INSIDE THE MAIN FUNCTION



As you learned in Chapter 10, a C program begins by executing a function named main, which is called from a startup function in

the C hosted environment. The main function will call other functions (*subfunctions*) to do most of the processing. Even a simple "Hello, World!" program needs to call another function to write the message on the screen.

In this chapter, we'll focus on the main function, but the concepts apply to all the functions we'll be writing. We'll begin with a detailed look at the call stack, which is used for saving values and for local variables. Then we'll look at how to process data in a function and how to pass arguments to other functions. I'll wrap up the chapter by showing you how to use this knowledge to write the main function in assembly language.

Using the Call Stack

The *call stack*, commonly referred to simply as the *stack*, is a very useful place for creating local variables and saving items within a function. Before we cover how to use the stack for these purposes, you need to understand what stacks are and how they work.

Stacks in General

A stack is a linear data structure created in memory to store data items. Insertion of a data item onto (or deletion from) a stack can be done at only one end, called the *top*. Programs keep track of the top of the stack with a *stack pointer*.

Informally, you can think of a stack as being organized like a stack of dinner plates on a shelf. You need to be able to access only the item at the top of the stack. (And, yes, if you pull out a plate from somewhere within the stack, you will probably break something.) There are two fundamental operations on a stack:

push data_item Places the data_item at the top of the stack and moves the stack pointer to point to this latest item

pop *location* Moves the data item at the top of the stack to *location* and moves the stack pointer to point to the item now at the top of the stack

The stack is a *last in, first out (LIFO)* data structure. The last thing to be pushed onto the stack is the first thing to be popped off.

To illustrate the stack concept, let's continue with the dinner plate example. Say we have three differently colored dinner plates: a red one on the dining table, a green one on the kitchen counter, and a blue one on the bedside table. We'll stack them on the shelf in the following way:

- 1. Push red plate.
- 2. Push green plate.
- 3. Push blue plate.

At this point, our stack of plates looks like Figure 11-1.

Blue plate		
Green plate		
Red plate		

Figure 11-1: Three dinner plates in a stack

Now we perform the next operation:

4. Pop kitchen counter.

This moves the blue plate to the kitchen counter (recall that the blue plate was previously on the bedside table) and leaves the stack of dinner plates as shown in Figure 11-2.

Green plate
Red plate

Figure 11-2: One dinner plate has been popped from the stack.

If you have guessed that it's easy to really mess up a stack, you're right. A stack must be used according to a strict discipline. Within any function:

- Always push an item onto the stack before popping anything off.
- Never pop more things off than you have pushed on.
- Always pop everything off the stack that you have pushed on.

If you have no use for the item(s) that you have pushed onto the stack, you may simply set the stack pointer to where it was when the function was first entered. This is equivalent to discarding the items that are popped off. (Our dinner plate analogy breaks down here.)

A good way to maintain this discipline is to think of the use of parentheses in an algebraic expression. A push is analogous to a left parenthesis and a pop to a right parenthesis. The pairs of parentheses can be nested, but they have to match. An attempt to push too many items onto a stack is called *stack overflow*. An attempt to pop items off the stack beyond the bottom is called *stack underflow*.

A stack is implemented by dedicating a contiguous area of main memory to it. Stacks can grow in either direction in memory, into higher addresses or lower. An *ascending stack* grows into higher addresses, and a *descending stack* grows into lower addresses. The stack pointer can point to the top item on the stack, a *full stack*, or to the memory location where the next item will be pushed onto the stack, an *empty stack*. These four possible stack implementations are shown in Figure 11-3, with the integers 1, 2, and 3 pushed onto the stack in that order. Notice that memory addresses are *increasing downward* in this figure, which is the way we usually view them in the gdb debugger.

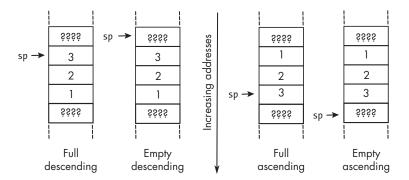


Figure 11-3: Four ways to implement a stack

The call stack in our environment is a *full descending stack*. To understand this choice, think about how you might organize things in memory. Recall that the control unit automatically increments the program counter as your program is executed. Programs come in vastly different sizes, so storing the program instructions at low memory addresses allows maximum flexibility with respect to program size.

