**VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY**

**UNIVERSITY OF TECHNOLOGY**

**FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING**

**DEPARTMENT OF ELECTRONICS ENGINEERING**

**------o0o-----**

A blue and white logo

Description automatically generated

**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 2025PROJECT SUMMARY**

**TABLE OF CONTENTS**

[**1.** **INTRODUCTION** 3](#_Toc199382401)

[***1.1.*** ***Overview*** 3](#_Toc199382402)

[***1.2.*** ***Project Objectives*** 3](#_Toc199382403)

[**2.** **DESIGN THEORY AND INTEREST** 5](#_Toc199382404)

[***2.1.*** ***RV32I Instruction Set*** 5](#_Toc199382405)

[***2.2.*** ***Microarchitectures*** 5](#_Toc199382406)

[***2.3.*** ***Pipeline Stages*** 6](#_Toc199382407)

[***2.4.*** ***Pipelined Data Path*** 7](#_Toc199382408)

[***2.5.*** ***Pipelined Control Unit*** 10](#_Toc199382409)

[***2.6.*** ***Hazards in Pipelining*** 10](#_Toc199382410)

[*2.6.1.* *Data Hazards and RAW Dependencies* 10](#_Toc199382411)

[*2.6.2.* *Forwarding (Bypassing)* 11](#_Toc199382412)

[*2.6.3.* *Stalling for Load-Use Hazards* 12](#_Toc199382413)

[*2.6.4.* *Control Hazards and Branch Misprediction* 13](#_Toc199382414)

[**3.** **DESIGN SPECIFICATIONS** 15](#_Toc199382415)

[***3.1.*** ***Design Specifications*** 15](#_Toc199382416)

[***3.2.*** ***Design Analysis*** 15](#_Toc199382417)

[**4.** **BLOCK DIAGRAMS** 17](#_Toc199382418)

[***4.1.*** ***High-Level Block Diagram*** 17](#_Toc199382419)

[***4.2.*** ***Detailed Block Diagram*** 18](#_Toc199382420)

1. **INTRODUCTION**
   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. ***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.

1. **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.

* 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.

A white sheet with numbers and lines

AI-generated content may be incorrect.

* 1. ***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].

* 1. ***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 7.47(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 7.47(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].

A diagram of a computer

AI-generated content may be incorrect.

Figure 7.48 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.

A diagram of a diagram

AI-generated content may be incorrect.

* 1. ***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 7.49, 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 7.49(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.

A diagram of a computer

AI-generated content may be incorrect.

Figure 7.50 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, covered in Section 2.7.

A diagram of a pipe

AI-generated content may be incorrect.

* 1. ***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 funct75 fields of the instruction in the Decode stage to produce the control signals. These control signals must be pipelined along with the data so that they remain synchronized with the instruction. The entire pipelined processor with control is shown in Figure 7.51. RegWrite must be pipelined into the Writeback stage before it feeds back to the register file, just as Rd was pipelined in Figure 7.50. In addition to R-type ALU instructions, lw, sw, and beq, this pipelined processor also supports jal and I-type ALU instructions.

A diagram of a computer

AI-generated content may be incorrect.

* 1. ***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.

* + 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.

A diagram of a diagram

AI-generated content may be incorrect.

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:

* + 1. *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.

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.

A diagram of a pipeline

AI-generated content may be incorrect.

* + 1. *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 isn't ready in time for the next cycle.

A diagram of a pipeline

AI-generated content may be incorrect.

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.

The stall condition is computed as:

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

When lwStall is true:

* StallF = 1
* StallD = 1
* FlushE = 1

A diagram of a pipeline

AI-generated content may be incorrect.

* + 1. *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.

A diagram of a flowchart

AI-generated content may be incorrect.

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.

1. **DESIGN SPECIFICATIONS**
   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)
  1. ***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.

