Fixing Pipeline Stall Bugs in Cache-Integrated RISC-V CPUs

Cache stalls cause data corruption in pipelined processors when write enables ignore stall conditions. Your five bugs stem from a single root cause: control signals propagating without checking `!cache_stall && !hazard_stall`, causing spurious register writes, invalid data sampling, and race conditions. The solution requires comprehensive stall gating at every write point in your pipeline.

The fundamental problem with cache integration

When you add caches to a working pipelined processor, the pipeline assumes single-cycle memory operations. Cache misses violate this assumption, creating multi-cycle stalls where the pipeline must freeze while waiting for data. Dongming Li +2 During these stalls, **your control signals continue propagating through pipeline registers, causing writes with invalid data**. The mem_data_reg samples garbage because it latches on every clock edge regardless of whether cache data is valid. Memory operations fire during stalls because nothing gates the read/write enables. The writeback stage writes zeros to registers because it doesn't check if the data is actually ready.

The solution is deceptively simple: **every write operation must check all stall conditions**. But implementing this correctly requires understanding how stalls propagate through pipelines and how different processor implementations handle this challenge.

Core principle: backward stall propagation

Stalls must propagate backward through the pipeline. Scribd When your MEM stage detects a cache miss, all earlier stages (EX, ID, IF) must freeze. Only later stages can complete their operations. Intel +2 Think of it like traffic: when cars ahead stop, cars behind must also stop, but cars that already passed the obstruction continue forward.

The correct stall propagation pattern:

verilog

```verilog
// Stalls propagate backward from their source
wire MEM_stall = d_cache_miss;
wire EX_stall = MEM_stall || ex_multicycle_op;
wire ID_stall = EX_stall || load_use_hazard;
wire IF_stall = ID_stall || i_cache_miss;

// Apply to pipeline register enables
assign PC_enable = ~IF_stall;
assign IFID_enable = ~ID_stall;
assign IDEX_enable = ~EX_stall;
assign EXMEM_enable = ~MEM_stall;
```

All major RISC-V implementations use this pattern. GitHub PicoRV32 takes the simplest approach with a global stall that freezes the entire pipeline. GitHub VexRiscv uses per-stage arbitration signals. GitHub +2 Rocket-chip employs valid bits with kill signals. GitHub BOOM's out-of-order architecture selectively stalls only dependent instructions.

Solution 1: Fix mem_data_reg sampling with valid gating

Your mem_data_reg returns zeros because it samples on every clock edge. During cache misses, the cache output is undefined or holds stale data. Redis Calstate **Only sample when data is actually valid.**

The correct pattern:

verilog

```verilog
// Cache controller must provide data_valid signal
wire cache_data_valid;
wire cache_ready;

always @(posedge clk) begin
   if (rst) begin
      mem_data_reg <= 32'b0;
   end else if (cache_ready && !cache_stall) begin
      // Sample ONLY when cache signals data is valid
      mem_data_reg <= cache_data_out;
   end
   // Otherwise: hold previous value
end
```

The cache controller's `cache_ready` signal indicates when data is actually available. During misses, this stays low until memory returns data. Iowa State University **The critical insight: combine cache_ready with !cache_stall to ensure you sample only when the pipeline can actually progress.**

PicoRV32 implements this through its mem_valid/mem_ready handshake. The core holds mem_valid high and keeps all outputs stable until mem_ready asserts. BoxLambda This guarantees that mem_rdata is only valid during the cycle where both signals are high. Rocket-chip uses a similar valid-ready protocol but with non-blocking caches that track multiple outstanding misses through MSHRs (Miss Status Holding Registers). GitHub GitHub

Alternative: FSM-based approach for cache controller

verilog

```verilog
parameter IDLE = 2'b00;
parameter COMPARE_TAG = 2'b01;
parameter ALLOCATE = 2'b10;

always @(posedge clk) begin
   case (state)
      COMPARE_TAG: begin
         if (cache_hit && !cache_stall) begin
            mem_data_reg <= cache_data;
            data_valid <= 1'b1;
         end else if (cache_miss) begin
            state <= ALLOCATE;
         end
      end
      ALLOCATE: begin
         if (mem_ready) begin
            mem_data_reg <= mem_data_in;
            data_valid <= 1'b1;
            state <= IDLE;
         end
      end
   endcase
end
```

This state machine approach ensures data sampling only occurs in defined states where data validity is guaranteed. The COMPARE_TAG state samples on cache hits, while ALLOCATE state samples when memory returns data after a miss. Blogger

