C Compiler, Part 9: Functions

Jun 27, 2018

This is the ninth post in a series. Read part 1 here.

In this post we're adding function calls! This is a particularly exciting post because we get to talk about calling conventions and stack frames and some weird corners of the C11 standard. Plus, by the end of this post we'll be able to compile "Hello, World!" 

As usual, accompanying tests are here.

Part 9: Functions

Of course, our compiler can already handle function definitions, because we can already define `main`. But in this post, we'll add support for function calls:

```
int three() {
    return 3;
}

int main() {
    return three();
}
```

We'll also add support for function parameters:

```
int sum(int a, int b) {
    return a + b;
}

int main() {
    return sum(1, 1);
}
```

And for forward declarations:

```c
int sum(int a, int b);

int main() {
    return sum(1, 1);
}

int sum(int a, int b) {
    return a + b;
}
```

Terminology

- A function **declaration** specifies a function's name, return type, and optionally its parameter list:

  ```c
  int foo();
  ```

- A function **prototype** is a special type of function declaration that includes parameter type information:

  ```c
  int foo(int a);
  ```

  Function prototypes are the only function declarations we'll support, even in places where the C11 standard allows non-prototype declarations.

- A function **definition** is a declaration plus a function body:

  ```c
  int foo(int a) {
      return a + 1;
  }
  ```

  Note that you can declare a function as many times as you like, but you can only define it once[1]. Also note that whenever we say "all function declarations," that includes function declarations that are part of function definitions.

- A **forward declaration** is a function declaration without a function body. It tells the compiler you're going to define the function later, possibly in a different file, and lets you use a function before it's defined.

  ```c
  int foo(int a);
  ```

You can also declare a function that has already been defined. This is legal but technically not a forward declaration...I guess it's a backwards declaration? It would also be pretty pointless:

```
int foo() {
    return 4;
}

int foo();
```

- A function's **arguments** are the values passed to a function call. A function's **parameters** are the variables defined in the function declaration. In this code snippet, `a` is a parameter and `3` is an argument:

```
int foo(int a) {
    return a + 1;
}

int main() {
    return foo(3);
}
```

Limitations

- For now, we'll only support functions with return type `int` and parameters with type `int`.
- We won't support function declarations with missing parameters or type information; in other words, we'll require all function declarations to be function prototypes, whether or not they're part of function definitions.
- We'll interpret an empty parameter list (e.g. in the declaration `int foo()`) to mean that the function has no parameters. This deviates from the C11 standard; according to the standard, `int foo(void)` is a function prototype indicating `foo` has no parameters, and `int foo()` is a declaration where the parameters aren't specified (i.e. not a function prototype).
- We won't support function definitions using identifier-list form, which looks like this:

```
int foo(a)
int a;
{
    return a * 2;
}
```

- We'll require parameter names in function declarations. For example, we won't support this:

```
int foo(int, int);
```

- We won't support storage class specifiers (e.g. `extern`, `static`), type qualifiers (e.g. `const`, `atomic`), function specifiers (`inline`, `_Noreturn`) or alignment specifiers (`_Alignas`)

Lexing

Nothing fancy here; we just need to add commas to separate the function arguments. Here's the full list of tokens so far:

- `{`
- `}`
- `(`
- `)`
- `;`
- `int`
- `return`
- Identifier `[a-zA-Z]\w*`
- Integer literal `[0-9]+`
- `-`
- `~`
- `!`
- `+`
- `*`
- `/`
- `&&`
- `||`
- `==`
- `!=`
- `<`
- `<=`
- `>`
- `>=`
- `=`
- `if`
- `else`
- `:`
- `?`
- `for`

- `while`
- `do`
- `break`
- `continue`
- `,`

☑ Task:

Add support for commas to the lexer.

Parsing

We'll deal with function definitions first, then function calls.

Function Definitions

In our old definition, a function just had a name and a body:

```
function_declaration = Function(string, block_item list) //string is the f
```

Now we need to add a list of parameters. We also need to support declarations that don't include a function body. I defined a single `function_declaration` AST rule, with an optional function body, to represent both declarations and definitions:

```
function_declaration = Function(string, // function name
                                string list, // parameters
                                block_item list option) // body
```

But you could also have different rules for function declarations and definitions if you wanted.

