Inline Assembly in C Worksheet

Learning from Common Mistakes

Table of Contents

Part 1: Basic Register Exchange

Version 1.0 - Simple Exchange (WORKS)

```
#include <stdio.h>

int main() {
    int var1 = 10;
    int var2 = 20;

    printf("Before exchange:\n");
    printf("var1 = %d\n", var1);
    printf("var2 = %d\n", var2);

    __asm__ __volatile__ (
        ".intel_syntax noprefix\n\t"
        "mov eax, %0\n\t"
        "mov ebx, %1\n\t"
        "xchg eax, ebx\n\t"
        "mov %0, eax\n\t"
        "mov %1, ebx\n\t"
        ".att_syntax prefix\n\t"
        : "+r" (var1), "+r" (var2)
        :
```

```
        : "eax", "ebx"
    );

    printf("\nAfter exchange:\n");
    printf("var1 = %d\n", var1);
    printf("var2 = %d\n", var2);

    return 0;
}
```

Key Concepts:

- `__asm__ __volatile__`: Inline assembly block
- `.intel_syntax noprefix`: Switch to Intel syntax
- `%0`, `%1`: Operand placeholders
- `"+r"`: Read-write register constraint
- Clobber list: `"eax", "ebx"` - tells compiler these registers are modified

How operand binding works:

```
: "+r" (var1), "+r" (var2)
  ^^^^        ^^^^
  %0          %1
```

The compiler:

1. Allocates registers for var1 and var2
2. Substitutes %0 with the register holding var1
3. Substitutes %1 with the register holding var2

Part 2: Input-Only Operands

Version 2.0 - Multiplication Example (WORKS)

```
#include <stdio.h>

int main() {
    int num1 = 5;
    int num2 = 7;
    int result = 0;
```

```
    printf("Input values:\n");
    printf("num1 = %d\n", num1);
    printf("num2 = %d\n", num2);

    __asm__ __volatile__ (
        ".intel_syntax noprefix\n\t"
        "mov eax, %1\n\t"
        "imul eax, %2\n\t"
        "mov %0, eax\n\t"
        ".att_syntax prefix\n\t"
        : "=r" (result)
        : "r" (num1), "r" (num2)
        : "eax"
    );

    printf("\nAfter multiplication:\n");
    printf("result = %d\n", result);

    return 0;
}
```

Operand Numbering:

: "=r" (result) ← %0 (outputs numbered first)
: "r" (num1), "r" (num2) ← %1, %2 (inputs continue numbering)

Constraint Types:

- `"=r"`: Write-only output
- `"r"`: Read-only input
- `"+r"`: Read-write

Part 3: The Memory Operand Challenge

Version 3.1 - BROKEN (Common Error)

// THIS VERSION HAS INTENTIONAL ERRORS!

#include <stdio.h>

```
int main() {
    int source = 42;
    int destination = 0;

    __asm__ __volatile__ (
        ".intel_syntax noprefix\n\t"
        "mov eax, %1\n\t"        // ERROR: No brackets with "m"
        "mov %0, eax\n\t"        // ERROR: No brackets with "m"
        ".att_syntax prefix\n\t"
        : "=m" (destination)
        : "m" (source)
        : "eax"
    );

    return 0;
}
```

Compilation Error:

Error: junk `(%rbp)' after expression

Why it fails:

- GCC expands `%1` (with "m" constraint) to AT&T syntax: `-8(%rbp)`
- The line becomes: `mov eax, -8(%rbp)`
- Intel syntax parser doesn't understand AT&T `(%rbp)` notation
- Adding brackets makes it worse: `mov eax, [-8(%rbp)]`

Version 3.2 - BROKEN (Another Attempt)

```
// THIS VERSION ALSO HAS ERRORS!

#include <stdio.h>

int main() {
    int counter = 10;

    __asm__ __volatile__ (
        ".intel_syntax noprefix\n\t"
        "add DWORD PTR %0, 5\n\t"    // ERROR: Missing brackets
```

```
        ".att_syntax prefix\n\t"
        : "+m" (counter)
        :
        : "memory"
    );

    return 0;
}
```

