x86 to ARMv8 Translation Pattern Analysis

Overview

This document analyzes specific problematic x86 patterns that an LLM transpiler attempted to mimic when translating to ARMv8, identifying the root causes of translation failures. By understanding these patterns, we can create targeted training data to improve the LLM's translation accuracy.

Translation Pattern Failures

Pattern 1: Complex LEA (Load Effective Address) Instructions

x86 Characteristics:

- The `lea` instruction performs complex arithmetic (`base + index*scale + displacement`) in a single step
- Extremely powerful for address calculations involving base registers, index registers, and scaled multipliers

ARMv8 Equivalent Approach:

- ARMv8 has flexible addressing modes (`[base, #offset]`, `[base + register]`) but with different syntax
- Correct approach often requires breaking calculations into separate ADD instructions followed by simple LDR/STR with register offsets

Translation Failure Analysis:

Problem ID	Symptom	Root Cause	Example
P63	Invalid Addressing Mode (Double Scaling)	LLM pre-calculated scaled offset then fed it into ARMv8 addressing mode that performs its own scaling	`str w0, [x2, x10]` where x10 already contains `i*4`, causing hardware to compute `base + (i*4)*4`
P87	Incorrect Pointer Setup	Failed complex addressing mode translation (`[rbp + rcx - 11]`)	`ldrsb w17, [x15]` with uninitialized x15 register

Correct ARMv8 Idiom:

```armv8
str w0, [x2, x9, lsl #2]  ; x9 holds raw index i, hardware does scaling
```

Pattern 2: x86 EFLAGS and Compound Conditional Logic

x86 Characteristics:

- Sequences of CMP and TEST instructions with results stored in EFLAGS register
- Subsequent conditional jumps (JNE, JA, etc.) or SETcc instructions act on combined flag state
- Natural support for compound conditions like `A && B` or `A || B`

ARMv8 Equivalent Approach:

- Use conditional execution or Conditional Compare (CCMP) instruction
- CCMP allows chaining conditions together but requires careful configuration

Translation Failure Analysis:

Problem ID	Symptom	Root Cause	Impact
P37	Misinterpretation of Algorithm's Goal	LLM created false dependency between independent checks `(i * C1) >= C2 && (i * C3) >= C4`	Broke `&&` logic by merging into single `cmp ... ccmp` sequence
P77	Misinterpretation of Algorithm's Goal	Failed to understand separate conditions (`power <= n` and `count < 100`)	Incorrectly merged independent loop continuation checks
P81	Misinterpretation of Algorithm's Goal	Jumbled complex flag-setting sequence from x86	Created nonsensical `cmp/cset/sub/ccmp` sequence

Correct ARMv8 Idiom:

```armv8
// For if (A && B):
cmp  x0, x1       ; check A
b.false L_fail
cmp  x2, x3       ; check B
b.false L_fail
```

Pattern 3: Non-Obvious Bit-Shifting for Arithmetic

x86 Characteristics:

- Uses sequences of shifts and adds/subtracts for multiplication/division by constants
- Example: `(val << 32) >> 29` to multiply by 8
- Optimization technique often faster than `imul`/`idiv`

ARMv8 Equivalent Approach:

- Dedicated instructions like LSL (Logical Shift Left) for simple multiplication

- SBFX/SBFIZ (Signed Bitfield Extract/Insert) for complex cases

Translation Failure Analysis:

Problem ID	Symptom	Root Cause	Example
P15	Literal Translation Artifacts	Translated method instead of intent	`lsl x23, x0, #32` followed by `asr x0, x23, #29` instead of single `lsl x0, x0, #3`

Correct ARMv8 Idiom:

```armv8
lsl x0, x0, #3     ; Clean multiplication by 8
; OR
sbfiz x0, x0, #3, #29  ; More complex bit manipulation
```

Pattern 4: Complex SIMD Instructions

x86 Characteristics:

- Powerful single instructions like `pshufb` (Packed Shuffle Bytes)
- Can reorder, duplicate, or zero-out bytes based on lookup mask
- Used for table lookups and complex data transformations

ARMv8 Equivalent Approach:

- ARM NEON TBL (Table Lookup) instruction provides equivalent functionality

Translation Failure Analysis:

Problem ID	Symptom	Root Cause	Impact
P51	Code Hallucination	LLM didn't know `pshufb` → `tbl` mapping	Generated massive block of 500+ nonsensical `saddw` instructions
P28	Misinterpretation of Algorithm's Goal	Tried to mirror x86 unrolled loop structure	Created oversized 64-byte unrolled loop, breaking register management

Correct ARMv8 Idiom:

```armv8
tbl v0.16b, {v1.16b}, v2.16b  ; Direct equivalent to pshufb
```

Pattern 5: Implicit "Use Then Update" Memory Operations

x86 Characteristics:

- Some instructions can read from memory and write back, or use pointer and update it
- Appears as single atomic operation: "load array[j] and prepare for j-1 next"

ARMv8 Equivalent Approach:

- Must explicitly serialize into separate load and update instructions
- Critical to maintain correct order: use current value, then update for next iteration

Translation Failure Analysis:

Problem ID	Symptom	Root Cause	Impact
P71, P47	Incorrect Loop Pointer/Index Management	Wrong instruction ordering in "use then update" translation	Performed `sub j, j, #1` before `ldr reg, [base, j]`, reading from wrong index

Correct ARMv8 Idiom:

```armv8
ldr w0, [x1, x2, lsl #2]  ; Use current index value
sub x2, x2, #1            ; Update index for next iteration
```

Key Insights for LLM Training

Root Cause Categories

1. **Literal Translation Syndrome**: LLM translates instruction-by-instruction instead of understanding higher-level intent
2. **Complex Instruction Panic**: When encountering unknown complex instructions, LLM enters failure mode and hallucinates code
3. **Flag Logic Misunderstanding**: Fails to properly map x86 EFLAGS-based conditional logic to ARMv8 patterns
4. **Ordering Dependency Failures**: Doesn't understand critical instruction ordering requirements in multi-step translations
5. **Scaling Mode Confusion**: Misunderstands when ARMv8 hardware performs scaling vs. when software must pre-scale

Training Data Recommendations

Focus on creating examples that demonstrate:

- Intent-based translation rather than literal instruction mapping
- Proper ARMv8 idioms for common x86 patterns
- Correct conditional logic chaining techniques
- Instruction ordering dependencies in memory operations
- Hardware vs. software scaling distinctions

This analysis provides the foundation for developing targeted training data to address these specific translation failure patterns.