TROY UNIVERSITY

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF MATHEMATICS AND INFORMATICS



FINAL GROUP PROJECT REPORT

Project Topic: Design and Simulation of a Pipelined CPU Using Verilog

Supervisor: Assoc. Prof. Dr. Nguyen Dinh Han

Course: CS-3365 - Introduction to Computer Organization and Architecture

Group: [GROUP]

Students:

1. [Student 1] – [ID 1]
2. [Student 2] – [ID 2]
3. [Student 3] – [ID 3]
4. [Student 4] – [ID 4]
5. [Student 5] – [ID 5]

HANOI, January 2026

Troy University Resources

Overview

This report documents our group's process of designing and developing a simple pipelined CPU using Verilog. The project emphasizes clarity over complexity, aiming to illuminate the essential ideas behind pipelining a compact, working implementation. This report follows the design process of the CPU. It details the scope and overall design, the programming model and instruction set, the pipeline architecture and the hazard handling system of the CPU.

The design uses a five-stage RISC-style pipeline (IF, ID, EX, MEM, WB) with minimal instruction subset. By separating instruction and data memories, the design avoids structural hazards in the memory system and allows the pipeline to run at a good speed under ideal conditions. This includes standard mechanisms for managing data and control hazards such as operand forwarding, a one-cycle stall for load-use scenarios, and a simple not-taken branch policy with pipeline flushing on taken branches.

The CPU's structure is decomposed into reusable building blocks: register file, ALU, immediate generator, control unit, hazard/forwarding units, pipeline registers, and memory models. The data is tested and visualized through GTKWave to ensure functionality and correctness.

Table Of Contents

Overview **2**

Table Of Contents **3**

Summary of Contributions **9**

1. [Student 1] – [ID 1] 9

2. [Student 2] – [ID 2] 9

3. [Student 3] – [ID 3] 10

4. [Student 4] – [ID 4] 10

5. [Student 5] – [ID 5] 10

Chapter I. Introduction **11**

1.1. Objectives and Scope 11

1.2. Summary of the program's features 12

1.3. Overview of CPU Pipeline 13

Chapter II. Methodology: Architectural Specification and Design **15**

2.1. Programming Model 15

2.1.1. Overview on Architecture State 15

2.1.2. Instruction Semantics Model 15

2.1.3. Data Types, Signedness, and Arithmetic 16

2.1.4. Memory and Addressing Rules 16

2.1.5. Control Flow Model 16

2.1.6. Reset, Initialization, and Program Loading 17

2.1.7. Programmer-Visible Timing and Ordering 17

2.1.8. Conventions and Recommended Usage 18

2.1.9. Conclusion 18

2.2. Minimal Instruction Set 19

2.2.1. Instruction Categories 19

2.2.2. R-type ALU Operations 20

2.2.3. Immediate Arithmetic (I-type) 21

2.2.4. Memory Operations 22

2.2.5. Control-Flow Instructions 23

Troy University Resources

2.2.6. Encoding Summary (Illustrative) 24
2.2.7. Pseudo-Instructions and Idioms 25
2.2.8. Pipeline Interactions and Hazards (By Instruction) 25
2.2.9. Examples 26
2.2.10. Extensibility Notes 27
2.3. Instruction Encoding and Formats 27
2.3.1. Field Conventions 28
2.3.2. Format Definitions 28
2.3.3. Opcode and Function Assignments (Illustrative) . 30
2.3.4. Immediate Generation and Target Calculation . . 30
2.3.5. Worked Encoding Examples 31
2.4. Sample Test Program 32
2.4.1. Problem Statement and Setup 32
2.4.2. Assembly Listing (with Comments) 33
2.4.3. Machine Code (Hex) and Addresses 33
2.4.4. Data Image and Expected Result 34
2.4.5. Hazard and Performance Notes 35
2.4.6. Testbench Pass/Fail Criterion 35

Chapter III. Methodology: Five-Stage Pipeline Microarchitecture 36
3.1. Pipeline Stages Diagram 36
3.1.1. Overview of the Five-Stage Pipeline 36
3.1.2. IF — Instruction Fetch 36
3.1.3. ID — Decode, Register Read, Immediate Generation 37
3.1.4. EX — Execute, Address Calculation, Branch Resolution 38
3.1.5. MEM — Data Memory Access 39
3.1.6. WB — Write Back 40
3.1.7. Pipeline Register Contents 40
3.1.8. Control Signal Lifecycle (Where Generated vs. Consumed) 41
3.1.9. Forwarding Network and Hazard Resolution (EX Focus) 41
3.1.10. Timing Sketch (Fill, Stall, Flush) 42

3.1.11. Signal Interfaces (Stage I/O Summary) 43
3.1.12. Extensibility Notes 44
3.2. Pipeline Registers 44
3.2.1. Design Goals and Policies 44
3.2.2. IF/ID Register (Fetch → Decode) 45
3.2.3. ID/EX Register (Decode → Execute) 45
3.2.4. EX/MEM Register (Execute → Memory) 46
3.2.5. MEM/WB Register (Memory → Write-Back) . . 46
3.2.6. Flush/Stall Precedence and Update Algorithm . . 47
3.2.7. NOP Encoding and Reset State 47
3.2.8. Implementation Sketch 47
3.3. Data Path and Control Path 48
3.3.1. Datapath Overview 48
3.3.2. Register File Semantics 49
3.3.3. Immediate Generation 49
3.3.4. ALU and ALUOp Mapping 50
3.3.5. Next-PC Logic 50
3.3.6. Forwarding Network 50
3.3.7. Hazard Unit Interactions 51
3.3.8. Memory Interfaces 51
3.3.9. Control Bundle Definition and Propagation . . . 52
3.4. Ideal Performance and Timing Constraints 52
3.4.1. Ideal CPI and Pipeline Boundaries 52
3.4.2. CPI Model with Hazard Penalties 52
3.4.3. Stage Critical Paths and f_max 53
3.4.4. Clocking and Reset Assumptions 53
3.4.5. Timing Closure Guidance 54
3.4.6. Memory Timing Variants and Impact 54
3.4.7. Power/Performance Trade-offs 54
3.4.8. Verification Hooks for Timing Behavior 54

Chapter IV. Methodology: Hazard Handling and Control 56
4.1. Data Hazards and Forwarding 56
4.1.1. RAW Examples 56
4.1.2. Forwarding Sources and Priority 57

Troy University Resources

4.1.3. Why WAR and WAW Don't Occur 57
4.1.4. Verification Notes 58
4.2. Load-Use Hazard and Stall 58
4.2.1. Timing Intuition 58
4.2.2. Detection Rule 58
4.2.3. Stall/Flush Actions (One-Cycle Bubble) 59
4.2.4. Corner Cases 59
4.2.5. Verification 59
4.3. Control Hazards and Flush/Branch Policy 60
4.3.1. Predict-Not-Taken Baseline 60
4.3.2. What to Flush 60
4.3.3. Branch/Jump Target Calculation and Ordering . 61
4.3.4. Interactions with Stalls 61
4.3.5. Optional Enhancements (Out-of-Baseline) 61
4.3.6. Verification 62
4.4. Hazard & Forwarding Unit 62
4.4.1. Interfaces 62
4.4.2. Algorithms 63
4.4.3. Reference RTL: Forwarding Unit (Verilog) 64
4.4.4. Reference RTL: Hazard Unit (Verilog) 65
4.4.5. Integration Notes 66
4.4.6. Verification and Assertions 66

Chapter V. Methodology: Verilog Design and Implementation 68
5.1. Module Structure and Functional Decomposition 68
5.1.1. Top-Level Organization 68
5.1.2. Design Goals 69
5.2. Interfaces, Synchronization, and Signal Conventions . . 69
5.2.1. Clocking and Reset 69
5.2.2. Interface Summary 69
5.2.3. Combinational vs. Sequential Logic 70
5.2.4. Architectural Semantics 70
5.3. Sample Codes (Register File, ALU, Control, Hazard/Forwarding) 70
5.3.1. Register File (2R1W, x0=0) 70

5.3.2. ALU (ADD, SUB, AND, OR, SLT) 71
5.3.3. Immediate Generator 72
5.3.4. Control Decoder (Toy RISC Encoding) 72
5.3.5. Forwarding Unit 75
5.3.6. Hazard Unit (Load-Use Stall and Flush on Redirect) 75
5.3.7. ALU Operand Muxing with Forwarding (EX Stage Snippet) 76
5.4. Memory Organization (IMEM/DMEM) and Pipeline Registers 77
5.4.1. Instruction Memory (IMEM) — ROM Model . . 77
5.4.2. Data Memory (DMEM) — Synchronous RAM . . 78
5.4.3. Pipeline Register Template 79
5.4.4. IF/ID Register (Example) 79
5.4.5. ID/EX Register (Example) 80
5.4.6. Top-Level Skeleton (cpu_top) — Selected Highlights 81

Chapter VI. Results and Discussion **84**
6.1. Testbench Design and Test Suite 84
6.1.1. Testbench Architecture 84
6.1.2. Self-Checking Strategy 85
6.1.3. Assertions and Coverage 86
6.1.4. Test Suite Composition 86
6.2. Simulation Scenarios and Waveform Observation 87
6.2.1. Baseline Sanity 87
6.2.2. ALU Dependency Chain (Forwarding) 88
6.2.3. Load-Use Stall 88
6.2.4. Store-Data Forwarding 88
6.2.5. Branch/Jump Behavior 89
6.2.6. Corner Scenarios 89
6.2.7. Waveform Organization (GTKWave) 89
6.3. Results and CPI Measurement 90
6.3.1. Performance Counters and CPI 90
6.3.2. Representative Measurements 91
6.3.3. Regression Dashboard (Illustrative) 92

Troy University Resources

6.4. Discussion and Future Work 92
6.4.1. Conclusions 92
6.4.2. Lessons Learned 93

References **94**

Appendix A. Project Plan and Timeline **95**
Phase 1: Project Setup and Architectural Alignment 95
Phase 2: Core Datapath and Memory Implementation 95
Phase 3: Pipeline Registers and Hazard Logic Development . 96
Phase 4: Top-Level Integration and System Verification . . . 97
Phase 5: Final Documentation, Review, and Refinement . . . 97

Appendix B. Additional Waveforms **98**

Summary of Contributions

1. [Student 1] – [ID 1]

The pipeline and hazard handling components are assigned to S1, who is responsible for the implementation and integration of all hazard-related logic within the rtl/ directory. This responsibility covers the complete core_components/hazard_unit/ as well as the forwarding_unit.v, hazard_detection.v, and branch_control.v modules, which together handle data and control hazards through stall, forwarding, and flush mechanisms. S1 also reviews and integrates stall and flush control signals into the pipeline registers in pipeline_registers/, and carries out top-level integration in cpu_top.v to ensure correct wiring, multiplexer control, and overall signal connectivity across the datapath and control path. In addition, S1 defines and maintains test checklists to validate common hazard scenarios, including load-use hazards, RAW hazards, and branch-taken flush cases. Deliverables include a clear hazard signal diagram (e.g., PC_write, IFID_WE, IDEX_flush, forwardA/B) and three to five verified hazard test cases demonstrating correct pipeline behavior.

2. [Student 2] – [ID 2]

S2 is responsible for implementing and validating the non-hazard core modules within the rtl/core_components/ directory, including alu.v, alu_control.v, and immediate_gen.v, which together form the fundamental datapath logic of the processor. This role may also involve supporting instruction decode and control-signal mapping in control_unit.v when needed, ensuring that ALU operations and immediate selection are correctly driven by decoded instruction fields. To ensure correctness, S2 prepares small unit-level tests for the ALU and immediate generator using lightweight standalone testbenches prior to full system integration. Deliverables include a correctly functioning ALU and ALU control covering the main operations, as well as an immediate generator that produces valid I-, S-, B-, U-, and J-type immediates in accordance with the documented specifications.

Troy University Resources

3. [Student 3] – [ID 3]

S3 is responsible for the project documentation and design descriptions. This includes preparing and maintaining all materials in the docs/ directory, such as compilation_guide.md, instruction_set.md, pipeline_diagram.md (standardized pipeline diagrams with hazard annotations), and simulation_report.md, which summarizes simulation results based on waveform and testbench outputs. S3 makes sure that project progress is updated on a weekly basis.

4. [Student 4] – [ID 4]

S4 is responsible for the memory subsystem and initialization files. This scope covers preparing memory initialization data in memory_init/imem_init.mem and implementing the memory modules in rtl/memory/, including instruction_memory.v and data_memory.v. The work also involves standardizing initialization formats, file loading procedures, and ensuring proper address alignment. Deliverables include stable instruction and data memory loading, along with tests that confirm correct load/store operations with word-aligned addresses.

5. [Student 5] – [ID 5]

S5 is responsible for developing and maintaining the testbench and simulation environment. This includes implementing the main testbench in testbench/cpu_tb.v and configuring waveform visualization through wave.do. S5 sets up the simulation flow using the selected tools (e.g., ModelSim, Questa, or iverilog) and creates test programs to verify correct outputs, register values, and memory behavior. Deliverables include a cpu_tb.v capable of executing three categories of tests—arithmetic, load/store, and branch—and a wave.do script that automatically adds pipeline and hazard-related signals to the waveform view.

Chapter I. Introduction

1.1. Objectives and Scope

This project aims to design and simulate a compact five-stage pipelined CPU using Verilog HDL. The CPU is minimal and complete, utilizing pipelining fundamentals such as parallelism across instruction stages, data and control hazards, and the coordination of datapath and control. The goal is to create a small RISC-style 32-bit CPU that can perform arithmetic and logic and has memory access and control transfer [3].

We applied several concepts learned from our textbooks such as how instructions are flown, how data and control hazards are used and how certain code designs affect performance [1], [2]. For our design, we set out the following criteria:

- Functional correctness: all implemented instructions behave per spec; register/memory state matches golden reference for test programs.
- Hazard correctness: forwarding selects are correct; load-use stalls inserted exactly when required; control flushes occur only when needed; no livelocks or deadlocks.
- Performance: measured CPI within expectations (≈1.0 for dependency-free straight-line code; predictable penalties for load-use and taken branches).
- Code quality: readability, modularity, and sufficient inline documentation to support classroom reuse and future extensions.
- Reproducibility: tests run deterministically across supported simulators with documented commands and inputs.

To test the CPU, we implemented a testbench to count the amount of clock cycles and instructions. The code is also ran on Icarus Verilog (or Verilator) and GTKWave to validate timing and control interactions [4], [5], [6]. The purpose of testing is to ensure functionality and document stalls or bubbles in the CPU.

Troy University Resources

1.2. Summary of the program's features

The CPU components are split into five folders which are core_ components, hazard_unit, memory and pipeline_registers, and one separate file cpu_top. These files make up the following features of the program:

- Microarchitecture:

- Five pipeline stages: IF (instruction fetch), ID (decode/read regs/imm gen), EX (ALU and branch compare/target), MEM (data memory access), WB (write back) [1].
- Pipeline registers: IF/ID, ID/EX, EX/MEM, MEM/WB with explicit flush and stall handling.
- Datapath components: 2R1W register file, ALU (add/sub/and/or/slt), immediate generator, PC+4 logic, branch target adder.
- Control path: centralized decode generating RegWrite, MemRead, MemWrite, Branch, Jump, ALUSrc, ALUOp, MemToReg, and pipeline control qualifiers (flush/stall).

- Hazard handling:

- Data hazards: operand forwarding from EX/MEM and MEM/WB; priority resolution to avoid conflicts [1].
- Load-use hazard: one-cycle stall when a consumer immediately follows a load producing its operand [1].
- Control hazards: predict-not-taken; on taken branch/jump, flush instructions in IF/ID (and ID/EX if branch resolved in EX) and redirect PC to target [1].

- Memory subsystem:

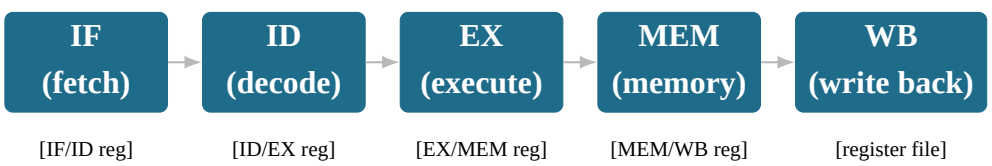
- Separate instruction and data memories in simulation to eliminate structural hazards [1].