The stack is a dynamic structure. You don't know how much stack space will be required by any given program as it executes, so it's impossible to know how much space to allocate. To allocate as much space as possible, while preventing it from colliding with program instructions, start the stack at the highest memory address and have it grow toward lower addresses.

This is a highly simplified rationalization for implementing stacks that grow "downward" in memory. The organization of various program elements in memory is much more complex than the description given here, but this may help you understand that there are some good reasons for what may seem to be a rather odd implementation.

The A64 architecture does not have push and pop instructions. It has instructions that allow you to effectively push items onto or pop items off of the stack, but most of the operations on the stack are done by allocating memory on the call stack and then directly storing items into or loading items from this allocated memory. Next, we'll look at how functions use the call stack.

The Stack Frame

Each function that calls another function needs to allocate memory on the stack for that function to use to save items and store local variables. This allocated memory is called a *stack frame* or *activation record*. To see how this works, we'll start with a program that has one local variable and calls two functions in the C standard library: printf and scanf. The program is shown in Listing 11-1.

inc_int.c // Increment an integer.

```
#include <stdio.h>
int main(void)
{
    int x;
    printf("Enter an integer: ");
    scanf("%i", &x);
    x++;
    printf("Result: %i\n", x);
```

return O;

```
}
```

Listing 11-1: A program to increment an integer

You can see how a stack frame is created by looking at the assembly language generated by the compiler, shown in Listing 11-2. I'll be referring to the numbered lines in this listing in the next several sections of this chapter, through page 222.

```
inc_int.s
                 .arch armv8-a
                        "inc int.c"
                 .file
                 .text
             1.section
                                .rodata
                 .align 3
        .LCO:
                 .string "Enter an integer: "
                 .align 3
        .LC1:
             ❷ .string "%i"
                 .align 3
        .LC2:
                .string "Result: %i\n"
                .text
                .align 2
                .global main
                 .type
                        main, %function
        main:
             ❸ stp
                        x29, x30, [sp, -32]! /// Create stack frame
                mov
                        x29, sp
                                               /// Set our frame pointer
                                              /// Page address
                adrp
                        x0, .LC0
                add
                        x0, x0, :lo12:.LCO
                                              /// Offset in page
                bl
                        printf
             add
                        x0, sp, 28
                                              /// Address of x
                mov
                        x1, x0
             6 adrp
                        x0, .LC1
                add
                        x0, x0, :lo12:.LC1
                bl
                         isoc99 scanf
             6 ldr
                        wO, [sp, 28]
                                               /// Load int
                add
                        w0, w0, 1
             🛛 str
                        wO, [sp, 28]
                                               /// x++;
                1dr
                        wO, [sp, 28]
                        w1, w0
                mov
                adrp
                        x0, .LC2
                add
                        x0, x0, :lo12:.LC2
```

```
bl printf
mov w0, 0

    ldp x29, x30, [sp], 32
ret
.size main, .-main
.ident "GCC: (Debian 10.2.1-6) 10.2.1 20210110"
.section .note.GNU-stack,"",@progbits
```

Listing 11-2: The compiler-generated assembly language for the program in Listing 11-1

The instructions used to create the stack frame form the *function prologue*. The first instruction in a function prologue is usually an stp instruction:

stp-Store register pair

stp ws1, ws2, $[xb{, offset}]$ stores the value in ws1 at the address in xb and the value in ws2 at xb + 4. If offset exists, it must be a multiple of 4 and is added to the address before storing the register values; xb is not changed.

stp xs1, xs2, [xb{, offset}] stores the value in xs1 at the address in xb and the value in xs2 at xb + 8. If offset exists, it must be a multiple of 8 and is added to the address before storing the register values; xb is not changed.

stp-Store register pair, pre-index

stp ws1, ws2, [xb, offset]! adds offset, which must be a multiple of 4, to xb. It then stores the value in ws1 at the new address in xb and the value in ws2 at xb + 4.

stp xs1, xs2, [xb, offset]! adds offset, which must be a multiple of 8, to xb. It then stores the value in xs1 at the new address in xb and the value in xs2 at xb + 8.

stp-Store register pair, post-index

stp ws_1 , ws_2 , [xb], offset stores the value in ws_1 at the address in xb and the value in ws_2 at xb + 4. It then adds offset, which must be a multiple of 4, to xb.

stp xs1, xs2, [xb], offset stores the value in xs1 at the address in xb and the value in xs2 at xb + 8. It then adds offset, which must be a multiple of 8, to xb.