Solution 2: Gate memory read/write enables with stall signals

Your memory operations execute during stalls because nothing prevents them. UW Computer Sciences **Every memory control signal must be AND-gated with inverted stall conditions.**

The comprehensive gating pattern:

```verilog
// Aggregate all stall sources
wire global_stall = cache_stall || hazard_stall || branch_stall;

// Gate ALL memory control signals
wire mem_read_enable_gated = mem_read_enable &&
                             !cache_stall &&
                             !hazard_stall &&
                             !branch_flush;

wire mem_write_enable_gated = mem_write_enable &&
                              !cache_stall &&
                              !hazard_stall &&
                              !branch_flush;

// Pass gated signals to cache/memory
dcache u_dcache (
    .read_enable(mem_read_enable_gated),
    .write_enable(mem_write_enable_gated),
    .addr(mem_addr),
    .write_data(mem_write_data),
    .read_data(mem_read_data),
    .stall(cache_stall)
);
```

VexRiscv implements this through its stage arbitration system. Each pipeline stage has an `arbitration.isStalled` signal that gates all operations in that stage. When the memory stage detects a cache miss, it asserts its stall signal, which automatically prevents memory operations from firing on subsequent cycles.

Why all three conditions matter: Cache stalls indicate the memory system isn't ready. Hazard stalls indicate data dependencies aren't resolved. Branch flushes indicate the instruction shouldn't execute at all. Missing any of these checks causes spurious operations.

Solution 3: Fix writeback register file writes

Your writeback stage writes invalid data to registers because RegWrite propagates without checking if the instruction actually completed. **The register file write enable must be the most heavily gated signal in your processor.**

The complete writeback write enable pattern:

```verilog
wire reg_write_enable_final;

assign reg_write_enable_final =
    WB_RegWrite &&              // Instruction requires write
    (WB_WriteReg != 5'b0) &&   // Not writing to x0 (RISC-V hardwired zero)
    !cache_stall &&            // No cache stall in pipeline
    !hazard_stall &&           // No data hazard stall
    !WB_flush &&               // Not flushed by branch mispredict
    WB_valid;                  // Instruction is valid (not a bubble)

// Register file with properly gated write
regfile RF (
    .clk(clk),
    .write_enable(reg_write_enable_final),
    .write_addr(WB_WriteReg),
    .write_data(WB_WriteData),
    .read_addr1(rs1),
    .read_addr2(rs2),
    .read_data1(rs1_data),
    .read_data2(rs2_data)
);
```

Rocket-chip uses a similar multi-condition approach with additional scoreboard tracking. The scoreboard maintains busy bits for each physical register, preventing reads of registers with pending writes. Tamu BOOM extends this with explicit register renaming, where physical register file writes can occur out-of-order but architectural register file updates happen only on instruction commit. Boom-core Chipyard

Critical protection: the x0 register check. In RISC-V, register x0 is hardwired to zero. Calstate Pipeline bubbles often manifest as NOP instructions that try to write to x0. Without the `(WB_WriteReg != 5'b0)` check, these can corrupt your register file logic or waste power.

Solution 4: Fix address decoding race conditions

Your top.v routing has race conditions because addresses change during stalls while tag comparisons are still active. **Latch addresses at the start of memory operations and hold them stable throughout the transaction.**

The address latching pattern:

verilog

```verilog
// Latch address when request begins
reg [31:0] mem_addr_latched;
reg addr_valid;

always @(posedge clk) begin
  if (rst) begin
    mem_addr_latched <= 32'b0;
    addr_valid <= 1'b0;
  end else if (!cache_stall && mem_request) begin
    // Latch on new request
    mem_addr_latched <= mem_addr;
    addr_valid <= 1'b1;
  end else if (cache_hit || (mem_ready && !cache_stall)) begin
    // Clear after completion
    addr_valid <= 1'b0;
  end
  // Hold during stall
end

// Use latched address for all cache operations
wire [TAG_BITS-1:0] tag = mem_addr_latched[31:INDEX_BITS+OFFSET_BITS];
wire [INDEX_BITS-1:0] index = mem_addr_latched[INDEX_BITS+OFFSET_BITS-1:OFFSET_BITS];
wire [OFFSET_BITS-1:0] offset = mem_addr_latched[OFFSET_BITS-1:0];
```

This prevents the address from changing while the cache controller processes a miss. During a multi-cycle cache miss, the EX stage might be computing a new address for the next instruction, but the MEM stage continues using the latched address for the current operation.

