AetherRISC: System Design Document (SDD)

Project: AetherRISC

Version: 0.0.0 (Pre-Alpha)

Date: 2025-12-21

Standard Compliance: IEEE 1016-2009 (System Design Description)

1. Introduction

1.1 Purpose

This document describes the architectural design and system decomposition of AetherRISC. It translates the requirements defined in the **General Specification (SRS v0.0.0)** into a technical blueprint for implementation. This document serves as the authoritative reference for the software architecture, class structure, data interfaces, and deployment topology.

1.2 Design Scope

The design encompasses the **AetherRISC.Core** (Domain Logic), **AetherRISC.UI** (Presentation), and the **Infrastructure Layers** (Desktop/Web Hosts). The design prioritizes:

- **Modularity:** Strict separation of concerns via the *Clean Architecture* pattern.
- **Portability:** The Core library must be platform-agnostic (Standard 2.1 / .NET 10 Class Library) to support both Native Desktop (MAUI) and WebAssembly (WASM) runtimes.
- **Determinism:** All simulation logic must be stateless outside of the explicit MachineState container to ensure reproducible execution.

2. Architectural Representation

2.1 Architectural Style

AetherRISC utilizes a **Hybrid Client-Server** architecture with a **Layered** internal structure.

- **Presentation Layer (Blazor InteractiveAuto):** Handles UI rendering. It intelligently switches between Server-Side Rendering (SSR) for static content and Client-Side WebAssembly (WASM) for the simulation engine.
- **Domain Layer (Core):** Contains the business rules (ISA, Pipeline, Memory). It has *zero* dependencies on external frameworks.
- **Infrastructure Layer:** Implements concrete services (File System, Networking, Compilation).

2.2 Technology Stack

- **Framework:** .NET 10 (Preview)
- **Language:** C# 14
- **Client Runtime:** Blazor WebAssembly (WASM)
- **Desktop Runtime:** .NET MAUI (Windows/macOS/Linux) with WebView2
- **Server Runtime:** ASP.NET Core
- **Data Serialization:** Google Protobuf (Network Sync), JSON (Configuration)
- **Graphics:** HTML5 Canvas via Blazor.Extensions.Canvas

3. System Decomposition & Module Design

The system is partitioned into the following high-level subsystems.

3.1 Subsystem: Central Processing Unit (CPU)

Namespace: AetherRISC.Core.Architecture

The CPU subsystem models the physical hardware of the processor.

3.1.1 Register File Module

- **GeneralPurposeRegisterFile (Class):**
  - **Responsibility:** Manages the 32 integer registers (x0-x31).
  - **Constraints:** x0 is hardwired to zero.
  - **API:** uint Read(int index), void Write(int index, uint value).
- **ControlStatusRegisterBank (Class):**
  - **Responsibility:** Manages CSRs (mstatus, mcause, mepc).
  - **Logic:** Implements privilege level transitions and exception vectoring.

3.1.2 Pipeline Module

Namespace: AetherRISC.Core.Architecture.Pipeline

Implements the cycle-accurate 5-stage pipeline.

- **PipelineController (Class):**
  - **Responsibility:** The central clock. Orchestrates the Tick() method for all stages and manages inter-stage latches.
- **IPipelineStage (Interface):**
  - **Contract:** void Tick(MachineState state, PipelineLatch inLatch, PipelineLatch outLatch);
  - **Implementations:** FetchStage, DecodeStage, ExecuteStage, MemoryStage, WritebackStage.
- **PipelineLatch (Data Structure):**
  - **Responsibility:** Holds the transient state between stages (e.g., IF/ID, ID/EX) to

simulate signal propagation delays.

3.1.3 Hazard Unit

Namespace: AetherRISC.Core.Architecture.Hazards

- **HazardDetectionUnit (Class):**
  - **Logic:** Compares ID.Rs1/Rs2 against EX.Rd and MEM.Rd.
  - **Action:** Injects bubbles (NO-OPs) into the pipeline if a Load-Use hazard is detected.
- **ForwardingUnit (Class):**
  - **Logic:** Controls MUX selectors to bypass the Register File and feed ALU results directly back to inputs.

3.2 Subsystem: Memory Management

Namespace: AetherRISC.Core.Architecture.Memory

3.2.1 Memory Bus & Hierarchy

- **IMemoryBus (Interface):**
  - **Contract:** uint ReadWord(uint address), void WriteWord(uint address, uint value).
- **MemoryController (Class):**
  - **Responsibility:** Routes requests based on the address map (0x80000000 -> RAM, 0xFFFFxxxx -> MMIO).
- **PhysicalMemory (Class):**
  - **Implementation:** Wraps a byte[] array. Implements bounds checking and Endianness translation.

3.2.2 Cache Module

Namespace: AetherRISC.Core.Hardware.Cache

- **CacheController (Class):**
  - **Responsibility:** Manages L1 Instruction, L1 Data, and L2 Unified caches.
  - **Logic:** Implements LRU (Least Recently Used) eviction policies.
- **CacheLine (Struct):**
  - **Fields:** bool Valid, bool Dirty, uint Tag, byte[] Data.

3.3 Subsystem: Runtime Intervention (RIS)

Namespace: AetherRISC.Core.Runtime.Intervention

Handles "God Mode" capabilities (Supervisory Control).

- **InterventionManager (Class):**
  - **Responsibility:** Facade for external UI commands to mutate state.
  - **API:** ForceRegisterWrite(index, val), ForcePcJump(addr).
- **MemoryPatcher (Class):**
  - **Responsibility:** Implements **Dynamic Code Injection (DCI)**.