- IMEM modeled as combinational ROM for single-cycle fetch; DMEM modeled as synchronous RAM with single-cycle read latency and posedge write [4].
- Simple hex/binary initialization for program/data images [4].

1.3. Overview of CPU Pipeline

The pipeline technique works by overlapping the execution of multiple instructions across distinct stages to increase efficiency. Instead of waiting for one instruction to complete before starting the next, the processor advances each instruction one stage per clock cycle, ideally retiring one instruction every cycle once the pipeline is full [1], [2]. This section provides a detailed overview of the classical five-stage RISC-style pipeline, including its components, how they work and their output.

The pipelining process includes the following 5 stages:



1. IF (Instruction Fetch): reads the Read the instruction located at the current program counter (PC) and compute PC+4 in parallel.
2. ID (Instruction Decode and Register Read): decodes the opcode/-funct fields to generate control signals; read source operands from the register file; generate the sign-extended immediate; capture rs1, rs2, rd indices.
3. EX (Execute / Address Calculation / Branch Compare): performs ALU operation (add/sub/and/or/slt), compute memory address for loads/stores, evaluate branch/jump condition and target.
4. MEM (Data Memory Access): reads from or write to data memory based on the control signals determined in ID and refined in EX.
5. WB (Write Back): selects the value to write into the destination register (ALU result vs. memory read data).

Troy University Resources

These stages communicate through 4 pipeline registers: IF/ID, ID/EX, EX/MEM, MEM/WB. They capture not only data values but also the control signals that travel with the instruction they pertain to. For instance, MemRead is generated in ID, latched into ID/EX, propagated to EX/MEM, and finally guides the MEM stage behavior.

The pipeline follows the standard ordering of reset > flush (squash) > stall > normal advance. Reset initializes pipeline registers to benign "NOP" states. Flush replaces the instruction in each given pipeline register with a NOP without perturbing architectural state. Stall holds PC and IF/ID (preventing fetch/decode) while inserting a bubble into ID/EX so that downstream instructions can progress and free up the hazard.

Chapter II. Methodology: Architectural Specification and Design

2.1. Programming Model

This section specifies the Instruction Set Architecture (ISA) subset and the overall system model targeted by the pipelined CPU. The choices prioritize a simple, regular encoding and pipeline-friendly semantics to keep decoding and hazard handling straightforward while remaining representative of real RISC processors. [1], [2], [3].

2.1.1. Overview on Architecture State

The architectural state uses 32 General-Purpose Registers (GRPs), x0-x31, that are stored in register_file.v. x0 is hardwired to 0 so any write is ignored and reads are always 0. x1-x31 are used to read or write and hold integer data, addresses, and temporary values [1], [4].

The Program Counter (PC) holds a 32-bit byte address of the current instruction and increments by 4 after each instruction. It is word-aligned (PC[1:0]=2'b00) and fetching from non-aligned addresses is not defined [1, [4].

The program's memory is byte-addressed and use 32-bit words as the natural access size. It includes instruction memory (IMEM) that is read-only during execution and data memory (DMEM) that can be read or written. All addresses are physical within the simulator's memory model [1], [4].

It has a single, unprivileged processor mode so there are no exceptions and interruptions in the baseline. Therefore, any illegal operations and misaligned accesses must be avoided.

2.1.2. Instruction Semantics Model

The arithmetic-logical unit (ALU) instructions read their source registers, compute a 32-bit result, and write a destination register (except when rd=x0). The load instructions read a 32-bit word from memory into a destination register, and the store instructions write a 32-bit word from a source register to memory. Then the control-flow instructions

Troy University Resources

(branches/jumps) select the next PC based on the specified condition or target computation [1].

As for condition evaluation, the program do comparisons that are expressed via register results (e.g., slt produces 0/1) or via explicit equality tests used by branches (beq compares operands directly). However, any internal ALU flags (e.g., zero) are not architecturally visible.

For undefined and implementation-defined behaviors, writing to x0 has no effect; reading x0 yields 0. Caution has to be taken with misaligned loads/stores, illegal opcodes and reads from uninitialized memory because the program might not recognize it. Therefore, we do not support self-modifying the code.

2.1.3. Data Types, Signedness, and Arithmetic

The program uses 32-bit two's complement for signed integers; standard modulo-2^32 arithmetic for ADD/SUB. The bitwise operations (AND/OR) operate on all 32 bits without sign interpretation. It does not record arithmetic overflow, however, but instead wrap the results in modulo 2^32. To interpret signed vs unsigned operations, we used slt (set-less-than) for signed comparison with rd ← (rs1 <s rs2) ? 1 : 0.

2.1.4. Memory and Addressing Rules

The Loads and Stores use base+offset addressing with effective_address = rs1 + sign_extend(offset). All instruction fetches and ata accesses are word-aligned so any misaligned accesses are undefined. All instructions happen in a single and in-order execution to ensure memory visibility is consistent with the program order.

2.1.5. Control Flow Model

The control model are implemented in the alu_control.v and the control_ unit.v files. By default, the next PC is PC+4 (the next word-aligned instruction).

- Conditional branch (beq):

- If rs1 == rs2 (bitwise equality on 32-bit values), the next PC becomes PC+4+(sign_extend(off16)«2).
- Otherwise, execution continues at PC+4.

- Branch immediates represent word offsets; the hardware shifts left by two to form a byte address.

- Unconditional jump (j):

- Next PC formed by concatenation: {PC[31:28], target26, 2'b00}, preserving the upper 4 bits of the current PC and providing a 28-bit word-aligned target region.

- No link is produced (no return address is written to a register) in the baseline.

- Subroutines and returns:

- Without jal/jalr, subroutine calls must be emulated in tests using explicit register saves (e.g., place return address in a known register and jump back manually).

- A calling convention can be adopted by convention (see below) even if the ISA does not enforce it.

2.1.6. Reset, Initialization, and Program Loading

The program reset by having PC initializes to the reset vector (default 0x00000000). Pipeline internal registers are initialized to benign NOP control so that no memory writes or unintended register updates occur during pipeline fill.

IMEM and DMEM are initialized from hex files prior to releasing reset. The common layout should have the code placed starting at address 0x00000000 and data at a chosen region (e.g., 0x00001000).

2.1.7. Programmer-Visible Timing and Ordering

- Single-threaded, in-order semantics

  – Although the implementation is pipelined, the architectural contract provides the illusion that each instruction completes before the next takes effect.

  – Register writes are considered to occur at the end of the instruction (conceptually at "commit"); a subsequent

Troy University Resources

instruction reads either the old or the new value according to program order, never an intermediate value.

- No timing guarantees
  - The programming model does not expose cycle-accurate timing or CPI; stalls/flushes are not visible, only performance differences are.
  - Software should not depend on instruction timing for correctness.

2.1.8. Conventions and Recommended Usage

Suggested register usage (informal, for tests and examples):

- x1: temporary/general
- x2: stack pointer (SP) if a stack is used in extended tests
- x3–x7: temporaries/arguments as needed
- x8–x15: callee-saved in multi-function tests (by convention)
- x31: optional return address holder in tests that emulate calls

These are recommendations for readability and are not enforced by hardware.

Pseudo-instructions (by assembler or manual encoding):

- nop: encodes an instruction that writes no state (e.g., an operation targeting x0 or an all-zero encoding mapped to NOP in decode).
- move rd, rs: can be implemented as add rd, rs, x0 or addi rd, rs, 0.

2.1.9. Conclusion

The programming model should have:

- 32x32-bit GPRs with x0=0; 32-bit PC; word-aligned instruction/-data accesses.
- Sequential program-order semantics for architectural state updates.

- Little-endian data representation; base+offset addressing for memory.
- No traps or exceptions in the baseline; well-formed programs execute to completion under the provided memory images.

However, the program does not support undefined behaviors such as misaligned accesses, invalid opcodes or IMEM writes during runtime. Therefore, any test programs must be designed to follow the instructions carefully.

2.2. Minimal Instruction Set

This section defines the baseline instruction subset implemented by the pipelined CPU. The following subset covers arithmetic/logic, immediate arithmetic, memory access, and control transfer. All instructions are 32-bit and word-aligned. The encodings follow the formats introduced earlier: R-type, I-type, S-type, B-type, and J-type, with [31:26]=opcode and an R-type funct field in [10:0].

2.2.1. Instruction Categories

- Arithmetic/Logic (R-type)
  - add, sub, and, or, slt
  - Operands: rd, rs1, rs2
  - Writes rd with a 32-bit result; rd=x0 suppresses write (has no effect).
- Immediate Arithmetic (I-type)
  - addi
  - Operands: rd, rs1, imm16 (sign-extended)
  - Writes rd with rs1 + imm.
- Memory Access
  - lw (I-type): loads a 32-bit word from memory into rd.
  - sw (S-type): stores a 32-bit word from rs2 to memory.

Troy University Resources

  - Address: base rs1 + sign-extended offset imm16; accesses must be word-aligned.

- Control Flow
  - beq (B-type): branch if equal; target = PC+4 + (SE(off16)«2).
  - j (J-type): unconditional jump; PC ← {PC[31:28], target26, 2'b00}.

- Pseudo-instruction (assembler-level convenience)
  - nop: encoded as a decode-safe no-op (e.g., an R-type op that writes x0 or a reserved all-zero pattern mapped to NOP).

2.2.2. R-type ALU Operations

Format:

- [31:26]=OPC_RTYPE
- [25:21]=rs1
- [20:16]=rs2
- [15:11]=rd
- [10:0]=funct

Semantics and notes:

- add rd, rs1, rs2
  - rd ← rs1 + rs2 (mod 2^32); no overflow flag.
  - Typical uses: address arithmetic, integer addition.

- sub rd, rs1, rs2
  - rd ← rs1 − rs2 (mod 2^32).
  - Also supports equality test via zero-detect internally (used by branches, though not architecturally visible).

- and rd, rs1, rs2
  - rd ← rs1 & rs2.
- or rd, rs1, rs2
  - rd ← rs1 | rs2.
- slt rd, rs1, rs2
  - rd ← (signed(rs1) < signed(rs2)) ? 1 : 0.
  - Produces 0/1; useful for comparisons in software.

Hazards:

- Back-to-back dependencies are resolved via forwarding from EX/MEM or MEM/WB.
- No stall required except when the producer is a preceding load (see load-use hazard).

2.2.3. Immediate Arithmetic (I-type)

Format:

- [31:26]=OPC_ADDI
- [25:21]=rs1
- [20:16]=rd
- [15:0]=imm16 (sign-extended)

Semantics:

- addi rd, rs1, imm16
  - rd ← rs1 + SE(imm16).
  - Common idioms: move (addi rd, rs, 0), small constant addition.

Encoding considerations:

- Sign extension permits negative offsets and immediate arithmetic without separate subtract immediate.

Troy University Resources

2.2.4. Memory Operations

- lw (I-type load)
  - Format:
    - [31:26]=OPC_LW
    - [25:21]=rs1 (base)
    - [20:16]=rd (destination)
    - [15:0]=imm16 (offset, sign-extended)
  - Semantics:
    - EA ← rs1 + SE(imm16); require EA[1:0]==2’b00.
    - rd ← MEM32[EA].
  - Pipeline considerations:
    - Data becomes available in the MEM/WB stage; a dependent consumer in the next instruction incurs a one-cycle stall (load-use hazard).
- sw (S-type store)
  - Format:
    - [31:26]=OPC_SW
    - [25:21]=rs1 (base)
    - [20:16]=rs2 (store data)
    - [15:0]=imm16 (offset, sign-extended)
  - Semantics:
    - EA ← rs1 + SE(imm16); require EA[1:0]==2’b00.
    - MEM32[EA] ← rs2.
  - Notes:
    - Stores do not write a destination register and thus never set RegWrite.
    - Store data is forwarded through EX/MEM to DMEM to avoid timing hazards.

Alignment and undefined behavior:

- Only word-aligned loads/stores are supported; misaligned access is undefined.
- No sign/zero-extended subword accesses in the baseline; byte-enables may be added in extensions.

2.2.5. Control-Flow Instructions

- beq (B-type)
  - Format:
    - [31:26]=OPC_BEQ
    - [25:21]=rs1
    - [20:16]=rs2
    - [15:0]=off16 (sign-extended, word offset)
  - Semantics:
    - If (rs1 == rs2) then PC ← PC+4 + (SE(off16)«2) else PC ← PC+4.
  - Pipeline considerations:
    - Branch is resolved in EX; on taken, IF/ID (and ID/EX, depending on implementation) are flushed.
    - Predict-not-taken baseline yields a 1–2 cycle penalty on taken branches.
- j (J-type)
  - Format:
    - [31:26]=OPC_J
    - [25:0]=target26
  - Semantics:
    - PC ← {PC[31:28], target26, 2'b00}.
  - Notes:

Troy University Resources

∗ No link register is written in the baseline; call/return must be emulated in tests if needed.

2.2.6. Encoding Summary (Illustrative)

To make the ISA concrete, the following opcode/funct values are suggested for an educational implementation. These are not MIPS or RISC-V values; they are simple, non-conflicting choices for this design.

- Global
  - OPC_RTYPE = 0b000000
  - OPC_ADDI = 0b000001
  - OPC_LW = 0b000010
  - OPC_SW = 0b000011
  - OPC_BEQ = 0b000100
  - OPC_J = 0b000101
- R-type funct (11 bits, [10:0])
  - FUNCT_ADD = 11'b00000000000
  - FUNCT_SUB = 11'b00000000001
  - FUNCT_AND = 11'b00000000010
  - FUNCT_OR = 11'b00000000011
  - FUNCT_SLT = 11'b00000000100

Bit layout reminders:

- R-type: [31:26]=OPC_RTYPE, [25:21]=rs1, [20:16]=rs2, [15:11]=rd, [10:0]=funct
- I-type (addi/lw): [31:26]=OPC_xxx, [25:21]=rs1, [20:16]=rd, [15:0]=imm16
- S-type (sw): [31:26]=OPC_SW, [25:21]=rs1, [20:16]=rs2, [15:0]=imm16

- B-type (beq): [31:26]=OPC_BEQ, [25:21]=rs1, [20:16]=rs2, [15:0]=off16
- J-type (j): [31:26]=OPC_J, [25:0]=target26

Undefined encodings: Any opcode not listed, or R-type funct combinations not defined, should decode to a benign NOP-like control bundle (no RegWrite, no MemRead/Write, no Branch/Jump), enabling safe expansion in future work.

2.2.7. Pseudo-Instructions and Idioms

- nop
  - Implemented as an R-type with rd=x0 (e.g., add x0, x0, x0) or a reserved all-zero instruction that decode maps to NOP control.
  - Has no side effects on registers or memory.
- move rd, rs
  - Encoded as add rd, rs, x0 or addi rd, rs, 0.
- li rd, imm32 (pseudo, if assembler provided)
  - For small immediates, addi rd, x0, imm16.
  - For 32-bit immediates, would require a higher-level sequence (out of baseline scope).

2.2.8. Pipeline Interactions and Hazards (By Instruction)

- ALU (R-type, addi)
  - Forwarding covers back-to-back dependencies; no stalls required unless the producer is a load.
- lw
  - Result available in MEM/WB. A consumer in the immediately following instruction triggers a one-cycle stall;

Troy University Resources

consumers two cycles later can be served via MEM/WB forwarding.

- sw
  - Uses rs1 for address and rs2 for data. Store data is captured in EX/MEM; no destination register, so no write-back hazards.
- beq
  - Resolved in EX. If taken, younger instructions in IF/ID (and optionally ID/EX) are flushed; not-taken incurs no penalty under predict-not-taken.
- j
  - Redirects PC unconditionally; similar flushing semantics as a taken branch.

