## 7.5. Functions in Assembly

In the previous section, we traced through simple functions in assembly. In this section, we discuss the interaction between multiple functions in assembly in the context of a larger program. We also introduce some new instructions involved with function management.

Let's begin with a refresher on how the call stack is managed. Recall that `%rsp` is the **stack pointer** and always points to the top of the stack. The register `%rbp` represents the base pointer (also known as the **frame pointer**) and points to the base of the current stack frame. The **stack frame** (also known as the **activation frame** or the **activation record**) refers to the portion of the stack allocated to a single function call. The currently executing function is always at the top of the stack, and its stack frame is referred to as the **active frame**. The active frame is bounded by the stack pointer (at the top of stack) and the frame pointer (at the bottom of the frame). The activation record typically holds local variables for a function.

Figure 1 shows the stack frames for `main` and a function it calls named `fname`. We will refer to the `main` function as the *caller* function and `fname` as the *callee* function.

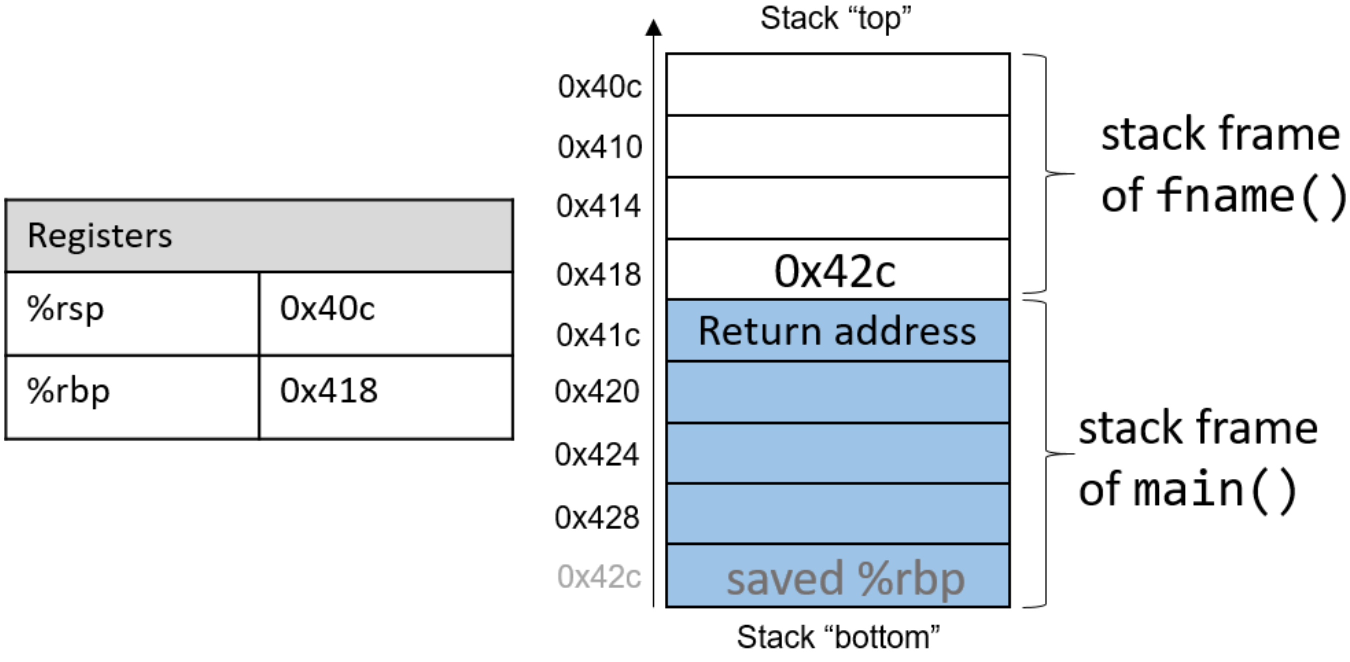



*Figure 1. Stack frame management*

In Figure 1, the current active frame belongs to the callee function ( `fname` ). The memory between the stack pointer and the frame pointer is used for local variables. The stack pointer moves as local values are pushed and popped from the stack. In contrast, the frame pointer remains relatively constant, pointing to the beginning (the bottom) of the current stack frame. As a result, compilers like GCC commonly reference values on the stack relative to the frame pointer. In Figure 1, the active frame is bounded below by the base pointer of `fname`, which is stack address 0x418. The value stored at address 0x418 is the "saved" `%rbp` value (0x42c), which itself is an address that indicates the bottom of the activation frame for the `main` function. The top of the activation frame of `main` is bounded by the **return address**, which indicates where in the `main` function program execution resumes once the callee function `fname` finishes executing.

WARNING

*The return address points to code segment memory, not stack memory*

Recall that the call stack region (stack memory) of a program is different from its code region (code segment memory). While `%rbp` and `%rsp` point to addresses in the stack memory, `%rip` points to an address in *code* segment memory. In other words, the return address is an address in code segment memory, not stack memory:

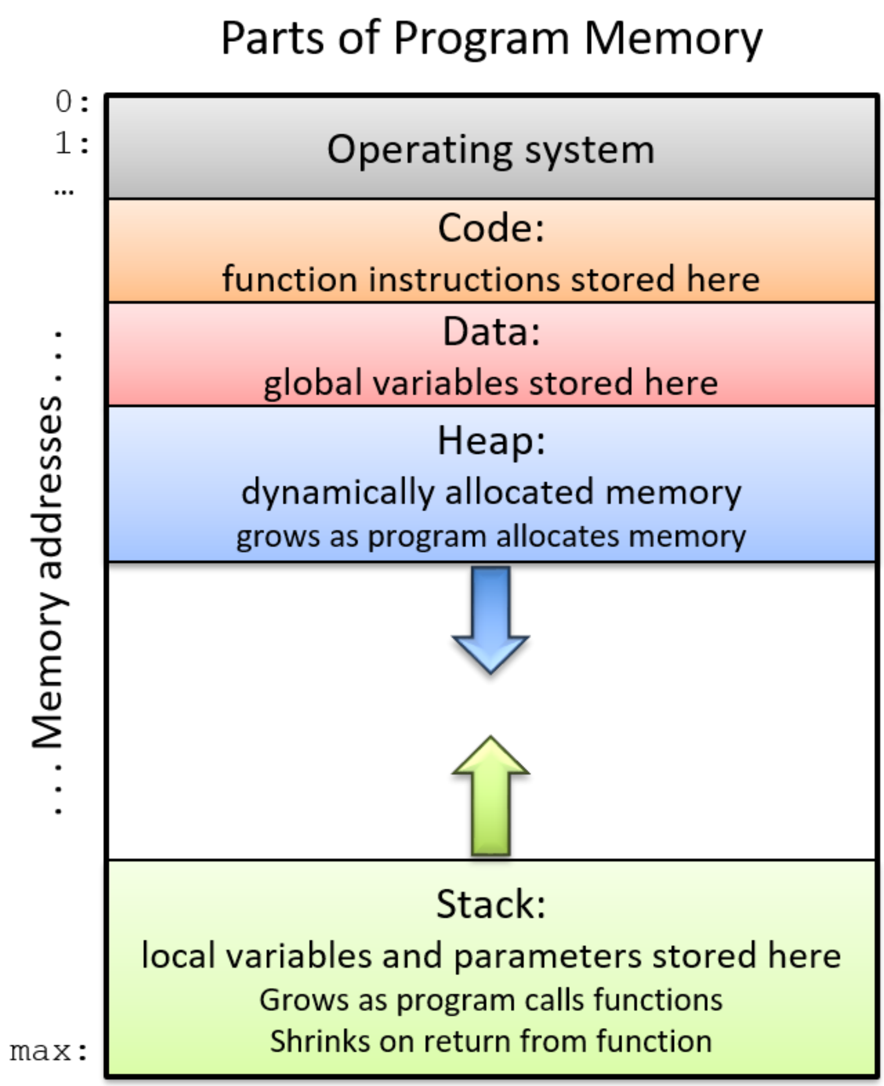



*Figure 2. The parts of a program's address space*

Table 1 contains several additional instructions that the compiler uses for basic function management.

*Table 1. Common Function Management Instructions*

| Instruction | Translation |
|---|---|
| `leaveq` | Prepares the stack for leaving a function. Equivalent to:<br>`mov %rbp, %rsp`<br>`pop %rbp` |
| `callq addr <fname>` | Switches active frame to callee function. Equivalent to:<br>`push %rip`<br>`mov addr, %rip` |
| `retq` | Restores active frame to caller function. Equivalent to:<br>`pop %rip` |

For example, the `leaveq` instruction is a shorthand that the compiler uses to restore the stack and frame pointers as it prepares to leave a function. When the callee function finishes execution, `leaveq` ensures that the frame pointer is *restored* to its previous value.

The `callq` and `retq` instructions play a prominent role in the process where one function calls another. Both instructions modify the instruction pointer (register `%rip`). When the caller function executes the `callq` instruction, the current value of `%rip` is saved on the stack to represent the return address, or the program address at which the caller resumes executing once the callee function finishes. The `callq` instruction also replaces the value of `%rip` with the address of the callee function.

The `retq` instruction restores the value of `%rip` to the value saved on the stack, ensuring that the program resumes execution at the program address specified in the caller function. Any value returned by the callee is stored in `%rax` or one of its component registers (e.g., `%eax`). The `retq` instruction is usually the last instruction that executes in any function.

### 7.5.1. Function Parameters

Unlike IA32, function parameters are typically preloaded into registers prior to a function call. Table 2 lists the parameters to a function and the register (if any) that they are loaded into prior to a function call.

*Table 2. Locations of Function Parameters.*

| Parameter | Location |
|---|---|
| Parameter 1 | %rdi |
| Parameter 2 | %rsi |
| Parameter 3 | %rdx |
| Parameter 4 | %rcx |
| Parameter 5 | %r8 |
| Parameter 6 | %r9 |
| Parameter 7+ | on call stack |

The first six parameters to a function are loaded into registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`, respectively. Any additional parameters are successively loaded into the call stack based on their size (4 byte offsets for 32-bit data, 8 byte offsets for 64-bit data).

### 7.5.2. Tracing Through an Example

Using our knowledge of function management, let's trace through the code example first introduced at the beginning of this chapter. Note that the `void` keyword is added to the parameter list of each function definition to specify that the functions take no arguments. This change does not modify the output of the program; however, it does simplify the corresponding assembly.

```c
#include <stdio.h>

int assign(void) {
    int y = 40;
    return y;
}

int adder(void) {
    int a;
    return a + 2;
}

int main(void) {
    int x;
    assign();
```

```
    x = adder();
    printf("x is: %d\n", x);
    return 0;
}
```

We compile this code with the command `gcc -o prog prog.c` and use `objdump -d` to view the underlying assembly. The latter command outputs a pretty big file that contains a lot of information that we don't need. Use `less` and the search functionality to extract the `adder`, `assign`, and `main` functions:

```
0000000000400526 <assign>:
  400526:       55                      push   %rbp
  400527:       48 89 e5                mov    %rsp,%rbp
  40052a:       c7 45 fc 28 00 00 00    movl   $0x28,-0x4(%rbp)
  400531:       8b 45 fc                mov    -0x4(%rbp),%eax
  400534:       5d                      pop    %rbp
  400535:       c3                      retq

0000000000400536 <adder>:
  400536:       55                      push   %rbp
  400537:       48 89 e5                mov    %rsp,%rbp
  40053a:       8b 45 fc                mov    -0x4(%rbp),%eax
  40053d:       83 c0 02                add    $0x2,%eax
  400540:       5d                      pop    %rbp
  400541:       c3                      retq

0000000000400542 <main>:
  400542:       55                      push   %rbp
  400543:       48 89 e5                mov    %rsp,%rbp
  400546:       48 83 ec 10             sub    $0x10,%rsp
  40054a:       e8 e3 ff ff ff          callq  400526 <assign>
  40054f:       e8 d2 ff ff ff          callq  400536 <adder>
  400554:       89 45 fc                mov    %eax,-0x4(%rbp)
  400557:       8b 45 fc                mov    -0x4(%rbp),%eax
  40055a:       89 c6                   mov    %eax,%esi
  40055c:       bf 04 06 40 00          mov    $0x400604,%edi
  400561:       b8 00 00 00 00          mov    $0x0,%eax
  400566:       e8 95 fe ff ff          callq  400400 <printf@plt>
  40056b:       b8 00 00 00 00          mov    $0x0,%eax
```

```
  400570:       c9                              leaveq
  400571:       c3                              retq
```

Each function begins with a symbolic label that corresponds to its declared name in the program. For example, `<main>:` is the symbolic label for the `main` function. The address of a function label is also the address of the first instruction in that function. To save space in the figures below, we truncate addresses to the lower 12 bits. So, program address 0x400542 is shown as 0x542.

### 7.5.3. Tracing Through main

Figure 3 shows the execution stack immediately prior to the execution of `main`.

