AetherRISC: Implementation Specifics Description (ISD)

Project: AetherRISC

Version: 0.0.0 (Pre-Alpha)

Date: 2025-12-21

Standard Compliance: IEEE 1016-2009 (Detailed Design)

1. Introduction

1.1 Purpose

This document details the low-level implementation strategies, algorithms, and data structures used to realize the architectural design defined in the **System Design Document (SDD)**. It serves as the primary guide for developers implementing the Core library and Simulation Engine.

1.2 Scope

The scope includes the **WebAssembly execution model**, **micro-architectural simulation logic** (Pipeline, Cache, MMIO), **dynamic runtime intervention** mechanisms, and **legacy compatibility layers**.

2. Simulation Engine Implementation

2.1 Hybrid Execution Model (Client-Side WASM)

To satisfy the non-functional requirement of >100 KHz clock speed without incurring server-side compute costs, the simulation engine utilizes a client-centric execution model within the **Blazor InteractiveAuto** architecture.

- **Service Scope:** The ISimulationRunner service is registered as Scoped. In the Blazor WebAssembly host, this creates a singleton instance per browser tab.
- **Execution Loop:**
  - **Mechanism:** The loop runs on the UI thread using requestAnimationFrame for synchronization with the display refresh rate, or a WebWorker for unthrottled execution.
  - **Batching Strategy:** To mitigate the overhead of C# $\leftrightarrow$ JavaScript interop, instructions are executed in "quanta" (batches) of $N$ cycles (configurable, default: 10,000) between UI renders.
- **State Synchronization:**

    - **Trigger:** Synchronization to the server occurs only on user-initiated events (Save Project, Snapshot) or specific debug triggers (Breakpoint hit).
    - **Protocol:** The MachineState is serialized to **Google Protobuf** format to minimize payload size during network transmission.

2.2 Instruction Decoding Strategy

To maximize performance and modularity, instruction decoding utilizes a **Tree-Based Lookup** combined with bitmask filtering.

- **Decoder Generation:** A Source Generator analyzes classes implementing IInstruction at compile-time to build a highly optimized decision tree.
- **Dynamic Extension:**
    - The InstructionDecoder maintains a secondary list of IInstructionSetExtension plugins.
    - **Strategy Pattern:** During the Decode stage, if the base ISA decoder fails (Invalid Opcode), the decoder iterates through registered extensions. This allows custom instructions to override or augment standard behavior.

3. Micro-Architecture Implementation

3.1 5-Stage Pipeline Logic

The pipeline is implemented as a Finite State Machine (FSM) managed by the PipelineController.

- **Token Passing:** Data flows between stages via PipelineLatch objects.
    - IF/ID Latch: Holds the raw instruction word and PC.
    - ID/EX Latch: Holds decoded control signals (RegWrite, MemRead) and operand values.
    - EX/MEM Latch: Holds ALU results and memory target addresses.
    - MEM/WB Latch: Holds memory read data or ALU pass-through data.
- **Stall Implementation:**
    - When the HazardDetectionUnit detects a hazard, it sets a Stall flag on the Fetch and Decode stages.
    - **Effect:** The PipelineController prevents the IF/ID latch from updating (locking the PC) and inserts a NOP (Bubble) into the ID/EX latch for the next cycle.

3.2 Cache Hierarchy (L1/L2)

The memory subsystem wraps the physical RAM with a CacheController implementing a **Write-Back, Write-Allocate** policy.

- **Addressing Logic:**
    - Address $A$ is decomposed: $Tag = A \gg (IndexBits + OffsetBits)$, $Index = (A \gg OffsetBits) \& IndexMask$.
- **L1 Implementation (Split I/D):**

    - **Storage:** Array of CacheSet structs, where each set contains $K$ CacheLine objects ($K$ = Associativity).
    - **Lookup Algorithm:**
        1. Index into Set array.
        2. Iterate through $K$ ways (simd-optimized).
        3. If Tag matches and Valid bit is set $\rightarrow$ **HIT**.
        4. Else $\rightarrow$ **MISS**.
- **Miss Handling:**
    1. Signal PipelineController to Stall.
    2. Select victim line using **Pseudo-LRU (PLRU)** algorithm.
    3. If victim is Dirty, write back to L2/RAM.
    4. Fetch new line from Lower Memory.
    5. Update Cache Line, set Valid = true, Dirty = false.
    6. Release Stall.

3.3 Memory-Mapped I/O (MMIO)

- **Routing:** The MemoryController checks the Most Significant Bits (MSB) of the address.
    - Range 0x80000000 - 0xBFFFFFFF: Routes to PhysicalMemory (RAM).
    - Range 0xFFFF0000 - 0xFFFFFFFF: Routes to MmioBus.
- **Device Interface:** Devices implement IMmioDevice.
    - Read(offset): Returns the current state of the device register.
    - Write(offset, value): Triggers device logic (e.g., updating a virtual LED status).

