Linking

The `ld` program determines where each function and data item will be located in memory when the program is executed. It then replaces the programmer's symbolic names where each of these items is referenced with the memory address of the item. The result of this linking is written to an *executable file*. The default name of the executable file is *a.out*, but you can specify another name with the `-o` option.

If the called function is in an external library, this is noted where the function is called, and the address of the external function is determined during program execution. The compiler directs the `ld` program to add the computer code to the executable file that sets up the C runtime environment. This includes operations such as opening paths to standard out (the screen) and standard in (the keyboard) for use by your program.

As you might know, if you don't use any of the `gcc` options to stop the process at the end of one of these steps (`-E`, `-S`, `-c`), the compiler will perform all four steps and automatically delete the intermediate files, leaving only the executable program as the final result. You can direct `gcc` to keep all the intermediate files with the `-save-temps` option.

The complement of being able to stop `gcc` along the way is that we can supply files that have effectively gone through the earlier steps, and `gcc` will incorporate those files into the remaining steps. For example, if we write a file in assembly language, `gcc` will skip the preprocessing and compilation steps and perform the assembly and linking steps. If we supply only object files (*.o*), `gcc` will go directly to the linking step. An implicit benefit of this is that we can write programs in assembly language that call functions in the C standard library (which are already in object file format), and `gcc` will automatically link our assembly language with those library functions.

Be careful to use the filename extensions that are specified in the GNU programming environment when naming a file. The default action of the compiler at each step depends upon the filename extension appropriate to that step. To see these naming conventions, type **`info gcc`** into the command line, select `Invoking GCC`, and then select `Overall Options`. If you don't use the specified filename extension, the compiler might not do what you want or even overwrite a required file.

From C to Assembly Language

Programs written in C are organized into functions. Each function has a name that is unique within the program. After the C runtime environment is set up, the `main` function is called, so our program starts with `main`.

Since we can easily look at the assembly language that the compiler generates, that is a good place to start. We'll start off by looking at the assembly language that `gcc` generates for the minimum C program in Listing 10-1. The program does nothing except return 0 to the operating system. A program can return various numerical error codes to the operating system; 0 indicates that the program did not detect any errors.

NOTE *If you are not familiar with the GNU* `make` *program, I urge you to learn how to use it to build your programs. It may seem like overkill at this point, but it's much easier to learn with simple programs. The manual is available in several formats at* https://www.gnu.org/software/make/manual/, *and I have some comments about using it on my website,* https://rgplantz.github.io/.

```
/* doNothingProg.c
 * Minimum components of a C program.
 */

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

Listing 10-1: Minimum C program

Even though this program accomplishes very little, some instructions need to be executed just to return 0. To see what takes place, we first translate this program from C to assembly language with the following GNU/Linux command:

```
$ gcc -O0 -Wall -masm=intel -S doNothingProg.c
```

Before showing the result of this command, I'll explain the options I've used. The `-O0` (uppercase O and zero) option tells the compiler not to use any optimization. The goal of this book is to show what's taking place at the machine level. Asking the compiler to optimize the code may obscure some important details.

You've already learned that the `-Wall` option asks the compiler to warn you about questionable constructions in your code. It's not likely in this simple program, but it's a good habit to get into.

The `-masm=intel` option directs the compiler to generate assembly language using the Intel syntax instead of the default AT&T syntax. I'll explain why we use Intel syntax later in this chapter.

The `-S` option directs the compiler to stop after the compilation phase and write the assembly language resulting from the compilation to a file with the same name as the C source code file, but with the *.s* extension instead of *.c*. The previous compiler command generates the assembly language shown in Listing 10-2, which is saved in the file *doNothingProg.s*.

```
        .file   "doNothingProg.c"
        .intel_syntax noprefix
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
      ❶ .cfi_startproc
        endbr64
```

```
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        mov     eax, 0
        pop     rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long    1f - 0f
        .long    4f - 1f
        .long    5
0:
        .string  "GNU"
1:
        .align 8
        .long    0xc0000002
        .long    3f - 2f
2:
        .long    0x3
3:
        .align 8
4:
```

Listing 10-2: Minimum C program, assembly language generated by compiler

The first thing you might notice in Listing 10-2 is that many of the identifiers begin with a `.` character. All of them, except the ones followed by a `:`, are *assembler directives*, also known as *pseudo-ops*. They are instructions to the assembler program itself, not computer instructions. We won't need all of them for the material in this book. The identifiers that are followed by a `:` are labels on memory locations, which we'll discuss in a few pages.

Assembler Directives That We Won't Use

The assembler directives in Listing 10-2 that begin with `.cfi` ❶ tell the assembler to generate information that can be used for debugging and certain error situations. The identifiers beginning with `.LF` mark places in the code used to generate this information. A discussion of this is beyond the scope of this book, but their appearance in the listing can be confusing. So, we'll tell the compiler not to include them in the assembly language file with the `-fno-asynchronous-unwind-tables` option:

```
$ gcc -O0 -Wall -masm=intel -S -fno-asynchronous-unwind-tables doNothingProg.c
```

This produces the file *doNothingProg.s* shown in Listing 10-3.

