3

MEMORY ACCESS AND ORGANIZATION

Chapters 1 and 2 showed you how to declare and access simple variables in an assembly language program. This chapter fully explains ARM memory access. You'll learn how to efficiently organize your variable declarations to speed up access to their data. You'll also learn about the ARM stack and how to manipulate data on it.

This chapter discusses several important concepts, including the following:

- Memory organization
- Memory access and the memory management unit
- Position-independent executables and address space layout randomization
- Variable storage and data alignment
- Endianness (memory byte order)

—-1
—0
—+1

- ARM memory addressing modes and address expressions
- Stack operations, return addresses, and preserving register data

This chapter will teach to you make efficient use of your computer's memory resources.

3.1 Runtime Memory Organization

A running program uses memory in many ways, depending on the data's type. Here are some common data classifications you'll find in an assembly language program:

Code Memory values that encode machine instructions (also known as the *text* section under Linux and macOS).

Uninitialized static data An area in memory set aside by the program for uninitialized variables that exist the whole time the program runs; the OS will initialize this storage area to 0s when it loads the program into memory.

Initialized static data A section of memory that also exists the whole time the program runs. However, the OS loads values for all the variables appearing in this section from the program's executable file, so they have an initial value when the program first begins execution.

Read-only data Similar to initialized static data, insofar as the OS loads initial data for this section of memory from the executable file. However, this section is marked *read-only* to prevent inadvertent modification of the data. Programs typically store constants and other unchanging data in this section (the code section is also marked read-only by the OS).

Heap This special section of memory is designated to hold dynamically allocated storage. Functions such as C's `malloc()` and `free()` are responsible for allocating and deallocating storage in the heap area. "Pointer Variables and Dynamic Memory Allocation" on page XX discusses dynamic storage allocation in greater detail.

Stack In this special section in memory, the program maintains local variables for procedures and functions, program state information, and other transient data. See "The Push and Pop Operations" on page XX for more information about the stack section.

These are the typical sections you will find in common programs, assembly language or otherwise. Smaller programs won't use all these sections, though most programs have at least code, stack, and data sections. Complex programs may create additional sections in memory for their own purposes. Some programs may combine several of these sections. For example, many programs will combine the code and read-only sections into the same section in memory (as the data in both sections gets marked as read-only). Some programs combine the uninitialized and initialized data

sections, initializing the uninitialized variables to 0. Combining sections is generally handled by the linker program. See "For More Information" on page XX concerning the GNU linker.

Linux and macOS tend to put different types of data into different sections (or *segments*) of memory. Although it is possible to reconfigure memory to your choice by running the linker and specifying various parameters, one typical organization might be similar to that in Figure 3-1.

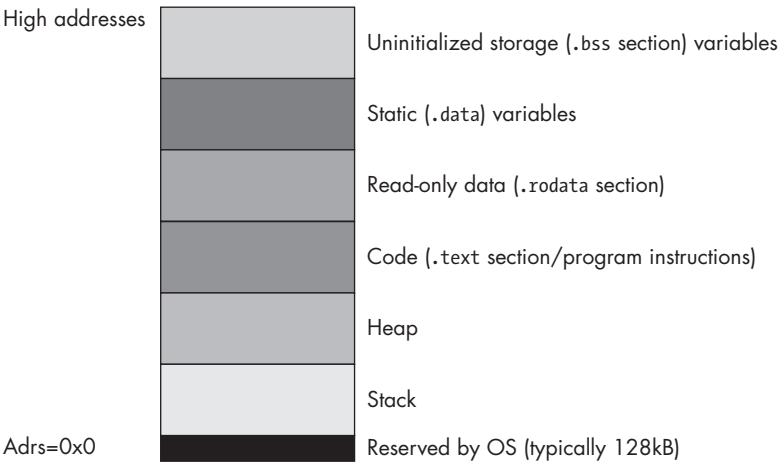

Figure 3-1: A Linux/macOS example runtime memory organization

This figure is just an example. Real programs will likely organize memory differently, especially when using address space layout randomization (ASLR), discussed later in this chapter.

The OS reserves the lowest memory addresses. Generally, your application cannot access data (or execute instructions) at these low addresses. One reason the OS reserves this space is to help trap NULL pointer references: if you attempt to access memory location 0x0 (NULL), the OS will generate a *segmentation fault* (also known as a *general protection fault*), meaning you've accessed a memory location that doesn't contain valid data.

The remaining six areas in the memory map hold different types of data associated with your program. These sections of memory include the stack section, the heap section, the `.text` (code) section, the `.data` section, the `.rodata` (read-only data) section, and the `.bss` (storage) section. Each of these memory sections corresponds to a type of data you can create in your Gas programs. I will describe the `.text`, `.data`, `.rodata`, and `.bss` sections in detail next. (The OS provides the stack and heap sections; you don't normally declare these two in an assembly language program, so there isn't anything more to discuss about them here.)