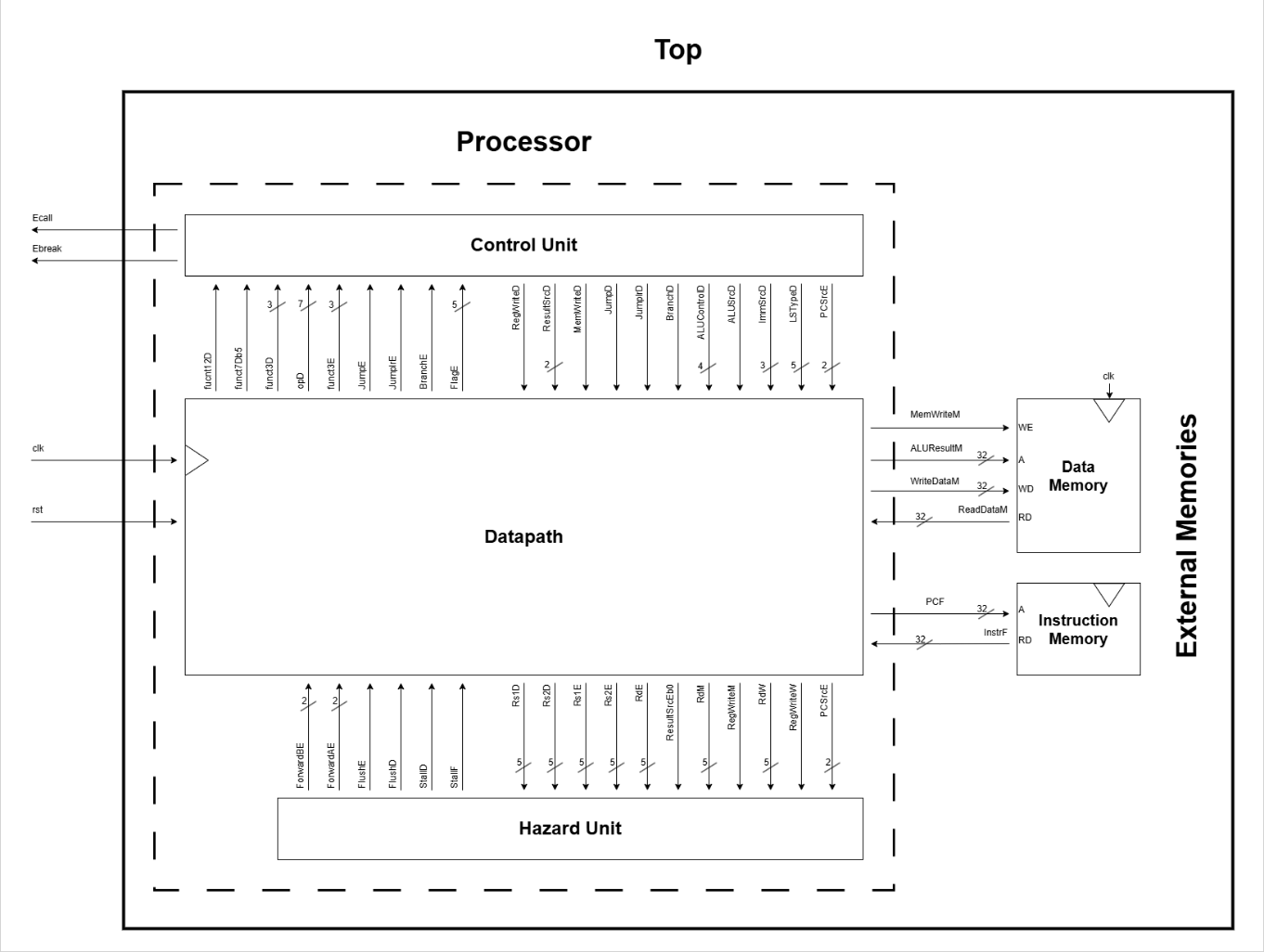
*Conclusion:*  
The pipelined architecture was chosen as the optimal approach due to its superior throughput and relevance to real-world processor design. Despite added complexity in control and hazard resolution, it provides a realistic and efficient processor model.

1. **BLOCK DIAGRAMS**
   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 2. 5-Stage RV32I Pipelined Processor with External Instruction and Data Memory Interfaces**

