overwriting main program values.
    - Instructions with dependencies triggered forwarding or stalls as expected.
    - Memory read/write instructions were validated by monitoring the corresponding memory addresses and write-back data.
    - Control instructions such as conditional branches (beq, blt, etc.) and unconditional jumps (jal, jalr) showed correct branching behavior and pipeline flushes.
- **Outcome**: All instructions executed successfully with no hazard-related failures. The final instruction loop (beq x2, x2, done) was reached, confirming program termination logic.

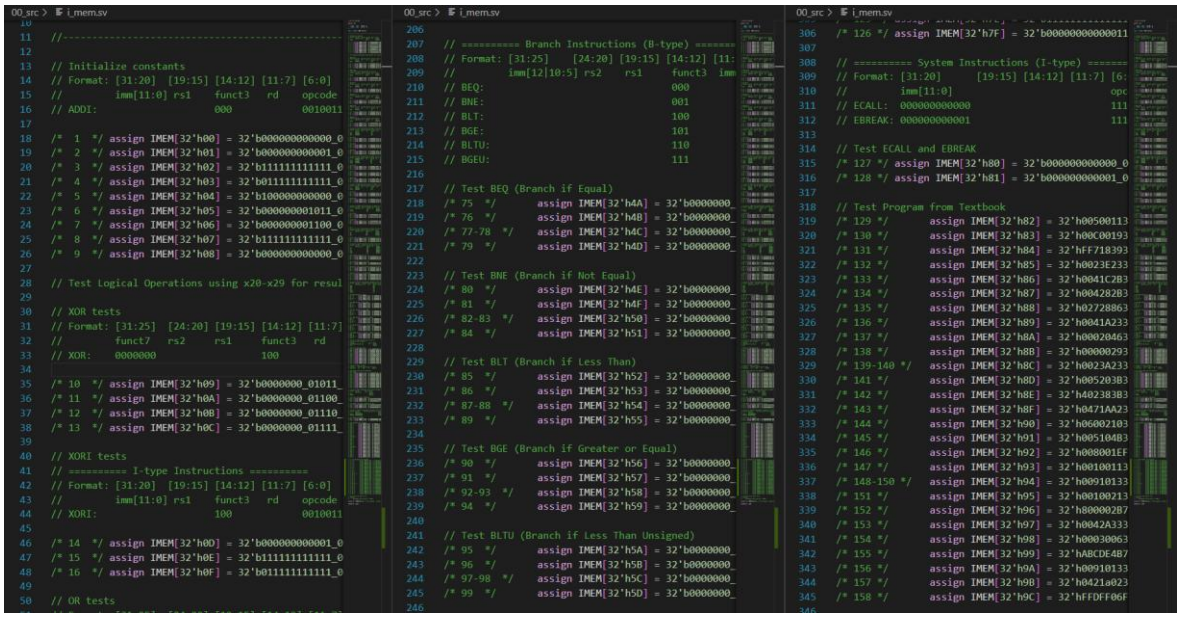

Figure 31. 156 instructions covering all 39 types of RV32I (manually feed to IMEM)

8. SYNTHESIS

8.1. Synthesis Tool and Target

The processor design was synthesized using **Intel Quartus Prime**, a widely used FPGA development environment. The synthesis targeted an **Intel Cyclone IV 5CSXFC6D6F31C6** FPGA for prototyping purposes. All modules, from the top-level processor to individual submodules (e.g., ALU, decoder, hazard unit), were described in synthesizable SystemVerilog HDL and successfully synthesized without warnings or errors related to logic or unconnected nets.

8.2. Netlist Hierarchy and Structure

The synthesis process translated the RTL description into gate-level netlists, which provide a structural view of the design after logic optimization and technology mapping. Below is a summary of the netlist organization, categorized by design abstraction and function:

Top-Level Netlist:

The top-level netlist represents the processor system with external interfaces for Instruction Memory and Data Memory. These memory blocks are modular and designed for replacement with physical memory (e.g., BRAM or SRAM) on FPGA boards (Figure 32).

- Inputs: clk, rst
- Interfaces:
    - Instruction Memory: imem_rd, imem_addr
    - Data Memory: dmem_rd, dmem_wr, dmem_addr, dmem_wdata, dmem_rdata
- Outputs: ecall, ebreak, write_back_result for debug monitoring

Processor Core Netlist:

The processor core consists of three tightly coupled subsystems:

- Datapath
- Controller
- Hazard Unit

Each of these is modular and implemented hierarchically for synthesis and reuse (Figure 33).

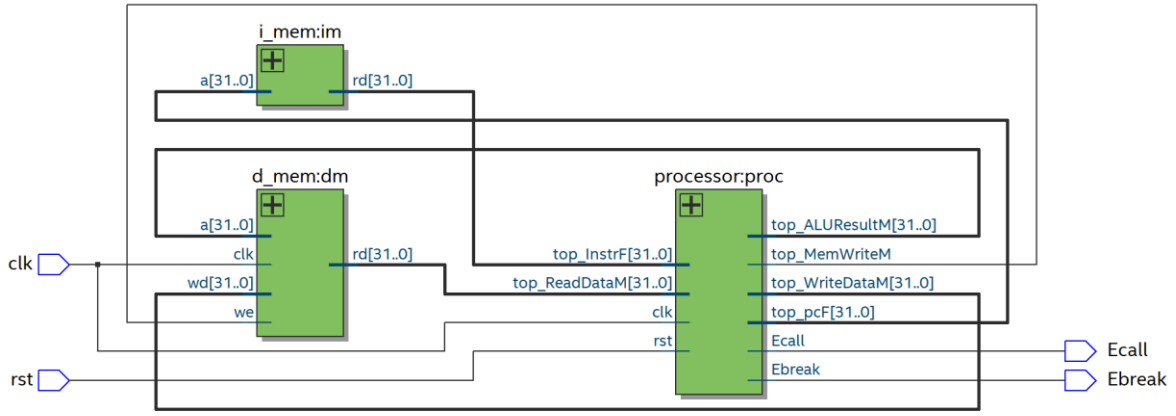

Figure 32. Netlist View of the Processor with External Instruction and Data Memory Interfaces