Same Error:

Error: junk `(%rbp)' after expression

The Problem: Intel syntax + "m" constraint = incompatible in GCC inline assembly

---

Part 4: Working Solutions

Solution A: Intel Syntax with Pointers (WORKS)

```
#include <stdio.h>

int main() {
    int source = 42;
    int destination = 0;
    int *p_src = &source;      // Use pointers!
    int *p_dst = &destination;

    printf("Before: source = %d, destination = %d\n", source, destination);

    __asm__ __volatile__ (
        ".intel_syntax noprefix\n\t"
        "mov eax, [%0]\n\t"      // Pointer in register - works!
        "mov [%1], eax\n\t"
        ".att_syntax prefix\n\t"
        :
        : "r" (p_src), "r" (p_dst)  // "r" not "m" - KEY!
        : "eax", "memory"
    );

    printf("After:  source = %d, destination = %d\n", source, destination);
```

```
    return 0;
}
```

Why it works:

- `"r"` constraint puts pointer VALUE in a register (e.g., rsi)
- `%0` expands to register name: `rsi`
- Result: `mov eax, [rsi]` - valid Intel syntax!

Solution B: AT&T Syntax with Memory (WORKS)

```
#include <stdio.h>

int main() {
    int data = 15;
    int output = 0;

    printf("Before: data = %d\n", data);

    __asm__ __volatile__ (
        "movl %1, %%eax\n\t"     // AT&T: source first
        "addl $20, %%eax\n\t"
        "movl %%eax, %0\n\t"
        : "=m" (output)          // "m" works in AT&T
        : "m" (data)
        : "eax"
    );

    printf("After:  output = %d\n", output);

    return 0;
}
```

AT&T Syntax Rules:

- Source operand comes FIRST
- Registers need `%%` prefix
- Immediates need `$` prefix
- Size suffixes: `movb`, `movw`, `movl`, `movq`