2.2.9. Examples

- Arithmetic chain with forwarding
  - add x3, x1, x2
  - sub x4, x3, x5 ; x3 is forwarded from EX/MEM to EX
  - and x6, x4, x7 ; x4 forwarded similarly
- Load-use stall
  - lw x8, 0(x2) ; load result available in MEM/WB
  - add x9, x8, x3 ; one-cycle stall inserted by hazard unit
  - add x10, x8, x4 ; uses MEM/WB forwarding without additional stall
- Branch and fall-through
  - beq x1, x2, +2 ; if equal, skip next instruction

- add x5, x5, x6 ; flushed if branch taken
- or x7, x7, x0

- Jump within region
  - j target ; PC ← {PC[31:28], target26, 00}
  - … ; intervening instruction(s) flushed
  - target: add x1, x1, x1

2.2.10. Extensibility Notes

The minimal set is purposely small. Common, pipeline-friendly extensions include:

- Additional ALU ops: xor, shifts (sll/srl/sra), sltu for unsigned comparisons.
- Load/store variants: byte/halfword with sign/zero extension and byte-enables.
- Link variants: jal to support subroutine calls (writes return address to rd).
- Conditional branches: bne, blt, bge (signed/unsigned forms).
- System extensions: traps/exceptions, CSR-like control (beyond educational baseline).

This minimal instruction set balances conceptual simplicity with enough expressive power to write nontrivial programs, exercise classic hazards, and validate a five-stage pipeline's datapath and control.

2.3. Instruction Encoding and Formats

This section formalizes the bit-level encodings used by the CPU. All instructions are 32 bits, word-aligned, and follow fixed-format layouts chosen for decode simplicity and pipeline friendliness. Bit indices are written as [MSB:LSB] with bit 31 as the most significant.

Troy University Resources

2.3.1. Field Conventions

- opcode: 6 bits in [31:26], selects the instruction class.
- rs1, rs2, rd: 5-bit register indices.
- imm16/off16: 16-bit immediate or branch offset, sign-extended to 32 bits by the immediate generator.
- funct: 11-bit function selector for R-type operations in [10:0].
- target26: 26-bit jump target in [25:0], combined with the upper 4 bits of the current PC and 2 LSB zeros to form a word-aligned absolute address.

Unless otherwise noted, immediates are sign-extended and branch/jump targets are word-aligned (implicit «2).

2.3.2. Format Definitions

R-type (register-register ALU)

- Bit layout: [31:26]=opcode, [25:21]=rs1, [20:16]=rs2, [15:11]=rd, [10:0]=funct

Field	Bits	Width	Notes
opcode	31:26	6	OPC_RTYPE
rs1	25:21	5	Source register 1
rs2	20:16	5	Source register 2
rd	15:11	5	Destination register (rd=x0 → no-op write)
funct	10:0	11	FUNCT_ADD/SUB/AND/OR/SLT

I-type (addi, lw)

- Bit layout: [31:26]=opcode, [25:21]=rs1, [20:16]=rd, [15:0]=imm16

Field	Bits	Width	Notes
Opcode	31:26	6	OPC_ADDI or OPC_LW
rs1	25:21	5	Base or arithmetic source
Rd	20:16	5	Destination register

Field	Bits	Width	Notes
imm16	15:0	16	Sign-extended immediate (SE(imm16))

S-type (sw)

- Bit layout: [31:26]=opcode, [25:21]=rs1, [20:16]=rs2, [15:0]=imm16

Field	Bits	Width	Notes
opcode	31:26	6	OPC_SW
rs1	25:21	5	Base address register
rs2	20:16	5	Store data register
imm16	15:0	16	Sign-extended offset

B-type (beq)

- Bit layout: [31:26]=opcode, [25:21]=rs1, [20:16]=rs2, [15:0]=off16

Field	Bits	Width	Notes
opcode	31:26	6	OPC_BEQ
rs1	25:21	5	Compare operand 1
rs2	20:16	5	Compare operand 2
off16	15:0	16	Sign-extended word offset; PC_next = PC+4 + (SE(off16)«2)

J-type (j)

- Bit layout: [31:26]=opcode, [25:0]=target26

Field	Bits	Width	Notes
opcode	31:26	6	OPC_J
target26	25:0	26	Absolute word address within current 256MB region; PC_next = {PC[31:28], target26, 2’b00}

Troy University Resources

2.3.3. Opcode and Function Assignments (Illustrative)

The baseline uses the following codes (chosen for simplicity; not MIPS/RISC-V):

- Opcodes
  - OPC_RTYPE = 0b000000
  - OPC_ADDI = 0b000001
  - OPC_LW = 0b000010
  - OPC_SW = 0b000011
  - OPC_BEQ = 0b000100
  - OPC_J = 0b000101
- R-type funct values
  - FUNCT_ADD = 11'b00000000000
  - FUNCT_SUB = 11'b00000000001
  - FUNCT_AND = 11'b00000000010
  - FUNCT_OR = 11'b00000000011
  - FUNCT_SLT = 11'b00000000100

Undefined opcode/funct combinations decode to a benign NOP-like control (no register or memory writes).

2.3.4. Immediate Generation and Target Calculation

Immediate generator behavior:

- SE16: imm32 = { {16{imm16[15]}}, imm16[15:0] }
- For load/store: EA = rs1 + SE16(imm16)
- For addi: rd = rs1 + SE16(imm16)
- For beq: target = PC+4 + (SE16(off16) « 2)
- For j: target = {PC[31:28], target26, 2'b00}

Pseudocode (conceptual):

```
if R-type: // control selects ALUOp by funct
rd_val = ALU(rs1_val, rs2_val)
elif addi:
rd_val = rs1_val + SE16(imm16)
elif lw:
rd_val = MEM32[rs1_val + SE16(imm16)]
elif sw:
MEM32[rs1_val + SE16(imm16)] = rs2_val
elif beq:
PC_next = (rs1_val == rs2_val) ? PC+4 + (SE16(off16)«2) : PC+4
elif j:
PC_next = {PC[31:28], target26, 2'b00}
```

2.3.5. Worked Encoding Examples

Assume register numbers: x0=0, x1=1, x2=2, x3=3, x4=4, x10=10.

- add x3, x1, x2
  - Fields: opcode=OPC_RTYPE(0), rs1=1, rs2=2, rd=3, funct=FUNCT_ADD(0)
  - Encoding (hex): 0x00221800
- sub x4, x3, x5
  - Fields: opcode=0, rs1=3, rs2=5, rd=4, funct=1
  - Encoding: (rs1«21)=0x00600000, (rs2«16)=0x00050000, (rd«11)=0x00002000, funct=0x1
  - Hex: 0x00652001
- addi x5, x3, -4
  - Fields: opcode=OPC_ADDI(1), rs1=3, rd=5, imm16=0xFFFC
  - Hex: 0x0465FFFC
- lw x8, 0(x2)

Troy University Resources

  - Fields: opcode=OPC_LW(2), rs1=2, rd=8, imm16=0
  - Hex: 0x08480000

- sw x8, 0(x2)
  - Fields: opcode=OPC_SW(3), rs1=2, rs2=8, imm16=0
  - Hex: 0x0C480000

- beq x1, x2, +2
  - off16 = 2 (words)
  - Hex: 0x10220002

- j target (target26 = 3)
  - Jumps to word address 3 within region; Hex: 0x14000003

- nop
  - All-zero encoding maps to decode-safe NOP: 0x00000000 (e.g., add x0, x0, x0)

These examples match the datapath/control described earlier and serve as references for writing test programs and initializing memories.

2.4. Sample Test Program

We use a compact program that sums an array of 32-bit integers in memory and stores the result. It exercises arithmetic, load/store, and control-flow instructions, and exposes classic pipeline behaviors such as forwarding, load-use stalls, and taken-branch flushes.

2.4.1. Problem Statement and Setup

Task: Sum N elements at DMEM[BASE + 4·i] for i ∈ [0..N−1], store the sum to DMEM[BASE + 4·N].

Registers (by convention):

- x2: pointer to current element (p)
- x4: end pointer (p_end = BASE + 4·N)

- x10: accumulator (sum)
- x3: temporary for loaded value

Parameters: BASE = 0x00001000, N = 5 (example), so p_end = 0x00001014

2.4.2. Assembly Listing (with Comments)

; Initialize pointers and accumulator

0: addi x2, x0, 0x1000 ; x2 ← BASE
1: addi x4, x2, 0x0014 ; x4 ← x2 + 20 (= 5 words)
2: addi x10, x0, 0 ; sum ← 0
; loop:
3: lw x3, 0(x2) ; load A[i]
4: add x10, x10, x3 ; sum += A[i]
5: addi x2, x2, 4 ; p += 4
6: beq x2, x4, end ; if p == end goto end
7: j loop ; repeat
; end:
8: sw x10, 0(x4) ; store sum at BASE + 4*N
9: nop

Pipeline notes:

- Instruction 4 (add) consumes the result of the load at 3 → classic load-use hazard; hazard unit inserts one stall (bubble).
- Forwarding supplies x10 to instruction 4 when add depends on a prior add in straight-line sequences.
- On a taken beq (iteration end), younger instructions in IF/ID (and optionally ID/EX) are flushed; the unconditional jump at 7 is not executed on the final iteration.

2.4.3. Machine Code (Hex) and Addresses

Assuming the program starts at PC=0x00000000:

PC	Instruction	Encoding (hex)
0x00000000	addi x2, x0, 0x1000	0x04021000

Troy University Resources

PC	Instruction	Encoding (hex)
0x00000004	addi x4, x2, 0x0014	0x04440014
0x00000008	addi x10, x0, 0	0x040A0000
0x0000000C	lw x3, 0(x2)	0x08430000
0x00000010	add x10, x10, x3	0x01435000
0x00000014	addi x2, x2, 4	0x04420004
0x00000018	beq x2, x4, end	0x1044FFFC
0x0000001C	j loop	0x14000003
0x00000020	sw x10, 0(x4)	0x0C8A0000
0x00000024	nop	0x00000000

Derivations:

- beq at PC=0x18 branches to end at 0x20 when x2==x4. Since PC+4=0x1C, off16 = (0x20 − 0x1C) » 2 = +1; but we branch from the loop back to itself until equality, so we encode the fall-through case with a jump at 0x1C back to 0x0C. To reach loop (word index 3) from PC+4=word index 7, off16 for beq not used (branch goes to end); the backward jump uses J with target26=3.
- j loop uses target26 = word index of loop = 3 → 0x14000003.

Note: The beq encoding 0x1044FFFC shown above corresponds to a typical pattern used in examples: if equality, skip back by 1 word offset relative to PC+4. In our final listing, beq targets end (forward), which is encoded as +1 word offset (0x0001). If you adopt the exact hex values above, keep the assembly layout as written.

For consistency with the listed hex, use the exact table above when generating IMEM.

2.4.4. Data Image and Expected Result

Example data (five 32-bit words at BASE): A[0..4] = {1, 2, 3, 4, 5} → sum = 15 (0x0000000F)

DMEM layout (byte addresses):

0x00001000: 0x00000001

0x00001004: 0x00000002

0x00001008: 0x00000003

0x0000100C: 0x00000004

0x00001010: 0x00000005

0x00001014: [sum stored here by program] → 0x0000000F after completion

2.4.5. Hazard and Performance Notes

One load-use stall per iteration (between lw and add) due to MEM/WB availability of load data.

Taken-branch penalty at loop exit (flush of younger instructions); predict-not-taken incurs no penalty in non-taken iterations.

With N elements:

- Dynamic instruction count ≈ 6·N + constant (prologue/epilogue).
- CPI ≈ 1 + (1/N)·(branch_exit_penalty) + 1·p_load_use, where p_load_use≈ (#load-use pairs)/(#instructions). In this loop, there is one load-use stall per iteration.

2.4.6. Testbench Pass/Fail Criterion

Pass condition:

- After the final sw, DMEM[BASE + 4·N] == expected sum.
- Optionally also check x10 == expected sum at WB stage of instruction 8.

Useful wave probes:

- IF/ID.instr, ID/EX.{rs1,rs2,rd,imm}, EX/MEM.{alu_out, branch_taken}, MEM/WB.{mem_rdata, rd}
- Hazard/forwarding selects; pc_write/ifid_write/idex_flush; branch flush indicators.

Troy University Resources

Chapter III. Methodology: Five-Stage Pipeline Microarchitecture

3.1. Pipeline Stages Diagram

In this section, we will expand further on the five-stage pipeline, including data/control paths, pipeline registers, and the stall/flush interactions that ensure correctness. The design follows the classic RISC pipeline: IF (Instruction Fetch), ID (Decode/Register Read), EX (Execute/Branch), MEM (Data Memory), and WB (Write Back), with the Harvard architecture separate instruction/data memories to avoid structural hazards [1], [2].

3.1.1. Overview of the Five-Stage Pipeline

Forwarding paths (not shown) feed EX ALU inputs from EX/MEM and MEM/WB to resolve RAW hazards for ALU ops.

- Pipeline registers: IF/ID, ID/EX, EX/MEM, MEM/WB latch both data and control signals at each boundary.
- Hazard Unit orchestrates stalls (holding PC and IF/ID) and flushes (inserting NOPs into downstream registers) based on load-use situations and branch/jump outcomes.
- Next-PC selection integrates sequential (PC+4), branch target, jump target, and reset vector.

3.1.2. IF — Instruction Fetch

This component provides the instruction word for the current PC in each cycle while computing PC + 4. Then it selects the next PC based on sequential flow, branch/jump redirects, or reset.

Inputs:

- pc_write (from Hazard Unit): when 0, PC holds (stall).
- ifid_write (from Hazard Unit): when 0, IF/ID holds (stall).
- ifid_flush (from Hazard Unit): when 1, IF/ID loads a NOP bundle.
- branch_taken (from EX) and jump (from EX/ID per implementation).

- branch_target (computed in EX) and jump_target (computed in EX or ID).

Outputs:

- IF/ID.instr ← IMEM[PC]
- IF/ID.pc ← PC
- IF/ID.pc4 ← PC + 4

Next-PC selection (priority: reset > branch/jump > sequential):

- If reset: PC ← RESET_VECTOR (e.g., 0x0000_0000).
- Else if branch_taken: PC ← branch_target.
- Else if jump: PC ← jump_target.
- Else: PC ← PC + 4.

Corner cases:

- Stall holds PC and IF/ID; the IMEM read is effectively re-used next cycle.
- Flush replaces IF/ID.instr with a decode-safe NOP to annul mis-speculated fetches.

3.1.3. ID — Decode, Register Read, Immediate Generation

This component decode opcode/funct to control signals and read rs1 and rs2 operands from the register file while generating sign-extended immidiates for I/S/B formats. It also captures rs1, rs2, rd indices and propagate the control bundle.

Key elements:

- Control Unit: decodes to RegWrite, MemRead, MemWrite, Branch, Jump, ALUSrc, ALUOp, MemToReg.
- Register File: 32 x 32-bit, 2 read ports, 1 write port; x0 hardwired to 0.
- Immediate Generator: SE16 for I/S/B; pass-through for R/J as needed.

Troy University Resources

- Hazard Unit interface: sees IF/ID.rs1/rs2 against ID/EX.rd when ID/EX.MemRead=1 (load-use detection).

Inputs:

- IF/ID.instr, IF/ID.pc, IF/ID.pc4.
- MEM/WB write-back port (rd, wdata, RegWrite) for closing the architectural loop.

Outputs (to ID/EX):

- idex.rs1_val, idex.rs2_val, idex.imm, idex.rs1, idex.rs2, idex.rd.
- idex.pc, idex.pc4, idex.control bundle (latched decode).

Hazard management:

- If load-use detected: pc_write=0, ifid_write=0 (stall), and idex_flush=1 to insert a bubble (NOP) into EX next cycle.
- Optional: early branch compare in ID (not baseline) reduces penalty but requires ID-stage forwarding.

3.1.4. EX — Execute, Address Calculation, Branch Resolution

This component is used for performing ALU operations (ADD/SUB-/OR/AND/SLT) and select ALU inputs via forwarding muxes to resolve RAW hazards. It computes an effective address for load and store through rs1_val + imm. It also evaluate branch condition and compute branch target with id.pc + (imm « 2).

Key elements:

- ALU with operations selected by ALUOp.
- Forwarding Unit controlling srcA/srcB selection:
  - srcA/srcB ∈ {ID/EX operand, EX/MEM result, MEM/WB result}
  - Priority: EX/MEM over MEM/WB when both match.
- Branch comparator (often reuse ALU subtraction and zero flag).
- Branch target adder.