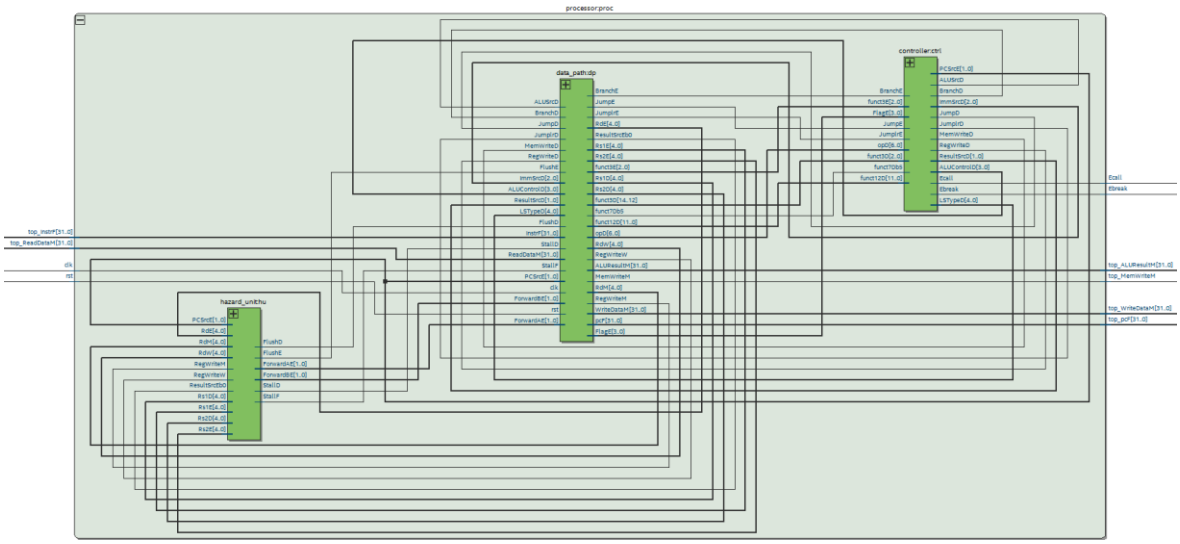

Figure 33. Netlist View of the Processor

8.3. Synthesized Submodules

Each critical submodule of the processor was synthesized and verified separately to ensure correct structural logic. The key modules include:

Control Unit

- Divided into **Main Decoder**, **ALU Decoder**, **Branch Unit (BRU)**, **Load-Store Unit (LSU)**, and **Environment Unit (EU)**
- Generates control signals based on opcode, funct3, and funct7
- Synthesized logic includes decoders, comparators, multiplexers, and control signal encoders (Figure 34)

Main Decoder

- Decodes opcode to determine instruction format and core control signals such as RegWrite, ImmSrc, MemWrite, ALUSrc, ResultSrc, Branch, Jump, Jumplr, ALUOp
- Synthesized as a combinational logic block with minimal area (Figure 35)

ALU Decoder

- Based on ALUOp, funct3, and funct7
- Outputs a 4-bit ALUControl signal to select ALU operation
- Covers all RV32I ALU-based instructions (Figure 36)

BRU (Branch Unit)

- Evaluates branch conditions based on flags (Z, C, N, V) and funct3
- Outputs PCSrc signal for determining next PC
- Synthesized with compact control logic and priority encoding (Figure 37)

LSU (Load-Store Unit)

- Determines memory access type (byte, half, word) and whether signed/unsigned based on funct3
- Outputs LSType used by datapath and memory interface logic (Figure 38)

EU (Environment Unit)

- Detects environment instructions like ecall and ebreak
- Enables software interrupt or debugger hooks (Figure 39)

Hazard Unit

- Detects data hazards (RAW) and control hazards
- Generates:
    - ForwardAE, ForwardBE (for forwarding)
    - StallF, StallD, FlushD, FlushE (for stalling and flushing)
- Implemented using comparators, control gates, and compact FSM logic
- Critical to ensure pipeline correctness and was verified via waveforms
- Uses 32-bit operands; outputs result and 5-bit flag for condition checks (Figure 40)

Datapath

- Connects pipeline registers, control signals, ALU, register file, memory access logic, PC update logic (Figure 41)
- Synthesized as a hierarchical module including:
    - PC logic
    - Instruction and data pipeline registers
    - Register file (multi-port)
    - ALU + forwarding logic
    - Immediate extender
    - PC muxes for branching and jumping

ALU

- Performs 16 arithmetic and logical operations as dictated by the 4-bit ALUControl (Figure 32)

Instruction Memory (IMEM) and Data Memory (DMEM)

- Modeled as dual-port RAMs in synthesis
- Use of imem.sv and dmem.sv modules enables read and write access during pipeline operation (Figure 43 and Figure 44)

