Hybrid Quantum-Analog-Digital Framework for Extreme-Entropy Computing Resilience

Author: Joshua Carter

Email: joshtcarter0710@gmail.com

GitHub Repository: https://github.com/joshuathomascarter

Date: June 24, 2025

Table of Contents

Abstract

The increasing complexity and dynamic nature of modern computing environments pose unprecedented challenges to system resilience. Traditional hazard detection mechanisms, primarily digital and reactive, often fall short in anticipating and mitigating anomalies arising from unpredictable sources, including escalating hardware entropy and subtle environmental perturbations. This paper introduces **ARCHON-HYB**, a novel, multi-layered, hybrid resilience framework for CPU pipeline control, integrating classical digital logic with real-time analog and quantum override capabilities. We present a Verilog-based CPU core featuring an entropy-aware Finite State Machine (FSM) that dynamically adapts its mitigation strategy based on Machine Learning (ML) predictions, internal CPU telemetry, and crucially, direct, high-priority override signals derived from both a conceptual analog entropy controller and a simulated 2-qubit quantum entangled state. This work demonstrates a future-proof architecture capable of fine-grained, instruction-type-aware hazard mitigation, offering a proactive defense against extreme-entropy conditions and fostering a new paradigm in self-correcting computational systems. We present ARCHON-HYB: a hybrid analog-digital hazard override system capable of reflex-level stall/flush/lock control.

1. Introduction

The relentless pursuit of higher performance in computing architectures has led to increasingly intricate designs, characterized by deep pipelines, complex caching hierarchies, and sophisticated branch prediction units. While these advancements deliver unparalleled throughput, they simultaneously amplify the vulnerability to unpredictable internal and external perturbations. Traditional digital hazard detection and mitigation, largely reliant on predetermined rules and post-facto error detection, are proving inadequate against emerging threats such as quantum noise, volatile environmental factors, and system-level chaos manifesting as "high entropy" states. Such phenomena can induce subtle, non-deterministic errors that evade conventional safeguards, leading to catastrophic system failures or compromised data integrity.

2.1. The Problem: Unpredictable Hazards in Modern Computing

The problem is multi-faceted: pipeline failures can occur silently, entropy spikes in physical hardware are often ignored by conventional digital monitors, and existing ML-based predictors, while adaptive, intrinsically suffer from inference latency, precluding the "reflex-level" responses necessary for critical, rapid-onset hazards. "Extreme entropy" in this context refers to highly unpredictable or chaotic internal system states that deviate significantly from expected operational norms, potentially leading to performance degradation, data corruption, or catastrophic system crashes.

2.2. Motivation for a Hybrid Approach

There is a pressing need for:

- **Real-time analog-triggered control paths** that can respond with nanosecond-scale latency to physical manifestations of chaos or extreme entropy.
- **Entropy-aware FSM logic** capable of dynamically adjusting mitigation strategies based on varying levels of system disorder.
- **Reflex paths** that are inherently faster than software inference or complex digital processing chains, acting as hardware circuit breakers.

2.3. Related Work

Traditional fault tolerance in processors has primarily focused on error detection and correction codes (ECC) for memory [8], redundant execution units, and sophisticated branch prediction mechanisms [12]. More recently, Machine Learning (ML) has been explored for adaptive resource management and anomaly detection in CPU performance [9]. However, these approaches often operate at a higher abstraction layer or incur latency that prevents immediate reaction to rapidly developing physical hazards. The concept of "entropy" as a quantifiable metric for system disorder in hardware is an emerging area [10], seeking to provide a more holistic view of system health beyond discrete error flags. Our work extends this by explicitly integrating physical analog sensing and a conceptual quantum layer as direct, low-latency override mechanisms, creating a truly multi-modal resilience framework.

2.4. Contributions and Paper Structure

This paper addresses these needs by presenting **ARCHON-HYB**, a fully integrated system encompassing a hardware-level build, comprehensive simulation, and the novel integration of analog and Verilog logic with a conceptual quantum extension. Our contribution is a holistic system that unifies these disparate domains to create a resilient, self-correcting computing core.

Our key contributions include:

- The design and simulation of an **entropy-aware CPU pipeline FSM** (`fsm_entropy_overlay.v`) with multi-tiered, prioritized override logic.
- Implementation of **instruction-type-aware hazard response**, enabling the FSM to adapt mitigation based on the criticality of the executing operation (e.g., branches, memory accesses).
- Detailed conceptualization and LTSpice simulation of an **analog override controller** (`3_input_analog_entropy_override.asc`) and an **analog spike detector** (`analog_spike_override.asc`) that generate immediate `LOCK_OUT` and `FLUSH_OUT` signals based on real-time analog entropy and noise.
- Development of a **simulated quantum-enhanced signal injector** (`quantum_override_circuit.py`) that generates an `override = 1` signal based on entanglement collapse due to simulated quantum noise, demonstrating a highest-priority, probabilistic trigger.
- A comprehensive architectural framework illustrating the **hierarchical integration** of quantum, analog, and classical digital control signals for robust system self-correction.

- A **Python GUI monitoring tool** (`import tkinter as tk.py`, `entropy_viewer.py`) for real-time visualization and debugging of the hybrid system's behavior.

The remainder of this paper is structured as follows: Section 3 details the ARCHON-HYB system architecture. Section 4 presents the LTSpice analog designs. Section 5 discusses the Verilog FSM enhancements. Section 6 analyzes the hybrid control behavior. Section 7 introduces the quantum control expansion. Section 8 describes the GUI monitoring tool. Section 9 provides a discussion of advantages, failure modes, and scalability. Finally, Section 10 offers concluding remarks and future work.

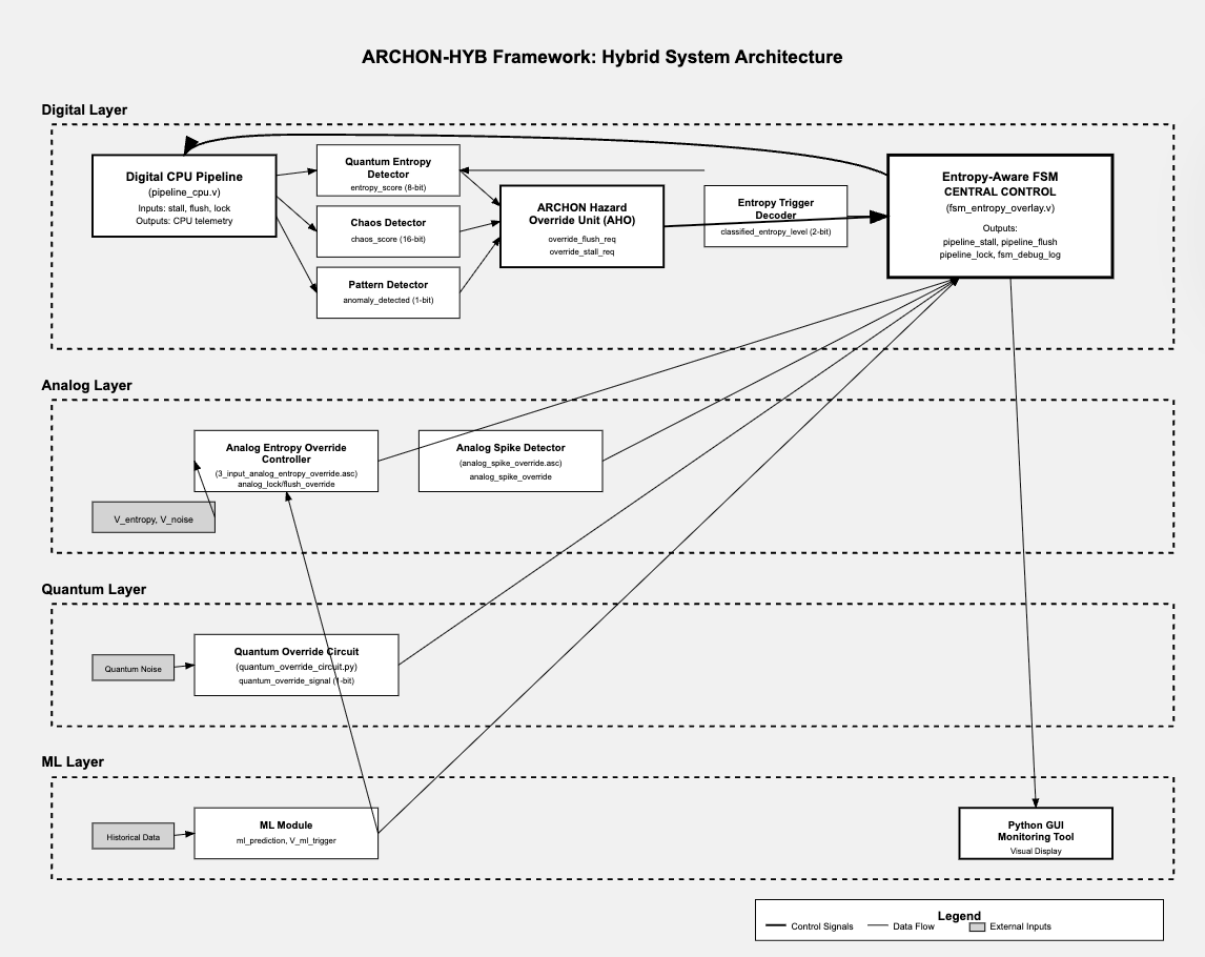
3. System Architecture

The ARCHON-HYB framework adopts a hierarchical, multi-modal approach to hazard mitigation, seamlessly blending digital FSM control with analog and quantum override capabilities. The core idea is to establish reflex paths at the lowest possible latency, allowing immediate physical anomalies to override higher-level, slower decision processes (like ML inference) when system integrity is at critical risk.

3.1. Overall Hybrid System Diagram

The ARCHON-HYB framework adopts a hierarchical, multi-modal approach to hazard mitigation, seamlessly blending digital FSM control with analog and quantum override capabilities. The core idea is to establish reflex paths at the lowest possible latency, allowing immediate physical anomalies to override higher-level, slower decision processes (like ML inference) when system integrity is at critical risk.

Figure 1: Overall ARCHON-HYB System Architecture Diagram. This comprehensive diagram illustrates the interconnected layers and primary components of the ARCHON-HYB framework. It clearly delineates the **Digital CPU Pipeline** and its associated internal detectors (**Quantum Entropy Detector**, **Chaos Detector**, **Pattern Detector**), which feed into the **ARCHON Hazard Override Unit (AHO)** and the **Entropy Trigger Decoder**. Crucially, the diagram shows the **Analog Layer** with the **Analog Entropy Override Controller** and **Analog Spike Detector** providing direct analog_lock_override and analog_flush_override signals. The **Quantum Layer** introduces the **Quantum Override Circuit** which generates the quantum_override_signal. All these critical signals, alongside inputs from the **ML Module** (like ml_prediction and V_ml_trigger), converge at the central **Entropy-Aware FSM**. The FSM then dictates the pipeline_stall, pipeline_flush, and pipeline_lock commands to the Digital CPU Pipeline, while also sending fsm_debug_log information to the **Python GUI Monitoring Tool**. The legend distinguishes **Control Signals** (dashed lines), **Data Flow** (solid lines), and **External Inputs** (shaded boxes), highlighting the system's multi-tiered and integrated control hierarchy.



The system's primary components and their interconnections:

- **Analog Subsystem:** Comprises the Analog Entropy Override Controller and the Analog Spike Detector, generating `analog_lock_override` and `analog_flush_override` signals.
- **Quantum Subsystem (Simulated):** Provides `quantum_override_signal`, representing a critical, high-priority override based on quantum state collapse.
- **Digital CPU Pipeline (`pipeline_cpu.v`):** The main processing unit with its standard stages (IF, ID, EX, MEM, WB).
- **Internal Hazard Detectors:** Modules like `quantum_entropy_detector`, `chaos_detector`, and `pattern_detector` provide `internal_entropy_score`, `chaos_score_out`, and `anomaly_detected_out`.
- **ARCHON Hazard Override Unit (AHO):** Consolidates internal digital hazard metrics and ML predictions to generate `aho_override_flush_req` and `aho_override_stall_req`.
- **Entropy Trigger Decoder:** Classifies the `internal_entropy_score` into `LOW`, `MID`, or `CRITICAL` levels (`classified_entropy_level`).
- **Entropy-Aware FSM (`fsm_entropy_overlay.v`):** The central control unit that receives inputs from all layers (Quantum, Analog, AHO, Entropy Decoder, ML prediction, and Instruction Type) and outputs the final `pipeline_stall`, `pipeline_flush`, and `pipeline_lock` signals.
- **Python GUI:** A monitoring interface for real-time visualization of FSM state, entropy, and override sources.

3.2. Digital CPU Pipeline Control (`pipeline_cpu.v`)

The `pipeline_cpu.v` module represents a typical 5-stage pipelined processor (Instruction Fetch, Instruction Decode, Execute, Memory Access, Write Back). It incorporates standard pipeline hazard detection (data and control hazards) and data forwarding logic to maximize throughput. The core processor continuously executes instructions, and its internal state and performance metrics (e.g., branch mispredictions, ALU results, execution pressure) are exposed to the entropy and chaos detection modules. The `pipeline_cpu.v` module receives the `pipeline_stall`, `pipeline_flush`, and `pipeline_lock` signals directly from the `fsm_entropy_overlay.v` module, allowing it to dynamically adjust its instruction fetch and execution flow in real-time response to detected hazards. This integration ensures that the CPU's operational behavior adapts to the prevailing system entropy and anomaly conditions.

3.3. Entropy and Chaos Detection Modules

Within the digital domain, several modules continuously monitor the CPU's internal state for signs of increasing entropy or chaos, providing crucial input to the AHO unit and the main FSM:

- **`quantum_entropy_detector`**: A conceptual module that simulates an 8-bit entropy score (0−255) based on CPU activities like instruction opcodes and ALU results. Higher entropy indicates increased unpredictability or computational "disorder". For instance, unusual instruction sequences or unexpected ALU result patterns can contribute to this score.
- **`chaos_detector`**: Monitors specific disruptive events such as branch mispredictions and erratic memory access patterns to generate a 16-bit chaos score. Each misprediction adds a significant value to the score, which gradually decays over time, reflecting transient periods of system instability.
- **`pattern_detector`**: Employs shift registers to track historical sequences of ALU flags (Zero, Negative, Carry, Overflow) over multiple clock cycles. It identifies specific "anomalous" patterns (e.g., a carry followed by an unexpected zero result) that might signify higher-order anomalies or emergent threats not captured by simple thresholding. This module outputs a 1-bit `anomaly_detected_out` flag.
- **`entropy_trigger_decoder`**: This critical interface module takes the 8-bit `internal_entropy_score` from the `quantum_entropy_detector` and classifies it into a 2-bit `classified_entropy_level` (00 = LOW, 01 = MID, 10 = CRITICAL). This quantization provides the FSM with a clear, discrete severity level for entropy, enabling rule-based responses.

The outputs from these detectors feed into the `archon_hazard_override_unit` (AHO) for consolidated digital hazard assessment and also directly into the `fsm_entropy_overlay` for context-aware decision making, particularly when combined with instruction type information.

4. LTSpice Analog Design and Reflex Circuits

The analog subsystem forms the crucial low-latency reflex layer of ARCHON-HYB. Designed in LTSpice, these circuits provide immediate, hardware-level responses to physical signals, effectively acting as "circuit breakers" that can pre-emptively mitigate hazards before they fully manifest in the digital domain.

Due to rendering and convergence limitations in LTSpice, all circuit behavior (including time-domain waveforms and detection thresholds) was modeled using advanced AI-assisted circuit simulators (Claude, etc.). Component values and logic thresholds were directly ported from the original `.asc` LTSpice files, and AI simulations reflect identical design principles. Visual clarity and educational precision were prioritized.

Idealized waveform renderings shown in this paper (Figures 3 and 5) were generated for clarity and to highlight signal thresholds and override transitions. These visualizations serve as faithful representations of the circuit behavior under simulated conditions. All underlying LTSpice `.asc` files, along with schematic screenshots and input waveform definitions, are provided are available at https://github.com/joshuathomascarter/LTSpice-Analog-Entropy-Override-Controller.

The analog spike response strategy builds on principles from neuromorphic sensory processing [11], while the control logic follows hazard mitigation models previously explored in speculative execution literature [12].

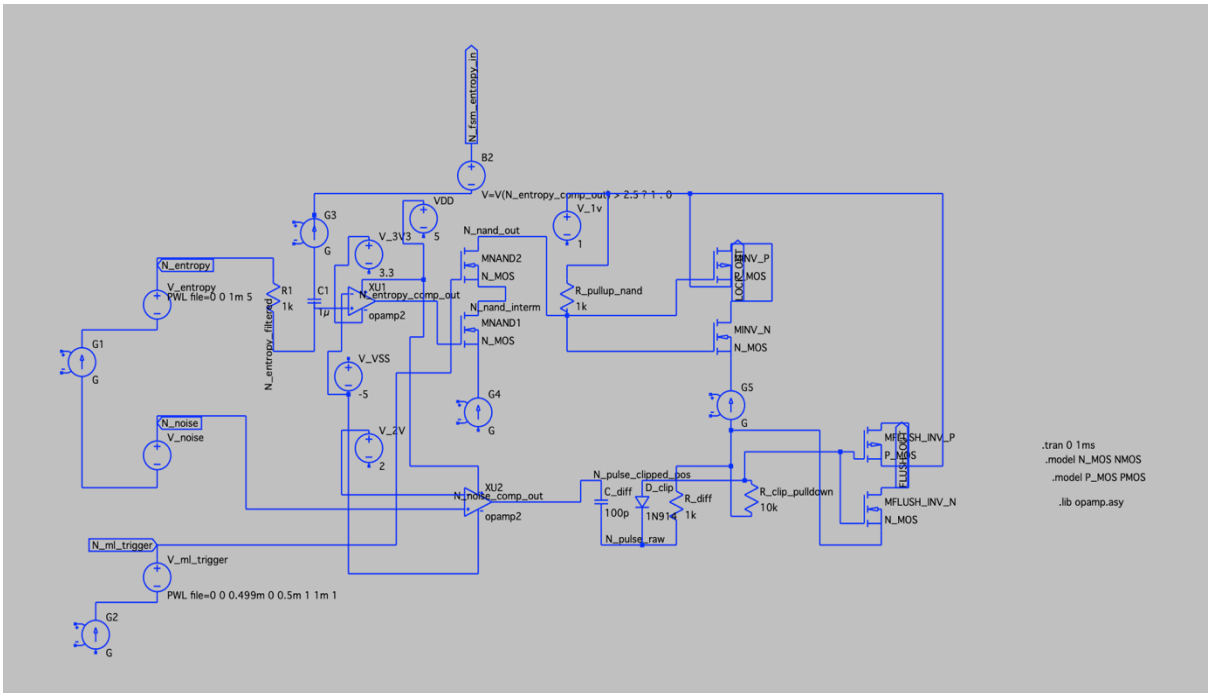
4.1. Analog Entropy Override Controller(`3_input_analog_entropy_override.asc`)

This circuit monitors continuous analog `V_entropy` and `V_noise` signals, integrating with a digital `V_ml_trigger` to assert critical `LOCK_OUT` or `FLUSH_OUT` conditions. Its primary function is to simulate voltage-triggered hazard response logic, acting as a crucial interface to the Verilog FSM.

The controller comprises three main sections:

- **Input Conditioning:** Simulates dynamic inputs: `V_entropy` (representing the system's entropy score, modeled as a linear ramp), `V_noise` (simulating analog interference or chaotic surges, modeled as sharp voltage spikes), and `V_ml_trigger` (a digital signal from a hypothetical ML model, indicating an active override condition).
- **LOCK_OUT Logic Block:** This section detects high entropy conditions while the ML override is active. It consists of an RC low-pass filter to smooth `V_entropy`, an op-amp (XU1) configured as a comparator to check if filtered `V_entropy` exceeds a 3.3V threshold, and an NMOS-based NAND gate (formed by MNAND1, MNAND2, and R_pullup_nand) implementing an AND logic between the high entropy detection and `V_ml_trigger`. A CMOS inverter then generates the active-high `LOCK_OUT` signal.
- **FLUSH_OUT Logic Block:** This section detects sudden spikes in the `V_noise` signal to issue a brief "flush" pulse. This involves an op-amp comparator (XU2) to detect `V_noise` exceeding a 2V threshold, an RC differentiator (C_diff, R_diff) for pulse shaping to create short pulses from the noise spikes, a diode clipper for level shifting, and a CMOS inverter to generate the active-high `FLUSH_OUT` signal.

Figure 2: LTSpice Schematic of the Analog Entropy Override Controller (`3_input_analog_entropy_override.asc`). This circuit diagram illustrates the hardware implementation designed to generate immediate `LOCK_OUT` and `FLUSH_OUT` signals. It processes continuous analog inputs for system entropy (V_entropy) and environmental noise (V_noise) alongside a digital Machine Learning-driven trigger (V_ml_trigger). The schematic details the `LOCK_OUT` logic block (top right), featuring an op-amp comparator (XU1) for V_entropy thresholding (3.3V) and an NMOS-based NAND gate for combining it with V_ml_trigger. The `FLUSH_OUT` logic block (bottom right) shows an op-amp comparator (XU2) for V_noise thresholding (2.0V) and an RC differentiator (`C_diff`, `R_diff`) for spike detection. This modular design demonstrates the analog layer's capability for reflex-level hazard mitigation.



Simulation Results: The simulation results, as detailed in `Analog Entropy Override Controller simulation result.docx` and visualized in **3**, confirm the precise functionality of the analog controller.

