Assembly Code Analysis: C to ARM Translation

1. C Code to Ground Truth (gd) ARM Assembly Mapping

C Code Function:

```c
void func0(int *numbers, int size, int *result) {
    int sum = 0, product = 1;
    for (int i = 0; i < size; i++) {
        sum += numbers[i];
        product *= numbers[i];
    }
    result[0] = sum;
    result[1] = product;
}
```

Register Mapping in Ground Truth (gd):

- **x0**: `int *numbers` (array pointer)
- **w1**: `int size` (array size)
- **x2**: `int *result` (result array pointer)
- **w10**: `sum` variable
- **w11**: `product` variable
- **x8**: loop counter/size copy
- **x9**: vectorized loop counter
- **x10-x12**: temporary registers for addressing

Line-by-Line C to Assembly Mapping:

Initial Setup:

```c
int sum = 0, product = 1;
```

Ground Truth:

```assembly
mov w10, #0       ; sum = 0
mov w11, #1       ; product = 1
```

Loop Condition Check:

c

```
for (int i = 0; i < size; i++)
```

Ground Truth:

assembly

```
cmp w1, #1       ; compare size with 1
b.lt LBB0_3      ; if size < 1, skip to end
```

Vectorized Loop (O2 optimization for size >= 16):

c

```
sum += numbers[i];
product *= numbers[i];
```

Ground Truth (vectorized):

assembly

```
; Process 16 elements at once using SIMD
ldp q16, q17, [x10, #-32]  ; load 8 integers
ldp q18, q19, [x10], #64   ; load 8 more integers
add.4s v4, v16, v4         ; sum accumulation
mul.4s v0, v16, v0         ; product accumulation
```

Scalar Loop (remainder elements):

assembly

```
LBB0_8:
ldr w9, [x12], #4     ; load numbers[i]
add w10, w9, w10      ; sum += numbers[i]
mul w11, w9, w11      ; product *= numbers[i]
subs x8, x8, #1       ; i++, check condition
b.ne LBB0_8           ; continue if i < size
```

Result Storage:

```c
result[0] = sum;
result[1] = product;
```

Ground Truth:

```assembly
stp w10, w11, [x2]     ; store sum and product
```

2. Vertical Comparison: Ground Truth vs Predicted

Key Differences Found:

Location	Ground Truth (gd)	Predicted (pred)	Issue
LBB0_2 Init	`mov w10, #0`<br>`mov w11, #1`	`mov w11, #0`<br>`mov w10, #1`	**SWAPPED** sum/product initialization
LBB0_3 Early Exit	`mov w10, #0`<br>`mov w11, #1`<br>`b LBB0_9`	`mov w11, #0`<br>`mov w10, #1`<br>`stp w10, w11, [x2]`<br>`ret`	**SWAPPED** + different control flow
Vector Init	`movi.4s v0, #1` (product)<br>`movi.2d v4, #0` (sum)	`movi.4s v1, #1` (product)<br>`movi.2d v0, #0` (sum)	Different vector register assignment
Vector Ops	`add.4s v4, v16, v4` (sum)<br>`mul.4s v0, v16, v0` (product)	`add.4s v0, v5, v0` (sum)<br>`mul.4s v1, v5, v1` (product)	**INCORRECT** vector operations
Scalar Loop	`add w10, w9, w10` (sum)<br>`mul w11, w9, w11` (product)	`add w11, w9, w11` (sum)<br>`mul w10, w9, w10` (product)	**SWAPPED** scalar operations
Final Store	`stp w10, w11, [x2]`	`stp w11, w10, [x2]`	**SWAPPED** final storage

3. Error Analysis and Root Causes

Primary Error: Register Role Confusion

The predicted code consistently swaps the roles of `w10` and `w11`:

- **Ground Truth**: `w10` = sum, `w11` = product
- **Predicted**: `w10` = product, `w11` = sum

Critical Issues in Predicted Code:

Issue 1: Initialization Swap

assembly

```
; CORRECT (gd):
mov w10, #0    ; sum = 0
mov w11, #1    ; product = 1

; INCORRECT (pred):
mov w11, #0    ; sum = 0 (but w11 should be product!)
mov w10, #1    ; product = 1 (but w10 should be sum!)
```

Issue 2: Vector Operation Corruption

assembly

```
; CORRECT (gd):
mul.4s v0, v16, v0    ; product vector multiply
mul.4s v1, v17, v1    ; product vector multiply
mul.4s v2, v18, v2    ; product vector multiply
mul.4s v3, v19, v3    ; product vector multiply

; INCORRECT (pred):
mul.4s v1, v5, v1     ; product multiply
mul.4s v0, v6, v0     ; ERROR: multiplying sum vector!
mul.4s v2, v7, v2     ; product multiply
mul.4s v3, v16, v3    ; product multiply
```

Issue 3: Vector Reduction Error

assembly

```
; CORRECT (gd):
mul.4s v0, v1, v0     ; combine product vectors
; ... proper product reduction

; INCORRECT (pred):
mul.4s v0, v1, v0     ; ERROR: mixing sum and product vectors!
```

Connection to O2 Optimization:

The errors stem from misunderstanding the **vectorization optimization** in O2:

1. **Vector Register Assignment**: The compiler uses separate vector registers for sum and product accumulation, but the predicted code confuses which vectors correspond to which operation.

2. **SIMD Lane Management**: O2 optimization processes multiple array elements simultaneously using SIMD instructions. The predicted code incorrectly mixes sum and product operations within the same vector lanes.
3. **Reduction Phase**: After vectorized processing, the compiler must reduce the vector results to scalars. The predicted code corrupts this by applying multiplication operations to sum vectors.

Logical Consequences:

1. **Incorrect Sum**: Sum gets multiplied instead of added during vector processing
2. **Incorrect Product**: Product gets corrupted by sum vector operations
3. **Wrong Final Values**: Both result[0] and result[1] will contain incorrect values

The root cause is a **systematic register role confusion** that propagates through the entire function, compounded by misunderstanding how O2's vectorization optimization separates and processes sum vs product operations.