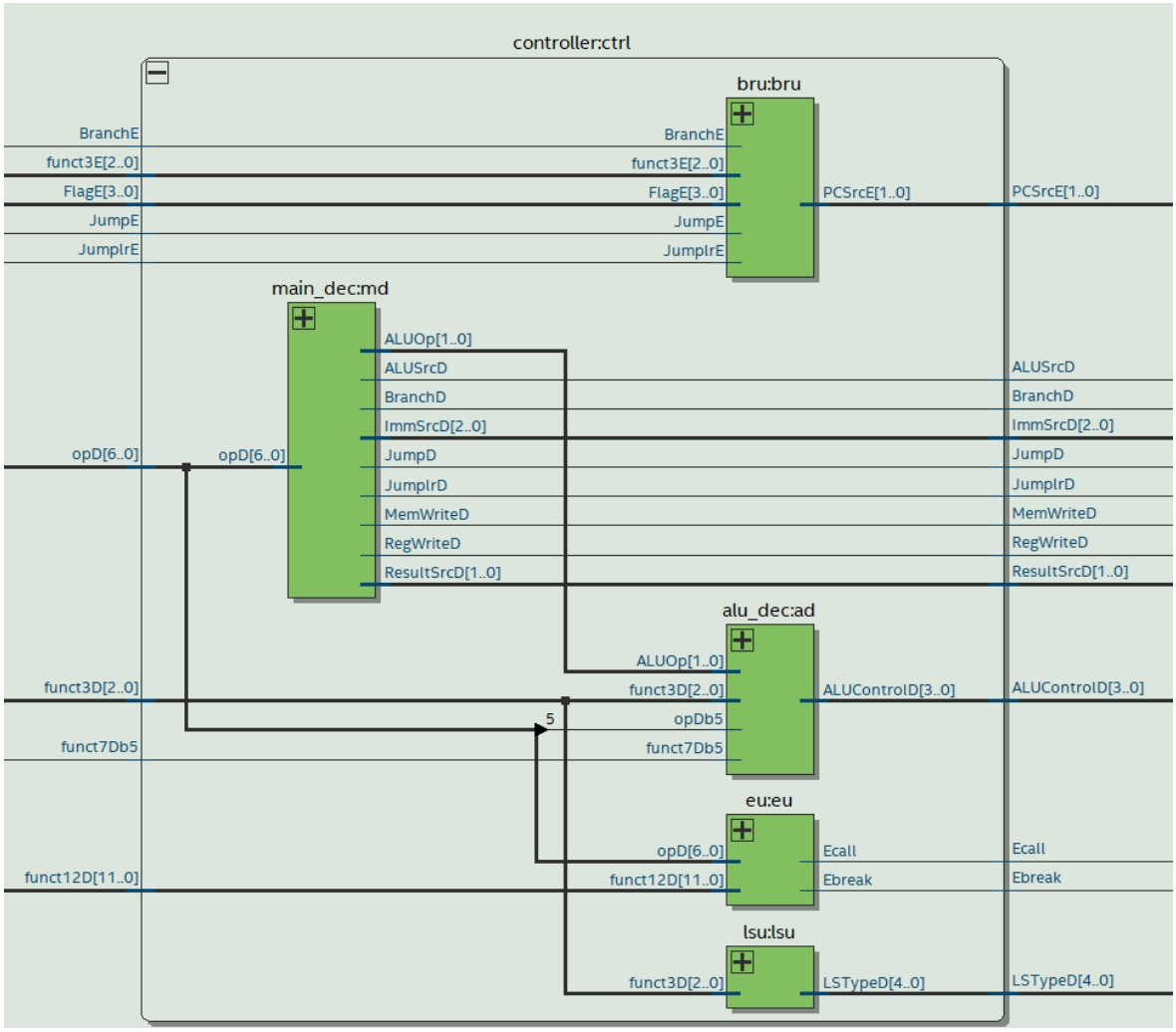

Figure 34. Netlist View of the Control Unit

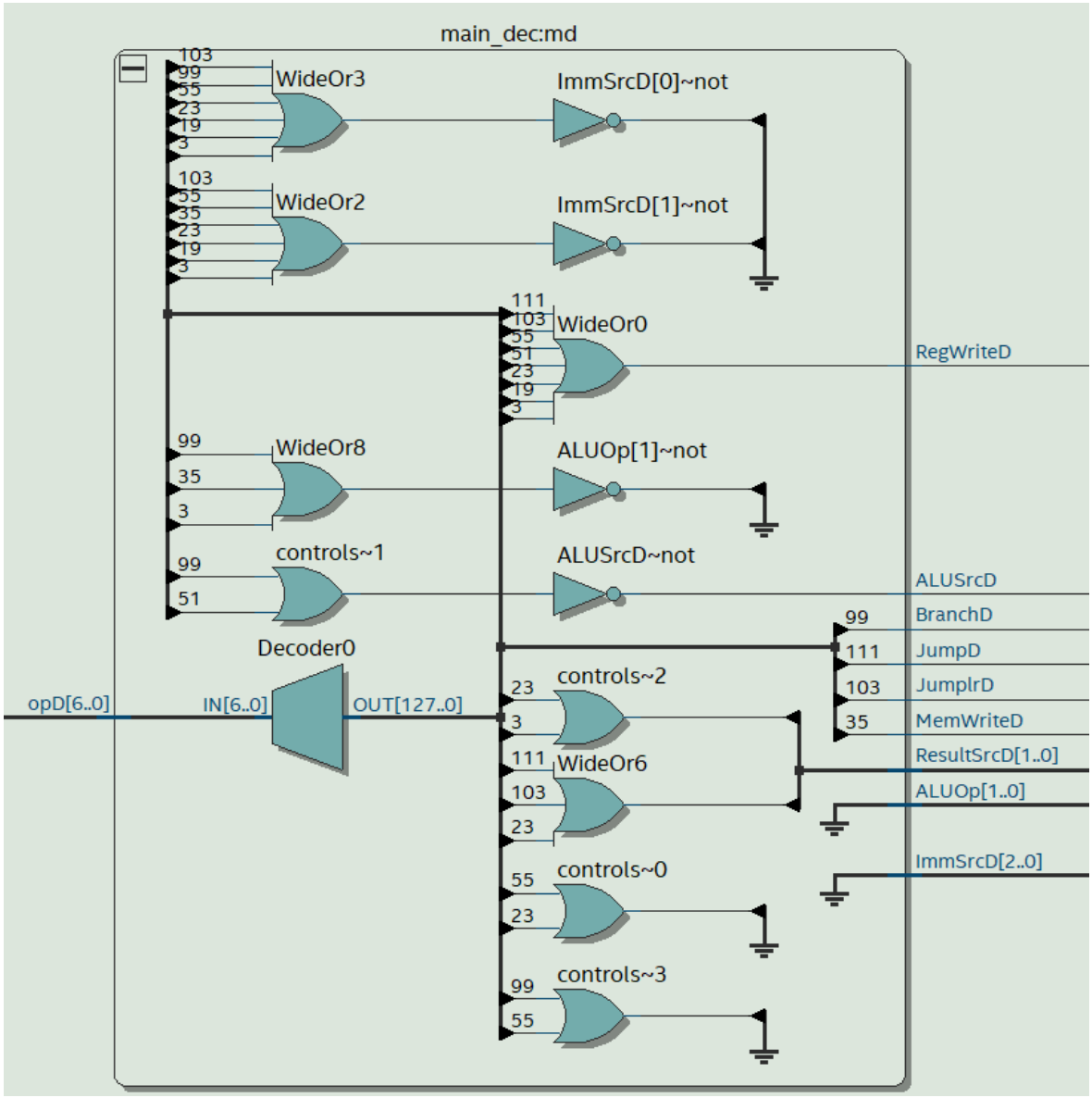

Figure 35. Netlist View of the Main Decoder