```
        .file   "doNothingProg.c"
        .intel_syntax noprefix
        .text
        .globl  main
        .type   main, @function
main:
      ❶ endbr64
        push    rbp
        mov     rbp, rsp
        mov     eax, 0
        pop     rbp
        ret
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
        .section        .note.GNU-stack,"",@progbits
      ❷ .section        .note.gnu.property,"a"
        .align 8
        .long    1f - 0f
        .long    4f - 1f
        .long    5
0:
        .string  "GNU"
1:
        .align 8
        .long    0xc0000002
        .long    3f - 2f
2:
        .long    0x3
3:
        .align 8
4:
```

Listing 10-3: Minimum C program, assembly language generated by compiler, without `.cfi` directives

Even without the `.cfi` directives, the assembly language still includes an instruction and several directives that we won't use for now. Intel has developed a technique, *Control-flow Enforcement Technology* (*CET*), for providing better defense against types of security attacks of computer programs that hijack a program's flow. The technology is supposed to be included in Intel CPUs starting in the second half of 2020. AMD has said they will include an equivalent technology in their CPUs at a later date.

The technology includes a new instruction, `endbr64`, which is used as the first instruction in a function to check whether program flow gets there ❶. The instruction has no effect if the CPU does not include CET.

The compiler also needs to include some information for the linker to use CET. This information is placed in a special section of the file that the assembler will create, denoted with the `.section  .note.gnu.property,"a"` assembler directive ❷, after the actual program code.

The version of gcc used in this book includes the CET feature by default in anticipation of the new CPUs. The details of using CET are beyond the scope of this book. If you're curious, you can read about it at *https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html*. The programs we're writing in this book are not intended for production use, so we won't be concerned about security issues in our programs. We'll use the `-fcf-protection=none` option to tell the compiler not to include CET, and we won't use it when writing directly in assembly language.

To keep our discussion focused on the fundamentals of how a computer works, we'll tell the compiler to generate assembly language with the following command:

```
$ gcc -O0 -Wall -masm=intel -S -fno-asynchronous-unwind-tables \
> -fcf-protection=none doNothingProg1.c
```

This command yields the assembly language file shown in Listing 10-4.

```
    ❶ .file   "doNothingProg.c"
    ❷ .intel_syntax noprefix
    ❸ .text
    ❹ .globl  main
    ❺ .type   main, @function
main:
        push    rbp
        mov     rbp, rsp
        mov     eax, 0
        pop     rbp
        ret
    ❻ .size   main, .-main
    ❼ .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
    ❽ .section  .note.GNU-stack,"",@progbits
```

Listing 10-4: Minimum C program, assembly language generated by compiler, without `.cfi` directives and CET code

Now that we've stripped away the advanced features, I'll discuss the assembler directives remaining in Listing 10-4 that we won't need when writing our own assembly language. The `.file` directive ❶ is used by gcc to specify the name of the C source file that this assembly language came from. When writing directly in assembly language, this isn't used. The `.size` directive ❻ computes the size of the machine code, in bytes, that results from assembling this file, and assigns the name of this function, `main`, to this value. This can be useful information in systems with limited memory but is of no concern in our programs.

I honestly don't know the reasons for using the `.ident` and `.section` directives ❼ ❽. I'm guessing from their arguments that they're being used to provide information to the developers of gcc when users report bugs. Yes, even compilers have bugs! But we won't use these directives in our assembly language.

Assembler Directives That We Will Use

Now we'll look at the directives that will be required when we write in assembly language. The `.text` assembler directive ❸ in Listing 10-4 tells the assembler to place whatever follows in the text section. What does *text section* mean?

In GNU/Linux, the object files produced by the assembler are in the *Executable and Linking Format* (*ELF*). The ELF standard specifies many types of sections, each specifying the type of information stored in it. We use assembler directives to tell the assembler in which section to place the code.

The GNU/Linux operating system also divides memory into *segments* for specific purposes when a program is loaded from the disk. The linker gathers all the sections that belong in each segment together and outputs an executable ELF file that's organized by segment to make it easier for the operating system to load the program into memory. The four general types of segments are as follows:

Text (also called code) The *text segment* is where program instructions and constant data are stored. The operating system prevents a program from changing anything stored in the text segment, making it read-only.

Data Global variables and static local variables are stored in the *data segment*. Global variables can be accessed by any of the functions in a program. A static local variable can be accessed only by the function it's defined in, but its value remains the same between calls to its function. Programs can both read from and write to variables in the data segment. These variables remain in place for the duration of the program.

Stack Automatic local variables and the information that links functions are stored on the *call stack*. Automatic local variables are created when a function is called, and deleted when the function returns to its calling function. Memory on the stack can be both read from and written to by the program. It's allocated and deallocated dynamically as the program executes.

Heap The *heap* is a pool of memory that's available for a program to use when running. A C program calls the `malloc` function (C++ calls `new`) to get a chunk of memory from the heap. It can be both read from and written to by the program. It's used to store data and is explicitly deallocated by calling `free` (`delete` in C++) in the program.

This has been a simplistic overview of ELF sections and segments. You can find further details by reading the `man` page for ELF and reading sources like "ELF-64 Object File Format," which can be downloaded at *https://uclibc.org/docs/elf-64-gen.pdf*, and John R. Levine's *Linkers & Loaders* (Morgan Kaufmann, 1999). The `readelf` program is also useful for learning about ELF files.

Now look back at Listing 10-4. The `.globl` directive ❹ has one argument, the identifier `main`. The `.globl` directive makes the name globally known so functions that are defined in other files can refer to this name. The code that sets up the C runtime environment was written to call the function named `main`, so the name must be global in scope. All C/C++ programs start with a `main` function. In this book, we'll also start our assembly language programs with a `main` function and execute them within the C runtime environment.

You can write stand-alone assembly language programs that don't depend on the C runtime environment, in which case you can create your own name for the first function in the program. You need to stop the compilation process at the end of the assembly step with the `-c` option. You then link the object (*.o*) files using the `ld` command by itself, not as part of `gcc`. I'll describe this in more detail in Chapter 20.

The assembler directive, `.type`, ❺ has two arguments, `main` and `@function`. This causes the identifier `main` to be recorded in the object file as the name of a function.

None of these three directives gets translated into actual machine instructions, and none of them occupies any memory in the finished program. Rather, they're used to describe the characteristics of the statements that follow.

You may have noticed that I haven't yet described the purpose of the `.intel_syntax noprefix` directive ❷. It specifies the syntax of the assembly language we'll use. You can probably guess that we'll be using the Intel syntax, but that will be easier to understand after I explain the assembly language instructions. We'll do this using the same function from Listing 10-1 but written directly in assembly language.

Creating a Program in Assembly Language

Listing 10-5 was written in assembly language by a programmer, rather than by a compiler. Naturally, the programmer has added comments to improve readability.

