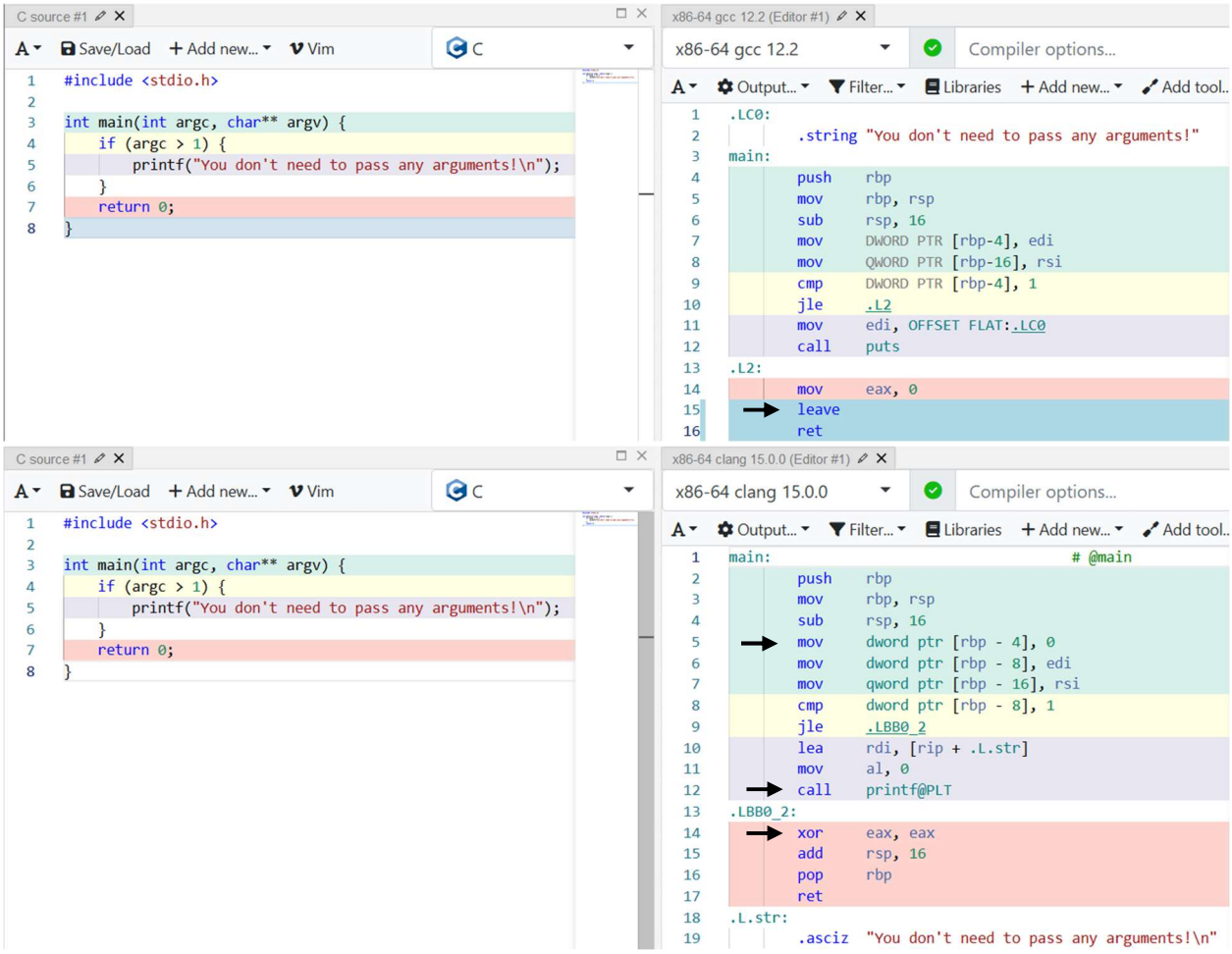
Assembly Language Research Report

Representative if-statement code

```
#include <stdio.h>

int main(int argc, char** argv) {
    if (argc > 1) {
        printf("You don't need to pass
                any arguments!\n");
    }
    return 0;
}
```

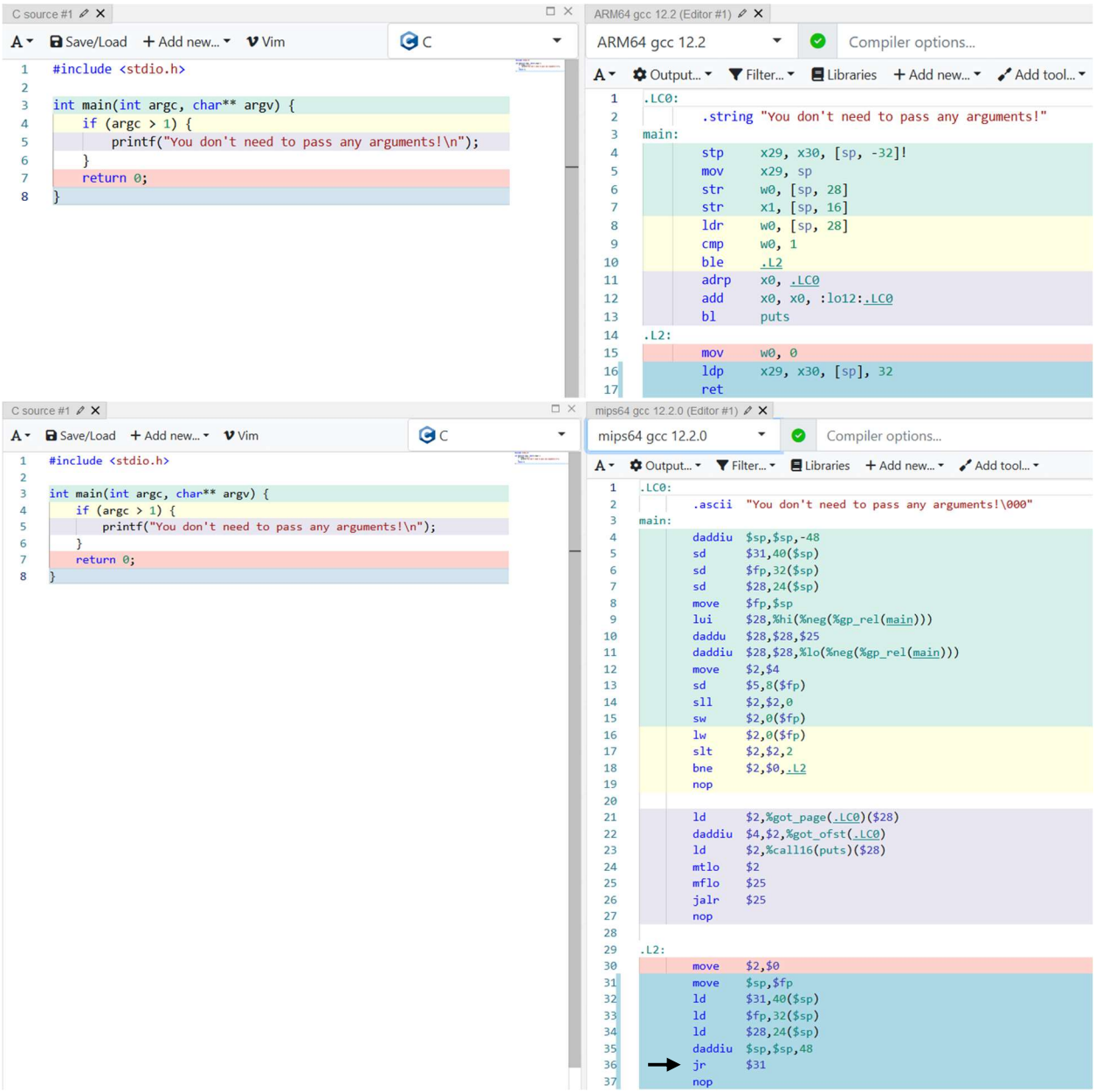
I chose to check `argc` in my if-statement because it meant that I didn't have to declare any variables. It also changes depending on how you run it, so it can't be optimized away.

CISC x86-64 Assembly



The `leave` instruction is unique to gcc's x86-64 output and the use of the `xor eax, eax` instruction to set eax to 0 is unique to clang's x86-64 output. Additionally, clang sets up the parameters on the stack differently, it sets the first two bytes to 0 (probably for alignment reasons) (line 5).

RISC Assembly



First, only x86-64 and ARM64 represent strings in the same way (.string, no \n character). Clang uses an .asciz section and the \n character and MIPS64 uses a .ascii section with, oddly, \000.

Second, though I have organized the assembly outputs into CISC and RISC, the ARM64 output is very similar to the x86-64 gcc output while the MIPS64 assembly is much longer (which is what I expected from assembly for a RISC architecture).

Unique to clang's assembly is the call to `printf@PLT` for `printf`, all other versions call `puts`. It's also cool to see it referring to the PLT, which we just learned about in class.