3.1.1 The .text Section

The `.text` section contains the machine instructions that appear in a Gas program. Gas translates each machine instruction you write into a

sequence of one or more word values. The CPU interprets these 32-bit word values as machine instructions during program execution.

By default, when GCC/Gas/*ld* links your program, it tells the system that your program can execute instructions and read data from the code segment, but cannot write data to the code segment. The OS will generate a segmentation fault if you attempt to store any data into the code segment.

3.1.2 The .data Section

You'll typically put your variables in the `.data` section. In addition to declaring static variables, you can embed lists of data into the `.data` declaration section. You use the same technique to embed data into your `.data` section that you use to embed data into the `.text` section: use the `.byte`, `.hword`, `.word`, `.dword`, and so on, directives. Consider the following example:

```
     .data
bb:  .byte    0
     .byte    1,2,3

u:   .word    1
     .dword   5,2,10

c:   .byte    0
     .byte    'a', 'b', 'c', 'd', 'e', 'f'

bn:  .byte    0
     .byte    true  // Assumes true is defined as 1.
```

Values that Gas places in the `.data` memory segment by using these directives are written to the segment after the preceding variables. For example, the byte values `1,2,3` are emitted to the `.data` section after `bb`'s `0` byte. Because there aren't any labels associated with these values, you do not have symbolic access to these values in your program. You can use the indexed addressing modes (described later in this chapter) to access these extra values.

3.1.3 Read-Only Data Sections

Gas does not provide a stand-alone directive for creating sections that hold read-only constants. However, you can easily use the Gas `.section` directive to create a generic read-only constant section as follows:

```
.section .rodata, ""
```

Most programs use the `.rodata` identifier, by convention, for read-only data. For example, GCC uses this name for read-only constant sections. You could use any identifier you choose here. For example, I often use the name `.const` for constant sections. However, as GCC uses `.rodata`, I'll stick to that convention in this book. I'll say more about the `.section` directive a little later; for the time being, note that as long as the second argument is the empty string, Gas will create a read-only data section by using this directive.

The `.section .rodata` section holds constants, tables, and other data that your program cannot change during execution. This section is similar to the `.data` section, with two differences:

- The `.rodata` section is defined with `.section .rodata, ""` rather than `.data`.
- The system does not allow you to write data to variables in an `.rodata` object while the program is running.

Here's an example:

```
        .section  .rodata, ""
pi:     .single   3.141592653589793 // (rounded)
e:      .single   2.718281828459045 // (rounded)
MaxU16: .hword    65535
MaxI16: .hword    32767
```

For many purposes, you can treat `.rodata` objects as literal constants. However, because they are actually memory objects, they behave like read-only `.data` objects. You cannot use an `.rodata` object anywhere a literal constant is allowed. For example, you cannot use them as *displacements* (constant *offsets* from a base pointer) in addressing modes (see "The ARM Memory Addressing Modes" on page XX), in constant expressions, or as immediate values. In practice, you can use them anywhere that reading a `.data` variable is legal.

LINUX VS. MACOS: FORCED CODE ALIGNMENT

ARM machine instructions must be aligned on a word (32-bit) boundary. The ARM cannot physically address an instruction that is not so aligned. Therefore, if you insert data into the `.text` section that is not a multiple of 4 bytes long, any instructions following that data will be misaligned. You must always include an `.align 2` (or `.balign 4`) directive before any code appearing after data that is not a multiple of 4 bytes long in the `.text` section.

The macOS assembler is so paranoid about this that it requires all symbols appearing in the `.text` section to be aligned on a 4-byte boundary, and it will generate an error if it encounters a label declaration (*label*:, where *label* represents any identifier) that is not associated with a word-aligned address. The only way to correct this error is to insert an `.align 2` (or `.balign 4`) directive before the label declaration. This can create a problem for certain data declarations in the `.text` section. Consider the following code:

```
      .align 2
bb:   .byte 0
c:    .byte 0
```

(continued)

The macOS assembler will require both of these symbols to be word-aligned (requiring an `.align 2` directive between them), even if you don't want this. You might, for example, want `c` to immediately follow `bb` in memory. The macOS assembler does not allow this. If you define a label, that label must be aligned on a word boundary.

One solution is to avoid putting data in the `.text` section; just put your read-only constants, such as `.rodata`, in their own section. However, there are good reasons for wanting to put data in the `.text` section. In those situations, you'll have to work around this limitation when writing code for macOS.

As with the `.data` section, you may embed data values in the `.rodata` section by using the `.byte`, `.hword`, `.word`, `.dword`, and so on, data declarations. For example:

```
        .section  .rodata, ""
roArray: .byte    0
        .byte     1, 2, 3, 4, 5
dwVal:   .dword   1
        .dword    0
```

You can also declare constant values in the `.text` section. Data values you declare in this section are also read-only objects, as Linux and macOS write-protect the `.text` section. If you do place constant declarations in a `.text` section, take care to place them in a location that the program will not attempt to execute as code (such as after a `b.al` or `ret` instruction). Unless you're using data declarations to manually encode ARM machine instructions (which would be rare and done only by expert programmers), you don't want your program to attempt to execute data as machine instructions; the result is usually undefined.