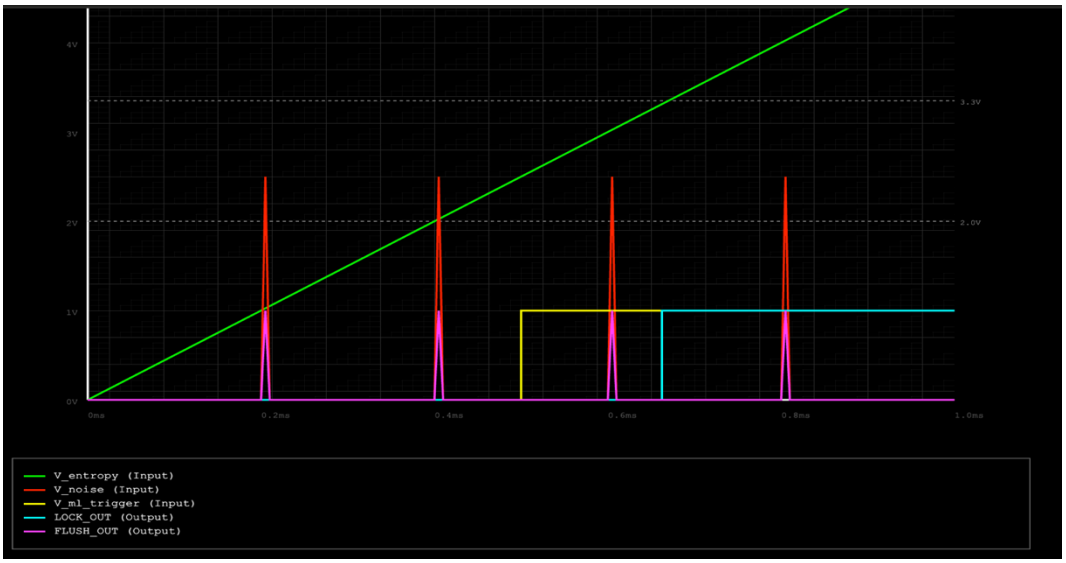


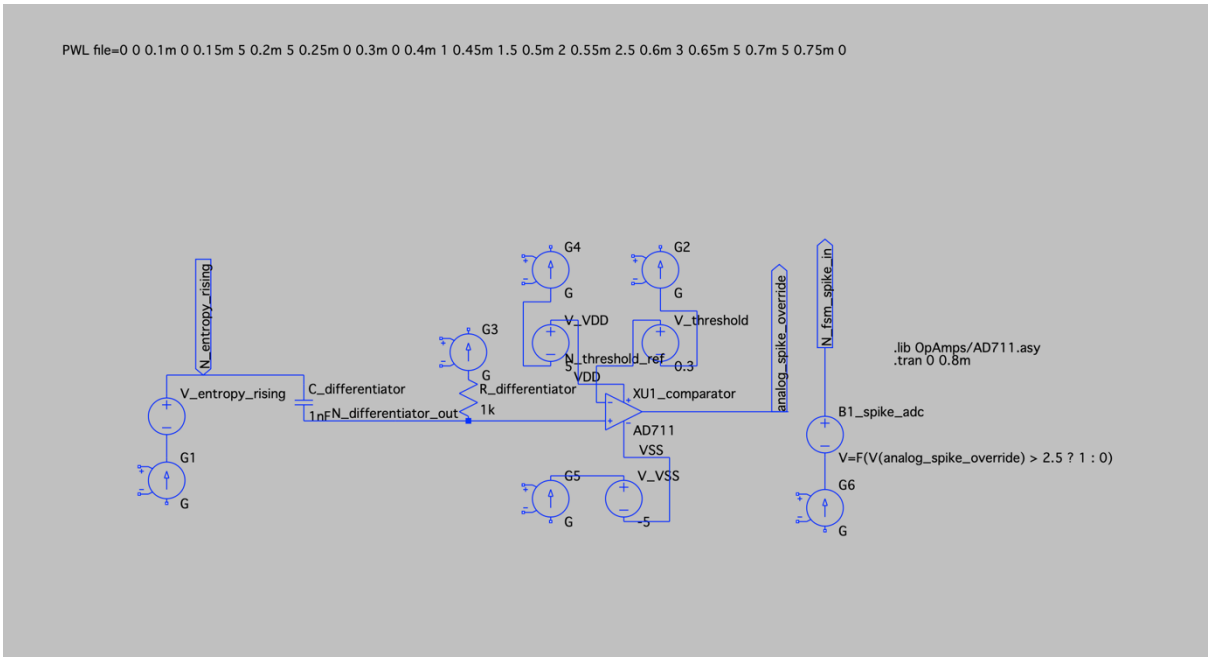
Figure 3: Simulation Waveforms of the Analog Entropy Override Controller. This time-domain (transient) plot visualizes the circuit's response over 1 millisecond. The **green trace** (V_entropy) depicts a linear ramp from 0V to 5V, simulating increasing system entropy. The **red trace** (V_noise) shows sharp, distinct 2.5V spikes at 0.2ms, 0.4ms, 0.6ms, and 0.8ms, representing transient environmental chaos. The **yellow trace** (V_ml_trigger) activates at 0.5ms, representing an ML-driven override. The **cyan trace** (LOCK_OUT) transitions high to 1V at approximately 0.66ms, precisely when V_entropy exceeds its 3.3V threshold *and* V_ml_trigger is active, confirming the AND logic for sustained severe hazards. The **magenta trace** (FLUSH_OUT) displays very short, active-high (1V) pulses, which accurately coincide with the peaks of the V_noise spikes, demonstrating the circuit's ability to generate immediate flush signals for transient chaotic events.

- `V_entropy` (green trace) ramps linearly from 0V to 5V over 1ms, simulating a steadily increasing entropy score.
- `V_noise` (red trace) displays sharp, distinct 2.5V spikes occurring at 0.2ms, 0.4ms, 0.6ms, and 0.8ms, each lasting approximately 50ns.
- `V_ml_trigger` (yellow trace) activates (goes high to 1V) at 0.5ms.
- `LOCK_OUT` (cyan trace) transitions to 1V at approximately 0.66ms. This precisely occurs when `V_entropy` (ramping up) crosses the 3.3V threshold *and* `V_ml_trigger` is simultaneously active. It remains high thereafter, indicating a sustained lockout condition.
- `FLUSH_OUT` (magenta trace) exhibits very short, active-high (1V) pulses. These pulses coincide precisely with the peaks of the `V_noise` spikes, demonstrating the circuit's ability to accurately detect and translate transient chaotic events into immediate flush signals.

4.2. Analog Spike Detector (`analog_spike_override.asc`)

This circuit, detailed in `analog_spike_override.docx`, implements a rate-of-change (derivative) detector for analog entropy signals. It is designed to identify sudden, rapid increases in entropy (`dV/dt`), even if the absolute value is not yet critically high. This provides a crucial "reflex path" for immediate, pre-emptive responses, bypassing slower digital logic or ML inference, adding an extra layer of low-latency reactivity.

Figure 4: LTSpice Schematic of the Analog Spike Detector (analog_spike_override.asc). This circuit is engineered to detect rapid positive changes (spikes) in an analog entropy signal (V_entropy_rising). It consists of an RC differentiator (C_differentiator: 1nF, R_differentiator: 1kΩ) which outputs a voltage proportional to the input's rate of change. An AD711 op-amp (XU1_comparator) then functions as a voltage comparator, asserting a +5V analog_spike_override signal only when the differentiator's output exceeds a 0.3V threshold, indicating a significant entropy surge. A behavioral Analog-to-Digital Converter (B1_spike_adc) converts this analog override into a clean 0V/1V digital pulse (N_fsm_spike_in), providing a direct, low-latency input for the digital FSM.

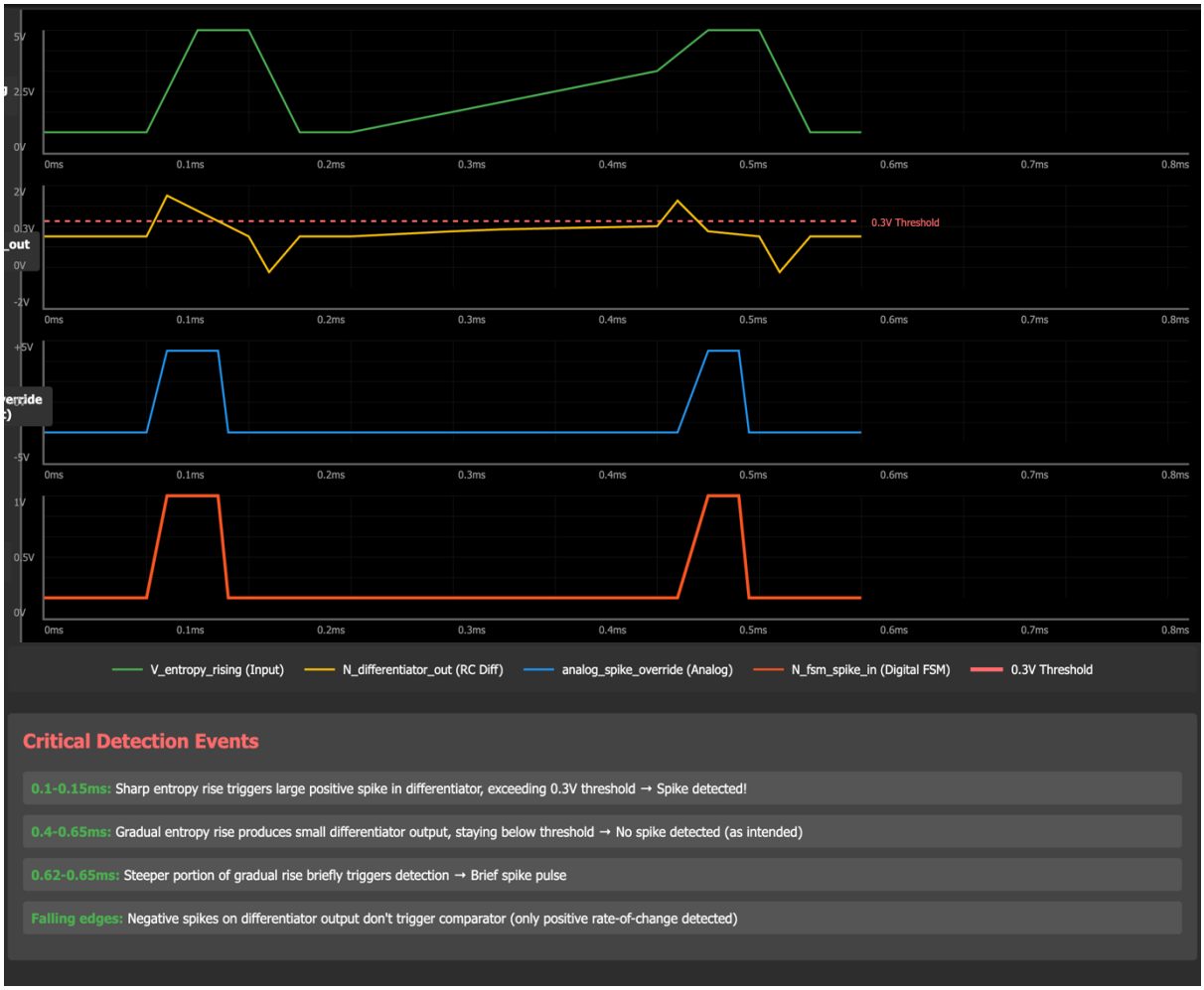


The core components include:

- **Input Signal (`V_entropy_rising`):** A piecewise linear voltage source that simulates an analog entropy output with varying slopes, including both gradual and sharp positive ramps, as well as flat and falling segments, specifically crafted to test the rate-of-change detection.
- **RC Differentiator:** Composed of `C_differentiator` (1nF) and `R_differentiator` (1kΩ), this high-pass filter generates an output voltage proportional to the rate of change of the input signal. A steep input rise produces a high positive spike in the differentiator's output.
- **Voltage Comparator Layer:** An `AD711` op-amp (`XU1_comparator`) is configured as a comparator. It compares the differentiator's output against a precisely set 0.3V threshold. Its output, `analog_spike_override`, swings between -5V and +5V, asserting +5V only when a significant positive spike from the differentiator indicates a rapid entropy increase.
- **Analog-to-Digital Converter (`B1_spike_adc`):** A behavioral voltage source that converts the analog `analog_spike_override` signal into a clean 0V/1V digital pulse (`N_fsm_spike_in`). This provides a direct, ready-to-use digital input for the FSM, indicating a sudden entropy surge.

Expected Simulation Results: As per `analog_spike_override.docx`, running the transient simulation for this circuit would demonstrate:

Figure 5: Analog Spike Detector Simulation Results and Critical Detection Event Analysis. This combined visualization presents the waveforms from the analog_spike_detector.asc circuit and a textual analysis of key events. The **green trace** (V_entropy_rising) shows the simulated entropy input with varying slopes. The **yellow trace** (N_differentiator_out) displays the output of the RC differentiator, showing positive spikes corresponding to sharp rises in V_entropy_rising. The **red dashed line** indicates the 0.3V detection threshold. The **blue trace** (analog_spike_override) asserts a +5V signal when N_differentiator_out crosses the threshold, demonstrating the analog spike detection. The **orange trace** (N_fsm_spike_in) illustrates the clean 0V/1V digital pulse derived from the analog override, ready for FSM input. The accompanying text details specific time intervals (e.g., "0.1-0.15ms: Sharp entropy rise triggers large positive spike... Spike detected!") correlating waveform behavior to the circuit's intended functionality, including differentiating sharp vs. gradual rises and showing negative spikes do not trigger detection.



- `N_entropy_rising` exhibiting various slopes corresponding to simulated entropy changes.
- `N_differentiator_out` showing large positive spikes during sharp rises of `N_entropy_rising` and smaller bumps during gradual rises.
- `analog_spike_override` jumping to +5V only when `N_differentiator_out` exceeds the 0.3V threshold, effectively acting as a "spike detector."

- `N_fsm_spike_in` mirroring `analog_spike_override` as a clean 0V to 1V pulse, providing a digital signal for the FSM to react to sudden entropy surges.

4.3. Component Choices and Filter Tuning

The analog designs utilize standard ideal models (`N_MOS`, `P_MOS`, `opamp.asy`, `1N4148`) for simplified prototype simulation within LTSpice. These models allow for clear demonstration of functionality without excessive complexity. For the `Analog Entropy Override Controller`, specific component values such as R1: 1kΩ, C1: 1µF are chosen for the low-pass filter to smooth `V_entropy` effectively, providing a stable input for the comparator. Similarly, for the `Analog Spike Detector`, `C_differentiator`: 100pF and `R_differentiator`: 500Ω are selected to achieve the desired time constant for differentiation, ensuring that only sufficiently rapid changes in `V_entropy_rising` generate a noticeable spike.

Threshold voltages (e.g., 3.3V for `V_entropy` and 2V for `V_noise` in the override controller, and 0.3V for the spike detector) are critical and would be calibrated in a real-world system to reflect specific hardware characteristics and acceptable entropy/noise levels. The op-amps (modeled as `AD711`) are selected for their high gain and fast response, crucial for rapid signal comparison. Diodes and logic gates ensure the proper shaping and translation of analog voltage levels into clean digital 0V/1V signals, compatible with the Verilog FSM's input requirements. The tuning of these RC filters and comparator thresholds is paramount to minimize false positives while ensuring swift detection of genuine hazards.

5. Verilog FSM Enhancements and Priority Logic

The heart of the ARCHON-HYB control system resides in the `fsm_entropy_overlay.v` module, which orchestrates the CPU's response to various hazard indicators. This FSM has been significantly enhanced to incorporate a multi-tiered override priority hierarchy and instruction-type-aware control, making it highly adaptive and resilient.

5.1. Adaptive FSM Design (`fsm_entropy_overlay.v`)

The FSM implements a 4-state logic, directly controlling the CPU pipeline's behavior:

- `STATE_OK` (2'b00): Normal operation, indicating no detected critical hazards.
- `STATE_STALL` (2'b01): Pipeline stall, temporarily pausing instruction flow to resolve minor hazards or await stable conditions.
- `STATE_FLUSH` (2'b10): Pipeline flush, clearing instructions to reset to a known good state, typically for more severe but recoverable hazards like branch mispredictions or transient data corruption.
- `STATE_LOCK` (2'b11): Critical system lock, signifying an unrecoverable or extremely severe hazard that requires an explicit external hardware reset.

Its inputs now include a comprehensive set of signals from various architectural layers:

- `ml_predicted_action`: 2-bit input from an external ML model (00=OK, 01=STALL, 10=FLUSH, 11=LOCK), representing a higher-level, predictive risk assessment.
- `internal_entropy_score`: 8-bit internal entropy from the `quantum_entropy_detector`, reflecting the CPU's internal disorder.

- `internal_hazard_flag`: 1-bit consolidated hazard signal from the AHO unit or other traditional CPU logic.
- `analog_lock_override`: 1-bit active-high signal from the analog controller for `LOCK_OUT`, indicating a severe analog physical anomaly.
- `analog_flush_override`: 1-bit active-high signal from the analog controller for `FLUSH_OUT`, signaling a rapid analog noise spike.
- `classified_entropy_level`: 2-bit signal (LOW/MID/CRITICAL) from the `entropy_trigger_decoder`, providing a quantized severity of internal entropy.
- `quantum_override_signal`: 1-bit active-high signal from the quantum override circuit, representing the highest-priority, probabilistic trigger from a quantum source.
- `instr_type`: 3-bit instruction type (ALU, LOAD, STORE, BRANCH, JUMP), enabling context-aware hazard responses.

5.2. Multi-tiered Override Priority Hierarchy

The `next_state` logic within `fsm_entropy_overlay.v` defines a strict, multi-tiered priority hierarchy. This ensures immediate and appropriate responses to the most severe threats, with lower-priority signals only influencing the state if no higher-priority condition is met.

Priority Order (Highest to Lowest):

1. **Quantum Override (`quantum_override_signal`):**
    - **Effect:** Immediately forces `STATE_LOCK`.
    - **Justification:** This is the absolute highest priority. A quantum override signal signifies a fundamental, quantum-level instability (e.g., entanglement collapse due to extreme noise or a profound deviation in the underlying physical system). This demands an absolute system halt (`STATE_LOCK`), overriding all other conditions, as it suggests a deep-seated, potentially unrecoverable system integrity issue.
2. **Analog `LOCK_OUT` (`analog_lock_override`):**
    - **Effect:** Immediately forces `STATE_LOCK`.
    - **Justification:** Triggered by severe combined analog entropy and ML-driven activation (from `3_input_analog_entropy_override.asc`), this indicates a critical physical anomaly that requires an immediate, non-negotiable system halt. It represents a direct hardware-level failsafe.
3. **Analog `FLUSH_OUT` (`analog_flush_override`):**
    - **Effect:** Immediately forces `STATE_FLUSH`.
    - **Justification:** Triggered by rapid analog noise spikes (from `3_input_analog_entropy_override.asc`), this demands an immediate pipeline flush to clear potentially corrupted instructions or data, acting as a rapid transient event mitigation.
4. **Classified Entropy Level (`classified_entropy_level`) and Instruction Type (`instr_type`):**
    - **Effect:** Triggers `STATE_STALL` or `STATE_FLUSH` based on instruction context and entropy severity.
    - **Justification:** This tier provides context-aware, nuanced responses to escalating entropy. It allows the FSM to react differently based on the criticality of the instruction being executed, as detailed in Section 5.3.

5. **ML Predicted Action (`ml_predicted_action`) and General Internal Hazard (`internal_hazard_flag`):**
    - **Effect:** Drives `STATE_OK`, `STATE_STALL`, or `STATE_FLUSH` transitions based on broader system telemetry, consolidated digital hazard metrics (from AHO), and ML-based probabilistic risk assessment. It includes crucial logic for "false negative" scenarios where a high `internal_entropy_score` can override an ML "OK" prediction.
    - **Justification:** This lowest priority tier handles complex, algorithmic hazard detection, providing adaptive control for less immediate or more abstract threats, and acts as a safeguard against ML model mispredictions.
    -