```
❶ # doNothingProg.s
  # Minimum components of a C program, in assembly language.
          .intel_syntax noprefix
          .text
          .globl  main
          .type   main, @function
❷ main:
        ❸ push    rbp         # save caller's frame pointer
        ❹ mov     rbp, rsp    # establish our frame pointer
❺
        ❻ mov     eax, 0      # return 0 to caller

          mov     rsp, rbp    # restore stack pointer
          pop     rbp         # restore caller's frame pointer
          ret                 # back to caller
```

Listing 10-5: Minimum C-style program written in assembly language

Assembly Language in General

The first thing to notice in Listing 10-5 is that assembly language is organized by lines. Only one assembly language statement is on each line, and none of the statements spans more than one line. This differs from the free-form nature of many high-level languages where the line structure is irrelevant. In fact, good programmers use the ability to write program statements across multiple lines and indentation to emphasize the structure of their code. Good assembly language programmers use blank lines to help separate parts of an algorithm, and they comment almost every line.

Next, notice that the first two lines begin with the # character ❶. The rest of the line is written in English and is easily read. Everything after the # character is a comment. Just as with a high-level language, comments are intended solely for the human reader and have no effect on the program. The comments at the top are followed by the assembler directives we discussed earlier.

Blank lines ❺ are intended to improve readability. Well, they improve readability once you learn how to read assembly language.

The remaining lines are organized roughly into columns. They probably do not make much sense to you at this point because they're written in assembly language, but if you look carefully, each of the assembly language lines is organized into four possible fields:

```
label:   operation   operand(s)   # comment
```

Not all the lines will have entries in all the fields. The assembler requires at least one space or tab character to separate the fields. When writing assembly language, your program will be much easier to read if you use the Tab key to move from one field to the next so that the columns line up.

Let's look at each field in some detail:

Label Allows us to give a symbolic name to any line in the program. Each line corresponds to a memory location in the program, so other parts of the program can then refer to the memory location by name.

A label consists of an identifier immediately followed by the : character. You, as the programmer, must make up these identifiers. We'll look at the rules for creating an identifier soon. Only the lines we need to refer to are labeled.

Operation Contains either an *instruction operation code* (*opcode*) or an *assembler directive* (*pseudo op*). The assembler translates the opcode, along with its operands, into machine instructions, which are copied into memory when the program is to be executed.

Operand Specifies the arguments to be used in the operation. The arguments can be explicit values, names of registers, or programmer-created identifiers. The number of operands can be zero, one, two, or three, depending on the operation.

Comment Everything on a line following a `#` character is ignored by the assembler, thus providing a way for the programmer to provide human-readable comments. Since assembly language is not as easy to read as higher-level languages, good programmers will place a comment on almost every line.

A word about program comments here. Beginners often comment on what the programming statement does, not its purpose relative to solving the problem. For example, a comment like

```
counter = 1;  /* let x = 1 */
```

in C is not very useful. But a comment like

```
counter = 1;  /* need to start at 1 */
```

could be very helpful. Your comments should describe what *you* are doing, not what the computer is doing.

The rules for creating an identifier are similar to those for C/C++. Each identifier consists of a sequence of alphanumeric characters and may include other printable characters such as `.`, `_`, and `$`. The first character must not be a numeral. An identifier may be any length, and all characters are significant. Although the letter case of keyword identifiers (operators, operands, directives) is not significant, it is significant for labels. For example, `myLabel` and `MyLabel` are different. Compiler-generated labels begin with the `.` character, and many system-related names begin with the `_` character. It's a good idea to avoid beginning your own labels with the `.` or the `_` character so that you don't inadvertently create one that's already being used by the system.

It's common to place a label on its own line ❷, in which case it applies to the address of the next assembly language statement that takes up memory ❸. This allows you to create longer, more meaningful labels while maintaining the column organization of your code.

Integers can be used as labels, but they have a special meaning. They're used as local labels, which are sometimes useful in advanced assembly language programming techniques. We won't be using them in this book.

First Assembly Language Instructions

Rather than list all the x86-64 instructions (there are more than 2,000, depending on how you count), I will introduce a few at a time, and only the ones that will be needed to illustrate the programming concept at hand. I will also give only the commonly used variants of the instructions I introduce.

For a detailed description of the instructions and all their variants, you'll need a copy of *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Volume Two, which can be downloaded at *https://software.intel.com/en-us/articles/intel-sdm/*, or *AMD64 Architecture Programmer's Manual*, Volume 3: *General-Purpose and System Instructions*, which can be downloaded at *https://developer.amd.com/resources/developer-guides-manuals/*. These are the instruction set reference manuals from the two major manufacturers of x86-64 CPUs. They can be a little difficult to read, but going back and forth between my descriptions of the instructions in this book and the descriptions in the manuals should help you to learn how to read the manuals.

Assembly language provides a set of mnemonics that correspond directly to the machine language instructions. A *mnemonic* is a short, English-like group of characters that suggests the action of the instruction. For example, `mov` is used to represent the instruction that copies (moves) a value from one place to another; the machine instruction `0x4889e5` copies the entire 64-bit value in the `rsp` register to the `rbp` register. Even if you've never seen assembly language before, the mnemonic representation of this instruction in Listing 10-5 ❹ probably makes much more sense to you than the machine code.

NOTE *Strictly speaking, the mnemonics are completely arbitrary, as long as you have an assembler program that will translate them into the desired machine instructions. However, most assembler programs follow the mnemonics used in the manuals provided by CPU vendors.*

The general format of an assembly language instruction in our usage of the assembler (Intel syntax) is