NOTE *Technically, the result of executing data in the `.text` section is well-defined: the machine will decode whatever bit pattern you place in memory as a machine instruction. However, few people will be able to look at a piece of data and interpret its meaning as a machine instruction.*

3.1.4 The .bss Section

The `.data` section requires that you initialize objects, even if you simply place a default value of `0` in the operand field. The `.bss` (block started by symbol) section lets you declare variables that are always uninitialized when the program begins running. This section begins with the `.bss` reserved word and contains variable declarations whose initializers must always be 0. Here is an example:

```
            .bss
UninitUns32: .word  0
i:          .word  0
character:  .byte  0
bb:         .byte  0
```

The OS will initialize all `.bss` objects to 0 when it loads your program into memory. However, it's probably not a good idea to depend on this implicit initialization. If you need an object initialized with 0, declare it in a `.data` section and explicitly set it to 0.

Annoyingly, Gas requires you to explicitly provide an initializer of 0 when declaring variables in the `.bss` section. Good assembly language programmers don't like doing this, because providing their source code with an explicit value tells the reader that they are expecting that variable to contain that value when the program runs. If the program explicitly isn't expecting the variable to be initialized, it would be nice to tell the reader that.

A very old convention to make this statement is to use the expression `.-.` in the operand field of such declarations. For example:

```
            .bss
UninitUns32: .word  .-.
i:          .word  .-.
character:  .byte  .-.
bb:         .byte  .-.
```

Gas substitutes the current value of the location counter (see "Gas Storage Allocation for Variables" on page XX) in place of the period (`.`). The expression *`location_counter`* minus *`location_counter`* is equal to 0, which satisfies the Gas requirements for initializers in the `.bss` section. This strange syntax lets the reader know that you're not explicitly expecting the variable to be initialized with 0 when the program runs.

If `.-.` is too bizarre for your tastes (or you don't want to have to type three characters), I've often used something like this to get the same results:

```
            .equ   _, 0  // "_" is a legitimate identifier
            .bss
UninitUns32: .word  _
i:          .word  _
character:  .byte  _
bb:         .byte  _
```

This book tends to use the `.-.` form (when not explicitly specifying `0`), as there is historical precedence for it. This form has one drawback, however: it does not work for `.qword` declarations (this is a Gas limitation).

Variables you declare in the `.bss` section may consume less disk space in the executable file for the program. This is because Gas writes out initial values for `.rodata` and `.data` objects to the executable file, but it may use a

compact representation for uninitialized variables you declare in the `.bss` section. Note, however, that this behavior is dependent on the OS version and object-module format.

3.1.5 The .section Directive

The `.section` directive allows you to create sections using any name you please (the `.rodata` section is an example). The syntax for this directive is

```
.section identifier, flags
```

where *identifier* is any legal Gas identifier (it does not have to begin with a period) and *flags* is a string surrounded by quotes. The contents of the string vary by OS, but both Linux and macOS seem to support the following characters:

- **b** Section is a `.bss` section and will hold uninitialized data. All data declarations must have a `0` initializer.
- **x** Section contains executable code.
- **w** Section contains writable data.
- **a** Section is allocatable (must be present for data sections).
- **d** Section is a data section.

The *flags* string may contain zero or more of these characters, though certain flags (such as `"b"` and `"x"` or `"d"`) are mutually exclusive. If the `"w"` flag is not present in the string, the section will be read-only. Here are some typical `.section` declarations:

```
.section aDataSection, "adw" // Typical data section
.section .const, ""          // Like .rodata
.section .code, "x"          // Code section (like .text)
```

Each unique section you define will be given its own block of memory (such as the blocks that appear in Figure 3-1). The GNU linker/loader will merge all sections with the same name when assigning them to blocks of memory.

3.1.6 Declaration Sections

The `.data`, `.rodata`, `.bss`, `.text`, and other named sections may appear zero or more times in your program. The declaration sections may appear in any order, as the following example demonstrates:

```
            .data
i_static:   .word     0

            .bss
i_uninit:   .word     .-.
```

-1—
0—
+1—

```
            .section  .rodata, ""
i_readonly: .word     5

            .data
j:          .word     0

            .section  .rodata, ""
i2:         .word     9

            .bss
c:          .byte     .-.

            .bss
d:          .word     .-.

            .text

Code goes here.
```

The sections may appear in an arbitrary order, and a given declaration section may appear more than once in your program. As noted previously, when multiple declaration sections of the same type (for example, the three `.bss` sections in the preceding example) appear in a declaration section of your program, Gas combines them into a single group, in any order it pleases.