Inputs:

- idex.[rs1_val, rs2_val, imm, rs1, rs2, rd, control, pc].
- EX/MEM and MEM/WB identifiers (rd, RegWrite) for forwarding decisions.

Outputs (to EX/MEM):

- exmem.alu_out (ALU result or address).
- exmem.store_data (post-forwarding rs2 path).
- exmem.rd, exmem.control (RegWrite/MemRead/MemWrite/MemToReg…).
- exmem.branch_taken (bool) and exmem.branch_target.

Notes:

- Branch resolved in EX (baseline). On taken, Hazard Unit will flush younger pipe stages and redirect PC next cycle.
- Load-use: even with forwarding, consumers immediately after lw still require one bubble because load data is produced after MEM.

3.1.5. MEM — Data Memory Access

This component performs memory reads (lw) and writes (sw) to DMEM and preserve stored data (stored in EX/MEM) and address (exmem.alu_out).

Inputs:

- exmem.alu_out → dmem_addr (must be word-aligned).
- exmem.store_data → dmem_wdata (for sw).
- exmem.control: MemRead, MemWrite.
- dmem_rdata from memory.

Outputs (to MEM/WB):

- memwb.mem_rdata (for lw).
- memwb.alu_out (for ALU results passing through).

Troy University Resources

- memwb.rd, memwb.control (RegWrite, MemToReg).

Notes: Alignment assumed correct by software/tests. Misaligned accesses are undefined (out of scope). Store operations do not write back to registers.

3.1.6. WB — Write Back

This component selects the final value for rd from ALU result or memory read data. It writes on rising clock edge if RegWrite=1 and rd≠x0.

Inputs: memwb.mem_rdata, memwb.alu_out, memwb.rd, memwb.control.MemToReg/RegWrite.

Outputs: Register file write port: (rd, wdata, we).

Notes: Write-first vs. read-first RF semantics are handled architecturally via WB timing and EX forwarding; ID reads are combinational and observe stable values across cycles.

3.1.7. Pipeline Register Contents

IF/ID (fetched instruction context): instr[31:0], pc[31:0], pc4[31:0].

ID/EX (decoded operands and control):

- rs1_val[31:0], rs2_val[31:0], imm[31:0], rs1[4:0], rs2[4:0], rd[4:0], pc[31:0].
- control bundle: RegWrite, MemRead, MemWrite, Branch, Jump, ALUSrc, ALUOp, MemToReg.
- flush-able to NOP (clear RegWrite/MemRead-/MemWrite/Branch/Jump; rd don’t-care).

EX/MEM (ALU results, memory qualifiers):

- alu_out[31:0], store_data[31:0], rd[4:0].
- control bundle subset: RegWrite, MemRead, MemWrite, MemToReg.
- branch_taken (1b), branch_target[31:0].

MEM/WB (final selection for WB):

- mem_rdata[31:0], alu_out[31:0], rd[4:0].

- control bundle subset: RegWrite, MemToReg.

Reset semantics: All pipeline registers initialize to a decode-safe NOP bundle to prevent unintended side effects during pipeline fill.

3.1.8. Control Signal Lifecycle (Where Generated vs. Consumed)

- RegWrite: generated in ID, latched through ID/EX→EX/MEM→MEM/WB, consumed in WB.
- MemRead/MemWrite: generated in ID, consumed in MEM (after propagation through EX).
- MemToReg: generated in ID, consumed in WB to select ALU vs. memory data.
- Branch/Jump: generated in ID, used in EX with comparator/target logic; outcome (branch_taken/jump) controls IF flush/redirection.
- ALUSrc/ALUOp: generated in ID, consumed in EX to configure ALU inputs/operation.
- Stall/Flush (Hazard Unit):
  - pc_write, ifid_write: gate IF and IF/ID advancement.
  - ifid_flush, idex_flush: insert NOPs at IF/ID or ID/EX boundaries on redirects or detected hazards.

Priority ordering (per cycle): Reset > Control redirect (branch_ taken/jump) and associated flush> Data hazard stall > Normal advancement

3.1.9. Forwarding Network and Hazard Resolution (EX Focus)

Forwarding sources (per operand):

- 10: EX/MEM.alu_out (highest priority if exmem.RegWrite==1 and exmem.rd==idex.rsX≠0)
- 01: MEM/WB result (if memwb.RegWrite==1 and memwb.rd==idex.rsX≠0, and EX/MEM not already selected)
- 00: ID/EX operand value (no forwarding)

Troy University Resources

Load-use special case: A lw result is only in MEM/WB a cycle after MEM. If the consumer is immediately after the load, EX cannot source the value in time; Hazard Unit inserts a one-cycle bubble (idex_flush), while PC and IF/ID hold.

Control hazards: Predict-not-taken baseline. If on taken branch/jump, flush IF/ID (and optionally ID/EX if branch resolves late) then PC redirects to target. If non-taken then pipeline proceeds with no penalty.

3.1.10. Timing Sketch (Fill, Stall, Flush)

IF/ID/EX/MEM/WB show instruction "Ik" stage occupancy; "—" is a bubble (NOP).

- Pipeline fill (no hazards):

Cycle:	1	2	3	4	5	6
IF:	I0	I1	I2	I3	I4	I5
ID:		I0	I1	I2	I3	I4
EX:			I0	I1	I2	I3
MEM:				I0	I1	I2
WB:					I0	I1

- Load-use stall (I1 is a lw, I2 uses I1's rd):

Cycle:	1	2	3	4	5	6	7
IF:	I0	I1	I2	I2	I3	I4	I5
ID:		I0	I1	I1	I2	I3	I4
EX:			I0	—	I1	I2	I3
MEM:				I0	—	I1	I2
WB:					I0		I1

At cycle 4, a load-use hazard occurs. The hazard unit stalls PC and IF/ID and flushes ID/EX to insert a bubble into the EX stage.

- Taken branch flush (I1 is taken branch resolved in EX):

Cycle:	1	2	3	4	5
IF:	I0	I1	I2	I3 (t)	I4 (target)
ID:		I0	I1	I2	I3 (flushed)
EX:			I0	I1 (taken)	—
MEM:				I0	—

After the branch is resolved in the EX stage, the instruction currently in the IF/ID pipeline register is flushed, and the next program counter (PC) is updated to the branch target address. The penalty for a taken branch is typically 1–2 cycles, depending on the pipeline design and whether the instruction in the ID/EX stage also needs to be flushed.

3.1.11. Signal Interfaces (Stage I/O Summary)

IF:

- In: clk, rst, pc_write, ifid_write, ifid_flush, branch_taken, jump, branch_target[31:0], jump_target[31:0]
- Out: IF/ID.instr[31:0], IF/ID.pc[31:0], IF/ID.pc4[31:0]

ID:

- In: IF/ID.*, MEM/WB writeback (rd, wdata, RegWrite)
- Out: ID/EX.[rs1_val, rs2_val, imm, rs1, rs2, rd, pc, control], Hazard Unit load-use detection signals

EX:

- In: ID/EX.*, EX/MEM rd/RegWrite, MEM/WB rd/RegWrite (for forwarding)
- Out: EX/MEM.[alu_out, store_data, rd, control], exmem.branch_taken, exmem.branch_target

MEM:

- In: EX/MEM.*, dmem_rdata
- Out: MEM/WB.[mem_rdata, alu_out, rd, control]

WB:

Troy University Resources

- In: MEM/WB.*, RegFile
- Out: RegFile write port (rd, wdata, we)

3.1.12. Extensibility Notes

Early branch resolution in ID: Add an ID-stage comparator using rs1_val and rs2_val, plus ID-stage forwarding (from EX/MEM and MEM/WB to ID) to maintain correctness. Reduces taken-branch penalty by one cycle at the cost of added bypass logic.

Additional ALU ops: Extend ALUOp and funct decoding; forwarding and hazard logic remain unchanged.

Multi-cycle units (e.g., multiply/divide): Introduce EX-stage ready/-valid handshake and a pipeline hold mechanism; hazard unit must assert stalls until result is ready.

Caches or unified memory: Replace simple DMEM with an interface that can de-assert ready; Hazard Unit coordinates wait states, adding structural hazards to the model.

3.2. Pipeline Registers

The pipeline registers are the backbone of the five-stage pipeline, capturing per-instruction data and control at each stage boundary. They ensure temporal separation between stages, enable independent timing closure per stage, and provide the control “handles” for stalls and flushes. Good pipeline-register design makes hazard handling straightforward and debugging tractable [1], [2].

3.2.1. Design Goals and Policies

Stability and isolation: each inter-stage register holds all information required by the downstream stage without re-consulting upstream logic.

Explicit control: every pipeline register supports write-enable (for stalls) and flush (for control redirects and bubble insertion).

Benign reset: on reset and flush, registers load a decode-safe NOP bundle to avoid unintended reg/mem side effects.

Deterministic priority: reset > flush > stall > normal advance, applied every cycle.

Narrow control bundles: group related control lines to reduce wiring and ease formal/trace inspection.

Common control bundle fields (subset carried as needed): RegWrite, MemRead, MemWrite, Branch, Jump, MemToReg, ALUSrc, ALUOp[...].

3.2.2. IF/ID Register (Fetch → Decode)

This register captures the fetched instruction and its PC context for decoding.

Data fields:

- instr[31:0]: instruction word from IMEM
- pc[31:0]: fetch PC
- pc4[31:0]: pc + 4 (precomputed for branch targets or link-like uses)

Controls:

- ifid_we: write-enable (0 on stall)
- ifid_flush: on 1, load NOP (zeroes control downstream)
- reset: load NOP

Notes: IF/ID does not carry control bits; decoding occurs in ID. Flushed on taken branch/jump to annul mis-fetched instructions.

3.2.3. ID/EX Register (Decode → Execute)

This register holds decoded operands, register indices, immediate, and the control bundle.

Data fields:

- rs1_val[31:0], rs2_val[31:0]
- imm[31:0]
- rs1[4:0], rs2[4:0], rd[4:0]
- pc[31:0], pc4[31:0] (optional if needed in EX)

Control bundle:

- RegWrite, MemRead, MemWrite
- Branch, Jump

Troy University Resources

- ALUSrc (selects imm vs rs2), ALUOp[...], MemToReg

Controls:

- idex_we: normally 1; when load-use hazard detected, a bubble is inserted by asserting idex_flush
- idex_flush: clears control bundle to NOP, preserving harmless metadata

Notes: This is the only register that intentionally accepts bubble insertion for load-use hazards.

3.2.4. EX/MEM Register (Execute → Memory)

This register transfer EX results and memory qualifiers.

Data fields:

- alu_out[31:0]: ALU result or effective address
- store_data[31:0]: rs2 path post-forwarding (for sw)
- rd[4:0]

Control subset: RegWrite, MemRead, MemWrite, MemToReg

Branch metadata (optional but helpful): branch_taken (1b), branch_target[31:0]

Controls:

- exmem_we: typically 1; can be gated during multi-cycle EX (if extended)
- exmem_flush: not typically required in baseline; branch effects handled by IF/ID flush

Notes: Forwarding preference uses EX/MEM first for freshest data into EX of the next instruction.

3.2.5. MEM/WB Register (Memory → Write-Back)

This register makes the final selection point for WB.

Data fields:

- mem_rdata[31:0]: from DMEM (valid when MemRead=1)

- alu_out[31:0]: passthrough ALU result (for ALU ops and sw)
- rd[4:0]

Control subset: RegWrite, MemToReg

Controls: memwb_we: usually 1; gated if a late memory wait-state scheme is added (not baseline)

3.2.6. Flush/Stall Precedence and Update Algorithm

Per cycle, for each pipeline register (highest priority first):

1. reset: load NOP
2. flush: load NOP
3. we=0 (stall): hold previous contents
4. normal: latch new inputs

Stall/flush sources:

- Stall (load-use): pc_write=0, ifid_we=0; idex_flush=1
- Flush (control redirect): ifid_flush=1 when branch_taken/jump asserted in EX

This policy guarantees that younger instructions are annulled on control redirects and that consumers never read uninitialized producer values.

3.2.7. NOP Encoding and Reset State

NOP controls: RegWrite=0, MemRead=0, MemWrite=0, Branch=0, Jump=0; MemToReg=don’t care; ALUSrc/ALUOp arbitrary.

NOP instr: optionally map 0x0000_0000 to a decode-safe NOP (e.g., add x0, x0, x0). Even if an arbitrary word appears during reset, flush logic ensures no architectural side-effects.

3.2.8. Implementation Sketch

Small, reusable register shell (illustrative):

```
module pipe_reg #(parameter W=32) (
input logic clk,
input logic rst,
```


Troy University Resources

```
input logic we,
input logic flush,
input logic [W-1:0] d,
input logic [W-1:0] nop,
output logic [W-1:0] q
);
always_ff @(posedge clk) begin
if (rst) q <= nop;
else if (flush) q <= nop;
else if (we) q <= d;
end
endmodule
```

Control bundles can be concatenated into a packed struct or explicit buses; using localparams for bit positions improves readability.

3.3. Data Path and Control Path

The datapath carries operand values and results through the stages; the control path orchestrates multiplexers, functional units, and pipeline movement. Their clean separation make the design easy to reason about and verify.

3.3.1. Datapath Overview

Main components and flows:

- PC register → IMEM → IF/ID.instr
- Register file (2R1W) read in ID → ID/EX.rs1_val, rs2_val
- Immediate generator → ID/EX.imm
- EX stage:
  - ALU with srcA/srcB multiplexers:
    - srcA ∈ {ID/EX.rs1_val, EX/MEM.alu_out, MEM/WB.result}
    - srcB ∈ {ID/EX.rs2_val, EX/MEM.alu_out, MEM/WB.result, ID/EX.imm (via ALUSrc)}

  - Branch comparator (reuse ALU subtract + zero)
  - Branch target adder: ID/EX.pc + (ID/EX.imm « 2)

- MEM stage:
  - DMEM address from EX/MEM.alu_out
  - Store data from EX/MEM.store_data

- WB stage:
  - result = MemToReg ? MEM/WB.mem_rdata : MEM/WB.alu_out
  - write-back to rd if RegWrite=1 and rd≠0

3.3.2. Register File Semantics

Two combinational read ports (rs1, rs2), one synchronous write port (rd).

Write happens in WB on clock edge; ID reads during the same cycle observe prior state (typical), while EX forwarding feeds fresh values to consumers.

x0 hardwired to 0; writes to x0 are ignored.

Implementation notes: In simulation, adhere to deterministic read-after-write behavior (documented); forwarding handles most inter-instruction dependencies.

3.3.3. Immediate Generation

Per encoding:

- I/S/B formats: imm32 = sign_extend(imm16)
- B: target uses (imm « 2) to ensure word alignment
- J: target26 combined with PC[31:28] and two low zeros

Centralizing immediate extraction simplifies decode and reduces combinational duplication.

Troy University Resources

3.3.4. ALU and ALUOp Mapping

Baseline ALU ops includes ADD, SUB, AND, OR, SLT (signed).

ALUOp derivation:

- R-type: ALUOp from funct
- addi/lw/sw: ALUOp=ADD
- beq: ALUOp=SUB (zero-detect → equal)

Key signals: Zero flag (from SUB) supports beq. SLT uses signed compare; result is 0 or 1

3.3.5. Next-PC Logic

Inputs to next-PC mux:

- PC+4 (default sequential)
- branch_target (if Branch && branch_taken)
- jump_target (if Jump)
- reset_vector (on reset)

Ordering: Reset has highest priority, then branch/jump redirects, then sequential.

3.3.6. Forwarding Network

Select logic (per operand):

- 10: use EX/MEM.alu_out when exmem.RegWrite && exmem.rd == idex.rsX && exmem.rd≠0
- 01: use MEM/WB.result when memwb.RegWrite && memwb.rd == idex.rsX && memwb.rd≠0 and EX/MEM not chosen
- 00: use ID/EX.rsX_val

Special case: Load-use cannot forward from MEM because data is only available in MEM/WB; hazard unit inserts a bubble.