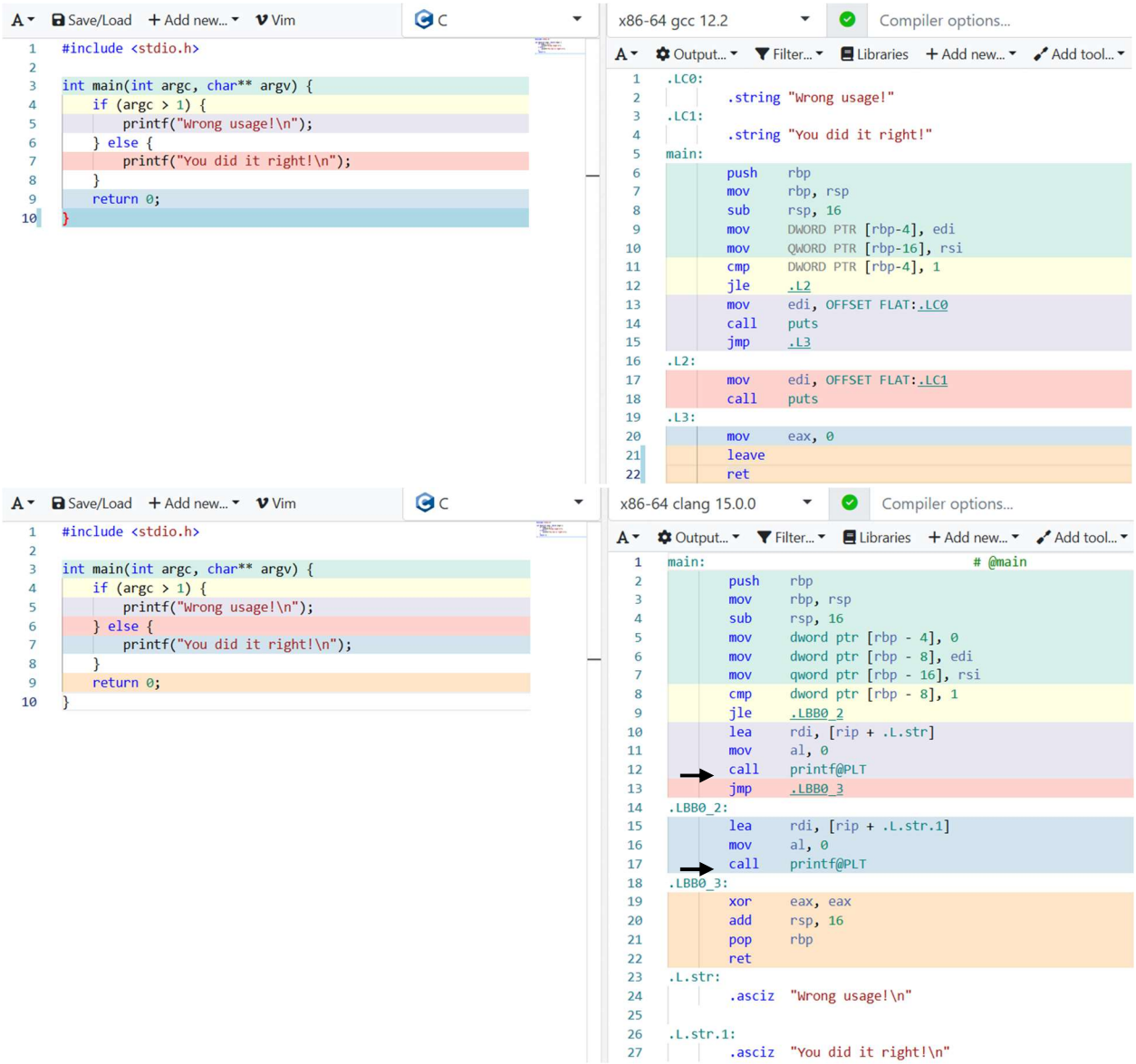
Only MIPS64 doesn't use `ret` instruction. Instead, it uses `jr` (jump to value in register).

Representative if-else-statement code

```
#include <stdio.h>

int main(int argc, char** argv) {
    if (argc > 1) {
        printf("Wrong usage!\n");
    } else {
        printf("You did it right!\n");
    }
    return 0;
}
```

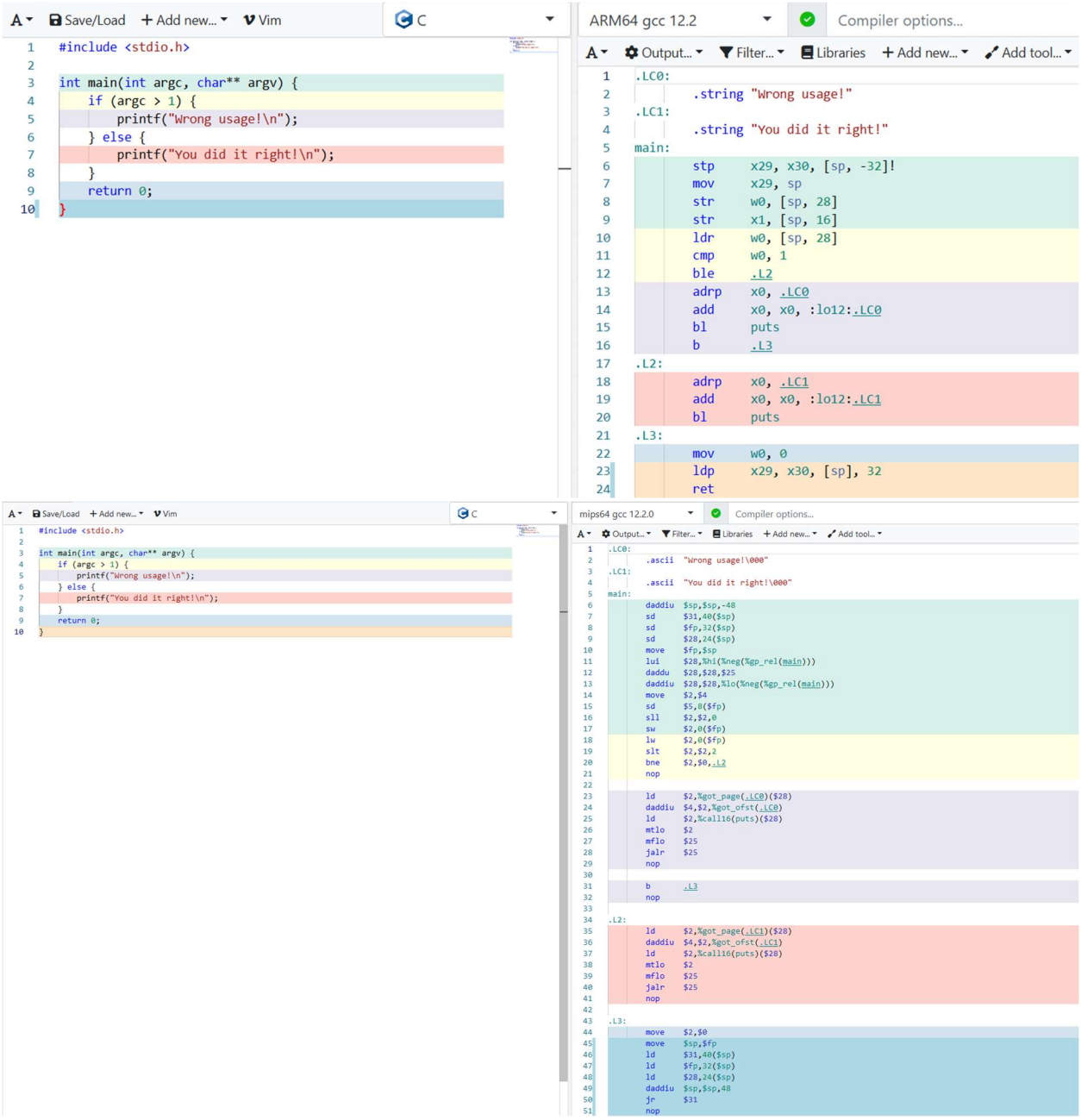
Since an if-else statement is a simple extension of the if statement the same rationale as previous applies.

CISC x86-64 Assembly



One idiosyncrasy I notice is the clang x86-64 is the instruction `mov al, 0`. In the assembly, I only saw this before the call to printf, yet I don't understand the benefit of zeroing out the top 32 bits.

RISC Assembly



RISC based architecture uses many general purpose registers, and the corresponding assembly language needs to have a way to represent that.

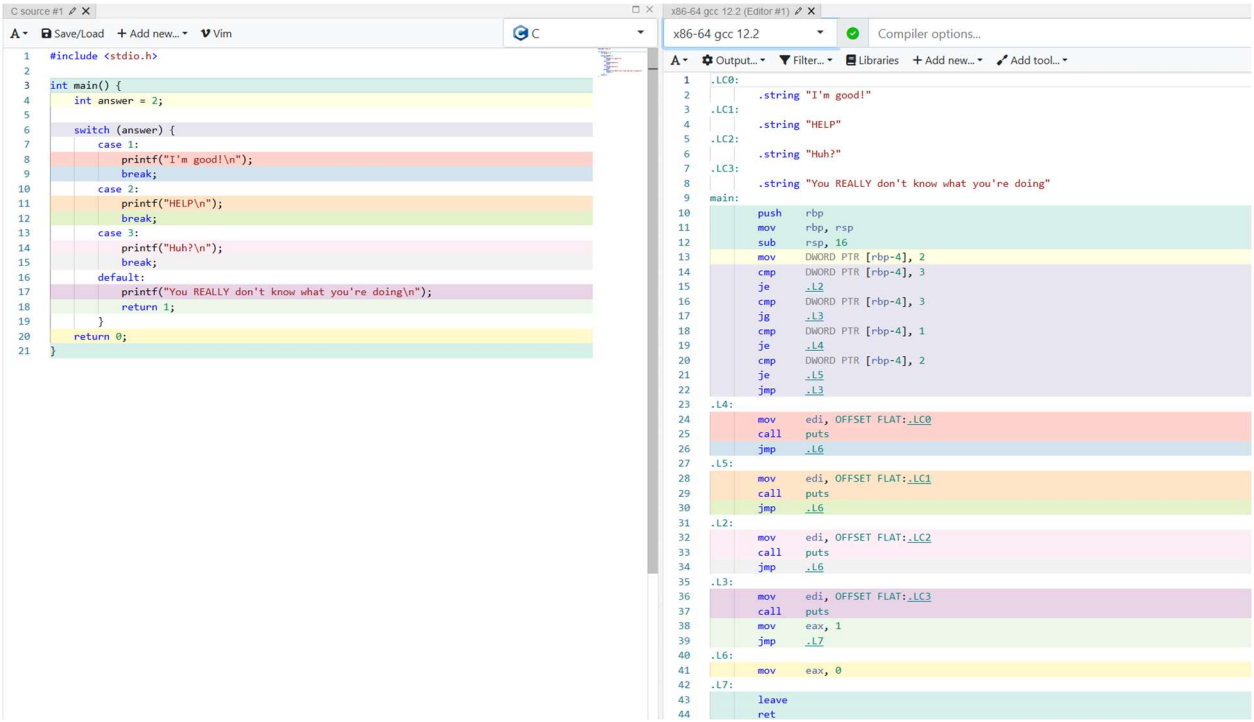
- MIPS has a simple system: registers all begin with the dollar ($) symbol. There are 32 general purpose and 32 floating point registers (named with $f[0-31]). The general purpose registers can be accessed by their number or their name (which corresponds with their defined use)
- ARM64 also has 32 general purpose and 32 floating point registers, however, the floating point registers can also be used for vector operations and can contain multiple independent values. The general purpose registers are 64 bits and can be accessed with x[0-31], and the lower 32 bits can be accessed with w[0-31].