Why this matters: Without address latching, your cache controller might start a miss handling sequence with one address, then switch to a different address mid-transaction when the stall allows earlier stages to advance. This corrupts cache state and causes data to be loaded into wrong cache lines.

Solution 5: Universal write enable pattern

The overarching solution is applying the stall check pattern everywhere. **Every state-changing operation must verify !cache_stall && !hazard_stall before proceeding.**

Template for any write operation:

```verilog
always @(posedge clk) begin
    if (rst) begin
        // Reset logic
    end else if (
        write_enable &&            // Control signal says write
        !cache_stall &&            // No cache stall
        !hazard_stall &&           // No hazard stall
        !structural_stall &&       // No structural hazard
        !branch_flush &&           // Not flushed
        valid_instruction &&       // Instruction is valid
        (dest_register != 5'b0)    // Not writing to x0
    ) begin
        // Perform write
        register_file[dest_register] <= write_data;
    end
end
```

Apply this pattern to:

- **Register file writes** in WB stage
- **Memory writes** in MEM stage
- **Cache state updates** (valid bits, dirty bits, tags)
- **Pipeline register updates** between stages
- **Program counter updates** in IF stage
- **CSR writes** for control/status registers

Stall signal checking at each pipeline stage

The answer to whether stalls should be checked at every stage is nuanced. **Different stages have different stall requirements.**

IF (Instruction Fetch) stage

Checks: Instruction cache misses, branch target resolution **Response:** Freeze PC, prevent IF/ID register update **Pattern:**

```verilog
always @(posedge clk) begin
   if (!IF_stall && !hazard_stall && !cache_stall) begin
      PC <= PC_next;
      IFID_instruction <= i_cache_data;
   end
   // PC and instruction register hold during stall
end
```

ID (Instruction Decode) stage

Checks: Load-use hazards, register scoreboard conflicts **Response:** Insert bubble into ID/EX, stall IF/ID **Pattern:**

```verilog
wire load_use_hazard = EX_MemRead &&
                  ((EX_rd == ID_rs1) || (EX_rd == ID_rs2)) &&
                  (EX_rd != 5'b0);

assign ID_stall = load_use_hazard || cache_stall || hazard_stall;
assign flush_IDEX = load_use_hazard;  // Insert bubble
```

EX (Execute) stage

Checks: Multi-cycle operations (division), memory stalls propagating backward **Response:** Hold EX/MEM register **Pattern:**

```verilog
always @(posedge clk) begin
   if (!EX_stall) begin
      EXMEM_alu_result <= alu_result;
      EXMEM_mem_data <= mem_write_data;
   end
end
```

MEM (Memory Access) stage

Checks: Data cache misses (this is where cache_stall originates) Wikipedia **Response:** Freeze entire pipeline, wait for memory **Pattern:**

```verilog
assign cache_stall = (mem_read || mem_write) && !cache_ready;

always @(posedge clk) begin
   if (!cache_stall && cache_ready) begin
      MEMWB_data <= cache_data;
   end
end
```

WB (Writeback) stage

Checks: Typically no new stalls generated, but must respect stalls from earlier stages **Response:** Gate register writes with all stall signals **Pattern:**

```verilog
wire wb_write_enable = WB_RegWrite &&
                  !cache_stall &&
                  !hazard_stall &&
                  (WB_rd != 0);
```

Key insight: Stalls propagate backward. A MEM stage cache stall must freeze IF, ID, and EX stages. Wikipedia But WB stage operations (for instructions already past MEM) can complete. Wikipedia +2 This is why you check stalls at every stage but with different implications—early stages freeze, late stages complete.

How major RISC-V implementations handle this

PicoRV32: Global stall approach

Philosophy: Simplicity over performance. The entire core freezes on any stall. GitHub

```verilog
// Single mem_valid/mem_ready handshake controls everything
assign mem_valid = cpu_needs_memory;
wire transaction_complete = mem_valid && mem_ready;

// Core does nothing until transaction completes
always @(posedge clk) begin
   if (transaction_complete) begin
      // Advance to next instruction
      cpu_state <= cpu_state_next;
   end
   // Else: all state frozen
end
```

Advantages: Dead simple, minimal area, guaranteed correct **Disadvantages:** Poor IPC (instructions per cycle), every cache miss stalls everything