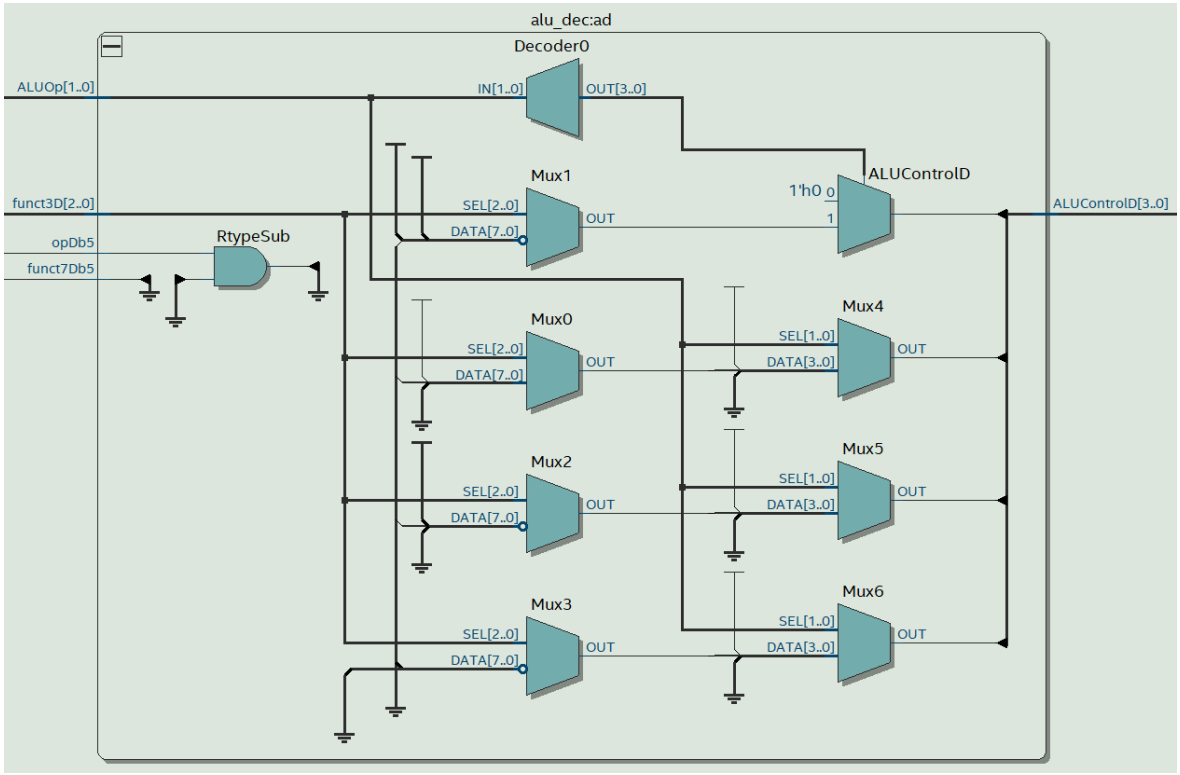

Figure 36. Netlist View of the ALU Decoder

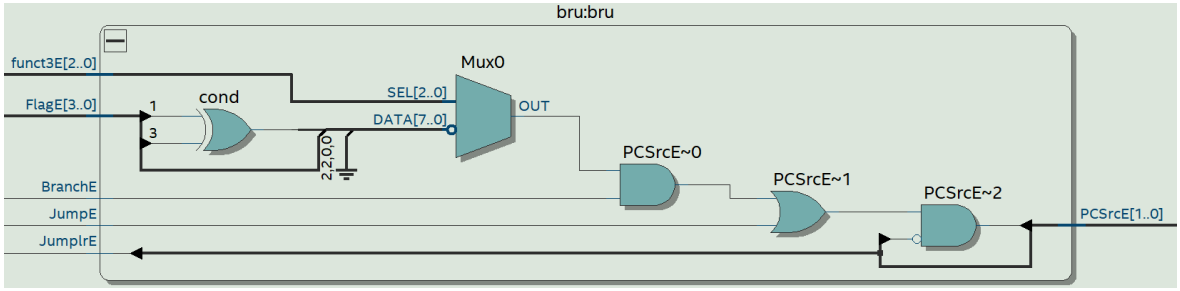

Figure 37. Netlist View of the Branch Unit (BRU)