3.1.7 Memory Access and MMU Pages

The ARM's *memory management unit (MMU)* divides memory into blocks known as *pages*. The OS is responsible for managing pages in memory, so application programs don't typically worry about page organization. However, when working with pages in memory, make sure you're aware of whether the CPU even allows access to a given memory location and whether it is read/write or read-only (write-protected).

Each program section appears in memory in contiguous MMU pages. That is, the `.rodata` section begins at offset 0 in an MMU page and sequentially consumes pages in memory for all the data appearing in that section. The next section in memory (perhaps `.data`) begins at offset 0 in the next MMU page following the last page of the previous section. If that previous section (for example, `.rodata`) does not consume an integral multiple of 4,096 bytes, padding space will be present between the end of that section's data and the end of its last page, to guarantee that the next section begins on an MMU page boundary.

Each new section starts in its own MMU page because the MMU controls access to memory by using page *granularity*. For example, the MMU controls whether a page in memory is readable/writable or read-only. For `.rodata` sections, you want the memory to be read-only. For the `.data` section, you want to allow reads and writes. Because the MMU can enforce these attributes only on a page-by-page basis, you cannot have `.data` section information in the same MMU page as an `.rodata` section.

Normally, all this is completely transparent to your code. Data you declare in a `.data` (or `.bss`) section is readable and writable, and data in an `.rodata` or `.text` section is read-only (`.text` sections are also executable). Beyond placing data in a particular section, you don't have to worry too much about the page attributes.

You do need to worry about MMU page organization in memory in one situation. Sometimes it is convenient to access (read) data beyond the end of a data structure in memory. However, if that data structure is aligned with the end of an MMU page, accessing the next page in memory could be problematic. Some pages in memory are *inaccessible*; the MMU does not allow reading, writing, or execution to occur on that page. Attempting to do so will generate an ARM *segmentation fault*. This will typically crash your program, unless you have an exception handler in place to handle segmentation faults. If you have a data access that crosses a page boundary, and the next page in memory is inaccessible, this will crash your program. For example, consider a half-word access to a byte object at the very end of an MMU page, as shown in Figure 3-2.

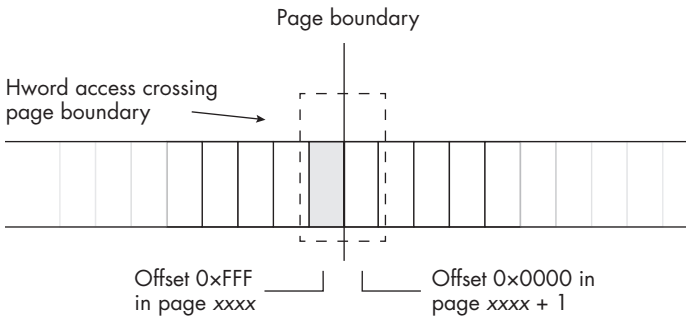

Figure 3-2: Half-word access at the end of a memory-management page

As a general rule, you should never read data beyond the end of a data structure. If for some reason you need to do so, ensure that it is legal to access the next page in memory. It goes without saying that you should never write data beyond the end of a given data structure; this is always incorrect and can create far more problems than just crashing your program (including severe security issues).

3.1.8 PIE and ASLR

As noted in Chapter 1, macOS forces all code to use a position-independent executables (PIE) form. Linux doesn't absolutely require this, but it allows you to write PIE code if you choose. There are two main reasons for PIE code: shared libraries and security, which were covered in "Linux vs. macOS: Position-Independent Executables" on page XX. However, as the behavior of PIE code profoundly affects the way you write ARM assembly language, it is worthwhile to spend a little more time discussing PIE, and especially *address space layout randomization (ASLR)*.

ASLR is an attempt by the OS to thwart various exploits (hacks) that try to figure out where the code and data reside in an application. Prior to PIE and ASLR, most OSs always loaded the executable code and data to the same address in memory, making it easy for a hacker to patch or otherwise mess with the executable program. By loading the code and data sections into random memory locations, PIE/ASLR make it much more difficult for exploits to tap into the executing code.

As a result of ASLR, the layout of an executing program in memory will not actually look like that in Figure 3-1. For one given instance of a program execution, it might look something like Figure 3-3.