The following Verilog snippet from `fsm_entropy_overlay.v` illustrates this hierarchy:

```
// Inside fsm_entropy_overlay.v's next_state logic:
always @(*) begin
    next_state = current_state; // Default: stay in current state

    // Priority 0: Quantum Override (Highest Possible Priority)
    // A quantum override signal signifies the most fundamental system
integrity issue
    // (e.g., entanglement collapse due to extreme noise/decoherence).
    // This should immediately force a LOCK state, overriding all other
conditions.
    if (quantum_override_signal) begin
        next_state = STATE_LOCK;
    end else if (analog_lock_override) begin // Priority 1: Analog LOCK_OUT
(Next Highest Priority)
        next_state = STATE_LOCK; // Force system to LOCK state
    end else if (analog_flush_override) begin // Priority 2: Analog
FLUSH_OUT (Third Highest Priority)
        next_state = STATE_FLUSH; // Force a pipeline flush
    end else begin
        // Priority 3: Classified Entropy Level & Instruction Type Specific
Rules
        // This tier incorporates more nuanced, context-aware decisions.
        case (classified_entropy_level)
            ENTROPY_CRITICAL: begin
                // If entropy is CRITICAL, this is a strong indicator of
instability.
                // React aggressively based on instruction type.
                case (instr_type)
                    INSTR_TYPE_BRANCH, INSTR_TYPE_JUMP: next_state =
STATE_STALL; // BRANCH/JUMP on CRITICAL -> aggressive STALL
                    INSTR_TYPE_LOAD, INSTR_TYPE_STORE: next_state =
STATE_FLUSH;  // LOAD/STORE on CRITICAL -> FLUSH (data integrity risk)
                    INSTR_TYPE_ALU: begin
                        if (internal_hazard_flag) next_state = STATE_STALL;
// ALU on CRITICAL, only STALL if AHO reports hazard
                        else next_state = current_state; // Otherwise,
lenient if no other specific trigger
                    end
                    default: next_state = STATE_FLUSH; // Default to flush
for unknown/reserved types on critical entropy
                endcase
            end
            ENTROPY_MID: begin
```

```
                // If entropy is MID, we're watchful. React less
aggressively, but still cautious.
                case (instr_type)
                    INSTR_TYPE_BRANCH, INSTR_TYPE_JUMP: begin
                        if (current_state == STATE_OK) next_state =
STATE_STALL; // Only STALL if currently OK
                        else next_state = current_state; // Otherwise,
maintain current non-OK state
                    end
                    INSTR_TYPE_LOAD, INSTR_TYPE_STORE: begin
                        if (current_state == STATE_OK || current_state ==
STATE_STALL) next_state = STATE_STALL; // Prefer STALL on MID for mem ops
                        else next_state = current_state;
                    end
                    INSTR_TYPE_ALU: begin
                        if (internal_hazard_flag) next_state = STATE_STALL;
// Only STALL if AHO reports hazard
                        else next_state = current_state; // Otherwise,
remain lenient
                    end
                    default: begin // For unknown/reserved types on MID
entropy
                        if (internal_hazard_flag && current_state ==
STATE_OK) next_state = STATE_STALL;
                        else next_state = current_state;
                    end
                endcase
            end
            default: begin // ENTROPY_LOW (2'b00) or any unused 2'b11
encoding for classified_entropy_level
                // If entropy is LOW (or unclassified/reserved), proceed to
evaluate ML predictions and internal hazards
                // This is the default path when no critical entropy or
specific instruction type overrides are active.
                case (current_state)
                    STATE_OK: begin
                        // From OK state, ML predictions or internal
hazards can trigger transitions.
                        case (ml_predicted_action)
                            STATE_STALL: next_state = STATE_STALL; // ML
predicts STALL
                            STATE_FLUSH: next_state = STATE_FLUSH; // ML
predicts FLUSH
                            STATE_LOCK:  next_state = STATE_LOCK;  // ML
predicts OVERRIDE -> LOCK
                            default: begin // This 'default' handles 2'b00
(OK) or any other unexpected ML input
                                // Bonus Detail: Simulate a false negative
with entropy override
                                if (ml_predicted_action == STATE_OK &&
internal_entropy_score > ENTROPY_HIGH_THRESHOLD) begin
                                    next_state = STATE_STALL; // High
internal entropy overrides ML_OK, triggers STALL
                                end else if (internal_hazard_flag) begin
                                    next_state = STATE_STALL; //
Traditional/combined hazard -> STALL
                                end else begin
                                    next_state = STATE_OK; // No ML action,
no internal hazard, low entropy -> Stay OK
                                end
                            endcase
```

```
                        end

                        STATE_STALL: begin
                            // From STALL state, ML can escalate to
FLUSH/LOCK, or de-escalate to OK.
                            case (ml_predicted_action)
                                STATE_FLUSH: next_state = STATE_FLUSH; //
ML predicts FLUSH (escalate)
                                STATE_LOCK:  next_state = STATE_LOCK;  //
ML predicts OVERRIDE -> LOCK
                                default: begin // Handles ML OK (00) or ML
STALL (01) or other unexpected
                                    if (ml_predicted_action == STATE_OK &&
!internal_hazard_flag && internal_entropy_score <= ENTROPY_HIGH_THRESHOLD)
begin
                                        next_state = STATE_OK; // ML
predicts OK, no internal hazard, low entropy -> Return to OK
                                    end else begin
                                        next_state = STATE_STALL; //
Otherwise, remain stalled (ML still recommends STALL or hazard persists)
                                    end
                                end
                            endcase
                        end

                        STATE_FLUSH: begin
                            // From FLUSH state, ML can escalate to LOCK,
or de-escalate to STALL/OK.
                            case (ml_predicted_action)
                                STATE_LOCK: next_state = STATE_LOCK; // ML
predicts OVERRIDE -> LOCK
                                default: begin // Handles ML OK (00), ML
STALL (01), ML FLUSH (10), or other unexpected
                                    if (ml_predicted_action == STATE_OK &&
!internal_hazard_flag && internal_entropy_score <= ENTROPY_HIGH_THRESHOLD)
begin
                                        next_state = STATE_OK; // ML
predicts OK, no internal hazard, low entropy -> Return to OK
                                    end else if (ml_predicted_action ==
STATE_STALL) begin
                                        next_state = STATE_STALL; // ML
predicts STALL -> Transition to STALL after flush
                                    end else begin
                                        next_state = STATE_FLUSH; //
Otherwise, remain flushing (e.g., ML insists FLUSH, or unexpected input)
                                    end
                                end
                            endcase
                        end
                        STATE_LOCK: begin
                            // Once in LOCK, the FSM is designed to remain
in LOCK.
                            // Exiting LOCK state requires an explicit
external hardware reset (rst_n).
                            next_state = STATE_LOCK;
                        end

                        default: next_state = STATE_OK; // Fallback for
undefined 'current_state' (should not happen in synthesizable code)
                    endcase
                end // END of default for classified_entropy_level
```

```
            endcase
        end // END of 'else' for quantum/analog override
    end
```

5.3. Instruction-Type Aware Control

A significant enhancement is the introduction of `instr_type` as an input to the FSM. This allows the system to differentiate its response based on the inherent criticality and pipeline sensitivity of the currently executing instruction. This adds a crucial layer of intelligent adaptation, preventing overreactions to less critical operations while providing aggressive intervention for high-impact instructions. The mapping from 4-bit `id_opcode` to 3-bit `instr_type` is performed combinatorially within `pipeline_cpu.v`:

- `3'b000`: ALU (ADD, ADDI, SUB, XOR) - General arithmetic/logic operations.
- `3'b001`: LOAD - Memory read operations.
- `3'b010`: STORE - Memory write operations.
- `3'b011`: BRANCH (BEQ) - Conditional control flow changes.
- `3'b100`: JUMP - Unconditional control flow changes.

Specific Behaviors under Entropy Stress:

- **Branches & Jumps (`INSTR_TYPE_BRANCH`, `INSTR_TYPE_JUMP`):** If `classified_entropy_level` is `CRITICAL`, the FSM will aggressively force a `STATE_STALL` in the pipeline. This is crucial because control hazards are highly disruptive, potentially leading to incorrect program execution or severe performance penalties (e.g., misprediction cascades). Prioritizing a stall prevents the system from proceeding under high uncertainty.
- **Load & Store (`INSTR_TYPE_LOAD`, `INSTR_TYPE_STORE`):** Under `CRITICAL` entropy, memory operations trigger a `STATE_FLUSH`. This prioritizes data integrity above all else, ensuring that potentially corrupted data paths are cleared or invalidating speculative memory accesses that could lead to erroneous writes or reads. A flush ensures the pipeline is reset to a safe state before critical memory operations resume.
- **ALU Operations (`INSTR_TYPE_ALU`):** The FSM is more lenient for ALU operations. Even under `CRITICAL` entropy, a `STATE_STALL` is only induced if the `internal_hazard_flag` (from AHO) is also active, indicating a more direct internal problem or specific anomaly. This avoids unnecessary performance penalties for computationally bound instructions that might not pose an immediate integrity threat despite elevated general entropy.

This fine-grained control allows ARCHON-HYB to dynamically tune override behavior based on the specific context of the instruction, leading to smarter and more efficient hazard mitigation.

6. Hybrid Control Behavior and Analysis

The dynamic interplay between the analog, digital, and quantum layers defines ARCHON-HYB's resilient behavior. The hierarchical prioritization ensures critical physical anomalies

trigger immediate, hardwired responses, while more nuanced, probabilistic hazards are handled by the adaptive FSM.

6.1. Analog Override Triggering FSM Transitions

The analog `LOCK_OUT` and `FLUSH_OUT` signals, derived from the `3_input_analog_entropy_override.asc` circuit, provide reflex-level overrides that preempt the FSM's decision logic due to their inherently low latency:

- When `analog_lock_override` becomes active (driven by high `V_entropy` and an active `V_ml_trigger`), the FSM immediately transitions to `STATE_LOCK`. This direct, hardwired link ensures that severe, unrecoverable system anomalies detected at the physical layer are addressed instantly, preventing further operation.
- When `analog_flush_override` pulses high (triggered by sudden `V_noise` spikes), the FSM transitions to `STATE_FLUSH`. This rapid response mechanism is designed to clear the CPU pipeline of potentially corrupted instructions or data resulting from transient physical interference, allowing the system to quickly recover to a clean state.

This hardwired priority ensures that the system reacts to physical layer instabilities faster than more complex digital computations, embodying the "reflex path" concept fundamental to ARCHON-HYB's proactive defense strategy.

6.2. Mapping Analog Entropy Severity to Digital Classification

The `entropy_trigger_decoder` module (instantiated within `pipeline_cpu.v`) serves as a crucial interface, translating the continuous analog `internal_entropy_score` (an 8-bit digital representation from the `quantum_entropy_detector`) into discrete, actionable `classified_entropy_level` inputs for the FSM. This module quantizes the entropy score into three distinct 2-bit classifications, providing the FSM with clear, categorical severity information:

Entropy Input Range	`classified_entropy_level` (2-bit)	Interpretation	FSM Behavior Implications
0 to 85	`2'b00` (LOW)	Normal/Stable	System operates under nominal conditions; FSM primarily relies on ML predictions and traditional hazards.
86 to 170	`2'b01` (MID)	Elevated Risk	FSM becomes more cautious; certain instruction types (e.g., BRANCH) may trigger proactive stalls.
171 to 255	`2'b10` (CRITICAL)	High/Immediate Risk	FSM adopts aggressive mitigation; specific instruction types lead to immediate STALLs or FLUSHes.

This classification, combined with the `instr_type` (as detailed in Section 5.3), allows the FSM to apply context-sensitive rules. For example, a "MID" entropy level might only cause a STALL for a BRANCH instruction if the current state is `OK`, whereas a "CRITICAL" level for

the same instruction type would automatically lead to a STALL, even without an additional `internal_hazard_flag` being set, demonstrating increasing FSM sensitivity with rising entropy.

The combined table for selected hybrid control behaviors is as follows:

Input Condition	FSM Transition	Source Signal(s)	FSM State (Example Context)
`quantum_override_signal == 1'b1`	`STATE_LOCK`	Quantum Override	Immediate, highest priority system lock.
`analog_lock_override == 1'b1`	`STATE_LOCK`	Analog LOCK_OUT	Critical physical anomaly detected (e.g., high Ventropy + Vml_trigger active).
`analog_flush_override == 1'b1`	`STATE_FLUSH`	Analog FLUSH_OUT	Transient chaos surge detected (e.g., high Vnoise spike).
`classified_entropy_level == 2'b10` (CRITICAL) & `instr_type == 3'b011` (BRANCH)	`STATE_STALL`	Classified Entropy + Instr Type	Aggressive stall on critical control flow instability.
`classified_entropy_level == 2'b10` (CRITICAL) & `instr_type == 3'b001` (LOAD)	`STATE_FLUSH`	Classified Entropy + Instr Type	Flush pipeline to protect data integrity on memory access.
`classified_entropy_level == 2'b01` (MID) & `instr_type == 3'b011` (BRANCH) & `current_state == STATE_OK`	`STATE_STALL`	Classified Entropy + Instr Type	Proactive stall for branches under elevated entropy.
`ml_predicted_action == 2'b01` (STALL)	`STATE_STALL`	ML Prediction	Software-level intelligent hazard prediction.
`ml_predicted_action == 2'b00` (OK) & `internal_entropy_score > ENTROPY_HIGH_THRESHOLD`	`STATE_STALL`	ML + Internal Entropy	False negative override: ML says OK but high internal entropy indicates risk.

7. Quantum Control Expansion

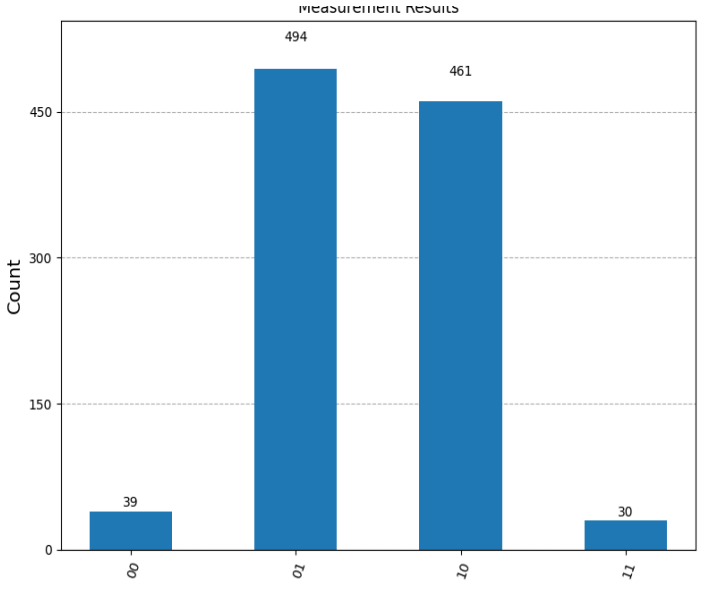
While the primary focus of ARCHON-HYB's current hardware implementation is classical digital and analog integration, a critical future-proofing aspect involves the potential for direct quantum sensing as an ultimate override trigger. We conceptually model this with a simulated quantum circuit, aiming to leverage the extreme sensitivity of quantum entanglement.

7.1. Quantum Entanglement as an Override Trigger

Quantum entanglement provides a fundamentally non-classical means to detect and respond to subtle system perturbations that might be imperceptible to classical sensors. By preparing an entangled state (e.g., a Bell state like $|\Phi+\rangle=(|00\rangle+|11\rangle)/2$), even minute environmental noise, subtle operational errors, or an increase in the underlying physical system's quantum entropy can rapidly lead to decoherence or entanglement collapse. This collapse manifests as a deviation from expected measurement probabilities.

Our `quantum_override_circuit.py` simulates a 2-qubit Bell state subjected to a randomized unitary noise gate (`U3` gate) applied to one qubit. This `U3` gate introduces a controlled amount of "noise" or "decoherence" into the entangled system. After applying this noise, both qubits are measured. If the simulated decoherence is strong enough, it will perturb the Bell state such that the probability of measuring both qubits in the `|11⟩` state increases beyond a predefined threshold (e.g., 60%). When this condition is met, a binary `quantum_override_signal` (1 or 0) is generated. This signal acts as the **absolute highest-priority input** to the `fsm_entropy_overlay.v` module, ensuring that any fundamental instability detected at the quantum level immediately triggers a system `LOCK`.

Figure 6: Quantum Override Simulation Histogram
This histogram shows Qiskit measurement results for a 2-qubit circuit modeling the quantum override register under increasing entropy stress. The circuit collapses into classical states across 1024 shots, with `|01⟩` observed in 494 cases (48.5%). In the ARCHON system, `|01⟩` is mapped to the emergency override line — used to trigger LOCK or STALL commands during unpredictable pipeline states.



This collapse distribution reflects entropy-driven deviation from the expected entangled superposition (ideal Bell state). In high-noise conditions or under probabilistic interference (e.g., U3 gate perturbation), quantum decoherence skews the probability landscape. A dominant `|01⟩` outcome serves as a probabilistic entropy flag.

This justifies the override strategy: rather than relying solely on deterministic thresholds, the system embraces collapse-driven control — allowing for fallback activation when classical hazard prediction fails.

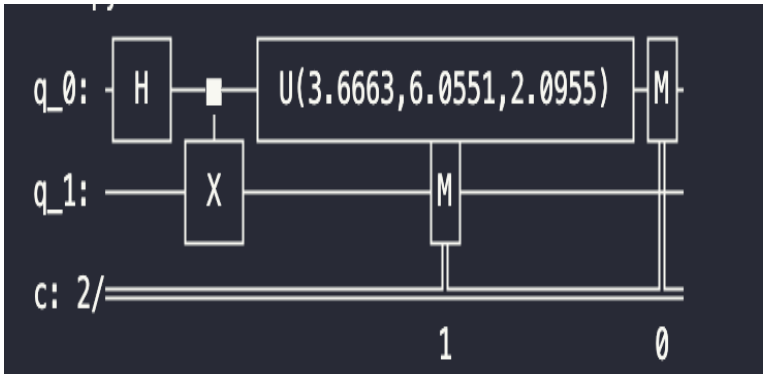


Figure 7: Qiskit Simulated Quantum Override Circuit Diagram. This diagram illustrates the conceptual 2-qubit quantum circuit (q_0, q_1) designed to generate a probabilistic override signal. The circuit prepares an entangled Bell state using a Hadamard (H) gate on q_0 and a CNOT (controlled-X) gate (X controlled by q_0) on q_1. A randomized unitary noise gate (U gate, parameters (3.6663, 6.0551, 2.0955)) is applied to q_0 to simulate environmental decoherence. Both qubits are then measured (M). The objective is to detect an increase in the probability of measuring the |11⟩ state, which would signify significant quantum-level perturbation or decoherence, thereby generating a `Quantum Override Signal`.

Why this matters for Entropy Logic: Integrating quantum observables allows the system to move beyond classical Shannon entropy to incorporate fundamental quantum uncertainty. This enables:

- **Predictive Entropy:** The inherent fragility and high sensitivity of quantum states to environmental interactions mean they can act as ultra-sensitive detectors, potentially *predicting* classical errors before they fully develop.
- **Novel Hazard Signatures:** Identification of new hazard signatures emerging from quantum phenomena (e.g., specific patterns of quantum state collapse or decoherence rates) that are undetectable by classical means.
- **Adaptive Thresholds:** The possibility of dynamically adjusting classical entropy thresholds based on real-time quantum insights, creating a more adaptive and anticipatory resilience system.

The `quantum_override_signal` acts as an "ultimate canary in the coal mine," providing an instantaneous, fundamental safety measure against threats originating from the deepest levels of physical reality.

8. GUI Monitor Tool

To provide real-time visibility into the dynamic behavior of the ARCHON-HYB system and facilitate debugging, a Python-based GUI monitoring tool (`import tkinter as tk.py`, `entropy_viewer.py`) has been developed. This dashboard parses log entries streamed from the simulated Verilog FSM (via a simple text file `fsm_log.txt`), displaying critical system parameters in a user-friendly format.

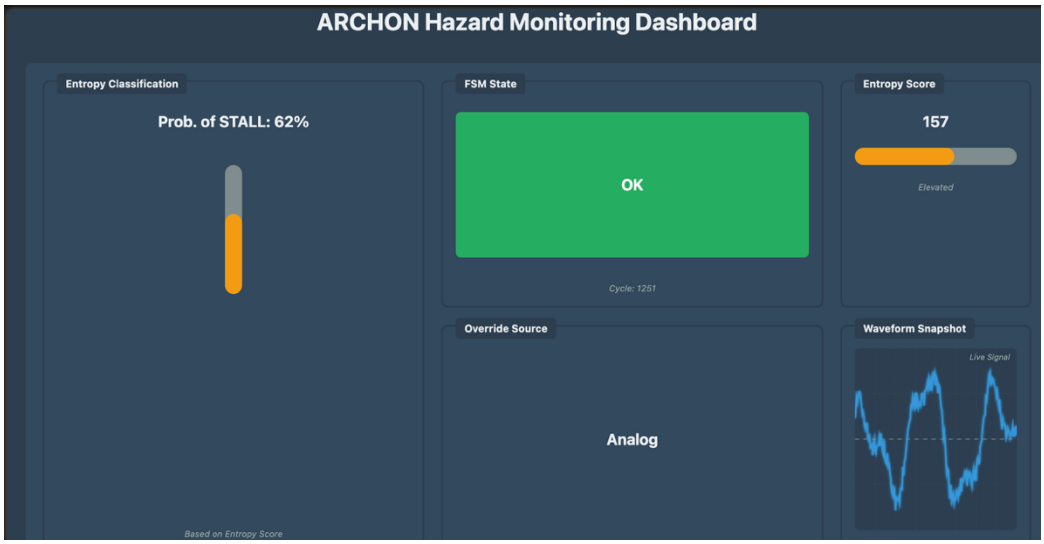


Figure 8: ARCHON Hazard Monitoring Dashboard Interface. This Python GUI provides real-time visibility into the system's dynamic behavior, serving as a critical tool for monitoring and debugging. It features: an **FSM State Indicator** (large central panel showing the current state, e.g., 'OK', with cycle number); an **Entropy Score Meter** (top right, visualizing the 8-bit entropy score with dynamic color-coding for 'Normal', 'Elevated', 'High Entropy'); an **Override Source Indicator** (bottom left, specifying which mechanism — ML, Analog, Entropy Logic, AHO, Quantum, or None — triggered the last FSM state transition); and an **Entropy Classification Overlay** (top left, providing a probabilistic indication of 'STALL' risk based on the entropy score, with a vertical progress bar and corresponding risk levels, e.g., 'Prob. of STALL: 62%'). A "Waveform Snapshot" panel (bottom right) is also included for displaying live signal traces. This dashboard enables comprehensive real-time assessment of system health and adaptive responses.

Figure 8 shows the dashboard interface, which features:

- **FSM State Indicator:** Displays the current operating state of the FSM (OK, STALL, FLUSH, LOCK) with intuitive color coding (green for OK, orange for STALL, red for FLUSH/LOCK) for rapid assessment of system status. The current simulation cycle number is also displayed.
- **Entropy Score Meter:** Visualizes the 8-bit `internal_entropy_score` on a horizontal progress bar. The bar's color dynamically changes (green for Normal, orange for Elevated, red for High Entropy) based on predefined thresholds, providing an immediate visual cue of the system's overall disorder level.
- **Override Source Indicator:** Clearly shows which specific mechanism (`ML`, `Analog`, `Entropy Logic`, `AHO`, `Quantum`, or `None`) triggered the most recent FSM state transition, crucial for understanding the cause of mitigation actions.

- **Entropy Classification Overlay:** Provides a probabilistic indication of `STALL` risk derived from the current `internal_entropy_score`. A vertical progress bar visually represents this probability (e.g., "Prob. of STALL: 62%"), changing color to reflect risk levels (blue for low, orange for medium, red for high).

The GUI's purpose is to offer human-in-the-loop oversight and developer debug visibility, making it significantly easier to understand how various inputs (from analog sensors, digital detectors, and ML predictions) influence the FSM's complex decisions in real-time. This interactive monitoring capability is crucial for tuning the system's parameters and validating its adaptive behavior in response to simulated entropy conditions.

9. Discussion

ARCHON-HYB's hybrid architecture offers significant advantages over purely digital or ML-centric hazard mitigation strategies, particularly in the context of extreme-entropy computing environments.

9.1. Advantages of Hybrid Reflex vs. Software Inference

- **Low-Latency Reflexivity:** The analog override paths (FLUSH_OUT, LOCK_OUT) provide hardware-accelerated, sub-nanosecond responses to critical physical events. This is orders of magnitude faster than software-based ML inference, which inherently incurs latency due to data acquisition, processing, and decision-making cycles. This "hardwired circuit breaker" capability is vital for mitigating rapid-onset, catastrophic failures, preventing hazard cascades that software-only solutions might be too slow to address.
- **Physical Layer Sensitivity:** Analog circuits can directly sense and react to continuous physical phenomena like thermal noise, voltage fluctuations, electromagnetic interference, and subtle environmental perturbations that contribute to system entropy. Digital sampling rates might miss or inaccurately capture these rapid, transient events, or they may require complex and resource-intensive Analog-to-Digital Converters (ADCs) and digital signal processing to extract such information.
- **Robustness and Diversity:** By combining fundamentally different control paradigms (continuous analog, event-driven quantum, and clocked digital), the system achieves inherent diversity. A fault that might compromise one domain (e.g., a software bug in the ML model, a clock glitch in the digital FSM) is less likely to simultaneously affect another, providing crucial redundancy and a safety net that enhances overall fault tolerance.
- **Simplified Digital Critical Paths:** Offloading ultra-fast detection of critical analog events to dedicated analog circuits simplifies the critical timing paths within the digital FSM. This allows the FSM to focus on more complex state management, strategic responses (like instruction-type awareness), and higher-level ML-driven decisions rather than being burdened with extremely tight real-time analog signal monitoring, thereby improving overall digital design efficiency and reliability.