NOTE

The operand order for almost all other A64 instructions is destination(s), source(s), but for the store instructions, it's the opposite.

Almost all the functions we'll write will begin with an stp instruction that looks like this:

```
stp x29, x30, [sp, -32]!
```

The compiler did this at the beginning of the function in Listing 11-2, creating the stack frame **③**.

The Procedure Call Standard for the Arm 64-Bit Architecture (AArch64) documentation (available in PDF and HTML formats at https://github.com/ARM -software/abi-aa/releases) specifies that the frame pointer (stored in register x29, also named fp) should point to the top of the stack frame, which is where the calling function's frame pointer is stored. The instruction mov x29, sp will set the called function's frame pointer, as shown in Listing 11-2.

The way the stp instruction has specified the stack memory address here, [sp, -32]!, probably doesn't make a lot of sense to you. Let's look at how instructions access memory in the A64 architecture.

A64 Memory Addressing

There are two ways that an instruction might refer to a memory address: the address could be encoded as part of the instruction, usually called an *absolute address*, or it could use *relative addressing*, where the instruction specifies an *offset* from a *base address*. In the latter case, the size of the offset and the location of the base address are encoded in the instruction.

All instructions in the A64 architecture are 32 bits long, but addresses are 64 bits long. We'll look at the details of the machine code in Chapter 12, but it's clear that a 64-bit address will not fit within a 32-bit instruction. To refer to a 64-bit address, instructions use one of the relative addressing modes listed in Table 11-1 to compute the address when they are executed.

Mode	Syntax	Note
Literal	label	pc-relative
Base register	[base]	Register only
Base plus offset	[base, offset]	Register-relative
Pre-indexed	[base, offset]!	Add offset to register before
Post-indexed	[base], offset	Add offset to register after

Table 11-1: A64 Addressing Modes

Each of the addressing modes in Table 11-1 starts with a 64-bit address in a *base register*. The literal mode uses *pc-relative addressing*, where the program counter serves as the base register. If *label* is in the same section as the instruction that references it, the assembler computes the address offset from the referencing instruction to the labeled instruction and fills in this offset as part of the referencing instruction. If the label is in another section, the linker will compute the offset and fill that in where the label is referenced. The number of bits allowed in the instruction limits the size of the address offset.

One of the advantages of pc-relative addressing is that it gives us *position-independent code (PIC)*, which means the function will execute correctly no matter where it is loaded into memory. The default for the gcc compiler in our environment is to produce PIC, with the linking phase producing a *position-independent executable (PIE)*. This means the linker doesn't specify a load address for the program, so the operating system can load the program

wherever it chooses. Not including the load address with the executable file improves security.

In the other four modes, the base register is a general-purpose register, x0-x30, or sp. For the base-plus-offset mode, the offset can be an immediate value or in a register. The offset is sign-extended to 64 bits and added to the value in the base register to compute the address. If the offset is in a register, it can be scaled so that it is a multiple of the number of bytes being loaded or stored. You'll see how this works when you learn to process integer arrays in Chapter 17.

In the pre-indexed mode, the computed address is stored in the base register *before* loading or storing the value. In the post-indexed mode, the computed address is stored in the base register *after* loading or storing the value.

For the pre-indexed mode, the offset can only be an immediate value. The post-indexed mode allows an immediate value for the offset or, for some advanced programming techniques, an offset value in a register.

The call stack in our environment is full-descending (see Figure 11-3), so the stp instruction uses the pre-indexed addressing mode. In the function in Listing 11-2, the address is specified as [sp, -32]! **③**. This subtracts 32 from the stack pointer *before* storing the caller's frame pointer and return address on the stack. This effectively allocates 16 bytes on the stack for this function's use, then pushes the return address and the caller's frame pointer onto the call stack. The number of bytes allocated for the stack frame must always be a multiple of 16 because the stack pointer, sp, must always be aligned on a 16-byte address boundary.

