

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 screenshot shows a debugger interface with four windows:

- C source #1:** Displays the C code provided above.
- x86-64 gcc 12.2 (Editor #1):** Displays the assembly output for gcc 12.2. The assembly code includes:

```
.LC0:
    .string "You don't need to pass any arguments!"
main:
    push    rbp
    mov     rbp, rsp
    sub    rsp, 16
    mov    DWORD PTR [rbp-4], edi
    mov    QWORD PTR [rbp-16], rsi
    cmp   DWORD PTR [rbp-4], 1
    jle    .L2
    mov    edi, OFFSET FLAT:.LC0
    call   puts
.L2:
    mov    eax, 0
    leave
    ret
```
- x86-64 clang 15.0.0 (Editor #1):** Displays the assembly output for clang 15.0.0. The assembly code includes:

```
main:                                # @main
    push    rbp
    mov     rbp, rsp
    sub    rsp, 16
    →     mov    dword ptr [rbp - 4], 0
    mov    dword ptr [rbp - 8], edi
    mov    qword ptr [rbp - 16], rsi
    cmp   dword ptr [rbp - 8], 1
    jle    .LBB0_2
    lea    rdi, [rip + .L.str]
    mov    al, 0
    →     call   printf@PLT
.LBB0_2:
    →     xor    eax, eax
    add    rsp, 16
    pop    rbp
    ret
.L.str:
    .asciz  "You don't need to pass any arguments!\n"
```
- C source #1:** A second C source code window, identical to the first.

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

The screenshot shows a debugger interface with four windows:

- C source #1:** Contains the C code:

```

1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     if (argc > 1) {
5         printf("You don't need to pass any arguments!\n");
6     }
7     return 0;
8 }
```
- ARM64 gcc 12.2 (Editor #1):** Shows the ARM64 assembly output:

```

1 .LC0:
2     .string "You don't need to pass any arguments!"
3 main:
4     stp    x29, x30, [sp, -32]!
5     mov    x29, sp
6     str    w0, [sp, 28]
7     str    x1, [sp, 16]
8     ldr    w0, [sp, 28]
9     cmp    w0, 1
10    ble   .L2
11    adrp   x0, .LC0
12    add    x0, x0, :lo12:.LC0
13    bl     puts
14 .L2:
15    mov    w0, 0
16    ldp    x29, x30, [sp], 32
17    ret
```
- C source #1:** Contains the same C code as the first window.
- mips64 gcc 12.2.0 (Editor #1):** Shows the MIPS64 assembly output:

```

1 .LC0:
2     .ascii "You don't need to pass any arguments!\000"
3 main:
4     daddiu $sp,$sp,-48
5     sd      $31,40($sp)
6     sd      $fp,32($sp)
7     sd      $28,24($sp)
8     move   $fp,$sp
9     lui    $28,%hi(%neg(%gp_rel(main)))
10    daddu $28,$28,$25
11    daddiu $28,$28,%lo(%neg(%gp_rel(main)))
12    move   $2,$4
13    sd      $5,8($fp)
14    sll   $2,$2,0
15    sw      $2,0($fp)
16    lw      $2,0($fp)
17    slt   $2,$2,2
18    bne   $2,$0,.L2
19    nop
20
21    ld      $2,%got_page(.LC0)($28)
22    daddiu $4,$2,%got_ofst(.LC0)
23    ld      $2,%call16(puts)($28)
24    mtlo   $2
25    mflo   $25
26    jalr   $25
27    nop
28
29 .L2:
30    move   $2,$0
31    move   $sp,$fp
32    ld      $31,40($sp)
33    ld      $fp,32($sp)
34    ld      $28,24($sp)
35    daddiu $sp,$sp,48
36    jr      $31
37    nop
```

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

Compiler	Assembly (x86-64)
gcc 12.2	<pre>.LC0: .string "Wrong usage!" .LC1: .string "You did it right!" main: push rbp mov rbp, rsp sub rsp, 16 mov DWORD PTR [rbp-4], edi mov QWORD PTR [rbp-16], rsi cmp DWORD PTR [rbp-4], 1 jle .L2 mov edi, OFFSET FLAT:.LC0 call puts jmp .L3 .L2: mov edi, OFFSET FLAT:.LC1 call puts .L3: mov eax, 0 leave ret</pre>
clang 15.0.0	<pre>main: # @main push rbp mov rbp, rsp sub rsp, 16 mov dword ptr [rbp - 4], 0 mov dword ptr [rbp - 8], edi mov qword ptr [rbp - 16], rsi cmp dword ptr [rbp - 8], 1 jle .LBB0_2 lea rdi, [rip + .L.str] mov al, 0 call printf@PLT jmp .LBB0_3 .LBB0_2: lea rdi, [rip + .L.str.1] mov al, 0 call printf@PLT .LBB0_3: xor eax, eax add rsp, 16 pop rbp ret .L.str: .asciz "Wrong usage!\n" .L.str.1: .asciz "You did it right!\n"</pre>