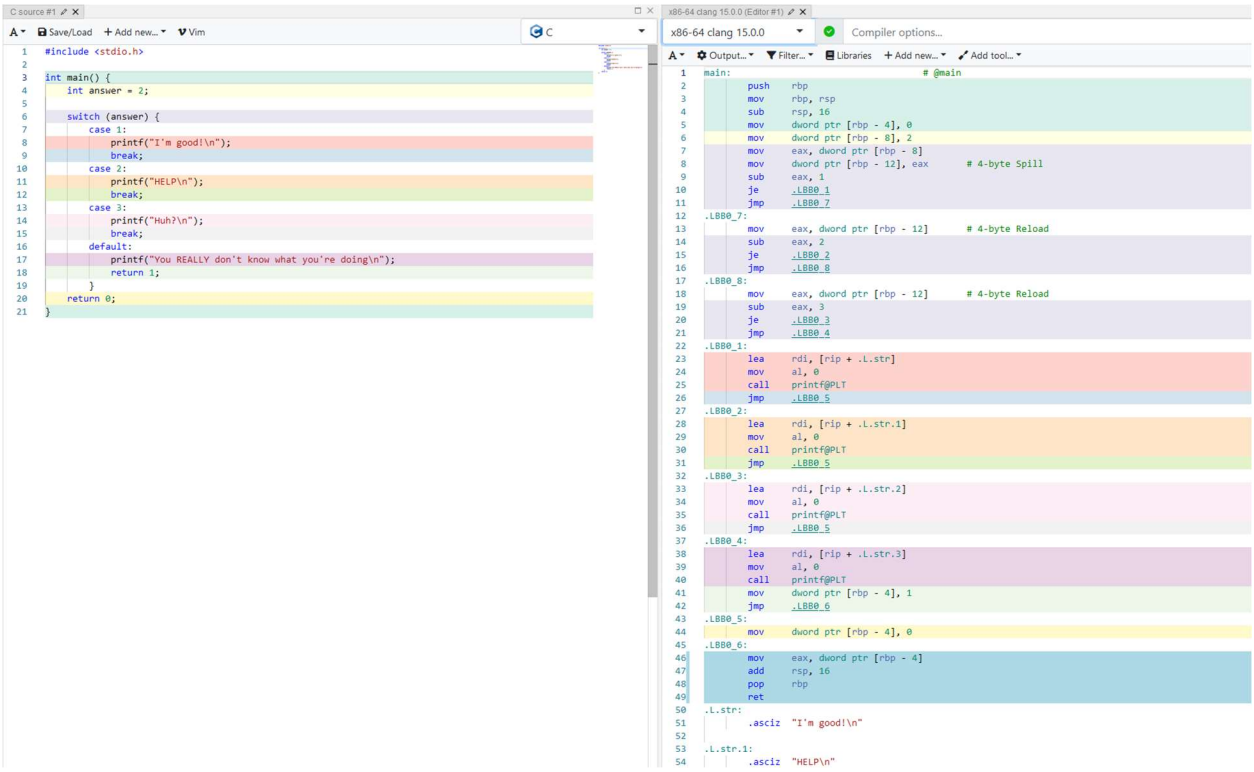
Representative switch statement code

```
#include <stdio.h>

int main() {
    int answer = 2;

    switch (answer) {
        case 1:
            printf("I'm good!\n");
            break;
        case 2:
            printf("HELP\n");
            break;
        case 3:
            printf("Huh?\n");
            break;
        default:
            printf("You REALLY don't
know what you're doing\n");
            return 1;
        }
    return 0;
}
```

For this piece of code, I figured it would be useful to see a variable declaration in the assembly.

CISC x86-64 Assembly





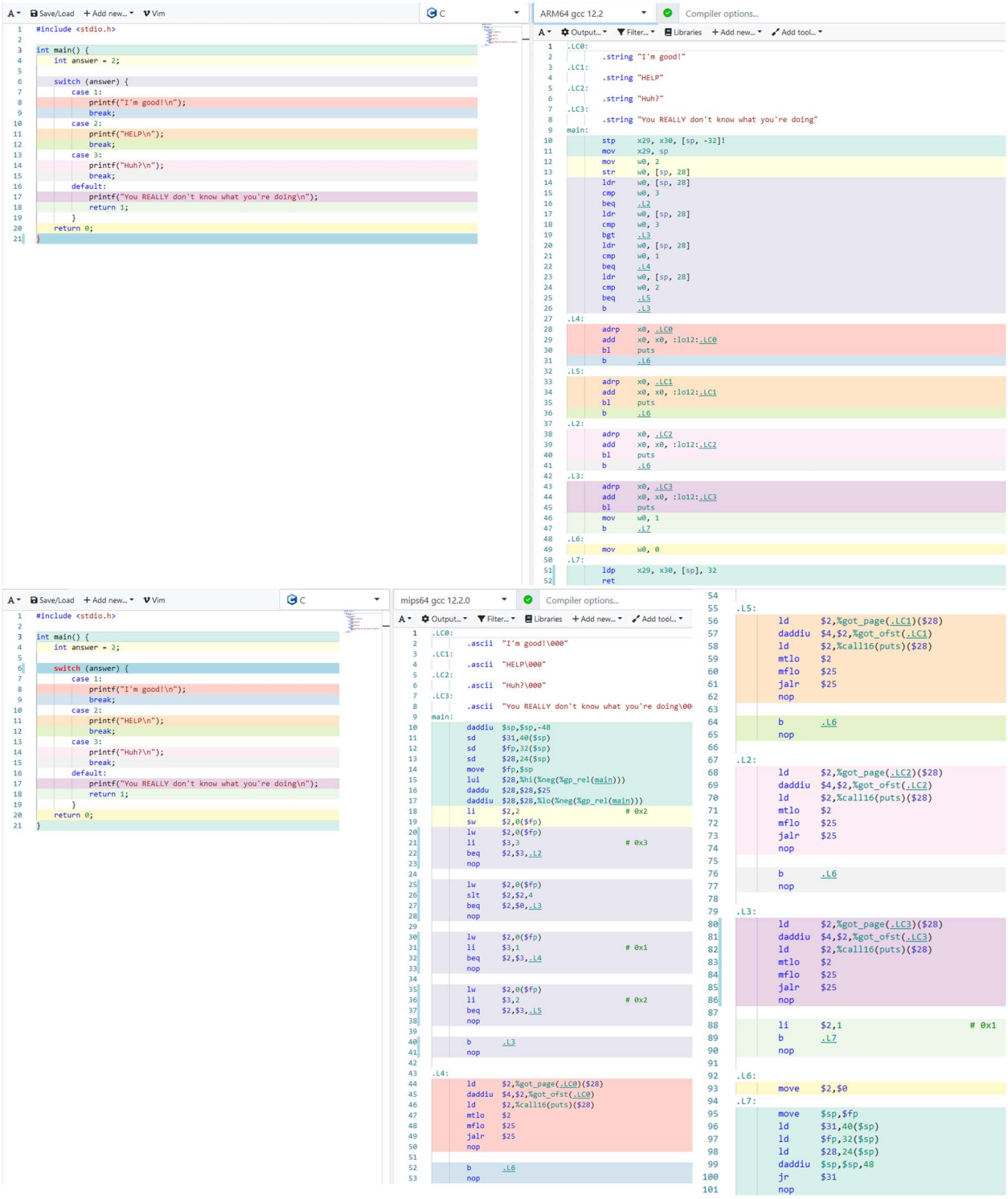
```c
#include <stdio.h>

int main() {
    int answer = 2;

    switch (answer) {
        case 1:
            printf("I'm good!\n");
            break;
        case 2:
            printf("HELP\n");
            break;
        case 3:
            printf("Huh?\n");
            break;
        default:
            printf("You REALLY don't know what you're doing\n");
            return 1;
    }
    return 0;
}
```

```asm
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], 0
        mov     dword ptr [rbp - 8], 2
        mov     eax, dword ptr [rbp - 8]
        mov     dword ptr [rbp - 12], eax       # 4-byte Spill
        sub     eax, 1
        je      .LBB0_1
        jmp     .LBB0_7
.LBB0_7:
        mov     eax, dword ptr [rbp - 12]       # 4-byte Reload
        sub     eax, 2
        je      .LBB0_2
        jmp     .LBB0_8
.LBB0_8:
        mov     eax, dword ptr [rbp - 12]       # 4-byte Reload
        sub     eax, 3
        je      .LBB0_3
        jmp     .LBB0_4
.LBB0_1:
        lea     rdi, [rip + .L.str]
        mov     al, 0
        call    printf@PLT
        jmp     .LBB0_5
.LBB0_2:
        lea     rdi, [rip + .L.str.1]
        mov     al, 0
        call    printf@PLT
        jmp     .LBB0_5
.LBB0_3:
        lea     rdi, [rip + .L.str.2]
        mov     al, 0
        call    printf@PLT
        jmp     .LBB0_5
.LBB0_4:
        lea     rdi, [rip + .L.str.3]
        mov     al, 0
        call    printf@PLT
        mov     dword ptr [rbp - 4], 1
        jmp     .LBB0_6
.LBB0_5:
        mov     dword ptr [rbp - 4], 0
.LBB0_6:
        mov     eax, dword ptr [rbp - 4]
        add     rsp, 16
        pop     rbp
        ret
.L.str:
        .asciz  "I'm good!\n"

.L.str.1:
        .asciz  "HELP\n"
```

* I have omitted the last few lines which define the last two strings so the text doesn't get any smaller.

There are some small, but interesting differences in these two assemblies (even though they ultimately make no meaningful difference). The two that I noticed is that clang writes the `dword ptr` size directive in lowercase letters and that it writes comments (which is the more interesting of the two, it shows they expect people to be reading the assembly and give them quick helps).
There are also somewhat major structural differences. From a quick glance, gcc has a block (lines 14-22), which does the comparisons and jumps to the correct section, while clang separates it into two named sections and seems to do it inefficiently.