After the function has completed its processing, we need a *function epilogue* to restore the caller's frame pointer and link register and to delete the stack frame. In the function in Listing 11-2, this is done with the following instruction:

ldp x29, x30, [sp], 32

This instruction loads the two values at the top of the stack into the frame pointer and link register, then adds 32 to the stack pointer ③. This effectively pops the two values off the top of the stack into the x29 and x30 registers and then deletes this function's stack frame. Let's look at some variants of the 1dp instruction, which allows us to load two values at a time from memory:

1dp-Load register pair

ldp wd1, wd2, [xb{, offset}] loads the value at the address in xb into wd1 and the value at xb + 4 into wd2. If offset exists, it must be a multiple of 4 and is added to the address before loading the values; xb is not changed.

ldp xd1, xd2, $[xb{, offset}]$ loads the value at the address in xb into xd1 and the value at xb + 8 into xd2. If offset exists, it must be a multiple of 8 and is added to the address before loading the values; xb is not changed.

1dp-Load register pair, pre-index

ldp wd1, wd2, [xb, offset]! adds offset, which must be a multiple of 4, to xb. It then loads the value at the new address in xb into wd1 and the value at xb + 4 into wd2.

ldp xd1, xd2, [xb, offset]! adds offset, which must be a multiple of 8, to xb. It then loads the value at the new address in xb into xd1 and the value at xb + 8 into xd2.

ldp-Load register pair, post-index

ldp wd1, wd2, [xb], offset loads the value at the address in xb into wd1 and the value at xb + 4 into wd2. It then adds offset, which must be a multiple of 4, to xb.

ldp xd1, xd2, [xb], offset loads the value at the address in xb into xd1 and the value at xb + 8 into xd2. It then adds offset, which must be a multiple of 8, to xb.

Next, we'll see how this function uses the other 16 bytes of stack memory.

Local Variables on the Call Stack

Local variables in C can be directly accessed by their names only in the function where they're defined. We can allow another function to access a local variable in our function, including changing its value, by passing the address of that variable to the other function. This is what enables scanf to store a value for x, as you'll see on page 221.

You learned in Chapter 9 that CPU registers can be used as variables. But if we were to use CPU registers to hold all of our variables, we'd soon run out of registers, even in a small program. So, we need to allocate space in memory for variables.

As we'll see later in this chapter, a function needs to preserve the contents of some registers for the calling function. If we want to use such a register in our function, a local variable would be a good place to store a copy of its content so we can restore it before returning to the calling function.

The stack frame meets the requirements of local variables. It's created when the function first starts, and it's deleted once the function completes. The memory in a stack frame is easily accessed using the base-plus-offset addressing mode (see Table 11-1), with sp as the base addressing register. An example in Listing 11-2 is where we load the integer:

ldr w0, [sp, 28]

This instruction loads the 32-bit word located 28 bytes from the address in sp into w0 **(b)**. The function treats its stack frame as a record rather than a stack with this code. You'll learn about records in Chapter 17.

Figure 11-4 gives a pictorial view of the completed stack frame for the main function in Listings 11-1 and 11-2.

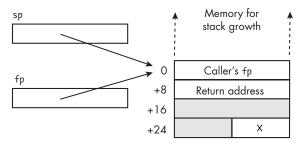


Figure 11-4: The stack frame for the function in Listings 11-1 and 11-2

The two addresses on the stack each take 8 bytes, and the int variable, x, takes 4 bytes. The memory in the gray area is unused but necessary for keeping the stack pointer, sp, aligned on a 16-byte address boundary.

Now that you know how to use a stack frame, let's look at how this function processes data.

Processing Data in a Function

A64 is a *load-store architecture*, which means the instructions that operate on data cannot access memory. There is a separate group of instructions for moving data to and from memory.

This is in contrast to a *register–memory architecture*, which includes instructions that can operate on data in memory. The data operations are still performed by the arithmetic/logic unit in the CPU (see Figure 9-1), but they use registers that are hidden from the programmer. The Intel x86 is an example of a register–memory architecture.