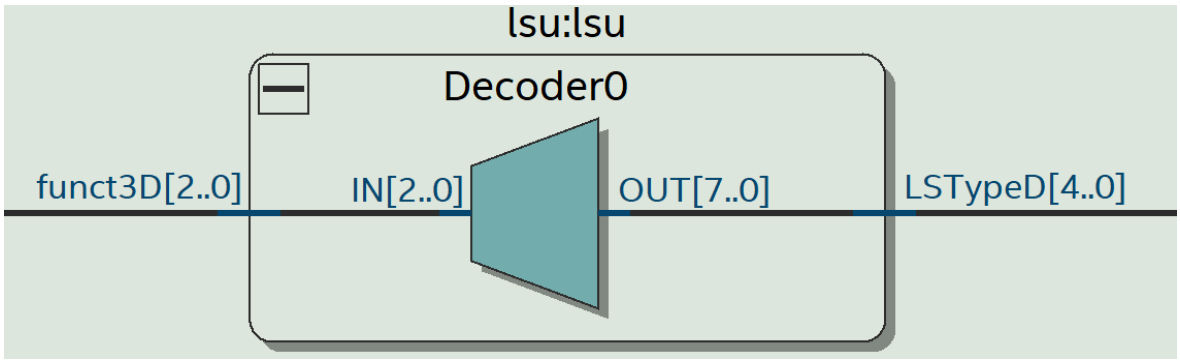

Figure 38. Netlist View of the Load – Store Unit (LSU)

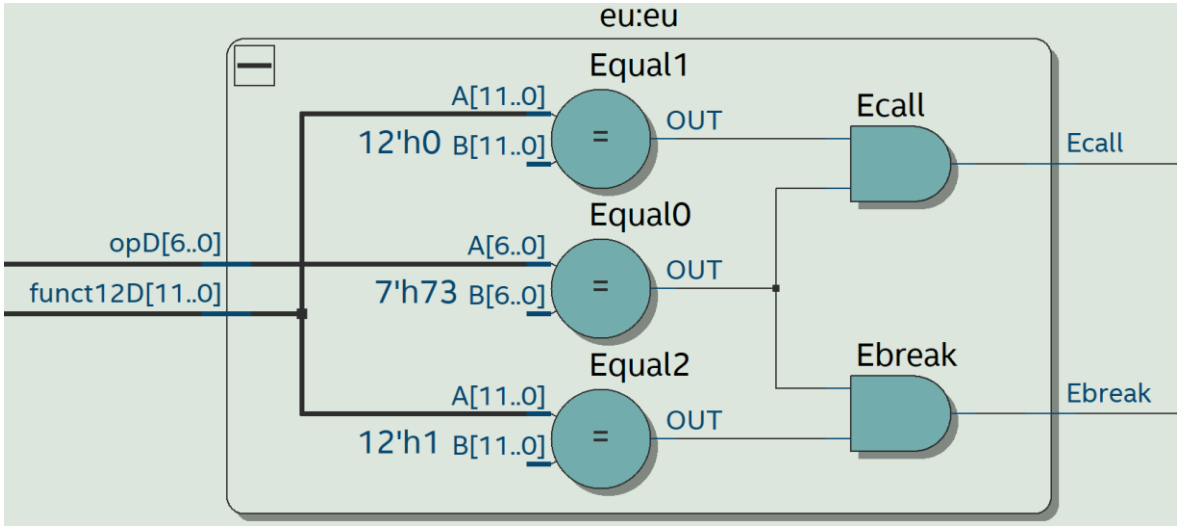

Figure 39. Netlist View of the Environment Unit (EU)