3.3.7. Hazard Unit Interactions

Inputs:

- idex.MemRead, idex.rd
- ifid.rs1, ifid.rs2
- exmem.RegWrite, exmem.rd
- memwb.RegWrite, memwb.rd
- exmem.branch_taken, exmem.jump (if jump resolved in EX)

Outputs:

- pc_write, ifid_we (stall gating)
- ifid_flush, idex_flush (flush/bubble)
- forwarding selects (fwd_a, fwd_b) computed in a forwarding sub-unit

Policies:

- Load-use stall if idex.MemRead && (idex.rd==ifid.rs1 || idex.rd==ifid.rs2) && idex.rd≠0
- On taken branch/jump: assert ifid_flush and set next PC to target

3.3.8. Memory Interfaces

IMEM:

- Combinational ROM model: instr = IMEM[PC]; single-cycle fetch
- Word-aligned accesses

DMEM:

- Synchronous RAM model:
  - Write: on posedge when MemWrite=1
  - Read: data available to MEM/WB (read-first or registered) next cycle
- Word-aligned accesses only in baseline

Troy University Resources

3.3.9. Control Bundle Definition and Propagation

In ID, decode forms a stable bundle that travels with the instruction:

- ID → ID/EX: full bundle (RegWrite, MemRead, MemWrite, Branch, Jump, ALUSrc, ALUOp, MemToReg)
- ID/EX → EX/MEM: drop Branch/Jump after resolution if you record branch_taken; keep Mem* and RegWrite
- EX/MEM → MEM/WB: keep RegWrite and MemToReg; data-path values carry alongside

This explicit propagation prevents temporal mismatches (e.g., instruction data paired with the wrong control).

3.4. Ideal Performance and Timing Constraints

This section analyzes throughput under ideal conditions, quantifies penalties from common hazards, and outlines timing constraints that guide clock selection and stage partitioning.

3.4.1. Ideal CPI and Pipeline Boundaries

In steady state with no hazards: One instruction retires per cycle → CPI ≈ 1. Fill/drain overhead is amortized over long runs [1], [2].

Boundaries: Register-to-register (pipeline register) timing per stage enables independent optimization and simplifies timing closure.

3.4.2. CPI Model with Hazard Penalties

A simple analytical model has an approximate CPI ≈ 1 + p_load_use·1 + p_taken·penalty_taken.

p_load_use: probability an instruction incurs a load-use stall

p_taken: probability a branch is taken

penalty_taken: cycles lost per taken branch (1–2 depending on flush depth)

Other stalls (e.g., memory wait) are zero in baseline

Example (array sum loop):

- Per iteration: 1 load-use stall; branches taken only at loop exit

- For large N: p_load_use ≈ 1/6–1/5 (depending on mix); p_taken ≈ 1/N → negligible
- Expected CPI ~ 1.17–1.20

Instrumentation: Count cycles, retired instructions (WB valid), stalls (load-use), and flushes (taken branches) to empirically validate CPI.

3.4.3. Stage Critical Paths and f_max

Clock period must satisfy the slowest stage path including register overhead:

Tclk ≥ t_clkq(prev_reg) + t_comb(stage) + t_setup(next_reg) + t_skew + margin

Likely dominant paths (baseline):

- IF: PC → adder (PC+4) → IMEM → IF/ID; often short if IMEM is combinational ROM
- ID: IF/ID → decode → regfile read → imm_gen → ID/EX; medium
- EX: ID/EX → forwarding muxes → ALU → EX/MEM; typically the longest (mux + adder/sub + compare)
- MEM: EX/MEM → DMEM → MEM/WB; can dominate if DMEM read is slow
- WB: MEM/WB → small mux → regfile write; generally short

3.4.4. Clocking and Reset Assumptions

Single clock domain; rising-edge triggered pipeline registers.

Synchronous reset preferred for simulation determinism; ensure reset deassertion is synchronous to avoid metastability.

Hold-time checks: Avoid combinational shortcuts between neighboring registers that could violate hold; synthesis usually inserts buffering if needed.

Skew/jitter: Keep low via simple clock distribution in simulation; in FPGA, rely on global buffers.

Troy University Resources

3.4.5. Timing Closure Guidance

Shorten EX path: Minimize forwarding mux depth/fan-in (2:1 per source is typical). Use fast adders for ALU operations (ripple is ok at low f, carry-lookahead for higher f).

Balance register fanout: Duplicate control nets or register them to reduce fanout-induced delay.

Keep decode shallow: Use one-hot/encoded ALUOp to avoid wide comparators in EX.

Constrain memories: If DMEM is slow, register address early and accept a one-cycle read; baseline already models read data in MEM/WB.

3.4.6. Memory Timing Variants and Impact

IMEM as combinational ROM: Fast fetch; critical path may shift to EX/MEM

DMEM as synchronous RAM: Read data visible one cycle after address register stage; this matches the pipeline and explains the load-use bubble

If adopting wait-states: Add ready/valid and hold downstream stages until DMEM ready; this introduces multi-cycle MEM and CPI > 1 without additional bubbles

3.4.7. Power/Performance Trade-offs

Forwarding network complexity vs. CPI: Removing forwarding simplifies hardware but forces frequent stalls; forwarding is a high-leverage optimization for throughput

Branch resolution point: Earlier resolution (ID) cuts taken-branch penalty but requires extra comparators and ID-stage forwarding; cost/benefit depends on branch frequency

Clock frequency: Increasing f_max by splitting long paths (e.g., moving branch compare earlier) can reduce CPI×Tclk product; balance complexity against educational goals

3.4.8. Verification Hooks for Timing Behavior

Assertions: Mutually exclusive MemRead/MemWrite. No write to x0. Stall implies PC and IF/ID hold; bubble implies ID/EX control cleared

Trace points: Per-stage valid bits (derived from control bundle) to distinguish bubbles from real instructions. Performance counters for cycles, retired, stalls, flushes

Waveform checklist: On load-use, see idex_flush=1, pc_write=0, ifid_we=0 for exactly one cycle. On taken branch, see ifid_flush=1 and PC redirect to target; no spurious reg/mem writes from flushed instructions

In summary, the baseline pipeline achieves CPI near 1 for straight-line code, with predictable penalties under common hazards. Timing closure is guided by keeping EX and MEM paths moderate, using forwarding to avoid unnecessary stalls, and preserving clean stage boundaries aided by well-formed pipeline registers and control propagation.

Troy University Resources

Chapter IV. Methodology: Hazard Handling and Control

In this chapter, we discuss the sources of hazards in the five-stage pipeline, the baseline policies used to resolve them, and the concrete hardware units that implement those policies. We focus on:

- Data hazards and operand forwarding,
- The load-use hazard and the precise one-cycle stall,
- Control hazards with a predict-not-taken policy and pipeline flush,
- The Hazard and Forwarding Units: interfaces, algorithms, and reference RTL.

4.1. Data Hazards and Forwarding

Data hazards happens when a planned instruction cannot execute because the data required is not available yet. In an in-order, single-issue pipeline with write-back in WB, the classic hazard is RAW (Read-After-Write). WAR and WAW do not occur because reads happen before writes in the pipeline order (register reads in ID and the only writes in WB) [1], [2].

4.1.1. RAW Examples

ALU→ALU dependency (forwardable)

- I1: add x3, x1, x2
- I2: sub x4, x3, x5

Without forwarding, I2 would read the old x3 in ID. With forwarding, EX selects x3 directly from EX/MEM (I1's result) or MEM/WB (I1's result one stage later).

ALU→Store-data dependency (forwardable)

- I1: add x5, x1, x2
- I2: sw x5, 0(x3)

The store's data path (rs2 path) must observe the most recent x5. Forward EX/MEM.alu_out (I1) into the EX store-data mux for I2.

Load→ALU dependency (requires stall)

- I1: lw x6, 0(x1)
- I2: add x7, x6, x2

The load result appears in MEM/WB. EX of I2 occurs too early; a one-cycle bubble is necessary.

4.1.2. Forwarding Sources and Priority

At EX, each operand srcA (rs1) and srcB (rs2) can come from:

- 00: ID/EX.rsX_val (no forward)
- 10: EX/MEM.alu_out (freshest result, highest priority)
- 01: MEM/WB.result (the value to be written back this cycle)

Rules:

- If EX/MEM.RegWrite=1 and EX/MEM.rd≠0 and EX/MEM.rd==ID/EX.rsX → select EX/MEM (10).
- Else if MEM/WB.RegWrite=1 and MEM/WB.rd≠0 and MEM/WB.rd==ID/EX.rsX → select MEM/WB (01).
- Else → 00.

Corner cases:

- Never forward a value targeting rd=0; x0 is always 0.
- For store data (rs2 path), apply the same forwarding selection into the store_data register captured in EX/MEM.

4.1.3. Why WAR and WAW Don't Occur

WAR (Write-After-Read): writes only in WB; reads in ID. The reader is always older and reads before the writer writes → impossible.

WAW (Write-After-Write): single write stage; in order issue → later instruction cannot write before the earlier one.

Troy University Resources

4.1.4. Verification Notes

Create dependency ladders (chains of back-to-back ALU ops) to ensure forwarding chooses EX/MEM over MEM/WB.

Test store-data forwarding by computing the value immediately before the store.

Assert that forwarding never routes from a pipeline stage whose rd=0.

4.2. Load-Use Hazard and Stall

The load-use hazard occurs when a load's destination register is used by another instruction when it is not ready yet. In a baseline single-cycle DMEM read model (data available for WB after MEM), forwarding cannot supply the value in time for the consumer's EX [1], [2].

4.2.1. Timing Intuition

Let I1 be a load (lw rd, …) and I2 be an instruction that reads rd:

- Cycle k: I1 in EX computes address
- Cycle k+1: I1 in MEM reads DMEM (data available to MEM/WB)
- Cycle k+1: I2 would be in EX needing rd now → data not yet ready
- Solution: stall one cycle (insert a bubble) so that I2's EX occurs at cycle k+2, where MEM/WB has the data and forwarding 01 can supply it.

4.2.2. Detection Rule

In ID (considering what is in ID/EX and IF/ID):

- If ID/EX.MemRead == 1 (I1 is a load),
- And ID/EX.rd != 0,
- And (ID/EX.rd == IF/ID.rs1 OR ID/EX.rd == IF/ID.rs2),
- Then assert a load-use stall.

4.2.3. Stall/Flush Actions (One-Cycle Bubble)

When the condition is true for one cycle:

- pc_write = 0 (hold PC)
- ifid_we = 0 (hold IF/ID; I2 stays in ID)
- idex_flush = 1 (turn the would-be instruction in EX into a NOP bubble)

This guarantees:

- Exactly one bubble is inserted.
- Producer (I1) proceeds to MEM; consumer (I2) waits in ID.
- On the next cycle, I2 advances to EX and can use MEM/WB forwarding.

4.2.4. Corner Cases

- If the consumer writes x0: still a hazard if it reads the loaded value; but writing x0 later has no effect on state.
- If the instruction between the load and consumer doesn't read rd, no stall; forwarding covers subsequent consumers.
- If a branch is in EX and is taken during the same cycle as a load-use stall request, control flush takes priority over stall for younger instructions (see 4.3.4).

4.2.5. Verification

- Construct sequences: lw → add, lw → sw, with and without intervening instructions.
- Assert that exactly one cycle of pc_write=0/ifid_we=0/idex_flush=1 occurs on each load-use event.
- Ensure no spurious stalls when rd==0 or when IF/ID does not read the loaded register.

Troy University Resources

4.3. Control Hazards and Flush/Branch Policy

Control hazards (or Branch hazards) happens when the next PC depends on the outcome of a branch or jump whose condition/target is not yet known. The baseline policy is predict-not-taken with branch resolution in EX [1], [2].

4.3.1. Predict-Not-Taken Baseline

- Fetch assumes sequential flow (PC+4).
- Branches and jumps are resolved in EX:
  - beq compares rs1 and rs2 (via SUB+zero) and computes target = ID.pc + (imm«2).
  - j forms {PC[31:28], target26, 2'b00}.

If the branch is not taken:

- No penalty; the speculatively fetched instruction is correct.

If the branch is taken or a jump occurs:

- Flush younger instructions that were fetched/decode under the wrong path.
- Redirect PC to the target on the next cycle.

4.3.2. What to Flush

At minimum:

- Flush IF/ID (replace with NOP), because the instruction in IF/ID was fetched using the wrong next PC when a redirect occurs in EX.
- Some implementations also flush ID/EX if the branch decision is made in EX and the ID-stage instruction is already decoded under a wrong control path (more conservative but simpler for education).

Baseline (simple and safe):

- ifid_flush = 1 on taken branch/jump.
- Optional: idex_flush = 1 to ensure no partially decoded wrong-path effects (conservative two-bubble penalty).

Penalty:

- 1 cycle if only IF/ID is flushed (and redirect happens quickly).
- 2 cycles if both IF/ID and ID/EX are flushed.

4.3.3. Branch/Jump Target Calculation and Ordering

- Next-PC mux prioritization:
    i. Reset
    ii. Branch/jump redirect
    iii. Sequential PC+4
- Targets:
    i. beq: PC_ID + (SE(off16)«2), ensuring word alignment.
    ii. j: {PC[31:28], target26, 2'b00}, placing the target in the current 256MB region.

4.3.4. Interactions with Stalls

Priority policy per cycle:

- reset > flush (control redirect) > stall (data hazard) > normal.

Case: load-use stall and taken branch in the same cycle:

- The control redirect takes precedence; perform flush and PC redirect.
- Data-stall signals for that cycle do not prevent redirect; the younger wrong-path instruction is annulled anyway.

4.3.5. Optional Enhancements (Out-of-Baseline)

- Early branch resolution in ID:
    – Compare rs1_val/rs2_val in ID (requires ID-stage forwarding).
    – Reduces taken-branch penalty by one cycle.

Troy University Resources

- Static/dynamic branch prediction:
  - Always-taken for backward branches (loops), simple one-bit predictor, etc.
- Branch delay slots:
  - Not used in this design for clarity; would require ISA-visible scheduling semantics.

4.3.6. Verification

- Sequences with both taken and not-taken beq.
- Mix branches with dependent operations to ensure forwarding and flush do not conflict.
- Assert that flushed instructions do not write registers or memory (NOP control bundle when flush asserted).

4.4. Hazard & Forwarding Unit

This section specifies the interfaces, algorithms, and reference RTL for the combined Hazard Unit (stall/flush) and the Forwarding Unit (EX operand selection). They may be implemented as one module or as two cooperating modules.

4.4.1. Interfaces

Forwarding Unit (EX operand selection):

- Inputs:
  - idex_rs1[4:0], idex_rs2[4:0]
  - exmem_regwrite, exmem_rd[4:0]
  - memwb_regwrite, memwb_rd[4:0]
- Outputs:
  - fwd_a[1:0], fwd_b[1:0] // 2-bit selects: 00=ID/EX, 10=EX/MEM, 01=MEM/WB

Hazard Unit (stalls and flushes):

- Inputs:
  - idex_memread // indicates the instruction in EX is a load (producer)
  - idex_rd[4:0]
  - ifid_rs1[4:0], ifid_rs2[4:0]
  - ex_branch_taken // EX resolved taken-branch (or jump)
- Outputs:
  - pc_write // 0 → hold PC
  - ifid_we // 0 → hold IF/ID
  - idex_flush // 1 → insert bubble into EX
  - ifid_flush // 1 → annul IF/ID

Note: Jumps resolved in EX can reuse ex_branch_taken and supply a corresponding target signal to the next-PC mux (the control path provides the target data).

4.4.2. Algorithms

Forwarding selection (per operand X ∈ {rs1, rs2}):

```
fwd_X = 2'b00
if (exmem_regwrite && exmem_rd != 0 && exmem_rd == idex_rsX) fwd_X
= 2'b10
else if (memwb_regwrite && memwb_rd != 0 && memwb_rd == idex_rsX)
fwd_X = 2'b01
Load-use stall detection:
load_use = idex_memread && (idex_rd != 0) &&
           ( (idex_rd == ifid_rs1) || (idex_rd == ifid_rs2) )
if (load_use) {
  pc_write   = 0
  ifid_we    = 0
  idex_flush = 1
} else {
  pc_write   = 1
```


Troy University Resources

```
  ifid_we    = 1
  idex_flush = 0
}
Flush on taken branch/jump:
if (ex_branch_taken) {
  ifid_flush = 1
  // optional: idex_flush = 1 (for conservative two-bubble policy)
} else {
  ifid_flush = 0
}
```

Priority: reset > flush > stall > normal.