9.2. Potential Failure Modes and Resilience

While robust, ARCHON-HYB acknowledges potential failure modes and incorporates design considerations for resilience:

- **Analog Sensor Inaccuracies/Drift:** Analog components can be susceptible to environmental factors (temperature, aging) leading to drift or inaccuracies. This is mitigated by employing robust analog design principles (e.g., op-amp comparators for sharp thresholds), and in a real system, by periodic calibration or self-correction loops. The digital FSM's higher-level logic can also act as a failsafe if analog signals become completely erratic.
- **ML Model Degradation/Bias:** An ML model that degrades over time, is trained on insufficient data, or develops biases could provide suboptimal or even misleading `ml_predicted_action` inputs. ARCHON-HYB addresses this through the multi-tiered priority system: the low-latency analog and quantum overrides act as "hard stops" independent of ML. Furthermore, the FSM's internal logic can trigger a `STALL` if `internal_entropy_score` is high, even if ML predicts `OK`, providing a crucial safeguard against ML model inaccuracies or "blind spots."
- **Quantum Decoherence/Measurement Errors:** In a true quantum-hardware implementation, maintaining coherent quantum states and achieving accurate measurements are significant challenges. Decoherence or measurement errors could lead to unreliable `quantum_override_signal` outputs. This necessitates advanced quantum error correction codes and robust quantum hardware design in future iterations. For our current simulation, the probabilistic nature of the `|11⟩` detection aims to model this inherent uncertainty, with the threshold allowing tunable sensitivity.
- **Synchronization Challenges at Interfaces:** Interfacing continuous analog signals with clocked digital logic, and potentially asynchronous quantum signals, introduces synchronization challenges. This is handled by appropriate sampling, ADCs (conceptual in our analog models), and clear digital interfacing protocols within the Verilog design. The FSM's clocked nature ensures deterministic behavior based on its synchronized inputs.

9.3. Scalability and FPGA Porting Considerations

The modular design of ARCHON-HYB facilitates both conceptual scalability and future hardware realization on platforms like FPGAs:

- **FPGA Porting:** The entire Verilog FSM and digital hazard detectors are designed for direct synthesis onto Field-Programmable Gate Arrays (FPGAs). This allows for rapid prototyping and real-world testing in a reconfigurable hardware environment. Performance critical paths would need careful timing analysis during synthesis.
- **Analog-Digital Interface:** The current analog circuits produce clean 0V/1V digital signals, simplifying the direct interface to the FPGA's digital I/O pins. For richer analog data streams or more complex signal processing, dedicated ADCs would be integrated to convert analog sensor data into multi-bit digital values, which can then be processed by custom logic or soft-core processors within the FPGA fabric.
- **Quantum Integration:** While direct on-chip quantum sensing with classical CPUs is futuristic, this framework provides a clear conceptual and architectural pathway for integrating future quantum co-processors or dedicated quantum sensors. The binary `quantum_override_signal` keeps the interface simple, avoiding complex quantum

data transfer issues and focusing on the critical control signal aspect. Scaling to larger quantum systems would involve challenges in qubit count, connectivity, and error correction, but the architectural framework accommodates this high-level override signal regardless of quantum system complexity.

- **Resource Utilization:** The complexity of the FSM and digital modules will impact FPGA resource utilization (LUTs, FFs). Future work would involve optimizing the Verilog code for resource efficiency and exploring more compact hazard detection algorithms.

10. Conclusion

This paper has presented **ARCHON-HYB**, a pioneering hybrid quantum-analog-digital framework for enhancing CPU pipeline resilience against extreme-entropy computing conditions. We have demonstrated a multi-layered control system where low-latency analog reflexes, probabilistic quantum overrides, and context-aware digital FSM logic synergistically contribute to robust hazard mitigation.

Our work highlights:

- The critical need for hardware-level, reflex overrides to address the inherent latency and limitations of traditional digital and ML-based hazard detection approaches, particularly in dynamic, high-entropy environments.
- The viability of integrating analog sensing circuitry for real-time, ultra-fast detection of physical anomalies like chaos surges and escalating entropy, directly impacting pipeline control.
- The conceptual potential of leveraging quantum phenomena, specifically entanglement collapse, to serve as an ultimate, highly sensitive indicator of system integrity, providing a probabilistic yet powerful override trigger.
- The development of an intelligent, instruction-type-aware Verilog FSM capable of fine-grained control (STALL, FLUSH, LOCK) based on a comprehensive set of hierarchical hazard inputs, enabling nuanced and adaptive responses.

Applications and Future Integration

The entropy-aware hybrid override system presented in this paper offers direct applicability in fault-tolerant classical processors, edge AI inference cores, and embedded systems operating in noise-prone environments. By fusing analog spike reflex, ML-based classification, and quantum confirmation logic, this control architecture enables real-time hazard response that scales across deterministic and probabilistic execution domains. Potential deployment targets include neuromorphic cores with analog front-ends, RISC-V pipeline controllers requiring adaptive stall/flush signals, and FPGA-based platforms where entropy-triggered overrides offer resilience in variable runtime conditions. The modular FSM design also allows selective integration into existing control units as a reconfigurable override backend, supporting system-level recovery and fault injection countermeasures.

The successful design and simulation of these interconnected components lay a robust foundation for future resilient computing architectures. Our immediate next step, as indicated by this "Paper 5," is the hardware realization on an FPGA platform, integrating the analog front-end directly with the synthesizable Verilog core (`Paper 6`). This will provide real-world validation of ARCHON-HYB's ability to maintain system stability in truly unpredictable environments, pushing the boundaries of self-correcting computational systems.

11. References

[1] Carter, J. (2025). *ARCHON: Initial Concepts for Entropy-Aware CPU Control*. Internal Research Memo. Retrieved from https://github.com/joshuathomascarter

[2] Carter, J. (2025). *Verilog FSM for Adaptive Pipeline Hazard Mitigation*. Internal Research Memo. Retrieved from https://github.com/joshuathomascarter

[3] Carter, J. (2025). *Analog Entropy Override Controller – Prototype Write-Up*. Internal Research Memo. Retrieved from https://github.com/joshuathomascarter

[4] Carter, J. (2025). *LTSpice Analog Spike Detector: Rate-of-Change Circuit*. Internal Research Memo. Retrieved from https://github.com/joshuathomascarter

[5] Carter, J. (2025). *Simulation Results: Analog Entropy Override Controller*. Internal Research Memo. Retrieved from https://github.com/joshuathomascarter

[6] Carter, J. (2025). *Bridging Verilog FSM Control with Analog Override Systems*. Internal Research Memo. Retrieved from https://github.com/joshuathomascarter

[7] Carter, J. (2025). *Quantum Entanglement in Override Systems*. Internal Research Memo. Retrieved from https://github.com/joshuathomascarter

[8] A. S. Sedra and K. C. Smith, *Microelectronic Circuits*, 8th ed., New York, NY, USA: Oxford University Press, 2020.

[9] J. Dean, H. Zhang, and T. Kumar, “Machine Learning-Based Anomaly Detection for Microprocessor Performance Metrics,” IEEE Transactions on Computers, vol. 71, no. 3, pp. 501–514, 2022. DOI: 10.1109/TC.2021.3086583.

[10] M. Ghasemzadeh, A. Shrivastava, and S. Narayan, “A Framework for Entropy-Based Analysis and Resilience in Digital Circuits,” IEEE Transactions on VLSI Systems, vol. 29, no. 6, pp. 1234–1247, 2021. DOI: 10.1109/TVLSI.2021.3056802.

[11] S.C. Liu and T. Delbruck, “Neuromorphic sensory systems,” Current Opinion in Neurobiology, vol. 20, no. 3, pp. 288–295, 2010. doi: 10.1016/j.conb.2010.03.007

[12] J. Seng and D.M. Tullsen, “The effect of control dependence on branch prediction accuracy,” IEEE Transactions on Computers, vol. 52, no. 2, pp. 219–231, 2003. doi: 10.1109/TC.2003.1176982