Note that we don't include the function's return type or parameter types, because right now `int` is the only type. We'll need to expand this definition when we add other types.

We also need to update the grammar. Here was the old `<function>` grammar rule:

```
<function> ::= "int" <id> "(" ")" "{" { <block-item> } "}"
```

And here's the new one. Note that the function declaration ends with either a function body (if it's a definition) or a semicolon (if it's not).

```
<function> ::= "int" <id> "(" [ "int" <id> { "," "int" <id> } ] ")" ( "{"
```

Function Calls

A function call is an expression that looks like this:

```
foo(arg1, arg2)
```

It has an ID (the function name) and a list of arguments. Its arguments can be arbitrary expressions:

```
foo(arg1 + 2, bar())
```

So we can update the AST definition for expressions like this:

```
exp = ...
    | FunCall(string, exp list) // string is the function name
    ...
```

We also need to update the grammar. Function calls have the highest possible precedence level, right up there with postfix unary operators. So we'll add them to the `<factor>` rule in the grammar:

```
<factor> ::= <function-call> | "(" <exp> ")" | <unary_op> <factor> | <int>
<function-call> ::= id "(" [ <exp> { "," <exp> } ] ")"
```

Top Level

In our old definition, a program consisted of a single function definition. Now it needs to permit multiple function declarations:

```
program = Program(function_declaration list)
```

```
<program> ::= { <function> }
```

☑ Task:

Update parsing to succeed on all valid stage 1-9 examples. You may or may not want to handle invalid examples here: see the next section on validation.

Validation

We need to validate that the function declarations and calls in our program are legal. You can either handle these checks during code generation, or add a new validation pass between parsing and code generation. **Edited to add:** I previously recommended performing validation during the parsing stage. This turns out to be a bad idea, because this will become increasingly cumbersome as we need to validate more things in future posts.

Your compiler must fail if:

- The program includes two definitions of the same function name.

  ```
  int foo(){
      return 3;
  }

  int foo(int a){
      return a + 1;
  }
  ```

- Two declarations of a function have different numbers of parameters. Different parameter names are okay, though.

  This is illegal[2]:

  ```
  int foo(int a, int b);

  int foo(int a){
      return a + 1;
  }
  ```

  But this is okay:

  ```
  int foo(int a);

  int foo(int b){
      return b + 1;
  }
  ```

- A function is called with the wrong number of arguments, e.g.

  ```
  int foo(int a){
      return a + 1;
  }
  ```

```
int main() {
    return foo(3, 4);
}
```

- Optionally, you may want to fail if a function is called before it's declared. Note that it's totally legal to call a function that has been declared but not defined. It's also legal to declare a function and *never* define it; however, linking will fail if the function isn't declared in some other library the linker can find[3].

  So this is illegal:

  ```
  int main() {
      return putchar(65);
  }

  int foo(){
      return 3;
  }
  ```

  But this is legal:

  ```
  int putchar(int c);

  int main() {
      putchar(65);
  }
  ```

  This last point is optional because neither GCC nor clang enforces it — they both warn but don't fail on the illegal example above. Calling a function before it's declared is called "implicit function declaration" and it was legal before C99, so I guess enforcing this rule would have broken a lot of older code. The test suite doesn't include any implicit function declarations, so you can handle it however you like and you can still pass all the tests.

☑ `Task:`

Update your compiler to fail on invalid stage 1-9 examples. You can handle this during code generation, or a new stage between parsing and code generation. Bonus points for useful error messages.

To handle this, you'll probably want to traverse the tree and maintain a map to track the number of arguments to each function, and whether that function has been defined yet.

Code Generation

Once again, we'll handle function definitions first, then function calls. But before we do any of that, let's discuss…

Calling Conventions

In most of the examples above, we defined a function and then called it in the same file. But we also want to call functions from shared libraries; we particularly want to call the standard library, so we can access I/O functions, so we can write "Hello, World". When you use a shared library, you generally don't recompile it yourself; you link to a precompiled binary. We definitely don't want to recompile the whole standard library! That means we need to generate machine code that can interact with object files built by other compilers. In earlier posts, I've often said "this isn't how a real compiler would do this thing, but it works." In this post, we *have* to do things the same way as everyone else or we can't use prebuilt libraries.

