Implementing Cache Integration in 5-Stage Pipelined RISC-V CPUs

Your garbage write problem during cache stalls reveals a critical but common bug pattern: **write enables aren't properly gated when pipeline bubbles are inserted**. During your 19-cycle instruction cache miss, even though ID_EX outputs valid=0, the execution unit continues computing and the memory stage attempts writes with stale data. The spurious write to 0x00000000 indicates an uninitialized address register value leaking through. Wikipedia This happens because **validity checks must occur before execution, not after** — yet your forwarding logic computes addresses combinationally before checking the valid bit.

The root cause: execution proceeds despite valid=0

Your description reveals the core issue: "The execution unit computes addresses even when valid_in=0 due to forwarding logic computing before validity checks." This is a fundamental misunderstanding of how validity should flow through pipeline stages. The solution requires understanding three critical principles:

Principle 1: Valid bits control state changes, not computation. In well-designed pipelines, ALU and address calculation units operate every cycle regardless of instruction validity — this simplifies control logic and improves timing. The critical distinction is that their outputs must be **gated before any architectural state changes**. ZipCPU The ZipCPU documentation explains this perfectly: "Control signals don't actually activate the components, but instead choose whether to accept or ignore their output(s) after they have computed something." Stack Overflow

Principle 2: Control signals ARE validity indicators. Most successful implementations don't use explicit valid bits. Instead, they use **implicit validity through control signal zeroing**. When a bubble is inserted, all control signals (RegWrite, MemWrite, MemRead) are set to 0. GitHub This naturally creates a NOP that propagates through the pipeline harmlessly. University of Maryland D... Your valid=0 signal should be setting all these control signals to 0, but apparently your write enables aren't being gated.

Principle 3: Forwarding must check validity first. The standard forwarding condition is: Forward = (RegWrite == 1) AND (Rd != 0) AND (Rd == Rs). Notice RegWrite is checked **first** — this is the validity check. During bubbles, RegWrite=0, so forwarding doesn't occur. U-aizu UCSD Your forwarding logic computing "before validity checks" violates this fundamental pattern.

How valid bits should flow through pipeline stages during cache stalls

The most robust approach uses a **three-signal pipeline control model** per stage, documented extensively in ZipCPU implementations:

Valid signal indicates whether a stage contains legitimate instruction data. It's a **registered signal** that propagates forward with the instruction. On reset or pipeline flush, all valid bits clear to 0. The update logic for each stage:

```verilog
always @(posedge clk) begin
    if (reset || clear_pipeline)
        stage[n]_valid <= 1'b0;
    else if (stage[n]_ce)  // Clock enable
        stage[n]_valid <= stage[n-1]_valid;
    else if (stage[n+1]_ce)
        stage[n]_valid <= 1'b0;  // Next stage took our data
end
```

Stalled signal indicates whether a stage cannot advance. It's computed combinationally and propagates **backwards**: `stage[n]_stalled = stage[n]_valid && (stage[n+1]_stalled || local_stall_conditions)`. This ensures upstream stages don't advance when downstream stages are blocked.

Clock Enable (CE) controls when data advances to the next stage: `stage[n]_ce = stage[n-1]_valid && !stage[n]_stalled`. This acts as the handshake — data only clocks forward when both source has valid data AND destination can accept it.

For your 19-cycle cache miss, here's the correct behavior cycle-by-cycle:

- **Cycle 1**: IF stage detects cache miss, asserts miss signal. Stall propagates backwards combinationally in same cycle. PC write disabled, IF/ID register write disabled. University of Maryland D...
- **Cycles 2-19**: PC frozen, IF/ID frozen (holds the address that caused the miss). ID stage receives a bubble (all control signals = 0) from the stall logic. This bubble propagates: ID→EX→MEM→WB over the next 3 cycles. University of Maryland D... After cycle 4, the pipeline contains only bubbles downstream of the stalled IF/ID.
- **Cycle 20**: Cache returns data, deasserts miss. Stall releases. PC and IF/ID resume normal operation.

