RISC-V CPU Store-to-Load Forwarding Bug Analysis and Fix Guide

Executive Summary

Your RISC-V CPU's store-to-load forwarding is completely broken, causing loads to return zero instead of recently stored values. This cascading failure corrupts arithmetic operations (x8=100 instead of 142), control flow decisions, and causes program crashes before x13/x14 can be written. The root cause is one or more missing components in the three-part forwarding system: detection logic, data path propagation, and forwarding multiplexer.

Root Cause: The Three Missing Links

Store-to-load forwarding requires **all three components** to work:
Chipmunk Logic +5

Component 1: Store Data Pipeline Propagation (Most Likely Broken)

The Bug: Store data from register file never reaches the MEM or WB stages where it can be forwarded.

What Should Happen:

- Cycle 1 (ID): Register file outputs store data on `rt_data` port
- Cycle 2 (EX): Data held in ID/EX pipeline register
- Cycle 3 (MEM): Data transferred to EX/MEM register as `EX_MEM_store_data`
- Cycle 4 (WB): Data transferred to MEM/WB register as `MEM_WB_store_data`

Critical Fix in riscv_cpu.v:

verilog

```verilog
// ID/EX Pipeline Register - Capture store data from register file
always @(posedge clk or posedge rst) begin
    if (rst) begin
        ID_EX_rt_data <= 32'b0;
        ID_EX_MemWrite <= 1'b0;
        ID_EX_MemRead <= 1'b0;
    end else if (!stall) begin
        ID_EX_rt_data <= reg_read_data2;  // Data to be stored
        ID_EX_MemWrite <= ID_MemWrite;
        ID_EX_MemRead <= ID_MemRead;
    end
end

// EX/MEM Pipeline Register - CRITICAL: Must include store_data
always @(posedge clk or posedge rst) begin
    if (rst) begin
        EX_MEM_store_data <= 32'b0;
        EX_MEM_mem_addr <= 32'b0;
        EX_MEM_MemWrite <= 1'b0;
        EX_MEM_MemRead <= 1'b0;
    end else if (!cache_stall) begin  // Only gate on cache miss
        EX_MEM_store_data <= ID_EX_rt_data;  // ← OFTEN MISSING
        EX_MEM_mem_addr <= EX_alu_result;
        EX_MEM_MemWrite <= ID_EX_MemWrite;
        EX_MEM_MemRead <= ID_EX_MemRead;
    end
end

// MEM/WB Pipeline Register - Forward from WB stage for 2-cycle-apart stores
always @(posedge clk or posedge rst) begin
    if (rst) begin
        MEM_WB_store_data <= 32'b0;
        MEM_WB_mem_addr <= 32'b0;
        MEM_WB_MemWrite <= 1'b0;
    end else if (!cache_stall) begin
        MEM_WB_store_data <= EX_MEM_store_data;  // ← OFTEN MISSING
        MEM_WB_mem_addr <= EX_MEM_mem_addr;
        MEM_WB_MemWrite <= EX_MEM_MemWrite;     // ← OFTEN MISSING
    end
end
```

Why This Causes Your Bug: Without `EX_MEM_store_data` and `MEM_WB_store_data` assignments, these registers default to zero. When forwarding logic tries to forward "store data", it forwards zero.

Component 2: Hazard Detection Logic (Likely Missing or Broken)

The Bug: Address comparison fails to detect that load and store access the same memory location.

Fixed store_load_detector.v:

verilog

```verilog
module store_load_detector (
    // Current load in EX stage calculating address
    input [31:0] EX_mem_addr,
    input EX_MemRead,
    input [1:0] EX_mem_size,  // 00=byte, 01=half, 10=word

    // Store currently in MEM stage
    input [31:0] MEM_mem_addr,
    input MEM_MemWrite,
    input [1:0] MEM_mem_size,

    // Store currently in WB stage (2 cycles old)
    input [31:0] WB_mem_addr,
    input WB_MemWrite,
    input [1:0] WB_mem_size,

    // Forwarding control outputs
    output wire forward_from_MEM,
    output wire forward_from_WB
);

    // Word-aligned address comparison (ignore bottom 2 bits for word access)
    wire addr_match_MEM_word = (EX_mem_addr[31:2] == MEM_mem_addr[31:2]);
    wire addr_match_WB_word = (EX_mem_addr[31:2] == WB_mem_addr[31:2]);

    // Size matching - both must be word access for simple forwarding
    wire size_match_MEM = (EX_mem_size == 2'b10) && (MEM_mem_size == 2'b10);
    wire size_match_WB = (EX_mem_size == 2'b10) && (WB_mem_size == 2'b10);

    // Full hazard detection
    wire hazard_MEM = EX_MemRead && MEM_MemWrite &&
                      addr_match_MEM_word && size_match_MEM;
    wire hazard_WB = EX_MemRead && WB_MemWrite &&
              addr_match_WB_word && size_match_WB;

    // Priority: MEM stage is more recent than WB stage
    assign forward_from_MEM = hazard_MEM;
    assign forward_from_WB = hazard_WB && !hazard_MEM;

endmodule
```