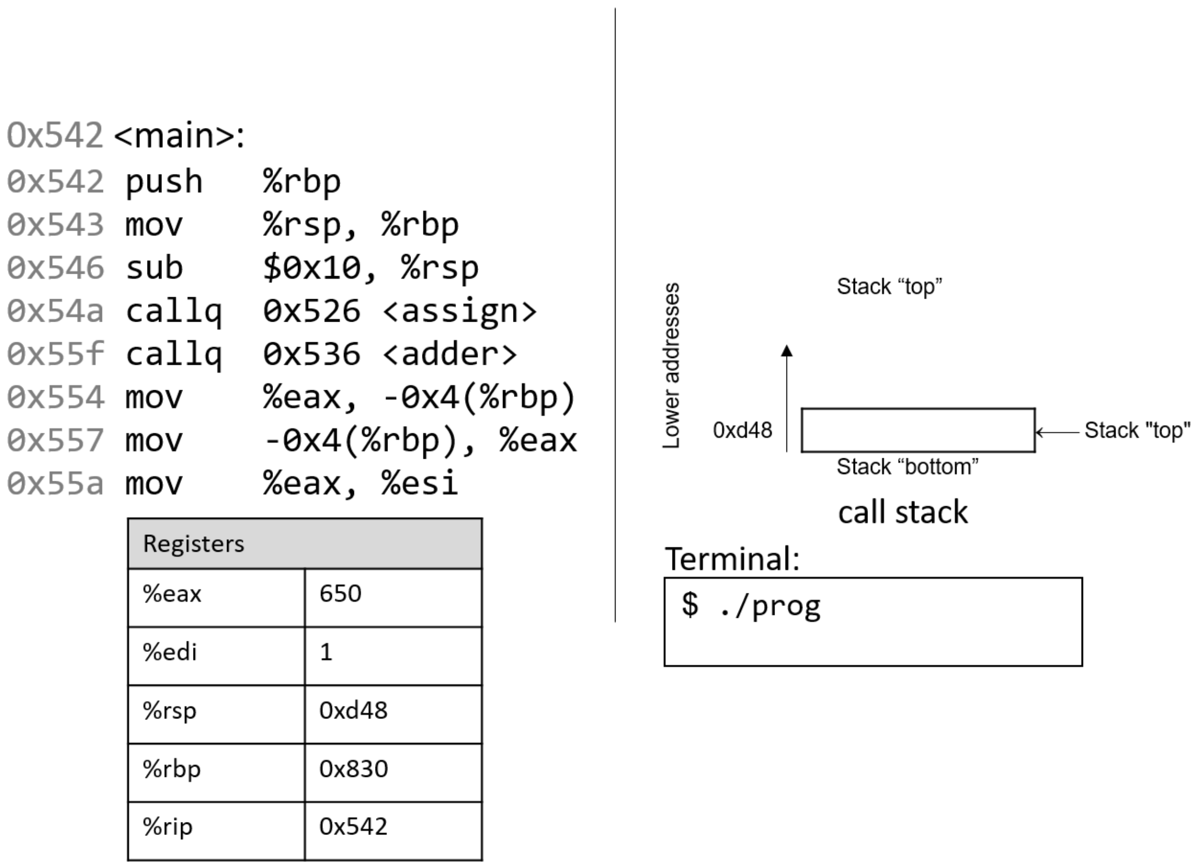


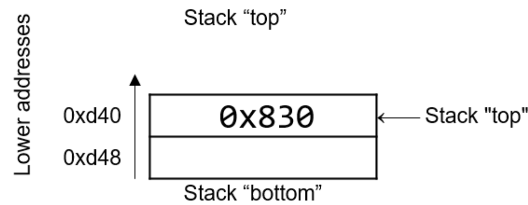

*Figure 3. The initial state of the CPU registers and call stack prior to executing the main function*

Recall that the stack grows toward lower addresses. In this example, `%rbp` initially is stack address 0x830, and `%rsp` initially is stack address 0xd48. Both of these values are made up for this example.

Since the functions shown in the previous example utilize integer data, we highlight component registers `%eax` and `%edi`, which initially contain junk values. The red (upper-left) arrow indicates the currently executing instruction. Initially, `%rip` contains address 0x542, which is the program memory address of the first line in the `main` function.

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 650 |
| %edi | 1 |
| %rsp | **0xd40** |
| %rbp | 0x830 |
| %rip | **0x543** |




call stack

Terminal:

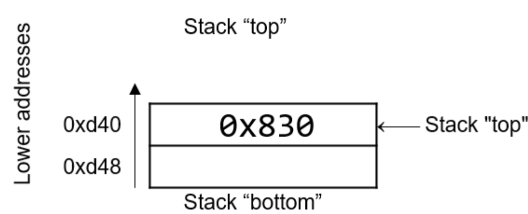
```
$ ./prog
```

The first instruction saves the current value of `%rbp` by pushing 0x830 onto the stack. Since the stack grows toward lower addresses, the stack pointer `%rsp` is updated to 0xd40, which is 8 bytes less than 0xd48. `%rip` advances to the next instruction in sequence.

---

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 650 |
| %edi | 1 |
| %rsp | 0xd40 |
| %rbp | **0xd40** |
| %rip | **0x546** |




call stack

Terminal:

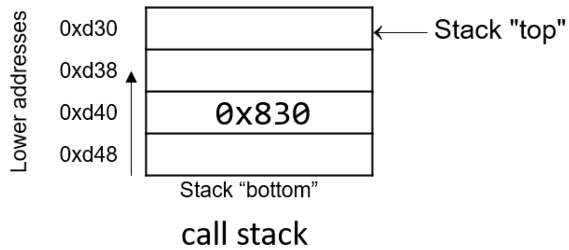
```
$ ./prog
```

The next instruction ( `mov %rsp, %rbp` ) updates the value of `%rbp` to be the same as `%rsp` . The frame pointer ( `%rbp` ) now points to the start of the stack frame for the `main` function. `%rip` advances to the next instruction in sequence.

---

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 650 |
| %edi | 1 |
| %rsp | **0xd30** |
| %rbp | 0xd40 |
| %rip | **0x54a** |




Terminal:

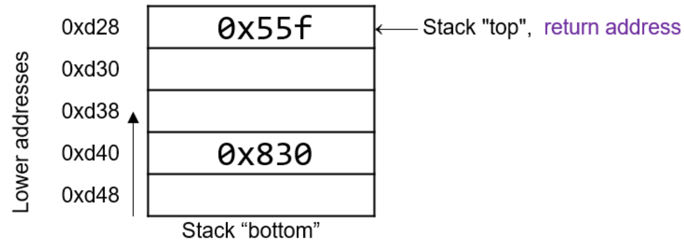
```
$ ./prog
```

The `sub` instruction subtracts 0x10 from the address of our stack pointer, which essentially causes the stack to "grow" by 16 bytes, which we represent by showing two 8-byte locations on the stack. Register `%rsp` therefore has the new value of 0xd30. `%rip` advances to the next instruction in sequence.

---

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x0 |
| %edi | 1 |
| %rsp | **0xd28** |
| %rbp | 0xd40 |
| %rip | **0x526** |




call stack

Terminal:

```
$ ./prog
```

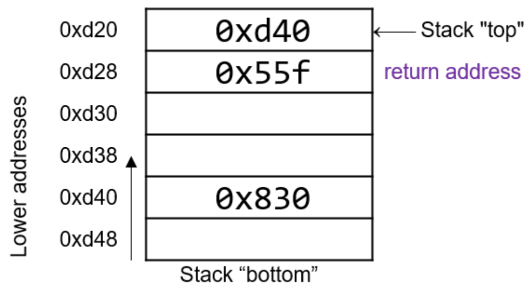
Equivalent to:
push %rip
mov 0x526, %rip

The `callq <assign>` instruction pushes the value inside register `%rip` (which denotes the address of the *next* instruction to execute) onto the stack. Since the next instruction after `callq <assign>` has an address of 0x55f, that value is pushed onto the stack as the return address. Recall that the return address indicates the program address where execution should resume when program execution returns to `main`.

