Assembly Code Analysis: C to ARM Mapping and Comparison

1. C Code to Assembly Mapping

C Code with Line Numbers

```c
1:  int func0(const char *str, const char *substring) {
2:      int out = 0;
3:      int str_len = strlen(str);
4:      int sub_len = strlen(substring);
5:      if (str_len == 0) return 0;
6:      for (int i = 0; i <= str_len - sub_len; i++) {
7:          if (strncmp(&str[i], substring, sub_len) == 0)
8:              out++;
9:      }
10:     return out;
11: }
```

Register Usage in GD (Ground Truth) ARM Assembly

C Variable	ARM Register	Purpose
`str` (param)	`x0` → `x20`	First parameter, saved to x20
`substring` (param)	`x1` → `x19`	Second parameter, saved to x19
`str_len`	`x23`	Length of main string
`sub_len`	`x0` → `x22`	Length of substring (sxtw x22, w0)
`out`	`w21`	Output counter
`i` (loop counter)	Implicit in `x20` increment	String pointer advancement

C Code to GD ARM Assembly Mapping

Lines 1-2: Function prologue and initialization

```assembly
_func0:
    stp x24, x23, [sp, #-64]!    ; Save registers
    stp x22, x21, [sp, #16]
    stp x20, x19, [sp, #32]
    stp x29, x30, [sp, #48]
    mov x19, x1                   ; substring → x19
    mov x20, x0                   ; str → x20
```

Line 3: `int str_len = strlen(str);`

assembly

```
    bl  _strlen                   ; Call strlen(str)
    mov x23, x0                   ; str_len = strlen result
```

Line 4: `int sub_len = strlen(substring);`

assembly

```
    mov x0, x19                   ; Load substring
    bl  _strlen                   ; Call strlen(substring)
    mov w21, #0                   ; Initialize out = 0
```

Line 5: `if (str_len == 0) return 0;`

assembly

```
    cbz w23, LBB0_4              ; If str_len == 0, jump to return
```

Line 6: Loop setup `for (int i = 0; i <= str_len - sub_len; i++)`

assembly

```
    cmp w23, w0                   ; Compare str_len with sub_len
    b.lt LBB0_4                   ; If str_len < sub_len, exit
    sxtw x22, w0                  ; sub_len (sign extend to 64-bit)
    sub w8, w23, w0               ; w8 = str_len - sub_len
    add w23, w8, #1               ; Loop counter = (str_len - sub_len) + 1
```

Lines 7-8: Loop body with strncmp

assembly

```
LBB0_3:
    mov x0, x20                   ; &str[i]
    mov x1, x19                   ; substring
    mov x2, x22                   ; sub_len
    bl  _strncmp                  ; Call strncmp
    cmp w0, #0                    ; Compare result with 0
    cinc w21, w21, eq             ; If equal, increment out
    add x20, x20, #1              ; i++ (advance string pointer)
    subs x23, x23, #1             ; Decrement loop counter
    b.ne LBB0_3                   ; Continue if not zero
```

2. Vertical Comparison: GD vs PRED

Key Differences Found

Line	GD (Ground Truth)	PRED (Prediction)	Difference
31	`mov x23, x0`	`mov x22, x0`	**CRITICAL**: Different register for str_len
36	`cbz w23, LBB0_4`	`cbz w22, LBB0_4`	Uses wrong register for zero check
38	`cmp w23, w0`	`cmp w22, w0`	Uses wrong register for comparison
41	`mov w21, #0`	`mov x23, x0`	**CRITICAL**: Wrong assignment
42	`sxtw x22, w0`	`mov w21, #0`	**CRITICAL**: Order swapped
43	`sub w8, w23, w0`	`sxtw x22, w23`	**CRITICAL**: Wrong operand
44	`add w23, w8, #1`	`sub w8, w22, w23`	**CRITICAL**: Wrong operands
45	-	`add w23, w8, #1`	Extra instruction

Detailed Analysis of Critical Differences

Difference 1: Register Assignment (Lines 31, 36, 38)

GD:

assembly

```
mov x23, x0         ; str_len → x23
cbz w23, LBB0_4     ; Check if str_len == 0
cmp w23, w0         ; Compare str_len with sub_len
```

PRED:

assembly

```
mov x22, x0         ; str_len → x22 (WRONG!)
cbz w22, LBB0_4     ; Check if str_len == 0 (using wrong register)
cmp w22, w0         ; Compare str_len with sub_len (using wrong register)
```

Difference 2: Variable Initialization Order (Lines 41-44)

GD:

assembly

```
mov w21, #0         ; out = 0
sxtw x22, w0        ; sub_len = w0 (sign extended)
sub w8, w23, w0     ; w8 = str_len - sub_len
add w23, w8, #1     ; loop_count = (str_len - sub_len) + 1
```

PRED:

```assembly
mov x23, x0        ; WRONG: x23 = sub_len (should be str_len)
mov w21, #0        ; out = 0 (correct but wrong order)
sxtw x22, w23      ; WRONG: x22 = x23 (which is sub_len, not w0)
sub w8, w22, w23   ; WRONG: w8 = sub_len - sub_len = 0
add w23, w8, #1    ; WRONG: loop_count = 0 + 1 = 1
```

3. Logical Errors and Root Cause Analysis

Error 1: Register Confusion

Problem: PRED confused the registers for `str_len` and `sub_len`.

- **Correct**: `str_len` should be in `x23`, `sub_len` should be in `x22`
- **PRED Error**: `str_len` goes to `x22`, `sub_len` goes to `x23`

Error 2: Loop Bounds Calculation

Problem: Due to register confusion, the loop bounds calculation becomes:

```
sub w8, w22, w23  ; w8 = sub_len - str_len (WRONG!)
```

Instead of:

```
sub w8, w23, w0   ; w8 = str_len - sub_len (CORRECT)
```

Error 3: Impact on Algorithm

This causes the function to:

1. **Wrong zero check**: Checks `sub_len == 0` instead of `str_len == 0`
2. **Wrong comparison**: Compares `sub_len < sub_len` instead of `str_len < sub_len`
3. **Wrong loop count**: Calculates `(sub_len - str_len) + 1` instead of `(str_len - sub_len) + 1`

Connection to x86 Code and O2 Optimization

Analysis of x86 Source Pattern

Looking at the x86 assembly (input.txt), the pattern is:

```assembly
movq %rax, %r15      ; str_len → r15
movslq %eax, %r13    ; sub_len → r13 (sign extended)
subl %r13d, %r15d    ; r15 = str_len - sub_len
```

Translation Error Root Cause

The error appears to stem from:

1. **Register Allocation Confusion**: The translator incorrectly mapped the x86 registers to ARM registers
2. **Instruction Reordering**: O2 optimization in the original x86 code may have reordered instructions in a way that confused the translator
3. **Sign Extension Misinterpretation**: The `movslq %eax, %r13` instruction was incorrectly translated, leading to the wrong source register being used

Specific O2 Optimization Impact

The x86 code shows aggressive register reuse and instruction reordering typical of O2 optimization:

- `%rax` is reused for both strlen results
- The subtraction `subl %r13d, %r15d` modifies the str_len register directly
- This optimization pattern may have confused the translator about which register contains which value

Summary

The PRED translation fails because it:

1. Swaps the registers for `str_len` and `sub_len`
2. This leads to incorrect loop bounds calculation
3. The algorithm becomes logically incorrect, potentially causing buffer overruns or incorrect counts
4. The error likely stems from misinterpreting the optimized x86 register usage patterns