Version 4.0 - Complete Working Example

```
#include <stdio.h>

int main() {
    // Example 1: Memory Access via Pointers (Intel syntax)
    int source = 42;
    int destination = 0;
    int *p_src = &source;
    int *p_dst = &destination;

    printf("=== Example 1: Memory Access via Pointers ===\n");
    printf("Before: source = %d, destination = %d\n", source, destination);

    __asm__ __volatile__ (
        ".intel_syntax noprefix\n\t"
        "mov eax, [%0]\n\t"
        "mov [%1], eax\n\t"
        ".att_syntax prefix\n\t"
        :
        : "r" (p_src), "r" (p_dst)
        : "eax", "memory"
    );

    printf("After:  source = %d, destination = %d\n\n", source, destination);


    // Example 2: Increment via pointer
    int counter = 10;
    int *p_counter = &counter;

    printf("=== Example 2: Direct Memory Increment ===\n");
    printf("Before: counter = %d\n", counter);

    __asm__ __volatile__ (
        ".intel_syntax noprefix\n\t"
        "mov rax, %0\n\t"
        "add DWORD PTR [rax], 5\n\t"
        ".att_syntax prefix\n\t"
        :
        : "r" (p_counter)
        : "rax", "memory"
    );

    printf("After:  counter = %d\n\n", counter);
```

```
// Example 3: Array element swap
int array[3] = {100, 200, 300};
int *p_array = array;

printf("=== Example 3: Array Element Swap ===\n");
printf("Before: array[0] = %d, array[1] = %d\n", array[0], array[1]);

__asm__ __volatile__ (
    ".intel_syntax noprefix\n\t"
    "mov rdi, %0\n\t"
    "mov eax, [rdi]\n\t"
    "mov ebx, [rdi+4]\n\t"
    "mov [rdi], ebx\n\t"
    "mov [rdi+4], eax\n\t"
    ".att_syntax prefix\n\t"
    :
    : "r" (p_array)
    : "rax", "rbx", "rdi", "memory"
);

printf("After:  array[0] = %d, array[1] = %d\n\n", array[0], array[1]);


// Example 4: AT&T Syntax with Memory Operands
int data = 15;
int output = 0;

printf("=== Example 4: AT&T Syntax Memory Operands ===\n");
printf("Before: data = %d\n", data);

__asm__ __volatile__ (
    "movl %1, %%eax\n\t"
    "addl $20, %%eax\n\t"
    "movl %%eax, %0\n\t"
    : "=m" (output)
    : "m" (data)
    : "eax"
);

printf("After:  output = %d\n\n", output);
```

```
// Example 5: Memory to Memory copy
int value1 = 99;
int value2 = 0;

printf("=== Example 5: Memory-to-Memory Copy (AT&T) ===\n");
printf("Before: value1 = %d, value2 = %d\n", value1, value2);

__asm__ __volatile__ (
    "movl %1, %%eax\n\t"
    "movl %%eax, %0\n\t"
    : "=m" (value2)
    : "m" (value1)
    : "eax"
);

printf("After:  value1 = %d, value2 = %d\n\n", value1, value2);


// Example 6: Comparison of approaches
printf("=== Example 6: Approach Comparison ===\n");
int test = 50;
int result_intel = 0;
int result_att = 0;

// Intel syntax - use pointers
int *p_test = &test;
int *p_result_intel = &result_intel;
__asm__ __volatile__ (
    ".intel_syntax noprefix\n\t"
    "mov rax, %0\n\t"
    "mov eax, [rax]\n\t"
    "add eax, 5\n\t"
    "mov rbx, %1\n\t"
    "mov [rbx], eax\n\t"
    ".att_syntax prefix\n\t"
    :
    : "r" (p_test), "r" (p_result_intel)
    : "rax", "rbx", "memory"
);

// AT&T syntax - use memory operands directly
__asm__ __volatile__ (
    "movl %1, %%eax\n\t"
    "addl $5, %%eax\n\t"
```

```
        "movl %%eax, %0\n\t"
        : "=m" (result_att)
        : "m" (test)
        : "eax"
    );

    printf("Intel approach result: %d\n", result_intel);
    printf("AT&T approach result:  %d\n", result_att);

    return 0;
}
```

Part 5: Common Errors and Fixes

Error 1: Operand Number Out of Range

```
// BROKEN CODE
__asm__ __volatile__ (
    "mov eax, [%1]\n\t"      // ERROR: %1 doesn't exist!
    "mov [%2], eax\n\t"      // ERROR: %2 doesn't exist!
    :                        // Empty output section
    : "r" (p_src), "r" (p_dst)  // These are %0 and %1, not %1 and %2!
);
```

Error Message:

error: invalid 'asm': operand number out of range

Fix: When output section is empty, inputs start at %0

```
// CORRECT CODE
__asm__ __volatile__ (
    "mov eax, [%0]\n\t"      // %0 = first input
    "mov [%1], eax\n\t"      // %1 = second input
    :                        // Empty
    : "r" (p_src), "r" (p_dst)
);
```

Numbering Rule:

- Outputs are numbered first: %0, %1, ...
- Inputs continue the sequence
- If no outputs, inputs start at %0

Error 2: Intel Syntax + Memory Constraint

```
// BROKEN CODE
__asm__ __volatile__ (
    ".intel_syntax noprefix\n\t"
    "mov eax, %0\n\t"        // or "mov eax, [%0]\n\t"
    : : "m" (variable)       // "m" constraint with Intel = FAIL
);
```

Error Message:

Error: junk `(%rbp)' after expression

Why: GCC expands "m" constraint to AT&T format, incompatible with Intel syntax

Fix Option 1: Use pointers with "r"

```
int *ptr = &variable;
__asm__ (
    ".intel_syntax noprefix\n\t"
    "mov eax, [%0]\n\t"
    : : "r" (ptr)            // "r" not "m"!
);
```