Next, the `callq` instruction moves the address of the `assign` function (0x526) into register `%rip`, signifying that program execution should continue into the callee function `assign` and not the next instruction in `main`.

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x0 |
| %edi | 1 |
| %rsp | **0xd20** |
| %rbp | 0xd40 |
| %rip | **0x527** |




call stack

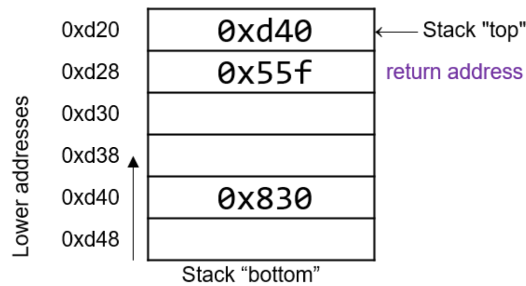
Terminal:

```
$ ./prog
```

The first two instructions that execute in the `assign` function are the usual book-keeping that every function performs. The first instruction pushes the value stored in `%rbp` (memory address 0xd40) onto the stack. Recall that this address points to the beginning of the stack frame for `main`. `%rip` advances to the second instruction in `assign`.

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x0 |
| %edi | 1 |
| %rsp | 0xd20 |
| %rbp | **0xd20** |
| %rip | **0x52a** |




call stack

Terminal:

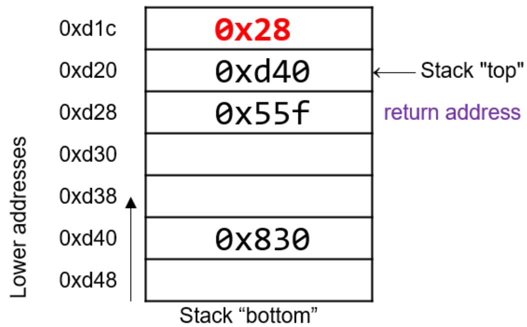
```
$ ./prog
```

The next instruction ( `mov %rsp, %rbp` ) updates `%rbp` to point to the top of the stack, marking the beginning of the stack frame for `assign` . The instruction pointer ( `%rip` ) advances to the next instruction in the `assign` function.

---

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x0 |
| %edi | 1 |
| %rsp | 0xd20 |
| %rbp | 0xd20 |
| %rip | **0x531** |




call stack

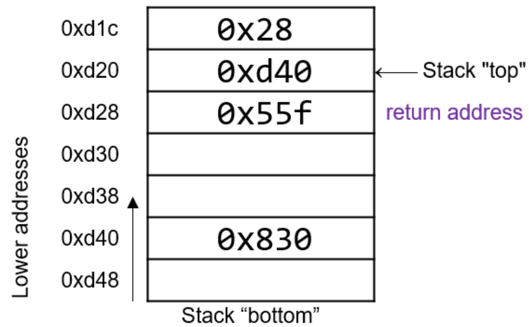
Terminal:

```
$ ./prog
```

The `mov` instruction at address 0x52a moves the value `$0x28` (or 40) onto the stack at address `-0x4(%rbp)`, which is four bytes above the frame pointer. Recall that the frame pointer is commonly used to reference locations on the stack. However, keep in mind that this operation does not change the value of `%rsp` — the stack pointer still points to address 0xd20. Register `%rip` advances to the next instruction in the `assign` function.

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | **0x28** |
| %edi | 1 |
| %rsp | 0xd20 |
| %rbp | 0xd20 |
| %rip | **0x534** |




call stack

Terminal:

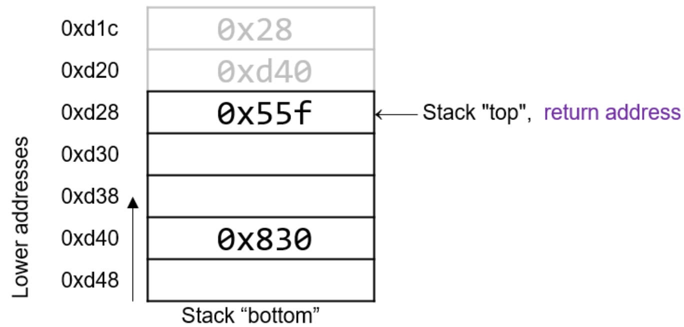
```
$ ./prog
```

The `mov` instruction at address 0x531 places the value `$0x28` into register `%eax`, which holds the return value of the function. `%rip` advances to the `pop` instruction in the `assign` function.

---

```
0x526 <assign>:
0x526 push   %rbp
0x527 mov    %rsp, %rbp
0x52a mov    $0x28, -0x4(%rbp)
0x531 mov    -0x4(%rbp), %eax
0x534 pop    %rbp
0x535 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x28 |
| %edi | 1 |
| %rsp | **0xd28** |
| %rbp | **0xd40** |
| %rip | **0x535** |

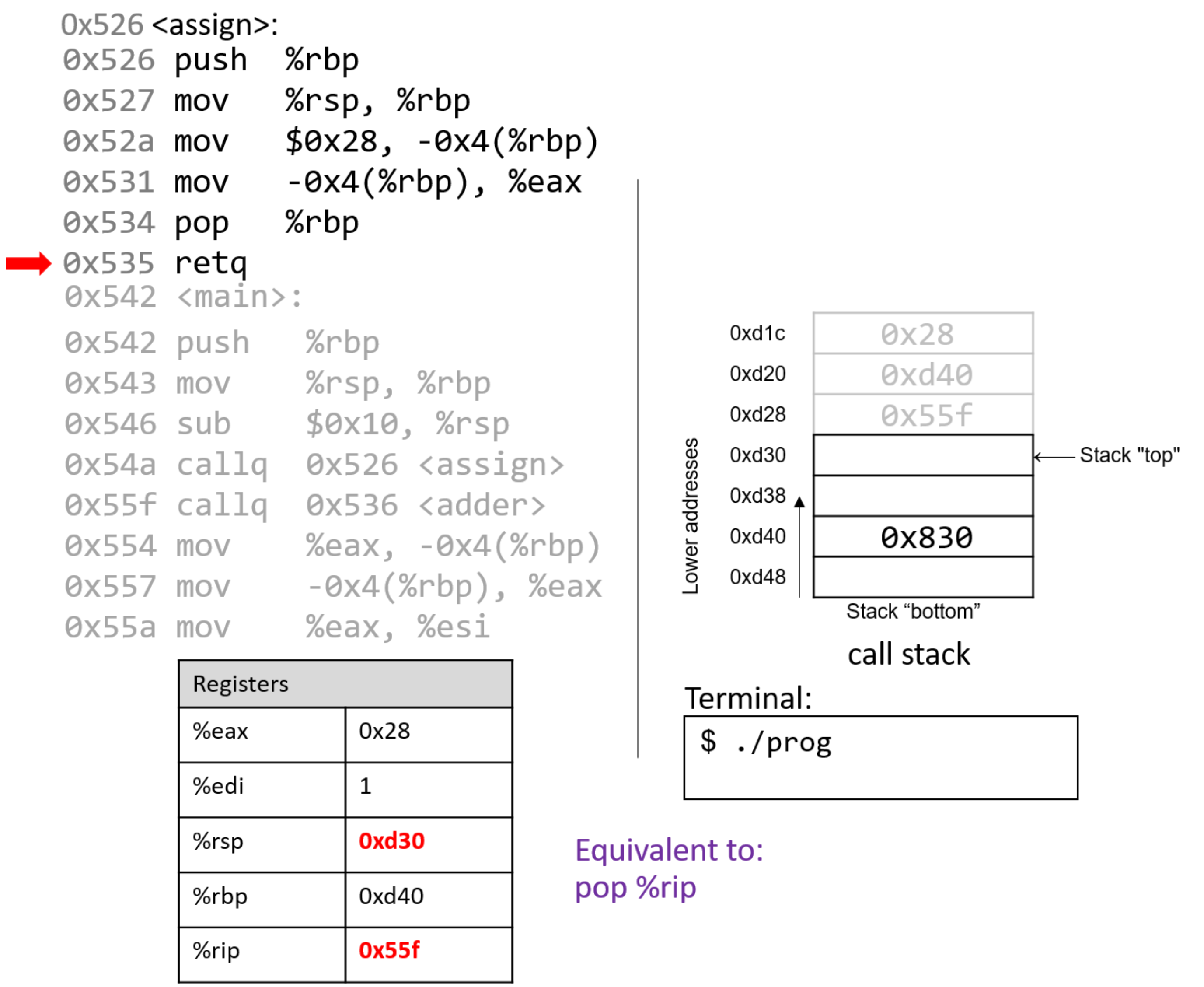



call stack

Terminal:

```
$ ./prog
```

At this point, the `assign` function has almost completed execution. The next instruction that executes is `pop %rbp`, which restores `%rbp` to its previous value, or 0xd40. Since the `pop` instruction modifies the stack pointer, `%rsp` updates to 0xd28.

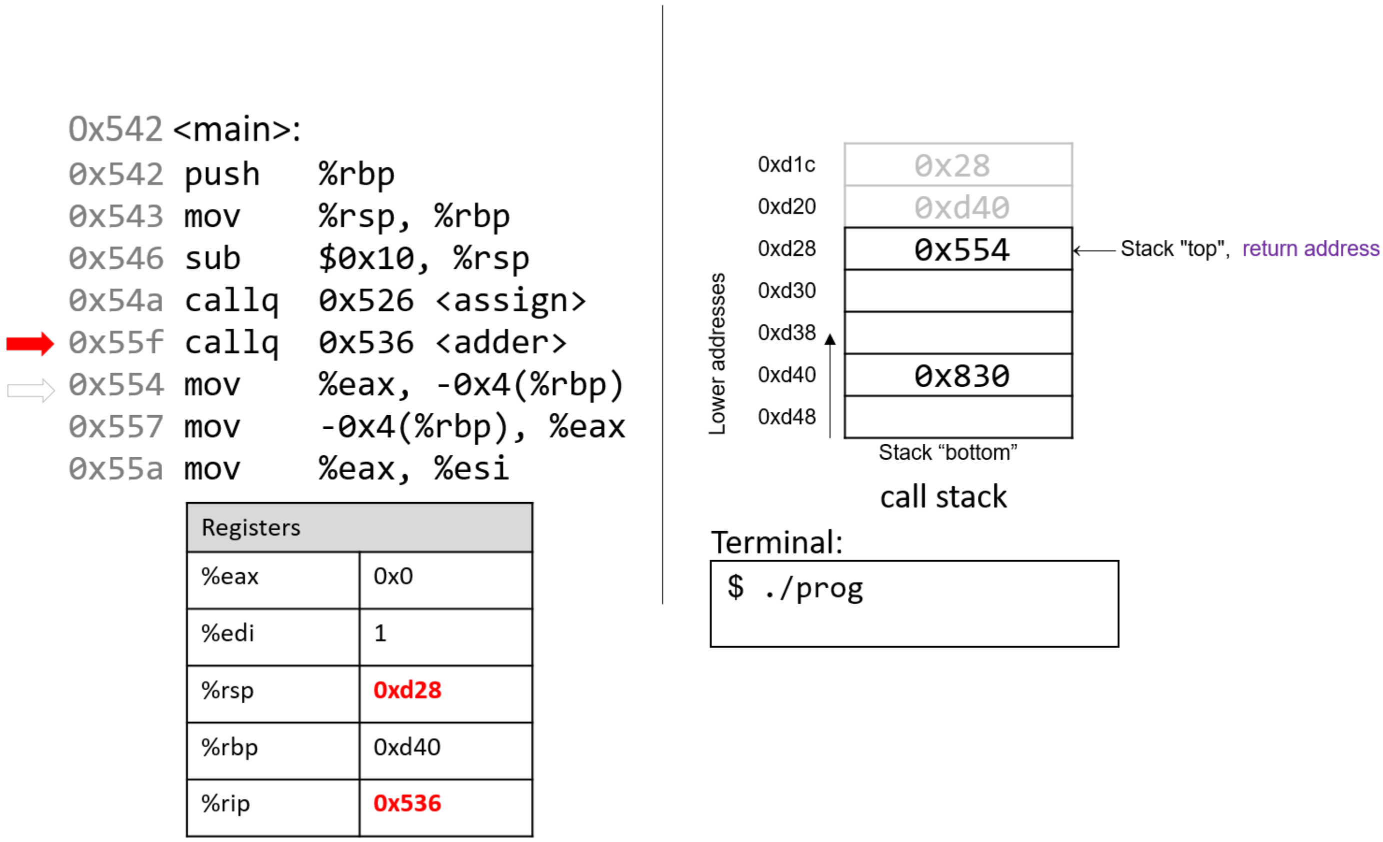



The last instruction in `assign` is a `retq` instruction. When `retq` executes, the return address is popped off the stack into register `%rip`. In our example, `%rip` now advances to point to the `callq` instruction in `main` at address 0x55f.

Some important things to notice at this juncture:

- The stack pointer and the frame pointer have been restored to their values prior to the call to `assign`, reflecting that the stack frame for `main` is once again the active frame.
- The old values on the stack from the prior active stack frame are *not* removed. They still exist on the call stack.

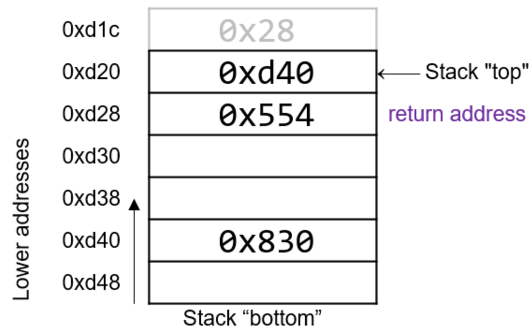
---




Back in `main`, the call to `adder` *overwrites* the old return address on the stack with a new return address (0x554). This return address points to the next instruction to be executed after `adder` returns, or `mov %eax, -0x4(%rbp)`. Register `%rip` updates to point to the first instruction to execute in `adder`, which is at address 0x536.

