

# Godbolt Compiler Explorer

December 2024

# Overview

- What is Compiler Explorer?
- Features
- Other Tools
- Summary

What is **Compiler Explorer**?

# What is Compiler Explorer?



[github.com/mattgodbolt](https://github.com/mattgodbolt)

- Created by Matt Godbolt
  - Former Google developer
  - Worked in the video game and Quant Trading industry
  - Unix Geek

# IDEA?

- Real time C/C++ Disassembly
  - Start in 2012
  - Just for C and C++ code
  - Simple Terminal commands
  - Only GCC on X86 machine
  - Grew in years

```

1 #include <stdio.h>
2
3 int main(void) {
4     printf("hello\n");
5     for (int i = 0; i < 5; i++) {
6         printf("counter: %d\n", i);
7     }
8 }
9

```

- Compile, Assembly, Print

watch gcc -S run.c -o -

When Started In 2012

```

Every 2.0s: gcc -S run.c -o -

.file "run.c"
.text
.section .rodata
.LC8:
.string "hello"
.LC1:
.string "counter: %d\n"
.text
.globl main
.type main, @function
main:
.LF88:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
leaq .LC8(%rip), %rax
movq %rax, %rdi
call puts@PLT
movl $0, -4(%rbp)
jmp .L2

.L3:
movl -4(%rbp), %eax
movl %eax, %s1
leaq .LC1(%rip), %rax
movq %rax, %rdi
movl $0, %eax
call printf@PLT
addl $1, -4(%rbp)

.L2:
cmpl $4, -4(%rbp)
jle .L3
movl $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFEB:
.size main, .-main
.ident "GCC: (Debian 14.2.0-8) 14.2.0"
.section .note.GNU-stack,"",@progbits

```

# New Version



- Web Application
  - Godbolt.org
  - Many Features
  - Extensibility
  - Compiler Testing Platform
  - Education Playground

Now In 2024

godbolt.org

The image shows a screenshot of the Compiler Explorer web application. The interface is split into two main panes. The left pane, titled 'C++ source', contains the following code:

```
1 // type your code here, or load an example.  
2 int square(int x) {  
3     return x * x;  
4 }  
5
```

The right pane, titled 'Disassembly', shows the assembly code for the 'square' function, compiled for x86\_64 using gcc 10.2. The assembly is as follows:

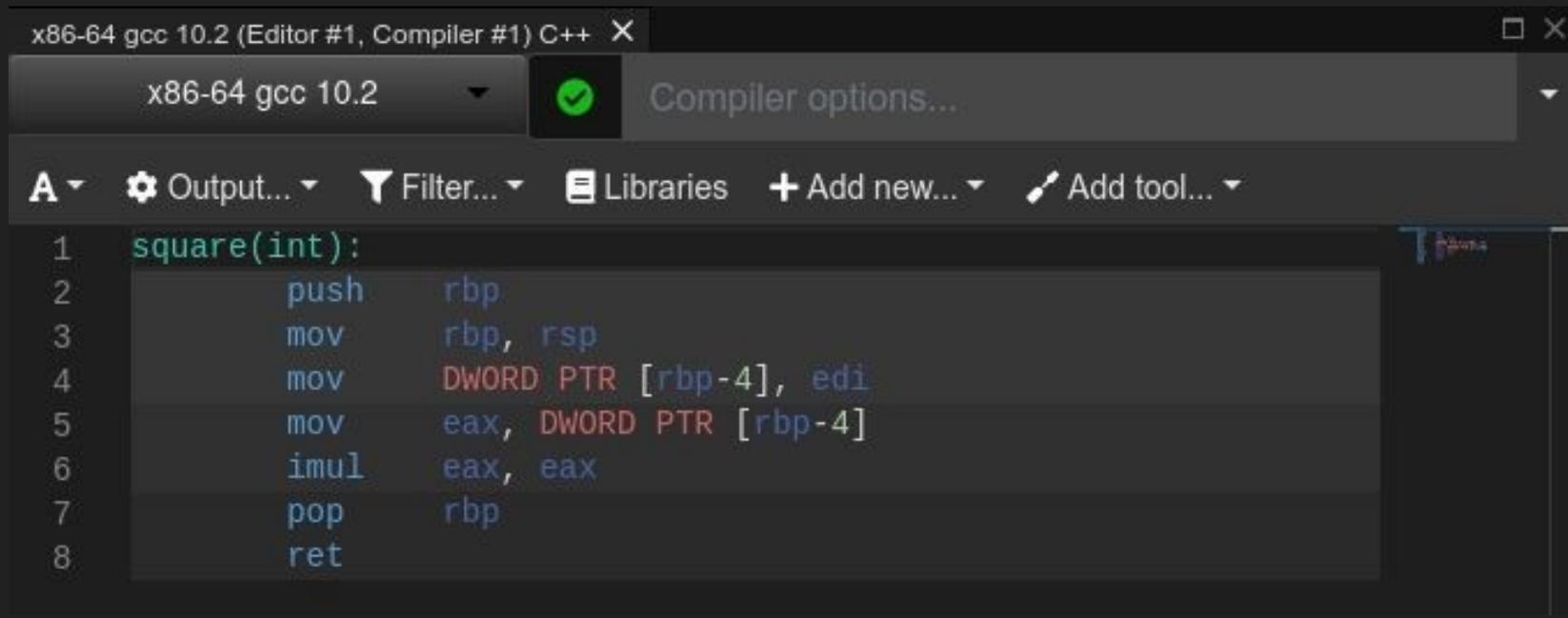
```
1 square(int):  
2     push    rbp  
3     mov     rbp, rsp  
4     mov     %eax, PTR [rbp+0], %eax  
5     mov     %eax, %eax  
6     and     %eax, %eax  
7     pop     rbp  
8     ret
```

The interface also includes a top navigation bar with 'Compiler Explorer' and 'Add...' buttons, and a right sidebar with 'Compiler options' and 'Output' sections.