Critical Details:

- **Word alignment:** Compare `addr[31:2]` not full address (bottom 2 bits are byte offset within word) Stack Overflow
- **Both control signals:** Check both `MemRead` AND `MemWrite` are asserted
- **Priority logic:** If both MEM and WB have matching stores, prefer MEM (more recent)

Component 3: Forwarding Multiplexer (Likely Missing)

The Bug: Memory unit always returns data from memory, never from forwarding path. Wikipedia

Fixed store_load_forward.v:

verilog

```verilog
module store_load_forward (
    // Three possible data sources
    input [31:0] mem_read_data,      // From data_mem.v
    input [31:0] MEM_store_data,     // From EX/MEM pipeline register
    input [31:0] WB_store_data,      // From MEM/WB pipeline register

    // Control signals from detector
    input forward_from_MEM,
    input forward_from_WB,

    // Final output to register file
    output reg [31:0] final_load_data
);

    // Three-way priority multiplexer
    always @(*) begin
        if (forward_from_MEM)
            final_load_data = MEM_store_data;      // Highest priority
        else if (forward_from_WB)
            final_load_data = WB_store_data;       // Second priority
        else
            final_load_data = mem_read_data;       // Default: read from memory
    end

endmodule
```

Integration in memory_unit.v:

verilog

```verilog
module memory_unit (
    input clk,
    input rst,

    // Current memory operation (MEM stage)
    input [31:0] MEM_mem_addr,
    input [31:0] MEM_store_data,
    input MEM_MemWrite,
    input MEM_MemRead,

    // For hazard detection - load in EX stage
    input [31:0] EX_mem_addr,
    input EX_MemRead,

    // Previous store in WB stage
    input [31:0] WB_mem_addr,
    input [31:0] WB_store_data,
    input WB_MemWrite,

    // Cache control
    input cache_stall,

    // Output
    output [31:0] load_data_out
);

    wire [31:0] mem_read_data;
    wire forward_from_MEM, forward_from_WB;

    // Instantiate data memory
    data_mem dmem (
        .clk(clk),
        .addr(MEM_mem_addr[15:2]),  // Word address
        .write_data(MEM_store_data),
        .write_enable(MEM_MemWrite && !cache_stall),
        .read_enable(MEM_MemRead),
        .read_data(mem_read_data)
    );

    // Instantiate hazard detector
    store_load_detector detector (
        .EX_mem_addr(EX_mem_addr),
        .EX_MemRead(EX_MemRead),
        .EX_mem_size(2'b10),  // Assuming word access
        .MEM_mem_addr(MEM_mem_addr),
        .MEM_MemWrite(MEM_MemWrite),
        .MEM_mem_size(2'b10),
        .WB_mem_addr(WB_mem_addr),
        .WB_MemWrite(WB_MemWrite),
        .WB_mem_size(2'b10),
        .forward_from_MEM(forward_from_MEM),
        .forward_from_WB(forward_from_WB)
    );

    // Instantiate forwarding mux
    store_load_forward forwarder (
        .mem_read_data(mem_read_data),
        .MEM_store_data(MEM_store_data),
        .WB_store_data(WB_store_data),
        .forward_from_MEM(forward_from_MEM),
        .forward_from_WB(forward_from_WB),
        .final_load_data(load_data_out)
    );

endmodule
```

Why Your Recent Stall Gating Fixes Made It Worse

You mentioned "comprehensive gating to memory_unit.v and writeback.v" - this likely **blocks store data propagation during stalls**.