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

The screenshot shows a debugger interface with three main windows:

- Top Left Window (C Code):**

```

1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     if (argc > 1) {
5         printf("Wrong usage!\n");
6     } else {
7         printf("You did it right!\n");
8     }
9     return 0;
10 }
```
- Top Right Window (ARM64 gcc 12.2):**

```

1 .LC0:    .string "Wrong usage!"
2 .LC1:    .string "You did it right!"
3 main:
4     stp    x29, x30, [sp, -32]!
5     mov    x29, sp
6     str    w0, [sp, 28]
7     str    x1, [sp, 16]
8     ldr    w0, [sp, 28]
9     cmp    w0, 1
10    ble   .L2
11    adrp   x0, .LC0
12    add    x0, x0, :lo12:.LC0
13    bl     puts
14    b     .L3
15
16 .L2:
17    adrp   x0, .LC1
18    add    x0, x0, :lo12:.LC1
19    bl     puts
20
21 .L3:
22    mov    w0, 0
23    ldp    x29, x30, [sp], 32
24    ret
```
- Middle Window (mips64 gcc 12.2.0):**

```

1 .LC0:    .ascii "Wrong usage!\000"
2 .LC1:    .ascii "You did it right!\000"
3 main:
4     daddiu $sp,$sp,-48
5     sd      $31,40($sp)
6     sd      $f9,12($sp)
7     sd      $28,24($sp)
8     move   $fp,$sp
9     lui     $28,%hi(%neg(%gp_rel(main)))
10    daddiu $28,$28,%lo(%neg(%gp_rel(main)))
11    daddiu $28,$28,%lo(%neg(%gp_rel(main)))
12    move   $2,$4
13    sd      $5,%0($fp)
14    sll    $2,$2,0
15    sw     $2,%0($fp)
16    lw     $2,%0($fp)
17    slt    $2,$2,2
18    bne   $2,$0,.L2
19    nop
20
21 .L2:
22    ld     $2,%got_page(.LC0)($28)
23    daddiu $4,$2,%got_ofst(.LC0)
24    ld     $2,%call16(puts)($28)
25    mtlo   $2
26    mflo   $25
27    jalr   $25
28    nop
29
30 .L3:
31    b     .L3
32    nop
33
34 .L2:
35    ld     $2,%got_page(.LC1)($28)
36    daddiu $4,$2,%got_ofst(.LC1)
37    ld     $2,%call16(puts)($28)
38    mtlo   $2
39    mflo   $25
40    jalr   $25
41    nop
42
43 .L3:
44    move   $2,$0
45    move   $sp,$fp
46    ld     $31,40($sp)
47    ldi   $31,32($sp)
48    ld     $28,24($sp)
49    daddiu $sp,$sp,48
50    jr     $31
51    nop
```