* 1. ***Detailed Block Diagram***

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

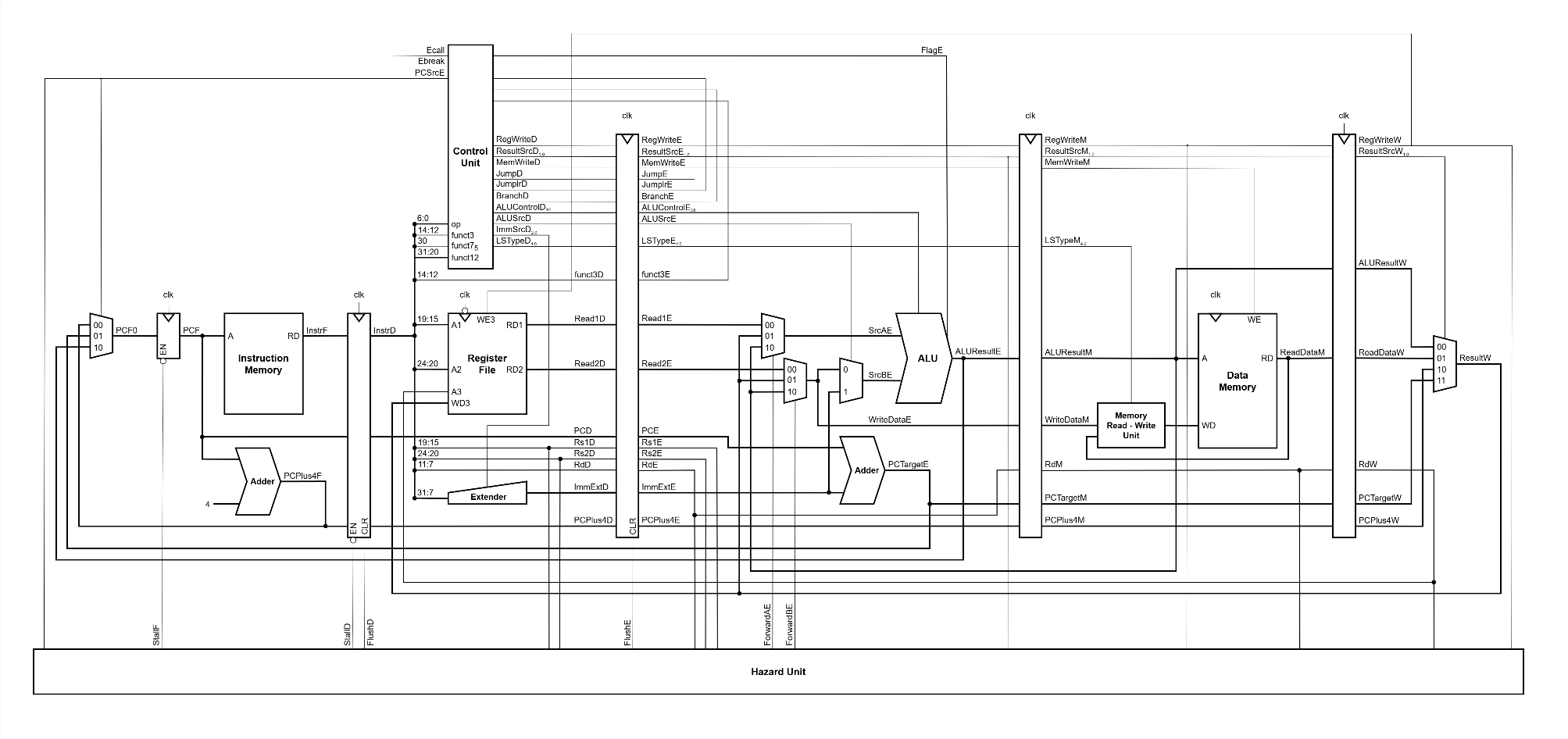
* *Datapath:* Implements the functional units (e.g., ALU, register file, extender) and handles the movement of data between pipeline stages.
* *Control Unit:* Includes multiple sub-decoders:
  + Main Decoder generates high-level control signals.
  + ALU Decoder determines ALU operation.
  + Branch Unit (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.

A diagram of a mathematical unit

AI-generated content may be incorrect.

**Figure 1. Arithmetic Logic Unit of 5-Stage RV32I Pipelined Processor**



**Figure 1. Datapath of 5-Stage RV32I Pipelined Processor with Full Hazard Detection, Forwarding, and Control Logic**

A diagram of a system

AI-generated content may be incorrect.

**Figure 1. Control Unit of 5-Stage RV32I Pipelined Processor**

1. **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 |

**A diagram of a computer

AI-generated content may be incorrect.**

**Figure 3. Netlist View of the Processor with External Instruction and Data Memory Interfaces**

**A computer screen shot of a diagram

AI-generated content may be incorrect.**

**Figure 4. Netlist View of the Processor**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Figure 7. Netlist View of the Control Unit**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Figure 7. Netlist View of the Main Decoder**

**A diagram of a computer

AI-generated content may be incorrect.**

**Figure 7. Netlist View of the ALU Decoder**

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 8. Netlist View of the Branch Unit (BRU)**

A diagram of a decoder

AI-generated content may be incorrect.

**Figure 8. Netlist View of the Load – Store Unit (LSU)**

A diagram of a number equal

AI-generated content may be incorrect.

**Figure 8. Netlist View of the Environment Unit (EU)**

**A computer screen shot of a diagram

AI-generated content may be incorrect.**

**Figure 8. Netlist View of the Hazard Unit (HU)**

**A computer screen shot of a diagram

AI-generated content may be incorrect.**

**Figure 8. Overall Netlist View of Data Path**

**A computer diagram of a computer

AI-generated content may be incorrect.**

**Figure 8. Netlist View of Arithmetic Logic Unit (ALU)**

**A computer screen shot of a computer

AI-generated content may be incorrect.**

**Figure 5. Netlist View of Data Memory**

**A computer program with text

AI-generated content may be incorrect.**

**Figure 6. Netlist View of Instruction Memory**

**Table 2. ImmSrc Encoding**

|  |  |  |
| --- | --- | --- |
| **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 | I-type ecall, ebreak |

**A table with numbers and symbols

AI-generated content may be incorrect.A table with numbers and symbols

AI-generated content may be incorrect.**

**A table with numbers and text

AI-generated content may be incorrect.A table with numbers and letters

AI-generated content may be incorrect.**

**Table 4. ALU Decoder Truth Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ALU-Op** | **funct3** | **{op5, funct75}** | **ALU-Control** | **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. 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 |

**A table with numbers and symbols

AI-generated content may be incorrect.A table with numbers and symbols

AI-generated content may be incorrect.A table with numbers and numbers

AI-generated content may be incorrect.A table with numbers and symbols

AI-generated content may be incorrect.**

**Table 5. BRU Truth Table**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Branch** | **Jump** | **Jumplr** | **funct3** | **Flag**  **= {V, C, N, Z}** | **PCSrc1** | **PCSrc0** | **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 6. EU Truth Table**

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