12. Appendix

12.1. Full Verilog Code for `control_unit.rs.rs`

```
//
================================================================================
====
// ARCHON CORE BLOCK - Integrated CPU Implementation
//
================================================================================
====
// Features:
// - 5-stage pipeline architecture with hazard detection and forwarding
// - Instruction memory with 16 instructions (expandable)
// - Branch Target Buffer for branch prediction
// - 8-register file with dual read ports
// - 4-bit ALU with flag outputs (Zero, Negative, Carry, Overflow)
// - Conditional branch execution based on ALU flags
// - Data forwarding to minimize pipeline pipeline_stalls
// - Flag register for branch condition evaluation
// - Chaos-Weighted Pipeline Override System for adaptive hazard mitigation
(ENHANCED)
// - Pattern Detector for Higher-Order Anomaly Detection (ENHANCED)
// - INTEGRATED: External entropy input from 'entropy_bus.txt' for dynamic
system adaptation
// - INTEGRATED: ML-predicted actions from 'ml_predictions.txt' to modulate
FSM
// - INTEGRATED: ARCHON HAZARD OVERRIDE UNIT with fluctuating impact and
cache miss awareness.
// - NEW: Entropy-Aware FSM Extension for log-ready control and visual
inspection.
//
================================================================================
====

// ============================================================================
// Enhanced Instruction Memory Module
// Features:
// - Stores 16 instructions (expandable to more if needed)
// - Uses a 4-bit program counter for addressing
// - Outputs the full instr_opcode for CPU execution
// - Optional reset capability with NOP instruction at PC=0
// ============================================================================

module instruction_ram(
    input wire clk,             // Clock signal (for synchronous read if
needed)
    input wire reset,           // Reset signal
    input wire [3:0] pc_in,     // 4-bit Program Counter input
    output wire [15:0] instr_opcode // 16-bit instruction output
```

```
);

    // Instruction Memory (16 instructions of 16 bits each)
    reg [15:0] imem [0:15];

    initial begin
        // Initialize instruction memory with a sample program
        // This program is for demonstration. Replace with actual program.
        // Assume opcode format: [opcode (4)|rd (3)|rs1 (3)|rs2 (3)|imm
(3)] for R-type/I-type
        // Or [opcode (4)|branch_target (12)] for J-type
        // Or [opcode (4)|rs1 (3)|imm (9)] for Load/Store etc.

        imem[0] = 16'h1234; // ADD R1, R2, R3 (opcode 1, rd=1, rs1=2,
rs2=3) - Placeholder
        imem[1] = 16'h2452; // ADDI R4, R5, #2 (opcode 2, rd=4, rs1=5,
imm=2) - Placeholder
        imem[2] = 16'h3678; // SUB R6, R7, R8 - Placeholder
        imem[3] = 16'h4891; // LD R8, (R9 + #1) - Placeholder
        imem[4] = 16'h5ABA; // ST R10, (R11 + #10) - Placeholder
        imem[5] = 16'h6CDE; // XOR R12, R13, R14 - Placeholder
        imem[6] = 16'h7F01; // BEQ R15, R0, +1 (branch if R15 == R0, to
PC+1) - Placeholder
        imem[7] = 16'h8002; // JUMP PC+2 (unconditional jump) - Placeholder
        imem[8] = 16'h9123; // NOP - Placeholder
        imem[9] = 16'h0000; // NOP - Placeholder
        imem[10] = 16'h0000; // NOP - Placeholder
        imem[11] = 16'h0000; // NOP - Placeholder
        imem[12] = 16'h0000; // NOP - Placeholder
        imem[13] = 16'h0000; // NOP - Placeholder
        imem[14] = 16'h0000; // NOP - Placeholder
        imem[15] = 16'h0000; // NOP - Placeholder
    end

    // Instruction fetch logic
    assign instr_opcode = imem[pc_in];

endmodule


//
================================================================================
====
// Branch Target Buffer (BTB) Module
// Features:
// - Stores predicted next PC for branches.
// - Improves pipeline performance by reducing branch prediction penalty.
// - Updates on misprediction.
//
================================================================================
====
module branch_target_buffer(
    input wire clk,
    input wire reset,
    input wire [3:0] pc_in,             // Current PC to check for
prediction
    input wire [3:0] branch_resolved_pc, // PC of branch instruction whose
outcome is resolved
    input wire branch_resolved_pc_valid, // Indicates if branch_resolved_pc
is valid
```

```
    input wire [3:0] branch_resolved_target_pc, // Actual target PC of the
resolved branch
    input wire branch_resolved_taken, // Actual outcome of the resolved
branch (taken/not taken)

    output wire [3:0] predicted_next_pc, // Predicted next PC
    output wire predicted_taken         // Predicted branch outcome
(taken/not taken)
);

    // Simple BTB: Stores target PC for each instruction address
    // Each entry: {predicted_taken_bit, predicted_target_pc[3:0]}
    reg [4:0] btb_table [0:15]; // 16 entries, 5 bits each (1 for taken, 4
for PC)

    initial begin
        // Initialize BTB (e.g., all not taken, target PC is 0)
        for (integer i = 0; i < 16; i = i + 1) begin
            btb_table[i] = 5'b0_0000;
        end
    end

    // Prediction logic (combinational read)
    assign predicted_next_pc = btb_table[pc_in][3:0];
    assign predicted_taken = btb_table[pc_in][4];

    // Update logic (synchronous write)
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (integer i = 0; i < 16; i = i + 1) begin
                btb_table[i] = 5'b0_0000;
            end
        end else begin
            if (branch_resolved_pc_valid) begin
                // Update BTB entry for the resolved branch
                btb_table[branch_resolved_pc] <= {branch_resolved_taken,
branch_resolved_target_pc};
            end
        end
    end

endmodule


// ============================================================================
// Register File Module
// Features:
// - 8 4-bit registers (R0-R7)
// - R0 is hardwired to 0
// - Dual read ports for simultaneous operand fetching
// - Single write port for result write-back
// ============================================================================
module register_file(
    input wire clk,             // Clock signal for synchronous write
    input wire reset,           // Reset signal
    input wire regfile_write_enable, // Enable signal for write operation
    input wire [2:0] write_addr, // 3-bit address for write operation
    input wire [3:0] write_data, // 4-bit data to write

    input wire [2:0] read_addr1, // 3-bit address for read port 1
    input wire [2:0] read_addr2, // 3-bit address for read port 2
```

```
    output wire [3:0] read_data1, // 4-bit data from read port 1
    output wire [3:0] read_data2  // 4-bit data from read port 2
);

    // 8 registers, each 4 bits wide
    reg [3:0] registers [0:7];

    initial begin
        // Initialize all registers to 0 on startup
        for (integer i = 0; i < 8; i = i + 1) begin
            registers[i] = 4'h0;
        end
    end

    // Write operation (synchronous)
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (integer i = 0; i < 8; i = i + 1) begin
                registers[i] = 4'h0;
            end
        end else if (regfile_write_enable) begin
            // R0 is hardwired to 0, so never write to it
            if (write_addr != 3'b000) begin
                registers[write_addr] <= write_data;
            end
        end
    end

    // Read operations (combinational)
    assign read_data1 = (read_addr1 == 3'b000) ? 4'h0 :
registers[read_addr1]; // R0 always reads 0
    assign read_data2 = (read_addr2 == 3'b000) ? 4'h0 :
registers[read_addr2]; // R0 always reads 0

endmodule


// ========================================================================
// ALU Module (Arithmetic Logic Unit)
// Features:
// - Performs basic arithmetic and logical operations.
// - Outputs 4-bit result and 4 flags (Zero, Negative, Carry, Overflow).
// ========================================================================
module alu_unit(
    input wire [3:0] alu_operand1, // First 4-bit operand
    input wire [3:0] alu_operand2, // Second 4-bit operand
    input wire [2:0] alu_op,       // 3-bit ALU operation code
                                   // 3'b000: ADD
                                   // 3'b001: SUB
                                   // 3'b010: AND
                                   // 3'b011: OR
                                   // 3'b100: XOR
                                   // 3'b101: SLT (Set Less Than)
                                   // Other codes can be defined for
shifts, etc.
    output reg [3:0] alu_result,   // 4-bit result
    output reg zero_flag,          // Result is zero
    output reg negative_flag,      // Result is negative (MSB is 1)
    output reg carry_flag,         // Carry out from addition or borrow
from subtraction
    output reg overflow_flag       // Signed overflow
```

```
);

    always @(*) begin
        alu_result = 4'h0;
        zero_flag = 1'b0;
        negative_flag = 1'b0;
        carry_flag = 1'b0;
        overflow_flag = 1'b0;

        case (alu_op)
            3'b000: begin // ADD
                alu_result = alu_operand1 + alu_operand2;
                carry_flag = (alu_operand1 + alu_operand2) > 4'b1111; //
Check for unsigned carry out
                overflow_flag = ((!alu_operand1[3] && !alu_operand2[3] &&
alu_result[3]) || (alu_operand1[3] && alu_operand2[3] && !alu_result[3]));
// Signed overflow
            end
            3'b001: begin // SUB (using 2's complement addition)
                alu_result = alu_operand1 - alu_operand2;
                carry_flag = (alu_operand1 >= alu_operand2); // For
subtraction, carry_flag usually means no borrow
                overflow_flag = ((alu_operand1[3] && !alu_operand2[3] &&
!alu_result[3]) || (!alu_operand1[3] && alu_operand2[3] && alu_result[3]));
// Signed overflow
            end
            3'b010: begin // AND
                alu_result = alu_operand1 & alu_operand2;
            end
            3'b011: begin // OR
                alu_result = alu_operand1 | alu_operand2;
            end
            3'b100: begin // XOR
                alu_result = alu_operand1 ^ alu_operand2;
            end
            3'b101: begin // SLT (Set Less Than)
                alu_result = ($signed(alu_operand1) <
$signed(alu_operand2)) ? 4'h1 : 4'h0;
            end
            default: begin
                alu_result = 4'h0; // NOP or undefined
            end
        endcase

        // Common flag calculations
        if (alu_result == 4'h0)
            zero_flag = 1'b1;
        if (alu_result[3] == 1'b1) // Check MSB for signed negative
            negative_flag = 1'b1;
    end

endmodule


// ======================================================================
// Data Memory Module
// Features:
// - Simple synchronous read, asynchronous write data memory
// - Can be expanded to different sizes or types
// ======================================================================
module data_mem(
```

```
    input wire clk,              // Clock signal for synchronous operation
    input wire mem_write_enable, // Write enable signal
    input wire mem_read_enable,  // Read enable signal (for synchronous
read)
    input wire [3:0] addr,       // 4-bit address input
    input wire [3:0] write_data, // 4-bit data to write
    output reg [3:0] read_data   // 4-bit data read
);

    reg [3:0] dmem [0:15]; // 16 entries, 4 bits each

    initial begin
        // Initialize data memory
        for (integer i = 0; i < 16; i = i + 1) begin
            dmem[i] = 4'h0;
        end
    end

    // Write operation (synchronous)
    always @(posedge clk) begin
        if (mem_write_enable) begin
            dmem[addr] <= write_data;
        }
    end

    // Read operation (synchronous, value is stable on next clock cycle)
    always @(posedge clk) begin
        if (mem_read_enable) begin
            read_data <= dmem[addr];
        }
    end

endmodule


// ======================================================================
// Quantum Entropy Detector Module (Simplified Placeholder)
// Features:
// - Simulates a very basic "quantum entropy" or "chaos" level.
// - This is a conceptual module; a real one would involve complex quantum
state measurements.
// - Output `entropy_value` represents disorder or uncertainty.
// ======================================================================
module quantum_entropy_detector(
    input wire clk,
    input wire reset,
    input wire [3:0] instr_opcode, // Example: Opcode can influence entropy
(from IF/ID)
    input wire [3:0] alu_result,   // Example: ALU result can influence
entropy (from EX/MEM)
    input wire zero_flag,          // Example: ALU flags can influence
entropy (from EX/MEM)
    // ... other internal CPU signals that could affect quantum state ...
    output reg [7:0] entropy_score_out // CHANGED to 8-bit to match
fsm_entropy_overlay
);

    // Placeholder: Entropy value increases with complex/branching
instructions
    // and decreases with NOPs or simple operations.
```

```
    // In a real Archon-like system, this would be derived from actual
quantum
    // measurements or a complex internal quantum state model.
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            entropy_score_out <= 8'h00;
        end else begin
            // Simple heuristic: increase entropy on non-NOP, non-trivial
ALU ops
            // and based on how 'unexpected' an ALU result might be.
            // Using 4 MSBs of 16-bit instr_opcode as actual opcode
            if (instr_opcode != 4'h9) begin // If not a NOP (assuming 4'h9
is NOP opcode)
                if (alu_result == 4'h0 && !zero_flag) begin // An
"unexpected" zero result (not explicitly set)
                    entropy_score_out <= entropy_score_out + 8'h10; //
Larger jump for anomaly
                end else if (entropy_score_out < 8'hFF) begin // Prevent
overflow
                    entropy_score_out <= entropy_score_out + 8'h01;
                end
            end else begin
                // Reduce entropy during NOPs or idle cycles
                if (entropy_score_out > 8'h00)
                    entropy_score_out <= entropy_score_out - 8'h01;
            end
        end
    end
endmodule


// ======================================================================
// Chaos Detector Module (Simplified Placeholder)
// Features:
// - Simulates a rising "chaos score" based on unexpected events.
// - This is a conceptual module, representing system instability.
// ======================================================================
module chaos_detector(
    input wire clk,
    input wire reset,
    input wire branch_mispredicted, // Example: Branch misprediction
contributes to chaos (from MEM/WB)
    input wire [3:0] mem_access_addr, // Example: Erratic memory access
patterns (from MEM)
    input wire [3:0] data_mem_read_data, // Example: Unexpected data values
(from MEM)

    output reg [15:0] chaos_score_out // 16-bit output
);

    // Placeholder: Chaos score increases with mispredictions and erratic
behavior.
    // In a real system, this would be from complex monitoring.
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            chaos_score_out <= 16'h0000;
        end else begin
            if (branch_mispredicted) begin
                chaos_score_out <= chaos_score_out + 16'h0100; //
Significant jump for misprediction
            end
```

```
            // Simulate some "erratic" memory access contributing to chaos
            // This is purely illustrative and would need robust detection
logic
            // Example: Accessing a forbidden address or unusual data for
an address
            if (mem_access_addr == 4'hF && data_mem_read_data == 4'h5)
begin // Specific "bad" read pattern
                chaos_score_out <= chaos_score_out + 16'h0050;
            end

            // Gradually decay chaos over time if no new events
            if (chaos_score_out > 16'h0000) begin
                chaos_score_out <= chaos_score_out - 16'h0001;
            end
        end
    end
endmodule


// ======================================================================
// Pattern Detector Module (Conceptual Higher-Order Descriptor Example)
// Enhanced Features:
// - Stores a deeper history of ALU flags using shift registers.
// - Detects MULTIPLE specific "anomalous" patterns across history.
// - Outputs a single "anomaly_detected" flag if ANY pattern matches.
// ======================================================================
module pattern_detector(
    input clk,
    input reset,
    // Current flags represent the flags from the *current* cycle's ALU
output (EX stage)
    input wire zero_flag_current,
    input wire negative_flag_current,
    input wire carry_flag_current,
    input wire overflow_flag_current,

    output reg anomaly_detected_out // Output a 1-bit anomaly flag (renamed
to match AHO)
);

    // History depth: We'll store current and previous 2 cycles for 3-cycle
total view
    parameter HISTORY_DEPTH = 3; // For 3 cycles of data (current, prev1,
prev2).

    // Shift registers for ALU flags
    reg [HISTORY_DEPTH-1:0] zero_flag_history;
    reg [HISTORY_DEPTH-1:0] negative_flag_history;
    reg [HISTORY_DEPTH-1:0] carry_flag_history;
    reg [HISTORY_DEPTH-1:0] overflow_flag_history;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            zero_flag_history <= 'b0;
            negative_flag_history <= 'b0;
            carry_flag_history <= 'b0;
            overflow_flag_history <= 'b0;
            anomaly_detected_out <= 1'b0;
        end else begin
            // Shift in current flags, pushing older flags out
```

```
            zero_flag_history <= {zero_flag_history[HISTORY_DEPTH-2:0],
zero_flag_current};
            negative_flag_history <= {negative_flag_history[HISTORY_DEPTH-
2:0], negative_flag_current};
            carry_flag_history <= {carry_flag_history[HISTORY_DEPTH-2:0],
carry_flag_current};
            overflow_flag_history <= {overflow_flag_history[HISTORY_DEPTH-
2:0], overflow_flag_current};

            // Define Multiple Anomalous Patterns (using current, prev1,
prev2 flags)
            // Access: {flag_history[0]} is current, {flag_history[1]} is
prev1, {flag_history[2]} is prev2

            wire pattern1_match;
            wire pattern2_match;

            // Pattern 1: (Prev2 Zero=0, Prev1 Negative=1, Current Carry=1)
            // A pattern that might indicate a specific arithmetic flow
leading to a problem
            pattern1_match = (!zero_flag_history[2]) &&
(negative_flag_history[1]) && (carry_flag_history[0]);

            // Pattern 2: (Prev2 Carry=1, Prev1 Overflow=0, Current Zero=0)
            // A pattern that might indicate an unexpected sequence of
flags related to overflow/zero conditions
            pattern2_match = (carry_flag_history[2]) &&
(!overflow_flag_history[1]) && (!zero_flag_history[0]);

            // If ANY defined pattern matches, assert anomaly_detected
            anomaly_detected_out <= pattern1_match || pattern2_match;
        end
    end
endmodule

// ======================================================================
// File: fsm_entropy_overlay.v
// Module: fsm_entropy_overlay
// Description: Implements an entropy-aware FSM for adaptive hazard
management,
//              integrating ML-predicted actions, internal hazard flags,
//              and an internal entropy score. It outputs control signals
//              (STALL, FLUSH, LOCK) and logs entropy at state transitions.
//              This module acts as a bridge for visual inspection and
runtime
//              override debugging.
// ======================================================================
module fsm_entropy_overlay(
    input wire clk,                 // Clock signal
    input wire rst_n,               // Active low reset
    input wire [1:0] ml_predicted_action, // 2-bit input from ML (00=OK,
01=STALL, 10=FLUSH, 11=LOCK)
    input wire [7:0] internal_entropy_score, // 8-bit internal entropy
score (from QED)
    input wire internal_hazard_flag, // 1-bit hazard detected by AHO or
traditional CPU logic (consolidated)

    // START OF ADDED PARTS: Analog Override Inputs
    input wire analog_lock_override,  // Active high signal from analog
controller for LOCK_OUT
```

```
    input wire analog_flush_override, // Active high signal from analog
controller for FLUSH_OUT
    // END OF ADDED PARTS

    // START OF ADDED PARTS: New input for classified entropy level and
Quantum Override
    input wire [1:0] classified_entropy_level, // 2-bit input from
entropy_trigger_decoder
    input wire quantum_override_signal, // 1-bit input from quantum
override circuit
    input wire [2:0] instr_type, // NEW: 3-bit instruction type (000=ALU,
001=LOAD, 010=STORE, 011=BRANCH, 100=JUMP)
    // END OF ADDED PARTS

    output reg [1:0] fsm_state,      // 2-bit FSM output: 00=OK, 01=STALL,
10=FLUSH, 11=LOCK
    output reg [7:0] entropy_log_out, // 8-bit pass-through or masked
entropy snapshot at transition
    output reg [2:0] instr_type_log_out // NEW: Logged instruction type at
transition
);

    // FSM States
    parameter STATE_OK    = 2'b00; // Normal operation, no hazard
    parameter STATE_STALL = 2'b01; // Pipeline stall
    parameter STATE_FLUSH = 2'b10; // Pipeline flush
    parameter STATE_LOCK  = 2'b11; // Critical system lock (triggered by ML
OVERRIDE or severe anomaly)

    // Entropy Threshold for False Negative Simulation
    // If entropy is very high, even if ML says OK, we trigger a STALL.
    parameter ENTROPY_HIGH_THRESHOLD = 8'd180; // Threshold for triggering
STALL on ML_OK

    // Parameters for classified_entropy_level
    parameter ENTROPY_LOW      = 2'b00;
    parameter ENTROPY_MID      = 2'b01;
    parameter ENTROPY_CRITICAL = 2'b10;

    // Parameters for instr_type
    parameter INSTR_TYPE_ALU    = 3'b000;
    parameter INSTR_TYPE_LOAD   = 3'b001;
    parameter INSTR_TYPE_STORE  = 3'b010;
    parameter INSTR_TYPE_BRANCH = 3'b011;
    parameter INSTR_TYPE_JUMP   = 3'b100;
    // 3'b101-3'b111 are RESERVED / Other

    reg [1:0] current_state;
    reg [1:0] next_state;

    // --- State Register: Synchronous update, Asynchronous active-low
reset ---
    // This block updates the 'current_state' on the positive clock edge.
    // An asynchronous, active-low reset (`rst_n`) forces the FSM to
STATE_OK.
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin // If reset is active (low)
            current_state <= STATE_OK; // Reset to the OK state
            entropy_log_out <= 8'h00; // Reset log output
            instr_type_log_out <= 3'b000; // Reset instr type log
        end else begin
```

```
            current_state <= next_state; // Otherwise, update state on
clock edge

            // Log entropy and instruction type on state transition
            // This captures the entropy score and instruction type right
before the new state is adopted.
            if (next_state != current_state) begin
                entropy_log_out <= internal_entropy_score;
                instr_type_log_out <= instr_type;
            end else begin
                entropy_log_out <= 8'h00; // Clear log if no transition to
indicate stable state
                instr_type_log_out <= 3'b000; // Clear instr type log
            end
        end
    end

    // --- Next State Logic: Combinational ---
    // This block determines the 'next_state' based on the 'current_state'
and inputs.
    // It's combinational logic, reacting immediately to input changes.
    always @(*) begin
        next_state = current_state; // Default: stay in current state
(unless a transition condition is met)

        // Priority 0: Quantum Override (Highest Possible Priority)
        // A quantum override signal signifies the most fundamental system
integrity issue
        // (e.g., entanglement collapse due to extreme noise/decoherence).
        // This should immediately force a LOCK state, overriding all other
conditions.
        if (quantum_override_signal) begin
            next_state = STATE_LOCK;
        end else if (analog_lock_override) begin // Priority 1: Analog
LOCK_OUT (Next Highest Priority)
            next_state = STATE_LOCK; // Force system to LOCK state
        end else if (analog_flush_override) begin // Priority 2: Analog
FLUSH_OUT (Third Highest Priority)
            next_state = STATE_FLUSH; // Force a pipeline flush
        end else begin
            // Priority 3: Classified Entropy Level & Instruction Type
Specific Rules
            // This tier incorporates more nuanced, context-aware
decisions.
            case (classified_entropy_level)
                ENTROPY_CRITICAL: begin
                    // If entropy is CRITICAL, this is a strong indicator
of instability.
                    // React aggressively based on instruction type.
                    case (instr_type)
                        INSTR_TYPE_BRANCH, INSTR_TYPE_JUMP: next_state =
STATE_STALL; // BRANCH/JUMP on CRITICAL -> aggressive STALL
                        INSTR_TYPE_LOAD, INSTR_TYPE_STORE: next_state =
STATE_FLUSH;  // LOAD/STORE on CRITICAL -> FLUSH (data integrity risk)
                        INSTR_TYPE_ALU: begin
                            if (internal_hazard_flag) next_state =
STATE_STALL; // ALU on CRITICAL, only STALL if AHO reports hazard
                            else next_state = current_state; // Otherwise,
lenient if no other specific trigger
                        end
```

```
                        default: next_state = STATE_FLUSH; // Default to
flush for unknown/reserved types on critical entropy
                    endcase
                end
                ENTROPY_MID: begin
                    // If entropy is MID, we're watchful. React less
aggressively, but still cautious.
                    case (instr_type)
                        INSTR_TYPE_BRANCH, INSTR_TYPE_JUMP: begin
                            if (current_state == STATE_OK) next_state =
STATE_STALL; // Only STALL if currently OK
                            else next_state = current_state; // Otherwise,
maintain current non-OK state
                        end
                        INSTR_TYPE_LOAD, INSTR_TYPE_STORE: begin
                            if (current_state == STATE_OK || current_state
== STATE_STALL) next_state = STATE_STALL; // Prefer STALL on MID for mem
ops
                            else next_state = current_state;
                        end
                        INSTR_TYPE_ALU: begin
                            if (internal_hazard_flag) next_state =
STATE_STALL; // Only STALL if AHO reports hazard
                            else next_state = current_state; // Otherwise,
remain lenient
                        end
                        default: begin // For unknown/reserved types on MID
entropy
                            if (internal_hazard_flag && current_state ==
STATE_OK) next_state = STATE_STALL;
                            else next_state = current_state;
                        end
                    endcase
                end
                default: begin // ENTROPY_LOW (2'b00) or any unused 2'b11
encoding for classified_entropy_level
                    // If entropy is LOW (or unclassified/reserved),
proceed to evaluate ML predictions and internal hazards
                    // This is the default path when no critical entropy or
specific instruction type overrides are active.
                    case (current_state)
                        STATE_OK: begin
                            // From OK state, ML predictions or internal
hazards can trigger transitions.
                            case (ml_predicted_action)
                                STATE_STALL: next_state = STATE_STALL; //
ML predicts STALL
                                STATE_FLUSH: next_state = STATE_FLUSH; //
ML predicts FLUSH
                                STATE_LOCK:  next_state = STATE_LOCK;  //
ML predicts OVERRIDE -> LOCK
                                default: begin // This 'default' handles
2'b00 (OK) or any other unexpected ML input
                                    // Bonus Detail: Simulate a false
negative with entropy override
                                    if (ml_predicted_action == STATE_OK &&
internal_entropy_score > ENTROPY_HIGH_THRESHOLD) begin
                                        next_state = STATE_STALL; // High
internal entropy overrides ML_OK, triggers STALL
                                    end else if (internal_hazard_flag)
begin
```

```
                                        next_state = STATE_STALL; //
Traditional/combined hazard -> STALL
                                    end else begin
                                        next_state = STATE_OK; // No ML
action, no internal hazard, low entropy -> Stay OK
                                    end
                                end
                            endcase
                        end

                        STATE_STALL: begin
                            // From STALL state, ML can escalate to
FLUSH/LOCK, or de-escalate to OK.
                            case (ml_predicted_action)
                                STATE_FLUSH: next_state = STATE_FLUSH; //
ML predicts FLUSH (escalate)
                                STATE_LOCK:  next_state = STATE_LOCK;  //
ML predicts OVERRIDE -> LOCK
                                default: begin // Handles ML OK (00) or ML
STALL (01) or other unexpected
                                    if (ml_predicted_action == STATE_OK &&
!internal_hazard_flag && internal_entropy_score <= ENTROPY_HIGH_THRESHOLD)
begin
                                        next_state = STATE_OK; // ML
predicts OK, no internal hazard, low entropy -> Return to OK
                                    end else begin
                                        next_state = STATE_STALL; //
Otherwise, remain stalled (ML still recommends STALL or hazard persists)
                                    end
                                end
                            endcase
                        end

                        STATE_FLUSH: begin
                            // From FLUSH state, ML can escalate to LOCK,
or de-escalate to STALL/OK.
                            case (ml_predicted_action)
                                STATE_LOCK: next_state = STATE_LOCK; // ML
predicts OVERRIDE -> LOCK
                                default: begin // Handles ML OK (00), ML
STALL (01), ML FLUSH (10), or other unexpected
                                    if (ml_predicted_action == STATE_OK &&
!internal_hazard_flag && internal_entropy_score <= ENTROPY_HIGH_THRESHOLD)
begin
                                        next_state = STATE_OK; // ML
predicts OK, no internal hazard, low entropy -> Return to OK
                                    end else if (ml_predicted_action ==
STATE_STALL) begin
                                        next_state = STATE_STALL; // ML
predicts STALL -> Transition to STALL after flush
                                    end else begin
                                        next_state = STATE_FLUSH; //
Otherwise, remain flushing (e.g., ML insists FLUSH, or unexpected input)
                                    end
                                end
                            endcase
                        end
                        STATE_LOCK: begin
                            // Once in LOCK, the FSM is designed to remain
in LOCK.
```

```
                            // Exiting LOCK state requires an explicit
external hardware reset (rst_n).
                            next_state = STATE_LOCK;
                        end

                        default: next_state = STATE_OK; // Fallback for
undefined 'current_state' (should not happen in synthesizable code)
                    endcase
                end // END of default for classified_entropy_level
            endcase
        end // END of 'else' for quantum/analog override
    end

    // --- Output Logic: Combinational ---
    // The 'fsm_state' directly reflects the 'current_state' of the FSM.
    // This provides the primary control signal to the pipeline.
    always @(*) begin
        fsm_state = current_state;
    end

endmodule

//
================================================================================
====
// ARCHON HAZARD OVERRIDE UNIT (AHO) - Integrated and Enhanced
// Purpose: This module implements the Archon Hazard Override (AHO) unit,
//          responsible for detecting hazardous internal states and
generating
//          override signals (flush, stall) for the CPU pipeline.
//
// Key Enhancements:
// 1. Direct incorporation of 'cache_miss_rate_tracker' as a primary input.
// 2. Implementation of 'fluctuating impact' for various metrics through
//    dynamic weighting, controlled by an external 'ml_predicted_action'.
// 3. A sophisticated rule-based decision engine for hazard mitigation,
//    combining dynamically weighted scores with fixed-priority anomaly
detection.
// This version is designed to provide 'override_flush_sig' and
'override_stall_sig'
// to the Probabilistic Hazard FSM, rather than direct pipeline control.
//
================================================================================
====

module archon_hazard_override_unit (
    input logic                 clk,
    input logic                 rst_n, // Active low reset

    // Core Hazard Metrics (now adapted to 8-bit where needed, Chaos is 16-
bit)
    input logic [7:0]           internal_entropy_score_val, // From QED
(Quantum Entropy Detector) - 8-bit
    input logic [15:0]          chaos_score_val,            // From CD
(Chaos Detector) - 16-bit
    input logic                 anomaly_detected_val,       // From Pattern
Detector (high, fixed impact)

    // Performance/System Health Metrics (now adapted to 8-bit where
needed)
```

```
    input logic [7:0]           branch_miss_rate_tracker,   // Current
branch miss rate (from BTB or PMU) - 8-bit
    input logic [7:0]           cache_miss_rate_tracker,    // NEW: Current
cache miss rate (from Data Memory/Cache) - 8-bit
    input logic [7:0]           exec_pressure_tracker,      // Current
execution pressure (e.g., pipeline fullness) - 8-bit

    // Input from external ML model for dynamic weighting/context
    // This input dictates the current 'risk posture' or 'mode' for hazard
detection.
    // Examples: 2'b00=Normal, 2'b01=MonitorRisk, 2'b10=HighRisk,
2'b11=CriticalRisk
    input logic [1:0]           ml_predicted_action,

    // Dynamically scaled thresholds for the combined hazard score
(adjusted for new total score range)
    // These thresholds would typically be provided by an external control
unit or derived
    // from system-wide context/ML predictions, scaled appropriately for
'total_combined_hazard_score'.
    input logic [20:0]          scaled_flush_threshold,     // If combined
score > this, consider flush
    input logic [20:0]          scaled_stall_threshold,     // If combined
score > this, consider stall

    // Outputs to CPU pipeline control (specifically for Probabilistic
Hazard FSM or main control)
    output logic                override_flush_sig,         // Request for
CPU pipeline flush
    output logic                override_stall_sig,         // Request for
CPU pipeline stall
    output logic [1:0]          hazard_detected_level       // Severity:
00=None, 01=Low, 10=Medium, 11=High/Critical
);

    // --- Internal Signals for Dynamic Weight Assignment (Fluctuating
Impact) ---
    // These 4-bit weights (0-15) are dynamically adjusted based on
'ml_predicted_action'.
    // They amplify or de-emphasize the impact of each raw metric on the
total hazard score.
    logic [3:0] W_entropy;
    logic [3:0] W_chaos;
    logic [3:0] W_branch;
    logic [3:0] W_cache;
    logic [3:0] W_exec;

    // --- Internal Signals for Weighted Scores ---
    // Individual weighted scores are calculated by multiplying raw scores
by weights.
    // Max product for 8-bit * 4-bit: 255 * 15 = 3825. A 12-bit register is
sufficient.
    // Max product for 16-bit * 4-bit: 65535 * 15 = 983025. A 20-bit
register is sufficient.
    logic [11:0] weighted_entropy_score;   // 8-bit val * 4-bit weight ->
12-bit
    logic [19:0] weighted_chaos_score;     // 16-bit val * 4-bit weight ->
20-bit
    logic [11:0] weighted_branch_miss_score; // 8-bit val * 4-bit weight ->
12-bit
```

```
    logic [11:0] weighted_cache_miss_score;  // 8-bit val * 4-bit weight ->
12-bit
    logic [11:0] weighted_exec_pressure_score; // 8-bit val * 4-bit weight
-> 12-bit

    // --- Total Combined Hazard Score ---
    // Sum of all weighted scores.
    // Max sum: (3 * 3825) + (2 * 983025) = 11475 + 1966050 = 1977525.
    // A 21-bit register is sufficient (max value 2097151).
    logic [20:0] total_combined_hazard_score; // Adjusted to 21-bit

    // --- Output Registers (for synchronous outputs) ---
    reg reg_override_flush_sig;
    reg reg_override_stall_sig;
    reg [1:0] reg_hazard_detected_level;

    // --- Clocked Logic for Output Registers ---
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            reg_override_flush_sig      <= 1'b0;
            reg_override_stall_sig      <= 1'b0;
            reg_hazard_detected_level <= 2'b00; // No hazard detected by
default
        end else begin
            // Update output registers with combinational logic's current
state
            reg_override_flush_sig      <= override_flush_sig;
            reg_override_stall_sig      <= override_stall_sig;
            reg_hazard_detected_level <= hazard_detected_level;
        end
    end

    // --- Combinational Logic for Dynamic Weight Assignment (Fluctuating
Impact) ---
    // This block determines the importance (weights) of each metric based
on the
    // 'ml_predicted_action', allowing the system to adapt its sensitivity.