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

The screenshot shows a debugger interface with two windows. The left window is titled "C source #1" and displays the C code above. The right window is titled "x86-64 gcc 12.2 (Editor #1)" and displays the generated assembly code. The assembly code includes labels for strings (.LC0-.LC3), the main function, and various jump points (.L1-.L7). The assembly code corresponds to the logic in the C code, printing different messages based on the value of 'answer'.

```
C source #1 x
A- Save/Load + Add new... Vim
x86-64 gcc 12.2 (Editor #1) x
A- Output... Filter... Libraries + Add new... Add tool...
1 #include <stdio.h>
2
3 int main() {
4     int answer = 2;
5
6     switch (answer) {
7         case 1:
8             printf("I'm good!\n");
9             break;
10        case 2:
11            printf("HELP\n");
12            break;
13        case 3:
14            printf("Huh?\n");
15            break;
16        default:
17            printf("You REALLY don't know what you're doing\n");
18            return 1;
19    }
20    return 0;
21 }
```

```
x86-64 gcc 12.2 x
A- Output... Filter... Libraries + Add new... Add tool...
1 .LC0: .string "I'm good!"
2 .LC1: .string "HELP"
3 .LC2: .string "Huh?"
4 .LC3: .string "You REALLY don't know what you're doing"
5 main:
6     push rbp
7     mov rbp, rsp
8     sub rsp, 16
9     mov DWORD PTR [rbp-4], 2
10    cmp DWORD PTR [rbp-4], 3
11    je .L2
12    cmp DWORD PTR [rbp-4], 3
13    jg .L3
14    cmp DWORD PTR [rbp-4], 1
15    je .L4
16    cmp DWORD PTR [rbp-4], 2
17    je .L5
18    jmp .L3
19 .L4:
20     mov edi, OFFSET FLAT:.LC0
21     call puts
22     jmp .L6
23 .L5:
24     mov edi, OFFSET FLAT:.LC1
25     call puts
26     jmp .L6
27 .L2:
28     mov edi, OFFSET FLAT:.LC2
29     call puts
30     jmp .L6
31 .L3:
32     mov edi, OFFSET FLAT:.LC3
33     call puts
34     mov eax, 1
35     jmp .L7
36 .L6:
37     mov eax, 0
38     leave
39     ret
40 .L7:
```

The screenshot shows two windows side-by-side. On the left is a C source code editor with the following code:

```

1 #include <stdio.h>
2
3 int main() {
4     int answer = 2;
5
6     switch (answer) {
7         case 1:
8             printf("I'm good!\n");
9             break;
10        case 2:
11            printf("HELP\n");
12            break;
13        case 3:
14            printf("Huh?\n");
15            break;
16        default:
17            printf("You REALLY don't know what you're doing!\n");
18        }
19    return 1;
20 }
21

```

On the right is an assembly output window titled "x86-64 clang 15.0.0 (Editor #1)". The assembly code is as follows:

```

1 main:                                # @main
2     push rbp
3     mov rbp, rsp
4     sub rsp, 16
5     mov dword ptr [rbp - 4], 0
6     mov dword ptr [rbp - 8], 2
7     mov eax, dword ptr [rbp - 8]
8     mov dword ptr [rbp - 12], eax      # 4-byte Spill
9     sub eax, 1
10    je .LBB0_1
11    jmp .LBB0_2
12 .LBB0_2:
13    mov eax, dword ptr [rbp - 12]    # 4-byte Reload
14    sub eax, 2
15    je .LBB0_2
16    jmp .LBB0_8
17 .LBB0_8:
18    mov eax, dword ptr [rbp - 12]    # 4-byte Reload
19    sub eax, 3
20    je .LBB0_3
21    jmp .LBB0_4
22 .LBB0_4:
23    lea rdi, [rip + .L.str]
24    mov al, 0
25    call printf@PLT
26    jmp .LBB0_5
27 .LBB0_5:
28    lea rdi, [rip + .L.str.1]
29    mov al, 0
30    call printf@PLT
31    jmp .LBB0_5
32 .LBB0_3:
33    lea rdi, [rip + .L.str.2]
34    mov al, 0
35    call printf@PLT
36    jmp .LBB0_5
37 .LBB0_4:
38    lea rdi, [rip + .L.str.3]
39    mov al, 0
40    call printf@PLT
41    mov dword ptr [rbp - 4], 1
42    jmp .LBB0_6
43 .LBB0_6:
44    mov dword ptr [rbp - 4], 0
45 .LBB0_0:
46    mov eax, dword ptr [rbp - 4]
47    add rsp, 16
48    pop rbp
49    ret
50 .L.str:
51     .asciz "I'm good!\n"
52 .L.str.1:
53     .asciz "HELP\n"
54 .L.str.2:

```

* 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

The image shows two side-by-side assembly code editors. The left editor displays ARM64 assembly generated by gcc 12.2.0, and the right editor displays MIPS64 assembly generated by the same compiler. Both assemblies are for the same C program, which includes a switch statement with four cases (1, 2, 3, and default) and prints different strings based on the case value.