```
operation destination, source1, source2
```

where *`destination`* is the location where the result of the *`operation`* will be stored, and *`source1`* and *`source2`* are the locations where the input(s) to the *`operation`* are located. There can be from zero to two sources, and some instructions don't require that you specify a destination. The destination can be a register or memory. A source value can be in a register, in memory, or *immediate data*. Immediate data is stored as part of the machine code implementation of the instruction and is hence a constant value in the program. You'll see how this works in Chapter 12, when we look at how instructions are encoded in the `1`s and `0`s of machine code.

When describing instructions, I use *`reg`*, *`reg1`*, or *`reg2`* to mean one of the names of a general-purpose register from Table 9-2 in Chapter 9. I use *`mem`* to mean a label of a memory location and *`imm`* to mean a literal data value. In most cases, the values specified by the operands must be the same. There are instructions for explicitly converting from one size to another.

Let's start with the most commonly used assembly language instruction, `mov`. In fact, in Listing 10-5 half the instructions are `mov`.

`mov`—Move

Copies a value from a source to a destination.

`mov` *`reg1, reg2`* moves the value in *`reg2`* to *`reg1`*.

`mov` *`reg, mem`* moves the value in *`mem`* to *`reg`*.

`mov` *`mem, reg`* moves the value in *`reg`* to *`mem`*.

`mov` *`reg, imm`* moves *`imm`* to *`reg`*.

`mov` *`mem, imm`* moves *`imm`* to *`mem`*.

The `mov` instruction does not affect the status flags in the `rflags` register.

The size (number of bits) of the value moved must be the same for the source and the destination. When the assembler program translates the assembly language instruction to machine code, it can figure out the size from the register name. For example, the `mov eax, 0` instruction ❻ in Listing 10-5 will cause the 32-bit integer, 0, to be stored in the `eax` register, which is the 32-bit portion of the `rax` register. Recall (from Chapter 9) that when the destination is the 32-bit portion of a register, the high-order 32 bits of that register are set to 0. If I had used `mov al, 0`, then only an 8-bit representation of 0 would be stored in the `al` portion of the `rax` register, and the other bits in the register would not be affected. For 8-bit and 16-bit operations, you should assume that the portion of any register that isn't explicitly modified by an instruction contains an unknown value.

You may have noticed that the variant that moves an immediate value to memory, `mov` *`mem, imm`*, doesn't use a register. In this case, you have to tell the assembler the data size with a size directive placed before the *`mem`* operand. Table 10-1 lists the size directives for each data size.

Table 10-1: Data Size Directives

Directive	Data type	Number of bits
`byte ptr`	Byte	8
`word ptr`	Word	16
`dword ptr`	Doubleword	32
`qword ptr`	Quadword	64

The size directive includes `ptr` because it specifies how many bytes the memory address points to. For immediate data, this address is in the `rip` register. For example,

```
        mov     byte ptr x[ebp], 123
        mov     qword ptr y[ebp], 123
```

would store 123 in the one-byte variable, `x`, and 123 in the four-byte variable, `y`. (This syntax for specifying the memory locations is explained in the next chapter.)

Notice that you can't move data from one memory location directly to another memory location. You have to first move the data into a register from memory and then move it from that register to the other memory location.

The other three instructions used in Listing 10-5 are `push`, `pop`, and `ret`. These three instructions use the call stack. We'll discuss the call stack in detail in the next chapter. For now, you can think of it as a place in memory where you can stack data items one on top of another and then remove them in reverse order. (Think of stacking dinner plates, one at a time, on a shelf and then removing each one as it's needed.) The `rsp` register always contains the address of the item on the top of the call stack; hence, it's called the *stack pointer*.

`push`—Push onto stack

Moves a 64-bit source value to the top of the call stack.

`push` *`reg`* places the 64-bit value in *`reg`* on the call stack, changing the `rsp` register such that it has the memory address of this new item on the stack.

`push` *`mem`* places the 64-bit value in *`mem`* on the call stack, changing the `rsp` register such that it has the memory address of this new item on the stack.

The `push` instruction does not affect the status flags in the `rflags` register.

`pop`—Pop from stack

Moves a 64-bit value from the top of the call stack to a destination.

`pop` *`reg`* copies the 64-bit value at the top of the stack to *`reg`*, changing the `rsp` register such that it has the memory address of the next item on the stack.

`pop` *`mem`* copies the 64-bit value at the top of the stack to *`mem`*, changing the `rsp` register such that it has the memory address of the next item on the stack.

The `pop` instruction does not affect the status flags in the `rflags` register.

`ret`—Return from function

Returns from a function call.

`ret` has no operands. It pops the 64-bit value at the top of the stack into the instruction pointer, `rip`, thus transferring program control to that memory address.

The `ret` instruction does not affect the status flags in the `rflags` register.

Now that you have an idea of how each of the instructions in Listing 10-5 works, let's see what they're doing in this program. As we walk through this code, keep in mind that this program doesn't do anything for a user. The code here forms a sort of infrastructure for any C-style function that you write. You'll see variations as you continue through the book, but you should take the time to become familiar with the basic structure of this program.

Minimal Processing in a Function

Aside from the data processing that a function does, it needs to perform some processing just so it can be called and return to the calling function. For example, the function needs to keep track of the address from where it was called so it can return to the correct place when the function has completed. Since there are a limited number of registers, the function needs a place in memory for storing the return address. After completion, the function returns to the calling place and no longer needs the return address, so it can release the memory where the return address was stored.

As you'll learn in the next chapter, the call stack is a great place for functions to temporarily store information. Each function uses a portion of the call stack for storage, which is called a *stack frame*. The function needs a reference to its stack frame, and this address is stored in the `rbp` register, usually called the *frame pointer*.

Let's walk through the actual processing that takes place in the program in Listing 10-5. I'll repeat the listing here to save you some page flipping (Listing 10-6).