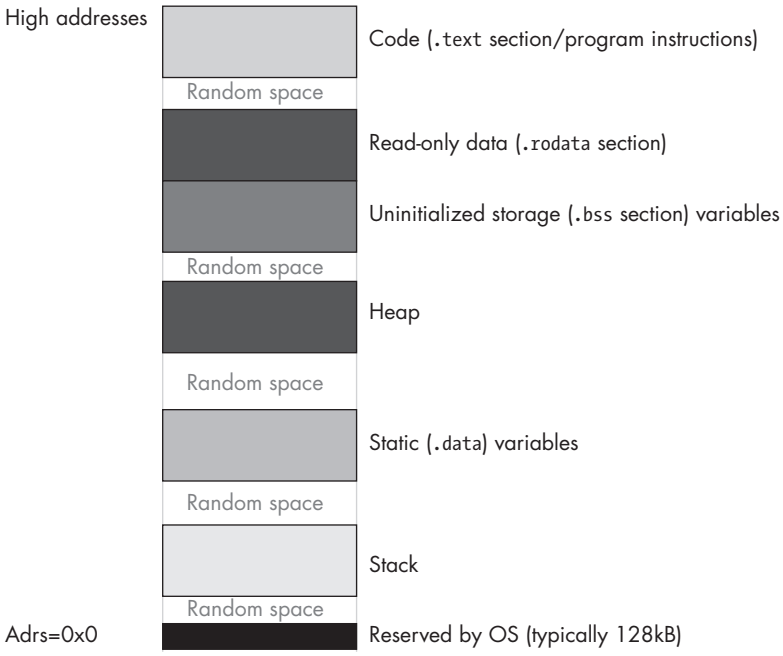

Figure 3-3: A possible memory layout for one execution of an application

However, on the next run of the program, the sections will likely be rearranged and placed at different locations in memory.

While PIE/ASLR makes it difficult for hackers to exploit your code, it also plays havoc with the ARM's instruction set. Consider the following (legitimate) ARM `ldr` instruction:

```
ldr w0, someWordVar  // Assume someWordVar is in .data:q.
!
```

This would normally load the W0 register from the 32-bit variable `someWordVar` found in the `.data` section. This particular instruction uses the *PC-relative addressing mode*, which means that the instruction encodes an offset from the address of the `ldr` instruction to the `someWordVar` variable in

memory. However, if you assemble this program under macOS, you get the following error:

```
error: unknown AArch64 fixup kind!
```

Under Linux (Ubuntu and Raspberry Pi OS seem to be different; your mileage may vary), you get something like

```
relocation truncated to fit: R_AARCH64_LD_PREL_LO19 against `.data'
```

This is a real ARM64 instruction and *should* work. In fact

```
ldr reg, =constant
```

is just a special form of this instruction, and it does work.

The problem is due to the ARM 32-bit instruction length. If you look up the encoding for the `ldr` instruction in the ARM reference manual, you'll discover that it sets aside 19 bits for the address of the memory location. This turns out to be an *offset* (a distance in bytes) from the address of the `ldr` instruction (that is, the value of the 19-bit field is added to the PC to get the actual memory address). Because it's referencing data in the `.text` section, and everything is word-aligned in the text section, the 19-bit offset is actually a word offset, not a byte offset. This effectively gives the `ldr` instruction another 2 bits (the LO 2 bits will always be 0). This effective 21-bit offset allows the `ldr` instruction to access data at a location ±1MB around the `ldr` instruction.

Unfortunately, when accessing data in the `.data` section, which the OS has been nice enough to place at a random address (probably farther than 1MB away), the 21-bit range of the `ldr` instruction won't be sufficient. This is why Gas complains about attempting to access a variable in the `.data` section with the `ldr` instruction. As a bottom line, you can't use that instruction to directly access data unless that data is also in the `.text` section and isn't more than ±1MB away.

3.1.9 The .pool Section

The `.pool` section is a Gas pseudo-section in your program. As noted previously, the following instruction loads a large constant into a register by placing that constant somewhere in memory, then loading the contents of that memory location into the destination register:

```
ldr reg, =largeConstant
```

In other words, this instruction is completely equivalent to either of the following:

```
    ldr x0, a64_bit_constant
    ldr w0, a32_bit_constant
     .
```

```
    .
    .
// Somewhere in the .text section that will never
// be executed as code:

a64_bit_constant: .dword  The_Actual_64bit_Constant_Value
a32_bit_constant: .word  The_Actual_32bit_Constant_Value
```

Gas automatically figures out an appropriate place to put such constants: near the instructions that reference them but out of the code path.

If you'd like to control the placement of these constants in your `.text` section, you can use the `.pool` directive. Wherever you place this directive in your `.text` section (and it must be in the `.text` section), Gas will emit the constants it produces. Just make sure that if you put a `.pool` directive in your code, you place it after an unconditional branch or return instruction so that the program flow won't attempt to execute that data as machine instructions.

Normally, you don't need to place a `.pool` directive in your source code, since Gas will do a reasonable job of finding a location to place its data. However, if you intend to also insert data of your own in the `.text` section, you may want to insert the `.pool` directive and place your data declarations immediately afterward. Note that the data after `.pool` is part of the `.text` section, so you can continue to place machine instructions after the `.text`.

3.2 Gas Storage Allocation for Variables

Gas associates a current *location counter* with each of the declaration sections (`.text`, `.data`, `.rodata`, `.bss`, and any other named sections). These location counters initially contain `0`. Whenever you declare a variable in one of these sections (or write code in a code section), Gas associates the current value of that section's location counter with the label and bumps up the value of that location counter by the size of the object you're declaring.