ARM64 Assembly (Left):

```

1 .LC0: .string "I'm good!"
2 .LC1: .string "HELP"
3 .LC2: .string "Huh?"
4 .LC3: .string "You REALLY don't know what you're doing"
5 main:
6     stp x29, x30, [sp, -32]!
7     mov x29, sp
8     mov w0, 2
9     str w0, [sp, 28]
10    ldr w0, [sp, 28]
11    cmp w0, 3
12    bne .L2
13    mov w0, 1
14    str w0, [sp, 28]
15    ldr w0, [sp, 28]
16    cmp w0, 3
17    bgt .L3
18    ldr w0, [sp, 28]
19    cmp w0, 1
20    beq .L4
21    ldr w0, [sp, 28]
22    cmp w0, 2
23    beq .L5
24    ldr w0, [sp, 28]
25    cmp w0, 2
26    beq .L6
27    ldr w0, [sp, 28]
28    addp x0, .LC0
29    add x0, x0, :lo12:.LC0
30    bl puts
31    b .L6
32    ldr w0, .LC1
33    add x0, x0, :lo12:.LC1
34    bl puts
35    b .L6
36    ldr w0, .LC2
37    add x0, .LC2
38    add x0, x0, :lo12:.LC2
39    bl puts
40    b .L6
41    ldr w0, .LC3
42    add x0, .LC3
43    add x0, x0, :lo12:.LC3
44    bl puts
45    mov w0, 1
46    b .L7
47    mov w0, 0
48    .L6:
49    mov w0, 0
50    .L7:
51    ldp x29, x30, [sp], 32
52    ret

```

MIPS64 Assembly (Right):

```

1 .LC0: .ascii "I'm good!\000"
2 .LC1: .ascii "HELP\000"
3 .LC2: .ascii "Huh?\000"
4 .LC3: .ascii "You REALLY don't know what you're doing\000"
5 main:
6     addiu $sp,$sp,-48
7     sd $31,40($sp)
8     sd $fp,32($sp)
9     sd $28,24($sp)
10    move $fp,$sp
11    li $28,0x1(%neg(%gp_rel(main)))
12    addiu $28,$28,%lo(%neg(%gp_rel(main)))
13    addiu $28,$28,%lo(%neg(%gp_rel(main)))
14    li $2,$2
15    sw $2,0($fp)
16    lw $2,0($fp)
17    li $3,3
18    beq $2,$3,.L2
19    nop
20    lw $2,0($fp)
21    slt $2,$2,4
22    beq $2,$0,.L3
23    nop
24    lw $2,0($fp)
25    slt $2,$2,4
26    beq $2,$0,.L3
27    nop
28    lw $2,0($fp)
29    slt $2,$2,4
30    beq $2,$0,.L3
31    li $3,1
32    beq $2,$3,.L4
33    nop
34    lw $2,0($fp)
35    li $3,2
36    beq $2,$3,.L5
37    nop
38    lw $2,0($fp)
39    li $3,2
40    beq $2,$3,.L5
41    nop
42    b .L3
43    nop
44    ld $3,%got_page(.LC0)($28)
45    addiu $4,$2,%got_ofst(.LC0)
46    ld $2,%call16(puts)($28)
47    mtlo $2
48    mflo $25
49    jalr $25
50    nop
51    b .L6
52    nop

```

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;
}
```

CISC x86-64 Assembly

The screenshot shows a debugger interface with three windows side-by-side:

- C source #1:** Displays the C code provided above.
- x86-64 gcc 12.2 (Editor #1):** Displays the assembly output for the gcc compiler. The assembly code is color-coded by section and mnemonic. It includes labels like .LC0, .LC1, and main, and instructions like push, mov, sub, and jmp.
- x86-64 clang 15.0.0 (Editor #1):** Displays the assembly output for the clang compiler. This assembly is more detailed, showing the inner loop header (.LBB0_1) and the loop body (.LBB0_4). It also includes labels for string literals (.L.str and .L.str.1) and uses the printf@PLT symbol.

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.

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

The screenshot shows a debugger interface with two panes. The left pane is titled "C source #1" and contains the following C code: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 }The right pane is titled "ARM64 gcc 12.2 (Editor #1)" and displays the generated assembly code:1 .LC0:
2 .string "Looping.. "
3 .LC1:
4 .string "\nDone"
5 main:
6 stp x29, x30, [sp, -32]!
7 mov x29, sp
8 str wZR, [sp, 28]
9 b .L2
10 .L3:
11 adrp x0, .LC0
12 add x0, x0, :lo12:.LC0
13 bl printf
14 ldr w0, [sp, 28]
15 add w0, w0, 1
16 str w0, [sp, 28]
17 .L2:
18 ldr w0, [sp, 28]
19 cmp w0, 2
20 ble .L3
21 adrp x0, .LC1
22 add x0, x0, :lo12:.LC1
23 bl puts
24 mov w0, 0
25 ldp x29, x30, [sp], 32
26 retThe assembly code uses labels (.LC0, .LC1, .L2, .L3) and memory operations (stp, mov, str, bl, adrp, add, ldr, cmp, ble, mov, ldp) typical of ARM64 assembly language.