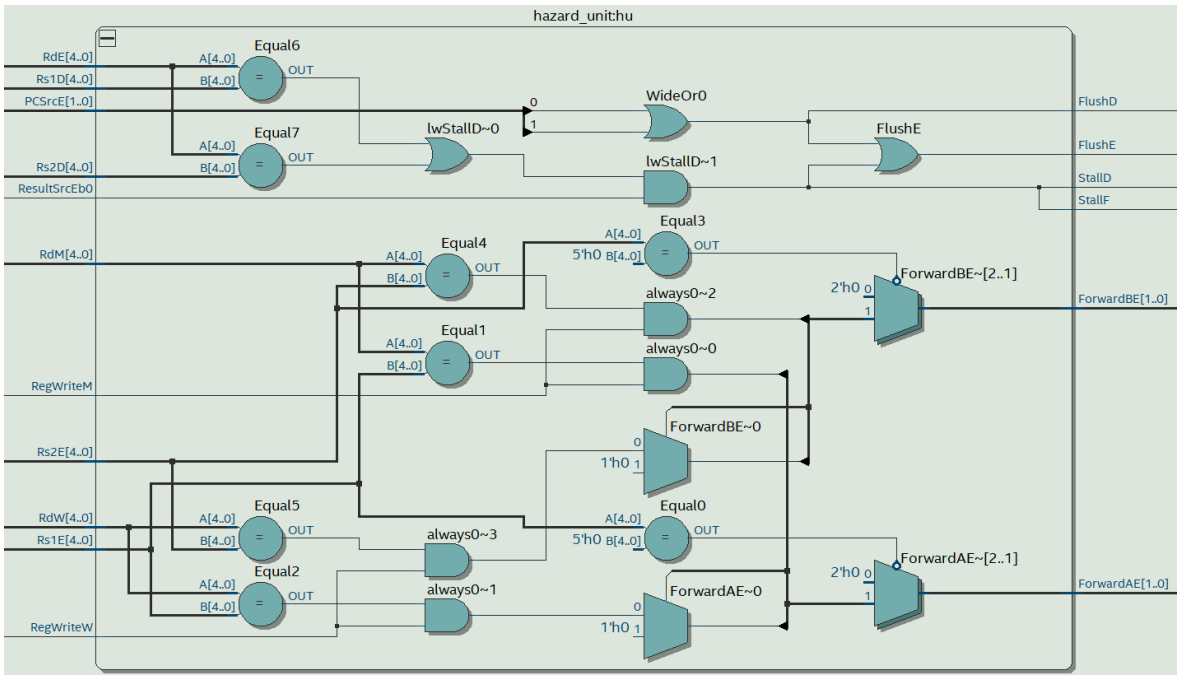

Figure 40. Netlist View of the Hazard Unit (HU)