The processing in the main function in Listing 11-2 is very simple: the program adds 1 to an integer. But before it can perform this operation, it needs to load the value into a register using the 1dr instruction O. Since this program changes the value in the variable, the new value must be stored back into memory with the str instruction O.

Let's look at some common instructions used for loading values from memory:

ldr-Load register, pc-relative

ldr wd, addr loads wd with the 32-bit value at memory location addr, which must be ± 1 MB from this instruction. Bits 63 to 32 of xd are set to 0.

ldr xd, addr loads xd with the 64-bit value at memory location addr, which must be ± 1 MB from this instruction.

ldr-Load register, base register-relative

ldr wd, [xb{, offset}] loads wd with the 32-bit value at the memory location obtained by adding the address in xb and the optional offset, which is a multiple of 4 in the range 0 to 16,380. Bits 63 to 32 of xd are set to 0.

ldr xd, [xb{, offset}] loads xd with the 64-bit value at the memory location obtained by adding the address in xb and the optional offset, which is a multiple of 8 in the range 0 to 32,760.

ldrsw-Load register, signed word, base register-relative

ldrsw wd, [xb{, offset}] loads wd with the 32-bit value at the memory location obtained by adding the address in xb and the optional offset, which is a multiple of 4 in the range 0 to 16,380. Bits 63 to 32 of xd are set to a copy of bit 31 of the loaded word.

ldrb-Load register, unsigned byte, base register-relative

ldrb wd, $[xb\{, offset\}]$ loads the low-order byte of wd with the 8-bit value at the memory location obtained by adding the address in xb and the optional offset, which is in the range 0 to 4,095. Bits 31 to 8 of xd are set to 0; bits 63 to 32 are unchanged.

ldrsb-Load register, signed byte, base register-relative

ldrsb wd, [xb{, offset}] loads the low-order byte of wd with the 8-bit value at the memory location obtained by adding the address in xb and the optional offset, which is in the range 0 to 4,095. Bits 31 to 8 of xd are set to a copy of bit 7 of the loaded byte; bits 63 to 32 are unchanged.

Here are some similar instructions for storing values in memory:

str-Store register, pc-relative

str ws, addr stores the 32-bit value in ws at memory location addr, which must be ± 1 MB from this instruction.

str xs, addr stores the 64-bit value in xs at memory location addr, which must be ± 1 MB from this instruction.

str-Store register, base register-relative

str ws, [xb{, offset}] stores the 32-bit value in ws at the memory location obtained by adding the address in xb and the optional offset, which is a multiple of 4 in the range 0 to 16,380.

str xs, [xb{, offset}] stores the 64-bit value in xs at the memory location obtained by adding the address in xb and the optional offset, which is a multiple of 8 in the range 0 to 32,670.

strb-Store register, byte, base register-relative

strb ws, [xb{, offset}] stores the low-order 8 bits in ws at the memory location obtained by adding the address in xb and the optional offset, which is in the range 0 to 4,095.

The program simply adds 1 to the variable, which can be done with the add instruction. I'll include the sub instruction here because it's very similar,

but I'll give only some basic syntax (both instructions have several options, which are described in the manuals):

add-Add extended register

add wd, ws1, ws2{, xtnd amnt} adds the values in ws1 and ws2 and stores the result in wd. The value added from ws2 can be a byte, halfword, word, or doubleword. It can be sign- or zero-extended and then left-shifted 0 to 4 bits before the addition, using the xtnd amnt option.

add xd, xs1, xs2{, xtnd amnt} adds the values in xs1 and xs2 and stores the result in xd. The value added from xs2 can be a byte, halfword, word, or doubleword. It can be sign- or zero-extended and then left-shifted 0 to 4 bits before the addition, using the xtnd amnt option.

add-Add immediate

add wd, ws, *imm*{, *shft*} adds *imm* to the value in ws and stores the result in wd. The *imm* operand is an unsigned integer in the range 0 to 4,095, which can be left-shifted 0 or 12 bits before the addition, using the *shft* option.

add xd, xs, imm{, shft} adds imm to the value in xs and stores the result in xd. The imm operand is an unsigned integer in the range 0 to 4,095, which can be left-shifted 0 or 12 bits before the addition, using the shft option.