C++ Source

Disassembly

# Right side: Disassembly



The screenshot shows a compiler window titled "x86-64 gcc 10.2 (Editor #1, Compiler #1) C++". The window has a menu bar with "x86-64 gcc 10.2" and "Compiler options...". Below the menu bar is a toolbar with icons for "Output...", "Filter...", "Libraries", "Add new...", and "Add tool...". The main area displays the disassembly of a C++ function named "square(int)". The disassembly consists of eight lines of assembly code, each with a line number on the left.

```
1 square(int):  
2     push    rbp  
3     mov     rbp, rsp  
4     mov     DWORD PTR [rbp-4], edi  
5     mov     eax, DWORD PTR [rbp-4]  
6     imul   eax, eax  
7     pop     rbp  
8     ret
```

X86-64 gcc 10.2 compiler

# Right side: Disassembly

The image shows two side-by-side code editors. The left editor is titled 'armv7-a clang 19.1.0 (Editor #1)' and shows the assembly for a 'square(int)' function on ARM7. The right editor is titled 'mips gcc 14.2.0 (Editor #1)' and shows the assembly for the same function on MIPS. Both functions calculate the square of an integer by pushing it onto the stack, multiplying it by itself, and then popping it back.

```
armv7-a clang 19.1.0 (Editor #1)
armv7-a clang 19.1.0  Compiler options...

1 square(int):
2     sub    sp, sp, #4
3     str    r0, [sp]
4     ldr    r0, [sp]
5     ldr    r1, [sp]
6     mul    r0, r0, r1
7     add    sp, sp, #4
8     bx    lr

mips gcc 14.2.0 (Editor #1)
mips gcc 14.2.0  Compiler options...

1 square(int):
2     addiu  $sp, $sp, -8
3     sw    $fp, 4($sp)
4     move  $fp, $sp
5     sw    $4, 8($fp)
6     lw    $2, 8($fp)
7     nop
8     mult  $2, $2
9     mflo  $2
10    move  $sp, $fp
11    lw    $fp, 4($sp)
12    addiu $sp, $sp, 8
13    jr    $31
14    nop
```

arm7 and MIPS

# Customization

source

gcc

clang



The image displays three side-by-side screenshots of a code editor, illustrating the customization of the compilation process. Each screenshot shows the same source code for a C++ function named `square`, which takes an integer `num` and returns `num * num`.

```
1 // Type your code here, or
2 int square(int num) {
3     return num * num;
4 }
```

The first screenshot shows the source code. The second screenshot shows the assembly output generated by GCC 10.2. The assembly code is as follows:

```
1 square(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], num
5     mov     eax, DWORD PTR [rbp-4]
6     imul   eax, eax
7     pop     rbp
8     ret
```

The third screenshot shows the assembly output generated by Clang 11.0.1. The assembly code is as follows:

```
1 square(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     dword ptr [rbp - 4], num
5     mov     eax, dword ptr [rbp + 4]
6     imul   eax, dword ptr [rbp - 4]
7     pop     rbp
8     ret
```

Editor, Compiler

# Workbench

- Editor
- Execution
- Compiler
- Tool chain

The screenshot displays the Visual Studio Code Compiler Explorer interface. It is divided into several panes:

- Source Code (Left):** Shows a C program with a loop that prints "hello" and a counter from 0 to 4.
- Assembly (Middle):** Displays the assembly code for the program, including instructions like `push rbp`, `mov rbp, rsp`, `call puts`, and `call printf`.
- Execution (Right):** Shows the execution output, which matches the source code's output: "hello" followed by counter values 0 through 4.
- Output (Bottom):** Shows the compiler options and the output of the compilation process.

# Lots of options...

<b>ARM GCC</b>	<b>MSVC X64</b>	<b>MIPS GCC</b>
ARM gcc 9.2.1 (none)	x64 msvc v19.28	MIPS64 gcc 5.4 (el)
ARM gcc 8.3.1 (none)	x64 msvc v19.27	MIPS64 gcc 5.4
ARM64 gcc 8.2	x64 msvc v19.25	MIPS gcc 5.4 (el)
ARM gcc 8.2 (WinCE)	x64 msvc v19.24	MIPS gcc 5.4
ARM gcc 8.2	x64 msvc v19.23	
ARM64 gcc 7.3	x64 msvc v19.22	<b>MSP GCC</b>
ARM gcc 7.3	x64 msvc v19.21	MSP430 gcc 6.2.1
ARM gcc 7.2.1 (none)	x64 msvc v19.20	MSP430 gcc 5.3.0
ARM64 gcc 6.4	x64 msvc v19.16	MSP430 gcc 4.5.3
ARM gcc 6.4	x64 msvc v19.15	
ARM64 gcc 6.3.0 (linux)	x64 msvc v19.14	
ARM gcc 6.3.0 (linux)		
ARM gcc 5.4.1 (none)		
ARM64 gcc 5.4 (linux)		

# Compiler Explorers Features

# Code Highlighting

```
A Save/Load + Add new... Vim CppInsights
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

Match Code with  
assembly instruction