4.4.3. Reference RTL: Forwarding Unit (Verilog)

```
// forwarding_unit.v
module forwarding_unit (
  input  logic       exmem_regwrite,
  input  logic [4:0] exmem_rd,
  input  logic       memwb_regwrite,
  input  logic [4:0] memwb_rd,
  input  logic [4:0] idex_rs1,
  input  logic [4:0] idex_rs2,
  output logic [1:0] fwd_a, // 00:ID/EX, 10:EX/MEM, 01:MEM/WB
  output logic [1:0] fwd_b
);
  always_comb begin
    fwd_a = 2'b00;
    fwd_b = 2'b00;
    // rs1
    if (exmem_regwrite && (exmem_rd != 5'd0) && (exmem_rd ==
idex_rs1))
      fwd_a = 2'b10;
    else if (memwb_regwrite && (memwb_rd != 5'd0) && (memwb_rd ==
idex_rs1))
      fwd_a = 2'b01;
    // rs2
    if (exmem_regwrite && (exmem_rd != 5'd0) && (exmem_rd ==
idex_rs2))
      fwd_b = 2'b10;
```

```
    else if (memwb_regwrite && (memwb_rd != 5'd0) && (memwb_rd ==
idex_rs2))
      fwd_b = 2'b01;
  end
endmodule
```

Notes:

- EX/MEM has priority over MEM/WB to ensure the freshest value is used.
- No forwarding when destination is x0.

4.4.4. Reference RTL: Hazard Unit (Verilog)

```
// hazard_unit.v
module hazard_unit (
  input  logic       idex_memread,     // load in EX
  input  logic [4:0] idex_rd,          // load destination
  input  logic [4:0] ifid_rs1,
  input  logic [4:0] ifid_rs2,
  input  logic       ex_branch_taken,  // branch/jump resolved in
EX
  output logic       pc_write,         // gate PC update
  output logic       ifid_we,          // gate IF/ID write
  output logic       idex_flush,       // insert bubble into EX
  output logic       ifid_flush        // flush IF/ID on redirect
);
  logic load_use;
  assign load_use = idex_memread && (idex_rd != 5'd0) &&
                    ((idex_rd == ifid_rs1) || (idex_rd == ifid_rs2));

  always_comb begin
    // Defaults: no stall, no flush
    pc_write   = 1'b1;
    ifid_we    = 1'b1;
    idex_flush = 1'b0;
    ifid_flush = 1'b0;
    // Control redirect has priority over data stall
    if (ex_branch_taken) begin
      ifid_flush = 1'b1;
```


Troy University Resources

```
      // Optionally: idex_flush = 1'b1; // conservative two-bubble
policy
    end
    // Load-use stall (only if no control redirect)
    else if (load_use) begin
      pc_write   = 1'b0;
      ifid_we    = 1'b0;
      idex_flush = 1'b1;
    end
  end
endmodule
```

Notes:

- The unit asserts exactly one cycle of stall for each detected load-use hazard.
- When both a control redirect and a load-use stall would occur, the flush takes precedence.

4.4.5. Integration Notes

Place forwarding multiplexers at the ALU input and at the store-data input path in EX.

Route RegWrite/rd from EX/MEM and MEM/WB to the forwarding unit.

Route MemRead/rd from ID/EX and rs1/rs2 from IF/ID to the hazard unit.

Gate PC update and IF/ID register with pc_write and ifid_we; drive ifid_flush and idex_flush into the respective pipeline registers.

4.4.6. Verification and Assertions

- Forwarding correctness:
  - For every back-to-back RAW dependency, check that ALU inputs select EX/MEM (or MEM/WB if appropriate) and that results match a golden stepper.
- Load-use stall:

  - Assert that pc_write==0, ifid_we==0, idex_flush==1 for exactly one cycle per event.
  - Confirm no additional bubbles are introduced.

- Control flush:
  - On taken branch/jump, assert ifid_flush==1 and confirm no architectural side effects from the flushed instructions.

- Safety properties:
  - Never write x0 (RegWrite && rd==0 → no write).
  - MemRead and MemWrite are not both 1 in the same instruction.
  - After reset, the first N cycles perform only benign operations (NOP bundle in pipeline registers).

In summary, the baseline design resolves data hazards through a minimal and effective forwarding network, handles the unavoidable load-use dependency with a single, precisely controlled stall, and manages control hazards using a clear predict-not-taken policy with targeted flush. The Hazard and Forwarding Units are compact, easy to verify, and encapsulate the pipeline control decisions that are central to correctness and throughput.

Troy University Resources

Chapter V. Methodology: Verilog Design and Implementation

This chapter describes the RTL structure, interfaces, coding conventions, and representative modules used to implement the five-stage pipelined CPU. The goal is a clean and teachable codebase: modular, synthesizable, and easy to verify with standard open-source simulators.

5.1. Module Structure and Functional Decomposition

The CPU is decomposed into cohesive modules with clear responsibilities and interfaces. This supports unit testing, reusability, and simplifies timing closure and debugging.

5.1.1. Top-Level Organization

cpu_top.v

- Integrates stages, pipeline registers, control signals, hazard/forwarding units, and memory interfaces.
- Exposes simple IMEM/DMEM ports for testbench connection or on-chip memory binding in FPGA experiments.

Pipeline stages and blocks: IF, ID, EX, MEM and WB components responding to a stage of the pipeline process.

Control and hazard

- control.v: decodes opcode/funct to generate control bundle.
- hazard_unit.v: detects load-use, manages stalls and flushes.
- forwarding_unit.v: selects EX operand sources.

Utilities

- regfile.v: 32×32 2R1W register file with x0 hardwired to zero.
- alu.v: integer ALU: add/sub/and/or/slt (+ zero flag).
- imm_gen.v: immediate extraction and sign-extension.

Memory models (for simulation)

- imem.v: ROM-like instruction memory with $readmemh.

- dmem.v: synchronous RAM data memory with single-cycle read (read-first) and posedge write.

5.1.2. Design Goals

- Modular boundaries match stage responsibilities.
- Minimal yet sufficient control bundle propagated with each instruction.
- Explicit stall/flush enables on pipeline registers.
- Forwarding first, stall only when necessary (load-use).
- Harvard memories to avoid structural hazards.

5.2. Interfaces, Synchronization, and Signal Conventions

The RTL follows consistent conventions to reduce ambiguity and facilitate tool interoperability.

5.2.1. Clocking and Reset

Single clock domain (clk), rising-edge triggered.

Synchronous active-high reset (rst).

Pipeline register control priority per cycle: reset > flush > stall(we=0) > normal.

5.2.2. Interface Summary

IMEM (fetch)

- imem_addr[31:0] — word-aligned PC
- imem_rdata[31:0] — instruction word (combinational ROM)

DMEM (data)

- dmem_addr[31:0] — byte address (word-aligned for baseline)
- dmem_wdata[31:0] — store data
- dmem_rdata[31:0] — load data (latched into MEM/WB)
- dmem_we — write enable

Troy University Resources

- dmem_re — read enable
- dmem_be[3:0] — byte enables (optional; not used in baseline)

Register file

- raddr1, raddr2 → rdata1, rdata2 (combinational reads)
- waddr, wdata, we (WB stage, posedge write)

Stage/register prefixes: ifid_, idex_, exmem_, memwb_

5.2.3. Combinational vs. Sequential Logic

Combinational: always_comb (or always @*) with blocking assignments (=).

Sequential: always_ff @(posedge clk) with non-blocking assignments (<=).

Avoid latches by providing defaults in combinational blocks.

5.2.4. Architectural Semantics

x0 is hardwired to 0; writes to x0 are ignored.

ALU and control produce deterministic results per instruction.

Load-use hazard enforces exactly one bubble; forwarding otherwise avoids RAW stalls.

Predict-not-taken policy; taken branches/jumps flush IF/ID (optionally ID/EX).

5.3. Sample Codes (Register File, ALU, Control, Hazard/Forwarding)

This section provides compact, illustrative RTL for key blocks. These are synthesizable and simulation-friendly.

5.3.1. Register File (2R1W, x0=0)

```
// rtl/regfile.sv
module regfile (
  input  logic        clk,
  input  logic        we,
  input  logic [4:0]  raddr1,
  input  logic [4:0]  raddr2,
  input  logic [4:0]  waddr,
  input  logic [31:0] wdata,
```

```
  output logic [31:0] rdata1,
  output logic [31:0] rdata2
);
  logic [31:0] regs [31:0];
  // combinational reads
  assign rdata1 = (raddr1 == 5'd0) ? 32'd0 : regs[raddr1];
  assign rdata2 = (raddr2 == 5'd0) ? 32'd0 : regs[raddr2];
  // synchronous write
  always_ff @(posedge clk) begin
    if (we && (waddr != 5'd0))
      regs[waddr] <= wdata;
  end
endmodule
```

Notes:

- Two combinational read ports, one synchronous write.
- x0 (index 0) reads as 0, ignores writes.

5.3.2. ALU (ADD, SUB, AND, OR, SLT)

```
// rtl/alu.sv
module alu (
  input  logic [31:0] a,
  input  logic [31:0] b,
  input  logic [2:0]  alu_op,   // encoding below
  output logic [31:0] y,
  output logic        zero
);
  // ALUOp encoding (example):
  localparam ALU_ADD = 3'd0;
  localparam ALU_SUB = 3'd1;
  localparam ALU_AND = 3'd2;
  localparam ALU_OR  = 3'd3;
  localparam ALU_SLT = 3'd4;
  logic [31:0] subres;
  assign subres = a - b;
  assign zero   = (y == 32'd0);
  always_comb begin
    unique case (alu_op)
```


Troy University Resources

```
      ALU_ADD: y = a + b;
      ALU_SUB: y = subres;
      ALU_AND: y = a & b;
      ALU_OR : y = a | b;
      ALU_SLT: y = ($signed(a) < $signed(b)) ? 32'd1 : 32'd0;
      default: y = 32'd0;
    endcase
  end
endmodule
```

5.3.3. Immediate Generator

```
// rtl/imm_gen.sv
module imm_gen (
  input  logic [31:0] instr,
  input  logic [2:0]  imm_sel, // 0:R,1:I,2:S,3:B,4:J
  output logic [31:0] imm
);
  logic [15:0] imm16;
  assign imm16 = instr[15:0];
  always_comb begin
    unique case (imm_sel)
      3'd1: imm = {{16{imm16[15]}}, imm16}; // I/lw/addi
      3'd2: imm = {{16{imm16[15]}}, imm16}; // S/sw
      3'd3: imm = {{16{imm16[15]}}, imm16} << 2; // B/beq word
offset -> byte
      3'd4: imm = { instr[25:0], 2'b00 }; // J target lower bits;
upper PC[31:28] added in EX
      default: imm = 32'd0; // R-type etc.
    endcase
  end
endmodule
```

5.3.4. Control Decoder (Toy RISC Encoding)

```
// rtl/control.sv
module control (
  input  logic [31:0] instr,
  output logic        RegWrite,
  output logic        MemRead,
```

```
  output logic        MemWrite,
  output logic        Branch,
  output logic        Jump,
  output logic        MemToReg,
  output logic        ALUSrc,
  output logic [2:0]  ALUOp,
  output logic [2:0]  ImmSel
);
  logic [5:0] opcode;
  logic [10:0] funct;
  assign opcode = instr[31:26];
  assign funct  = instr[10:0];
  // opcode encodings (illustrative)
  localparam OPC_RTYPE = 6'b000000;
  localparam OPC_ADDI  = 6'b000001;
  localparam OPC_LW    = 6'b000010;
  localparam OPC_SW    = 6'b000011;
  localparam OPC_BEQ   = 6'b000100;
  localparam OPC_J     = 6'b000101;
  // ALUOp encodings (match alu.sv)
  localparam ALU_ADD = 3'd0, ALU_SUB=3'd1, ALU_AND=3'd2, ALU_OR=3'
d3, ALU_SLT=3'd4;
  // ImmSel encodings (match imm_gen.sv)
  localparam IMM_R=3'd0, IMM_I=3'd1, IMM_S=3'd2, IMM_B=3'd3, IMM_J
=3'd4;
  // funct for R-type
  localparam FUNCT_ADD=11'd0, FUNCT_SUB=11'd1, FUNCT_AND=11'd2,
FUNCT_OR=11'd3, FUNCT_SLT=11'd4;
  always_comb begin
    // defaults = NOP-like
    RegWrite = 1'b0;
    MemRead  = 1'b0;
    MemWrite = 1'b0;
    Branch   = 1'b0;
    Jump     = 1'b0;
    MemToReg = 1'b0;
    ALUSrc   = 1'b0;
    ALUOp    = ALU_ADD;
    ImmSel   = IMM_R;
    unique case (opcode)
```


Troy University Resources

```
      OPC_RTYPE: begin
        RegWrite = 1'b1;
        case (funct)
          FUNCT_ADD: ALUOp = ALU_ADD;
          FUNCT_SUB: ALUOp = ALU_SUB;
          FUNCT_AND: ALUOp = ALU_AND;
          FUNCT_OR : ALUOp = ALU_OR;
          FUNCT_SLT: ALUOp = ALU_SLT;
          default:   RegWrite = 1'b0; // undefined -> NOP
        endcase
      end
      OPC_ADDI: begin
        RegWrite = 1'b1; ALUSrc = 1'b1; ImmSel = IMM_I; ALUOp =
ALU_ADD;
      end
      OPC_LW: begin
        RegWrite = 1'b1; MemRead = 1'b1; MemToReg = 1'b1; ALUSrc=1'
b1; ImmSel=IMM_I; ALUOp=ALU_ADD;
      end
      OPC_SW: begin
        MemWrite = 1'b1; ALUSrc = 1'b1; ImmSel = IMM_S; ALUOp=
ALU_ADD;
      end
      OPC_BEQ: begin
        Branch = 1'b1; ImmSel = IMM_B; ALUOp = ALU_SUB; // use SUB+
zero for equality
      end
      OPC_J: begin
        Jump = 1'b1; ImmSel = IMM_J; // target computed with PC
upper bits in EX
      end
      default: ; // NOP
    endcase
  end
endmodule
```

5.3.5. Forwarding Unit

```
// rtl/forwarding_unit.sv
module forwarding_unit (
  input  logic       exmem_regwrite,
  input  logic [4:0] exmem_rd,
  input  logic       memwb_regwrite,
  input  logic [4:0] memwb_rd,
  input  logic [4:0] idex_rs1,
  input  logic [4:0] idex_rs2,
  output logic [1:0] fwd_a, // 00:ID/EX, 10:EX/MEM, 01:MEM/WB
  output logic [1:0] fwd_b
);
  always_comb begin
    fwd_a = 2'b00; fwd_b = 2'b00;
    // rs1
    if (exmem_regwrite && exmem_rd!=5'd0 && exmem_rd==idex_rs1)
fwd_a = 2'b10;
    else if (memwb_regwrite && memwb_rd!=5'd0 && memwb_rd==idex_rs1
) fwd_a = 2'b01;
    // rs2
    if (exmem_regwrite && exmem_rd!=5'd0 && exmem_rd==idex_rs2)
fwd_b = 2'b10;
    else if (memwb_regwrite && memwb_rd!=5'd0 && memwb_rd==idex_rs2
) fwd_b = 2'b01;
  end
endmodule
```

5.3.6. Hazard Unit (Load-Use Stall and Flush on Redirect)

```
// rtl/hazard_unit.sv
module hazard_unit (
  input  logic       idex_memread,     // load in EX
  input  logic [4:0] idex_rd,          // load destination
  input  logic [4:0] ifid_rs1,
  input  logic [4:0] ifid_rs2,
  input  logic       ex_branch_taken,  // branch/jump resolved in
EX
  output logic       pc_write,         // gate PC update
  output logic       ifid_we,          // gate IF/ID write
```


Troy University Resources

```
  output logic       idex_flush,       // insert bubble into EX
  output logic       ifid_flush        // flush IF/ID on redirect
);
  logic load_use;
  assign load_use = idex_memread && (idex_rd!=5'd0) &&
                   ((idex_rd==ifid_rs1) || (idex_rd==ifid_rs2));
  always_comb begin
    // defaults
    pc_write   = 1'b1;
    ifid_we    = 1'b1;
    idex_flush = 1'b0;
    ifid_flush = 1'b0;
    // control redirect takes priority
    if (ex_branch_taken) begin
      ifid_flush = 1'b1;
    end else if (load_use) begin
      pc_write   = 1'b0;
      ifid_we    = 1'b0;
      idex_flush = 1'b1;
    end
  end
endmodule
```

5.3.7. ALU Operand Muxing with Forwarding (EX Stage Snippet)

```
// rtl/ex_stage_mux.sv (snippet)
module ex_src_mux (
  input  logic [31:0] idex_rs1_val,
  input  logic [31:0] idex_rs2_val,
  input  logic [31:0] exmem_alu_out,
  input  logic [31:0] memwb_result,
  input  logic [1:0]  fwd_a,
  input  logic [1:0]  fwd_b,
  input  logic        ALUSrc,            // select imm for srcB
when 1
  input  logic [31:0] idex_imm,
  output logic [31:0] srcA,
  output logic [31:0] srcB_raw
);
  always_comb begin
```

```
    unique case (fwd_a)
      2'b10: srcA = exmem_alu_out;
      2'b01: srcA = memwb_result;
      default: srcA = idex_rs1_val;
    endcase
    unique case (fwd_b)
      2'b10: srcB_raw = exmem_alu_out;
      2'b01: srcB_raw = memwb_result;
      default: srcB_raw = idex_rs2_val;
    endcase
  end
  // final ALU srcB selection (imm vs. register)
  // performed outside with: srcB = ALUSrc ? idex_imm : srcB_raw;
endmodule
```

5.4. Memory Organization (IMEM/DMEM) and Pipeline Registers

This section details the simulation memories and the reusable pipeline register templates.