In other words, we need to follow the appropriate *calling convention*. A calling convention answers questions like:

- How are arguments passed to the callee? Are they passed in registers or on the stack?
- Is the caller or callee responsible for removing arguments from the stack after the callee has executed?
- How are return values passed back to the caller?
- Which registers are caller-saved and which are callee-saved[4]?

C programs on 32-bit OS X, Linux, and other Unix-like systems use the `cdecl` calling convention[5], which means:

- Arguments are passed on the stack. They're pushed on the stack from right to left (so the first function argument is at the lowest address).
- The caller cleans the arguments from the stack.
- Return values are passed in the `EAX` register. (The full answer is more complicated, but this is good enough as long as we can only return integers.)
- The `EAX`, `ECX`, and `EDX` registers are caller-saved, and all others are callee-saved. We'll see in the next section that the callee has to restore `EBP` and `ESP` before it returns, and restores `EIP` with the `ret` instruction. Normally it would also need to restore `ESI`, `EDI`, and `EBX`, but we don't actually use these registers. And we already push values from `EAX`, `ECX`, and `EDX` onto the stack right away if we're going to need them later. So basically, we don't have to worry about saving and restoring registers at all.

There are two import differences between OS X and Linux:

- Stack alignment. On OS X, the stack needs to be 16-byte aligned at the beginning of a function call (i.e. when the `call` instruction is issued)[6]. This isn't required on Linux, but GCC

still keeps the stack 16-byte aligned[7].

- Name decoration. On OS X, function names in assembly are prepended with an underscore (e.g. `main` becomes `_main`). On systems that use the ELF file format (Linux and most other *nix systems), there's no underscore. This isn't part of the calling convention per se but it is important.

We'll need to be really comfortable with all this to implement it ourselves, so let's look at...

`cdecl` Function Calls in Excruciating Detail

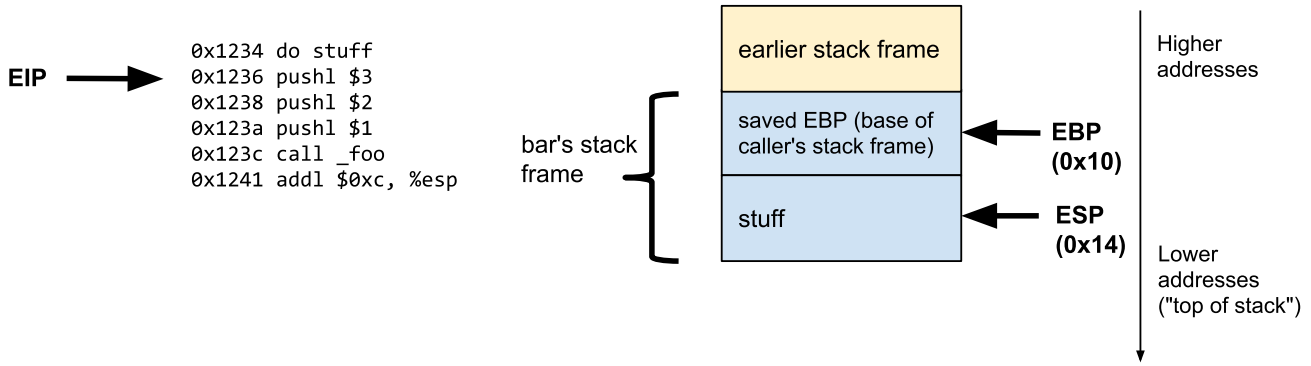
```
foo(1, 2, 3);
```

What, exactly, happens when your computer executes this line of code? We touched on this in part 5, but now we'll dig into it a lot more. We won't worry about keeping the stack 16-byte aligned for now.