```
x86-64 gcc 10.2 Compiler options...
A Output... Filter... Libraries + Add new... Add tool...
1 square(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul   eax, eax
7     pop     rbp
8     ret
```

# Documentation

x86-64 gcc 10.2  Compiler options...

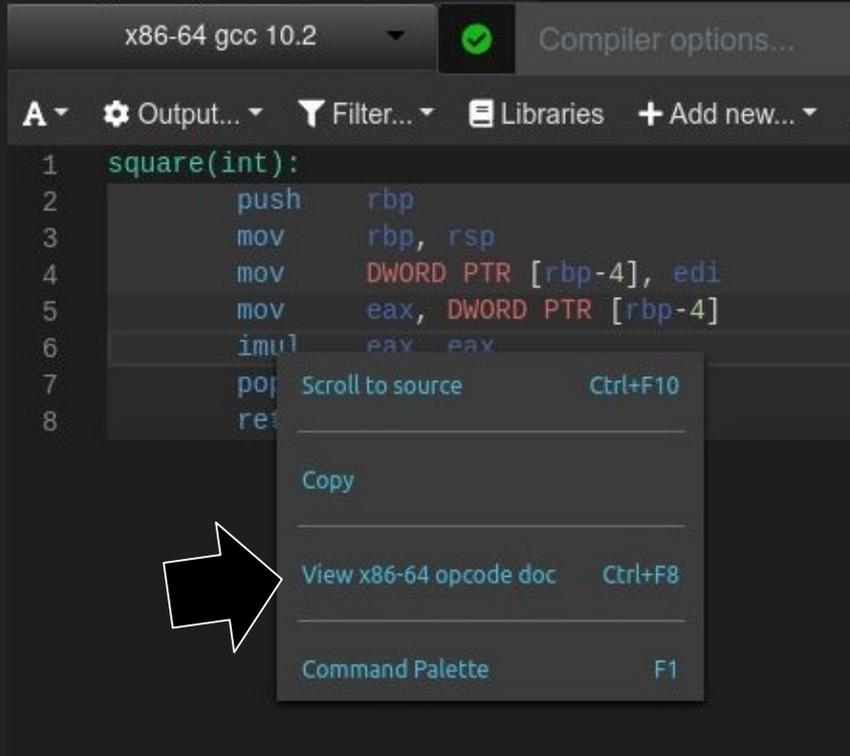
**A**  Output...  Filter...  Libraries  Add new...  Add tool...

```
1 square(int):  
2     push    rbp  
3     mov     rbp, rbp  
4     mov     edi, edi  
5     mov     ecx, 0  
6     imul   eax, eax  
7     pop     ebp  
8     ret
```

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands. More information available in the context menu.



# Documentation



x86-64 gcc 10.2 ✓ Compiler options...

A ⚙️ Output... 🔍 Filter... 📖 Libraries + Add new...