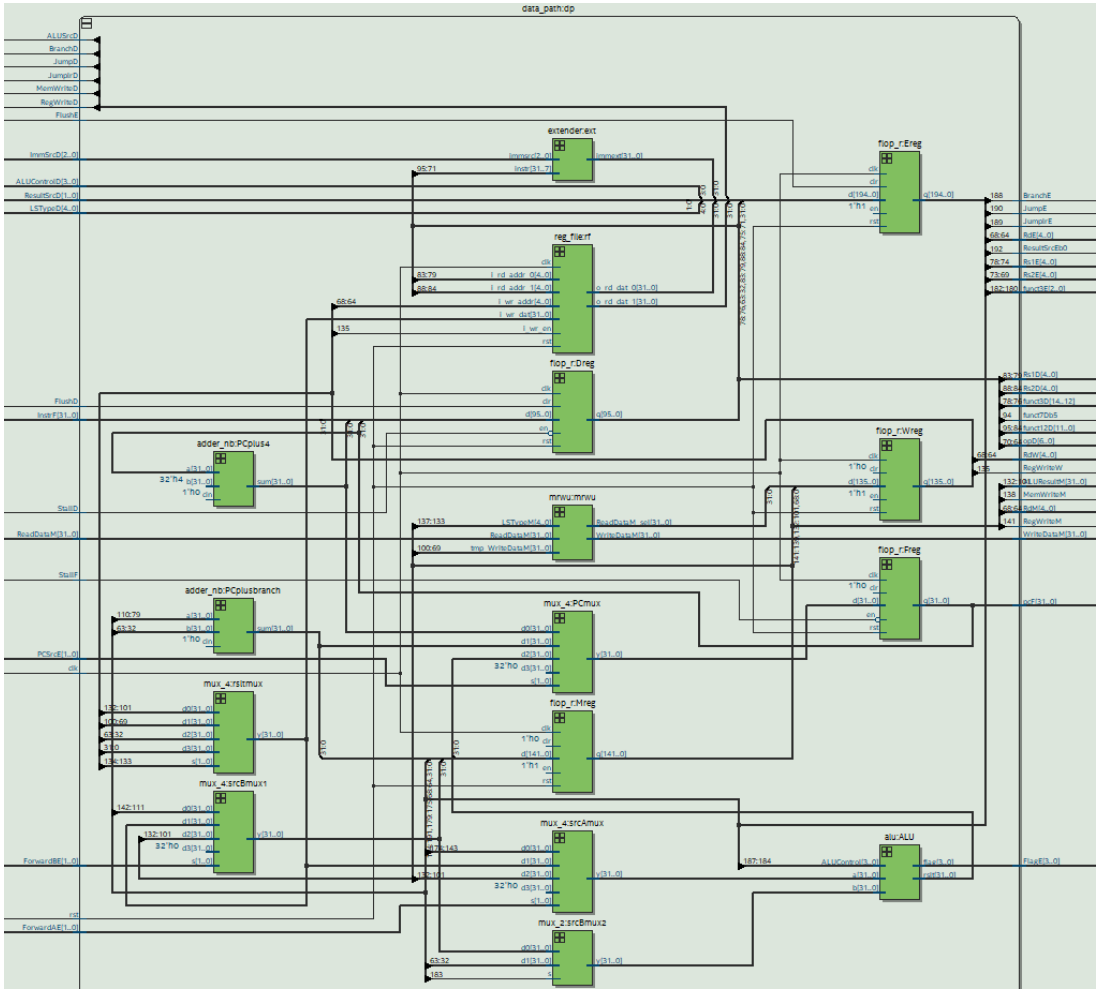

Figure 41. Overall Netlist View of Data Path

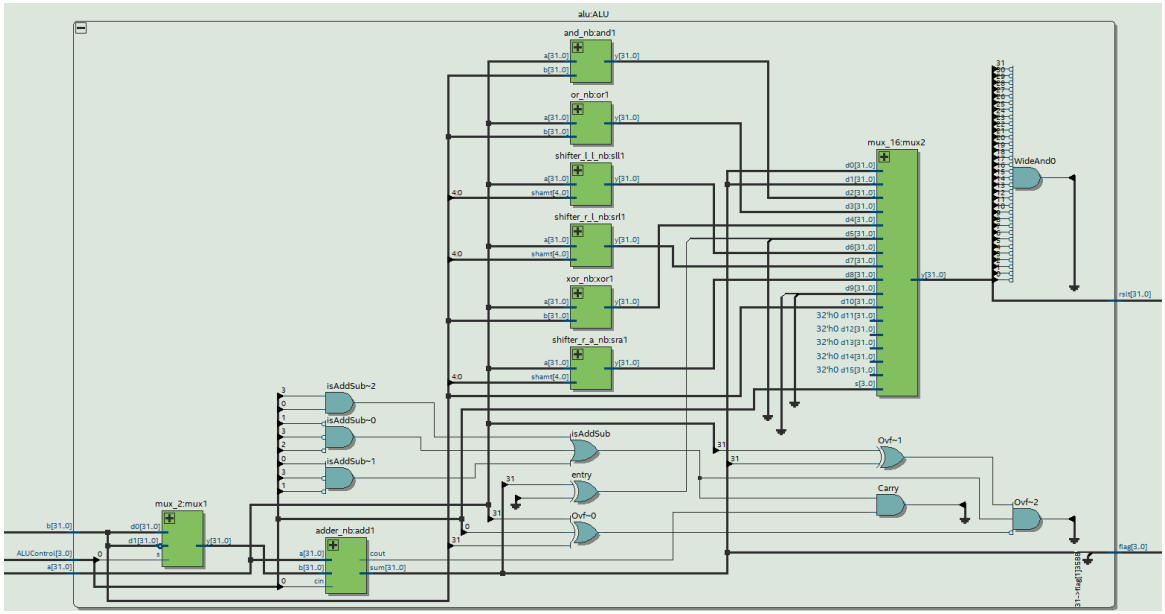

Figure 42. Netlist View of Arithmetic Logic Unit (ALU)