4. Runtime Intervention Subsystem (RIS)

4.1 Dynamic Code Injection (DCI) Implementation

The MemoryPatcher service implements the "Trampoline" technique to allow insertion of code into a compiled binary at runtime.

- **Allocation:** A specialized HeapAllocator reserves a block in the high memory arena (typically 0x7FF00000).
- **Injection Process:**
    1. **Target Identification:** User selects insertion point $P_{target}$ (Address $X$).
    2. **Payload Construction:**
        - $I_{new}$: The user's injected instruction(s).
        - $I_{original}$: The instruction originally residing at $X$.
        - $I_{jump\_back}$: JAL x0, (X + 4) (Unconditional jump back to next instruction).
    3. **Patch Writing:** The payload is written to the allocated block at Address $Y$.
    4. **Hook Installation:** The instruction at $X$ is overwritten with JAL x0, Y.
    5. **Relocation Handling:** If $I_{original}$ was PC-relative (e.g., AUIPC, BNE), its immediate value is re-calculated relative to $Y$.

4.2 Direct Register Access (DRA)

- **Concurrency Safety:** Modifications via the UI invoke MachineState.SetRegister().
- **Pipeline Flushing:** To ensure consistency, modifying a register triggering a "Pipeline Flush" event:
    - Instructions in IF, ID, and EX stages are discarded (converted to bubbles).
    - The PC is reset to the instruction in the Execute stage to re-evaluate dependencies with the new register value.

4.3 State Checkpointing (Memento)

- **Delta Compression:** The StateSnapshotManager does not store full memory copies.
    - It records a Transaction Log of memory writes per cycle.
    - **Snapshot:** BaseState + List<MemoryWriteOp>.
- **Restoration:** To restore Cycle $T-100$, the engine reverts the memory writes recorded in the transaction log in reverse order (Undo operation).

5. Heuristic Diagnostic Engine (HDE)

5.1 Deterministic Hazard Analysis

- **Static Analysis:** Before execution, the StaticAnalyzer builds a dependency graph of the basic block.
- **Dynamic Explanation:**
    - If Stall is active:
        - Query PipelineController.
        - If ID.Rs1 == EX.Rd AND EX.OpCode == LOAD: Return Explanation.LoadUseHazard.
        - If ID.Rs1 == EX.Rd AND EX.OpCode == ALU: Return Explanation.DataHazardRaw.

5.2 Deterministic Decompiler (Pseudo-C)

The DecompilerService projects assembly to C-like syntax using purely deterministic pattern matching (no probabilistic AI).

- **Control Flow Graph (CFG):**
    - Identifies Basic Blocks (sequences ending in Branch/Jump).
- **Pattern Matchers:**
    - Assignment: ADDI xA, x0, Imm $\rightarrow$ xA = Imm
    - Increment: ADDI xA, xA, 1 $\rightarrow$ xA++
    - Array Access: LW xA, Imm(xB) $\rightarrow$ xA = xB[Imm/4]
    - Branch: BGE xA, xB, Label $\rightarrow$ if (xA >= xB) goto Label
- **Symbol Integration:**
    - The output string generator replaces register identifiers (x10) with aliases from the SymbolTable (score) if present.

6. Legacy Compatibility Implementation

6.1 Venus / RARS Parity

The SyscallDispatcher acts as a facade, routing ecall triggers based on register a7 (x17).

- **File I/O (Venus IDs 57-60):**
  - Mapped to a virtualized **In-Memory File System (IMFS)**.
  - Open(path): Looks up file entry in the project asset bundle.
  - Read(fd, buffer): Copies bytes from the IMFS asset to the simulated RAM buffer.
- **Sbrk (Venus ID 9):**
  - Manages the ProgramBreak pointer in the MachineState, growing the heap upwards.

6.2 Asset Import

- **Format Readers:**
  - VenusProjectReader: Parses .json project files from Venus.
  - ElfReader: Uses **ELFSharp** to parse standard ELF binaries, extracting .text, .data, and .symtab sections.

7. Visualization & UI Binding

7.1 Virtual PCB Implementation

The PcbView.razor component utilizes **SVG Data Binding** for high-performance rendering.

- **Definition:** The board layout is defined in an SVG template with specific id attributes (e.g., id="led_0").
- **Update Loop:**
  - On every UI refresh tick, the component queries the MmioBus for the state of the GPIO register.
  - **Logic:** string fill = (gpioReg & 1) != 0 ? "red" : "#333";
  - **Render:** Updates the style.fill property of the SVG element via Blazor binding.

7.2 Cache Grid Visualization

- **Component:** CacheGrid.razor.
- **Data Source:** Binds directly to the CacheController.Sets array.
- **Visual States:**
  - **Invalid:** Grey background.
  - **Valid:** White background.
  - **Dirty:** Red border.
  - **Hit Animation:** CSS @keyframes flash (Green) triggered by a StateHasChanged event when a hit is recorded on a specific line.