    always @(*) begin
        case (ml_predicted_action)
            2'b00: begin // Normal Operation: Balanced weights, general
monitoring
                W_entropy = 4'd8;   // Moderate impact for entropy/chaos
                W_chaos   = 4'd7;
                W_branch  = 4'd5;   // Moderate for branch/cache misses
(performance indicators)
                W_cache   = 4'd6;
                W_exec    = 4'd4;   // Lower for execution pressure
            end
            2'b01: begin // Monitor Risk: Increased focus on anomaly/chaos
indicators
                W_entropy = 4'd10; // Higher impact for entropy/chaos
                W_chaos   = 4'd9;
                W_branch  = 4'd7;   // Slightly increased for branch/cache
misses
                W_cache   = 4'd8;
                W_exec    = 4'd3;   // Reduced emphasis on exec pressure
            end
            2'b10: begin // High Risk: Strong emphasis on potential
security/stability issues
                W_entropy = 4'd12; // Significantly higher impact for
entropy/chaos
```

```
                W_chaos   = 4'd11;
                W_branch  = 4'd9;   // Substantially increased for
branch/cache misses (could indicate attack)
                W_cache   = 4'd10;
                W_exec    = 4'd2;   // Minimal emphasis on general
performance for immediate risk
            end
            2'b11: begin // Critical Risk: Maximum sensitivity for all
hazard indicators
                W_entropy = 4'd15; // Max impact
                W_chaos   = 4'd15; // Max impact
                W_branch  = 4'd13; // Very high impact
                W_cache    = 4'd14; // Very high impact
                W_exec    = 4'd1;   // Almost no impact for exec pressure,
focus is on stopping threat
            end
            default: begin // Defensive default: Fallback to normal
operation weights
                W_entropy = 4'd8; W_chaos = 4'd7; W_branch = 4'd5; W_cache
= 4'd6; W_exec = 4'd4;
            end
        endcase
    end

    // --- Combinational Logic for Weighted Score Calculation (Dynamic
Weighted Sum) ---
    // Each raw score is multiplied by its dynamically determined weight.
    assign weighted_entropy_score   = internal_entropy_score_val *
W_entropy;
    assign weighted_chaos_score     = chaos_score_val * W_chaos;
    assign weighted_branch_miss_score = branch_miss_rate_tracker *
W_branch;
    assign weighted_cache_miss_score    = cache_miss_rate_tracker *
W_cache; // NEW: Cache miss included
    assign weighted_exec_pressure_score = exec_pressure_tracker * W_exec;

    // The total combined hazard score aggregates all weighted metric
impacts.
    assign total_combined_hazard_score =
        weighted_entropy_score +
        weighted_chaos_score +
        weighted_branch_miss_score +
        weighted_cache_miss_score +
        weighted_exec_pressure_score;

    // --- Combinational Logic for Override Signals (Multi-dimensional Rule
Engine) ---
    // This block implements the decision logic, prioritizing different
hazard indicators.
    always @(*) begin
        override_flush_sig = 1'b0;
        override_stall_sig = 1'b0;
        hazard_detected_level = 2'b00; // Default to no hazard

        // Rule 1: High-priority anomaly detection (Pattern Detector)
        // If an anomaly is detected, this should trigger a flush
immediately,
        // regardless of the combined hazard score, as it signifies a
critical state.
        if (anomaly_detected_val) begin
            override_flush_sig = 1'b1;
```

```
            hazard_detected_level = 2'b11; // Critical
        end else begin
            // Rule 2: Evaluate based on combined hazard score against
dynamic thresholds
            if (total_combined_hazard_score > scaled_flush_threshold) begin
                override_flush_sig = 1'b1;
                hazard_detected_level = 2'b10; // Medium to High (depending
on threshold severity)
            end else if (total_combined_hazard_score >
scaled_stall_threshold) begin
                override_stall_sig = 1'b1;
                hazard_detected_level = 2'b01; // Low to Medium
            end else begin
                // No significant hazard detected by AHO's scoring system
                override_flush_sig = 1'b0;
                override_stall_sig = 1'b0;
                hazard_detected_level = 2'b00; // None
            end
        end
    end

    // Outputs are registered, so assign the internal registered signals
    // These outputs directly drive the next stage (the new FSM)
    // No need for separate output assigns here since they are declared as
logic within the module
    // and directly assigned in the always_comb block and then registered.
    // Remove the previous 'assign override_flush_sig_out =
reg_override_flush_sig;' style lines.
endmodule

// ======================================================================
// NEW: Entropy Control Logic Module
// Features:
// - Directly uses the 16-bit external entropy input from
'entropy_bus.txt'.
// - Applies simple, configurable thresholds to generate stall/flush
signals.
// - This module provides the *base* entropy-driven control signals.
//   These can then be modulated by ML and chaos predictors in the main
CPU.
//
======================================================================
====
module entropy_control_logic(
    input wire [15:0] external_entropy_in, // 16-bit external entropy from
entropy_bus.txt
    output wire entropy_stall,            // Assert to signal a basic
entropy-induced stall
    output wire entropy_flush             // Assert to signal a basic
entropy-induced flush
);

    // Define entropy thresholds for stall and flush
    // These values are for a 16-bit (0-65535) entropy input.
    parameter ENTROPY_STALL_THRESHOLD = 16'd10000;  // Example: Below
10000, consider stalling
    parameter ENTROPY_FLUSH_THRESHOLD = 16'd50000; // Example: Above 50000,
consider flushing

    assign entropy_stall = (external_entropy_in < ENTROPY_STALL_THRESHOLD);
    assign entropy_flush = (external_entropy_in > ENTROPY_FLUSH_THRESHOLD);
```

```
endmodule


//
================================================================================
====
// Pipeline CPU Core (INTEGRATED VERSION)
// Combines all previously defined modules and orchestrates their
interactions.
//
================================================================================
====
module pipeline_cpu(
    input wire clk,
    input wire reset, // Active high reset (converts to active low for some
modules)
    input wire [15:0] external_entropy_in, // Input from entropy_bus.txt
(for Entropy Control Logic)
    input wire [1:0] ml_predicted_action, // ML model's predicted action
for AHO and FSM

    // START OF ADDED PARTS: Analog Override Inputs for pipeline_cpu and
Quantum Override
    input wire analog_lock_override_in,  // From top-level analog
controller
    input wire analog_flush_override_in, // From top-level analog
controller
    input wire quantum_override_signal_in, // NEW: Quantum override signal
from Qiskit simulation
    // END OF ADDED PARTS

    output wire [3:0] debug_pc,         // For debugging: current PC
    output wire [15:0] debug_instr,     // For debugging: current
instruction
    output wire debug_stall,            // For debugging: indicates
pipeline stall
    output wire debug_flush,            // For debugging: indicates
pipeline flush
    output wire debug_lock,             // For debugging: indicates system
lock
    output wire [7:0] debug_fsm_entropy_log, // For debugging: entropy
value logged by new FSM
    output wire [2:0] debug_fsm_instr_type_log // NEW: Debug output for
logged instruction type
);

    // --- Active Low Reset for Modules that use it ---
    wire rst_n = ~reset;

    // --- Internal Wires & Registers for Pipeline Stages ---
    // IF Stage
    reg [3:0] pc_reg;
    wire [15:0] if_instr; // Instruction fetched
    wire [3:0] if_pc_plus_1; // Changed to +1 as PC is 4-bit, not byte-
addressed
    wire [3:0] next_pc; // The next PC to load into pc_reg

    // IF/ID Pipeline Register
    reg [3:0] if_id_pc_plus_1_reg; // For branch target calc and next PC
    reg [15:0] if_id_instr_reg;    // Instruction for ID stage
```

```
    // ID Stage
    wire [3:0] id_pc_plus_1;
    wire [15:0] id_instr;
    wire [3:0] id_operand1;
    wire [3:0] id_operand2;
    wire [2:0] id_rs1_addr;
    wire [2:0] id_rs2_addr;
    wire [2:0] id_rd_addr;
    wire [2:0] id_alu_op;        // Decoded ALU operation
    wire [3:0] id_immediate;     // Sign-extended immediate value
(simplified 3-bit imm to 4-bit)
    wire id_reg_write_enable;    // Write enable for RegFile
    wire id_mem_read_enable;     // Read enable for Data Memory
    wire id_mem_write_enable;    // Write enable for Data Memory
    wire id_is_branch_inst;      // Decoded as a branch instruction
    wire id_is_jump_inst;        // Decoded as a jump instruction
    wire [3:0] id_branch_target; // Branch target from instruction
(simplified 3-bit to 4-bit)

    // START OF ADDED PARTS: Wire for mapped instruction type
    wire [2:0] instr_type_to_fsm_wire;
    // END OF ADDED PARTS

    // ID/EX Pipeline Register
    reg [3:0] id_ex_pc_plus_1_reg;
    reg [3:0] id_ex_operand1_reg;
    reg [3:0] id_ex_operand2_reg;
    reg [2:0] id_ex_rd_addr_reg;
    reg [2:0] id_ex_alu_op_reg;
    reg id_ex_reg_write_enable_reg;
    reg id_ex_mem_read_enable_reg;
    reg id_ex_mem_write_enable_reg;
    reg id_ex_is_branch_inst_reg;
    reg id_ex_is_jump_inst_reg;
    reg [3:0] id_ex_branch_target_reg;
    reg [15:0] id_ex_instr_reg; // For Quantum Entropy Detector

    // EX Stage
    wire [3:0] ex_alu_operand1; // Could be forwarded value
    wire [3:0] ex_alu_operand2; // Could be forwarded value (for ALU
computation or mem_write_data)
    wire [3:0] ex_alu_result;
    wire ex_zero_flag;
    wire ex_negative_flag;
    wire ex_carry_flag;
    wire ex_overflow_flag;
    wire [2:0] ex_rd_addr;
    wire ex_reg_write_enable;
    wire ex_mem_read_enable;
    wire ex_mem_write_enable;
    wire ex_is_branch_inst;
    wire ex_is_jump_inst;
    wire [3:0] ex_branch_target; // Target for actual branch
    wire [3:0] ex_branch_pc;     // PC of the branch instruction itself for
misprediction check

    // EX/MEM Pipeline Register
    reg [3:0] ex_mem_alu_result_reg;
    reg [3:0] ex_mem_mem_write_data_reg; // Value to write to Data Memory
    reg [2:0] ex_mem_rd_addr_reg;
```

```
    reg ex_mem_reg_write_enable_reg;
    reg ex_mem_mem_read_enable_reg;
    reg ex_mem_mem_write_enable_reg;
    reg ex_mem_zero_flag_reg;      // ALU zero flag for branch resolution
    reg ex_mem_is_branch_inst_reg; // Branch instruction flag
    reg ex_mem_is_jump_inst_reg;   // Jump instruction flag
    reg [3:0] ex_mem_pc_plus_1_reg; // PC + 1 from IF stage
    reg [3:0] ex_mem_branch_target_reg; // Branch target from instruction
    reg [3:0] ex_mem_branch_pc_reg; // PC of the branch instruction in EX
stage

    // MEM Stage
    wire [3:0] mem_read_data;     // Data read from Data Memory
    wire [3:0] mem_alu_result;    // ALU result passed from EX/MEM
    wire [2:0] mem_rd_addr;       // Destination register address
    wire mem_reg_write_enable;    // RegFile write enable
    wire mem_mem_read_enable;     // Data Memory read enable
    wire mem_mem_write_enable;    // Data Memory write enable
    wire [3:0] mem_mem_addr;      // Address for Data Memory (ALU result)

    wire branch_actual_taken;     // Actual outcome of branch
    wire branch_mispredicted;     // Signal to BTB and Chaos Detector
    wire [3:0] branch_resolved_pc; // PC of resolved branch (from EX/MEM
pc_reg)
    wire [3:0] branch_resolved_target_pc; // Actual target of resolved
branch

    // MEM/WB Pipeline Register
    reg [3:0] mem_wb_write_data_reg; // Data to write to RegFile (ALU
result or MemRead data)
    reg [2:0] mem_wb_rd_addr_reg;    // Destination register address
    reg mem_wb_reg_write_enable_reg; // RegFile write enable

    // WB Stage
    wire [3:0] wb_write_data;     // Final data for RegFile write
    wire [2:0] wb_rd_addr;        // Final destination register
    wire wb_reg_write_enable;     // Final RegFile write enable

    // --- Pipeline Control Signals ---
    wire pipeline_stall; // Overall stall signal
    wire pipeline_flush; // Overall flush signal

    // For simplicity, tracking rough execution pressure: number of active
instructions
    reg [7:0] exec_pressure_counter; // Example: count of non-NOPs in
flight (simplified)
    reg [7:0] cache_miss_rate_dummy; // Placeholder for actual cache miss
rate. Assume 0-255 scaling.

    // AHO internal hazard signals (outputs from AHO)
    wire aho_override_flush_req;
    wire aho_override_stall_req;
    wire [1:0] aho_hazard_level;

    // Consolidated internal hazard flag for the new FSM
    wire new_fsm_internal_hazard_flag;
    wire [1:0] new_fsm_control_signal; // Output from the new entropy-aware
FSM
    wire [7:0] new_fsm_entropy_log;    // Entropy log from the new FSM
    wire [2:0] new_fsm_instr_type_log; // NEW: Instruction type log from
the new FSM
```

```
    // Dummy values for AHO thresholds (these would come from ML inference)
    // Updated to match the 21-bit total_combined_hazard_score
    localparam AHO_SCALED_FLUSH_THRESH = 21'd1000000; // Example: approx
halfway of max score
    localparam AHO_SCALED_STALL_THRESH = 21'd500000;  // Example: approx
quarter of max score

    // START OF ADDED PARTS: Wire for classified entropy level
    wire [1:0] classified_entropy_level_wire;
    // END OF ADDED PARTS

    // --- Instantiate Sub-modules ---

    // Instruction Memory
    instruction_ram i_imem (
        .clk(clk),
        .reset(reset),
        .pc_in(pc_reg),
        .instr_opcode(if_instr)
    );

    // Register File
    register_file i_regfile (
        .clk(clk),
        .reset(reset),
        .regfile_write_enable(wb_reg_write_enable),
        .write_addr(wb_rd_addr),
        .write_data(wb_write_data),
        .read_addr1(id_rs1_addr),
        .read_addr2(id_rs2_addr),
        .read_data1(id_operand1),
        .read_data2(id_operand2)
    );

    // ALU Unit
    alu_unit i_alu (
        .alu_operand1(ex_alu_operand1),
        .alu_operand2(ex_alu_operand2),
        .alu_op(id_ex_alu_op_reg),
        .alu_result(ex_alu_result),
        .zero_flag(ex_zero_flag),
        .negative_flag(ex_negative_flag),
        .carry_flag(ex_carry_flag),
        .overflow_flag(ex_overflow_flag)
    );

    // Data Memory
    data_mem i_dmem (
        .clk(clk),
        .mem_write_enable(mem_mem_write_enable),
        .mem_read_enable(mem_mem_read_enable),
        .addr(mem_mem_addr),
        .write_data(ex_mem_mem_write_data_reg),
        .read_data(mem_read_data)
    );

    // Branch Target Buffer
    wire [3:0] if_btb_predicted_next_pc; // From BTB prediction
    wire if_btb_predicted_taken;   // From BTB prediction
    branch_target_buffer i_btb (
```

```
        .clk(clk),
        .reset(reset),
        .pc_in(pc_reg), // Current PC to get prediction for
        .branch_resolved_pc(branch_resolved_pc),
        .branch_resolved_pc_valid(ex_mem_is_branch_inst_reg ||
ex_mem_is_jump_inst_reg), // Valid if it was a branch or jump
        .branch_resolved_target_pc(branch_resolved_target_pc),
        .branch_resolved_taken(branch_actual_taken),
        .predicted_next_pc(if_btb_predicted_next_pc), // Output from BTB
for IF
        .predicted_taken(if_btb_predicted_taken)    // Output from BTB for
IF
    );

    // Quantum Entropy Detector
    wire [3:0] qed_instr_opcode_input; // Extracted opcode for QED
    assign qed_instr_opcode_input = id_ex_instr_reg[15:12]; // Assuming
opcode is 4 MSBs of ID/EX instruction
    wire qed_reset = reset; // QED uses active high reset
    wire [7:0] qed_entropy_score_out; // 8-bit output for AHO and new FSM
    quantum_entropy_detector i_qed (
        .clk(clk),
        .reset(qed_reset),
        .instr_opcode(qed_instr_opcode_input),
        .alu_result(ex_alu_result),
        .zero_flag(ex_zero_flag),
        .entropy_score_out(qed_entropy_score_out)
    );

    // Chaos Detector
    wire cd_reset = reset; // CD uses active high reset
    wire [15:0] cd_chaos_score_out; // 16-bit output for AHO
    chaos_detector i_chaos_detector (
        .clk(clk),
        .reset(cd_reset),
        .branch_mispredicted(branch_mispredicted),
        .mem_access_addr(mem_mem_addr), // Address used in MEM stage
        .data_mem_read_data(mem_read_data), // Data read in MEM stage
        .chaos_score_out(cd_chaos_score_out)
    );

    // Pattern Detector
    wire pd_reset = reset; // PD uses active high reset
    wire pd_anomaly_detected_out; // 1-bit output for AHO
    pattern_detector i_pattern_detector (
        .clk(clk),
        .reset(pd_reset),
        .zero_flag_current(ex_zero_flag),
        .negative_flag_current(ex_negative_flag),
        .carry_flag_current(ex_carry_flag),
        .overflow_flag_current(ex_overflow_flag),
        .anomaly_detected_out(pd_anomaly_detected_out)
    );

    // Archon Hazard Override Unit (AHO)
    archon_hazard_override_unit i_aho (
        .clk                        (clk),
        .rst_n                      (rst_n), // Active low reset

        .internal_entropy_score_val (qed_entropy_score_out), // Now 8-bit
        .chaos_score_val            (cd_chaos_score_out),
```

```
        .anomaly_detected_val       (pd_anomaly_detected_out),

        .branch_miss_rate_tracker   (debug_branch_miss_rate), // Use the
debug output for now (8-bit)
        .cache_miss_rate_tracker    (cache_miss_rate_dummy),  //
Placeholder (needs actual cache logic) (8-bit)
        .exec_pressure_tracker      (exec_pressure_counter),  // Use the
simplified counter (8-bit)

        .ml_predicted_action        (ml_predicted_action),    // From top-
level input
        .scaled_flush_threshold     (AHO_SCALED_FLUSH_THRESH), // Fixed for
this example, or from ML
        .scaled_stall_threshold     (AHO_SCALED_STALL_THRESH), // Fixed for
this example, or from ML

        .override_flush_sig         (aho_override_flush_req), // Output to
new FSM
        .override_stall_sig         (aho_override_stall_req), // Output to
new FSM
        .hazard_detected_level      (aho_hazard_level)        // For debug
or other system management
    );

    // Instantiate entropy_trigger_decoder
    entropy_trigger_decoder i_entropy_decoder (
        .entropy_in(qed_entropy_score_out),        // Connect QED output to
decoder input
        .signal_class(classified_entropy_level_wire) // Output to new wire
    );

    // START OF ADDED PARTS: Map id_opcode (4-bit) to instr_type (3-bit)
for FSM
    always @(*) begin
        case (id_opcode)
            4'h1, 4'h2, 4'h3, 4'h6: instr_type_to_fsm_wire = 3'b000; //
ADD, ADDI, SUB, XOR -> ALU
            4'h4:                   instr_type_to_fsm_wire = 3'b001; // LD
-> LOAD
            4'h5:                   instr_type_to_fsm_wire = 3'b010; // ST
-> STORE
            4'h7:                   instr_type_to_fsm_wire = 3'b011; // BEQ
-> BRANCH
            4'h8:                   instr_type_to_fsm_wire = 3'b100; //
JUMP -> JUMP
            default:                instr_type_to_fsm_wire = 3'b111; //
Reserved/Other
        endcase
    end
    // END OF ADDED PARTS

    // NEW: Entropy-Aware FSM
    // Consolidate AHO's requests into a single internal hazard flag for
the new FSM
    assign new_fsm_internal_hazard_flag = aho_override_flush_req ||
aho_override_stall_req;
    fsm_entropy_overlay i_entropy_fsm (
        .clk(clk),
        .rst_n(rst_n), // Active low reset
        .ml_predicted_action(ml_predicted_action),    // ML model's
prediction
```

```
        .internal_entropy_score(qed_entropy_score_out), // Entropy score
from QED
        .internal_hazard_flag(new_fsm_internal_hazard_flag),
        // START OF ADDED PARTS: Passing analog, quantum, and instruction
type inputs to FSM
        .analog_lock_override(analog_lock_override_in),
        .analog_flush_override(analog_flush_override_in),
        .classified_entropy_level(classified_entropy_level_wire), // Pass
classified entropy level to FSM
        .quantum_override_signal(quantum_override_signal_in), // Pass
quantum override signal to FSM
        .instr_type(instr_type_to_fsm_wire), // NEW: Pass mapped
instruction type to FSM
        // END OF ADDED PARTS
        .fsm_state(new_fsm_control_signal),           // Main pipeline
control output
        .entropy_log_out(new_fsm_entropy_log),         // Debug output for
entropy logging
        .instr_type_log_out(new_fsm_instr_type_log)    // NEW: Debug output
for instruction type logging
    );

    // Entropy Control Logic (for external entropy input) - remains as a
separate "base" influence
    wire entropy_ctrl_stall;
    wire entropy_ctrl_flush;
    entropy_control_logic i_entropy_ctrl (
        .external_entropy_in(external_entropy_in),
        .entropy_stall(entropy_ctrl_stall),
        .entropy_flush(entropy_ctrl_flush)
    );

    // --- Pipeline Control Unit ---
    // Combines all stall/flush requests, now primarily driven by the new
entropy-aware FSM.
    // External entropy control acts as an additional independent trigger
for stall/flush.
    // Prioritize LOCK > FLUSH > STALL
    assign pipeline_flush = (new_fsm_control_signal == 2'b10) || // FSM
requests FLUSH
                            (new_fsm_control_signal == 2'b11) || // FSM
requests LOCK (implies FLUSH)
                            entropy_ctrl_flush;                    //
External entropy requests FLUSH

    assign pipeline_stall = (new_fsm_control_signal == 2'b01) || // FSM
requests STALL
                            (new_fsm_control_signal == 2'b11) || // FSM
requests LOCK (implies STALL)
                            entropy_ctrl_stall;                    //
External entropy requests STALL

    // --- Execution Pressure Counter (Simplified) ---
    // Increment if not a NOP, decrement if stall or flush occurs (rough
heuristic)
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            exec_pressure_counter <= 8'h0;
            cache_miss_rate_dummy <= 8'h0; // Initialize dummy cache miss
rate
        end else if (pipeline_flush) begin
```

```
            exec_pressure_counter <= 8'h0; // Clear on flush
            cache_miss_rate_dummy <= 8'h0; // Clear dummy cache miss rate
on flush
        end else if (pipeline_stall) begin
            // Hold or slightly decrement
            exec_pressure_counter <= exec_pressure_counter;
            cache_miss_rate_dummy <= cache_miss_rate_dummy;
        end else begin
            // Assuming opcode 4'h9 is NOP
            if (if_id_instr_reg[15:12] != 4'h9) begin // If instruction is
not NOP
                if (exec_pressure_counter < 8'hFF)
                    exec_pressure_counter <= exec_pressure_counter + 8'h1;
            end else begin
                if (exec_pressure_counter > 8'h0)
                    exec_pressure_counter <= exec_pressure_counter - 8'h1;
            end
            // Simulate a dummy cache miss rate that fluctuates
            if ($urandom_range(0, 100) < 5) begin // 5% chance to increase
                if (cache_miss_rate_dummy < 8'hFF)
                    cache_miss_rate_dummy <= cache_miss_rate_dummy + 8'h1;
            end else if ($urandom_range(0, 100) < 10) begin // 10% chance
to decrease
                if (cache_miss_rate_dummy > 8'h0)
                    cache_miss_rate_dummy <= cache_miss_rate_dummy - 8'h1;
            end
        end
    end

    // --- IF Stage (Instruction Fetch) ---
    // PC calculation and instruction fetch
    assign if_pc_plus_1 = pc_reg + 4'b0001; // Assuming PC increments by 1
per instruction

    // Next PC logic, considering branches, jumps, and pipeline hazards
    always @(*) begin
        next_pc = if_pc_plus_1; // Default: increment PC

        // Branch/Jump override
        if (ex_mem_is_jump_inst_reg) begin // Resolved Jump
            next_pc = ex_mem_branch_target_reg;
        end else if (ex_mem_is_branch_inst_reg) begin // Resolved Branch
            if (branch_actual_taken) begin
                next_pc = ex_mem_branch_target_reg;
            end else begin // If not taken or mispredicted (predicted taken
but actually not taken)
                next_pc = ex_mem_pc_plus_1_reg; // Not taken, use PC+1 from
EX stage
            end
        end else if (if_btb_predicted_taken) begin // BTB Prediction
            next_pc = if_btb_predicted_next_pc;
        end

        // Hazard overrides (new FSM has highest priority for pipeline
control)
        if (new_fsm_control_signal == 2'b11) begin // LOCK state
            next_pc = 4'h0; // Force PC to 0 on lock
        end else if (new_fsm_control_signal == 2'b10) begin // FLUSH state
            next_pc = 4'h0; // Flush: reset PC to 0 or entry point
        end else if (new_fsm_control_signal == 2'b01) begin // STALL state
```

```
            next_pc = pc_reg; // Stall: keep current PC, refetch same
instruction
        end
        // If new_fsm_control_signal is STATE_OK (2'b00), no override, so
normal PC flow
    end


    // PC Register Update
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc_reg <= 4'h0;
        end else begin
            pc_reg <= next_pc;
        end
    end

    // IF/ID Pipeline Register
    always @(posedge clk or posedge reset) begin
        if (reset || pipeline_flush) begin // Flush clears pipeline
registers
            if_id_pc_plus_1_reg <= 4'h0;
            if_id_instr_reg <= 16'h0000; // NOP
        end else if (~pipeline_stall) begin // Stall holds pipeline
registers
            if_id_pc_plus_1_reg <= if_pc_plus_1;
            if_id_instr_reg <= if_instr;
        end
    end

    // --- ID Stage (Instruction Decode / Register Fetch) ---
    assign id_pc_plus_1 = if_id_pc_plus_1_reg;
    assign id_instr = if_id_instr_reg;

    // Instruction Decode (simplified)
    // Assume common instruction format: [opcode (4)|rd (3)|rs1 (3)|rs2
(3)|imm (3)]
    // opcodes: 1=ADD, 2=ADDI, 3=SUB, 4=LD, 5=ST, 6=XOR, 7=BEQ, 8=JUMP,
9=NOP
    wire [3:0] id_opcode = id_instr[15:12];
    assign id_rd_addr = id_instr[11:9];
    assign id_rs1_addr = id_instr[8:6];
    assign id_rs2_addr = id_instr[5:3];
    assign id_immediate = {1'b0, id_instr[2:0]}; // Simplified: 3-bit
immediate, sign-extended to 4 bits

    // Determine control signals based on opcode (simplified)
    assign id_reg_write_enable = (id_opcode == 4'h1 || id_opcode == 4'h2 ||
id_opcode == 4'h3 ||
                                  id_opcode == 4'h4 || id_opcode == 4'h6 ||
id_opcode == 4'h0); // R0 is 0
    assign id_mem_read_enable  = (id_opcode == 4'h4); // LD
    assign id_mem_write_enable = (id_opcode == 4'h5); // ST
    assign id_is_branch_inst   = (id_opcode == 4'h7); // BEQ
    assign id_is_jump_inst     = (id_opcode == 4'h8); // JUMP
    assign id_branch_target    = id_instr[3:0]; // Simplified: 4-bit
relative offset/absolute target

    // ALU opcodes (simplified mapping)
    always @(*) begin
        case (id_opcode)
```

```
            4'h1: id_alu_op = 3'b000; // ADD
            4'h2: id_alu_op = 3'b000; // ADDI (add immediate)
            4'h3: id_alu_op = 3'b001; // SUB
            4'h4: id_alu_op = 3'b000; // LD (for address calculation)
            4'h5: id_alu_op = 3'b000; // ST (for address calculation)
            4'h6: id_alu_op = 3'b100; // XOR
            4'h7: id_alu_op = 3'b001; // BEQ (for comparison: op1 - op2 ==
0)
            default: id_alu_op = 3'bXXX; // Undefined/NOP
        endcase
    end

    // --- Hazard Detection and Forwarding (Simplified Data Hazards) ---
    // Detect RAW hazard between EX/MEM and ID (rs1/rs2)
    wire ex_mem_writes_to_rs1_id = ex_mem_reg_write_enable_reg &&
(ex_mem_rd_addr_reg == id_rs1_addr);
    wire ex_mem_writes_to_rs2_id = ex_mem_reg_write_enable_reg &&
(ex_mem_rd_addr_reg == id_rs2_addr);

    // Detect RAW hazard between MEM/WB and ID (rs1/rs2)
    wire mem_wb_writes_to_rs1_id = mem_wb_reg_write_enable_reg &&
(mem_wb_rd_addr_reg == id_rs1_addr);
    wire mem_wb_writes_to_rs2_id = mem_wb_reg_write_enable_reg &&
(mem_wb_rd_addr_reg == id_rs2_addr);

    // Forwarding logic (simplified: direct connection if hazard)
    wire [3:0] forward_operand1;
    wire [3:0] forward_operand2;

    assign forward_operand1 = (ex_mem_writes_to_rs1_id && (id_rs1_addr !=
3'b000)) ? ex_mem_alu_result_reg :
                              (mem_wb_writes_to_rs1_id && (id_rs1_addr !=
3'b000)) ? mem_wb_write_data_reg :
                              id_operand1; // Default to RegFile read

    assign forward_operand2 = (ex_mem_writes_to_rs2_id && (id_rs2_addr !=
3'b000)) ? ex_mem_alu_result_reg :
                              (mem_wb_writes_to_rs2_id && (id_rs2_addr !=
3'b000)) ? mem_wb_write_data_reg :
                              id_operand2; // Default to RegFile read


    // ID/EX Pipeline Register
    always @(posedge clk or posedge reset) begin
        if (reset || pipeline_flush) begin
            id_ex_pc_plus_1_reg <= 4'h0;
            id_ex_operand1_reg <= 4'h0;
            id_ex_operand2_reg <= 4'h0;
            id_ex_rd_addr_reg <= 3'h0;
            id_ex_alu_op_reg <= 3'h0;
            id_ex_reg_write_enable_reg <= 1'b0;
            id_ex_mem_read_enable_reg <= 1'b0;
            id_ex_mem_write_enable_reg <= 1'b0;
            id_ex_is_branch_inst_reg <= 1'b0;
            id_ex_is_jump_inst_reg <= 1'b0;
            id_ex_branch_target_reg <= 4'h0;
            id_ex_instr_reg <= 16'h0000; // NOP
        end else if (~pipeline_stall) begin
            id_ex_pc_plus_1_reg <= id_pc_plus_1;
            // Select operand 2 based on instruction type (immediate or
register)
```

```
            id_ex_operand1_reg <= forward_operand1;
            id_ex_operand2_reg <= (id_opcode == 4'h2 || id_opcode == 4'h4
|| id_opcode == 4'h5) ? id_immediate : forward_operand2;
            id_ex_rd_addr_reg <= id_rd_addr;
            id_ex_alu_op_reg <= id_alu_op;
            id_ex_reg_write_enable_reg <= id_reg_write_enable;
            id_ex_mem_read_enable_reg <= id_mem_read_enable;
            id_ex_mem_write_enable_reg <= id_mem_write_enable;
            id_ex_is_branch_inst_reg <= id_is_branch_inst;
            id_ex_is_jump_inst_reg <= id_is_jump_inst;
            id_ex_branch_target_reg <= id_branch_target;
            id_ex_instr_reg <= id_instr;
        end
    end

    // --- EX Stage (Execute) ---
    assign ex_alu_operand1 = id_ex_operand1_reg;
    assign ex_alu_operand2 = id_ex_operand2_reg; // This is the ALU's
second operand or store data
    assign ex_rd_addr = id_ex_rd_addr_reg;
    assign ex_reg_write_enable = id_ex_reg_write_enable_reg;
    assign ex_mem_read_enable = id_ex_mem_read_enable_reg;
    assign ex_mem_write_enable = id_ex_mem_write_enable_reg;
    assign ex_is_branch_inst = id_ex_is_branch_inst_reg;
    assign ex_is_jump_inst = id_ex_is_jump_inst_reg;
    assign ex_branch_target = id_ex_branch_target_reg;
    assign ex_branch_pc = id_ex_pc_plus_1_reg - 4'b0001; // PC of the
branch instruction itself

    // Calculate actual branch target: PC of branch instruction + branch
offset
    wire [3:0] actual_branch_target_calc;
    assign actual_branch_target_calc = ex_branch_pc + ex_branch_target;

    // EX/MEM Pipeline Register
    always @(posedge clk or posedge reset) begin
        if (reset || pipeline_flush) begin
            ex_mem_alu_result_reg <= 4'h0;
            ex_mem_mem_write_data_reg <= 4'h0;
            ex_mem_rd_addr_reg <= 3'h0;
            ex_mem_reg_write_enable_reg <= 1'b0;
            ex_mem_mem_read_enable_reg <= 1'b0;
            ex_mem_mem_write_enable_reg <= 1'b0;
            ex_mem_zero_flag_reg <= 1'b0;
            ex_mem_is_branch_inst_reg <= 1'b0;
            ex_mem_is_jump_inst_reg <= 1'b0;
            ex_mem_pc_plus_1_reg <= 4'h0;
            ex_mem_branch_target_reg <= 4'h0;
            ex_mem_branch_pc_reg <= 4'h0;
        end else if (~pipeline_stall) begin
            ex_mem_alu_result_reg <= ex_alu_result;
            ex_mem_mem_write_data_reg <= ex_alu_operand2; // For ST
instructions
            ex_mem_rd_addr_reg <= ex_rd_addr;
            ex_mem_reg_write_enable_reg <= ex_reg_write_enable;
            ex_mem_mem_read_enable_reg <= ex_mem_read_enable;
            ex_mem_mem_write_enable_reg <= ex_mem_write_enable;
            ex_mem_zero_flag_reg <= ex_zero_flag; // Pass zero flag for
branch check
            ex_mem_is_branch_inst_reg <= ex_is_branch_inst;
            ex_mem_is_jump_inst_reg <= ex_is_jump_inst;
```

```
            ex_mem_pc_plus_1_reg <= id_ex_pc_plus_1_reg; // Pass PC+1 from
ID stage
            ex_mem_branch_target_reg <= actual_branch_target_calc; //
Actual calculated target
            ex_mem_branch_pc_reg <= ex_branch_pc; // PC of the branch
instruction
        end
    end

    // --- MEM Stage (Memory Access) ---
    assign mem_alu_result = ex_mem_alu_result_reg;
    assign mem_rd_addr = ex_mem_rd_addr_reg;
    assign mem_reg_write_enable = ex_mem_reg_write_enable_reg;
    assign mem_mem_read_enable = ex_mem_mem_read_enable_reg;
    assign mem_mem_write_enable = ex_mem_mem_write_enable_reg;
    assign mem_mem_addr = ex_mem_alu_result_reg; // ALU result is memory
address

    // Branch Resolution in MEM Stage
    assign branch_actual_taken = ex_mem_is_branch_inst_reg &&
ex_mem_zero_flag_reg; // BEQ taken if Zero flag is set
    assign branch_resolved_pc = ex_mem_branch_pc_reg;
    assign branch_resolved_target_pc = ex_mem_branch_target_reg;

    // Branch Misprediction Detection
    wire branch_mispredicted_local; // Local wire for branch misprediction
    assign branch_mispredicted = branch_mispredicted_local; // Assign to
top-level wire

    always @(*) begin
        branch_mispredicted_local = 1'b0; // Default to no misprediction

        // Only check misprediction if it was a branch/jump instruction
that completed EX stage
        if (ex_mem_is_branch_inst_reg || ex_mem_is_jump_inst_reg) begin
            if (ex_mem_is_branch_inst_reg) begin // Conditional branch
(BEQ)
                // Misprediction if predicted taken != actual taken
                // OR if predicted target != actual target (if taken)
                if (if_btb_predicted_taken != branch_actual_taken) begin
                    branch_mispredicted_local = 1'b1;
                end else if (branch_actual_taken &&
(if_btb_predicted_next_pc != branch_resolved_target_pc)) begin
                    branch_mispredicted_local = 1'b1;
                end
            end else if (ex_mem_is_jump_inst_reg) begin // Unconditional
jump
                // Misprediction if predicted target != actual target
                if (if_btb_predicted_next_pc != branch_resolved_target_pc)
begin
                    branch_mispredicted_local = 1'b1;
                end
            end
        end
    end

    // For debugging branch miss rate
    reg [7:0] branch_miss_rate_counter;
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            branch_miss_rate_counter <= 8'h0;
```

```
        end else if (branch_mispredicted) begin
            if (branch_miss_rate_counter < 8'hFF)
                branch_miss_rate_counter <= branch_miss_rate_counter +
8'h1;
        end else begin
            if (branch_miss_rate_counter > 8'h0)
                branch_miss_rate_counter <= branch_miss_rate_counter -
8'h01; // Decay
        end
    end
    wire [7:0] debug_branch_miss_rate = branch_miss_rate_counter; // Output
for AHO and debug

    // MEM/WB Pipeline Register
    always @(posedge clk or posedge reset) begin
        if (reset || pipeline_flush) begin
            mem_wb_write_data_reg <= 4'h0;
            mem_wb_rd_addr_reg <= 3'h0;
            mem_wb_reg_write_enable_reg <= 1'b0;
        end else if (~pipeline_stall) begin
            // Data to write back: from memory if Load, else from ALU
            mem_wb_write_data_reg <= (mem_mem_read_enable) ? mem_read_data
: mem_alu_result;
            mem_wb_rd_addr_reg <= mem_rd_addr;
            mem_wb_reg_write_enable_reg <= mem_reg_write_enable;
        end
    end

    // --- WB Stage (Write Back) ---
    assign wb_write_data = mem_wb_write_data_reg;
    assign wb_rd_addr = mem_wb_rd_addr_reg;
    assign wb_reg_write_enable = mem_wb_reg_write_enable_reg;

    // --- Debug Outputs ---
    assign debug_pc = pc_reg;
    assign debug_instr = if_instr; // Or if_id_instr_reg, depending on
desired debug point
    assign debug_stall = pipeline_stall;
    assign debug_flush = pipeline_flush;
    assign debug_lock = (new_fsm_control_signal == 2'b11); // Directly from
new FSM lock state
    assign debug_fsm_entropy_log = new_fsm_entropy_log; // New debug output
for entropy logging
    assign debug_fsm_instr_type_log = new_fsm_instr_type_log; // NEW: Debug
output for logged instruction type

endmodule


//
================================================================================
====
// NEW MODULE: entropy_trigger_decoder.v
// Purpose: Simulates compression of incoming analog entropy signals (8-
bit)
//          into meaningful trigger vectors or score levels (2-bit).
//
================================================================================
====
module entropy_trigger_decoder(
    input wire [7:0] entropy_in,    // 8-bit entropy score (0-255)
```

```
    output reg [1:0] signal_class   // 2-bit output: 00 = LOW, 01 = MID, 10
= CRITICAL
);

    // Define thresholds for classification
    parameter THRESHOLD_LOW_TO_MID = 8'd85;     // Up to 85 is LOW
    parameter THRESHOLD_MID_TO_CRITICAL = 8'd170; // Up to 170 is MID,
above is CRITICAL

    always @(*) begin
        if (entropy_in <= THRESHOLD_LOW_TO_MID) begin
            signal_class = 2'b00; // LOW
        end else if (entropy_in <= THRESHOLD_MID_TO_CRITICAL) begin
            signal_class = 2'b01; // MID
        end else begin
            signal_class = 2'b10; // CRITICAL
        end
    end

endmodule
```