VexRiscv: Per-stage arbitration

Philosophy: Modular plugin architecture with stage-level control. GitHub GitHub

```scala
// Each stage has arbitration signals
stage.arbitration.isStalled := Bool
stage.arbitration.isFlushed := Bool
stage.arbitration.isValid := Bool

// Cache plugin sets stall when miss detected
when(cache_miss) {
  memoryStage.arbitration.isStalled := True
}
```

Advantages: Flexible, configurable pipeline depth, later stages can complete **Disadvantages:** More complex than PicoRV32, requires careful plugin coordination

Rocket-chip: Valid bits and kill signals

Philosophy: High-performance in-order with non-blocking caches.

```scala
// Each pipeline register has valid bit
val ex_reg_valid = Reg(Bool())
val mem_reg_valid = Reg(Bool())

// Scoreboard tracks pending writes
val scoreboard = Reg(Vec(32, Bool()))

// Kill signals cancel instructions
val ctrl_killd = Bool()
val ctrl_killx = Bool()
```

Advantages: Non-blocking cache allows independent instructions to proceed **Disadvantages:** Requires MSHRs, scoreboard, and forwarding logic GitHub

BOOM: Out-of-order execution

Philosophy: Hide cache latency through out-of-order execution. Boom-core Chipyard

Key mechanisms:

- Reorder buffer (ROB) tracks instruction completion order Boom-core Chipyard
- Issue queue holds instructions waiting for operands
- Physical register renaming eliminates false dependencies Boom-core Chipyard
- Load queue and store queue handle memory operations

During cache miss:

- Dependent instructions wait in issue queue
- Independent instructions continue executing
- ROB ensures in-order commit of architectural state Boom-core

Best practices for propagating stall signals

1. Centralize stall generation

Create a single module that aggregates all stall sources:

```verilog
module stall_controller(
    input i_cache_miss,
    input d_cache_miss,
    input load_use_hazard,
    input div_busy,
    input mem_not_ready,

    output global_stall,
    output IF_stall,
    output ID_stall,
    output EX_stall,
    output MEM_stall
);
    assign global_stall = i_cache_miss || d_cache_miss ||
                         load_use_hazard || div_busy ||
                         mem_not_ready;

    // Propagate backward
    assign MEM_stall = d_cache_miss || mem_not_ready;
    assign EX_stall = MEM_stall || div_busy;
    assign ID_stall = EX_stall || load_use_hazard;
    assign IF_stall = ID_stall || i_cache_miss;
endmodule
```

2. Register long combinational stall paths

If your stall detection logic creates timing violations, register the signal:

```verilog
// Combinational detection
wire stall_detected = /* complex logic */;

// Registered for timing
reg stall_registered;
always @(posedge clk) begin
   stall_registered <= stall_detected;
end

// Use registered version
assign PC_enable = ~stall_registered;
```

Tradeoff: This adds one cycle of latency to stall detection but fixes critical paths.

3. Separate stalls from flushes

Stalls mean "wait for data." Flushes mean "discard instruction." Wikipedia +2

```verilog
wire stall = cache_miss || load_use_hazard;
wire flush = branch_mispredict || exception;

// Different behaviors
assign IFID_enable = ~stall;
assign IFID_instruction = flush ? 32'h00000013 : // NOP
                          (stall ? IFID_instruction : new_instruction);
```

Branch mispredictions should flush the pipeline (convert wrong-path instructions to NOPs), not stall it. Cache misses should stall (wait for data), not flush.

4. Use valid bits throughout pipeline

Track instruction validity explicitly:

```verilog
// Add to pipeline registers
reg IFID_valid, IDEX_valid, EXMEM_valid, MEMWB_valid;

// Update valid bits
always @(posedge clk) begin
   if (rst || flush) begin
      IFID_valid <= 1'b0;
   end else if (!stall) begin
      IFID_valid <= IF_instruction_fetched;
   end
end

// Use for control
wire regfile_write = MEMWB_valid && WB_RegWrite && !stall;
```

Common mistakes when integrating caches

Mistake 1: Not propagating stalls backward

Wrong:

```verilog
// Only stalls MEM stage
assign EXMEM_enable = ~cache_miss;
```

Correct:

```verilog
// Propagates to all earlier stages
assign PC_enable = ~cache_miss;
assign IFID_enable = ~cache_miss;
assign IDEX_enable = ~cache_miss;
assign EXMEM_enable = ~cache_miss;
```