```
    0x536 <adder>:
➡   0x536 push  %rbp
    0x537 mov   %rsp, %rbp
    0x53a mov   $-0x4(%rbp), %eax
    0x53d add   $0x2, %eax
    0x540 pop %rbp
    0x541 retq
    0x542 <main>:
    0x542 push    %rbp
    0x543 mov     %rsp, %rbp
    0x546 sub     $0x10, %rsp
    0x54a callq   0x526 <assign>
    0x55f callq   0x536 <adder>
    0x554 mov     %eax, -0x4(%rbp)
    0x557 mov     -0x4(%rbp), %eax
    0x55a mov     %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x0 |
| %edi | 1 |
| %rsp | **0xd20** |
| %rbp | 0xd40 |
| %rip | **0x537** |




call stack

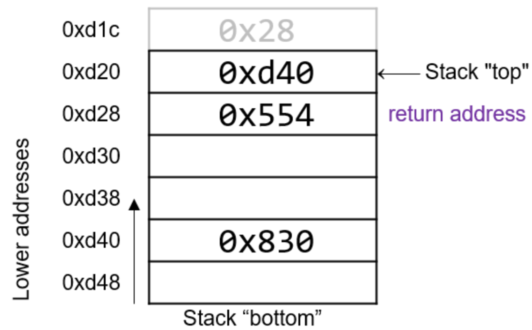
Terminal:

```
$ ./prog
```

The first instruction in the `adder` function saves the caller’s frame pointer (`%rbp` of `main`) on the stack.

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x0 |
| %edi | 1 |
| %rsp | 0xd20 |
| %rbp | **0xd20** |
| %rip | **0x53a** |




call stack

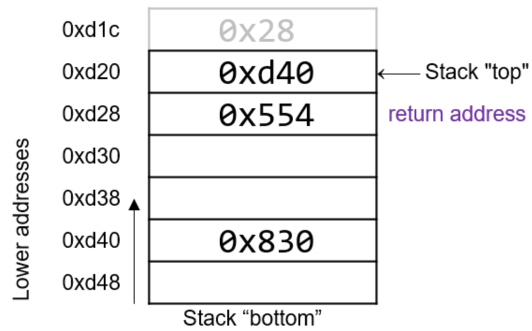
Terminal:

```
$ ./prog
```

The next instruction updates `%rbp` with the current value of `%rsp`, or address 0xd20. Together, these last two instructions establish the beginning of the stack frame for `adder`.

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | **0x28** |
| %edi | 1 |
| %rsp | 0xd20 |
| %rbp | 0xd20 |
| %rip | **0x53d** |




call stack

Terminal:

```
$ ./prog
```

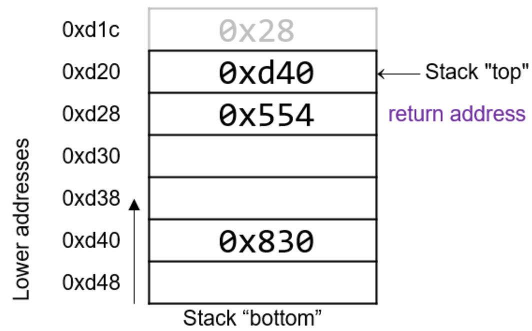
Using an old value on the stack!

Pay close attention to the next instruction that executes. Recall that `$0x28` was placed on the stack during the call to `assign`. The `mov $-0x4(%rbp), %eax` instruction moves an *old* value that is on the stack into register `%eax`! This would not have occurred if the programmer had initialized variable `a` in the `adder` function.

---

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | **0x2A** |
| %edi | 1 |
| %rsp | 0xd20 |
| %rbp | 0xd20 |
| %rip | **0x540** |




call stack

Terminal:

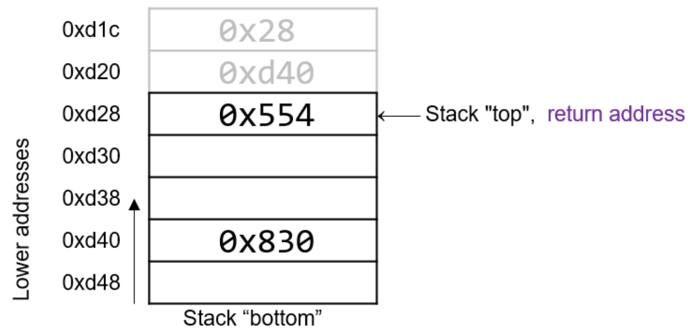
```
$ ./prog
```

The `add` instruction at address 0x53d adds 2 to register `%eax`. Recall that when a 32-bit integer is being returned, x86-64 utilizes component register `%eax` instead of `%rax`. Together the last two instructions are equivalent to the following code in `adder`:

```c
int a;
return a + 2;
```

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x2A |
| %edi | 1 |
| %rsp | **0xd28** |
| %rbp | **0xd40** |
| %rip | **0x541** |




call stack

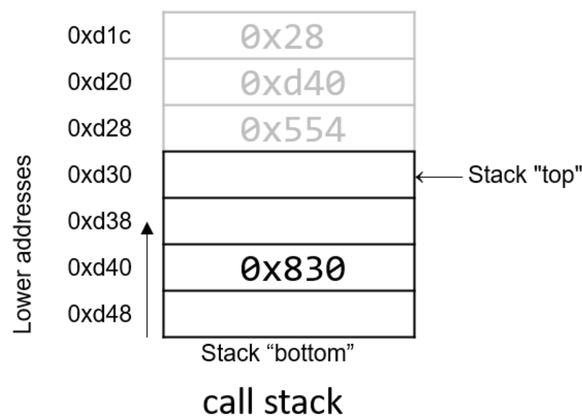
Terminal:

```
$ ./prog
```

After `pop` executes, the frame pointer again points to the beginning of the stack frame for `main`, or address 0xd40. The stack pointer now contains the address 0xd28.

```
0x536 <adder>:
0x536 push  %rbp
0x537 mov   %rsp, %rbp
0x53a mov   $-0x4(%rbp), %eax
0x53d add   $0x2, %eax
0x540 pop %rbp
0x541 retq
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
```

| Registers | |
|---|---|
| %eax | 0x2A |
| %edi | 1 |
| %rsp | **0xd30** |
| %rbp | 0xd40 |
| %rip | **0x554** |



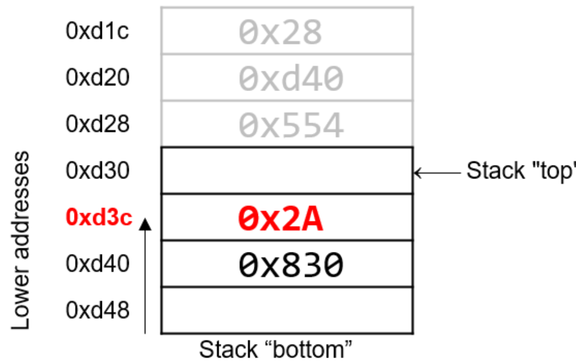

Terminal:

```
$ ./prog
```

The execution of `retq` pops the return address off the stack, restoring the instruction pointer back to 0x554, or the address of the next instruction to execute in `main`. The address contained in `%rsp` is now 0xd30.