sub-Subtract extended register

sub wd, ws1, ws2{, xtnd amnt} subtracts the value in ws2 from ws1 and stores the result in wd. The value subtracted from ws2 can be a byte, halfword, word, or doubleword. It can be sign- or zero-extended and then left-shifted 0 to 4 bits before the subtraction, using the xtnd amnt option.

sub xd, xs1, xs2{, xtnd annt} subtracts the value in xs2 from xs1 and stores the result in xd. The value subtracted from xs2 can be a byte, halfword, word, or doubleword. It can be sign- or zero-extended and then left-shifted 0 to 4 bits before the subtraction, using the xtnd annt option.

sub-Subtract immediate

sub wd, ws, *imm*{, *shft*} subtracts *imm* from the value in ws and stores the result in wd. The *imm* operand is an unsigned integer in the range 0 to 4,095, which can be left-shifted 0 or 12 bits before the subtraction, using the *shft* option.

sub xd, xs, *imm*{, *shft*} subtracts *imm* from the value in xs and stores the result in xd. The *imm* operand is an unsigned integer in the range 0 to 4,095, which can be left-shifted 0 or 12 bits before the subtraction, using the *shft* option.

Table 11-2 lists the allowable values for the *xtnd* option in the add and sub instructions.

 Table 11-2:
 Allowable Values for xtnd in add

 and sub Instructions
 Instructions

xtnd	Effect	
uxtb	Unsigned extension of byte	
uxth	Unsigned extension of halfword	
uxtw	Unsigned extension of word	
uxtx	Unsigned extension of doubleword	
sxtb	Signed extension of byte	
sxth	Signed extension of halfword	
sxtw	Signed extension of word	
sxtx	Signed extension of doubleword	

The extension begins with the indicated low-order portion of the source register and adds bits to the left to match the width of the other registers in the instruction. For unsigned extension, the added bits are all 0. For signed extension, the added bits are copies of the highest-order bit of the starting value. When using w registers, uxtw can be replaced with lsl; with x registers, uxtx can be replaced with lsl.

It might seem meaningless to extend a doubleword, which is already 64 bits wide, to match the size of an x register, but the instruction syntax requires that we use the entire *xtnd amnt* option if we wish to shift the value.

As an example of how these size extensions work, let's start with the following values in x2 and x3:

	x2: 0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa				
x3: 0x8					
The in	nstruction sequence				
add	w0, w3, w2, uxtb				
add	w1, w3, w2, sxtb				
gives:					
	x0: 0xfedcce99				
x3: 0x	x3: 0xfedccd99				

We'll see other instructions that use these width extensions as we continue through the book.

Now that you know how to do the arithmetic, let's look at how to call the other functions.

Passing Arguments in Registers

There are several ways for a function to pass arguments to another function. I'll start by describing how to use registers for passing arguments. I'll discuss other ways when I cover subfunctions in more detail in Chapter 14.

Recall from Chapter 2 that when a function calls another function, it can pass arguments that the called function can use as parameters. In principle, the C compiler—or you, when you're writing in assembly language—could use any of the 31 general-purpose registers, except the link register, x30, to pass arguments from one function to another. Just store the arguments in the registers and call the desired function. Of course, the calling and called functions need to agree on exactly which register each argument is in.

The best way to avoid making mistakes is to follow a standard set of rules. This is especially important if more than one person is writing code for a program. Other people have realized the importance of having such standards and have developed an *application binary interface (ABI)* that includes a set of standards for passing arguments in the A64 architecture. The compiler we're using, gcc, follows the rules in the *Procedure Call Standard for the Arm 64-Bit Architecture* (referenced on page 213), and we'll do the same for the assembly language we write.

Table 11-3 summarizes the standards for how the called function uses the registers.

Register	Usage	Save?
x0–x7	Parameter; result	No
x8	Address of result	No
x9–x18	Scratch	No
x19–x28	Variables	Yes
x29	Frame pointer	Yes
x30	Link register	Yes
sp	Stack pointer	Yes
xzr	Zero register	N/A

Table 11-3: General-Purpose Register Usage

We would use *wn* instead of *xn* for 32-bit register names. We're using 64-bit addressing in this book. Because x29 and x30 will always contain addresses, we'll never use w29 or w30.

