Assembly Code Analysis: C to ARMv8 Mapping and Comparison

1. C Code to Assembly Mapping

C Code Structure:

c

```c
void func0(float numbers[], int size, float out[2]) {
    float min_diff = FLT_MAX;        // Line 6
    int i, j;                        // Line 7

    out[0] = numbers[0];             // Line 9
    out[1] = numbers[1];             // Line 10

    for (i = 0; i < size; i++) {     // Line 12
        for (j = i + 1; j < size; j++) { // Line 13
            float diff = fabs(numbers[i] - numbers[j]); // Line 14
            if (diff < min_diff) {   // Line 15
                min_diff = diff;     // Line 16
                out[0] = numbers[i]; // Line 17
                out[1] = numbers[j]; // Line 18
            }
        }
    }

    if (out[0] > out[1]) {           // Line 23
        float temp = out[0];         // Line 24
        out[0] = out[1];             // Line 25
        out[1] = temp;               // Line 26
    }
}
```

Register Mapping (ARMv8):

Ground Truth (gd) Register Usage:

- `x0`: numbers[] array pointer
- `x1`: size parameter (w1 for 32-bit)
- `x2`: out[] array pointer
- `s0`: out[1] / numbers[1] (float)
- `s1`: out[0] / numbers[0] (float)
- `s2`: min_diff (initially 2139095039 = FLT_MAX)

- `s3`: numbers[i] (current element)
- `s4`: difference calculation result
- `x8`: size (copied from w1)
- `x9`: pointer arithmetic for inner loop
- `x10`: outer loop counter i
- `x11`: inner loop base index
- `x12`: inner loop counter j

Predicted (pred) Register Usage:

- Similar base mapping but with key differences in loop structure
- `x9`: stores constant 2139095040 (incorrect FLT_MAX)
- Different pointer arithmetic approach

Line-by-Line Mapping:

Lines 9-10 (Initialization):

```assembly
; Both gd and pred:
ldr s1, [x0]        ; out[0] = numbers[0]
str s1, [x2]
ldr s0, [x0, #4]    ; out[1] = numbers[1]
str s0, [x2, #4]
```

Line 12 (Outer loop condition):

```assembly
; Both gd and pred:
cmp w1, #1          ; if (size < 1)
b.lt LBB0_8         ; goto end
```

Lines 13-18 (Inner loops and comparison):

- Complex nested loop structure with different implementations
- Key differences in pointer arithmetic and loop counters

2. Vertical Comparison: gd vs pred

Major Differences:

Difference 1: FLT_MAX Constant

```assembly
; gd (CORRECT):
mov w10, #2139095039    ; FLT_MAX = 0x7F7FFFFF
fmov s2, w10

; pred (INCORRECT):
mov w9, #2139095040     ; 0x7F800000 (This is +INFINITY, not FLT_MAX!)
```

Difference 2: Loop Structure

```assembly
; gd - More complex but correct nested loop:
LBB0_3:                 ; Outer loop header
    add x12, x11, #1    ; j = i + 1
    cmp x12, x8         ; compare j with size
    b.hs LBB0_2         ; if j >= size, continue outer loop

; pred - Simplified but potentially incorrect:
LBB0_3:                 ; Different loop structure
    add x13, x11, #1    ; Similar but different register usage
    cmp x13, x8
    b.hs LBB0_2
```

Difference 3: Pointer Arithmetic

```assembly
; gd:
add x9, x0, #4          ; Base pointer + 4
; Later: ldr s4, [x14]   ; Load from calculated address

; pred:
mov x12, x0             ; Different base pointer handling
; Later: ldr s3, [x15]   ; Different addressing
```

Difference 4: Final Comparison Logic

```assembly
; gd (CORRECT):
fcmp s1, s0             ; Compare out[0] with out[1]
b.le LBB0_10            ; if out[0] <= out[1], skip swap
str s0, [x2]            ; out[0] = out[1]
str s1, [x2, #4]        ; out[1] = temp

; pred (INCORRECT):
fcmp s1, s0             ; Same comparison
b.pl LBB0_10            ; WRONG BRANCH: b.pl means "branch if greater"
stp s0, s1, [x2]        ; Store pair in wrong order
```

3. Critical Logical Errors in pred

Error 1: Incorrect FLT_MAX Value

Problem: `mov w9, #2139095040` sets min_diff to +INFINITY (0x7F800000) instead of FLT_MAX (0x7F7FFFFF)

Impact: The algorithm will never find a difference smaller than +INFINITY, so it will always keep the initial values numbers[0] and numbers[1], regardless of whether there are closer pairs.

Root Cause: Likely a constant folding error during translation from x86 to ARM, where the translator confused IEEE 754 representations.

Error 2: Wrong Branch Condition in Final Swap

Problem: Uses `b.pl` (branch if positive or zero) instead of `b.le` (branch if less or equal)

C Code Logic:

```c
if (out[0] > out[1]) {  // Swap if out[0] is greater
    // swap logic
}
```

Correct Assembly (gd):

```assembly
fcmp s1, s0      ; Compare out[0] (s1) with out[1] (s0)
b.le LBB0_10     ; Skip swap if out[0] <= out[1]
```

Incorrect Assembly (pred):

```assembly
fcmp s1, s0      ; Same comparison
b.pl LBB0_10     ; WRONG: Skip swap if out[0] >= out[1] (includes equality case incorr
```

Impact: When out[0] == out[1], the code will incorrectly swap them, potentially changing the order unnecessarily.

Error 3: Incorrect Store Order in Swap

Problem: `stp s0, s1, [x2]` stores s0 (out[1]) first, then s1 (out[0])

Expected: Should store the smaller value first, larger value second **Actual:** Stores in reverse order due to the stp instruction semantics

4. Connection to x86 Code and O2 Optimization

x86 Original Patterns:

The x86 code shows:

```assembly
movss LCPI0_0(%rip), %xmm3    ; Load FLT_MAX constant
movaps LCPI0_1(%rip), %xmm2   ; Load NaN mask for abs operation
```

Translation Errors:

1. **Constant Pool Translation:** The x86 code properly loads FLT_MAX from a literal pool (LCPI0_0), but the ARM translator incorrectly converted this constant.
2. **Branch Logic Translation:** x86 uses `jbe` (jump if below or equal) for the final comparison, which should translate to ARM's `b.le`, but pred uses `b.pl`.
3. **SIMD Register Handling:** x86 uses XMM registers with specific semantics that weren't properly translated to ARM's scalar floating-point registers.

O2 Optimization Impact:

The O2 optimization likely:

- Performed aggressive constant folding that introduced the FLT_MAX error
- Optimized branch patterns in a way that confused the translator
- Reordered instructions for performance, leading to incorrect register usage patterns

Conclusion: The pred translation contains critical errors that would cause the algorithm to fail in finding the closest pair of numbers, instead always returning the first two elements of the array, potentially in the

wrong order.