```
1 square(int):  
2     push    rbp  
3     mov     rbp, rsp  
4     mov     DWORD PTR [rbp-4], edi  
5     mov     eax, DWORD PTR [rbp-4]  
6     imul   eax, eax  
7     pop     rbp  
8     ret
```

Scroll to source Ctrl+F10

---

Copy

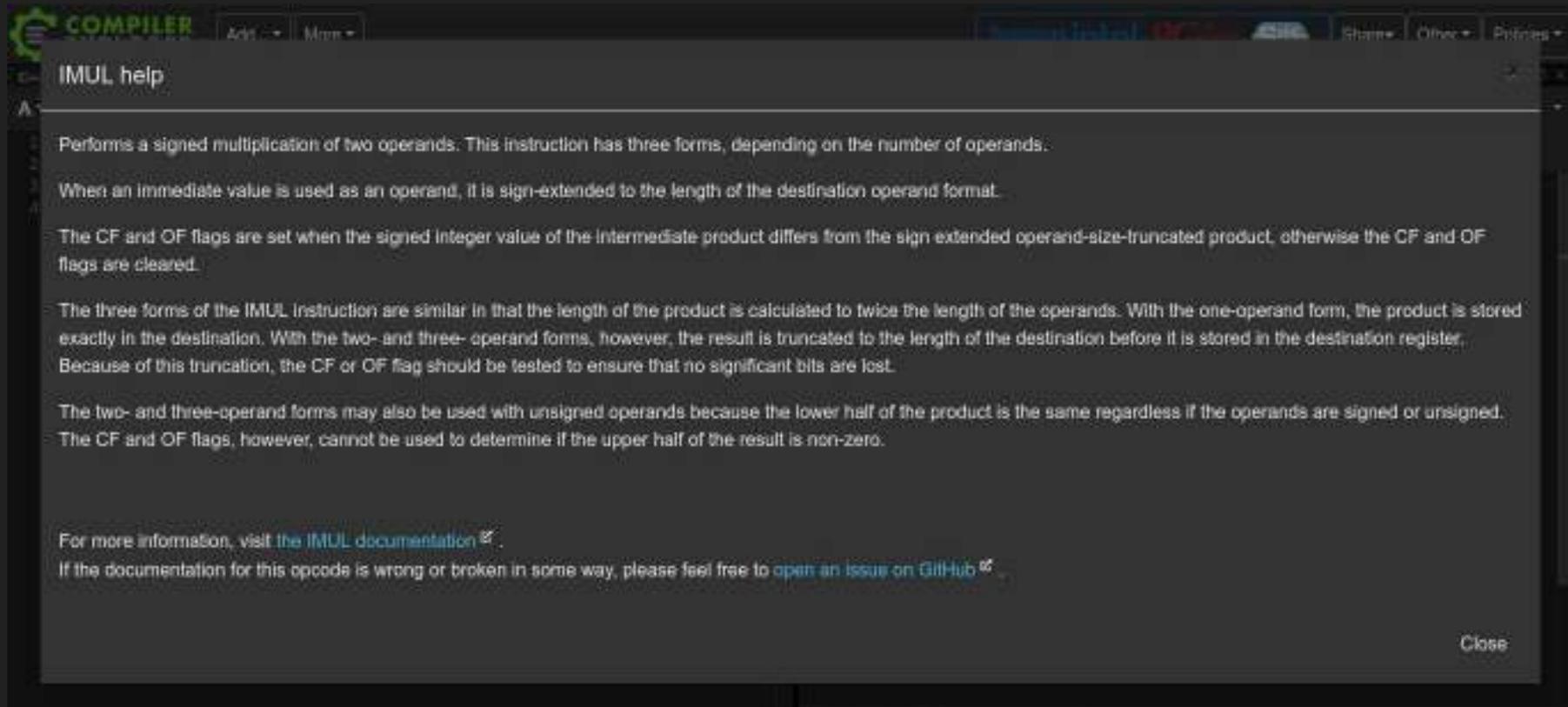
---

View x86-64 opcode doc Ctrl+F8

---

Command Palette F1

# Documentation



The screenshot shows a dark-themed window titled "IMUL help" from a "COMPILER" application. The window contains the following text:

IMUL help

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign-extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three-operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

For more information, visit the [IMUL documentation](#).

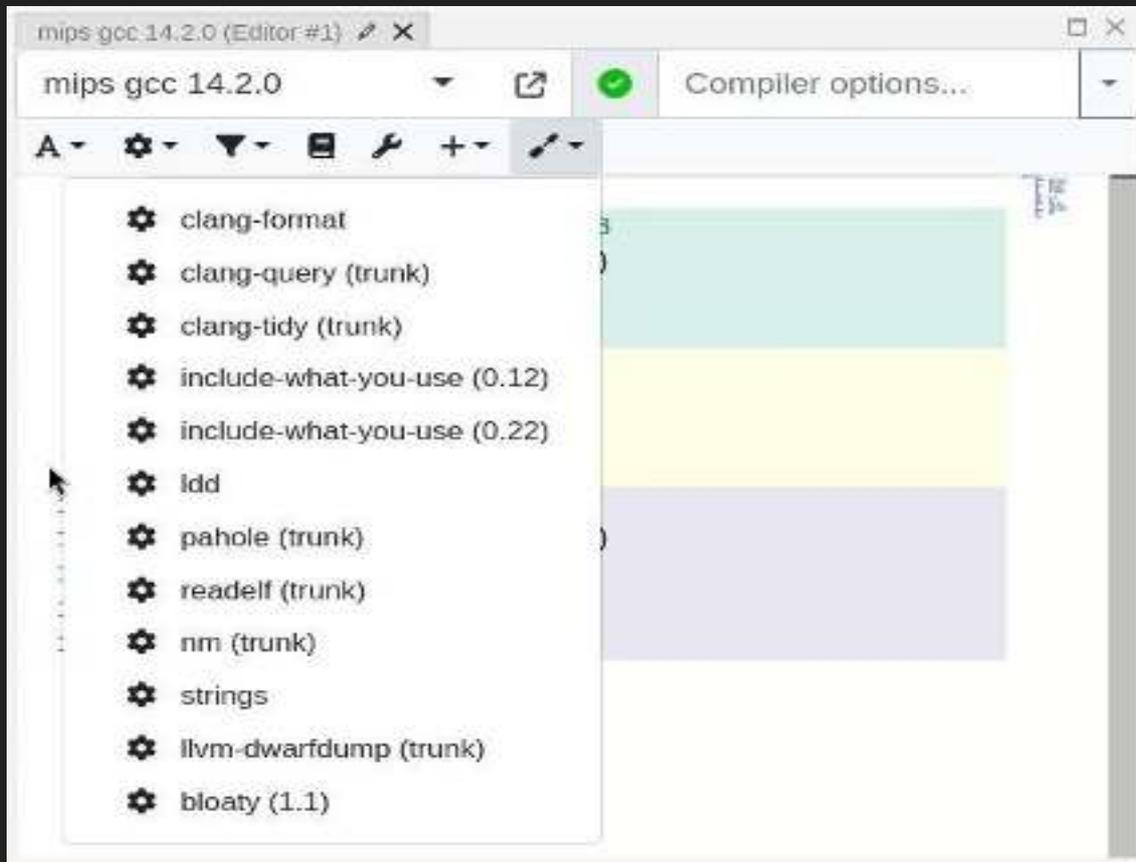
If the documentation for this opcode is wrong or broken in some way, please feel free to [open an issue on GitHub](#).

Close

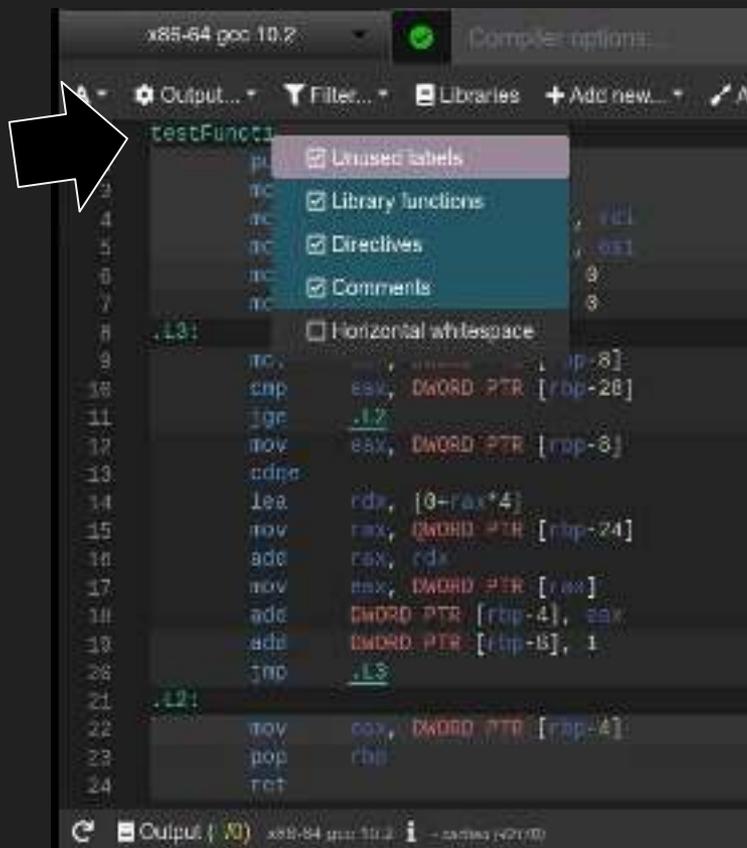