Mistake 2: PC advancing during instruction cache miss

Wrong:

```verilog
// PC increments even during i-cache miss
always @(posedge clk) begin
   PC <= PC + 4;
end
```

Correct:

```verilog
// PC freezes during any stall
always @(posedge clk) begin
    if (!i_cache_miss && !d_cache_miss && !hazard_stall) begin
        PC <= PC_next;
    end
end
```

Mistake 3: Assuming fixed cache miss penalty

Wrong:

```verilog
// Assumes 10-cycle miss penalty
reg [3:0] stall_counter;
always @(posedge clk) begin
    if (cache_miss) stall_counter <= 4'd10;
    else if (stall_counter > 0) stall_counter <= stall_counter - 1;
end
assign cache_stall = (stall_counter > 0);
```

Correct:

```verilog
// Use ready/valid handshaking
assign cache_stall = mem_request && !mem_ready;

always @(posedge clk) begin
    if (mem_ready && mem_request) begin
        // Sample data when memory signals ready
        mem_data_reg <= mem_data;
    end
end
```

Cache miss latency varies with DRAM timing, bus arbitration, and memory controller state. Redis Use handshaking protocols, not fixed-cycle counting. Stack Exchange Stack Exchange

Mistake 4: Ignoring write buffer interactions

Wrong:

```verilog
// Assumes stores complete immediately
always @(posedge clk) begin
    if (mem_write) begin
        memory[addr] <= write_data;
        // Immediately available to loads?
    end
end
```

Correct:

```verilog
// Check store queue for forwarding
wire store_pending = (pending_store_addr == load_addr) && store_in_queue;
wire load_stall = store_pending || cache_miss;

// Forward from store queue if addresses match
assign load_data = store_pending ? pending_store_data : cache_data;
```

Loads that follow stores to the same address must either stall until the store completes or forward data from the store queue. University of Maryland D... Otherwise, load-store ordering violations occur.

Specific Verilog implementation patterns

Complete hazard detection unit with cache awareness

verilog

```verilog
module hazard_detection_unit(
    // Current instruction in ID stage
    input [4:0] ID_rs1, ID_rs2,
    input ID_mem_read, ID_mem_write,

    // Instruction in EX stage
    input [4:0] EX_rd,
    input EX_mem_read,

    // Cache status
    input i_cache_miss,
    input d_cache_miss,

    // Stall outputs
    output reg PC_stall,
    output reg IFID_stall,
    output reg flush_IDEX
);
    // Load-use hazard detection
    wire load_use = EX_mem_read &&
                    ((EX_rd == ID_rs1) || (EX_rd == ID_rs2)) &&
                    (EX_rd != 5'b0);

    // Cache stalls
    wire cache_stall = i_cache_miss || d_cache_miss;

    always @(*) begin
        if (load_use) begin
            // Stall for 1 cycle, insert bubble
            PC_stall = 1'b1;
            IFID_stall = 1'b1;
            flush_IDEX = 1'b1;
        end else if (cache_stall) begin
            // Freeze entire pipeline
            PC_stall = 1'b1;
            IFID_stall = 1'b1;
            flush_IDEX = 1'b0;  // Don't flush, just wait
        end else begin
            PC_stall = 1'b0;
            IFID_stall = 1'b0;
            flush_IDEX = 1'b0;
        end
    end
endmodule
```

Cache-aware MEM stage with proper data sampling

verilog

```verilog
module MEM_stage(
    input clk, rst,
    input [31:0] alu_result,
    input [31:0] write_data,
    input mem_read, mem_write,
    output reg [31:0] mem_data,
    output reg mem_data_valid,
    output cache_stall
);
    // Cache interface
    wire cache_hit, cache_ready;
    wire [31:0] cache_data_out;

    data_cache dcache(
        .clk(clk),
        .rst(rst),
        .addr(alu_result),
        .write_data(write_data),
        .mem_read(mem_read && !cache_stall),    // Gated
        .mem_write(mem_write && !cache_stall),  // Gated
        .data_out(cache_data_out),
        .hit(cache_hit),
        .ready(cache_ready)
    );

    // Stall when operation active but not ready
    assign cache_stall = (mem_read || mem_write) && !cache_ready;

    // Sample data only when valid
    always @(posedge clk) begin
        if (rst) begin
            mem_data <= 32'b0;
            mem_data_valid <= 1'b0;
        end else if (cache_ready && mem_read && !cache_stall) begin
            mem_data <= cache_data_out;
            mem_data_valid <= 1'b1;
        end else if (!mem_read) begin
            mem_data_valid <= 1'b0;
        end
    end
endmodule
```