RISC Assembly



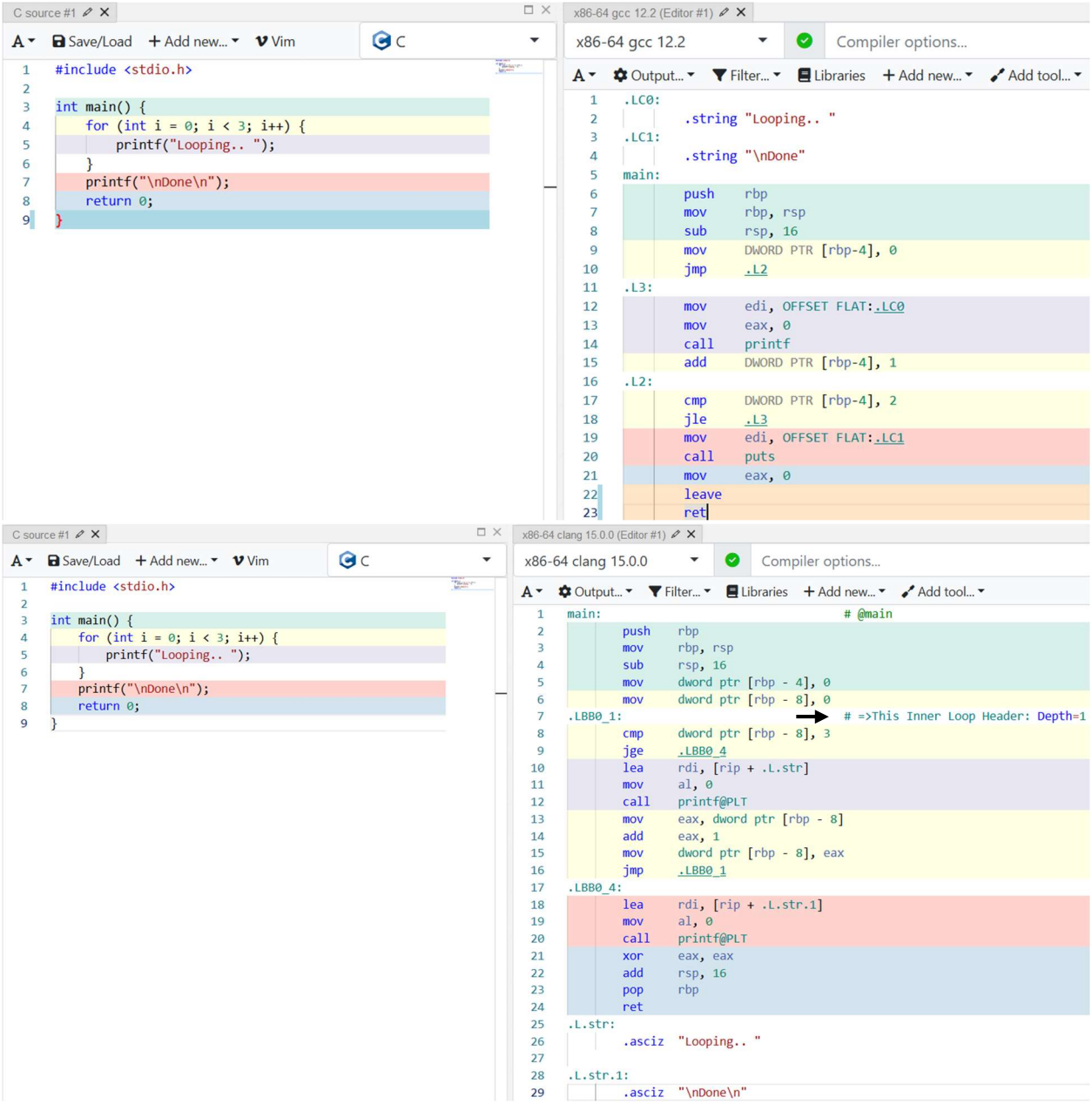
Looking at these two assemblies, it looks like ARM64 is doing the branching logic in a similar way to the x86-64 gcc output. Oddly, for default, both check if it's greater than 3, which seems to ignore the possibility of zero and negative numbers. All of them have a section that loads a 0 to the return code register which is skipped for the default case, because the default returns a 1.

Representative for-loop code

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 3; i++) {
        printf("Looping.. ");
    }
    printf("\nDone\n");
    return 0;
}
```

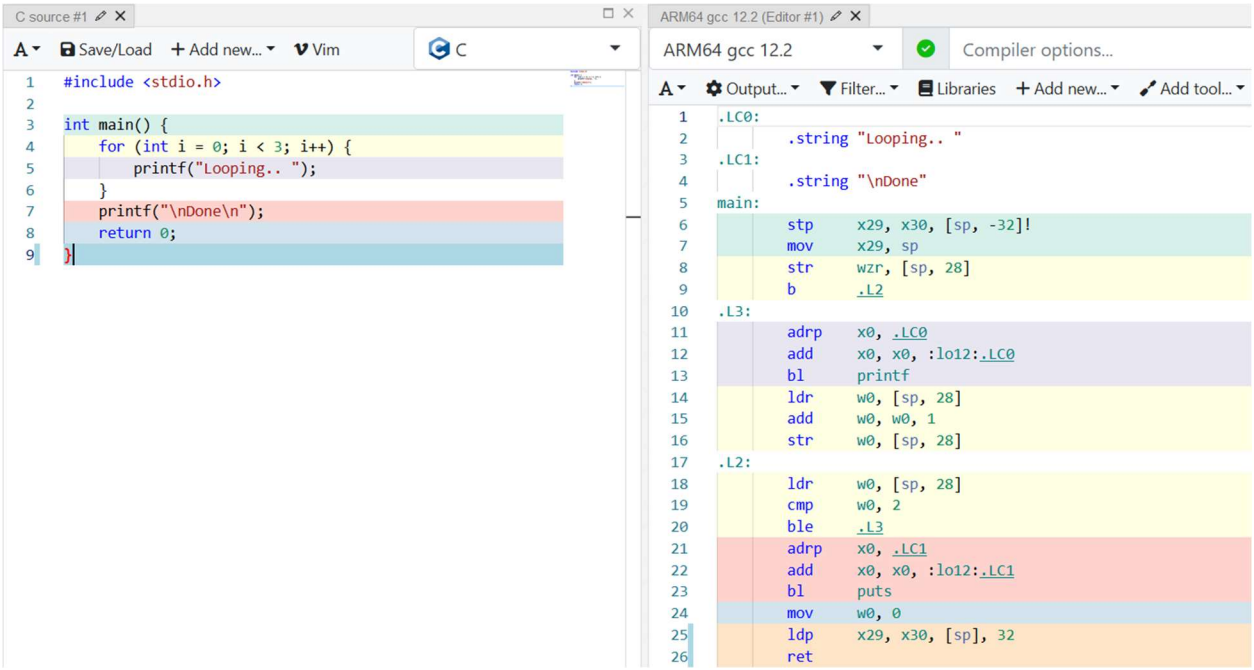
For the for-loop, I just printed out “Looping.. ”, and printed “Done” on a new line to indicate it finished. I wanted to keep the code simple.

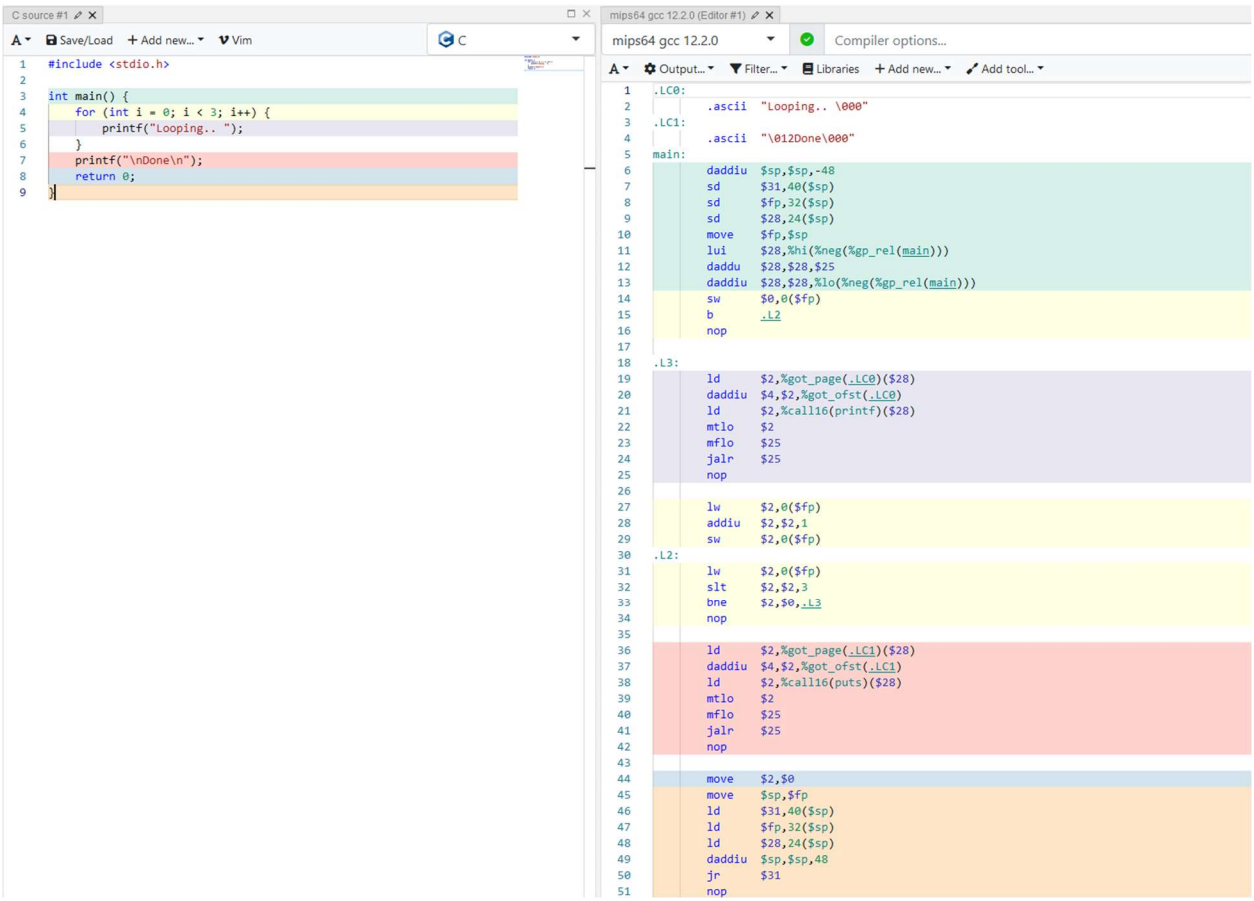
CISC x86-64 Assembly



I just commented on clang's use of comments, but their loop comment is worth another mention. I tried adding another for loop within the first, yet the depth was still 1; even so, it's really cool to see the clang developers add comments to the assembly to make it easier to people to inspect and understand it.

RISC Assembly





```
C source #1
Save/Load  + Add new...  Vim                                   C
1  #include <stdio.h>
2
3  int main() {
4      for (int i = 0; i < 3; i++) {
5          printf("Looping.. ");
6      }
7      printf("\nDone\n");
8      return 0;
9  }
```