# Even more probing tools...

## CLI toolbox

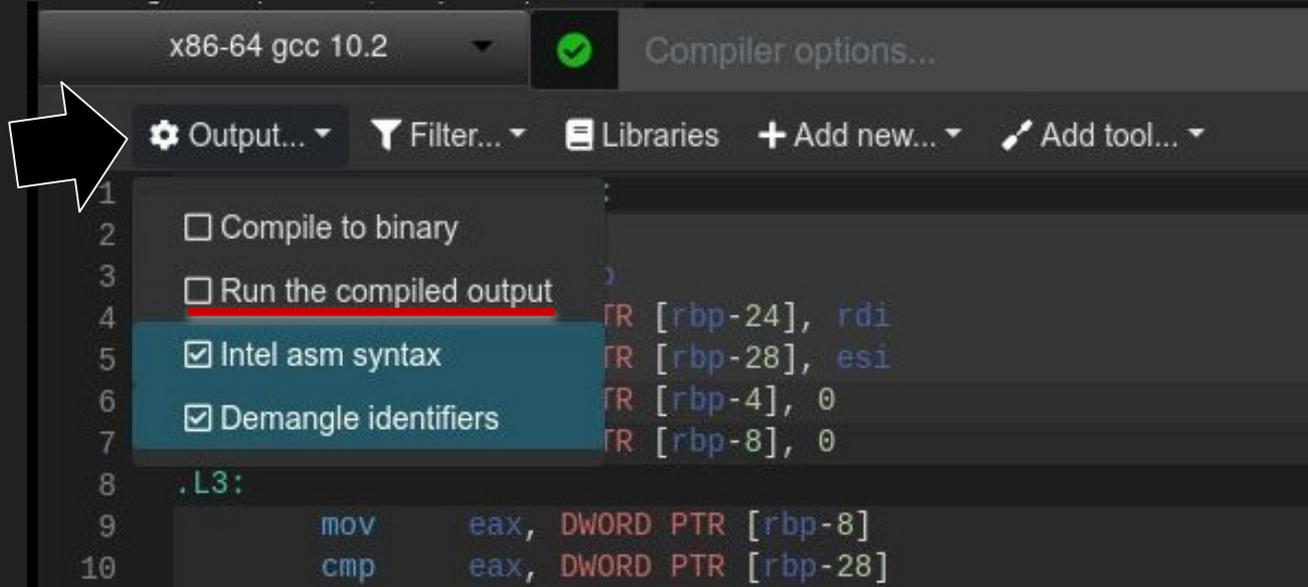
- ldd
- strings
- readelf



# Filtering options



# Output options

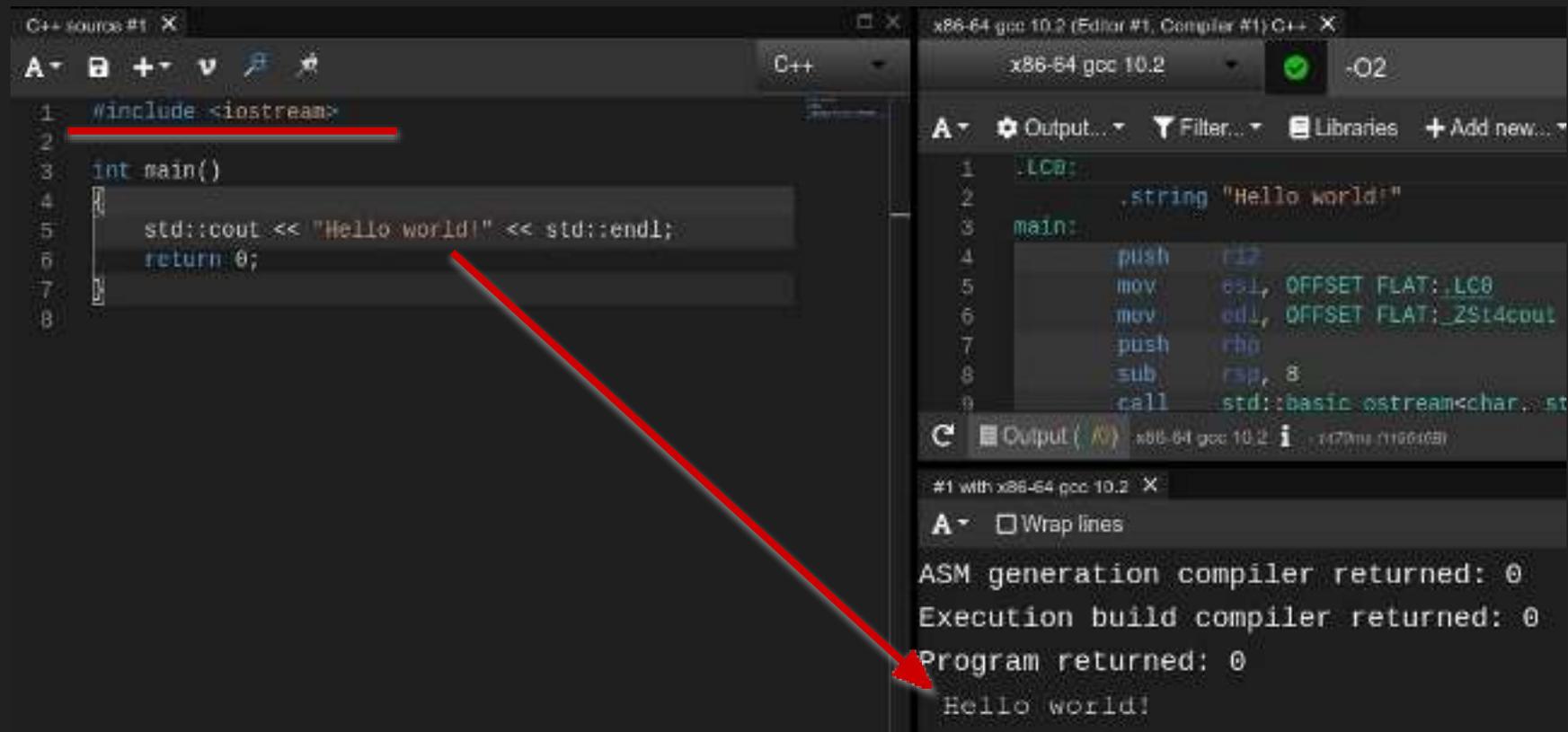


The image shows a screenshot of a compiler's interface. At the top, it displays 'x86-64 gcc 10.2' and a green checkmark icon next to 'Compiler options...'. Below this is a toolbar with icons for 'Output...', 'Filter...', 'Libraries', '+ Add new...', and 'Add tool...'. A white arrow points to the 'Output...' icon. A dropdown menu is open, showing the following options:

- Compile to binary
- Run the compiled output
- Intel asm syntax
- Demangle identifiers

The background shows assembly code with line numbers 1 through 10. Lines 4-7 show register operations: `FR [rbp-24], rdi`, `FR [rbp-28], esi`, `FR [rbp-4], 0`, and `FR [rbp-8], 0`. Line 8 is a label `.L3:`. Lines 9 and 10 show instructions: `mov eax, DWORD PTR [rbp-8]` and `cmp eax, DWORD PTR [rbp-28]`.

# Code execution



