C Code to Assembly Mapping and Comparison Analysis

1. C Code to Assembly Mapping

Key Variables and Their Register Mappings

Ground Truth (gd) ARM Assembly:

- `numbers` (input parameter): `x0`
- `count[10]` array: `[sp, #16]` to `[sp, #56]` (40 bytes)
- `current[6]` buffer: `[sp, #10]` to `[sp, #15]`
- `index` (output position): `w16`
- `i` (inner loop counter): `w5` (parsing), `x8` (output loop)
- `j` (count loop): `w13` (output)
- `k` (string copy): inner loop variable
- `numto` array: `l___const.func0.numto` (pointer table)

Predicted (pred) ARM Assembly:

- `numbers` (input parameter): `x0`
- `count[10]` array: `[sp, #32]` to `[sp, #72]` (40 bytes)
- `current[6]` buffer: `[sp, #22]` to `[sp, #27]`
- `index` (output position): `w8`
- Similar loop variables with different registers

C Code Line-by-Line Mapping

Function Setup and Variable Initialization

```c
int count[10] = {0};
const char* numto[10] = {"zero", "one", "two", ...};
int index, i, j, k;
static char out[1000];
char current[6];
```

Ground Truth (gd):

```assembly
; Stack allocation and initialization
sub sp, sp, #80
str xzr, [sp, #48]          ; Clear some stack
movi.2d v0, #0              ; Zero vector
stp q0, q0, [sp, #16]       ; Initialize count[10] = {0}
```

Predicted (pred):

```assembly
; Stack allocation and initialization
sub sp, sp, #112            ; Different stack size!
str xzr, [sp, #64]
movi.2d v0, #0
stp q0, q0, [sp, #32]       ; count[10] at different offset
```

Input Parsing Loop

```c
if (*numbers) {
    do {
        for (i = 0; numbers[i] != ' ' && numbers[i] != '\0'; ++i) {
            current[i] = numbers[i];
        }
        current[i] = '\0';
        // ... string comparison logic
        numbers += i + 1;
    } while (numbers[-1]);
}
```

Ground Truth (gd):

```assembly
LBB0_2:                         ; Main parsing loop
    mov x5, #0                  ; i = 0
LBB0_3:                         ; Character extraction loop
    ldrb w6, [x0, x5]           ; Load numbers[i]
    orr w7, w6, #0x20           ; Convert to lowercase
    cmp w7, #32                 ; Compare with space
    b.eq LBB0_5                 ; Break if space/null
    strb w6, [x9, x5]           ; current[i] = numbers[i]
    add x5, x5, #1              ; i++
    b LBB0_3
```

Predicted (pred):

```assembly
LBB0_2:                         ; Main parsing loop
    mov x4, #0                  ; i = 0 (different register)
LBB0_3:                         ; Character extraction loop
    ldrb w5, [x0, x4]           ; Load numbers[i]
    orr w5, w5, #0x20           ; Convert to lowercase
    cmp w5, #32                 ; Compare with space
    b.eq LBB0_5                 ; Break if space/null
    add x5, sp, #22             ; Calculate current buffer address
    strb w5, [x5, x4]           ; current[i] = numbers[i]
    add x4, x4, #1              ; i++
    b LBB0_3
```

String Comparison Logic

```c
for (j = 0; j < 10; ++j) {
    if (strcmp(current, numto[j]) == 0) {
        count[j]++;
        break;
    }
}
```

Both assemblies implement this as a series of hardcoded comparisons using immediate values representing the first 4 bytes of each number word.

Output Generation Loop

c

```c
for (i = 0; i < 10; ++i) {
    for (j = 0; j < count[i]; ++j) {
        for (k = 0; numto[i][k] != '\0'; ++k, ++index) {
            out[index] = numto[i][k];
        }
        out[index++] = ' ';
    }
}
```

Ground Truth (gd):

assembly

```assembly
LBB0_29:                    ; Main output loop (i)
    ldr w12, [x11, x8, lsl #2]  ; Load count[i]
    cmp w12, #1
    b.lt LBB0_28            ; Skip if count[i] < 1

LBB0_32:                    ; Count loop (j)
    ldrb w16, [x14]         ; Load character from numto[i]
    cbz w16, LBB0_31        ; If null, add space

LBB0_34:                    ; Character copy loop (k)
    strb w16, [x1], #1      ; Store character and increment
    ldrb w16, [x2], #1      ; Load next character
    add w17, w17, #1        ; Increment index
    cbnz w16, LBB0_34       ; Continue if not null
```

