LLM Transpilation Bug Patterns for Targeted Retraining

Overview

This document provides a systematic categorization of LLM transpilation errors when converting x86 assembly to ARMv8. Each pattern includes detection methods, examples, and specific retraining strategies to address the root causes of these errors.

Quick Reference Guide

Category	Primary Issue	Detection Focus
Register State Management	Incorrect value tracking	Register usage patterns
Control Flow & Logic	Wrong execution paths	Branch targets and conditions
Critical Instruction Omission	Missing essential operations	Sequence completeness
Instruction-Level Semantics	Wrong instruction usage	Individual instruction validity
x86 Translation Artifacts	Non-idiomatic ARMv8 code	Complex vs. simple patterns

---

Category 1: Register State Management Errors

The most critical category where LLMs fail to track register values correctly, leading to data corruption.

1.1 Premature Update / Stale Read

Problem: Register updated before current value is used in the same iteration.

Detection Method: Look for increment/update instructions appearing before memory access that depends on the original value.

Bug Example (Problem 8):

```
arm

; PREDICTION (BUG)
add  w23, w23, #1          ; count = count + 1 (Premature update)
str  x8, [x0, w23, sxtw #3] ; array[count+1] <- Stores to wrong index

; GROUND TRUTH (CORRECT)
add  w25, w23, #1          ; w25 = count + 1 (Uses temp register)
str  x8, [x0, w23, sxtw #3] ; array[count]   <- Uses original count
mov  x23, x25              ; count = w25     <- Updates after use
```

Retraining Strategy: Emphasize "load/use, then update" patterns with temporary registers.

1.2 Register Clobbering / Aliasing

Problem: Live register value overwritten before it's no longer needed.

Detection Method: Trace variable live ranges; look for mid-calculation reassignments.

Bug Examples:

- Problem 9: SIMD sum accumulators overwritten by multiplication results
- Problem 19: Loop counter immediately overwritten with input value

Retraining Strategy: Train with functions having many local variables using distinct registers per logical value.

1.3 Function Return Value Propagation Failure

Problem: Critical return values (from malloc/realloc) not saved from x0 register.

Detection Method: Check for missing `mov dest_reg, x0` after function calls.

Bug Example (Problem 18):

```
arm

; PREDICTION (BUG)
bl   _realloc
; mov x21, x0 IS MISSING!
str w28, [x0, w24, sxtw #2] ; WRONG: uses volatile x0 instead of x21
```

Retraining Strategy: Pair every `bl <function>` with `mov <destination>, x0` examples.

Category 2: Control Flow and Logic Errors

Fundamental errors in program execution paths.

2.1 Incorrect Branch Targets

Problem: Branches jump to wrong labels, altering program logic.

Detection Method: Compare branch targets after comparisons and at basic block endings.

Bug Example (Problem 18):

```arm
; PREDICTION (BUG)
b.hi  LBB0_20 ; Jumps to loop block

; GROUND TRUTH (CORRECT)
b.hi  LBB0_19 ; Jumps to error-handling block
```

Retraining Strategy: Provide clear if/else and switch/case structures with distinct logic blocks.

2.2 Premature or Incorrect Function Returns

Problem: Early returns or wrong return value types.

Detection Method: Check `ret` instruction placement and x0 contents before return.

Bug Example (Problem 10): Function returns integer instead of allocated pointer.

Retraining Strategy: Focus on memory allocation functions that must return pointers.

Category 3: Critical Instruction Omission

Missing essential operations that break program functionality.

Core Problem: LLM "forgets" necessary instructions like initialization, calculations, or state updates.

Detection Method: Compare high-level operation sequences between source and prediction.

Common Examples:

- Problem 9: Missing SIMD vector reduction step
- Problem 18: Missing `mov x21, x0` after realloc

Retraining Strategy: Consistently show complete setup → loop → finalization phases in training data.

Category 4: Instruction-Level Semantic Errors

Correct instructions present but used incorrectly.

4.1 Invalid Addressing Modes

Problem: Syntactically or logically invalid memory address calculations.

Detection Method: Scrutinize `ldr` and `str` instructions for invalid register combinations.

Bug Example (Problem 15):

```
arm

; PREDICTION (BUG)
strb  w8, [x0, x0] ; Invalid: x0 used as both base and offset
```

Retraining Strategy: Extensive examples of valid ARMv8 addressing modes.

4.2 Incorrect Constants

Problem: Wrong immediate values in comparisons or operations.

Detection Method: Verify all immediate values against source logic.

Bug Example (Problem 20): Wrong ASCII values for character comparisons.

Retraining Strategy: Focus on switch statements and multi-branch conditionals.

4.3 Wrong Instruction Ordering

Problem: Correct instructions in wrong sequence.

Detection Method: Analyze dependent instruction sequences for logical flow.

Bug Example (Problem 7):

```
arm

; PREDICTION (BUG)
mov   w22, #0         ; max_level = 0
str  w22, [x20, x21, lsl #2] ; Stores 0

; GROUND TRUTH (CORRECT)
str  w22, [x20, x21, lsl #2] ; Stores correct value first
mov   w22, #0                ; Then resets for next round
```

Retraining Strategy: Emphasize "store then reset" and "calculate then store" patterns.

Category 5: x86 Translation Artifacts

Non-idiomatic ARMv8 code resulting from literal x86 translation.

Core Problem: LLM translates x86 form rather than semantic intent, creating unnecessarily complex ARMv8 code.

Detection Method: Look for convoluted arithmetic that could be simplified to single instructions.

Bug Example (Problem 15):

```
arm

; PREDICTION (x86-ism)
lsl   x23, x0, #32
asr x0, x23, #29

; GROUND TRUTH (Idiomatic ARMv8)
sbfiz x0, x0, #3, #32 ; Single instruction equivalent
```

Retraining Strategy: Create x86-to-ARMv8 pattern dictionary:

- `lea rax, [rdi+rcx*4]` → `add x0, x1, x2, lsl #2`
- x86 shift tricks → ARMv8 `lsl`, `sbfiz`
- x86 stack management → ARMv8 conventions

Implementation Recommendations

Priority Order for Retraining

1. **Register State Management** (highest impact on correctness)
2. **Control Flow Errors** (affects program logic)
3. **Critical Omissions** (breaks functionality)
4. **Instruction Semantics** (correctness issues)
5. **x86 Artifacts** (optimization and idiom improvement)

Training Data Requirements

- Functions with multiple local variables
- Clear setup/loop/finalization phases
- Memory allocation patterns
- Complex control flow with error handling
- ARMv8-idiomatic code examples

Validation Approach

For each category, create test cases that specifically trigger these error patterns, allowing systematic measurement of improvement after retraining.