Critical insight: You don't stall the entire pipeline — you freeze fetch/decode while draining execution stages. The bubble insertion is what prevents garbage writes.

Preventing garbage data propagation when pipeline stages stall

Multiple levels of write protection are required:

Level 1: Control signal zeroing at bubble insertion point. When your hazard detection unit detects the cache miss and decides to insert a bubble, it must zero ALL control signals for the affected stage:

```verilog
if (cache_miss_detected) begin
    // Insert bubble in ID/EX pipeline register
    ID_EX_RegWrite <= 0;
    ID_EX_MemWrite <= 0;
    ID_EX_MemRead <= 0;
    ID_EX_Branch <= 0;
    // All other control signals to 0
    // This creates a NOP
end
```

The Patterson & Hennessy approach specifies: "Deasserting all nine control signals (setting them to 0) in the EX, MEM, and WB stages will create a 'do nothing' or NOP instruction." University of Maryland D...

Level 2: Write enable gating with stall signal. Even with control signals zeroed, add paranoid AND gates on all write enables:

```verilog
assign actual_reg_write_enable = WB_RegWrite && WB_valid && !pipeline_stalled;
assign actual_mem_write_enable = MEM_MemWrite && MEM_valid && !pipeline_stalled;
```

This is **defense in depth** — even if control signal zeroing fails, the write enable gating prevents corruption.

Level 3: Valid bit propagation. Your ID_EX valid bit should travel with the instruction through all subsequent stages:

```verilog
always @(posedge clk) begin
    if (flush)
        EX_MEM_valid <= 0;
    else if (!stall)
        EX_MEM_valid <= ID_EX_valid;
    // else hold current value
end
```

Then use this at writeback: `register_write = WB_RegWrite && WB_valid`. Never write when valid=0.

Your specific bug: The write to 0x00000000 during cycle 19 suggests your MEM stage write enable isn't being gated by the valid bit or stall signal. Even though ID_EX outputs valid=0, this isn't preventing the MEM stage from attempting a write. You need to trace where the write enable signal originates and ensure it includes: `mem_write_enable = instruction_MemWrite && stage_valid && !stall_signal`.

Proper handling of 19-cycle cache misses and pipeline bubbles

For multi-cycle stalls like your 19-cycle DRAM access, the key is **partial pipeline stalling** with **bubble propagation**:

Instruction cache miss protocol:

1. **Freeze upstream stages**: PC register and IF/ID pipeline register write enables are disabled. These stages hold their current values for all 19 cycles.
   University of Maryland D...
2. **Drain downstream stages**: ID, EX, MEM, WB stages continue processing existing instructions. After the last valid instruction exits, they contain only bubbles.
   University of Maryland D...
3. **Insert initial bubble**: On the first cycle of the stall, the hazard unit forces ID_EX control signals to 0, creating a bubble that will propagate through EX→MEM→WB.
   University of Texas University of Maryland D...
4. **Maintain bubbles**: For cycles 2-19, the frozen IF/ID stage continues trying to advance to ID/EX, but since it's frozen, only bubbles (zeros) propagate downstream.
5. **Resume on completion**: When cache_ready asserts, PC and IF/ID unfreeze simultaneously, and the stalled instruction proceeds to decode.

The pipeline state visualization:

```
Cycle 1:  LOAD(IF-MISS) | ADD(ID) | SUB(EX) | OR(MEM) | AND(WB)
Cycle 2:  LOAD(IF-WAIT) | ADD(ID) | BUBBLE  | SUB(MEM)| OR(WB)
Cycle 3:  LOAD(IF-WAIT) | ADD(ID) | BUBBLE  | BUBBLE  | SUB(WB)
Cycle 4:  LOAD(IF-WAIT) | ADD(ID) | BUBBLE  | BUBBLE  | BUBBLE
...
Cycle 19: LOAD(IF-WAIT) | ADD(ID) | BUBBLE  | BUBBLE  | BUBBLE
Cycle 20: NEXT(IF)      | LOAD(ID)| ADD(EX) | BUBBLE  | BUBBLE
```

Note how ADD stays in ID stage along with the frozen LOAD. Both IF and ID are frozen together.

