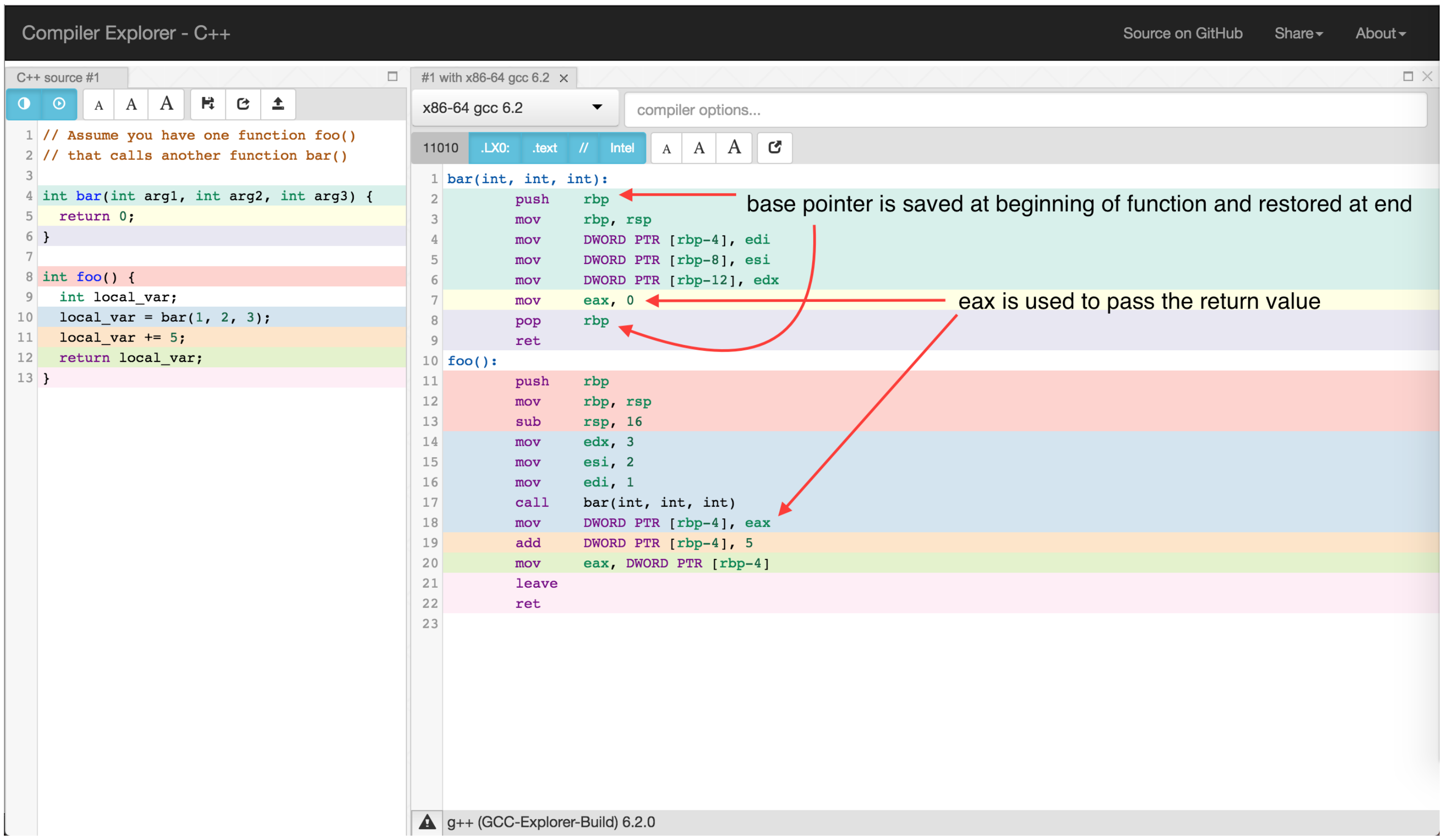
A Guide to x86 Calling Convention

cstack Follow
Nov 12, 2016 · 2 min read

I'm writing a hobby OS, and I've run into a lot of bugs that required me to understand how functions call each other in x86. I have to call C functions from assembly, call assembly functions from C, and bypass function calls entirely when entering or exiting interrupt handlers.

I decided to put in some work and make a cheat sheet to help me remember how all this works. I found a *really* handy website called Compiler Explorer that lets you look at the assembly of any C code in your browser. I used it to make this quick reference:



```
// Assume you have one function foo()
// that calls another function bar()

int bar(int arg1, int arg2, int arg3) {
  return 0;
}

int foo() {
  int local_var;
  local_var = bar(1, 2, 3);
  local_var += 5;
  return local_var;
}
```

```
bar(int, int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, 0
        pop     rbp
        ret
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    bar(int, int, int)
        mov     DWORD PTR [rbp-4], eax
        add     DWORD PTR [rbp-4], 5
        mov     eax, DWORD PTR [rbp-4]
        leave
        ret
```

base pointer is saved at beginning of function and restored at end

eax is used to pass the return value

x86 calling convention quick reference

But one part of the calling convention I had a hard time visualizing was how the stack changes as the code executes. So I put in some time and made an animation that walks though the code step by step and updates the stack:

Animation visualizing the stack during a function call

Step-by-Step

I'll walk through the steps now, but it makes the most sense when you watch the visualization as well.

Assume you have one function foo() that calls another function bar():

```
int bar(int arg1, int arg2, int arg3) {
  return arg1;
}

int foo() {
  int local_var = bar(1, 2, 3);
  local_var += 5;
  return local_var;
}
```

1. The first thing foo() does (and the first thing every function does) is push the base pointer onto the stack, then save the stack pointer into the base pointer. The base pointer points to the beginning of the stack frame. Just after it are the local variables of the current function. Just before is the return address.
2. foo() grows the stack to make room for local variables.
3. foo() saves arguments to bar() in registers. (If the arguments were too big to fit into register, it would have pushed them onto the stack instead.
4. foo() calls bar(). This pushes the return address onto the stack and jumps to the first instruction of bar().
5. Just like foo(), the first thing bar() does is push the base pointer onto the stack and save the stack pointer into the base pointer. (This chain of pointers is also how stack traces work).
6. bar() moves its arguments from registers onto the stack.
7. bar() saves the return value into the `eax` register.
8. The `leave` instruction copies the base pointer into the stack pointer, then pops the saved base pointer back into `ebp` .
9. The `ret` instruction pops the return address off the stack and jumps to it.
10. Back in foo(), the return value moves from the `eax` register into the area on the stack reserved for local variables. foo() finishes out its last instructions, then follows the same procedures as bar() to return to its own caller.

Programming C Programming Assembly Operating Systems Code