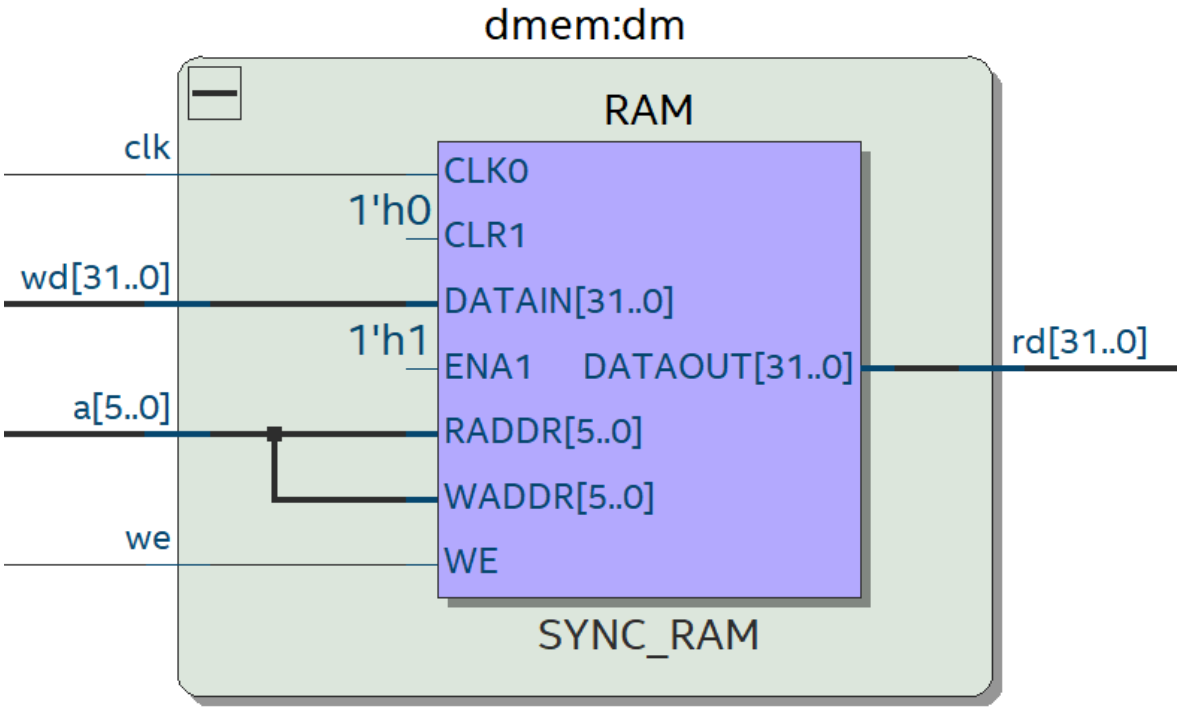

Figure 43. Netlist View of Data Memory

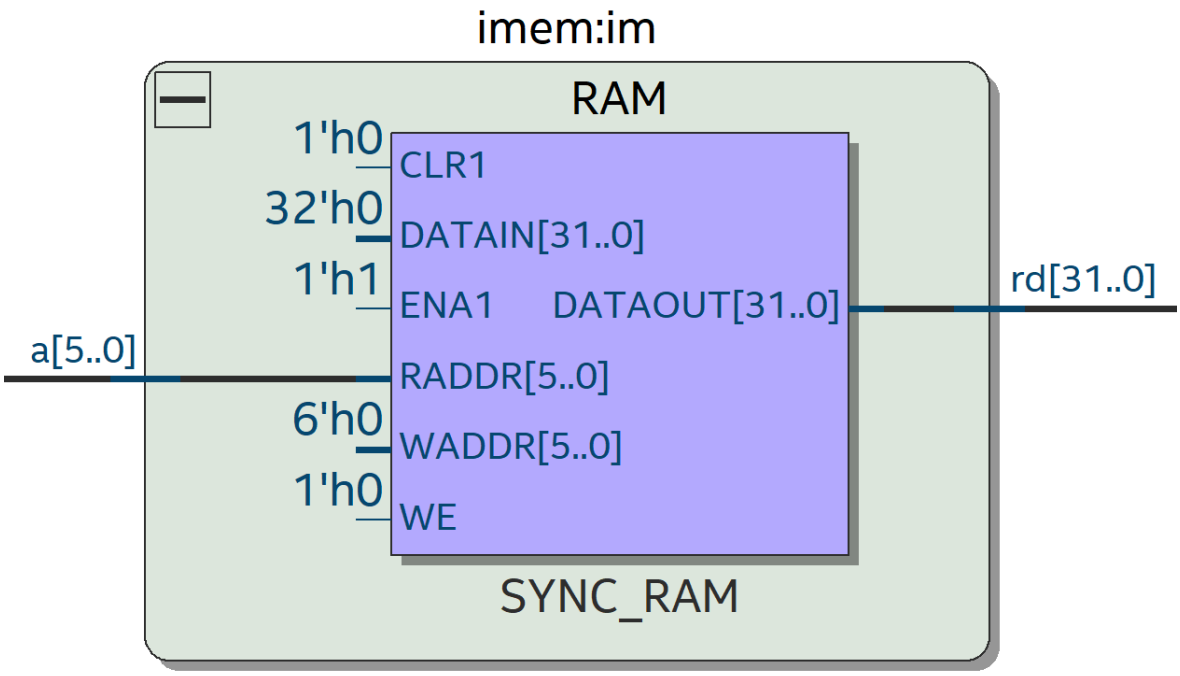

Figure 44. Netlist View of Instruction Memory

9. WORKING DIRECTORY

The complete design source code, testbenches, simulation scripts, synthesis files, and documentation for the 5-stage pipelined RV32I processor are organized and version-controlled within a public GitLab repository:

Repository Link: https://gitlab.com/duyngoc131004/CapstoneProject1

Directory Structure:

- 00_doc/: Design report, waveform images, reference documents.
- 01_rtl/: Verilog RTL source files for processor modules: controller, datapath, hazard unit, submodules, memories.
- 01_bench/: Unit and integration testbenches (*_tb.sv) for modules and top-level processor (processor_tb.sv).
- 10_sim/verilator/: Simulation results, logs, test output files (sim_rslt.sv).
- 20_quartus/: Quartus Prime project files, netlist views, synthesis results (including top-Resource Utilization by Entity.rpt).
- README.md: Overview of project purpose, build instructions, and usage guide.

Access to the repository is open to reviewers and collaborators. It includes detailed commit history, README documentation, and simulation instructions.

10. CONCLUSION AND FUTURE VISION

10.1. Conclusion

In this project, we successfully designed, implemented, and verified a **5-stage pipelined RV32I processor** based on the RISC-V instruction set architecture. The design includes:

- A complete datapath and control path that support **all 39 base RV32I instructions**.
- Functional units such as an ALU, branch unit (BRU), load/store unit (LSU), and extender.
- A forwarding and hazard detection unit to resolve **data and control hazards**.
- Synthesis to an FPGA-compatible netlist using **Intel Quartus Prime** with verified resource usage.
- Full functional simulation of both unit modules and the entire processor using **Verilator**.

The processor passes all defined test cases, including cases involving stalling, forwarding, and branching. Waveform analysis confirms correct instruction sequencing and pipeline interactions.

10.2. Future Vision

To expand on this foundational work, several directions are proposed:

1. *Performance Enhancements:*
    - Add branch prediction mechanisms to reduce the control hazard penalty.
    - Implement instruction and data caching.
2. *Feature Extensions:*
    - Extend to support **RV32IM** (multiplication/division).
    - Include **CSR instructions** for privilege modes and trap handling.
3. *Tooling Improvements:*
    - Automate the testbench and waveform analysis pipeline.
    - Integrate support for assembling standard RISC-V programs into binary/memory format.
4. *Educational Use:*
    - Package the project into a teaching kit with visual RTL viewers and simulation guides for learning pipelined architecture.

5. *Hardware Deployment:*
    - Port the synthesized design to a real FPGA development board and run programs using UART or GPIO interfaces.

This project establishes a robust foundation for learning and experimenting with open instruction set architectures, paving the way for advanced processor design research and practical embedded systems development.

11. REFERENCES

[1] D. M. Harris and S. L. Harris, “Digital Design and Computer Architecture”, 2nd ed., Amsterdam, Netherlands: Morgan Kaufmann, 2012.

[2] D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th ed., Burlington, MA: Morgan Kaufmann, 2014.

[3] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA”, Version 20250508, RISC-V Foundation, 2025. [Online]. Available: https://riscv.org/technical/specifications/