The Problem:

```verilog
// WRONG - Blocks all updates during any stall
always @(posedge clk) begin
  if (!stall && !cache_stall) begin  // Too restrictive
    EX_MEM_store_data <= ID_EX_rt_data;
  end
end
```

Why This Breaks Forwarding: Pipeline stalls for load-use hazards or cache misses. Wikipedia During stalls, instructions must continue holding their data. Stack Overflow Wikipedia If gating prevents `EX_MEM_store_data` from updating, forwarding sees stale/zero values.

The Fix:

```verilog
// CORRECT - Store data always propagates except during cache miss
always @(posedge clk) begin
  if (!cache_stall) begin  // Only gate on cache miss, NOT on stalls
    EX_MEM_store_data <= ID_EX_rt_data;
    EX_MEM_mem_addr <= EX_alu_result;
  end
  // During regular stalls, data just holds its value (normal register behavior)
end
```

Key Principle: Pipeline registers should update on every clock cycle **except during cache misses**. Regular pipeline stalls (load-use hazards, branch delays) work by inserting bubbles (NOP instructions), not by freezing registers.

University of Maryland D...

Why x13 and x14 Never Get Written: The Cascade Effect

Failure Chain Analysis

Initial Corruption (x6 = 0 instead of 42):

```assembly
sw x5, 0(x10)      # Stores 42 to memory
lw x6, 0(x10)      # BUG: Returns 0 instead of 42
```

Arithmetic Corruption (x8 = 100 instead of 142):

```assembly
addi x7, x0, 100   # x7 = 100
add x8, x6, x7     # x8 = 0 + 100 = 100 (should be 42 + 100 = 142)
```

Control Flow Corruption (x10 = 1 instead of 143):

```assembly
addi x10, x8, 1    # x10 = 100 + 1 = 101 (should be 142 + 1 = 143)
```

Program Crash Before x13/x14:

If x10 is used as an address or branch condition: Medium

```assembly
beq x10, x11, skip     # Wrong branch taken due to x10=101 vs 143
  # ... code that should execute ...
  addi x13, x12, 5     # Never reached
skip:
  lw x14, 0(x10)       # Address 101 is invalid → Load Access Fault
                       # Exception handler or timeout, x13/x14 never written
```

Three Crash Scenarios:

1. **Invalid Memory Access:** x10=101 used as address causes Load/Store Access Fault (exception code 5/7)
2. **Infinite Loop:** Wrong branch condition causes loop to never terminate, watchdog timeout
3. **Illegal Instruction:** Corrupted PC jumps to data region, attempts to execute non-instructions (exception code 2) Uni-freiburg +2

Data Memory Timing Issue: Write-Through vs Forwarding

The Secondary Bug

Even with perfect forwarding logic, `data_mem.v` may have a timing issue:

Problem Scenario:

```
Cycle N:   Store writes to data_mem[addr] = 42
Cycle N+1: Load reads from data_mem[addr] = ?
```

If data_mem uses synchronous write:

verilog

```verilog
// PROBLEMATIC - Write doesn't complete until next cycle
always @(posedge clk) begin
    if (write_enable)
        mem[addr] <= write_data;  // Updates at clock edge
end

assign read_data = mem[read_addr];  // Reads old value before update
```

The load in cycle N+1 reads **before** the store's write completes, getting stale data.

Wikipedia

Solution 1: Add internal forwarding in data_mem.v:

verilog

```verilog
module data_mem (
    input clk,
    input [13:0] addr,          // Word address
    input [31:0] write_data,
    input write_enable,
    input [13:0] read_addr,
    output reg [31:0] read_data
);

    reg [31:0] mem [0:16383];
    reg [31:0] write_buffer_data;
    reg [13:0] write_buffer_addr;
    reg write_buffer_valid;

    // Synchronous write
    always @(posedge clk) begin
        if (write_enable) begin
            mem[addr] <= write_data;
            write_buffer_data <= write_data;
            write_buffer_addr <= addr;
            write_buffer_valid <= 1'b1;
        end else begin
            write_buffer_valid <= 1'b0;
        end
    end

    // Read with write forwarding
    always @(*) begin
        if (write_buffer_valid && (read_addr == write_buffer_addr))
            read_data = write_buffer_data;  // Forward just-written data
        else
            read_data = mem[read_addr];
    end
```

```
endmodule
```

Solution 2: Rely on pipeline forwarding (preferred):

If your store-load forwarding in `memory_unit.v` is correct, it will bypass `data_mem` entirely for back-to-back sequences. The memory timing only matters for loads 2+ cycles after stores.