We'll say that `foo` is being called from another function, `bar`. The line of C above will get turned into this assembly:

```
push $3
push $2
push $1
call _foo
add $0xc, %esp
```

First, let's look at the state of the world before we start calling `foo`[8]:



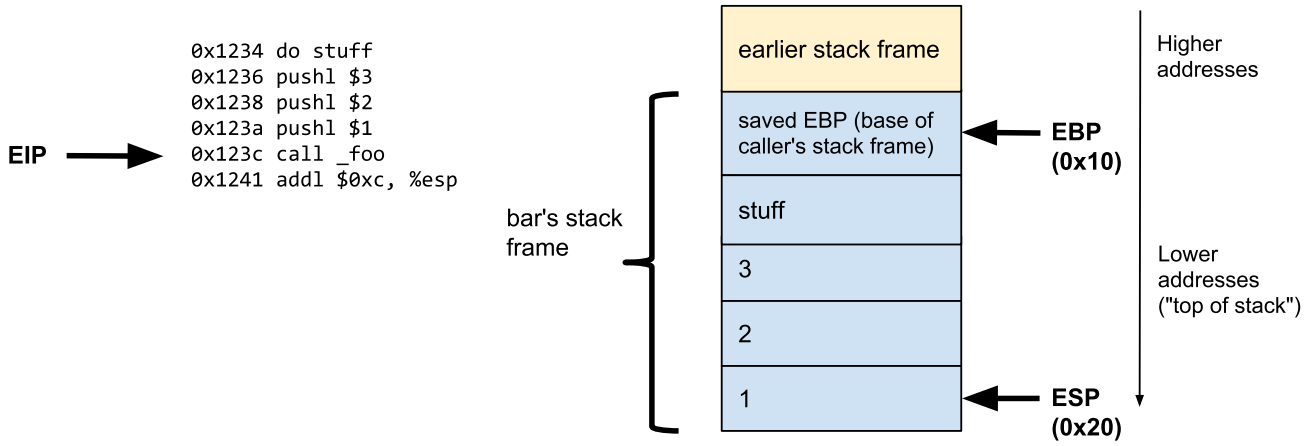
One chunk of memory contains the stack frame, which we're already familiar with. The `EBP` and `ESP` registers point to the bottom and top of the stack frame, respectively, so the processor can figure out where the stack is.

Another chunk of memory, which we haven't talked about yet, contains the CPU instructions being executed. The `EIP` register contains the memory address of the current instruction. To advance to the next instruction, the CPU just increments `EIP`[9]. The `call` instruction, and all the jump instructions we've already encountered, work by manipulating EIP. In these diagrams I'll show `EIP` pointing to the instruction we're about to execute.

When `bar` wants to call `foo`, the first step is putting the function arguments on the stack where `foo` can find them[10]. They're pushed onto the stack in reverse order[11]:

```
push $3
push $2
push $1
```

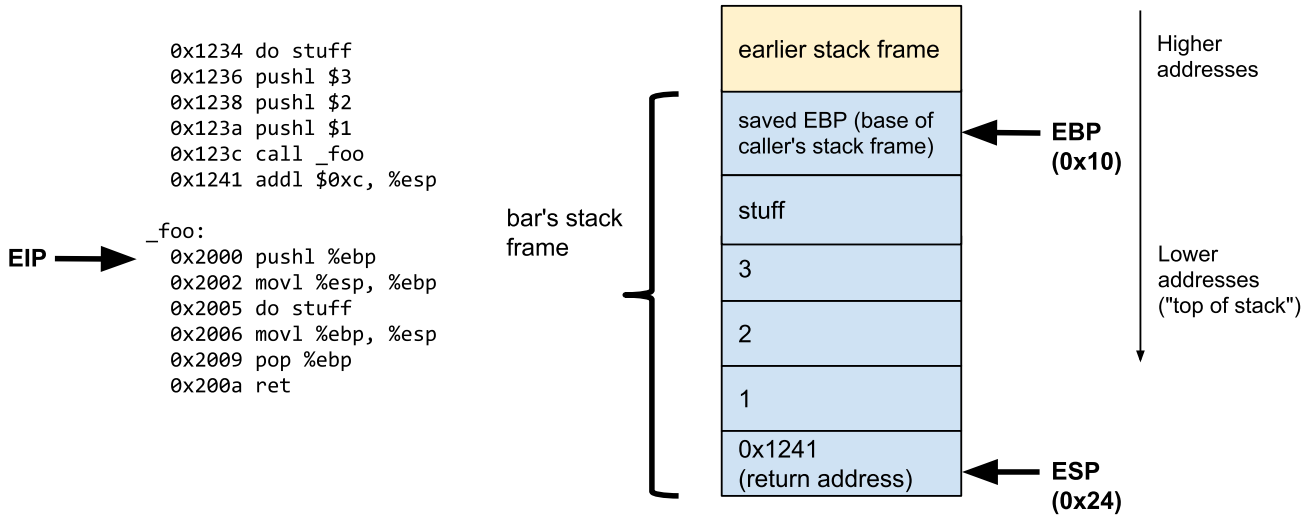
Which means the world now looks like this:



Next `bar` issues the `call` instruction, which does two things:

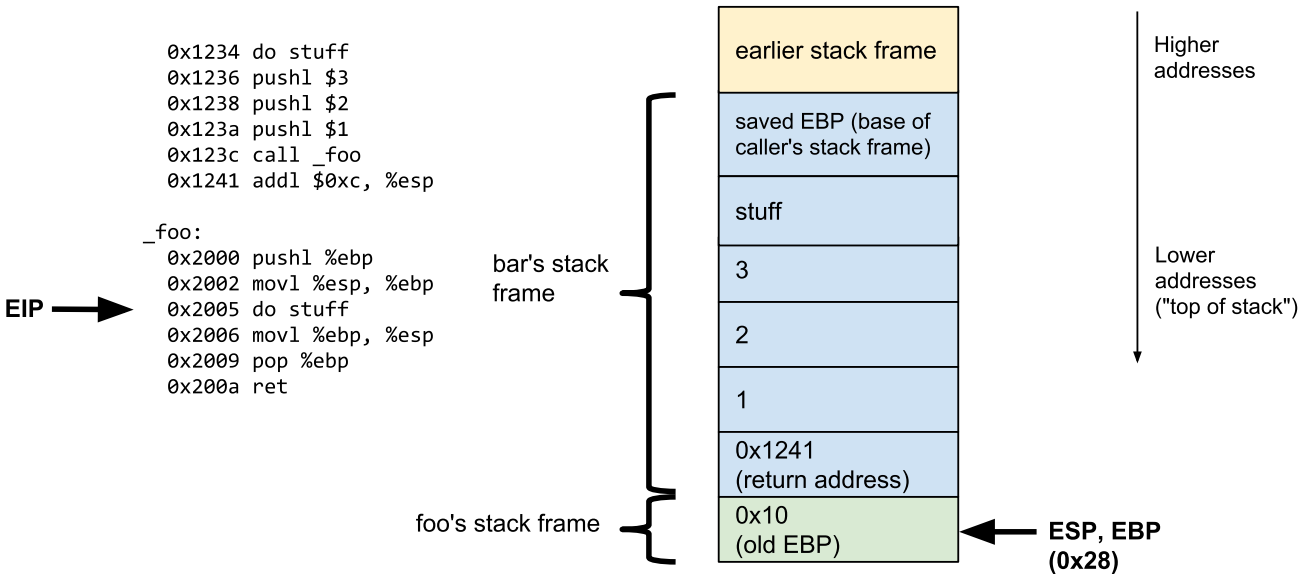
1. Push the address of the instruction *after* `call` (the "return address") onto the stack.
2. Jump to `_foo` (by moving the address of `_foo` into `EIP`).

Now the world looks like this:



Okay, we're officially in `foo` now. Next step is the function prologue to set up a new stack frame:

```
push %ebp
mov %esp, %ebp
```



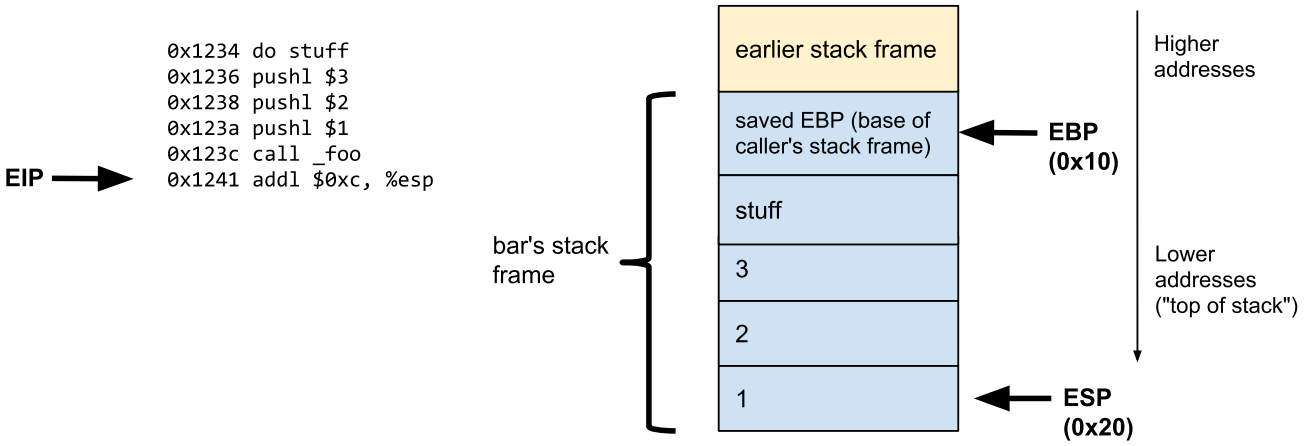
Now we can execute the body of `foo`. We can access its parameters because they're at a predictable location on the stack relative to `EBP`: `%ebp + 0x8`, `%ebp + 0xc`, and `%ebp + 0x10`, respectively.