Data cache miss (for load/store in MEM stage) works differently:

1. **Freeze earlier stages**: PC, IF/ID, ID/EX all freeze
2. **Hold MEM stage**: The load instruction waits in MEM for data
3. **WB stage drains**: Last instruction completes
4. **Bubbles inserted upstream**: Since ID/EX is frozen, EX stage receives bubbles

The key difference: instruction cache miss stalls at IF; data cache miss stalls at MEM.

How combinational logic in execution units should handle invalid instructions

The ALU should compute regardless of validity — but its results must be discarded if the instruction is invalid. This design choice optimizes for:

- **Simpler control logic**: No conditional clock gating of functional units
- **Better timing**: Computation starts immediately without waiting for validity confirmation
- **Cleaner critical paths**: Control signals evaluated in parallel with computation

The execution unit structure should be:

```verilog
// ALU always computes
assign alu_result = (alu_op == ADD) ? operand_a + operand_b :
                    (alu_op == SUB) ? operand_a - operand_b :
                    // ... other operations

// Address calculation always happens
assign memory_address = base_register + immediate_offset;

// But writes are gated
assign register_write = alu_result;
assign register_write_enable = RegWrite && valid_bit && !exception;

assign memory_write_data = store_data;
assign memory_write_enable = MemWrite && valid_bit && !exception;
```

The crucial distinction: Functional units compute speculatively, but **state changes are guarded**. The register file and memory only accept writes when all validity conditions pass.

For your forwarding logic computing addresses when valid_in=0, this is actually **acceptable behavior for the computation itself**. What's not acceptable is using those computed addresses for memory writes. Your bug isn't that the address gets computed — it's that the write enable isn't properly gated.

Advanced optimization: Modern processors may use integrated clock gating (ICG) to disable functional units during invalid cycles for power savings: `enable = stage_valid && !stage_stalled`. But this is optional — correctness doesn't require it.

Standard implementations from working RISC-V CPUs on GitHub

Research examined multiple high-quality implementations with cache support:

zhaoyu-li/RISC-V_CPU (SJTU course project): Uses a "done" signal from cache that lasts one cycle. Pipeline registers and PC register are state machines that transition to "waiting state" during cache misses. github Control signals are buffered during stalls to prevent corruption. Branch execution moved to EX stage to handle stall timing properly.

Michaelvll/RISCV_CPU: Similar done-signal approach. Cache modified to handle uncached addresses. State machines hold current values until cache completes. Branch and jump signals buffered during stalls. GitHub

pietroglyph/pipelined-rv32i: Excellent SystemVerilog implementation with clear hazard unit. Key pattern for forwarding:

```systemverilog
// Forward from Memory stage if valid
if ((RegWriteM) && (RdM != 0) && (RdM == Rs1E))
   ForwardAE = 10;  // Forward from Memory
else if ((RegWriteW) && (RdW != 0) && (RdW == Rs1E))
   ForwardAE = 01;  // Forward from Writeback
else
   ForwardAE = 00;  // No forwarding
```

Notice **RegWrite is checked first** — this is the validity check preventing forwarding during bubbles.

For stalls: `lwStall = ResultSrcE[0] & ((Rs1D == RdE) | (Rs2D == RdE))`. When stall occurs: freeze PC and IF/ID (stall signals), flush ID/EX (set control to 0). University of Maryland D...

Common implementation patterns across all repositories:

1. **Implicit valid bits**: Control signals act as validity indicators. RegWrite=0 means "don't write" which is functionally equivalent to "invalid instruction." Wallawalla
2. **Enable-based pipeline registers**: Registers have enable inputs. When stalled, enable=0, so registers hold current values:

```verilog
always @(posedge clk) begin
    if (enable) stage_out <= stage_in;
    // else hold current value
end
```

3. **Flush via control zeroing**: Bubbles created by setting all control signals to 0, not by inserting explicit NOP instructions. Wallawalla ScienceDirect
4. **Forwarding checks RegWrite**: Every implementation's forwarding logic includes RegWrite as the first condition, preventing forwarding from bubbles. University of Maryland D...
5. **Priority to recent stages**: When both MEM and WB match source registers, MEM takes priority (more recent data).