Waveform Analysis: Your Debugging Roadmap

Phase 1: Verify Store Data Propagation (30 minutes)

Test Program:

```assembly
li x5, 42          # Load immediate
sw x5, 0(x10)      # Store 42 to memory
nop
nop
```

Critical Waveform Signals:

```
Signal                 | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Expected
-----------------------|---------|---------|---------|---------|----------
reg_read_data2         |    -    |   42    |    -    |    -    | 42 in cycle 2
ID_EX_rt_data          |    -    |    -    |   42    |    -    | 42 in cycle 3
EX_MEM_store_data      |    -    |    -    |    -    |   42    | 42 in cycle 4
MEM_WB_store_data      |    -    |    -    |    -    |    -    | 42 in cycle 5
```

If EX_MEM_store_data = 0: Pipeline register assignment missing in EX/MEM always block **If ID_EX_rt_data = 0:** Not capturing register file output in ID/EX register

Phase 2: Verify Hazard Detection (30 minutes)

Test Program:

```assembly
sw x5, 0(x10)      # Store 42
lw x6, 0(x10)      # Load immediately after
```

Critical Waveform Signals:

```
Signal                | Cycle N | Cycle N+1 | Expected
----------------------|---------|-----------|----------
MEM_MemWrite          |    1    |     0     | High when store in MEM
EX_MemRead            |    0    |     1     | High when load in EX
MEM_mem_addr[31:2]    |  0x400  |   0x400   | Store address
EX_mem_addr[31:2]     |  0x???  |   0x400   | Load address (same)
addr_match_MEM        |    0    |     1     | Should detect match
forward_from_MEM      |    0    |     1     | Should activate
```

If forward_from_MEM = 0:

- Check `MEM_MemWrite` signal is HIGH during store's MEM stage
- Check `EX_MemRead` signal is HIGH during load's EX stage
- Check address comparison: `EX_mem_addr[31:2] == MEM_mem_addr[31:2]`
- Verify store_load_detector instantiated and connected

Phase 3: Verify Forwarding Data Path (30 minutes)

Critical Waveform Signals:

```
Signal               | Value | Expected | Diagnosis if Wrong
---------------------|-------|----------|------------------
EX_MEM_store_data    |  42  |   42    | Pipeline propagation broken
forward_from_MEM     |   1  |    1    | Detection logic broken
mem_read_data        |   0  |  (any)  | Memory might have stale data
final_load_data      |  42  |   42    | ← THIS IS YOUR GOAL
destination_register |  x6  |   x6    | Correct register
x6_final_value       |  42  |   42    | Written to register file
```

If final_load_data = 0 when forward_from_MEM = 1:

- Forwarding MUX not implemented or not connected
- Check `store_load_forward` module instantiation
- Verify `MEM_store_data` input connected to `EX_MEM_store_data`
- Verify `forward_from_MEM` input connected to detector output

If final_load_data = mem_read_data when should be MEM_store_data:

- MUX select logic inverted or incorrect
- Check `always @(*)` block in `store_load_forward.v`
- Verify priority: `forward_from_MEM` has highest priority

Phase 4: Check Stall/Cache Interaction (45 minutes)

Test with cache activity or artificial stalls

Critical Waveform Signals:

```
Signal                | Check For
----------------------|----------------------------------------
cache_stall           | Should be 0 during normal operation
pipeline_stall        | May be 1 for load-use hazards
EX_MEM write_enable   | Should be 1 even during pipeline_stall
EX_MEM_store_data     | Should update every cycle (not gated by stall)
```

If data stops propagating during stalls:

- Pipeline register gating too restrictive
- Change condition from `if (!stall)` to `if (!cache_stall)`
- Store data must propagate independently of pipeline stalls

Phase 5: Full Integration Test (1 hour)

Run complete test program:

```assembly
li x5, 42
sw x5, 0(x10)
lw x6, 0(x10)      # Should get 42
addi x7, x0, 100
add x8, x6, x7     # Should get 142
addi x10, x8, 1    # Should get 143
sw x10, 4(x10)
lw x13, 4(x10)     # Should get 143
addi x14, x13, 1   # Should get 144
```

Trace execution flow:

- Monitor PC each cycle
- Watch for unexpected branches
- Check for exceptions (monitor `mcause`, `mepc`)
- Verify all registers get correct values

If program crashes before x13:

- Identify which instruction fails (check `mepc`)
- Trace backwards to find corrupted data
- Follow data dependency chain to original load

Prioritized Action Plan

IMMEDIATE (Next 1 Hour) - Fix Pipeline Registers

Action: Open `riscv_cpu.v` and verify/add these assignments:

```verilog
// In EX/MEM pipeline register always block
EX_MEM_store_data <= ID_EX_rt_data;  // ADD IF MISSING

// In MEM/WB pipeline register always block
MEM_WB_store_data <= EX_MEM_store_data;  // ADD IF MISSING
MEM_WB_MemWrite <= EX_MEM_MemWrite;      // ADD IF MISSING
```

Test: Recompile and check waveforms for `EX_MEM_store_data` signal.

Expected Result: This single fix may solve 60-70% of the problem if data was never propagating.

CRITICAL (Next 2 Hours) - Implement Forwarding Logic

Action: Create or fix these three modules:

1. **store_load_detector.v** - Use complete code provided above
2. **store_load_forward.v** - Use complete code provided above
3. **memory_unit.v** - Instantiate both modules with correct connections

Test: Run store-load sequence and verify `forward_from_MEM` signal activates.

Expected Result: Loads should now receive forwarded data instead of zero.

IMPORTANT (Next 1 Hour) - Fix Stall Gating

Action: Review all `always @(posedge clk)` blocks that have stall conditions.

Change: Remove `pipeline_stall` from gating conditions, only keep `cache_stall`:

```verilog
// BEFORE (wrong)
if (!stall && !cache_stall) begin

// AFTER (correct)
if (!cache_stall) begin
```

Expected Result: Forwarding continues working during load-use hazard stalls.

VERIFICATION (Next 2 Hours) - Full Integration

Action: Run complete test suite:

1. Simple store-load (back-to-back)
2. Store-load with 1 NOP between
3. Store-load with 2 NOPs between
4. Multiple stores to same address (test priority)
5. Original failing test (x6, x8, x10, x13, x14)

Expected Result: All tests pass, all registers get correct values.

Quick Diagnostic Checklist

Use this checklist to rapidly identify which component is broken:

```
☐ Step 1: Check EX_MEM_store_data in waveform during store
  → If 0: Pipeline register assignment missing (MOST LIKELY)
  → If correct (42): Continue to Step 2

☐ Step 2: Check forward_from_MEM during store-load sequence
  → If 0: Hazard detection broken (check addresses, control signals)
  → If 1: Continue to Step 3

☐ Step 3: Check final_load_data during forwarding
  → If 0 or mem_read_data: Forwarding MUX broken/missing
  → If 42: Forwarding works! Check register file write

☐ Step 4: Check destination register gets written
  → If not written: Check RegWrite signal, register file module
  → If written with wrong value: Check data path from final_load_data
```

Expected Outcomes After Fixes

Test Results Should Change To:

```
Register | Before | After Fix | Explanation
---------|--------|-----------|-------------
x6       |  42    |   42      | Direct store (already working)
x8       |  100   |   142     | Gets forwarded 42, computes 42+100=142
x10      |   1    |   143     | Gets correct 142, computes 142+1=143
x13      | none   |   143     | Program completes, loads forwarded value
x14      | none   |   144     | Program completes, computes 143+1=144
```

Program Execution:

- No crashes or exceptions
- All branches take correct paths
- No infinite loops or timeouts
- All instructions complete successfully

Summary: The Core Problem

Your CPU has **no working store-to-load forwarding**. This is not a subtle bug - it's a missing feature. The three components (detection, data path, forwarding MUX) are either missing or incorrectly implemented. Fiveable Wikipedia

Primary culprit (80% probability): Store data never propagates through `EX_MEM_store_data` pipeline register, so forwarding logic has nothing to forward.

Secondary culprit (15% probability): Forwarding MUX not implemented in `memory_unit.v`, always returns memory data.

Tertiary culprit (5% probability): Hazard detection address comparison broken.

Start with Phase 1 waveform analysis - within 30 minutes you'll know exactly which component is broken. Fix pipeline registers first, then add forwarding logic, then verify integration.

The cascading failures (crashes, missing register writes) will resolve automatically once forwarding works correctly. Medium