Predicted (pred):

```assembly
LBB0_30:                    ; Main output loop (i)
    ldr w13, [x10, x9, lsl #2]  ; Load count[i]
    cmp w13, #1
    b.lt LBB0_29            ; Skip if count[i] < 1

LBB0_33:                    ; Count loop (j)
    ldrb w16, [x15]         ; Load character from numto[i]
    cbz w16, LBB0_32        ; If null, add space

LBB0_35:                    ; Character copy loop (k)
    strb w16, [x0, x17]     ; Store character
    ldrb w16, [x15], #1     ; Load next character
    add x17, x17, #1        ; Increment index
    cbnz w16, LBB0_35       ; Continue if not null
```

2. Vertical Comparison of gd vs pred

Major Differences:

A. Stack Layout Differences

Aspect	Ground Truth (gd)	Predicted (pred)
Stack size	80 bytes	112 bytes
count[10] location	[sp, #16]	[sp, #32]
current[6] location	[sp, #10]	[sp, #22]
Stack canary offset	[x29, #-8]	[x29, #-24]

B. Register Usage Differences

Variable	Ground Truth (gd)	Predicted (pred)
Loop counter i	w5, x5	w4, x4
Output index	w16	w8
Character temp	w6	w5
Buffer address calc	Direct offset	Add instruction

C. Critical Logic Differences

1. String Comparison Constants: Ground Truth uses correct hardcoded values:

```assembly
mov w8, #25971              ; "zero" representation
movk w8, #25974, lsl #16
```

Predicted uses different/incorrect constants:

```assembly
mov w8, #25978              ; Different value!
movk w8, #28530, lsl #16
```

2. Buffer Address Calculation: Ground Truth calculates buffer address once:

```assembly
add x9, sp, #10             ; current buffer base
strb w6, [x9, x5]          ; Direct indexed store
```

Predicted recalculates each time:

```assembly
add x5, sp, #22            ; Recalculate each iteration
strb w5, [x5, x4]          ; Store with calculated address
```

3. Output Index Management: Ground Truth maintains index in w16 consistently. Predicted uses w8 for index, which may conflict with other uses.

3. Error Analysis and Root Causes

Primary Errors in Predicted Assembly:

Error 1: Incorrect String Comparison Constants

Location: String matching section (LBB0_6 to LBB0_14) **C Code Reference:** `strcmp(current, numto[j])` **Issue:** The hardcoded 32-bit constants used for string comparison don't match the actual string values.

Ground Truth: Uses correct little-endian representations of "zero", "one", etc. **Predicted:** Uses incorrect constants, leading to failed matches.

Error 2: Inefficient Buffer Address Calculation

Location: Character extraction loop (LBB0_3) **C Code Reference:** `current[i] = numbers[i]` **Issue:** Recalculates buffer base address in each iteration instead of using indexed addressing.

Impact: Performance degradation but functionally correct.

Error 3: Register Allocation Conflicts

Location: Throughout the function **C Code Reference:** Various loop variables and temporaries **Issue:** Suboptimal register allocation may cause unnecessary register pressure.

Connection to x86 Translation and Optimization:

x86 to ARM Translation Issues:

1. **Immediate Value Encoding:** x86 uses different immediate value representations than ARM. The translator may have incorrectly converted the string comparison constants.
2. **Addressing Mode Differences:** x86's complex addressing modes (e.g., `[base + index + offset]`) don't directly map to ARM's simpler modes, leading to extra address calculations.
3. **Register Set Differences:** x86's smaller register set may have influenced register allocation decisions that don't optimize well for ARM's larger register file.

O2 Optimization Effects:

1. **Constant Propagation:** The compiler tried to optimize string comparisons by hardcoding immediate values, but the translation got the values wrong.
2. **Loop Optimization:** The compiler may have tried to optimize the nested loops but created suboptimal code due to incorrect analysis during translation.
3. **Register Allocation:** O2 optimization attempted aggressive register reuse, but the translation may have introduced conflicts or suboptimal assignments.

Functional Impact:

The most critical error is **Error 1** - incorrect string comparison constants. This will cause the function to fail to recognize number words correctly, leading to incorrect counting and output generation. The other errors primarily affect performance but maintain functional correctness.