```
# doNothingProg.s
# Minimum components of a C program, in assembly language.
        .intel_syntax noprefix
        .text
        .globl  main
        .type   main, @function
main:
      ❶ push    rbp         # save caller's frame pointer
      ❷ mov     rbp, rsp    # establish our frame pointer

      ❸ mov     eax, 0      # return 0 to caller

      ❹ mov     rsp, rbp    # restore stack pointer
      ❺ pop     rbp         # restore caller's frame pointer
      ❻ ret                 # back to caller
```

Listing 10-6: Code repeated for your convenience

The first thing a function must do is to save the calling function's frame pointer so the calling function can use `rbp` for its own frame pointer and then restore the calling function's frame pointer before returning. It does this by pushing the value onto the call stack ❶. Now that we've saved the calling function's frame pointer, we can use the `rbp` register as the frame pointer for the current function. The frame pointer is set to the current location of the stack pointer ❷.

NOTE *Remember that we are telling the compiler not to use any code optimization in this book with the `-O0` option to `gcc`. If you tell `gcc` to optimize the code, it may determine that these values may not need to be saved, so you wouldn't see some of these instructions. After you understand the concepts presented in this book, you can start thinking about how to optimize your code.*

This probably sounds confusing at this point. Don't worry, we'll go into this mechanism in detail in the next chapter. For now, make sure that every function you write in assembly language begins with these two instructions, in this order. Together, they make up the beginning of the *function prologue* that prepares the call stack and the registers for the actual computational work that will be done by the function.

C functions can return values to the calling function. This is the `main` function, and the operating system expects it to return the 32-bit integer 0 if the function ran without errors. The `rax` register is used to return up to a 64-bit value, so we store 0 in the `eax` register ❸ just before returning.

The function prologue prepared the call stack and registers for this function, and we need to follow a strict protocol for preparing the call stack and registers for return to the calling function. This is accomplished with the *function epilogue*. The function epilogue is essentially the mirror image of the function prologue. The first thing to do is to make sure the stack pointer is restored to where it was at the beginning of the prologue ❹. Although we can see that the stack pointer was not changed in this simple function, it will be changed in most functions, so you should get in the habit of restoring it. Restoring the stack pointer is essential for the next step to work.

Now that we've restored the stack pointer from the `rbp` register, we need to restore the calling function's value in the `rbp` register. That value was pushed onto the stack in the prologue, so we'll pop it off the top of the stack back into the `rbp` register ❺. Finally, we can return to the calling function ❻. Since this is the `main` function, this will return to the operating system.

One of the most valuable uses of `gdb` is as a learning tool. It has a mode that is especially helpful in learning what each assembly language instruction does. I'll show you how to do this in the next section, using the program in Listing 10-5. This will also help you to become more familiar with using `gdb`, which is an important skill to have when debugging your programs.

Using gdb to Learn Assembly Language

This would be a good place for you to run the program in Listing 10-5 so you can follow along with the discussion. It can be assembled, linked, and executed with the following commands:

```
$ as --gstabs -o doNothingProg.o doNothingProg.s
$ gcc -o doNothingProg doNothingProg.o
$ ./doNothingProg
```

The `--gstabs` option (note the two dashes here) tells the assembler to include debugging information with the object file. The `gcc` program recognizes that the only input file is already an object file, so it goes directly to the linking stage. There is no need to tell `gcc` to include the debugging information because it was already included in the object file by the assembler.

As you might guess from the name, you won't see anything on the screen from running this program. We'll need this for later in the chapter when we use `gdb` to walk through the execution of this program. Then you'll see that this program actually does something.

The `gdb` debugger has a mode that's useful for seeing the effects of each assembly language instruction as it's executed one step at a time. The *text user interface* (*TUI*) mode splits the terminal window into a display area at the top and the usual command area at the bottom. The display area can be further split into two display areas.

Each display area can show either the source code (`src`), the registers (`regs`), or the *disassembled* machine code (`asm`). Disassembly is the process of translating the machine code (1s and 0s) into the corresponding assembly language. The disassembly process does not know the programmer-defined names, so you will see only the numerical values that were generated by the assembly and linking processes. The `asm` display will probably be more useful when we look at the details of instructions in Chapter 12.

The documentation for using the TUI mode is in `info` for `gdb`. I'll give a simple introduction here of using the TUI mode with the program *doNothingProg.s*, from Listing 10-5. I'll step through most of the instructions. You'll get a chance to single-step through each of them when it's Your Turn.

NOTE *My example here shows `gdb` being run from the command line. I've been told that this doesn't work well if you try to run `gdb` under the `Emacs` editor.*

```
$ gdb ./doNothingProg
--snip--
Reading symbols from ./doNothingProg...
❶ (gdb) set disassembly-flavor intel
❷ (gdb) b main
Breakpoint 1 at 0x1129: file doNothingProg.s, line 8.
(gdb) r
Starting program: /home/bob/progs/chap11/doNothingProg_asm/doNothingProg