```
mips64 gcc 12.2.0 (Editor #1)
mips64 gcc 12.2.0                                   Compiler options...
Output...  Filter...  Libraries  + Add new...  Add tool...
1  .LC0:
2          .ascii  "Looping.. \000"
3  .LC1:
4          .ascii  "\012Done\000"
5  main:
6          daddiu  $sp,$sp,-48
7          sd      $31,40($sp)
8          sd      $fp,32($sp)
9          sd      $28,24($sp)
10         move    $fp,$sp
11         lui     $28,%hi(%neg(%gp_rel(main)))
12         daddu   $28,$28,$25
13         daddiu  $28,$28,%lo(%neg(%gp_rel(main)))
14         sw      $0,0($fp)
15         b       .L2
16         nop
17
18 .L3:
19         ld      $2,%got_page(.LC0)($28)
20         daddiu  $4,$2,%got_ofst(.LC0)
21         ld      $2,%call16(printf)($28)
22         mtlo    $2
23         mflo    $25
24         jalr    $25
25         nop
26
27         lw      $2,0($fp)
28         addiu   $2,$2,1
29         sw      $2,0($fp)
30 .L2:
31         lw      $2,0($fp)
32         slt     $2,$2,3
33         bne     $2,$0,.L3
34         nop
35
36         ld      $2,%got_page(.LC1)($28)
37         daddiu  $4,$2,%got_ofst(.LC1)
38         ld      $2,%call16(puts)($28)
39         mtlo    $2
40         mflo    $25
41         jalr    $25
42         nop
43
44         move    $2,$0
45         move    $sp,$fp
46         ld      $31,40($sp)
47         ld      $fp,32($sp)
48         ld      $28,24($sp)
49         daddiu  $sp,$sp,48
50         jr      $31
51         nop
```

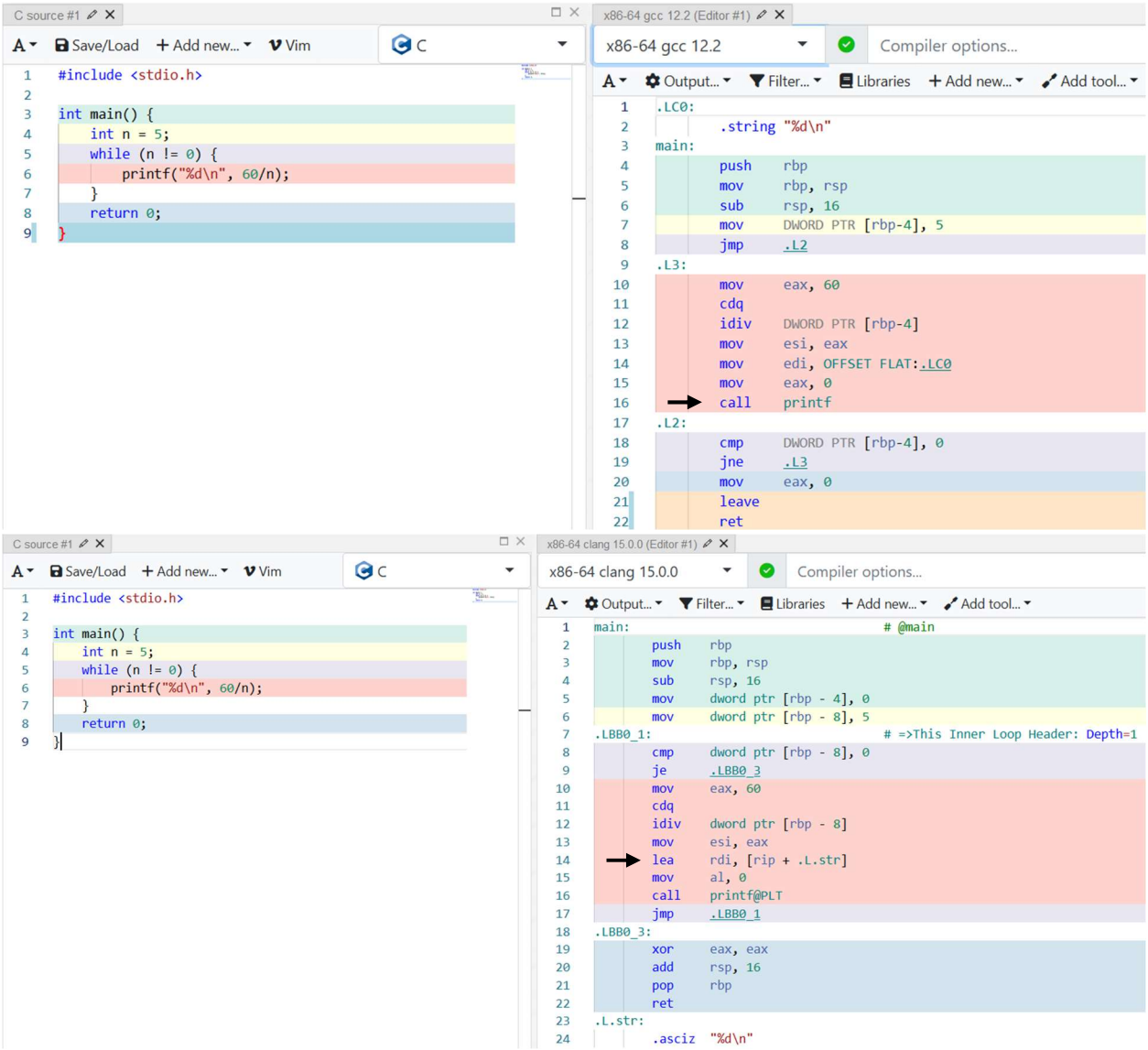
One thing interesting to look at between the four generated assemblies is the different looping structures they employ. One thing common between them was how the loop statement (line 4) was implemented. For the first loop, it would skip past the increment/decrement part, which meant there was a section break within the loop block.

Representative while-loop code