Common bugs when adding caches to existing pipelines

The research uncovered several recurring bug patterns:

Bug 1: Write enables not gated during stalls (your bug). Symptoms: Garbage writes to 0x00000000 or random addresses during cache misses. Root cause: Memory write enable signal doesn't include stall or valid bit in its logic. Solution: `mem_write = instr_MemWrite && stage_valid && !stall_signal`.

Bug 2: Race condition between cache miss and stall assertion. The cache must detect miss and assert stall in the **same cycle**. Stack Overflow If stall takes an extra cycle to propagate, the next instruction may partially execute. Stack Exchange This requires careful attention to the stall signal critical path — it must reach PC update, all pipeline register enables, and all write enable gates within one cycle.

Bug 3: Instruction cache coherency (documented in MIT pdos/xv6-riscv Issue #67). When process A loads code to physical page P, then exits, and process B reuses that physical page, the instruction cache may contain stale code from process A. GitHub This caused real hardware to execute wrong instructions. Solution: Execute FENCE.I after loading code to invalidate instruction cache. github GitHub RISC-V makes this software's responsibility (unlike x86 which has hardware cache snooping).

Bug 4: Forwarding from stalled instructions. If forwarding logic doesn't check whether the source instruction is stalled or contains a bubble, it forwards garbage. Solution: Include RegWrite in forwarding condition — when instruction is a bubble, RegWrite=0, preventing forwarding. University of Maryland D...

Bug 5: Valid bits not initialized on reset. Cache valid bits must start at 0. If uninitialized, random tag comparisons may spuriously hit, returning garbage data. Wikipedia

Bug 6: Premature valid bit setting. Valid bit must be set only **after** the entire cache line fill completes, not when it starts. During the fill, the line should remain Invalid even though a fill buffer is working. Stack Overflow

Bug 7: Branch misprediction during cache miss. If a branch resolves while IF stage is stalled on cache miss, the flush signals may conflict with stall signals. Careful priority must be established: flush typically takes precedence over stall.

Warning sign checklist for your debugging:

- ✓ Writes to address 0x00000000: Indicates uninitialized register used during stall (your symptom!)
- Sporadic incorrect results: Suggests forwarding invalid data
- Failures only on cache-miss-heavy code: Indicates stall handling broken, not cache itself
- Different results on repeated runs: Uninitialized state being used
- Results change with memory timing: Race condition between cache ready and stall release

How forwarding logic should interact with valid bits

The **standard forwarding condition** from Patterson & Hennessy and confirmed in all examined implementations:

```verilog
// Forward A from EX/MEM stage
if ((EX_MEM_RegWrite == 1) &&     // Instruction writes register
    (EX_MEM_Rd != 0) &&           // Not writing to x0
    (EX_MEM_Rd == ID_EX_Rs1))     // Matches source
    ForwardA = 2'b10;  // Forward from EX/MEM

// Forward A from MEM/WB stage
else if ((MEM_WB_RegWrite == 1) &&
         (MEM_WB_Rd != 0) &&
         (MEM_WB_Rd == ID_EX_Rs1))
    ForwardA = 2'b01;  // Forward from MEM/WB

// Otherwise no forwarding
else
    ForwardA = 2'b00;  // Use register file
```

Critical point: RegWrite is checked **first**. This is the validity check. During bubbles, all control signals including RegWrite are 0, so the forwarding condition evaluates to false. **No forwarding occurs from bubbles**. U-aizu UCSD

Why forwarding must check RegWrite: Not all instructions write registers. Stores, branches, and NOPs have RegWrite=0. If forwarding only checked register number matches without verifying RegWrite, it would forward garbage from these non-writing instructions. University of Maryland D...