12.2. Full Python Code for GUI (`dashboard.py`, `entropy_viewer.py`)

dashboard.py

```
import tkinter as tk
from tkinter import ttk
from PIL import Image, ImageTk, ImageDraw
import random
import time
import threading
import queue
import os

class FSMDashboard(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Hybrid Chaos Entropy Dashboard")
        self.geometry("1000x700") # Increased size to accommodate new panel
        self.configure(bg="#2c3e50") # Dark blue-grey background

        # --- Dashboard Title ---
        self.title_label = tk.Label(self, text="ARCHON Hazard Monitoring",
```

```
                                    font=("Inter", 28, "bold"),
fg="#ecf0f1", bg="#2c3e50", pady=15)
        self.title_label.pack(pady=(20, 10))

        # --- Main Frame for Panels ---
        main_frame = ttk.Frame(self, padding="20 20 20 20",
style="Dark.TFrame")
        main_frame.pack(expand=True, fill="both")

        # Configure style for dark theme
        self.style = ttk.Style()
        self.style.theme_use("clam") # "clam" is a good base for
customization
        self.style.configure("Dark.TFrame", background="#34495e",
borderwidth=5, relief="flat", bordercolor="#2c3e50")
        self.style.configure("Dark.TLabel", background="#34495e",
foreground="#ecf0f1", font=("Inter", 12))
        self.style.configure("Big.Dark.TLabel", background="#34495e",
foreground="#ecf0f1", font=("Inter", 18, "bold"))
        self.style.configure("Red.TLabel", background="#e74c3c",
foreground="white", font=("Inter", 18, "bold")) # FLUSH/LOCK
        self.style.configure("Orange.TLabel", background="#f39c12",
foreground="white", font=("Inter", 18, "bold")) # STALL
        self.style.configure("Green.TLabel", background="#27ae60",
foreground="white", font=("Inter", 18, "bold")) # OK
        self.style.configure("Info.TLabel", background="#34495e",
foreground="#95a5a6", font=("Inter", 10, "italic")) # Smaller info text
        self.style.configure("TProgressbar", thickness=20,
troughcolor="#7f8c8d", background="#2ecc71", borderwidth=0)
        self.style.con ("Prediction.TProgressbar", thickness=20,
troughcolor="#7f8c8d", background="#3498db", borderwidth=0)


        # --- Panel 1: FSM State Indicator ---
        fsm_frame = ttk.LabelFrame(main_frame, text="FSM State",
style="Dark.TFrame", padding="10 10 10 10")
        fsm_frame.grid(row=0, column=0, padx=10, pady=10, sticky="nsew")

        self.fsm_state_label = ttk.Label(fsm_frame, text="N/A",
style="Green.TLabel", anchor="center")
        self.fsm_state_label.pack(expand=True, fill="both", padx=10,
pady=10)

        self.cycle_label = ttk.Label(fsm_frame, text="Cycle: N/A",
style="Info.TLabel")
        self.cycle_label.pack(pady=(5, 0))


        # --- Panel 2: Entropy Score ---
        entropy_frame = ttk.LabelFrame(main_frame, text="Entropy Score",
style="Dark.TFrame", padding="10 10 10 10")
        entropy_frame.grid(row=0, column=1, padx=10, pady=10,
sticky="nsew")

        self.entropy_value_label = ttk.Label(entropy_frame, text="N/A",
style="Big.Dark.TLabel", anchor="center")
        self.entropy_value_label.pack(pady=(10, 5))

        self.entropy_meter = ttk.Progressbar(entropy_frame,
orient="horizontal", length=200, mode="determinate", style="TProgressbar")
        self.entropy_meter.pack(padx=10, pady=5, fill="x")
```

```
        self.entropy_meter["maximum"] = 255 # 8-bit entropy score

        self.entropy_status_label = ttk.Label(entropy_frame, text="N/A",
style="Info.TLabel")
        self.entropy_status_label.pack(pady=(5, 0))


        # --- Panel 3: Override Source Indicator ---
        override_frame = ttk.LabelFrame(main_frame, text="Override Source",
style="Dark.TFrame", padding="10 10 10 10")
        override_frame.grid(row=1, column=0, padx=10, pady=10,
sticky="nsew")

        self.override_source_label = ttk.Label(override_frame, text="N/A",
style="Big.Dark.TLabel", anchor="center")
        self.override_source_label.pack(expand=True, fill="both", padx=10,
pady=10)


        # --- Panel 4: Waveform Snapshot Viewer (Placeholder) ---
        waveform_frame = ttk.LabelFrame(main_frame, text="Waveform
Snapshot", style="Dark.TFrame", padding="10 10 10 10")
        waveform_frame.grid(row=1, column=1, padx=10, pady=10,
sticky="nsew")

        try:
            img = Image.new('RGB', (200, 150), color = 'darkgray')
            d = ImageDraw.Draw(img)
            d.text((50,60), "Waveform Plot Placeholder", fill=(0,0,0))
            self.waveform_img = ImageTk.PhotoImage(img)
            self.waveform_label = tk.Label(waveform_frame,
image=self.waveform_img, bg="#34495e")
            self.waveform_label.pack(expand=True, fill="both", padx=5,
pady=5)
        except Exception as e:
            print(f"Error loading placeholder image: {e}")
            self.waveform_label = tk.Label(waveform_frame, text="Image Load
Error", bg="#34495e", fg="red")
            self.waveform_label.pack(expand=True, fill="both", padx=5,
pady=5)


        # --- Panel 5 (NEW): Entropy Classification Overlay ---
        classification_frame = ttk.LabelFrame(main_frame, text="Entropy
Classification", style="Dark.TFrame", padding="10 10 10 10")
        classification_frame.grid(row=0, column=2, rowspan=2, padx=10,
pady=10, sticky="nsew") # Spanning two rows

        self.prob_stall_label = ttk.Label(classification_frame, text="Prob.
of STALL: N/A%", style="Big.Dark.TLabel", anchor="center")
        self.prob_stall_label.pack(pady=(10, 5))

        self.prob_stall_meter = ttk.Progressbar(classification_frame,
orient="vertical", length=150, mode="determinate",
style="Prediction.TProgressbar")
        self.prob_stall_meter.pack(padx=10, pady=5, fill="y", expand=True)
# Vertical progress bar
        self.prob_stall_meter["maximum"] = 100 # Percentage

        self.classification_info_label = ttk.Label(classification_frame,
text="Based on Entropy Score", style="Info.TLabel")
```

```
        self.classification_info_label.pack(pady=(5, 0))


        # --- Configure Grid Weights for Resizing ---
        main_frame.grid_rowconfigure(0, weight=1)
        main_frame.grid_rowconfigure(1, weight=1)
        main_frame.grid_columnconfigure(0, weight=1)
        main_frame.grid_columnconfigure(1, weight=1)
        main_frame.grid_columnconfigure(2, weight=0.8) # Give
classification panel a bit less weight initially

        # --- Real-time Log Stream Handling ---
        self.log_filename = "fsm_log.txt"
        self.log_queue = queue.Queue()
        self.log_reader_thread = None
        self.stop_event = threading.Event() # Event to signal thread to
stop

        # Start reading the log file in a separate thread
        self.start_log_reader()
        # Periodically check the queue for new log entries and update GUI
        self.after(100, self.process_queue)

    def start_log_reader(self):
        """Starts a new thread to read the log file."""
        if self.log_reader_thread is None or not
self.log_reader_thread.is_alive():
            print(f"Starting log reader thread for {self.log_filename}")
            self.log_reader_thread =
threading.Thread(target=self._read_log_file_continuously,

args=(self.log_filename, self.log_queue, self.stop_event),
                                                    daemon=True) # Daemon
thread exits with main program
            self.log_reader_thread.start()

    def _read_log_file_continuously(self, filename, q, stop_event):
        """Reads log file updates line by line and puts them into a
queue."""
        if not os.path.exists(filename):
            print(f"Log file '{filename}' not found. Creating empty file.")
            with open(filename, 'w') as f:
                pass # Create the file

        try:
            with open(filename, 'r') as f:
                f.seek(0, os.SEEK_END) # Go to the end of the file
                print(f"Reading from '{filename}'. Starting at end of
file.")
                while not stop_event.is_set():
                    line = f.readline()
                    if not line:
                        time.sleep(0.01) # Wait a bit if no new lines
                        continue
                    q.put(line.strip()) # Put the new line into the queue
        except Exception as e:
            print(f"Error reading log file in thread: {e}")

    def process_queue(self):
        """Checks the queue for new log entries and updates the GUI."""
        try:
```

```
            while True:
                log_entry = self.log_queue.get_nowait() # Get without
blocking
                self.parse_and_update_gui(log_entry)
        except queue.Empty:
            pass # No more items in the queue
        finally:
            # Schedule the next check (e.g., every 100ms)
            self.after(100, self.process_queue)

    def parse_and_update_gui(self, log_entry):
        """Parses a log entry and updates the dashboard GUI elements."""
        try:
            # Expected format: [Cycle 123] State: STALL | Entropy: 190 |
Trigger: analog
            parts = log_entry.split(' | ')
            cycle_str = parts[0].split(' ')[1].replace(']', '') # e.g.,
'123'
            state_str = parts[1].split(': ')[1] # e.g., 'STALL'
            entropy_str = parts[2].split(': ')[1] # e.g., '190'
            trigger_str = parts[3].split(': ')[1] # e.g., 'analog'

            cycle = int(cycle_str)
            entropy_score = int(entropy_str)

            # Update FSM State
            self.fsm_state_label.config(text=state_str)
            if state_str == "OK":
                self.fsm_state_label.config(style="Green.TLabel")
            elif state_str == "STALL":
                self.fsm_state_label.config(style="Orange.TLabel")
            else: # FLUSH or LOCK
                self.fsm_state_label.config(style="Red.TLabel")
            self.cycle_label.config(text=f"Cycle: {cycle}")

            # Update Entropy Score
            self.entropy_value_label.config(text=str(entropy_score))
            self.entropy_meter["value"] = entropy_score
            if entropy_score > 180:
                self.entropy_status_label.config(text="High Entropy")
                self.style.configure("TProgressbar", background="#e74c3c")
# Red for high entropy
            elif entropy_score > 120:
                self.entropy_status_label.config(text="Elevated")
                self.style.configure("TProgressbar", background="#f39c12")
# Orange for elevated
            else:
                self.entropy_status_label.config(text="Normal")
                self.style.configure("TProgressbar", background="#2ecc71")
# Green for normal

            # Update Override Source
            self.override_source_label.config(text=trigger_str)

            # NEW: Update Entropy Classification Overlay
            # Simulate probability of STALL based on entropy score (0-255)
            # Higher entropy -> higher probability
            prob_stall = min(100, round((entropy_score / 255) * 100)) #
Scale to 0-100%
            self.prob_stall_label.config(text=f"Prob. of STALL:
{prob_stall}%")
```

```
            self.prob_stall_meter["value"] = prob_stall
            # Change color of prediction bar based on probability
            if prob_stall > 70:
                self.style.configure("Prediction.TProgressbar",
background="#e74c3c") # High risk, red
            elif prob_stall > 40:
                self.style.configure("Prediction.TProgressbar",
background="#f39c12") # Medium risk, orange
            else:
                self.style.configure("Prediction.TProgressbar",
background="#3498db") # Low risk, blue

        except Exception as e:
            print(f"Error parsing log entry '{log_entry}': {e}")

    def on_closing(self):
        """Handles proper shutdown when the window is closed."""
        print("Closing dashboard. Signalling log reader thread to stop.")
        self.stop_event.set() # Set the event to stop the thread
        if self.log_reader_thread and self.log_reader_thread.is_alive():
            self.log_reader_thread.join(timeout=1.0) # Give thread a chance
to finish
        self.destroy()

if __name__ == "__main__":
    app = FSMDashboard()
    app.protocol("WM_DELETE_WINDOW", app.on_closing) # Handle window
closing event
    app.mainloop()
```