    - **Algorithm:**
        1. Allocates "Trampoline" space in the High Memory Arena.
        2. Writes the injected opcode sequence + displaced instruction + JUMP back.
        3. Overwrites the target instruction with a JUMP to the Trampoline.
- **StateSnapshotManager (Class):**
    - **Responsibility:** Implements the Memento pattern.
    - **Storage:** Maintains a Ring Buffer of StateDelta objects for time-travel debugging.

3.4 Subsystem: Heuristic Diagnostics (HDE)

Namespace: AetherRISC.Core.Runtime.Diagnostics

Handles deterministic analysis (formerly "The Professor").

- **StaticAnalyzer (Class):**
    - **Responsibility:** Scans the instruction stream for potential errors before execution.
- **StallExplainer (Class):**
    - **Responsibility:** Analyzes the PipelineController state. If Stalled == true, returns a structured explanation (e.g., StallReason.LoadUseHazard).
- **CrashReporter (Class):**
    - **Responsibility:** Invoked on CPU Exception. Backtraces the Reorder Buffer to find the root cause instruction.

4. Process View (Execution Model)

4.1 The Simulation Loop (Client-Side)

To maintain high performance (>100 KHz) and UI responsiveness (60 FPS), the simulation runs in a dedicated WebWorker or UI Thread Loop within the WASM context.

1. **Input:** User clicks "Run".
2. **Cycle Batching:** The SimulationRunner executes instructions in batches (e.g., 10,000 cycles) to amortize JS Interop costs.
3. **Synchronization:**
    - If BatchCount % RefreshRate == 0: Serialize MachineState delta.
    - Dispatch event OnStateChanged.
4. **Rendering:** Blazor UI receives the event and invokes PipelineVisualizer.Render(delta) via HTML5 Canvas Interop.

4.2 Compilation Process (Server-Side)

1. **Request:** Client sends C/Rust source code string to CompilerController (API).
2. **Containerization:** Server spins up a transient Docker container (riscv-toolchain).
3. **Compilation:** Executes riscv64-unknown-elf-gcc.
4. **Analysis:** Parses the resulting ELF for .text, .data, and .debug_info.
5. **Response:** Returns a protobuf object containing binary sections and source-map

correlations.

5. Interface Design

5.1 ISA Extension API (Plugin System)

Third-party developers can extend the simulator by implementing the IInstructionSetExtension interface.

```
public interface IInstructionSetExtension {
    string Name { get; }
    string Prefix { get; } // e.g., "custom_"
    void RegisterOpcodes(InstructionDecoder decoder);
    void Execute(uint instruction, MachineState state);
}
```

5.2 GDB Remote Serial Protocol (RSP)

The desktop host implements a TCP listener to bridge external debuggers.

- **Socket:** TCP 127.0.0.1:3333
- **Handshake:** Supports qSupported, vCont, g, G, m, M packets.
- **Translation:** Converts GDB memory/register requests into InterventionManager calls.

6. Data Design

6.1 Configuration Schema (system_config.json)

Defines the "hardware" specifications of the emulated machine.

```
{
  "cpu": {
    "cores": 1,
    "isa": "rv64imafdc",
    "vector_len": 128
  },
  "memory": {
    "ram_size": 268435456,
    "endianness": "little"
  },
  "cache": {
    "l1_i": { "size": 32768, "ways": 4 },
    "l1_d": { "size": 32768, "ways": 4 },
    "l2":   { "size": 262144, "ways": 8 }
  },
```

```
  "mmio": {
    "devices": [
      { "id": "uart0", "addr": "0x10000000", "irq": 10 }
    ]
  }
}
```

6.2 State Serialization (Protobuf)

Used for creating save files and network synchronization.

```
message MachineStateSnapshot {
  uint64 cycle_count = 1;
  uint32 pc = 2;
  repeated uint32 registers = 3; // x0-x31
  map<uint32, bytes> memory_pages = 4; // Sparse map of 4KB pages
}
```

7. Directory Topology

The solution structure enforces the strict separation defined in Section 3.

```
Solution: AetherRISC
├── AetherRISC.Core (Standard 2.1)
│   ├── Architecture
│   │   ├── Registers
│   │   ├── Pipeline
│   │   │   ├── Stages (IF, ID, EX, MEM, WB)
│   │   │   └── PipelineController.cs
│   │   ├── Memory
│   │   │   ├── Bus
│   │   │   └── Virtualization (MMU/TLB)
│   │   └── Hazards
│   ├── Hardware
│   │   ├── Cache (L1, L2, Controllers)
│   │   └── Devices (GPIO, UART, InterruptController)
│   ├── Isa
│   │   ├── Base (Rv32i, Rv64i)
│   │   ├── Extensions (RvM, RvA, RvF, RvD, RvC, RvV)
│   │   └── Decoding (InstructionDecoder)
│   ├── Runtime
│   │   ├── Intervention (DCI, DRA, Snapshots)
```

```
│   │   └── Diagnostics (HDE, StaticAnalysis)
│   └── Services
│       ├── FileFormats (Elf, Hex)
│       └── Decompiler (Pseudo-C Projection)
│
├── AetherRISC.UI (Razor Class Library)
│   ├── Components
│   │   ├── Simulation (Canvas Visualizers, PCB)
│   │   ├── Editor (Monaco, Reference Pane)
│   │   └── Diagnostics (Register Grid, Heatmaps)
│   └── Layouts
│
├── AetherRISC.Web (ASP.NET Core)
│   ├── Controllers (Compiler API)
│   └── Hubs (Collaboration SignalR)
│
└── AetherRISC.Desktop (MAUI)
    └── NativeServices (FileSystem, GDB Socket)
```