Breakpoint 1, main () at doNothingProg.s:8
8               push    rbp              # save caller's frame pointer
❸ (gdb) tui enable
```

We start the program under `gdb` the usual way. The default assembly language syntax that `gdb` uses for disassembly under GNU/Linux is AT&T, so we need to set it to Intel ❶. This syntax issue will be explained at the end of this chapter. It matters if you use the `asm` display.

Then we set a breakpoint at the beginning of the program ❷. We used source code line numbers for setting breakpoints in C code. But each C statement typically translates into several assembly language instructions, so we can't be sure that `gdb` will break at a specific instruction. The *`label`* syntax gives us a way to ensure that `gdb` will break at a specific instruction if it is labeled.

When we run the program, it breaks at the `main` label, which is on the first instruction of the function. Next, we enable the TUI mode ❸, which shows the source code, as shown in Figure 10-1.

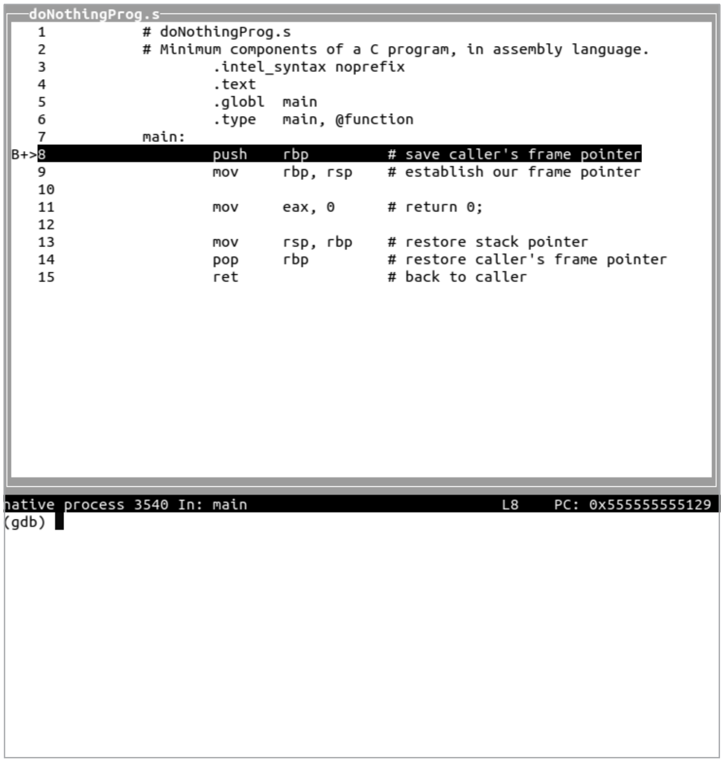

Figure 10-1: `gdb` in TUI mode with `src` display

The bottom section of the terminal window shows the usual `(gdb)` prompt, which is where you enter `gdb` commands and examine memory contents. The top section shows the source code for this function with the line about to be executed shown in reverse video to highlight it. There's also an indication on the left side that there's a breakpoint at this line (`B+`) and that the instruction pointer, `rip`, currently points to this line, `>`. The display also shows the current address in the `rip` register, using the name `PC`, in the lower-right margin of the source display section. (*Program counter* is another name for *instruction pointer*.)

The `layout regs` command splits the display area of the terminal window and displays the registers, as shown in Figure 10-2. We're about to execute the first instruction in the `main` function.

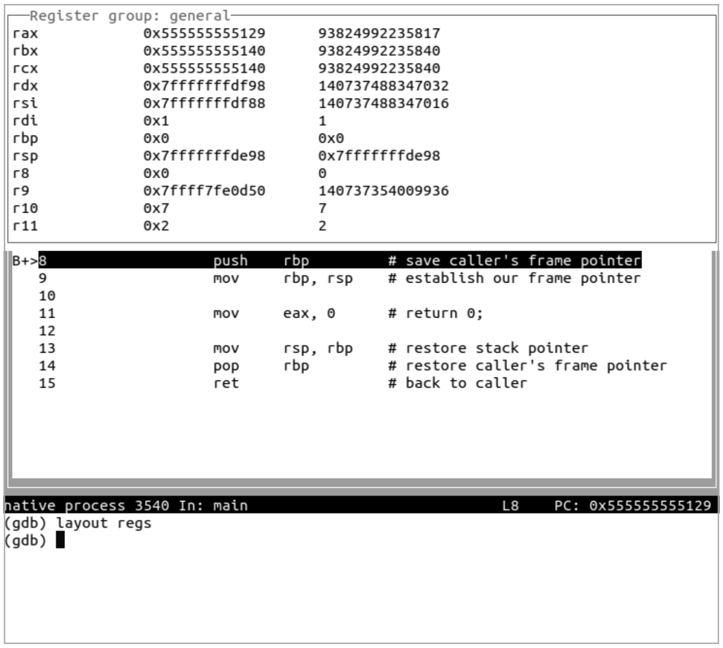

Figure 10-2: gdb in TUI mode with the source and registers windows

The s command executes the current instruction and moves on to the next instruction, which becomes highlighted, as shown in Figure 10-3.

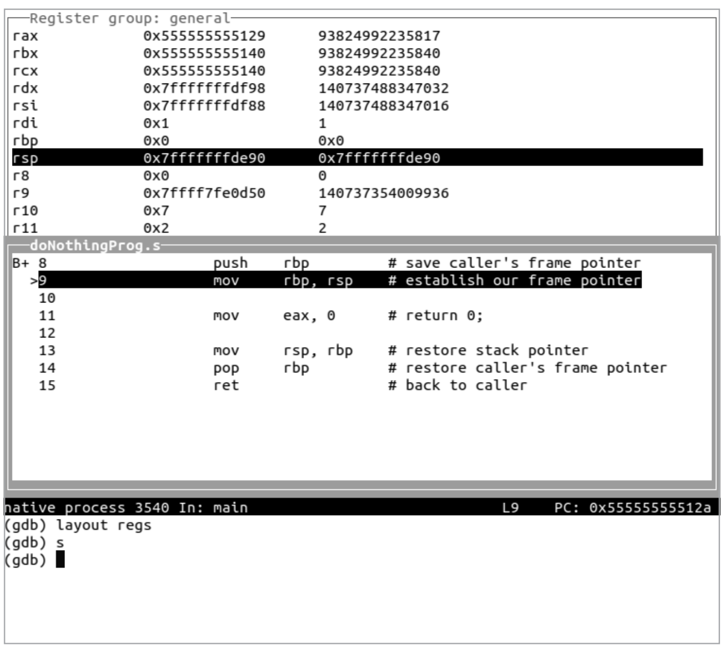

Figure 10-3: Executing an instruction causes any registers that have changed to be highlighted.

Executing the first instruction, `push rbp`, has caused `gdb` to highlight the `rsp` register and its contents in the registers display window shown in Figure 10-3. This instruction has pushed the contents of the `rbp` register onto the call stack and changed the stack pointer, `rsp`, accordingly. Pushing a 64-bit register onto the call stack has changed the stack pointer from `0x7fffffffde98` (Figure 10-2) to `0x7fffffffde90`; that is, it decremented the stack pointer by the number of bytes (8) pushed onto the stack. You'll learn more about the call stack and its usage in the next chapter.