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

| Registers | |
|---|---|
| %eax | 0x2A |
| %edi | 1 |
| %rsp | 0xd30 |
| %rbp | 0xd40 |
| %rip | **0x557** |




call stack

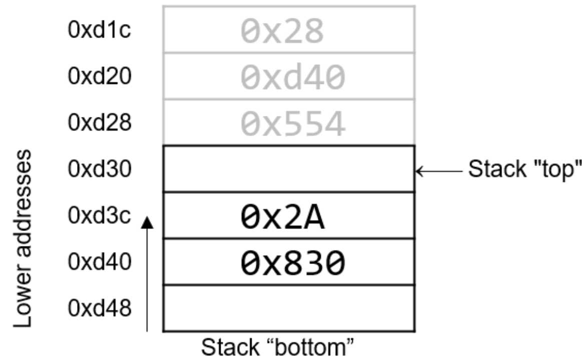
Terminal:

```
$ ./prog
```

Back in `main`, the `mov %eax, -0x4(%rbp)` instruction places the value in `%eax` at a location four bytes above `%rbp`, or at address 0xd3c. The next instruction replaces it back into register `%eax`.

---

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```




call stack

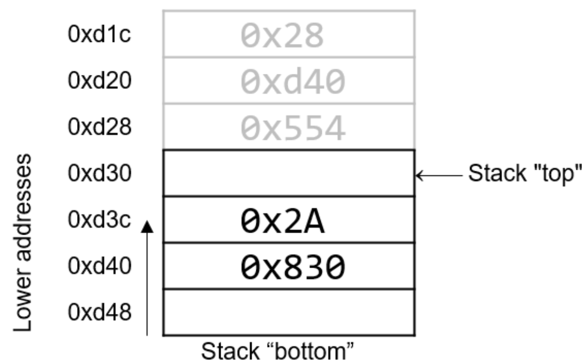
| Registers | | | |
|---|---|---|---|
| %eax | 0x2A | | |
| %edi | 1 | | |
| %rsp | 0xd30 | | |
| %rbp | 0xd40 | | |
| %rip | **0x55c** | %esi | **0x2A** |

Terminal:

```
$ ./prog
```

Skipping ahead a little, the `mov` instruction at address 0x55a copies the value in `%eax` (or 0x2A) to register `%esi`, which is the 32-bit component register associated with `%rsi` and typically stores the second parameter to a function.

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```




call stack

Terminal:

```
$ ./prog
```

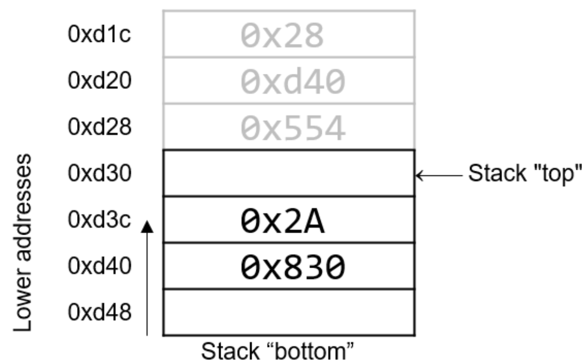
| Memory | |
|---|---|
| 0x400604 | “x is %d\n” |

| Registers | |
|---|---|
| %eax | 0x2A |
| %edi | **0x400604** |
| %rsp | 0xd30 |
| %rbp | 0xd40 |
| %rip | **0x561** |

| | |
|---|---|
| %esi | 0x2A |

The next instruction ( `mov $0x400604, %edi` ) copies a constant value (an address in code segment memory) to register `%edi` . Recall that register `%edi` is the 32-bit component register of `%rdi` , which typically stores the first parameter to a function. The code segment memory address 0x400604 is the base address of the string `"x is %d\n"` .

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```




call stack

| Registers | |
|---|---|
| %eax | **0x0** |
| %edi | 0x400604 |
| %rsp | 0xd30 |
| %rbp | 0xd40 |
| %rip | **0x566** |