```
#include <stdio.h>

int main() {
    int n = 5;
    while (n != 0) {
        printf("%d\n", 60/n);
    }
    return 0;
}
```

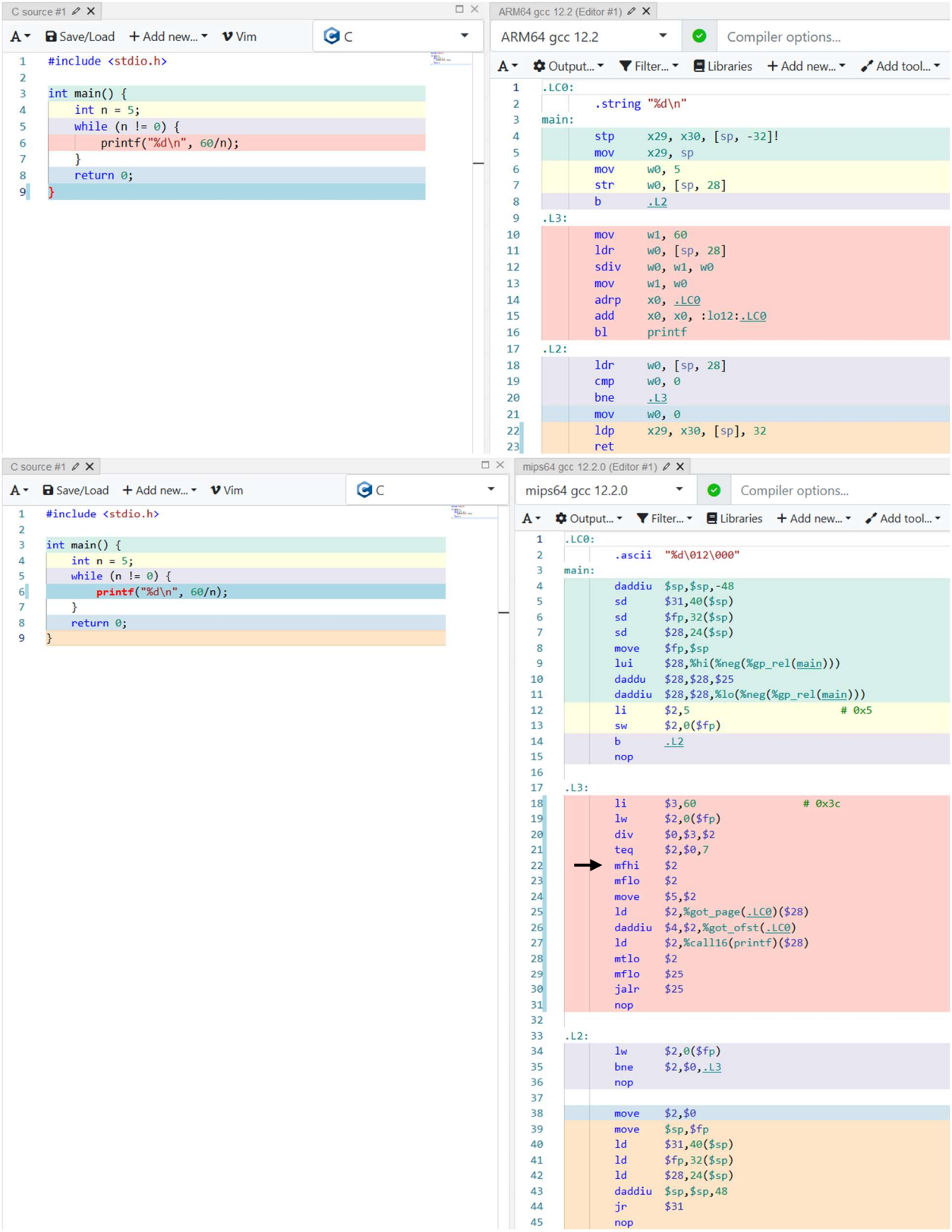
For this piece of code, I decided to do some math, so I printed out increasing numbers while n decreased to 0. It's simple while also demonstrating some basic math.

CISC x86-64 Assembly



These two outputs are remarkably similar, except clang uses the 64-bit register to store the pointer to the format string.

RISC Assembly



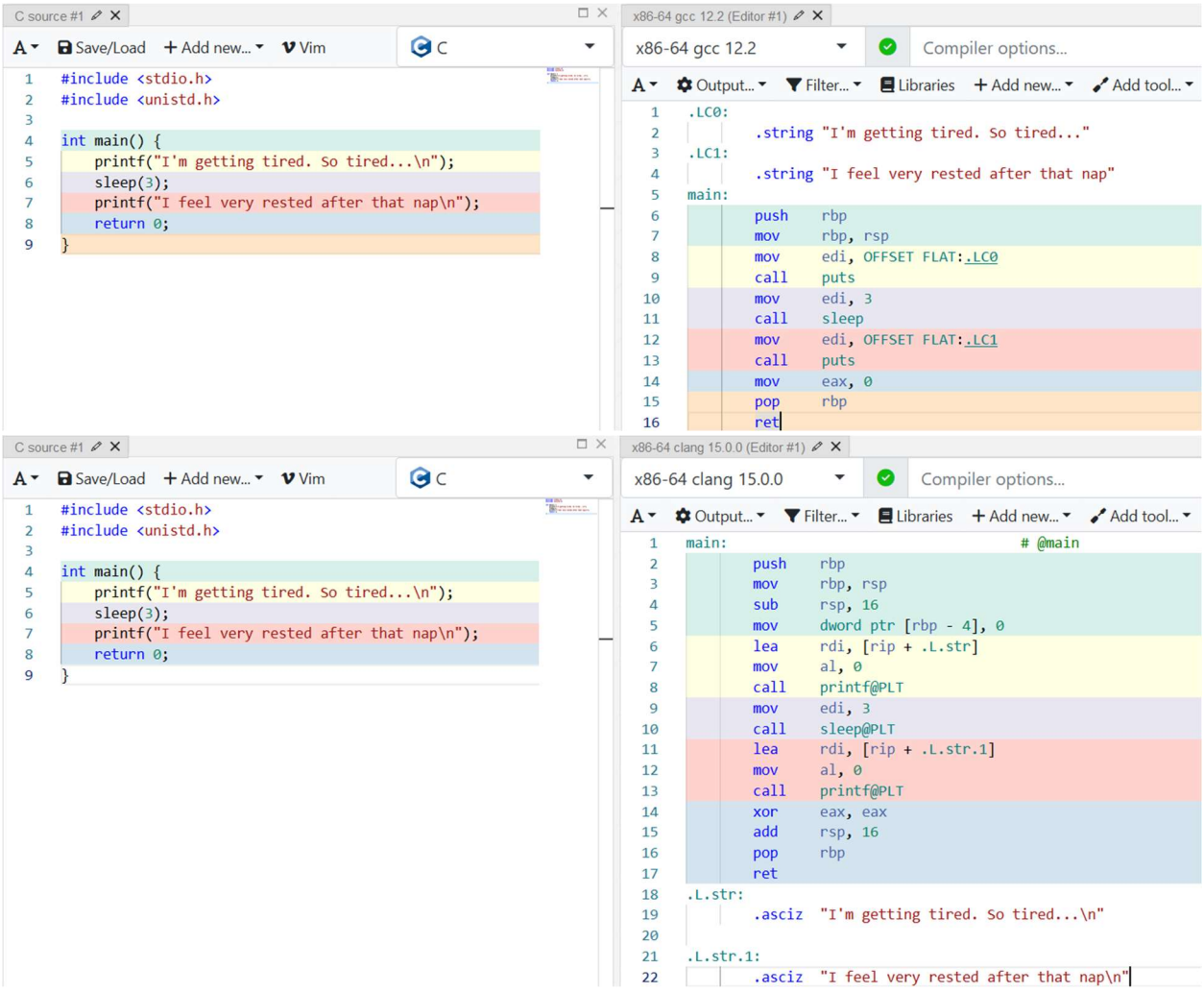
There isn't too much more to comment on. I'm still surprised at how few instructions ARM64 uses, even though it's a RISC assembly language (I have a hunch that each instruction expands to more bytecode, which would make the result closer to MIPS than x86-64).
MIPS' ~~multiply instruction uses a special HI and LO register (lines 22-23), which is unique.~~ Strike that, there is a HI and LO register, but `mfhi/lo` means move from HI/LO. I'm not sure what they are doing.
When you use a format string with `printf`, gcc calls `printf`, otherwise it just calls `puts`.

Representative system call code

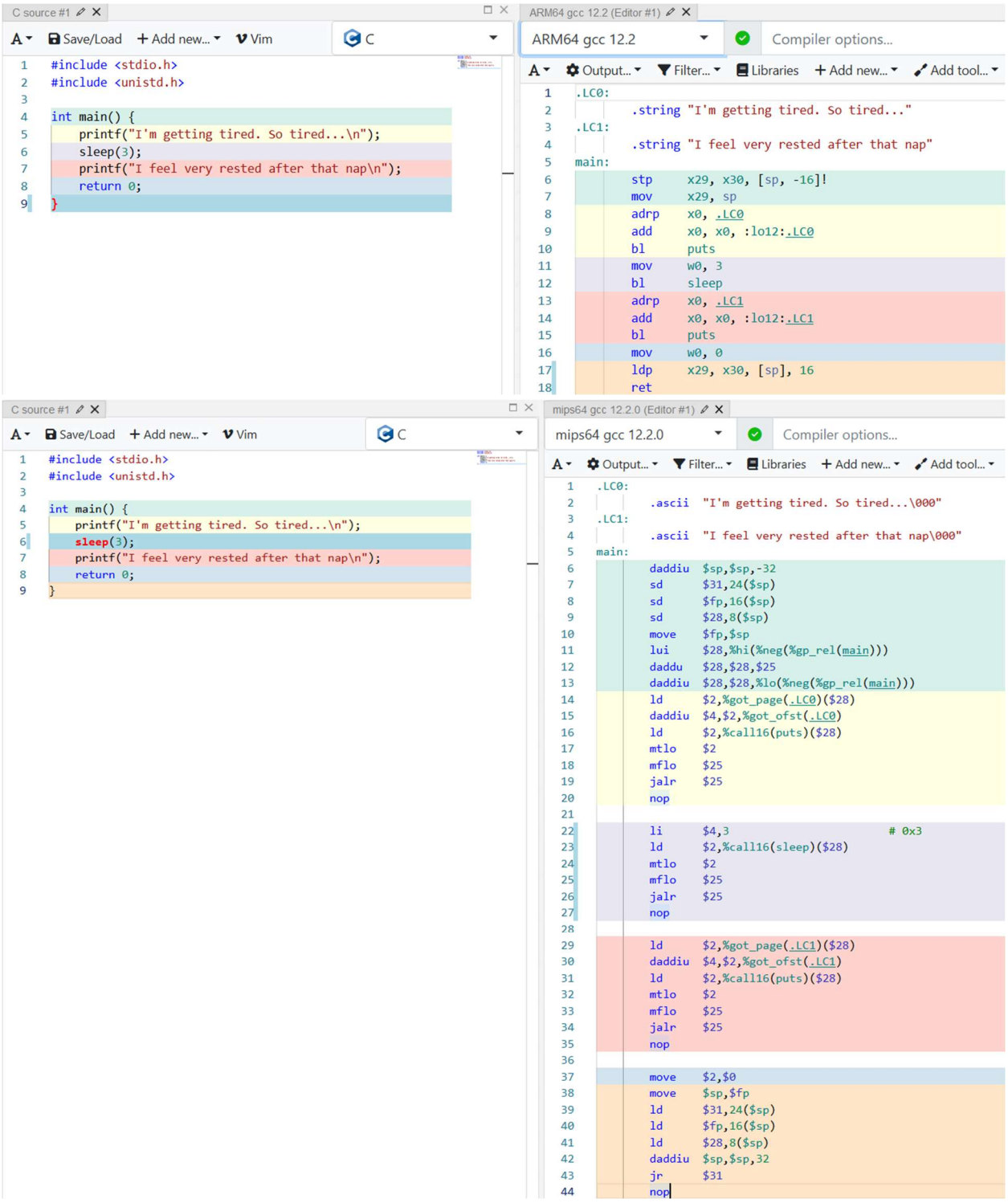
```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("I'm getting tired. So
tired...\n");
    sleep(3);
    printf("I feel very rested after
that nap\n");
    return 0;
}
```

Sleep was the first system call I could think of off the top of my head. And I was feeling very uncreative.

CISC x86-64 Assembly



RISC Assembly



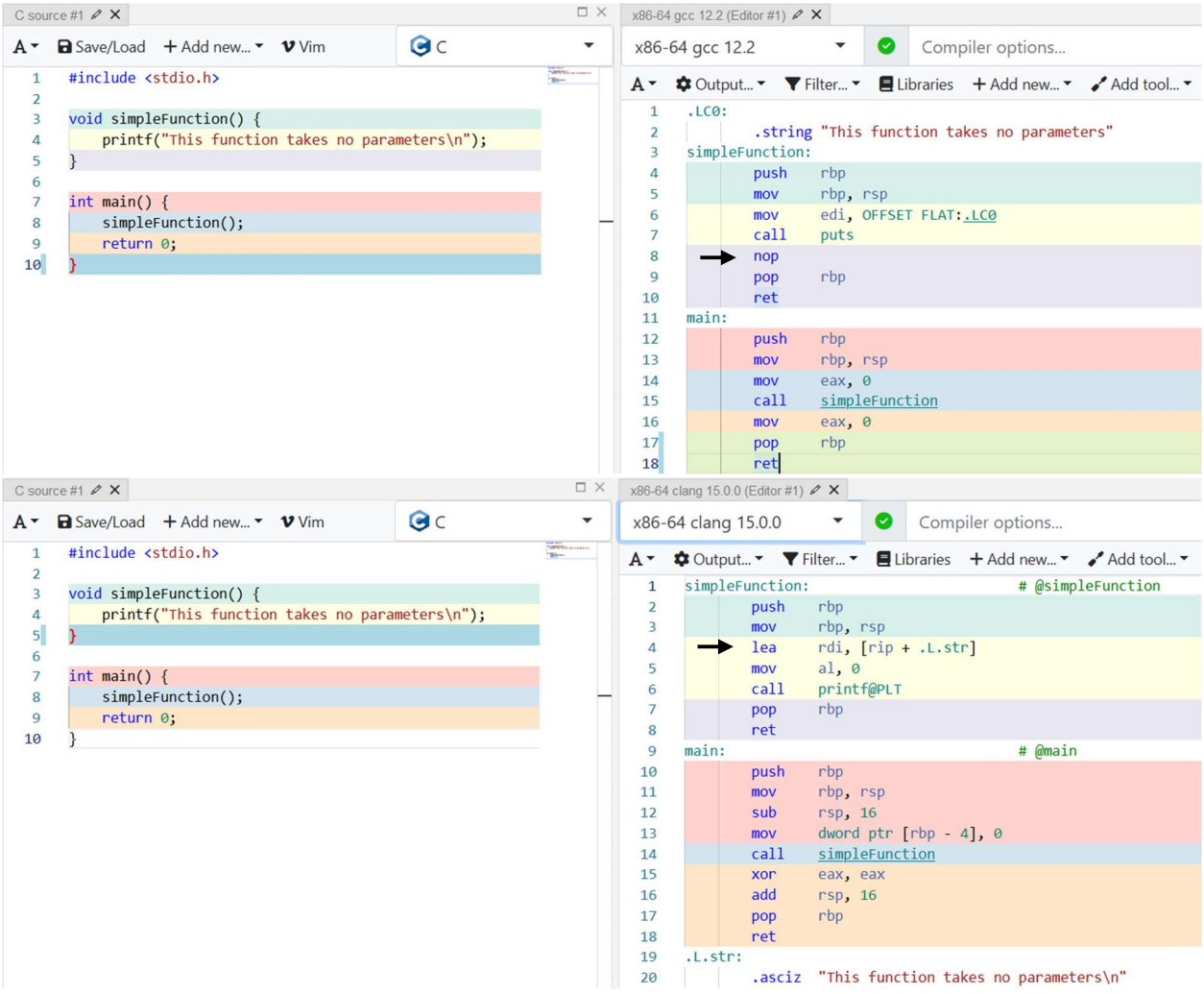
Once again, the two x86-64 assemblies and the ARM64 assemblies are very similar.
I am astonished at how many instructions MIPS uses for this simple program, it's almost three times longer than the gcc x86-64 assembly.

Representative function call code