For example, assume that the following is the only `.data` declaration section in a program:

```
    .data
bb: .byte  0        // Location counter = 0, size = 1
s:  .hword 0        // Location counter = 1, size = 2
w:  .word  0        // Location counter = 3, size = 4
d:  .dword 0        // Location counter = 7, size = 8
q:  .qword 0        // Location counter = 15, size = 16
                    // Location counter is now 31.
```

As you can see, variable declarations listed in a single `.data` section have contiguous offsets (location counter values) into the `.data` section. Given the preceding declaration, `s` will immediately follow `bb` in memory, `w` will immediately follow `s` in memory, `d` will immediately follow `w`, and so on. These offsets aren't the actual runtime addresses of the variables. At runtime, the system loads each section to a base address in memory. The linker

and the OS add the base address of the memory section to each of these location counter values (which we call *displacements*, or *offsets*) to produce the actual memory address of the variables.

OBTAINING THE CURRENT LOCATION COUNTER VALUE

If you ever want to use the current location counter value in your program, Gas will substitute it for a single period (`.`) wherever a constant is allowed, as in the following example:

```
.dword . // Stores the address of this dword in memory
```

You'd normally use the `.` operator to compute lengths of sections of code, using something like the following:

```
lbl:  .byte 0, 1, 2, 3, 4
lbl2: .hword 55
size: .word . - lbl
```

The `. - lbl` expression computes the number of bytes between the `lbl` symbol and the `size` label. The `.` operator returns the location counter value at the beginning of the `.word` directive and does not include the 4 bytes that `.word` will emit to the output file.

Keep in mind that you may link other modules with your program (for example, from the C stdlib) or even additional `.data` sections in the same source file, and the linker has to merge the `.data` sections. Each individual section (even when it has the same name as another section) has its own location counter that starts from 0 when allocating storage for the variables in the section. Hence, the offset of an individual variable may have little bearing on its final memory address.

Gas allocates memory objects you declare in `.rodata`, `.data`, and `.bss` sections in completely different regions of memory. Therefore, you cannot assume that the following three memory objects appear in adjacent memory locations (indeed, they probably will not):

```
    .data
bb: .byte    0

    .section  .rodata, ""
w:  .word    0x1234

    .bss
d:  .dword   .-.
```

In fact, Gas will not even guarantee that variables you declare in separate `.data` (or other) sections are adjacent in memory, even if there is

nothing between the declarations in your code. For example, you cannot assume whether `bb`, `w`, and `d` are—or aren't—in adjacent memory locations in the following declarations:

```
    .data
bb: .byte   0

    .data
w:  .word   0x1234

    .data
d:  .dword  0
```

If your code requires these variables to consume adjacent memory locations, you must declare them in the same `.data` section.

3.3 Little-Endian and Big-Endian Data Organization

As you learned in "The Memory Subsystem" on page XX, the ARM stores multibyte data types in memory, with the LO byte at the lowest address in memory and the HO byte at the highest address (see Figure 1-6). This type of data organization in memory is known as *little endian*. Little-endian data organization, in which the LO byte comes first and the HO byte comes last, is common in many modern CPUs. It is not, however, the only possible approach.

Big-endian data organization reverses the order of the bytes in memory. The HO byte of the data structure appears first, in the lowest memory address, and the LO byte appears in the highest memory address. Table 3-1 describes the memory organization for half words.

Table 3-1: Half-Word Object Little- and Big-Endian Data Organization

Data byte	Memory organization for little endian	Memory organization for big endian
0 (LO byte)	base + 0	base + 1
1 (HO byte)	base + 1	base + 0

Table 3-2 describes the memory organization for words.

Table 3-2: Word Object Little- and Big-Endian Data Organization

Data byte	Memory organization for little endian	Memory organization for big endian
0 (LO byte)	base + 0	base + 3
1	base + 1	base + 2
2	base + 2	base + 1
3 (HO byte)	base + 3	base + 0

Table 3-3 describe the memory organization for double words.

Table 3-3: Dword Object Little- and Big-Endian Data Organization

Data byte	Memory organization for little endian	Memory organization for big endian
0 (LO byte)	base + 0	base + *7*
1	base + 1	base + 6
2	base + 2	base + 5
3	base + 3	base + 4
4	base + 4	base + 3
5	base + 5	base + 2
6	base + 6	base + 1
7 (HO byte)	base + *7*	base + 0

Normally, you wouldn't be too concerned with big-endian memory organization on an ARM CPU. However, on occasion, you may need to deal with data produced by a different CPU (or by a protocol, such as Transmission Control Protocol/Internet Protocol, or TCP/IP) that uses big-endian organization as its canonical integer format. If you were to load a big-endian value in memory into a CPU register, the value would be incorrect.