The "Save?" column shows whether a called function needs to preserve the value in that register for the calling function. If we need to use a register that must be preserved, we'll create a local variable in our stack frame for that purpose.

The calling function passes the arguments in the registers in the order in which they're listed, from left to right in a C function, starting with x0 (or w0 for a 32-bit value). This allows for the passing of up to eight arguments, x0-x7. You'll see how to use the call stack to pass more than eight arguments in Chapter 14.

For an example of how to pass arguments, let's look at the call to scanf in Listing 11-1:

```
scanf("%i", &x);
```

Let's start with the second argument, the address of x. In Figure 11-4, x is located at an offset of 28 bytes from the stack pointer, sp. Looking at the assembly language generated by the compiler in Listing 11-2, you can see that computing the address can be done by adding 28 to sp **9**. Since it's the second argument, it needs to be moved to x1.

The first argument—the text string "%i", which is created with a .string assembler directive **2**—is more complex. The general format for the .string directive is:

```
.string "text"
```

This creates a C-style text string as a char array with one byte for each character code point in *text*, plus one byte for the terminating NUL character.

The compiler places the three text strings in this program in the .rodata section **①** of the object file. The loader/linker typically loads .rodata sections into the text segment following the executable code. Notice that each text string is aligned to an 8-byte (64 bits) address boundary with a .align 3 directive. This might make the code execute a little faster.

When you pass an array to a function in C, only the address of the first element in the array gets passed. So, the address of the first character of "%i" is passed to scanf. The A64 architecture provides two instructions for getting an address into a register:

adr-Address

adr xd, addr loads the memory address addr into xd; addr must be within ± 1 MB of this instruction.

adrp-Address page

adrp xd, addr loads the page address of addr into bits 63 to 12 of xd, with bits 11 to 0 set to 0. The page address is the next-lower 4KB address boundary of addr, and addr must be within \pm 4GB of this instruction.

Both instructions use the literal addressing mode (see Table 11-1) to refer to a memory address. They each allow a 21-bit offset value, hence the ± 1 MB range for adr. With 0s in the low-order 12 bits, the adrp instruction gives a 33-bit offset from the pc, for an addressing range of ± 4 GB from the pc, but with 4KB granularity.

The adrp instruction effectively treats memory as being divided into 4KB *pages*. (These pages are conceptually distinct from the memory pages that the operating system uses to manage main memory.) It loads the beginning address of a 4KB page, the *page address*, into the destination register. Compared to the adr instruction, this increases the range of addresses we can

load into a register from ± 1 MB to ± 4 GB, but we still need to add the offset within the 4KB page to the page address in the register.

Thus, we can load a 64-bit address located within ± 4 GB with a twoinstruction sequence. The compiler did this in Listing 11-2 using the following code Θ :

adrp x0, .LC1 add x0, x0, :lo12:.LC1

Since the label .LC1 is in the .rodata section, the linker computes the offset from the instruction to the label. The adrp instruction loads the page number of that offset into x0. The :lo12: modifier tells the assembler to use only the low-order 12 bits of the offset as the immediate value for the add instruction. This two-step process may seem a bit puzzling to you. It's due to the limited number of bits available for immediate values in an instruction; you'll see the details when we cover how instructions are coded in binary in Chapter 12.

After loading the arguments into registers, we transfer the program flow to the other function with a bl or a blr instruction:

b1-Branch and link

bl *addr* adds 4 to the address in the pc and loads the sum into x30. It then loads the memory address of *addr* into the pc, thus branching to *addr*, which must be within ± 128 MB of this instruction.

blr-Branch and link, register

blr xs adds 4 to the address in the pc and loads the sum into x30. It then moves the 64-bit address in xs to the pc, thus branching to that address.

These instructions are used to call a function. Adding 4 to the address in the pc gives the address of the instruction immediately after the bl or blr in memory. We usually want the called function to return to this location. The x30 register is used as a link register by these two branching instructions.

In the next section, we'll write the program in assembly language. It'll be very similar to what the compiler generated, but we'll use names that make it easier to read.