```
#include <stdio.h>

void simpleFunction() {
    printf("This function takes no
parameters\n");
}

int main() {
    simpleFunction();
    return 0;
}
```

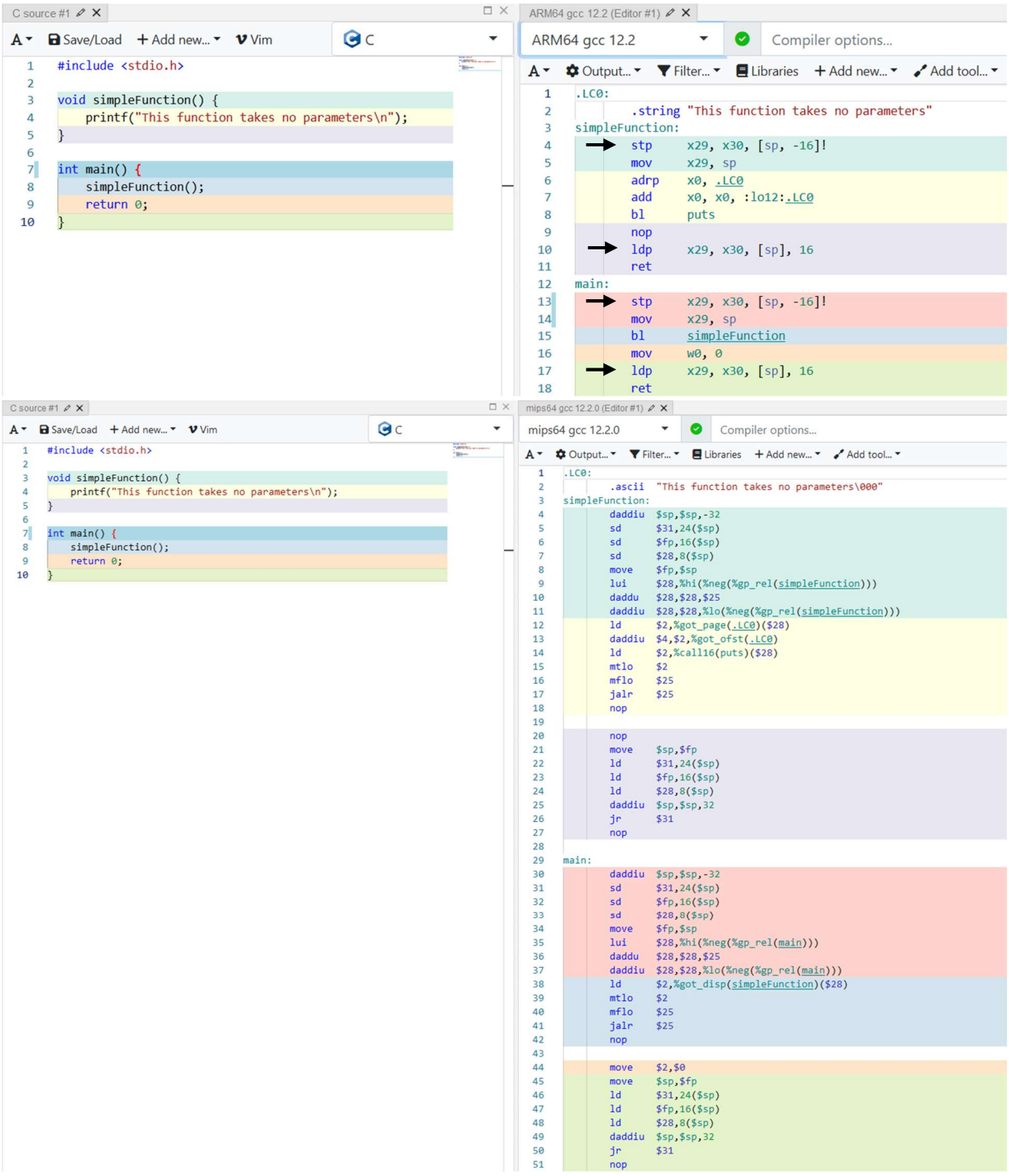
As you can see, no creativity, but that just means it is easier to see how this is written in assembly.

CISC x86-64 Assembly



One interesting thing I've noticed is that clang uses the `lea` instruction in favor of the mov instruction for loading addresses. I'm assuming this is a slight optimization, though I'm not sure.
Another one is the gcc compiler's use of the `nop` instruction of line 8. I'm not sure what the reason for this is.

RISC Assembly



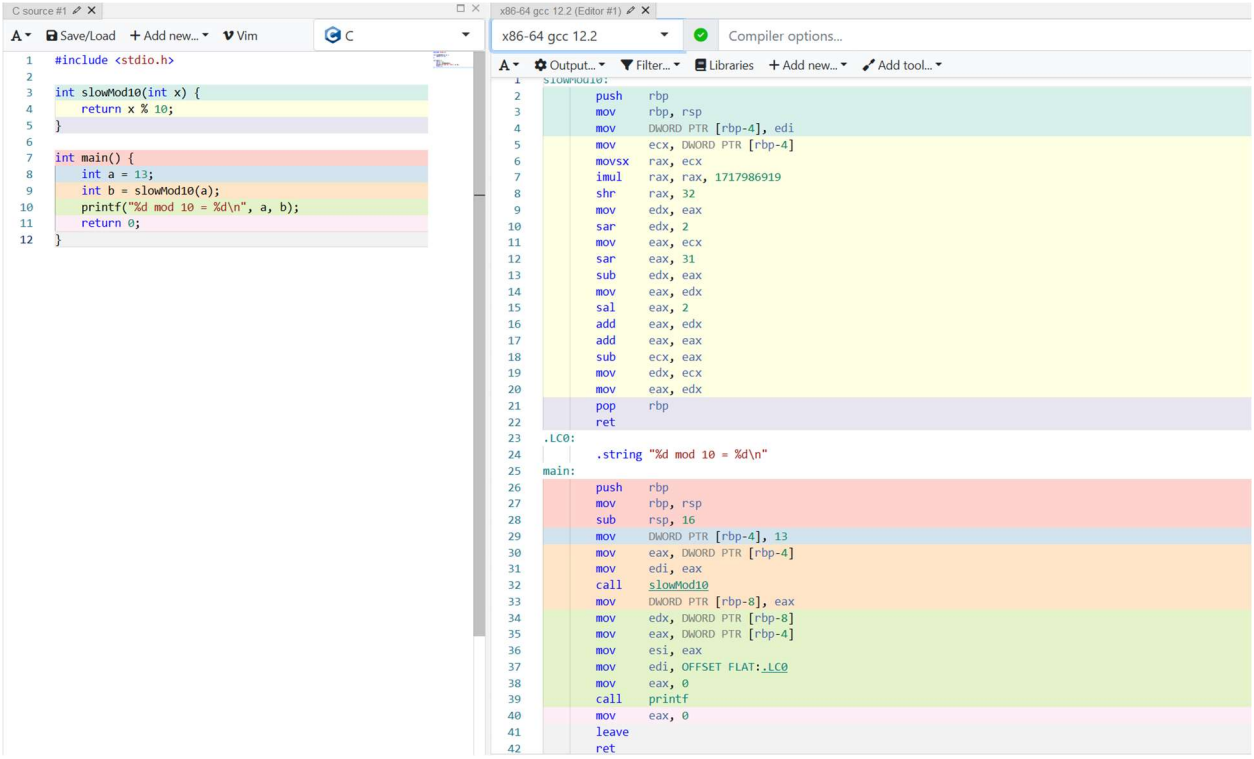
In ARM64, the "!" operator specifies write-back, or pre-increment. So on line 4, -16 is added to sp and that new value is stored in sp. If you look at lines 4 and 13, and 10 and 17, stp and ldp are used for pushing to and popping from the stack frame.

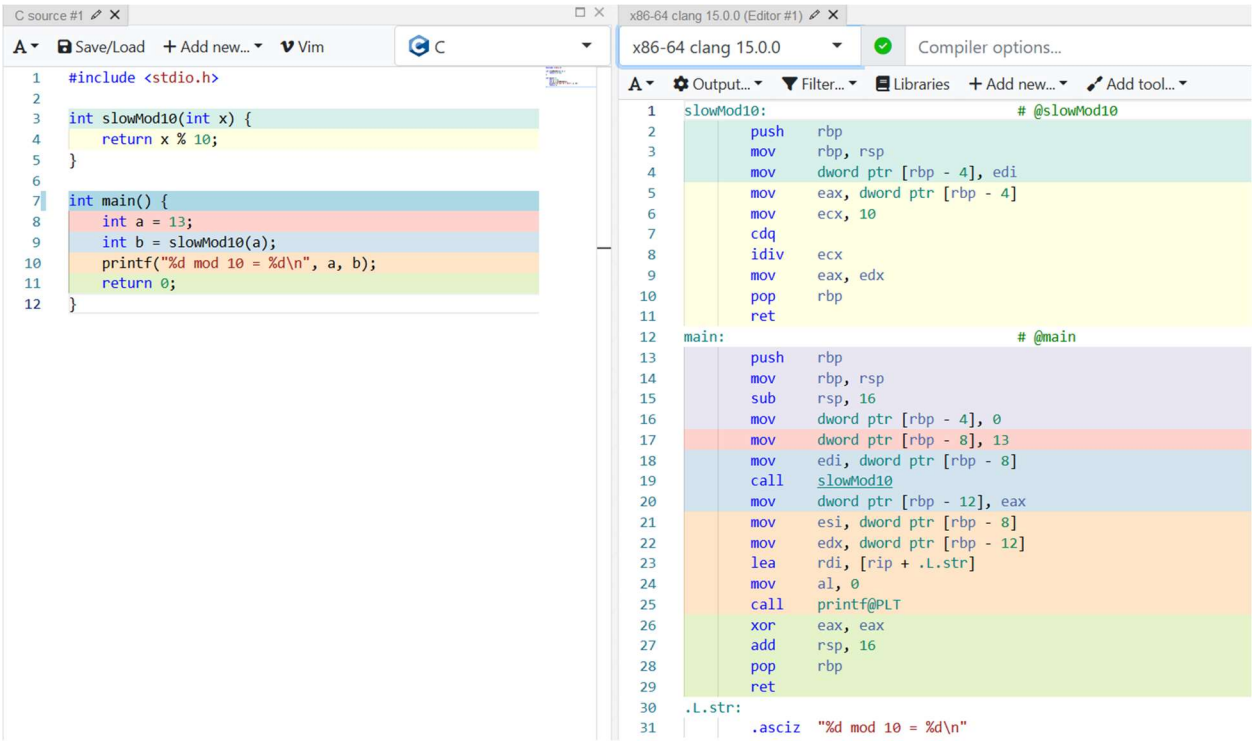
Representative function call with parameters code