```

C source #1 ▾ X
A ▾ 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) ▾ X
A ▾ Output... Filter... Libraries + Add new... Add tool...
Compiler options...
1 .LC0:
2     .ascii "Looping.. \000"
3 .LC1:
4     .ascii "\012Done\000"
5
6 main:
7     addiu $sp,$sp,-48
8     sd    $31,40($sp)
9     sd    $fp,32($sp)
10    sd   $28,24($sp)
11    move  $fp,$sp
12    lui   $28,%hi(%neg(%gp_rel(main)))
13    adddu $28,$28,$25
14    addiu $28,$28,%lo(%neg(%gp_rel(main)))
15    sw   $0,0($fp)
16    b    .L2
17    nop
18
19 .L3:
20     ld   $2,%got_page(.LC0)($28)
21     addiu $4,$2,%got_ofst(.LC0)
22     ld   $2,%call16(sprintf)($28)
23     mtlo $2
24     mflo $25
25     jalr $25
26     nop
27
28 .L2:
29     lw   $2,0($fp)
30     addiu $2,$2,1
31     sw   $2,0($fp)
32
33 .L1:
34     lw   $2,0($fp)
35     slt  $2,$2,3
36     bne $2,$0,.L3
37     nop
38
39     ld   $2,%got_page(.LC1)($28)
40     addiu $4,$2,%got_ofst(.LC1)
41     ld   $2,%call16(puts)($28)
42     mtlo $2
43     mflo $25
44     jalr $25
45     nop
46
47     move $2,$0
48     move $sp,$fp
49     ld   $31,40($sp)
50     ld   $fp,32($sp)
51     addiu $sp,$sp,48
52     jr   $31
53     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;
}
```

CISC x86-64 Assembly

The screenshot displays two assembly windows side-by-side, each showing the assembly output for a different compiler. The left window contains the original C source code, and the right window contains the assembly output.

Assembly Output (x86-64 gcc 12.2):

```
.LC0:
    .string "%d\n"
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR [rbp-4], 5
    jmp     .L2
.L3:
    mov     eax, 60
    cdq
    idiv   DWORD PTR [rbp-4]
    mov     esi, eax
    mov     edi, OFFSET FLAT:.LC0
    mov     eax, 0
    call    printf
.L2:
    cmp     DWORD PTR [rbp-4], 0
    jne     .L3
    mov     eax, 0
    leave
    ret
```

Assembly Output (x86-64 clang 15.0.0):

```
main:          # @main
    push    rbp
    mov     rbp, rsp
    sub     rbp, 16
    mov     dword ptr [rbp - 4], 0
    mov     dword ptr [rbp - 8], 5
.LBB0_1:      # =>This Inner Loop Header: Depth=1
    cmp     dword ptr [rbp - 8], 0
    je     .LBB0_3
    mov     eax, 60
    cdq
    idiv   dword ptr [rbp - 8]
    mov     esi, eax
    lea     rdi, [rip + .L.str]
    mov     al, 0
    call    printf@PLT
    jmp     .LBB0_1
.LBB0_3:
    xor     eax, eax
    add     rsp, 16
    pop     rbp
    ret
.L.str:       .asciz  "%d\n"
```

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.

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

RISC Assembly

Two columns of assembly code output from different compilers for the same C program.

Top Column (ARM64 gcc 12.2.2):

```

1 #include <stdio.h>
2
3 int main() {
4     int n = 5;
5     while (n != 0) {
6         printf("%d\n", 60/n);
7     }
8     return 0;
9 }
```

```

1 .LC0:
2     .string "%d\n"
3 main:
4     stp    x29, x30, [sp, -32]!
5     mov    x29, sp
6     mov    w0, 5
7     str    w0, [sp, 28]
8     b     .L2
9
10    .L3:
11        mov    w1, 60
12        ldr    w0, [sp, 28]
13        sdiv   w0, w1, w0
14        mov    w1, w0
15        adrp   x0, .LC0
16        add    x0, x0, :lo12:.LC0
17        bl     printf
18
19    .L2:
20        ldr    w0, [sp, 28]
21        cmp    w0, 0
22        bne   .L3
23        mov    w0, 0
24        ldp    x29, x30, [sp], 32
25        ret
```

Bottom Column (mips64 gcc 12.2.0):