Fix Option 2: Use AT&T syntax

```
__asm__ (
    "movl %0, %%eax\n\t"
    : : "m" (variable)       // "m" works in AT&T
);
```

Error 3: Missing Memory Clobber

```
// POTENTIALLY BROKEN CODE
__asm__ __volatile__ (
    "add DWORD PTR [rax], 5\n\t"
```

```
    :
    : "r" (ptr)
    : "rax"                 // Missing "memory"!
);
```

Problem: Compiler doesn't know memory changed, may optimize incorrectly

Fix: Always include "memory" when modifying memory

```
__asm__ __volatile__ (
    "add DWORD PTR [rax], 5\n\t"
    :
    : "r" (ptr)
    : "rax", "memory"       // Added "memory"
);
```

Part 6: Quick Reference Cheat Sheet

Constraint Types

Constraint	Meaning	Example Use
`"r"`	Read-only, any register	Input values
`"=r"`	Write-only, any register	Output only
`"+r"`	Read-write, any register	Modified values
`"m"`	Memory operand (AT&T only!)	Direct memory access
`"=m"`	Write-only memory (AT&T)	Output to memory
`"+m"`	Read-write memory (AT&T)	Modify memory in-place
`"i"`	Immediate constant	Compile-time values

Intel vs AT&T Syntax

Feature	Intel	AT&T

Direction	`mov dest, src`	`mov src, dest`
Register	`eax`	`%eax` (inline: `%%eax`)
Immediate	`5`	`$5`
Memory	`[rax]`	`(%rax)`
Offset	`[rax+4]`	`4(%rax)`
Size	`DWORD PTR [rax]`	`movl (%rax)`

Common Clobbers

Clobber	Meaning
`"eax"`	EAX register modified
`"memory"`	Memory contents changed
`"cc"`	Condition codes (flags) changed

The Golden Rules

1. **Intel syntax + "m" constraint = ERROR**
    - Use "r" with pointers instead
    - Or switch to AT&T syntax
2. **Operand numbering:**
    - Outputs first: %0, %1, ...
    - Inputs continue: %n, %n+1, ...
    - Empty output? Inputs start at %0
3. **Always include clobbers:**
    - Modified registers
    - "memory" if memory changes
    - "cc" if flags change

4. **Memory access patterns:**
    - Intel + pointers: `int *p = &var; "mov eax, [%0]" : "r" (p)`
    - AT&T + direct: `"movl %0, %%eax" : "m" (var)`

Compilation Commands

```
# 32-bit
gcc -m32 -o program program.c

# 64-bit (default)
gcc -o program program.c

# With warnings
gcc -Wall -o program program.c

# View generated assembly
gcc -S -masm=intel program.c
# Creates program.s with assembly code
```

Practice Exercises

Exercise 1: Basic Operations

Write inline assembly to:

- Add two numbers
- Subtract two numbers
- Use both as inputs, store result in output variable

Exercise 2: Bitwise Operations

Implement using inline assembly:

- XOR swap (without XCHG instruction)
- Bit rotation
- Count leading zeros

Exercise 3: Memory Operations

Create functions that:

- Copy array elements using assembly
- Find maximum value in array
- Reverse an array in-place

Exercise 4: Mixed Syntax

Rewrite the same operation in:

- Intel syntax with pointers
- AT&T syntax with memory operands
- Compare generated code

Summary

Key Takeaways:

1. Inline assembly bridges C and machine code
2. Constraints tell compiler how to bind variables
3. Intel syntax is more readable but has limitations with "m"
4. AT&T syntax works better with direct memory operands
5. Always specify clobbers to prevent optimization bugs
6. Use pointers with "r" constraint for Intel syntax memory access

The Fundamental Limitation: GCC's inline assembly was designed for AT&T syntax. When using Intel syntax, prefer register operands ("r") over memory operands ("m").

Best Practice:

- Start simple with register operations
- Use Intel syntax for clarity
- Switch to AT&T when you need direct memory operands
- Always test your code!