```
#include <stdio.h>

int slowMod10(int x) {
    return x % 10;
}

int main() {
    int a = 13;
    int b = slowMod10(a);
    printf("%d mod 10 = %d\n", a, b);
    return 0;
}
```

Though there is no use in calling this function over using the modulo operator, it does provide a simple example of a function that takes a parameter.

CISC x86-64 Assembly





C source #1

Save/Load + Add new... Vim C

```c
#include <stdio.h>

int slowMod10(int x) {
    return x % 10;
}

int main() {
    int a = 13;
    int b = slowMod10(a);
    printf("%d mod 10 = %d\n", a, b);
    return 0;
}
```

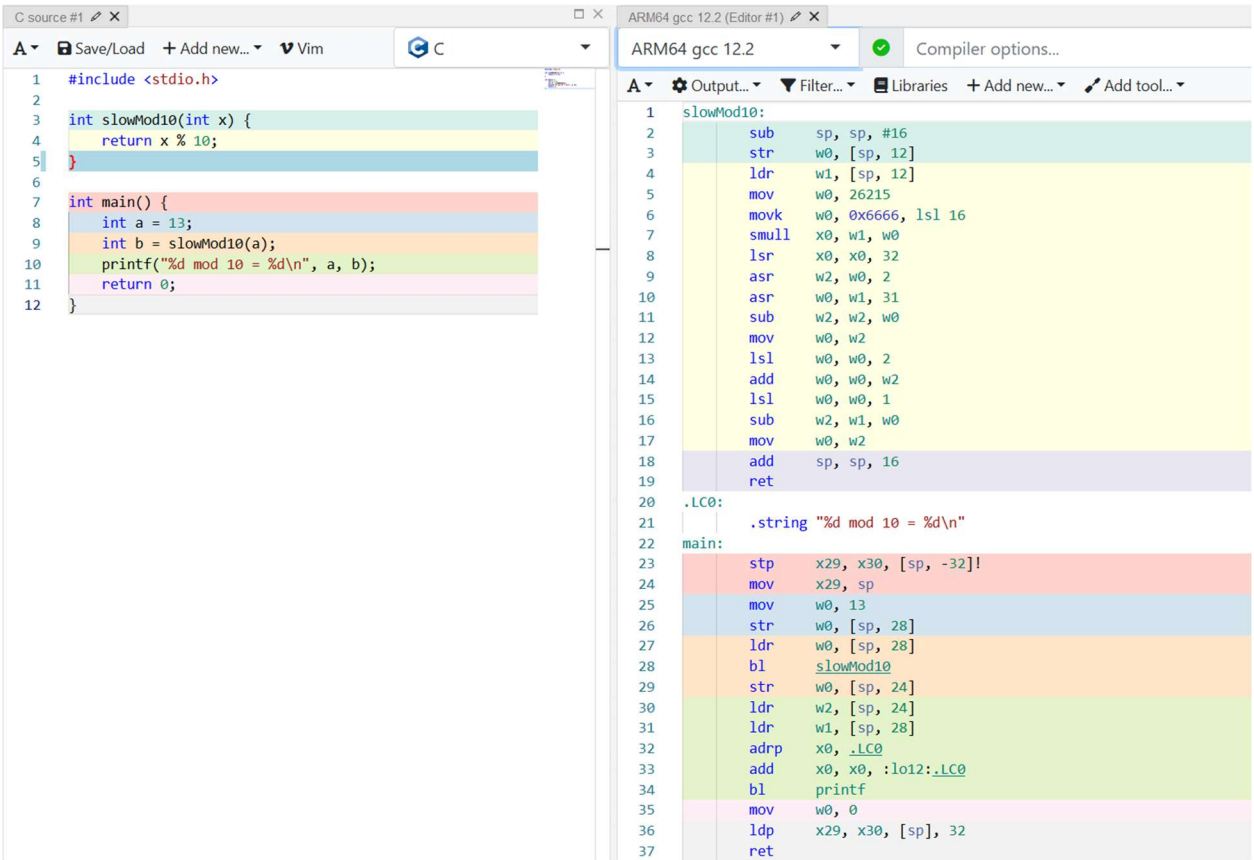
x86-64 clang 15.0.0 (Editor #1)

x86-64 clang 15.0.0 Compiler options...

Output... Filter... Libraries + Add new... Add tool...

```asm
slowMod10:                              # @slowMod10
        push    rbp
        mov     rbp, rsp
        mov     dword ptr [rbp - 4], edi
        mov     eax, dword ptr [rbp - 4]
        mov     ecx, 10
        cdq
        idiv    ecx
        mov     eax, edx
        pop     rbp
        ret
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], 0
        mov     dword ptr [rbp - 8], 13
        mov     edi, dword ptr [rbp - 8]
        call    slowMod10
        mov     dword ptr [rbp - 12], eax
        mov     esi, dword ptr [rbp - 8]
        mov     edx, dword ptr [rbp - 12]
        lea     rdi, [rip + .L.str]
        mov     al, 0
        call    printf@PLT
        xor     eax, eax
        add     rsp, 16
        pop     rbp
        ret
.L.str:
        .asciz  "%d mod 10 = %d\n"
```

RISC Assembly



C source #1

Save/Load + Add new... Vim C

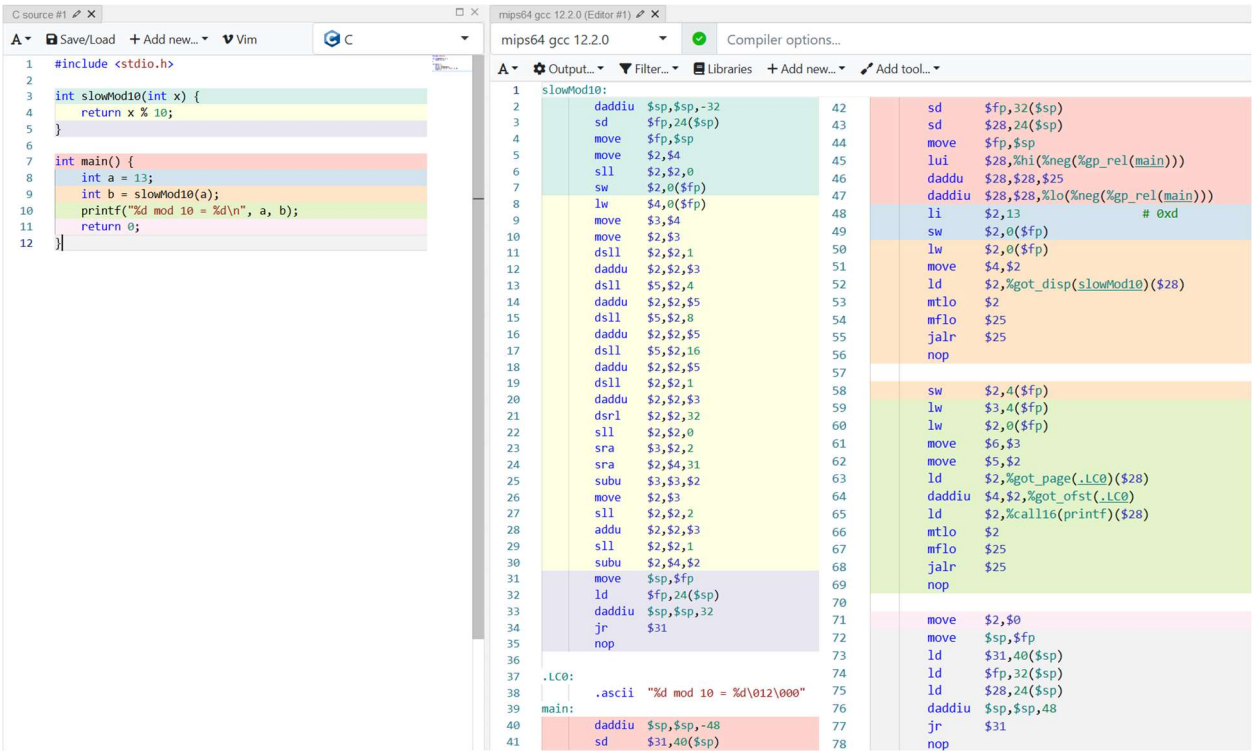
```c
#include <stdio.h>

int slowMod10(int x) {
    return x % 10;
}

int main() {
    int a = 13;
    int b = slowMod10(a);
    printf("%d mod 10 = %d\n", a, b);
    return 0;
}
```

ARM64 gcc 12.2 (Editor #1)

ARM64 gcc 12.2 Compiler options...

Output... Filter... Libraries + Add new... Add tool...

```asm
slowMod10:
        sub     sp, sp, #16
        str     w0, [sp, 12]
        ldr     w1, [sp, 12]
        mov     w0, 26215
        movk    w0, 0x6666, lsl 16
        smull   x0, w1, w0
        lsr     x0, x0, 32
        asr     w2, w0, 2
        asr     w0, w1, 31
        sub     w2, w2, w0
        mov     w0, w2
        lsl     w0, w0, 2
        add     w0, w0, w2
        lsl     w0, w0, 1
        sub     w2, w1, w0
        mov     w0, w2
        add     sp, sp, 16
        ret
.LC0:
        .string "%d mod 10 = %d\n"
main:
        stp     x29, x30, [sp, -32]!
        mov     x29, sp
        mov     w0, 13
        str     w0, [sp, 28]
        ldr     w0, [sp, 28]
        bl      slowMod10
        str     w0, [sp, 24]
        ldr     w2, [sp, 24]
        ldr     w1, [sp, 28]
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 32
        ret
```



Alright, this is fascinating. gcc's x86-64 assembly uses shifting, a few additions and subtractions, and one multiply to avoid using the div instruction because it takes 20+ instructions (reference: StackOverflow). In fact, only clang's implementation uses division to perform the modulo operation.
Also, there is, in general, more pushing and popping going on because of the use of a parameter in the function.