```

1 #include <stdio.h>
2
3 int main() {
4     int n = 5;
5     while (n != 0) {
6         printf("%d\n", 60/n);
7     }
8     return 0;
9 }
```

```

1 .LC0:
2     .ascii  "%d\012\000"
3 main:
4     daddiu $sp,$sp,-48
5     sd      $31,40($sp)
6     sd      $fp,32($sp)
7     sd      $28,24($sp)
8     move   $fp,$sp
9     lui    $28,%hi(%neg(%gp_rel(main)))
10    daddiu $28,$28,$25
11    daddiu $28,$28,%lo(%neg(%gp_rel(main)))
12    li     $2,5          # 0x5
13    sw     $2,0($fp)
14    b     .L2
15    nop
16
17 .L3:
18    li     $3,60          # 0x3c
19    lw     $2,0($fp)
20    div
21    teq   $2,$0,7
22    mfhi  $2
23    mflo  $2
24    move   $5,$2
25    ld     $2,%got_page(.LC0)($28)
26    daddiu $4,$2,%got_ofst(.LC0)
27    ld     $2,%call16(sprintf)($28)
28    mtlo  $2
29    mflo  $25
30    jalr  $25
31    nop
32
33 .L2:
34    lw     $2,0($fp)
35    bne   $2,$0,.L3
36    nop
37
38    move   $2,$0
39    move   $sp,$fp
40    ld     $31,40($sp)
41    ld     $fp,32($sp)
42    ld     $28,24($sp)
43    daddiu $sp,$sp,48
44    jr     $31
45    nop
```

A red arrow points to the instruction at line 22 in the mips64 assembly, which is `mfhi $2`.

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

The screenshot shows a debugger interface with four panes:

- C source #1:** Contains the C code provided above.
- x86-64 gcc 12.2 (Editor #1):** Shows the assembly output for the gcc 12.2 compiler. It includes sections for .LC0 and .LC1 containing strings, and the main function which performs the required operations.
- x86-64 clang 15.0.0 (Editor #1):** Shows the assembly output for the clang 15.0.0 compiler. It uses a different calling convention and includes sections for .L.str and .L.str.1 containing strings.
- x86-64 clang 15.0.0 (Editor #1):** This pane is identical to the previous one but has annotations overlaid on it. Annotations include:
 - A green box highlights the section from line 1 to line 5 of the C code.
 - A yellow box highlights the section from line 6 to line 10.
 - A red box highlights the section from line 11 to line 15.
 - An orange box highlights the section from line 16 to line 17.
 - Annotations for the assembly code include:
 - "# @main" is placed next to the first instruction of the main function.
 - "# @LC0" is placed next to the first instruction of the .LC0 section.
 - "# @LC1" is placed next to the first instruction of the .LC1 section.
 - "# @L.str" is placed next to the first instruction of the .L.str section.
 - "# @L.str.1" is placed next to the first instruction of the .L.str.1 section.

RISC Assembly

The screenshot shows a debugger interface with four windows:

- C source #1:** Contains the C source code for a program that prints two messages and sleeps for 3 seconds.
- ARM64 gcc 12.2:** Shows the generated ARM64 assembly code. It includes sections for strings (.LC0 and .LC1), the main function, and various memory operations like stp, mov, adrp, add, bl, puts, and sleep.
- C source #2:** Contains the same C source code as the first window.
- mips64 gcc 12.2.0:** Shows the generated MIPS assembly code. It uses different instruction sets and includes sections for strings (.LC0 and .LC1), the main function, and various MIPS-specific instructions like addiu, sd, move, lui, addiu, ld, daddiu, mtllo, mflo, jalr, and nop.

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

The screenshot shows a debugger interface with four panes:

- C source #1:** Contains the C code provided above.
- x86-64 gcc 12.2 (Editor #1):** Shows the assembly output for the gcc 12.2 compiler. It includes a string label `.LC0:` and a `simpleFunction:` label. The assembly code uses `push rbp`, `mov rbp, rsp`, `mov edi, OFFSET FLAT:.LC0`, `call puts`, `nop`, `pop rbp`, and `ret`.
- x86-64 clang 15.0.0 (Editor #1):** Shows the assembly output for the clang 15.0.0 compiler. It includes a `simpleFunction:` label and a `main:` label. The assembly code uses `push rbp`, `mov rbp, rsp`, `mov eax, 0`, `call simpleFunction`, `mov eax, 0`, `pop rbp`, and `ret`.
- x86-64 clang 15.0.0 (Editor #1):** This pane shows the same clang assembly output as the previous one, but with annotations. An arrow points to the `lea rdi, [rip + .L.str]` instruction in the `simpleFunction:` section, indicating the use of the `lea` instruction instead of `mov`. Another arrow points to the `nop` instruction in the first clang assembly output, highlighting its use.

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

The screenshot shows a multi-pane assembly debugger interface. On the left, there are two panes for C source code:

- Top Left:** C source code for ARM64 gcc 12.2 (Editor #1). It contains a simple function `simpleFunction` and a `main` function.
- Bottom Left:** C source code for mips64 gcc 12.2.0 (Editor #1), showing the same code structure.

On the right, there are three panes for assembly output:

- Top Right:** ARM64 gcc 12.2 (Editor #1) showing the assembly for the ARM64 code. It includes a section header `.LC0` and labels for `simpleFunction` and `main`. The assembly uses the S register (sp) for the stack pointer.
- Middle Right:** mips64 gcc 12.2.0 (Editor #1) showing the assembly for the Mips64 code. It includes a section header `.LC0` and labels for `simpleFunction` and `main`. The assembly uses the \$sp register for the stack pointer.
- Bottom Right:** Another assembly pane for a different architecture, showing the assembly for the same code. This pane also includes a section header `.LC0` and labels for `simpleFunction` and `main`.

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

The screenshot shows a debugger interface with two panes. The left pane displays the C source code with syntax highlighting. The right pane displays the generated assembly code. The assembly code uses standard x86-64 conventions, including the use of rbp, rsp, and the stack for temporary storage.