Order of operations: The forwarding unit is **purely combinational** and operates within a single cycle:

1. **Early in cycle**: Pipeline registers hold stable values (including RegWrite signals)
2. **Combinational evaluation**: Forwarding unit reads RegWrite, Rd, Rs from pipeline registers and computes ForwardA/ForwardB control signals
3. **Mux selection**: These control signals select which data source feeds the ALU
4. **ALU computation**: ALU operates on the selected (possibly forwarded) data
5. **Late in cycle**: Result latches into next pipeline register

The forwarding logic doesn't "compute addresses" — it **selects pre-computed values** via multiplexers. Your description of "forwarding logic computing before validity checks" suggests a misunderstanding. The forwarding logic should be:

```verilog
// Forwarding unit (combinational)
assign forward_data_a = (ForwardA == 2'b10) ? EX_MEM_alu_result :
                        (ForwardA == 2'b01) ? MEM_WB_write_data :
                        register_file_data_a;

// ALU uses forwarded data
assign alu_result = alu_op(forward_data_a, forward_data_b);
```

The forwarding muxes select between sources; the ALU then computes. Both happen combinationally in the same cycle. **Validity checking happens in the forwarding condition computation** (checking RegWrite), not as a separate step.

Your specific issue: You state "forwarding logic computes before validity checks." This suggests your forwarding unit isn't checking RegWrite. The fix:

```verilog
// WRONG: Missing RegWrite check
assign ForwardA = (EX_MEM_Rd == ID_EX_Rs1) ? 2'b10 : 2'b00;

// CORRECT: Include RegWrite check
assign ForwardA = (EX_MEM_RegWrite && (EX_MEM_Rd != 0) &&
                   (EX_MEM_Rd == ID_EX_Rs1)) ? 2'b10 : 2'b00;
```

When a bubble is in EX_MEM stage, RegWrite=0, so ForwardA remains 00 (no forwarding).

Timing and data flow during multi-cycle stalls

The critical timing sequence for your 19-cycle cache miss:

Cycle 0 (before miss): Normal operation, pipeline full of valid instructions.

Cycle 1 (miss detection):

- **Early**: IF stage sends address to instruction cache
- **Mid-cycle**: Cache tag comparison detects miss, asserts `cache_miss` signal **combinationally**
- **Mid-cycle**: Hazard detection unit receives miss signal, asserts `stall_IF` and `stall_ID`
- **Mid-cycle**: Stall signals propagate to PC write enable and IF/ID write enable, University of Maryland D... both go to 0
- **Mid-cycle**: Hazard unit asserts `flush_EX`, forcing ID/EX control signals to 0 (creating bubble)
- **Late**: Clock edge: PC holds current value, IF/ID holds current instruction, ID/EX latches all-zero controls University of Maryland D...

Cycles 2-19 (waiting for memory):

- PC frozen at same address that caused miss
- IF/ID frozen with same instruction
- ID/EX receives bubbles (zeros) each cycle because ID is frozen
- Bubbles propagate: EX→MEM→WB, exiting at cycle 4
- All stages contain bubbles after cycle 4
- DRAM access proceeds in parallel (controller fetching cache line)

Cycle 20 (data arrives):

- **Early**: DRAM returns data, cache line fill completes
- **Mid-cycle**: Cache asserts `cache_ready`, deasserts `cache_miss`
- **Mid-cycle**: Hazard unit deasserts stall signals
- **Mid-cycle**: PC write enable and IF/ID write enable return to 1
- **Late**: Clock edge: PC advances, IF/ID latches the instruction (that was waiting for 19 cycles), ID/EX receives valid control signals

Cycle 21+: Normal operation resumes.