Writing main in Assembly Language

Listing 11-3 shows my assembly language version of the inc_int program. It closely follows the assembly language generated from the C version by the compiler in Listing 11-2, but I've added comments and used more meaningful labels for the string constants. This should make it a little easier to understand how the program uses the stack and passes arguments to other functions.

```
🛛 .equ
                FRAME, 32
// Constant data
     ③.section .rodata
prompt:
        .string "Enter an integer: "
input format:
        .string "%i"
result:
        .string "Result: %i\n"
// Code
        .text
        .align 2
        .global main
               main, %function
        .type
main:
                fp, lr, [sp, FRAME]! // Create stack frame
     4 stp
                                     // Set our frame pointer
                fp, sp
        mov
        adr
                x0, prompt
                                     // Prompt user
        bl
                printf
        add
                x1, sp, x
                                     // Address for input
        adr
                x0, input format
                                     // scanf format string
     6 bl
                scanf
        ldr
               w0, [sp, x]
                                      // Get x
                                      // Add 1
        add
               w1, w0, 1
               w1, [sp, x]
                                      // x++
        str
                x0, result
                                      // printf format string
        adr
        bl
                printf
                                      // Result is in w1
               wO, wzr
        mov
     6 ldp
                fp, lr, [sp], FRAME
                                     // Delete stack frame
        ret
```

Listing 11-3: A program to increment an integer, in assembly language

We see another assembler directive, .equ, in Listing 11-3. The format is:

.equ symbol, expression

The *expression* must evaluate to an integer, and the assembler sets *symbol* equal to that value. You can then use the symbol in your code, making it much easier to read, and the assembler will plug in the value of the expression. The expression is often just an integer. For example, I have equated the symbol FRAME to the integer 32 **2**. This allows us to write code that is self-documenting **9**. I've also used the assembler names, fp and lr, for the register names x29 and x30, respectively.

Note that we don't need to specify the .text segment for the .rodata section 6. The assembler and linker produce a .rodata section, and it's up to the operating system to determine where to load it. I didn't align the text strings in the .rodata section either. Although alignment might make the code execute a little faster, it could waste a few bytes of memory. (Both factors are irrelevant for the programming we're doing in this book.) I've also used adr instead of adrp to load the addresses of the strings. The programs we'll be writing in this book are very simple, so I expect the strings in the .rodata section to be within ± 1 MB of the instructions that use them.

Finally, I've called scanf instead of __isoc99_scanf **6**. The __isoc99_ prefix disallows several nonstandard conversion specifiers; again, this is beyond the scope of this book.

Our variable, x, is in the stack frame **①**. The stack frame is created in the function prologue **③** and deleted in the function epilogue **③**, making x an automatic local variable.

YOUR TURN

- 11.1 You can tell the gcc compiler to optimize the code it generates for speed with the -Ofast option or for size with the -Os option. Generate the assembly language for the program in Listing 11-1 for each option. What are the differences?
- 11.2 Modify the program in Listing 11-3 so that it inputs two integers and then displays the sum and difference of the two.
- 11.3 Enter the following C code in a file named *sum_diff.c*:

// Add and subtract two integers.

```
void sum_diff(int x, int y, int *sum, int *diff)
{
    *sum = x + y;
    *diff = x - y;
}
```

Modify the program in Listing 11-3 so that it inputs two integers, calls sum_diff to compute the sum and difference of the two integers, and then displays the two results.

What You've Learned

Call stack An area of memory used for storing program data and addresses that grows and shrinks as needed.

Stack frame Memory on the call stack used for saving the return address and caller's frame pointer, as well as for creating local variables.

Function prologue The instructions that create a stack frame.

Function epilogue The instructions that restore the caller's link register and frame pointer and delete the stack frame.

Automatic local variables Variables created anew each time a function is called. They can easily be created on the call stack.

Passing arguments to a subfunction Up to eight arguments are passed in the x0-x7 registers.

Calling a function The branch and link instructions, b1 and b1r, transfer program flow to a function, storing the return address in x30.

A64 addressing There are several modes for generating a 64-bit address with a 32-bit instruction.

Position-independent executable The operating system can load the program anywhere in memory, and it will execute correctly.

Load-store architecture Instructions can operate only on data that is in registers.

In the next chapter, we'll take a brief look at how instructions are coded in machine language. This will help you understand the reasons for some of the limitations of instructions, such as the size of offset when referring to a memory address.