```
C++ source #1 X
A- B +> v [ ] *
C++
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello world!" << std::endl;
6     return 0;
7 }
8
```

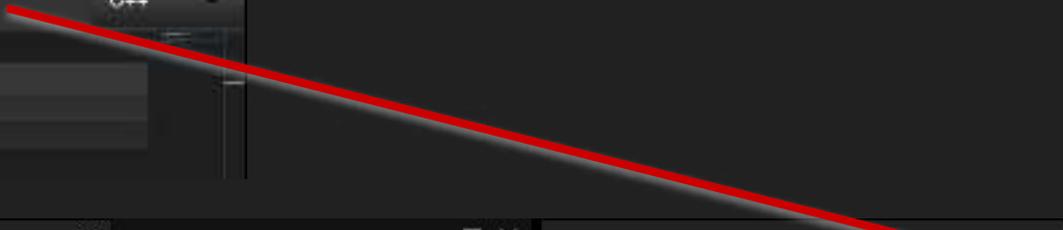
```
x86-64 gcc 10.2 (Editor #1, Compiler #1) C++ X
x86-64 gcc 10.2 -O2
A- Output... Filter... Libraries + Add new...
1 .LC0:
2     .string "Hello world:"
3 main:
4     push    r12
5     mov     esi, OFFSET FLAT:.LC0
6     mov     edi, OFFSET FLAT:_ZSt4cout
7     push   rbp
8     sub    rsp, 8
9     call   std::basic_ostream<char, st
Output (0) x86-64 gcc 10.2 i 147ms (11660B)
#1 with x86-64 gcc 10.2 X
A- [ ] Wrap lines
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
Hello world!
```

# Compiler options