In Figure 10-3, you can also see that the current location within the program has moved to the next instruction. This instruction is now highlighted; the instruction pointer character, `>`, has moved to this instruction; and the address in the `rip` register (`PC` in lower right) has changed from `0x555555555129` to `0x55555555512a`. This change in `rip` shows that the instruction that was just executed, `push rbp`, occupies only one byte in memory. You'll learn more about this in Chapter 12.

The TUI enhancement does not provide a data or address view of memory, only a disassembly view. We need to view data and addresses that are stored in memory in the command area. For example, if we want to see what the `push rbp` instruction stored in memory, we need to use the `x` command to view the memory pointed to by the stack pointer, `rsp`. Figure 10-4 shows the giant (64-bit) contents in hexadecimal at the memory address in `rsp`.

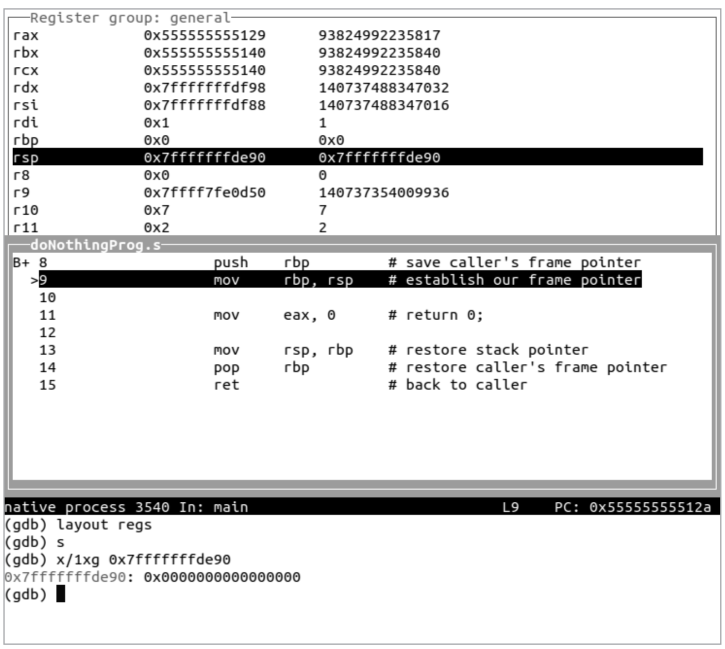

Figure 10-4: Examining memory in TUI mode is done in the command area.

Executing two more instructions shows that the `mov rax, 0` instruction stores 0 in the `rax` register, as shown in Figure 10-5. Comparing Figures 10-4 and 10-5, you can also see the effects of the `mov rbp, rsp` instruction.

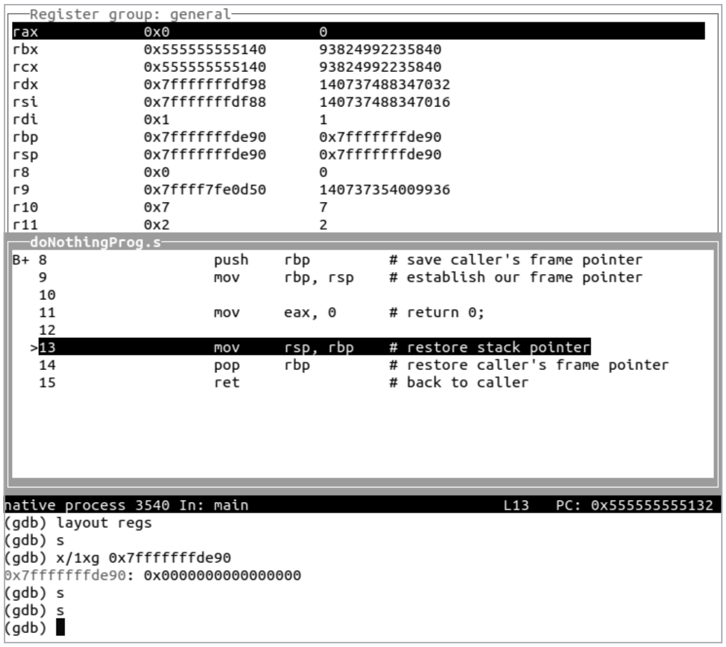

Figure 10-5: Effects of the `mov eax, 0` *instruction*

Another step takes us to the ret instruction, shown in Figure 10-6, ready to return to the calling function.

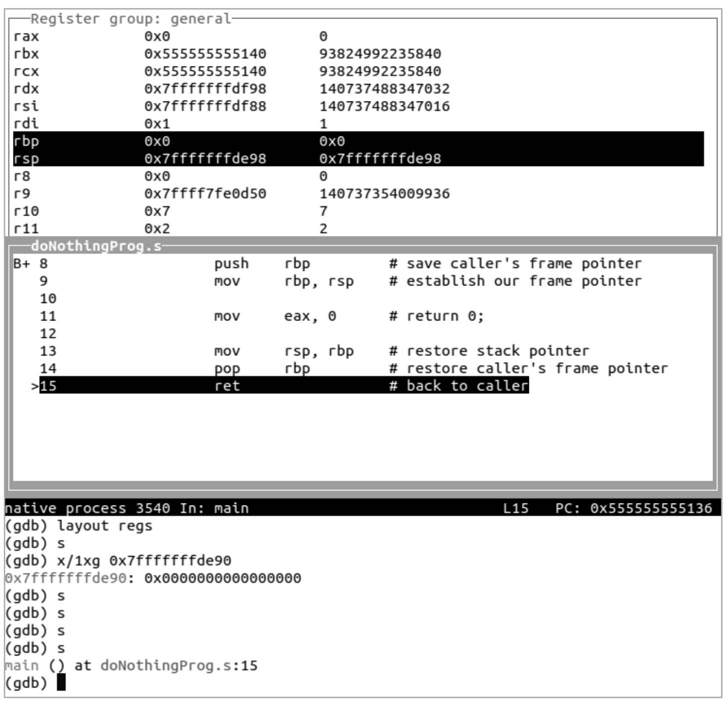

Figure 10-6: Ready to return to the calling function

Comparing Figure 10-6 with Figure 10-2 shows us that the frame pointer, `rbp`, has been restored to the calling function's value. We can also see that the stack pointer, `rsp`, has been moved back to the same location it was at when our function first started. If both the frame pointer and stack pointer are not restored before returning to the calling function, it's almost certain that your program will crash. For this reason, I often set a breakpoint at the `ret` instruction so I can check that my function restored both these registers properly, highlighted in Figure 10-7.

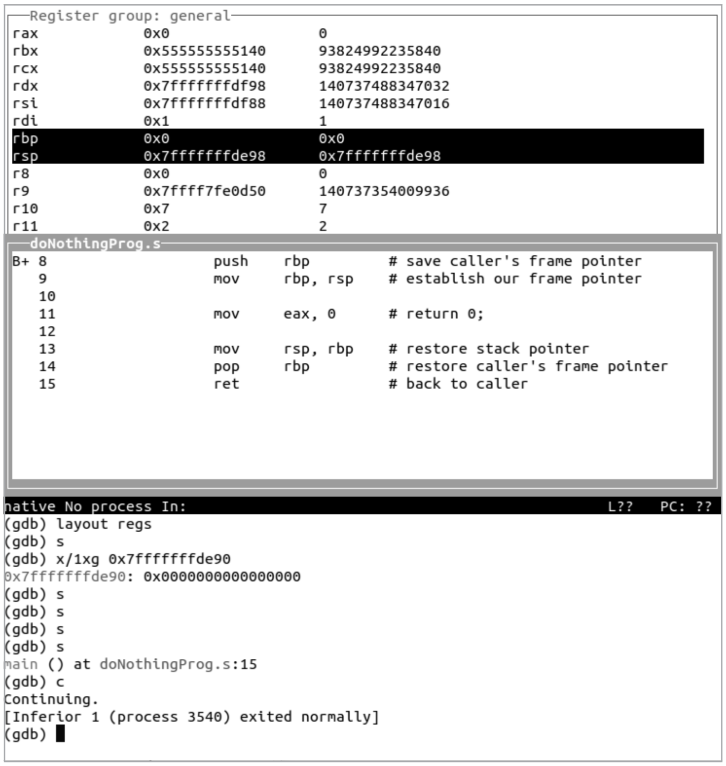

Figure 10-7: The program has completed.

All that remains is to quit `gdb`.

YOUR TURN

1. Enter the program in Listing 10-5 and use `gdb` to single-step through the code. Notice that when you execute the `mov rsp, rbp` instruction in the epilogue, TUI does not highlight the registers. Explain. Next, change the program so that it returns the integer 123. Run it with `gdb`. What number base does `gdb` use to display the exit code?

2. Enter the program in Listing 10-1 and compile it with debugging turned on (-g option). Set a breakpoint at main. Does gdb break at the entry to the function? Can you follow the actions of the prologue by using the s command? Can you continue through the program and step through the epilogue?
3. Write the following C function in assembly language:

```
/* f.c */
  int f(void) {
  return 0;
}
```

   Make sure that it assembles with no errors. Use the -S option to compile *f.c* and compare gcc's assembly language with yours. Write a main function in C that tests your assembly language function, f, and prints out the function's return value.
4. Write three assembly language functions that do nothing but return an integer. They should each return a different, nonzero integer. Write a main function in C that tests your assembly language functions and prints out the functions' return values by using printf.
5. Write three assembly language functions that do nothing but return a character. Each should return a different character. Write a main function in C that tests your assembly language functions and prints out the functions' return values by using printf.

In the next chapter, we'll take a more detailed look inside the main function. I'll describe how to use the call stack in detail. This will include how to create local variables in a function. But first, I'll give a brief summary of the AT&T assembly language syntax. If you look at any assembly language in a Linux or Unix environment, you'll probably see the AT&T syntax being used.

AT&T Syntax

I am using the Intel syntax for the assembly language in this book, but for those who might prefer the AT&T syntax, I'll briefly describe it here. AT&T syntax is the default in most Linux distributions.

Listing 10-7 is a repeat of the program in Listing 10-5 but written using the AT&T syntax.

```
# doNothingProg_att.s
# Minimum components of a C program, in assembly language.
        .text
        .globl  main
        .type   main, @function
```

```
main:
   ❶ pushq ❷ %rbp       # save caller's frame pointer
     movq  ❸ %rsp, %rbp  # establish our frame pointer

     movq  ❹ $0, %rax    # return 0;

     movq    %rbp, %rsp  # restore stack pointer
     popq    %rbp        # restore caller's frame pointer
     ret                 # back to caller
```

Listing 10-7: Minimum C program written in assembly language using AT&T syntax

The first difference that you probably notice is that a character specifying the size of the operand is added as a suffix to most instruction mnemonics ❶. Table 10-2 lists the size letters. (Yes, this is redundant in the cases where one of the operands is a register, but it's part of the syntax.) The next difference you probably see is that each register is prefixed with the `%` character ❷.

The most significant difference is that the order of the operands is reversed ❸. Instead of placing the destination first, it's last. If you move between the two syntaxes, Intel and AT&T, it's easy to get the operands in the wrong order, especially with instructions that use two registers. You also need to prefix an immediate data value with the `$` character ❹ in the AT&T syntax.

Table 10-2: Data Size Suffix for AT&T Syntax

Suffix letter	Data type	Number of bits
b	Byte	8
w	Word	16
l	Doubleword	32
q	Quadword	64

As stated in the preface, I chose to use the Intel syntax in this book to be consistent with the Intel and AMD manuals. As far as I know, the GNU assembler, `as`, is the only one that defaults to the AT&T syntax. All other assemblers use the Intel syntax, and `as` offers that as an option.

What You've Learned

Editor A program used to write the source code for a program in the chosen programming language.

Preprocessor The first stage of compilation. It brings other files into the source, defines macros, and so forth, in preparation for actual compilation.

Compilation Translates from the chosen programming language into assembly language.

Assembly Translates assembly language into machine language.

Linking Links separate object code modules and libraries together to produce the final executable program.

Assembler directives Guide the assembler program during the assembly process.

`mov` instruction Moves values between memory and the CPU and within the CPU.

`push` instruction Places values on the call stack.

`pop` instruction Retrieves values from the call stack.

`ret` instruction Returns program flow to the calling function.

`gdb` TUI mode Displays changes in registers in real time as you step through a program. It's an excellent learning tool.

Prologue Sets up the call stack for the called function.

Epilogue Restores the call stack for the calling function.

In the next chapter, you'll learn the details about how to pass arguments to functions, how the call stack works, and how to create local variables in functions.