Complete top-level integration

verilog

```verilog
module riscv_pipeline_with_cache(
    input clk, rst
);
    // Stall signals
    wire hazard_stall, i_cache_stall, d_cache_stall;
    wire global_stall = hazard_stall || i_cache_stall || d_cache_stall;

    // IF Stage with stall awareness
    always @(posedge clk) begin
        if (rst) begin
            PC <= 32'b0;
        end else if (!global_stall && !branch_flush) begin
            PC <= branch_taken ? branch_target : PC + 4;
        end
        // PC frozen during stall
    end

    // IF/ID Pipeline Register
    always @(posedge clk) begin
        if (rst || branch_flush) begin
            IFID_instruction <= 32'h00000013;  // NOP
            IFID_PC <= 32'b0;
            IFID_valid <= 1'b0;
        end else if (!global_stall) begin
            IFID_instruction <= instruction;
            IFID_PC <= PC;
            IFID_valid <= 1'b1;
        end
        // Hold during stall
    end

    // Hazard Detection
    hazard_detection_unit HDU(
        .ID_rs1(rs1),
        .ID_rs2(rs2),
        .EX_rd(EX_rd),
        .EX_mem_read(EX_MemRead),
        .i_cache_miss(i_cache_stall),
        .d_cache_miss(d_cache_stall),
        .PC_stall(hazard_stall),
        .flush_IDEX(flush_signal)
    );

    // Register file with comprehensive write enable gating
    wire reg_write_enable = WB_RegWrite &&
                            !global_stall &&
                            !flush &&
                    (WB_rd != 5'b0) &&
                    WB_valid;

    regfile RF(
        .clk(clk),
        .write_enable(reg_write_enable),
        .write_addr(WB_rd),
        .write_data(WB_write_data),
        .read_addr1(rs1),
        .read_addr2(rs2),
        .read_data1(rs1_data),
        .read_data2(rs2_data)
    );

    // Remaining stages with similar stall-aware control...
endmodule
```

Testing and validation

Assertion-based verification

Add SystemVerilog assertions to catch bugs during simulation:

systemverilog

```systemverilog
// Assert: never write during stall
property no_write_during_stall;
    @(posedge clk) global_stall |-> !reg_write_enable;
endproperty
assert property (no_write_during_stall)
    else $error("Register write during stall detected!");

// Assert: data valid implies cache ready
property data_valid_means_ready;
    @(posedge clk) mem_data_valid |-> cache_ready;
endproperty
assert property (data_valid_means_ready)
    else $error("Data marked valid but cache not ready!");

// Assert: PC doesn't change during stall
property pc_stable_during_stall;
    @(posedge clk) global_stall |=> ($stable(PC) || branch_taken);
endproperty
assert property (pc_stable_during_stall)
    else $error("PC changed during stall!");
```

Critical test cases

1. **Load-after-store with cache miss:** Store to address X, immediately load from X, with cache miss on the load
2. **Consecutive loads from same line:** Multiple loads hitting same cache line, with miss on first load
3. **Write-during-stall:** Trigger cache miss, verify no register writes occur during stall cycles
4. **Branch-flush during cache miss:** Branch misprediction while waiting for cache miss to resolve

Waveform debugging checklist

When debugging in simulation, verify:

- **Stall propagation:** When cache_stall asserts, verify PC_enable, IFID_enable all deassert on same cycle
- **Data sampling:** mem_data_reg only changes when both cache_ready and !cache_stall are true
- **Write enable timing:** reg_write_enable goes low immediately when any stall asserts
- **Address stability:** mem_addr_latched remains constant throughout cache miss sequence

Conclusion

Your five bugs all stem from incomplete stall checking. The solution requires a systematic approach: identify every state-changing operation in your processor (register writes, memory operations, pipeline register updates, PC changes), and gate each with `!cache_stall && !hazard_stall`. Memory data registers must sample only when cache_ready signals valid data. Medium Addresses must latch at request start and remain stable during multi-cycle operations.

The pattern is universal across all successful RISC-V implementations—PicoRV32's simple global stall, VexRiscv's stage arbitration, Rocket's valid bits with scoreboards, and BOOM's out-of-order execution all implement this fundamental principle: **state changes only when the pipeline can actually progress**. Apply this pattern everywhere, validate with assertions and comprehensive tests, and your cache-integrated pipeline will be both correct and robust.