Once we've done some things in `foo`, and placed a return value in `EAX`, it's time to return to `bar`. Except for that return value, we want everything on the stack to be exactly the same as it was before the call. The first step is to run the function epilogue to restore the old stack frame:

```
mov %ebp, %esp ; deallocate any local variables on the stack
pop %ebp       ; restore old EBP
```

The stack now looks exactly the same as it did right after the `call` instruction, before the function prologue. That means the return address is on top of the stack again.

Then we execute the `ret` instruction, which pops the top value off the stack and jumps to it unconditionally (i.e. copies it into `EIP`).



Now we just have to remove the function arguments from the stack, and we're done. No need to pop them off one by one; we can just adjust the value of `ESP`.

```
add $0xc, %esp
```

Now the stack has been restored to exactly the way it was before the call, and we can proceed with the rest of `bar`.

And now we're finally ready to implement the code-generation stage of the compiler!

Function Definitions

As with `main`, we want to make each function global (so it can be called from other files) and label it:

```
    .globl _fun
_fun:
```

Make sure to include the leading underscore before the function name if you're on OS X, and not otherwise.

We already know how to generate the function prologue and epilogue, because that's also exactly the same as `main`. We just need to add all the function parameters to `var_map` and `current_scope`. As we saw above, the first paramter will be at `ebp + 8`, and each subsequent parameter will be four bytes higher than the last:

```
param_offset = 8 // first parameter is at EBP + 8
for each function parameter:
    var_map.put(parameter, param offset)
    current_scope.add(parameter)
    param_offset += 4
```

Then parameters get handled like any other variable in the function body.

Function Prototypes

We don't generate any assembly for function prototypes that aren't part of definitions.

Function Calls

As we saw above, the caller needs to:

1. Put the arguments on the stack, in reverse order[1][2]:

   ```
   for each argument in reversed(function_call.arguments):
       generate_exp(arg) // puts arg in eax
       emit 'pushl %eax'
   ```

2. Issue the `call` instruction.

   ```
       emit 'call _{}'.format(function_name)
   ```

3. Remove the arguments from the stack after the callee returns.

   ```
       bytes_to_remove = 4 * number of function arguments
       emit 'addl ${}, %esp'.format(bytes_to_remove)
   ```

Stack Alignment

On OS X, the stack needs to be 16-byte aligned when the call instruction is issued. A normal C compiler would know exactly how much padding to add to maintain that alignment. But because we push intermediate results of expressions onto the stack, and function calls can occur within larger expressions, we have no idea where the stack pointer is when we encounter a function call. My solution was to emit assembly just before each function call that calculates how much padding is needed, subtracts from ESP accordingly, and then pushes the result of the padding calculation onto the stack, all before putting the function arguments on the stack. After the function returns, the caller first removes the arguments, then pops off the result of the padding calculation, and finally adds that value to ESP to restore it to its original state.

Here's the assembly to do that:

```
    movl %esp, %eax
    subl $n, %eax    ; n = (4*(arg_count + 1)), # of bytes allocated for a
                     ; eax now contains the value ESP will have when call
    xorl %edx, %edx  ; zero out EDX, which will contain remainder of divis
    movl $0x20, %ecx ; 0x20 = 16
    idivl %ecx       ; calculate eax / 16. EDX contains remainder, i.e. #
    subl %edx, %esp  ; pad ESP
    pushl %edx       ; push padding result onto stack; we'll need it to de
    ; ...push arguments, call function, remove arguments...
    popl %edx        ; pop padding result
    addl %edx, %esp  ; remove padding
```

This solution is kind of hideous, so let me know if you come up with a better one.