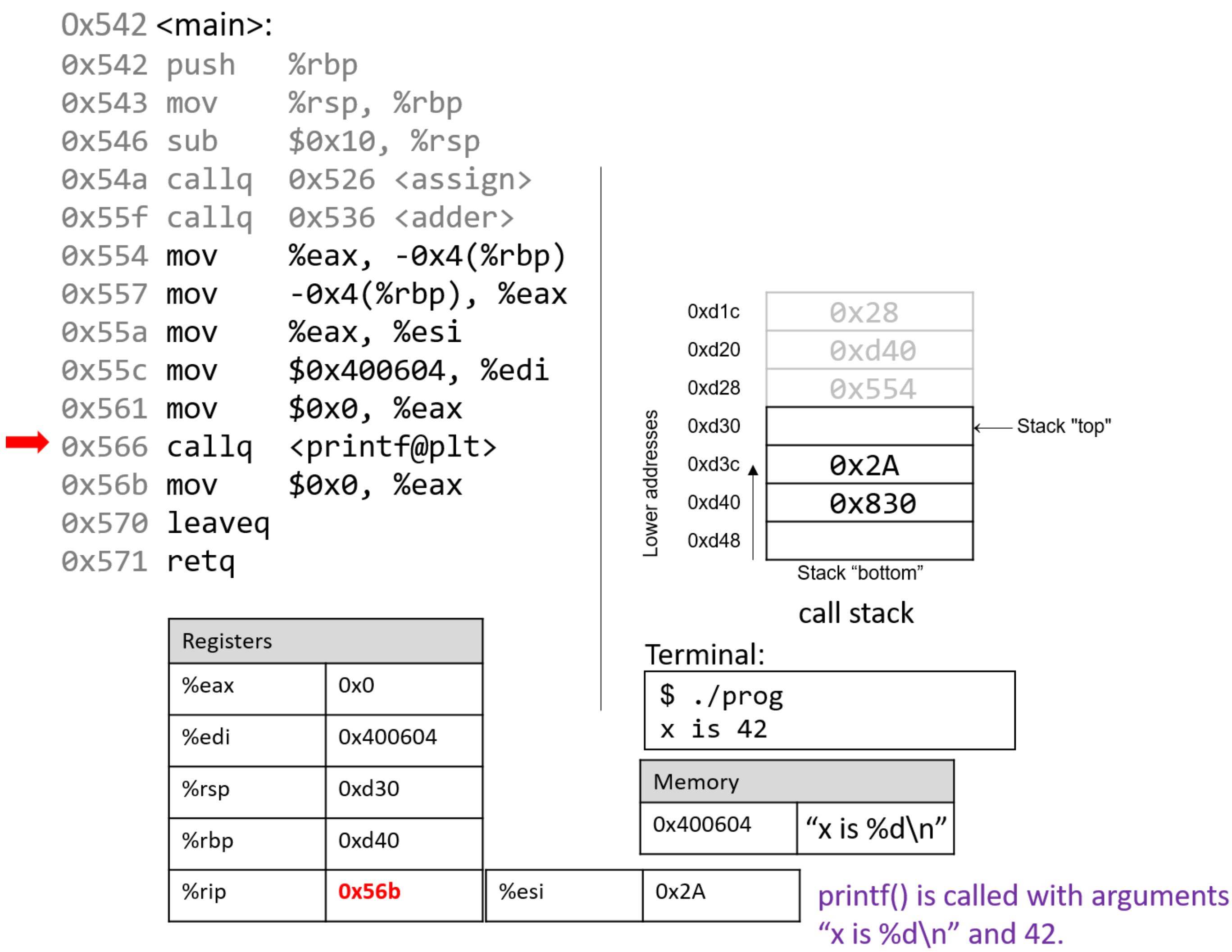
| | |
|---|---|
| %esi | 0x2A |

Terminal:

```
$ ./prog
```

| Memory | |
|---|---|
| 0x400604 | “x is %d\n” |

The next instruction resets register `%eax` with the value 0. The instruction pointer advances to the call to the `printf` function (which is denoted with the label `<printf@plt>`).




The next instruction calls the `printf` function. For the sake of brevity, we will not trace the `printf` function, which is part of `stdio.h`. However, we know from the manual page (`man -s3 printf`) that `printf` has the following format:

```
int printf(const char * format, ...)
```

In other words, the first argument is a pointer to a string specifying the format, and the second argument onward specify the values that are used in that format. The instructions specified by addresses 0x55a - 0x566 correspond to the following line in the `main` function:

```c
printf("x is %d\n", x);
```

When the `printf` function is called:

- A return address specifying the instruction that executes after the call to `printf` is pushed onto the stack.

- The value of `%rbp` is pushed onto the stack, and `%rbp` is updated to point to the top of the stack, indicating the beginning of the stack frame for `printf`.

At some point, `printf` references its arguments, which are the string `"x is %d\n"` and the value 0x2A. The first parameter is stored in component register `%edi`, and the second parameter is stored in component register `%esi`. The return address is located directly below `%rbp` at location `%rbp+8`.

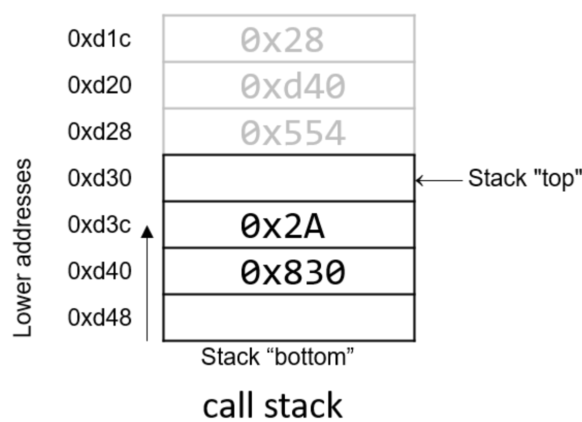
For any function with *n* arguments, GCC places the first six arguments in registers, as shown in Table 2, and the remaining arguments onto the stack *below* the return address.

After the call to `printf`, the value 0x2A is output to the user in integer format. Thus, the value 42 is printed to the screen!

---

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

| Registers | |
|---|---|
| %eax | **0x0** |
| %edi | 0x400604 |
| %rsp | 0xd30 |
| %rbp | 0xd40 |
| %rip | **0x570** |




Terminal:

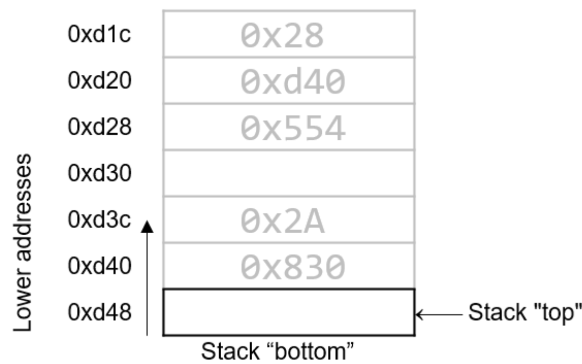
```
$ ./prog
x is 42
```

After the call to `printf`, the last few instructions clean up the stack and prepare a clean exit from the `main` function. First, the `mov` instruction at address 0x56b ensures that 0 is in the return register (since the last thing `main` does is return 0).

---

```
0x542 <main>:
0x542 push   %rbp
0x543 mov    %rsp, %rbp
0x546 sub    $0x10, %rsp
0x54a callq  0x526 <assign>
0x55f callq  0x536 <adder>
0x554 mov    %eax, -0x4(%rbp)
0x557 mov    -0x4(%rbp), %eax
0x55a mov    %eax, %esi
0x55c mov    $0x400604, %edi
0x561 mov    $0x0, %eax
0x566 callq  <printf@plt>
0x56b mov    $0x0, %eax
0x570 leaveq
0x571 retq
```

| Registers | |
|---|---|
| %eax | 0x0 |
| %edi | 0x400604 |
| %rsp | **0xd48** |
| %rbp | **0x830** |
| %rip | **0x571** |




call stack

Terminal:

```
$ ./prog
x is 42
```

Equivalent to:
mov %rbp, %rsp
pop %rbp

The `leaveq` instruction prepares the stack for returning from the function call. Recall that `leaveq` is analogous to the following pair of instructions:

```
mov %rbp, %rsp
pop %rbp
```

In other words, the CPU overwrites the stack pointer with the frame pointer. In our example, the stack pointer is initially updated from 0xd30 to 0xd40. Next, the CPU executes `pop %rbp`, which takes the value located at 0xd40 (in our example, the address 0x830) and places it in `%rbp`. After `leaveq` executes, the stack and frame pointers revert to their original values prior to the execution of `main`.

The last instruction that executes is `retq`. With 0x0 in the return register `%eax`, the program returns zero, indicating correct termination.

If you have carefully read through this section, you should understand why our program prints out the value 42. In essence, the program inadvertently uses old values on the stack to cause it to behave in a way that we didn’t expect. This example was pretty harmless; however, we discuss in future sections how hackers have misused function calls to make programs misbehave in truly malicious ways.

Contents