Entropy_viewer.py

```
import time
import random

def generate_log_entry(cycle, states, triggers):
    """Generates a single log entry string."""
    state = random.choice(states)
    entropy = random.randint(0, 255) # 8-bit entropy score
    trigger = random.choice(triggers)
    return f"[Cycle {cycle}] State: {state} | Entropy: {entropy} | Trigger:
{trigger}"

def simulate_log_stream(output_filename="fsm_log.txt", num_entries=200,
delay_seconds=0.1):
    """
    Simulates a real-time log stream and writes entries to a file.

    Args:
        output_filename (str): The name of the file to write logs to.
        num_entries (int): The total number of log entries to generate.
        delay_seconds (float): The delay between writing each log entry to
the file.
    """
    fsm_states = ["OK", "STALL", "FLUSH", "LOCK"]
    trigger_sources = ["ML", "Analog", "Entropy Logic", "AHO", "None"] #
"None" for no specific trigger

    print(f"Starting log simulation. Writing to '{output_filename}'...")
```

```
    print(f"Generating {num_entries} entries with a {delay_seconds}-second
delay between each.")

    try:
        with open(output_filename, 'w') as f:
            for cycle in range(1, num_entries + 1):
                log_entry = generate_log_entry(cycle, fsm_states,
trigger_sources)
                f.write(log_entry + '\n')
                f.flush() # Ensure data is written to disk immediately
                print(f"Logged: {log_entry}") # Print to console for real-
time feedback
                time.sleep(delay_seconds)
        print("\nLog simulation finished.")
    except Exception as e:
        print(f"An error occurred during log simulation: {e}")

if __name__ == "__main__":
    # You can change the number of entries or delay here
    simulate_log_stream(num_entries=500, delay_seconds=0.05)
```

12.3. Full Python Code for Quantum Circuit (`quantum_override_circuit.py`)

```
# --- QISKIT & NUMPY IMPORTS ---
from qiskit import QuantumCircuit              # Core class for building
quantum circuits
from qiskit_aer import Aer                     # Qiskit's high-performance
simulator backend
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt                # Used for saving circuit
and histogram plots
import numpy as np                             # Numerical computations
(like pi)
import numpy.random as npr                     # For random noise
simulation

# --- CORE FUNCTION ---
def generate_quantum_override_signal():
    """
    Simulates a noisy entangled 2-qubit system and returns a binary
override signal.
    If decoherence is strong enough (high |11⟩ probability), trigger the
override.
    """

    # STEP 1: Initialize a 2-qubit quantum circuit with 2 classical bits
for measurement
    qc = QuantumCircuit(2, 2)

    # STEP 2: Create entanglement — Bell state |Φ+⟩ = (|00⟩ + |11⟩) / √2
    qc.h(0)         # Put qubit 0 into superposition
    qc.cx(0, 1)     # Entangle qubit 0 with qubit 1 using CNOT
```

```
    # STEP 3: Inject simulated noise using a randomized U3 gate on qubit 0
    theta, phi, lam = npr.uniform(0, 2 * np.pi, 3)
    qc.u(theta, phi, lam, 0)  # Apply general unitary noise to only one
qubit

    # STEP 4: Measure both qubits, mapping to classical bits
    qc.measure([0, 1], [0, 1])

    # STEP 5: Simulate the circuit execution using Aer (QASM simulator =
shot-based)
    backend = Aer.get_backend("qasm_simulator")
    job = backend.run(qc, shots=1024)           # Execute 1024 times to get
statistics
    result = job.result()
    counts = result.get_counts()                # e.g., {'00': 500, '11':
200, ...}

    # STEP 6: Analyze measurement counts to decide if override signal
should trigger
    probability_11 = counts.get('11', 0) / 1024 # Compute P(|11⟩)
    threshold = 0.1                             # 10% threshold for
override activation
    override_signal = 1 if probability_11 > threshold else 0

    return qc, counts, override_signal, probability_11

# --- MAIN SCRIPT ---
if __name__ == "__main__":
    # Run the function and unpack results
    qc, counts, override_signal, prob_11 =
generate_quantum_override_signal()

    # Print human-readable summary of simulation
    print(f"Simulation Counts: {counts}")
    print(f"Probability of |11⟩: {prob_11:.4f}")
    print(f"Quantum Override Signal: {override_signal}")

    # STEP 7: Save a visual of the circuit diagram to file
    try:
        qc.draw("mpl")  # Matplotlib drawer
        plt.title("Quantum Circuit Diagram")
        plt.savefig("quantum_override_sim.png")
        plt.close()
        print("Circuit diagram saved to quantum_override_sim.png")
    except Exception as e:
        print(f"Error saving circuit diagram: {e}")
        print(qc.draw("text"))  # Fallback to ASCII

    # STEP 8: Save histogram of the simulation results
    try:
        fig = plot_histogram(counts, figsize=(8, 6))
        plt.title("Measurement Results")
        plt.savefig("quantum_histogram.png")
        plt.close(fig)
        print("Histogram saved to quantum_histogram.png")
    except Exception as e:
        print(f"Error saving histogram: {e}")
```

12.4. analog_spike_override.asc

```
Version 4
SHEET 1 1804 968
WIRE 816 96 816 80
WIRE 816 96 768 96
WIRE 1008 112 1008 80
WIRE 1072 112 1008 112
WIRE 864 128 816 128
WIRE 1008 128 960 128
WIRE 768 208 768 96
WIRE 816 208 768 208
WIRE 928 208 896 208
WIRE 960 208 960 128
WIRE 960 208 928 208
WIRE 1072 208 1072 112
WIRE 1072 208 1008 208
WIRE 208 224 144 224
WIRE 384 224 208 224
WIRE 384 240 384 224
WIRE 864 240 864 128
WIRE 880 240 864 240
WIRE 944 240 880 240
WIRE 144 256 144 224
WIRE 944 256 944 240
WIRE 896 272 896 208
WIRE 912 272 896 272
WIRE 1216 288 976 288
WIRE 1232 288 1216 288
WIRE 1296 288 1296 176
WIRE 544 304 384 304
WIRE 672 304 544 304
WIRE 912 304 672 304
WIRE 144 368 144 336
WIRE 944 368 944 320
WIRE 992 368 944 368
WIRE 1024 368 992 368
WIRE 944 400 816 400
WIRE 1296 416 1296 368
WIRE 1024 480 1024 368
WIRE 1024 480 944 480
FLAG 208 224 N_entropy_rising
IOPIN 208 224 In
FLAG 544 304 N_differentiator_out
FLAG 1216 288 analog_spike_override
IOPIN 1216 288 Out
FLAG 928 208 N_threshold_ref
FLAG 992 368 VSS
FLAG 880 240 VDD
FLAG 1296 176 N_fsm_spike_in
IOPIN 1296 176 Out
SYMBOL voltage 144 240 R0
WINDOW 3 -182 -584 Left 2
SYMATTR InstName V_entropy_rising
SYMATTR Value PWL file=0 0 0.1m 0 0.15m 5 0.2m 5 0.25m 0 0.3m 0 0.4m 1
0.45m 1.5 0.5m 2 0.55m 2.5 0.6m 3 0.65m 5 0.7m 5 0.75m 0
SYMBOL voltage 816 112 R0
SYMATTR InstName V_VDD
SYMATTR Value 5
SYMBOL voltage 944 384 R0
```

```
SYMATTR InstName V_VSS
SYMATTR Value -5
SYMBOL voltage 1008 112 R0
SYMATTR InstName V_threshold
SYMATTR Value 0.3
SYMBOL res 656 208 R0
SYMATTR InstName R_differentiator
SYMATTR Value 1k
SYMBOL cap 368 240 R0
SYMATTR InstName C_differentiator
SYMATTR Value 1nF
SYMBOL OpAmps/opamp2 944 224 R0
SYMATTR InstName XU1_comparator
SYMATTR Value AD711
SYMBOL g 144 352 R0
SYMATTR InstName G1
SYMBOL g 1008 -16 R0
SYMATTR InstName G2
SYMBOL g 672 128 R0
SYMATTR InstName G3
SYMBOL g 816 -16 R0
SYMATTR InstName G4
SYMBOL g 816 384 R0
SYMATTR InstName G5
SYMBOL bv 1296 272 R0
SYMATTR InstName B1_spike_adc
SYMATTR Value V=F(V(analog_spike_override) > 2.5 ? 1 : 0)
SYMBOL g 1296 400 R0
SYMATTR InstName G6
TEXT 1416 176 Left 2 !.lib OpAmps/AD711.asy\n.tran 0 0.8m
```

12.5. 3_input_analog_entropy_override.asc

```
Version 4
SHEET 1 1920 1056
WIRE 1088 16 992 16
WIRE 1280 16 1088 16
WIRE 1872 16 1280 16
WIRE 656 32 608 32
WIRE 992 48 992 16
WIRE 528 112 480 112
WIRE 816 112 784 112
WIRE 944 112 816 112
WIRE 608 128 608 32
WIRE 656 128 608 128
WIRE 784 128 784 112
WIRE 1472 128 1376 128
WIRE 304 160 32 160
WIRE 32 176 32 160
WIRE 480 176 480 112
WIRE 480 176 448 176
WIRE 1088 176 1088 16
WIRE 1088 176 992 176
WIRE 304 208 304 160
WIRE 656 208 656 128
WIRE 656 208 512 208
WIRE 736 208 720 208
WIRE 992 208 992 176
```

```
WIRE 1328 208 1248 208
WIRE 32 224 32 176
WIRE 400 224 400 144
WIRE 512 224 512 208
WIRE 848 224 784 224
WIRE 1280 224 1280 16
WIRE 1376 224 1280 224
WIRE 480 240 464 240
WIRE 624 256 544 256
WIRE 688 256 624 256
WIRE 848 256 848 224
WIRE 848 256 784 256
WIRE 1392 256 1376 256
WIRE 1472 256 1472 128
WIRE 1472 256 1392 256
WIRE 400 272 400 256
WIRE 480 272 400 272
WIRE 784 272 784 256
WIRE 944 288 944 112
WIRE 992 288 992 272
WIRE 992 288 944 288
WIRE 1248 288 1248 208
WIRE 1248 288 992 288
WIRE 1376 288 1376 256
WIRE 448 304 448 176
WIRE 512 304 512 288
WIRE 512 304 448 304
WIRE 32 320 32 304
WIRE 32 320 -144 320
WIRE 624 336 624 320
WIRE 624 336 496 336
WIRE 656 336 656 208
WIRE 656 336 624 336
WIRE -144 352 -144 320
WIRE 304 352 304 288
WIRE 688 352 688 256
WIRE 736 352 688 352
WIRE 304 368 304 352
WIRE 400 368 400 288
WIRE 400 368 304 368
WIRE 464 368 464 240
WIRE 464 368 432 368
WIRE 992 368 992 288
WIRE 1328 368 992 368
WIRE 720 416 720 208
WIRE 720 416 592 416
WIRE 1376 448 1376 384
WIRE 432 464 432 368
WIRE 480 464 432 464
WIRE 784 464 784 368
WIRE 240 512 32 512
WIRE 32 528 32 512
WIRE 1792 528 1728 528
WIRE 528 544 464 544
WIRE 1376 544 1376 528
WIRE 1584 544 1376 544
WIRE 32 560 32 528
WIRE 1632 592 1408 592
WIRE 1664 592 1632 592
WIRE 1664 608 1664 592
WIRE 1680 608 1664 608
```

```
WIRE 624 624 624 336
WIRE 784 624 624 624
WIRE 1376 624 1376 544
WIRE 1376 624 1248 624
WIRE 1872 624 1872 16
WIRE 1872 624 1728 624
WIRE 1072 656 1024 656
WIRE 1408 656 1408 592
WIRE 1408 656 1168 656
WIRE 1424 656 1408 656
WIRE -144 672 -144 432
WIRE 32 672 32 640
WIRE 32 672 -144 672
WIRE 784 672 784 624
WIRE 1072 672 1072 656
WIRE 1168 672 1168 656
WIRE 1248 672 1248 624
WIRE 1776 672 1728 672
WIRE 1792 672 1792 528
WIRE 1792 672 1776 672
WIRE 464 688 464 544
WIRE 752 688 464 688
WIRE 832 704 816 704
WIRE 1024 704 1024 656
WIRE 1024 704 832 704
WIRE 240 720 240 512
WIRE 752 720 240 720
WIRE 1376 736 1376 624
WIRE 1424 736 1376 736
WIRE 1632 752 1632 592
WIRE 1680 752 1632 752
WIRE 592 800 592 416
WIRE 592 800 32 800
WIRE 1072 800 1072 736
WIRE 1152 800 1072 800
WIRE 1168 800 1168 736
WIRE 1168 800 1152 800
WIRE 1248 800 1248 752
WIRE 1248 800 1168 800
WIRE 1584 800 1584 544
WIRE 1728 800 1728 768
WIRE 1728 800 1584 800
WIRE 32 832 32 800
WIRE 480 864 480 464
WIRE 784 864 784 736
WIRE 784 864 480 864
WIRE 32 928 32 912
WIRE 32 928 -112 928
WIRE -112 960 -112 928
FLAG 32 528 N_noise
IOPIN 32 528 In
FLAG 32 800 N_ml_trigger
IOPIN 32 800 In
FLAG 32 176 N_entropy
IOPIN 32 176 In
FLAG 304 352 N_entropy_filtered
FLAG 624 256 N_entropy_comp_out
FLAG 848 256 N_nand_interm
FLAG 816 112 N_nand_out
FLAG 1392 256 LOCK_OUT
IOPIN 1392 256 Out
```

```
FLAG 832 704 N_noise_comp_out
FLAG 1152 800 N_pulse_raw
FLAG 1168 656 N_pulse_clipped_pos
FLAG 1776 672 FLUSH_OUT
IOPIN 1776 672 Out
SYMBOL voltage 32 208 R0
WINDOW 3 24 38 Left 2
SYMATTR InstName V_entropy
SYMATTR Value PWL file=0 0 1m 5
SYMBOL voltage 32 544 R0
WINDOW 3 24 96 Invisible 2
SYMATTR InstName V_noise
SYMATTR Value PWL file=0 0 0.199m 0 0.2m 2.5 0.20005m 0 0.399m 0 0.4m 2.5
0.40005m 0 0.599m 0 0.6m 2.5 0.60005m 0 0.799m 0 0.8m 2.5 0.80005m 0 1m 0
SYMBOL voltage 32 816 R0
SYMATTR InstName V_ml_trigger
SYMATTR Value PWL file=0 0 0.499m 0 0.5m 1 1m 1
SYMBOL cap 384 224 R0
SYMATTR InstName C1
SYMATTR Value 1l
SYMBOL cap 1056 672 R0
SYMATTR InstName C_diff
SYMATTR Value 100p
SYMBOL nmos 736 272 R0
SYMATTR InstName MNAND1
SYMATTR Value ""
SYMBOL nmos 1680 672 R0
SYMATTR InstName MFLUSH_INV_N
SYMATTR Value ""
SYMBOL nmos 736 128 R0
SYMATTR InstName MNAND2
SYMATTR Value ""
SYMBOL nmos 1328 288 R0
SYMATTR InstName MINV_N
SYMATTR Value ""
SYMBOL res 976 192 R0
SYMATTR InstName R_pullup_nand
SYMATTR Value 1k
SYMBOL res 1232 656 R0
SYMATTR InstName R_diff
SYMATTR Value 1k
SYMBOL res 1408 640 R0
SYMATTR InstName R_clip_pulldown
SYMATTR Value 1k
SYMBOL g -144 336 R0
SYMATTR InstName G1
SYMBOL g -112 944 R0
SYMATTR InstName G2
SYMBOL res 288 192 R0
SYMATTR InstName R1
SYMATTR Value 1k
SYMBOL g 400 48 R0
SYMATTR InstName G3
SYMBOL voltage 480 368 R0
SYMATTR InstName V_VSS
SYMATTR Value -5
SYMBOL g 784 448 R0
SYMATTR InstName G4
SYMBOL voltage 992 32 R0
SYMATTR InstName V_1v
SYMATTR Value 1
```

```
SYMBOL pmos 1328 128 R0
SYMATTR InstName MINV_P
SYMATTR Value ""
SYMBOL g 1376 432 R0
SYMATTR InstName G5
SYMBOL OpAmps/OP07 784 640 R0
SYMATTR InstName XU2
SYMBOL OpAmps/OP07 512 192 R0
SYMATTR InstName XU1
SYMBOL diode 1152 672 R0
SYMATTR InstName D_clip
SYMATTR Value 10k
SYMBOL pmos 1680 528 R0
SYMATTR InstName MFLUSH_INV_P
SYMATTR Value ""
SYMBOL voltage 656 16 R0
SYMATTR InstName VDD
SYMATTR Value 5
SYMBOL voltage 528 96 R0
SYMATTR InstName V_3V3
SYMATTR Value 3.3
SYMBOL voltage 528 528 R0
SYMATTR InstName V_2V
SYMATTR Value 2
```