Top Level

Obviously, you need to generate assembly for every function definition, not just one.

☑ Task:

Update your compiler to handle all stage 9 examples. Make sure it produces the right return code *and*, for the "hello world" test case, the right output to stdout.

Fibonacci & Hello, World!

Now we can calculate Fibonacci numbers:

```
int fib(int n) {
    if (n == 0 || n == 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

int main() {
    int n = 10;
    return fib(n);
}
```

We can also make calls to the standard library! Since we only know about ints, we can only call standard library functions where the parameters are all ints and the return value is also an int. Lucky for us, `putchar` is just such a function. For example, since the ASCII value of 'A' is 65, we could print 'A' to standard out like this:

```
int main() {
    putchar(65);
}
```

And we can print out 'Hello, World!' like this:

```
int putchar(int c);

int main() {
    putchar(72);
    putchar(101);
    putchar(108);
    putchar(108);
    putchar(111);
    putchar(44);
    putchar(32);
    putchar(87);
    putchar(111);
    putchar(114);
    putchar(108);
    putchar(100);
    putchar(33);
    putchar(10);
}
```

Up Next

My next post or two won't be about compilers. After that I'll get back to this series, but I haven't decided what to implement next. Maybe pointers? We'll see!

Update: just kidding, the next post is about compilers after all, and covers global variables.

If you have any questions, corrections, or other feedback, you can email me or open an issue.

[1] Technically, you can redefine a function in the same *program* but not in the same *translation unit*. A translation unit is a source file plus everything that gets pulled in during preprocessing from `#include` directives. (Source: C11 standard, section 5.1.1.1)

So it's legal to redefine a function from a linked library. But linking happens after the compiler runs, so for our purposes the rule is that each function can only be defined once. ↩

[2] However, this is legal according to C11:

```
int foo();

int foo(int a){
    return a + 1;
}
```

That's because `int foo();` doesn't mean "declare a function foo with no variables"; it means "declare a function foo, but we don't know anything about its variables." But our compiler diverges from the standard in this respect; it assumes that `int foo();` means "declare foo with no variables," so it will fail here. ↩

[3] What the linker does and where it looks for function definitions is way beyond the scope of this blog post; if you want to learn more you might like the Beginner's Guide to Linkers or this series on linkers. ↩

[4] If a register is caller-saved, that means the callee is allowed to overwrite it. So if the caller wants to access the value in that register after the callee returns, it needs to push that value onto the stack, then pop it back into the register after the function call has completed.

If a register is callee-saved, the caller can assume that the register will be unchanged after the function call finishes. So if the callee wants to use that register, it has to save the register's contents to the stack and restore those contents before returning control to the caller. ↩

[5] Windows is a lot more complicated; sometimes it uses `cdecl`, sometimes it uses different calling conventions. A lot of Linux/OS X documentation doesn't even call it `cdecl`, presumably because it's the only calling convention in *nix-world. ↩

[6] Source: OS X ABI Function Call Guide. It's not 100% clear why OS X imposes this requirement but it probably has something to do with making SSE instructions run faster. ↩

[7] See the GCC documentation on `-mpreferred-stack-boundary`. ↩

[8] Note that these are not valid memory addresses; at least on Linux, the lowest memory address in use is 0x08048000. (See here and here). I think this is also true on OS X but I haven't checked. ↩

[9] It's actually a little more complicated than this; instructions are variable-width, so you can't increment EIP by the same amount for every instruction. ↩

[10] Actually, the first step is pushing some caller-saved registers onto the stack. But, like I mentioned earlier, the janky way we're managing registers means we can ignore this. ↩

[11] Pushing arguments onto the stack in reverse order makes it easier to handle functions with a variable number of arguments; the callee knows the location of the first argument even if it doesn't know how many arguments there are. ↩

[12] This means we'll also *evaluate* the arguments in reverse order. This is valid; function arguments may be evaluated in any order. (Source: C11 standard section 6.5.2.2, paragraph 10.) ↩