If you have a 16-bit big-endian value in memory and you load it into a register, its bytes will be swapped. For 16-bit values, you can correct this issue by using the `rev16` instruction, which has the following syntax:

```
rev16 reg_dest, reg_src
```

Here, reg_{dest} and reg_{src} are any 32- or 64-bit general-purpose registers (both must be the same size). This instruction will swap the 2 bytes in each of the 16-bit half-words in the source register; that is, this operates on `hword0` and `hword1` in a 32-bit register and on `hword0`, `hword1`, `hword2`, and `hword3` in a 64-bit register. For example

```
ldr     w1, =0x12345678
rev16   w1, w1
```

will produce 0x34127856 in the W1 register, having swapped bytes 0 and 1 as well as bytes 2 and 3.

If you have a 32-bit value in a register (32- or 64-bit), you can swap the 4 bytes in that register by using the `rev32` instruction:

```
rev32 reg_dest, reg_src
```

Again, the registers can be 32- or 64-bit, but both must be the same size. In a 32-bit register, this will swap bytes 0 and 3 as well as 1 and 2. In a

64-bit register, it will swap bytes 0 and 3, 1 and 2, 7 and 4, and 6 and 5 (see Figure 3-4).

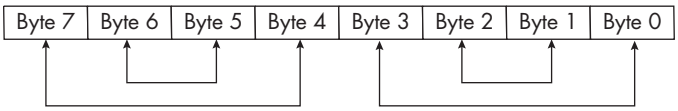

Figure 3-4: Operation of the `rev32` instruction

The rev instruction will swap bytes 7 and 0, 6 and 1, 5 and 2, and 4 and 3 in a 64-bit register (see Figure 3-5).

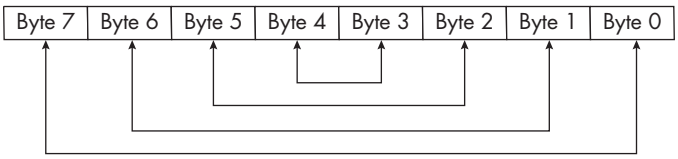

Figure 3-5: Operation of the `rev` instruction

The rev instruction accepts only 64-bit registers.

3.4 Memory Access

"The Memory Subsystem" on page XX describes how the ARM CPU fetches data from memory on the data bus. In an idealized CPU, the data bus is the size of the standard integer registers on the CPU; therefore, you would expect the ARM CPUs to have a 64-bit data bus. In practice, modern CPUs often make the physical data bus connection to main memory much larger in order to improve system performance. The bus brings in large chunks of data from memory in a single operation and places that data in the CPU's *cache*, which acts as a buffer between the CPU and physical memory.

From the CPU's point of view, the cache *is* memory. Therefore, when the remainder of this section discusses memory, it's generally talking about data sitting in the cache. As the system transparently maps memory accesses into the cache, we can discuss memory as though the cache were not present and discuss the advantages of the cache as necessary.

On early processors predating the ARM, memory was arranged as an array of bytes (8-bit machines, such as the Intel 8088), half words (16-bit machines, such as the Intel 8086 and 80286), or words (32-bit machines, such as the 32-bit ARM CPUs). On a 16-bit machine, the LO bit of the address did not physically appear on the address bus. This means the addresses 126 and 127 put the same bit pattern on the address bus (126, with an implicit 0 in bit position 0), as shown in Figure 3-6.

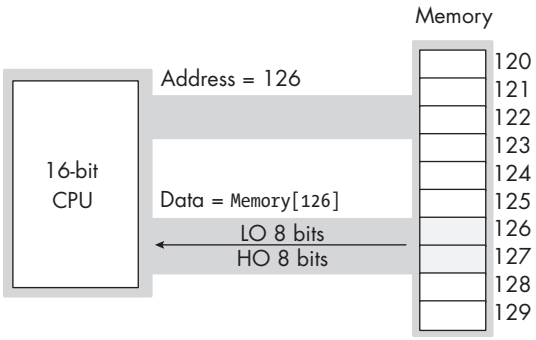

Figure 3-6: The address and data bus for 16-bit processors

When reading a byte, the CPU uses the LO bit of the address to select the LO byte or HO byte on the data bus. Figure 3-7 shows the process when accessing a byte at an even address (126 in this figure).

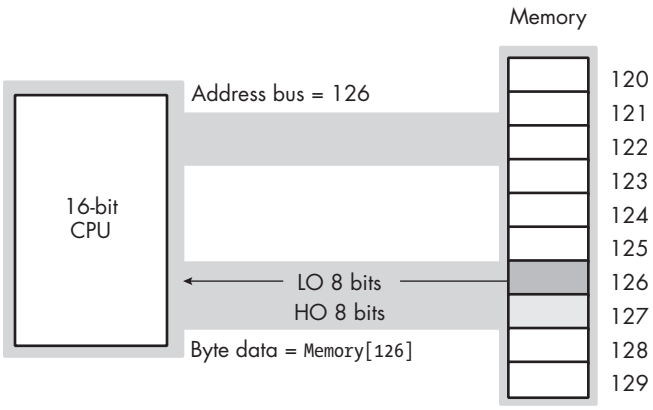

Figure 3-7: Reading a byte from an even address on a 16-bit CPU

Figure 3-8 shows memory access for the byte at an odd address (127 in this figure). Note that in both Figures 3-7 and 3-8, the address appearing on the address bus is 126.