```
C source #1 x C
A Save/Load + Add new... Vim
x86-64 gcc 12.2 (Editor #1) x
Compiler options...
A Output... Filter... Libraries + Add new... Add tool...
1 #include <stdio.h>
2
3 int slowMod10(int x) {
4     return x % 10;
5 }
6
7 int main() {
8     int a = 13;
9     int b = slowMod10(a);
10    printf("%d mod 10 = %d\n", a, b);
11    return 0;
12 }

x86-64 gcc 12.2 (Editor #1) x
Compiler options...
A Output... Filter... Libraries + Add new... Add tool...
1 push rbp
2 mov rbp, rsp
3 mov DWORD PTR [rbp-4], edi
4 mov ecx, DWORD PTR [rbp-4]
5 movsx rax, ecx
6 imul rax, rax, 1717986919
7 shr rax, 32
8 mov edx, eax
9 sar edx, 2
10 mov eax, ecx
11 sar eax, 31
12 sub edx, eax
13 mov eax, edx
14 sal eax, 2
15 add eax, edx
16 add eax, eax
17 sub ecx, eax
18 mov edx, ecx
19 mov eax, edx
20 pop rbp
21 ret
22 .LC0:
23 .string "%d mod 10 = %d\n"
24 main:
25 push rbp
26 mov rbp, rsp
27 sub rbp, 16
28 mov DWORD PTR [rbp-4], 13
29 mov eax, DWORD PTR [rbp-4]
30 mov edi, eax
31 mov eax, edi
32 call slowMod10
33 mov DWORD PTR [rbp-8], eax
34 mov edx, DWORD PTR [rbp-8]
35 mov eax, DWORD PTR [rbp-4]
36 mov esi, eax
37 mov edi, OFFSET FLAT:.LC0
38 mov eax, 0
39 call printf
40 mov eax, 0
41 leave
42 ret
```

C source #1

```

1 #include <stdio.h>
2
3 int slowMod10(int x) {
4     return x % 10;
5 }
6
7 int main() {
8     int a = 13;
9     int b = slowMod10(a);
10    printf("%d mod 10 = %d\n", a, b);
11    return 0;
12 }
```

x86-64 clang 15.0.0 (Editor #1)