```
C++ source #1 X
A- [ ] +- v [ ] [ ] C++
1 int getInteger(int x)
2 {
3     return x;
4 }
5
```

```
x86-64 gcc 10.2 (Editor #1, Compiler #1) C++ X
x86-64 gcc 10.2 [ ] -O0
A- [ ] [ ] [ ] [ ] [ ] [ ]
1 getInteger(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     pop     rbp
7     ret
```

```
x86-64 gcc 10.2 (Editor #1, Compiler #2) C++ X
x86-64 gcc 10.2 [ ] -O2
A- [ ] [ ] [ ] [ ] [ ] [ ]
1 getInteger(int):
2     mov     eax, edi
3     ret
```



# Compiler options

The screenshot displays the Visual Studio Code interface with a C program and its assembly output for three different optimization levels: -O1, -O2, and -fschedule-insns2.

```
1 short foo(short a, short b) {
2     short result;
3     result = b;
4     while(a > 0) {
5         result += a;
6         b -= a;
7     }
8     return result;
9 }
```

**Assembly for -O1:**

```
1 .foo
2     movl    %eax, %eax
3     movl    %eax, %eax
4     testl  %eax, %eax
5     jle    .L3
6     .L3:
7     imull  %eax, %eax
8     subl  %eax, %eax
9     testl  %eax, %eax
10    jg     .L2
11    .L2:
```

**Assembly for -O2:**

```
1 .foo:
2     mov    %eax, %eax
3     mov    %eax, %eax
4     test  %i, %i
5     jle   .L3
6     .L3:
7     sub   %eax, %eax
8     test %eax, %eax
9     jg   .L2
10    .L2:
11    ret
```

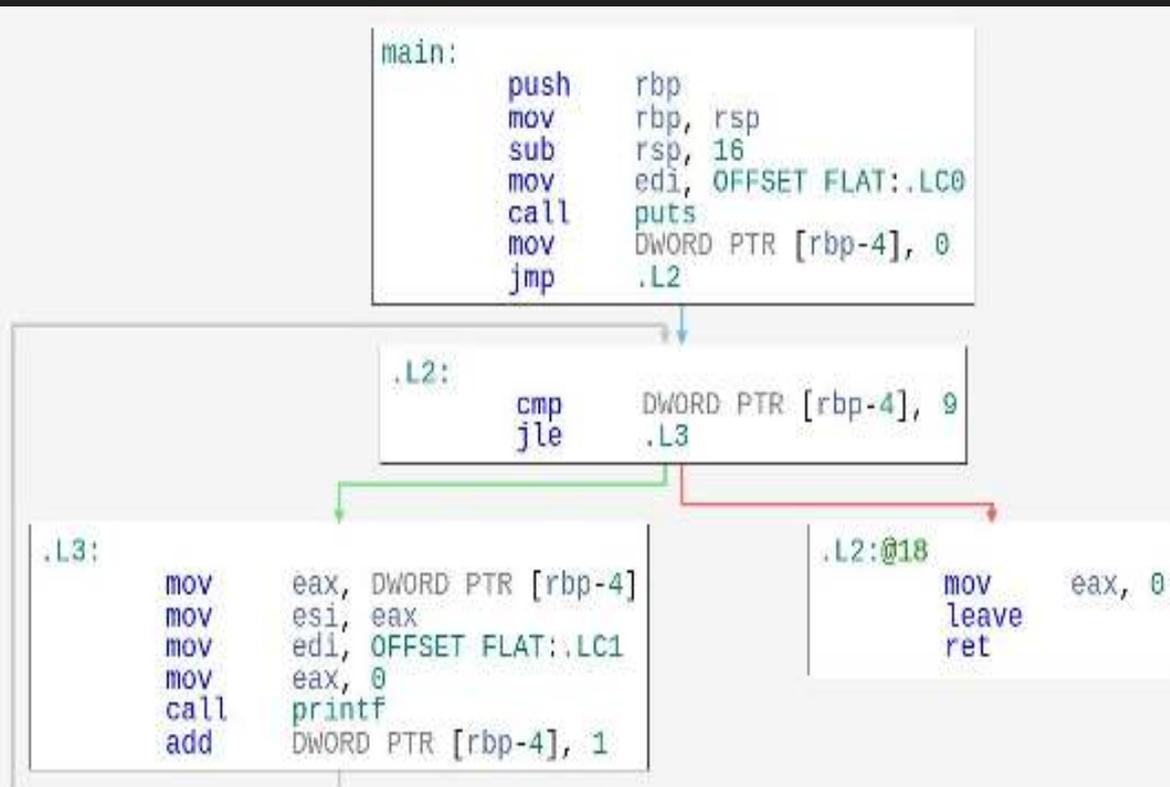
**Assembly for -fschedule-insns2:**

```
1 .foo
2     movl    %eax, %eax
3     movl    %eax, %eax
4     testl  %eax, %eax
5     jle    .L3
6     .L3:
7     subl  %eax, %eax
8     imull %eax, %eax
9     testl %eax, %eax
10    jg     .L2
11    .L2:
```

Instruction Order changed

- imul
- Subl
- Gcc O1, O2 optimization level

# Control Flow Graph



# Other Tools

# C++ Insights

# C++ Insights



The image shows a screenshot of the Compiler Explorer web application. At the top left is the logo for Compiler Explorer, which consists of a green gear icon with three horizontal bars inside, followed by the text "COMPILER EXPLORER" in green and white. To the right of the logo are two buttons: "Add..." and "More", both with dropdown arrows. Below the logo is a tab labeled "C++ source #1" with a close button (X). The main interface has a dark background with a toolbar containing several icons and labels: a magnifying glass icon, "Save/Load", a plus sign icon, "Add new...", a Vim logo, a magnifying glass icon, "CppInsights", and a lightning bolt icon, "Quick-bench". A white arrow points to the "CppInsights" button. Below the toolbar is a code editor with the following C++ code:

```
17 double result[N];
18
19 void initialize();
20 void vm_multiply();
21 void output();
22
23 int main() {
24     int i;
```

# C++ Insights

The screenshot displays the C++ Insights web-based IDE. The interface is split into two main vertical panes: 'Source' on the left and 'Insight' on the right. At the top, there is a toolbar with navigation icons and a dropdown menu currently set to 'C++17 - standard c++17'. The 'Source' pane contains the following code:

```
11 auto Best() -> int {
12     {}
13     return 1;
14 }
15
16 constexpr auto CBest() -> int {
17     {}
18     return 1;
19 }
20
21 decltype(auto) best() {
22     {}
23     return {};
24 }
25
26 constexpr decltype(auto) CBest() {
27     {}
28     return {};
29 }
30
31 inline constexpr decltype(auto) WithBest() {
32     {}
33     return {};
34 }
```

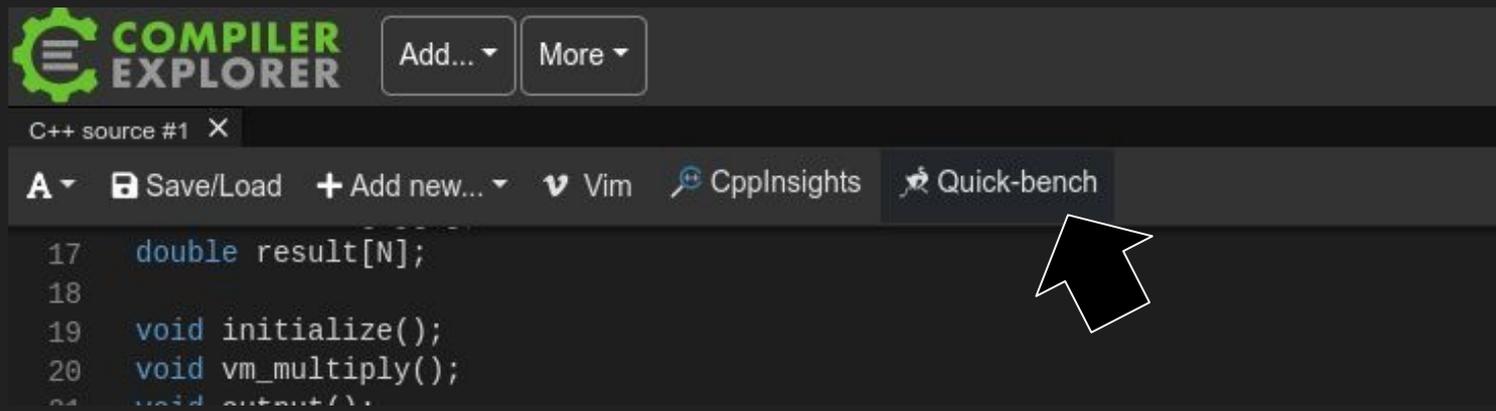
The 'Insight' pane shows the compiler-generated code for the above source, with line numbers corresponding to the source code. Red arrows point from the source code to the insight code:

- Line 11 of source points to line 17 of insight: `int Best()`
- Line 16 of source points to line 24 of insight: `inline constexpr int CBest()`
- Line 21 of source points to line 35 of insight: `auto best()`

At the bottom of the interface is a 'Console' pane with the message: 'Insights edited with result code: 0'.

Quick-bench

# Quick-bench



# Quick C++ Benchmark

The screenshot displays the Quick C++ Benchmark web interface. On the left, the source code is shown in a dark-themed editor. The code includes a main function that calls `BM_SomeFunction` and registers it as a benchmark. The `BM_SomeFunction` function is highlighted in blue and contains a loop that calls `rand()`. A red arrow points from the `rand()` call in the code to the benchmark results on the right.

```
80     if (argc == 0)
81         printf("v\n");
82         printf(" %6.4f", result[i]);
83     }
84     printf("v\n");
85     return
86     result;
87 }
88
89 static void BM_SomeFunction(benchmark::State& state) {
90     // Perform setup here
91     for (auto_ : state) {
92         // This code gets timed
93         rand();
94     }
95 }
96
97 // Register the function as a benchmark
98 BENCHMARK(BM_SomeFunction);
99
```

On the right side of the interface, the configuration is set to `compiler = GCC 10.1`, `std = c++11`, and `optim = O2`. The `stdlib` is set to `libstdc++(GNU)`. The `Run Benchmark` button is highlighted in green, and the `Record disassembly` checkbox is checked. Below the configuration, there are tabs for `Charts` and `Assembly`. The `Charts` tab is active, showing a bar chart with a single bar for `BM_SomeFunction`. The bar is blue and has a value of `1.1221-1.1221/2767824`. A red arrow points from the `rand()` call in the code to this bar.

Discover Build Bench! Support Quick Bench! Auto? More?

compiler = GCC 10.1 std = c++11 optim = O2  
stdlib = libstdc++(GNU)

Run Benchmark Record disassembly Clear cached results

Charts Assembly

1.1221221  
1.1221221  
1.1221221  
1.1221221  
1.1221221  
1.1221221  
1.1221221  
1.1221221  
1.1221221

BM\_SomeFunction  
1.1221-1.1221/2767824

BM\_SomeFunction

# Quick C++ Benchmark

- Relies on Google Benchmark
  - [github.com/google/benchmark](https://github.com/google/benchmark)

# Summary

- It's in the browser!
- Edit, Compile, View Disassembly, benchmarking and execution.
- Many Languages, Compilers, architectures and configuration options
- C, C++, Rust, C#, GO, FORTRAN, Python, Ruby, Java, ...
- Armv7, AVR, X86, MIPS, RISC-V, SPARC, VAX, and many more.

Questions?

Thanks