Critical timing paths that must complete in one cycle:

1. **Cache miss detection**: Tag comparison → miss signal (combinational, ~2-3 gate delays)
2. **Stall propagation**: Miss signal → hazard unit → stall signals to PC/pipeline registers (combinational, ~3-5 gate delays)
3. **Write enable gating**: Stall signal → AND with RegWrite/MemWrite → write enable to register file/memory (combinational, ~1-2 gate delays)

If any of these paths is too slow, writes may occur during the first stall cycle before write enables are disabled. This could explain your 0x00000000 write — **the stall signal didn't reach the write enable gate in time**.

Solution: Ensure your hazard detection unit and write enable gating have minimal logic depth. Consider registering the stall signal (accepting one cycle latency) rather than having a long combinational path. For your cache integration:

```verilog
// Hazard detection must be fast
always @(*) begin
    stall_IF = cache_miss || dcache_miss;
    stall_ID = stall_IF;  // Propagate backward
    flush_EX = stall_ID;  // Insert bubble
end

// Write enables must include stall check
assign reg_write_enable = WB_RegWrite && !stall_WB;
assign mem_write_enable = MEM_MemWrite && !stall_MEM;
```

Data flow during forwarding with stalls: When a load stalls in MEM waiting for cache, and a dependent instruction is in EX:

- Load: `LW $1, 0($2)` in MEM stage, stalled on data cache miss
- Dependent: `ADD $3, $1, $4` in EX stage

The forwarding logic detects the dependency: `(MEM_WB_Rd == ID_EX_Rs1)`. But should it forward? **No** — because the load hasn't completed. The forwarding condition must include: `MEM_WB_RegWrite && !MEM_stalled`. Since the load is stalled, don't forward. The dependent instruction must also stall (or be converted to bubble) until load completes.

This is why **load-use hazards require stalls even with forwarding**: the forwarded data simply isn't available yet, and the validity checking (via RegWrite and stall signals) prevents premature forwarding.

Recommended fix for your specific issue

Based on your description, here's the systematic fix:

Step 1: Add explicit valid bit to each pipeline register (optional but recommended for debugging):

```verilog
// ID/EX pipeline register
always @(posedge clk) begin
    if (reset || flush_EX)
        ID_EX_valid <= 0;
    else if (stall_EX)
        ID_EX_valid <= ID_EX_valid;  // Hold
    else
        ID_EX_valid <= IF_ID_valid;
end
```

Step 2: Ensure control signals zero on bubble insertion:

```verilog
always @(posedge clk) begin
    if (reset || flush_EX) begin
        ID_EX_RegWrite <= 0;
        ID_EX_MemWrite <= 0;
        ID_EX_MemRead <= 0;
        // All control signals to 0
    end
    else if (!stall_EX) begin
        ID_EX_RegWrite <= IF_ID_RegWrite;
        // Normal operation
    end
end
```

Step 3: Gate all write enables with valid bit AND stall signal:

```verilog
// At register file
assign reg_write_en = WB_RegWrite && WB_valid && !pipeline_stalled;

// At data memory
assign mem_write_en = MEM_MemWrite && MEM_valid && !MEM_stalled;
```

Step 4: Fix forwarding to check RegWrite:

verilog

```
wire forward_from_mem = EX_MEM_RegWrite && (EX_MEM_Rd != 0) &&
                        (EX_MEM_Rd == ID_EX_Rs1);

wire forward_from_wb = MEM_WB_RegWrite && (MEM_WB_Rd != 0) &&
                       (MEM_WB_Rd == ID_EX_Rs1) && !forward_from_mem;

assign ForwardA = forward_from_mem ? 2'b10 :
                  forward_from_wb  ? 2'b01 : 2'b00;
```

Step 5: Verify in simulation:

- Add assertions: `assert property (@(posedge clk) MEM_stalled |-> !mem_write_en);`
- Monitor write address and enable signals during cache miss
- Confirm no writes occur during cycles 1-19 of your cache miss
- Check that valid bits properly clear and propagate

The garbage write to 0x00000000 should disappear once write enables are properly gated. The execution unit will still compute addresses (this is fine), but those computed values won't reach memory because the write enable will be 0. Stack Overflow