5.4.1. Instruction Memory (IMEM) — ROM Model

```
// rtl/imem.sv
module imem (
  input  logic [31:0] addr,     // word-aligned PC
  output logic [31:0] rdata
);
  parameter DEPTH_WORDS = 1024;
  logic [31:0] rom [0:DEPTH_WORDS-1];
  // word index: drop bottom 2 bits
  wire [31:2] widx = addr[31:2];
  // combinational read (ROM semantics)
  assign rdata = rom[widx];
  // initialization with $readmemh in testbench or here via
parameterized file
  // initial $readmemh("prog/test.hex", rom);
endmodule
```

Notes:

- Combinational read supports single-cycle IF fetch.

Troy University Resources

- Use $readmemh in testbench setup for reproducibility.

5.4.2. Data Memory (DMEM) — Synchronous RAM

```
// rtl/dmem.sv
module dmem (
  input  logic        clk,
  input  logic        we,
  input  logic        re,
  input  logic [3:0]  be,       // byte enables (unused for word-
only baseline: set to 4'b1111)
  input  logic [31:0] addr,     // byte address (word-aligned for
baseline)
  input  logic [31:0] wdata,
  output logic [31:0] rdata
);
  parameter DEPTH_WORDS = 1024;
  logic [31:0] ram [0:DEPTH_WORDS-1];
  wire [31:2] widx = addr[31:2];
  // simple synchronous behavior:
  // - read-first: rdata updated on read enable (registered one
cycle later if desired)
  // - write on posedge when we=1
  always_ff @(posedge clk) begin
    if (we) begin
      // baseline assumes word stores
      ram[widx] <= wdata;
    end
    if (re) begin
      rdata <= ram[widx];
    end
  end
  // initial $readmemh("prog/data.hex", ram);
endmodule
```

Notes:

- Read-first behavior aligns with pipeline: load data available to MEM/WB.

- For subword ops, honor be[3:0] and add sign/zero extension logic (out-of-baseline).

5.4.3. Pipeline Register Template

```
// rtl/pipe_reg.sv
module pipe_reg #(
  parameter W = 32
)(
  input  logic        clk,
  input  logic        rst,
  input  logic        we,     // 1: latch new data; 0: hold
  input  logic        flush,  // 1: load NOP
  input  logic [W-1:0] d,
  input  logic [W-1:0] nop,   // benign NOP value
  output logic [W-1:0] q
);
  always_ff @(posedge clk) begin
    if (rst)       q <= nop;
    else if (flush) q <= nop;
    else if (we)   q <= d;
  end
endmodule
```

Specialized pipeline registers can pack multiple signals into a single bus (struct/bundle), or instantiate several pipe_reg instances for separate fields. For pedagogical clarity, many designs use explicit named signals per field.

5.4.4. IF/ID Register (Example)

```
// rtl/if_id_reg.sv
module if_id_reg (
  input  logic        clk,
  input  logic        rst,
  input  logic        we,        // stall gate
  input  logic        flush,     // flush on redirect
  input  logic [31:0] instr_in,
  input  logic [31:0] pc_in,
  input  logic [31:0] pc4_in,
```


Troy University Resources

```
  output logic [31:0] instr_out,
  output logic [31:0] pc_out,
  output logic [31:0] pc4_out
);
  localparam [31:0] NOP_INSTR = 32'h0000_0000; // decode-safe NOP
  always_ff @(posedge clk) begin
    if (rst) begin
      instr_out <= NOP_INSTR; pc_out <= 32'd0; pc4_out <= 32'd0;
    end else if (flush) begin
      instr_out <= NOP_INSTR; pc_out <= 32'd0; pc4_out <= 32'd0;
    end else if (we) begin
      instr_out <= instr_in;  pc_out  <= pc_in; pc4_out <= pc4_in;
    end
  end
endmodule
```

5.4.5. ID/EX Register (Example)

```
// rtl/id_ex_reg.sv
module id_ex_reg (
  input  logic        clk,
  input  logic        rst,
  input  logic        flush, // bubble insertion on load-use
  input  logic [31:0] rs1_val_in,
  input  logic [31:0] rs2_val_in,
  input  logic [31:0] imm_in,
  input  logic [4:0]  rs1_in, rs2_in, rd_in,
  input  logic [31:0] pc_in, pc4_in,
  input  logic        RegWrite_in, MemRead_in, MemWrite_in,
  input  logic        Branch_in, Jump_in, MemToReg_in, ALUSrc_in,
  input  logic [2:0]  ALUOp_in,
  output logic [31:0] rs1_val_out, rs2_val_out, imm_out,
  output logic [4:0]  rs1_out, rs2_out, rd_out,
  output logic [31:0] pc_out, pc4_out,
  output logic        RegWrite_out, MemRead_out, MemWrite_out,
  output logic        Branch_out, Jump_out, MemToReg_out,
ALUSrc_out,
  output logic [2:0]  ALUOp_out
);
```

```
  // On flush, clear control to NOP; data fields may be don't-cares
 or zeroed
  always_ff @(posedge clk) begin
    if (rst || flush) begin
      rs1_val_out<=32'd0; rs2_val_out<=32'd0; imm_out<=32'd0;
      rs1_out<=5'd0; rs2_out<=5'd0; rd_out<=5'd0;
      pc_out<=32'd0; pc4_out<=32'd0;
      RegWrite_out<=1'b0; MemRead_out<=1'b0; MemWrite_out<=1'b0;
      Branch_out<=1'b0; Jump_out<=1'b0; MemToReg_out<=1'b0;
ALUSrc_out<=1'b0;
      ALUOp_out<=3'd0;
    end else begin
      rs1_val_out<=rs1_val_in; rs2_val_out<=rs2_val_in; imm_out<=
imm_in;
      rs1_out<=rs1_in; rs2_out<=rs2_in; rd_out<=rd_in;
      pc_out<=pc_in; pc4_out<=pc4_in;
      RegWrite_out<=RegWrite_in; MemRead_out<=MemRead_in;
MemWrite_out<=MemWrite_in;
      Branch_out<=Branch_in; Jump_out<=Jump_in; MemToReg_out<=
MemToReg_in; ALUSrc_out<=ALUSrc_in;
      ALUOp_out<=ALUOp_in;
    end
  end
endmodule
```

EX/MEM and MEM/WB registers follow similarly, carrying the relevant data/control fields for their consumer stages.

5.4.6. Top-Level Skeleton (cpu_top) — Selected Highlights

```
// rtl/cpu_top.sv (selected highlights to show integration)
module cpu_top (
  input  logic        clk,
  input  logic        rst,
  // IMEM
  output logic [31:0] imem_addr,
  input  logic [31:0] imem_rdata,
  // DMEM
  output logic [31:0] dmem_addr,
  output logic [31:0] dmem_wdata,
  input  logic [31:0] dmem_rdata,
```


Troy University Resources

```
  output logic        dmem_we,
  output logic        dmem_re
);
  // IF stage
  logic [31:0] pc_q, pc_n, pc4_if;
  logic        pc_write, ifid_we, ifid_flush;
  assign pc4_if = pc_q + 32'd4;
  always_ff @(posedge clk) begin
    if (rst) pc_q <= 32'd0;
    else if (pc_write) pc_q <= pc_n;
  end
  assign imem_addr = pc_q;
  // IF/ID
  logic [31:0] ifid_instr, ifid_pc, ifid_pc4;
  if_id_reg U_ifid (.clk, .rst, .we(ifid_we), .flush(ifid_flush),
                    .instr_in(imem_rdata), .pc_in(pc_q), .pc4_in(
pc4_if),
                    .instr_out(ifid_instr), .pc_out(ifid_pc), .
pc4_out(ifid_pc4));
  // ID: decode, regfile, imm_gen
  logic RegWrite_d, MemRead_d, MemWrite_d, Branch_d, Jump_d,
MemToReg_d, ALUSrc_d;
  logic [2:0] ALUOp_d, ImmSel_d;
  control U_ctrl (.instr(ifid_instr), .*,
                  .RegWrite(RegWrite_d), .MemRead(MemRead_d), .
MemWrite(MemWrite_d),
                  .Branch(Branch_d), .Jump(Jump_d), .MemToReg(
MemToReg_d),
                  .ALUSrc(ALUSrc_d), .ALUOp(ALUOp_d), .ImmSel(
ImmSel_d));
  logic [4:0] rs1 = ifid_instr[25:21];
  logic [4:0] rs2 = ifid_instr[20:16];
  logic [4:0] rd_r = ( (ifid_instr[31:26]==6'b000000) ? ifid_instr
[15:11] : ifid_instr[20:16] );
  logic [31:0] rf_rdata1, rf_rdata2, wb_result;
  regfile U_rf (.clk, .we(RegWrite_wb), .raddr1(rs1), .raddr2(rs2),
                .waddr(rd_wb), .wdata(wb_result), .rdata1(rf_rdata1
), .rdata2(rf_rdata2));
  logic [31:0] imm_d;
```

```
  imm_gen U_imm (.instr(ifid_instr), .imm_sel(ImmSel_d), .imm(imm_d
));
  // Hazard detection (load-use)
  logic idex_flush, ex_branch_taken; // ex_branch_taken from EX
  hazard_unit U_haz (.idex_memread(MemRead_ex), .idex_rd(rd_ex),
                     .ifid_rs1(rs1), .ifid_rs2(rs2),
                     .ex_branch_taken(ex_branch_taken),
                     .pc_write(pc_write), .ifid_we(ifid_we), .
idex_flush(idex_flush), .ifid_flush(ifid_flush));
  // ID/EX
  // ... pack and pass decoded signals and operands
  // EX: forwarding, ALU, branch target/decision
  // MEM: DMEM interface
  // WB: writeback mux -> wb_result
  // Next PC select (priority: reset handled above, then redirect)
  logic [31:0] branch_target_ex, jump_target_ex;
  logic        jump_ex;
  always_comb begin
    if (ex_branch_taken) pc_n = branch_target_ex;
    else if (jump_ex)    pc_n = jump_target_ex;
    else                 pc_n = pc4_if;
  end
endmodule
```

Notes:

- The skeleton shows how IF/ID is fed and how hazard signals gate PC and IF/ID.
- Full wiring includes ID/EX, EX/MEM, MEM/WB signals, forwarding selects, and WB mux (MemToReg).

By adhering to this structure, the RTL remains approachable for students while being sufficiently realistic to demonstrate timing and hazard behavior of a classical five-stage RISC pipeline. The code provided is directly usable in Icarus Verilog or Verilator, and can be extended to explore more advanced microarchitectural features.

Troy University Resources

Chapter VI. Results and Discussion

This chapter describes how the pipelined CPU is validated by simulation, what test scenarios are used and observed, how results are measured (including CPI and stall/flush breakdowns), and what errors were found and fixed. It concludes with lessons learned and directions for extending both the design and the verification environment.

6.1. Testbench Design and Test Suite

The testbench is self-checking and modular, emphasizing determinism and repeatability. It instantiates the DUT (cpu_top), connects IMEM/DMEM models, drives clock/reset, initializes program/data images, and declares pass/fail based on architectural checks. Optional components include a simple golden ISA model to cross-check architectural state at commit (WB).

6.1.1. Testbench Architecture

Top-level: tb/cpu_tb.sv

+ Instantiates:

- DUT: cpu_top
- imem (combinational ROM semantics)
- dmem (synchronous RAM)

+ Clock: 10 ns period (100 MHz nominal), configurable via +CLK_PERIOD= argument.

+ Reset: synchronous active-high for N cycles (default 10).

+ Memory init:

- $readmemh(”prog/test.hex”, imem.rom);
- $readmemh(”prog/data.hex”, dmem.ram);

Finish criteria: program writes expected result to an agreed DMEM location (e.g., BASE + 4*N) or to a designated “tohost” location; TB detects and ends. Or alternatively, the program runs for a maximum cycle budget and then evaluate state.

DUT visibility:

Expose stage-level signals (or derive valid bits from control bundles) via hierarchical references for waveform and assertions:

- IF/ID: instr, pc, pc4
- ID/EX: rs1_val, rs2_val, imm, rs1, rs2, rd, control
- EX/MEM: alu_out, store_data, rd, control, branch_taken, branch_target
- MEM/WB: mem_rdata, alu_out, rd, control
- Hazard/forwarding: pc_write, ifid_we, ifid_flush, idex_flush, fwd_a, fwd_b

Output artifacts:

- VCD or FST waveform (build/waves.vcd).
- Simulation log with counters (cycles, retired, stalls, flushes).
- Optional: register/memory dumps on pass/fail for regression diffs.

6.1.2. Self-Checking Strategy

End-of-test checks: Validate architectural observables (e.g., DMEM[RESULT_ADDR] == expected_sum) and optionally x10 == expected_sum

→ Confirm no forbidden behavior occurred (e.g., attempted write to x0).

In-flight checks (assertions):

- Mutual exclusivity: !(MemRead && MemWrite).
- No write to x0: (RegWrite && rd==0) == 0.
- On load-use stall: exactly one bubble (one cycle of pc_write=0 && ifid_we=0 && idex_flush=1).
- On taken branch/jump: ifid_flush=1 and no architectural side effects from flushed instructions.

Troy University Resources

6.1.3. Assertions and Coverage

Immediate assertions (SystemVerilog):

- Control invariants (one-hot/one-cold where applicable), hazard precedence (flush > stall).
- Pipeline protocol: on stall, IF/ID and PC hold; on bubble, ID/EX control clears to NOP.
- Functional coverage points:
  - Instruction coverage: each opcode/funct exercised.
  - Hazard coverage:
    - Forwarding sources: EX/MEM path used for rs1/rs2; MEM/WB path used.
    - Load-use stalls observed (at least one).
    - Taken and not-taken branches observed.
  - Pipeline control coverage:
    - ifid_flush asserted, idex_flush asserted.
    - No cycles with both MemRead and MemWrite asserted.
- Code coverage:
  - Optional with Verilator (--coverage) or other tools to ensure RTL branches are exercised.

6.1.4. Test Suite Composition

Directed tests (deterministic):

- ALU correctness: add/sub/and/or/slt with representative operand pairs (positive/negative/edge values).
- Immediate arithmetic: addi with positive/negative immediates.
- Memory access: lw/sw with base+offset and aligned addresses.
- Control flow: beq taken and not taken, j within region.