```

1 slowMod10:                                # @slowMod10
2     push  rbp
3     mov   rbp, rsp
4     mov   dword ptr [rbp - 4], edi
5     mov   eax, dword ptr [rbp - 4]
6     mov   ecx, 10
7     cdq
8     idiv  ecx
9     mov   eax, edx
10    pop   rbp
11    ret
12 main:                                     # @main
13    push  rbp
14    mov   rbp, rsp
15    sub   rbp, 16
16    mov   dword ptr [rbp - 4], 0
17    mov   dword ptr [rbp - 8], 13
18    mov   edi, dword ptr [rbp - 8]
19    call  slowMod10
20    mov   dword ptr [rbp - 12], eax
21    mov   esi, dword ptr [rbp - 8]
22    mov   edx, dword ptr [rbp - 12]
23    lea   rdi, [rip + .L.str]
24    mov   al, 0
25    call  printf@PLT
26    xor   eax, eax
27    add   rsp, 16
28    pop   rbp
29    ret
30 .L.str:
31     .asciz "%d mod 10 = %d\n"
```

RISC Assembly

C source #1

```

1 #include <stdio.h>
2
3 int slowMod10(int x) {
4     return x % 10;
5 }
6
7 int main() {
8     int a = 13;
9     int b = slowMod10(a);
10    printf("%d mod 10 = %d\n", a, b);
11    return 0;
12 }
```

ARM64 gcc 12.2 (Editor #1)

```

1 slowMod10:
2     sub   sp, sp, #16
3     str   w0, [sp, 12]
4     ldr   w1, [sp, 12]
5     mov   w0, 26215
6     movk  w0, 0x6666, lsl 16
7     smull x0, w1, w0
8     lsr   x0, x0, 32
9     asr   w2, w0, 2
10    asr   w0, w1, 31
11    sub   w2, w2, w0
12    mov   w0, w2
13    lsl   w0, w0, 2
14    add   w0, w0, w2
15    lsl   w0, w0, 1
16    sub   w2, w1, w0
17    mov   w0, w2
18    add   sp, sp, 16
19    ret
20 .LC0:
21     .string "%d mod 10 = %d\n"
22 main:
23     stp   x29, x30, [sp, -32]!
24     mov   x29, sp
25     mov   w0, 13
26     str   w0, [sp, 28]
27     ldr   w0, [sp, 28]
28     bl    slowMod10
29     str   w0, [sp, 24]
30     ldr   w2, [sp, 24]
31     ldr   w1, [sp, 28]
32     adrp  x0, .LC0
33     add   x0, x0, :lo12:.LC0
34     bl    printf
35     mov   w0, 0
36     ldp   x29, x30, [sp], 32
37     ret
```

The screenshot shows two windows side-by-side. The left window is a C source code editor with the title "C source #1". It contains the following code:

```

1 #include <stdio.h>
2
3 int slowMod10(int x) {
4     return x % 10;
5 }
6
7 int main() {
8     int a = 13;
9     int b = slowMod10(a);
10    printf("%d mod 10 = %d\n", a, b);
11    return 0;
12 }

```

The right window is an assembly editor with the title "mips64 gcc 12.2.0 (Editor #1)". It displays the generated assembly code:

```

1 slowMod10:
2     addiu   $sp,$sp,-32      42
3     sd      $fp,24($sp)      43
4     move    $fp,$sp          44
5     move    $2,$4             45
6     sll     $2,$2,0            46
7     sw      $2,0($fp)         47
8     lw      $4,0($fp)          48
9     move    $3,$4             49
10    move   $2,$3             50
11    dsll   $2,$2,1            51
12    addu   $2,$2,$3           52
13    dsll   $5,$2,4            53
14    addu   $2,$2,$5           54
15    dsll   $5,$2,8            55
16    addu   $2,$2,$5           56
17    dsll   $5,$2,16           57
18    addu   $2,$2,$5           58
19    dsll   $2,$2,1            59
20    addu   $2,$2,$3           60
21    dsrl   $2,$2,32           61
22    sll    $2,$2,0             62
23    sra    $3,$2,2             63
24    sra    $2,$4,31            64
25    subu  $3,$3,$2            65
26    move   $2,$3             66
27    sll    $2,$2,2             67
28    addu   $2,$2,$3           68
29    sll    $2,$2,1             69
30    subu  $2,$4,$2            70
31    move   $sp,$fp            71
32    ld     $fp,24($sp)         72
33    addiu $sp,$sp,32          73
34    jr     $31                74
35    nop                            75
36
37 .LC0:
38     .ascii  "%d mod 10 = %d\012\000" 76
39 main:
40     addiu $sp,$sp,-48          77
41     sd      $31,40($sp)        78

```

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.