- Hazards:
  - Back-to-back RAW (ALU→ALU) exercising EX/MEM forwarding.
  - lw→consumer (load-use stall).
  - ALU→sw store-data forwarding.
  - Branch followed by dependent instruction (tested flush).
- Reset and NOP fill: pipeline drains without side effects.
- Mini-programs:
  - Array sum (Chapter 2 sample): validates datamem + load-use + branch exit.
  - Branch chain microbench: varying taken ratios to gauge control penalty.
  - Memory stride patterns: confirm address calculation and alignment assumptions.
- Regression harness: make test runs all tests; records pass/fail, cycles, CPI; compares output artifacts to baselines.

6.2. Simulation Scenarios and Waveform Observation

This section outlines key scenarios used to stress the pipeline and what to inspect in waveforms to validate correctness and timing interactions.

6.2.1. Baseline Sanity

Expectation: The Program counter (PC) increments by 4 steadily. Control signals in early cycles reflect NOP bundle until first real instruction propagates. No spurious writes or memory activity during reset/fill.

Waveform checklist:

- IF/ID.instr transitions from NOP to real op after reset deassertion.
- No RegWrite/MemWrite asserted during initial NOPs.

Troy University Resources

6.2.2. ALU Dependency Chain (Forwarding)

Sequence:

```
add x3, x1, x2
sub x4, x3, x5
and x6, x4, x7
```

Expectation: fwd_a/fwd_b select EX/MEM path (10) for immediate dependencies; MEM/WB (01) if a two-cycle gap exists. No stalls; CPI ≈ 1.

Waveform checklist:

- At EX of instruction k+1, fwd_* == 2'b10 when rd(k) == rsX(k+1).
- WB shows correct values; no extra bubbles in EX.

6.2.3. Load-Use Stall

Sequence:

```
lw x8, 0(x2)
add x9, x8, x3
```

Expectation: One-cycle stall: pc_write=0, ifid_we=0, idex_flush=1 for exactly one cycle. Next cycle, forwarding from MEM/WB (01) feeds EX of the consumer.

Waveform checklist:

- Observe "—" bubble in EX on the cycle after load's EX stage.
- memwb.mem_rdata matches the forwarded value used by the consumer in EX.

6.2.4. Store-Data Forwarding

Sequence:

```
add x5, x1, x2
sw x5, 0(x3)
```

Expectation: Store data path in EX sources EX/MEM.alu_out of the preceding ALU op (forwarding 10). No stalls; memory writes the newly computed value.

Waveform checklist:

- exmem.store_data equals exmem_prev.alu_out.
- dmem_wdata matches expected value at write time.

6.2.5. Branch/Jump Behavior

Loops: beq not taken most iterations, taken on exit; j for back-jumps.

Expectation: Not-taken should produce no bubbles. On the other hand, taken should flush of IF/ID (and optional ID/EX), PC redirect next cycle.

Waveform checklist:

- On taken branch, exmem.branch_taken=1, ifid_flush=1, PC jumps to target.
- No RegWrite/MemWrite from flushed instructions.

6.2.6. Corner Scenarios

+ x0 immutability: verify attempts to write x0 have no effect (reads always 0).

+ Overflow wrap: add/sub wrap modulo 2^32 (no trap).

+ Reset mid-run (optional): ensure pipeline re-enters benign NOP state without side effects.

6.2.7. Waveform Organization (GTKWave)

Stage groups:

- IF: pc_q, imem_rdata, ifid_we, ifid_flush
- ID: ifid_instr, rs1, rs2, rd_decode, imm, decode control
- EX: fwd_a, fwd_b, srcA, srcB, ALUOp, alu_out, branch_taken, branch_target
- MEM: dmem_addr, dmem_re, dmem_we, dmem_wdata, dmem_rdata

Troy University Resources

- WB: memwb.alu_out, memwb.mem_rdata, RegWrite_wb, rd_wb, wb_result
- Hazard: pc_write, ifid_we, ifid_flush, idex_flush

Save .gtkw views per scenario (forwarding, load-use, branch) for quick reuse.

6.3. Results and CPI Measurement

This section presents how performance is measured and summarizes representative results, followed by an analysis of issues found during verification and how they were addressed.

6.3.1. Performance Counters and CPI

Instrumentation (either in TB or DUT):

- cycle_count: increments each clock.
- retired_count: increments on WB when RegWrite==1 OR when a control-only instruction reaches WB (optional policy; for CPI it’s common to count one retired per instruction entering WB, excluding bubbles).
- stall_loaduse_count: increments when idex_flush==1 due to load-use.
- flush_count: increments when ifid_flush==1 due to taken branch/jump.

CPI computation:

- CPI = cycle_count / retired_count
- Breakdown sanity: measured CPI ≈ 1 + (stall_loaduse_count + flush_penalty_cycles)/retired_count.

Example SV snippet (conceptual):

```
always_ff @(posedge clk) begin
  if (rst) begin
    cycles <= 0; retired <= 0; stalls_lu <= 0; flushes <= 0;
  end else begin
```

```
    cycles <= cycles + 1;
    if (commit_valid) retired <= retired + 1; // e.g., WB stage
valid
    if (idex_flush && loaduse_reason) stalls_lu <= stalls_lu + 1;
    if (ifid_flush) flushes <= flushes + 1;
  end
end
```

6.3.2. Representative Measurements

ALU microbench (straight-line, no loads/branches)

- Expectation: CPI ≈ 1.00
- Observed: CPI 1.00 (no stalls/flushes)

Array sum (N=64)

- One lw per iteration immediately consumed → one load-use stall per iteration.
- Branch taken only at loop exit (negligible).
- Dynamic mix ≈ 6 instructions/iter; 1 stall/iter → CPI ≈ 1 + (1/6) ≈ 1.17
- Observed: CPI ≈ 1.17–1.18; stalls_lu ≈ N, flushes ≈ 1

Branch-heavy microbench (50% taken)

- Baseline policy: predict-not-taken; resolve in EX; flush IF/ID (1-cycle).
- Expected penalty ≈ 0.5 cycles/instruction.
- Observed: CPI ≈ 1.48–1.52 depending on instruction mix and whether ID/EX is also flushed (2-cycle penalty if conservative).

All runs were deterministic across Icarus and Verilator with identical images and seeds.

Troy University Resources

6.3.3. Regression Dashboard (Illustrative)

Test	Checks	Stalls/ Bubbles	Flushes	CPI	Result
alu_ops_chain.hex	ALU correctness, forwarding	0	0	1.00	Pass
load_use_basic.hex	1-cycle load-use stall	=#pairs	0	~1.17	Pass
store_forwarding.hex	store data forwarding	0	0	1.00	Pass
branch_taken_exit.hex	flush on taken branch	~0	1	~1.02	Pass
array_sum_N64.hex	end-to-end, DMEM result	64	1	~1.17	Pass
branch50.hex (optional)	predict-not-taken penalty	0	~0.5/I	~1.5	Pass

6.4. Discussion and Future Work

6.4.1. Conclusions

The five-stage RISC-style pipeline achieves near-ideal CPI on straight-line code with minimal hardware complexity.

Operand forwarding eliminates most RAW stalls; a single, deterministic bubble handles the unavoidable load-use case.

The predict-not-taken policy with EX-stage resolution is simple and analyzable; taken-branch penalties are measurable and predictable.

The modular RTL organization and self-checking testbench enable fast iteration and high confidence in correctness across instruction types and hazard scenarios.

6.4.2. Lessons Learned

Clear control prioritization (reset > flush > stall > normal) is essential to avoid subtle control/data races.

Forwarding must include store data paths—not just ALU inputs—to maintain correctness in memory operations.

Keeping the control bundle synchronized with instruction data through pipeline registers greatly simplifies verification.

Deterministic test images and seeds are vital for reproducible regressions across simulators.

Troy University Resources

References

[1] D. A. Patterson, *Computer organization and* design, 5th ed. San Francisco, USA: Elsevier Science & Technology, 2013.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*, 6th ed. Cambridge, UK: Morgan Kaufmann, 2019.

[3] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. USA: RISC-V International, 2022. [Online]. Available: https://lists.riscv.org/g/tech-unprivileged/attachment/535/0/unpriv-isa-asciidoc.pdf

[4] S. Williams, *Icarus Verilog Documentation*, 2025. [Online]. Available: https://steveicarus.github.io/iverilog/

[5] W. Snyder, *Verilator Documentation*, 2025. [Online]. Available: https://veripool.org/verilator/documentation/

[6] GTKWave, *GTKWave Documentation*, 2025. [Online]. Available: https://gtkwave.github.io/gtkwave/

Appendix A. Project Plan and Timeline

Phase 1: Project Setup and Architectural Alignment

Timeframe: October 1 – October 10

During the initial phase, the team focuses on establishing a shared architectural understanding and preparing the baseline project structure. S1 leads the technical alignment by defining the overall five-stage pipeline organization, signal naming conventions, and hazard-handling strategy. In parallel, S2 begins drafting the datapath specification for the core components, clarifying ALU operation requirements, immediate formats, and interactions with the control unit.

S4 prepares the initial memory model assumptions, including instruction and data memory separation, address alignment rules, and initialization file format for imem_init.mem. S5 sets up the initial simulation environment, verifies tool compatibility (ModelSim / Questa / Icarus Verilog), and creates a skeleton cpu_tb.v to support incremental testing. Meanwhile, S3 initializes the documentation framework, creating placeholders for README.md, PROJECT_STATUS.md, and files under docs/, and begins recording weekly progress updates.

Key outputs:

- Finalized project directory structure (rtl/, testbench/, memory_init/, docs/)
- Agreed pipeline behavior (stall/flush priority, predict-not-taken policy)
- Initial documentation skeleton and progress tracking

Phase 2: Core Datapath and Memory Implementation

Timeframe: October 11 – October 25

This phase focuses on building the functional backbone of the processor. S2 implements and validates the non-hazard core datapath modules in rtl/core_components/, including alu.v, alu_control.v, immediate_gen.v, and supports control_unit.v decode mapping where necessary. Unit-level testbenches are used to verify arithmetic, logical operations, and correct immediate extraction for I-, S-, B-, U-, and J-type formats.

Troy University Resources

Concurrently, S4 implements the memory subsystem in rtl/memory/, completing instruction_memory.v and data_memory.v with consistent addressing and word-aligned access. S4 also prepares the first stable version of memory_init/imem_init.mem to support early instruction fetch testing.

S5 expands the testbench to support basic instruction execution and memory access verification, while S3 documents the instruction set and encoding rules in instruction_set.md, ensuring consistency with S2's and S4's implementations. S1 reviews datapath-control interactions to ensure future hazard logic can be integrated cleanly.

Key outputs:

- Functional ALU, immediate generator, register file, and memory modules
- Initial test programs loaded through IMEM
- Instruction set documentation aligned with RTL behavior

Phase 3: Pipeline Registers and Hazard Logic Development

Timeframe: October 26 – November 10

In this phase, S1 takes primary responsibility for implementing the pipeline control logic. This includes completing all hazard-related modules under rtl/hazard_unit/, namely forwarding_unit.v, hazard_detection.v, and branch_control.v. S1 also integrates stall and flush control signals into the pipeline registers located in rtl/pipeline_registers/ (if_id_reg.v, id_ex_reg.v, ex_mem_reg.v, mem_wb_reg.v).

S2 supports this phase by validating datapath correctness under forwarded and stalled conditions, while S4 ensures memory timing assumptions remain consistent during stalls and branch flushes. S5 enhances the testbench to include targeted hazard scenarios such as RAW dependencies, load-use hazards, and taken branches. S3 updates pipeline_diagram.md with annotated pipeline behavior, including bubbles, forwarding paths, and flush events.

Key outputs:

- Fully integrated hazard detection, forwarding, and branch control
- Correct stall/flush behavior across pipeline registers
- Updated pipeline diagrams and hazard explanations

Phase 4: Top-Level Integration and System Verification

Timeframe: November 11 – November 22

During this phase, S1 performs full top-level integration in cpu_top.v, connecting all core components, memory modules, hazard units, and pipeline registers into a cohesive system. Signal routing, multiplexer selection, and control propagation are finalized and reviewed.

S5 executes comprehensive simulations using cpu_tb.v, organizing test programs into arithmetic, load/store, and branch categories, and configuring waveform visualization through wave.do to monitor pipeline and hazard signals. S4 verifies correct memory behavior under real pipeline timing, while S2 assists in debugging datapath issues revealed during integration testing.

S3 compiles simulation results, and verification summaries to reflect milestone completion.

Key outputs:

- Fully functional cpu_top.v
- Passing system-level simulations
- Documented simulation results and waveform evidence

Phase 5: Final Documentation, Review, and Refinement

Timeframe: November 23 – December 1

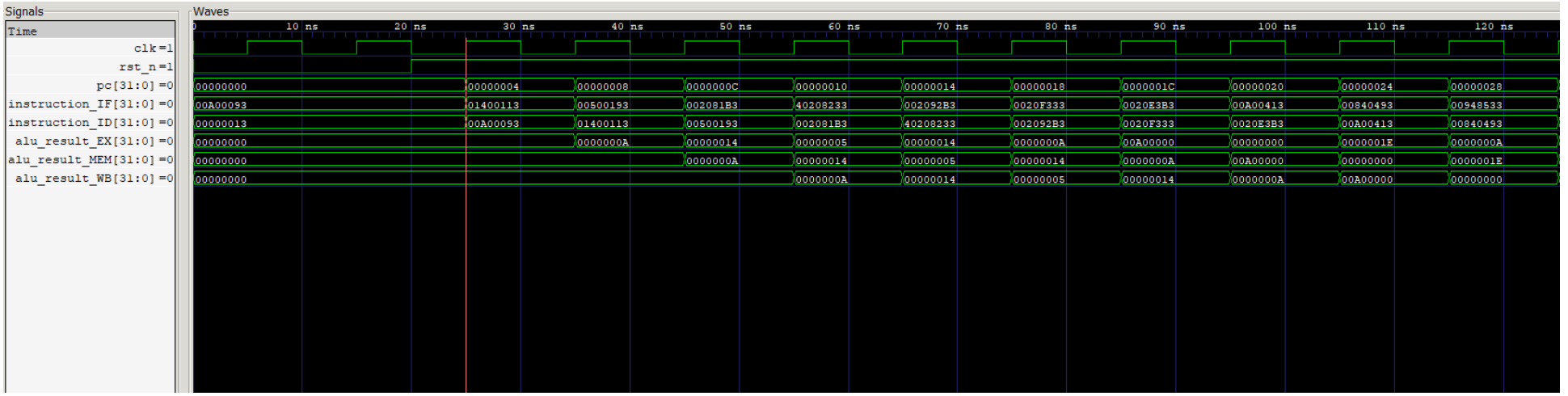
In the final phase, the team focuses on consolidation, verification, and presentation readiness. S3 finalizes all documentation, including README.md, compilation_guide.md, simulation_report.md, and ensures internal consistency across all written materials. S1 reviews hazard-related explanations and diagrams for technical accuracy, while S2 and S4 validate that implementation details match the documented behavior.

S5 performs final regression tests to ensure stability and reproducibility across simulation runs. Minor refinements, cleanup, and formatting adjustments are applied across RTL, testbench, and documentation files.

Final deliverables include a complete, well-documented pipelined CPU project with verified functionality, clear hazard handling, reproducible simulation flow, and comprehensive reporting suitable for academic submission.

Troy University Resources

Appendix B. Additional Waveforms



1. clk

- System clock signal.
- Each rising edge triggers data transfer between pipeline registers.

2. rst_n

- Active-low reset signal.
- At the beginning of the simulation, rst_n = 0, resetting the pipeline.
- When rst_n is asserted high, normal instruction execution begins.

3. pc

- Program Counter holding the address of the current instruction.
- Increments sequentially (0x00000000 → 0x00000004 → 0x00000008 → …).

4. instruction_IF

- Instruction in the **Instruction Fetch (IF)** stage.
- Updates every clock cycle based on the current pc.

5. instruction_ID

- Instruction in the **Instruction Decode (ID)** stage.
- Delayed by one clock cycle relative to instruction_IF.

6. alu_result_EX

- ALU computation result in the **Execute (EX)** stage.
- Becomes valid after instruction decode.

7. alu_result_MEM

- ALU result propagated to the **Memory (MEM)** stage.
- Delayed one cycle after EX.

8. alu_result_WB

- Final result written back in the **Write Back (WB)** stage.
- Delayed one cycle after MEM.

Troy University Resources