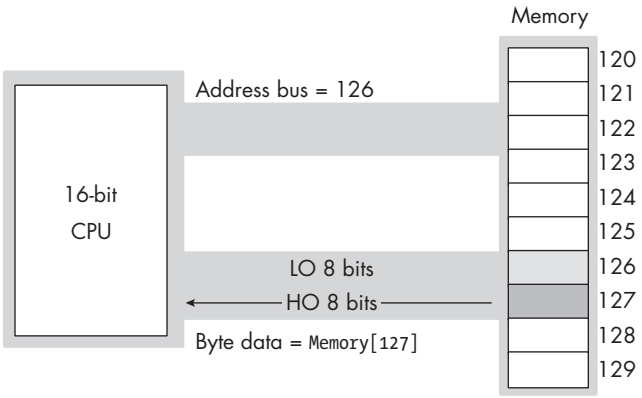

Figure 3-8: Reading a byte from an odd address on a 16-bit CPU

What happens when this 16-bit CPU wants to access 16 bits of data at an odd address? For example, suppose that in these figures, the CPU reads the word at address 125. When the CPU puts address 125 on the address bus, the LO bit doesn't physically appear. Therefore, the actual address on the bus is 124. If the CPU were to read the LO 8 bits off the data bus at this point, it would get the data at address 124, not address 125.

Fortunately, the CPU is smart enough to figure out what's going on here: it extracts the data from the HO 8 bits on the data bus and uses this as the LO 8 bits of the data operand. However, the HO 8 bits that the CPU needs are not found on the data bus. The CPU has to initiate a second read operation, placing address 126 on the address bus, to get the HO 8 bits (these will be sitting in the LO 8 bits of the data bus, but the CPU can figure that out). It takes two memory cycles for this read operation to complete. Therefore, the instruction reading the data from memory will take longer to execute than it would have if the data had been read from an address that was an integral multiple of 2 (16-bit alignment).

The same problem exists on 32-bit processors, except that the 32-bit data bus allows the CPU to read 4 bytes at a time. Reading a 32-bit value at an address that is not an integral multiple of 4 incurs the same performance penalty. However, accessing a 16-bit operand at an odd address doesn't always guarantee an extra memory cycle—only addresses that, when divided by 4, have a remainder of 3 incur the penalty. In particular, if you access a 16-bit value (on a 32-bit bus) at an address where the LO 2 bits contain 0b01, the CPU can read the word in a single memory cycle, as shown in Figure 3-9.

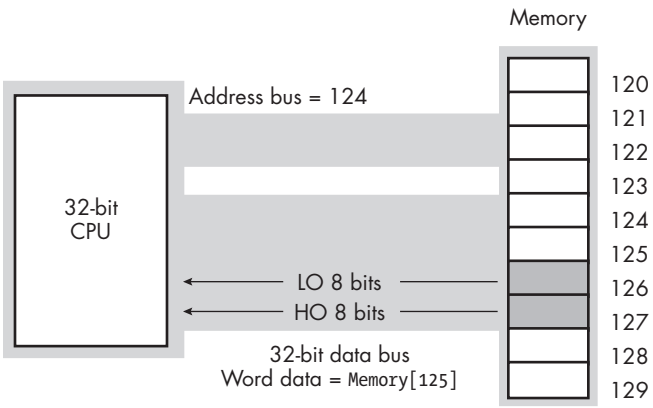

Figure 3-9: Accessing a word on a 32-bit data bus

Modern ARM CPUs with cache systems have largely eliminated this problem. As long as the data (1, 2, 4, or 8 bytes in size) is fully within a *cache line*—a processor-defined number of bytes—no memory cycle penalty occurs for an unaligned access. If the access does cross a cache-line boundary, the CPU will run a little slower while it executes two memory operations to get (or store) the data.

3.5 Gas Support for Data Alignment

To write fast programs, you must ensure that you properly align data objects in memory. Proper *alignment* means that the starting address for an object is a multiple of a certain size—usually the size of an object, if the object's size is a power of 2 for values up to 32 bytes in length. For objects greater than 32 bytes, aligning the object on an 8-, 16-, or 32-byte address boundary is probably sufficient. For objects fewer than 16 bytes, aligning the object at an address that is the next power of 2 greater than the object's size is usually fine.

As noted in the previous section, accessing data that is not aligned at an appropriate address may require extra time. Therefore, if you want to ensure that your program runs as rapidly as possible, you should try to align data objects according to their size.

Data becomes misaligned whenever you allocate storage for different-sized objects in adjacent memory locations. For example, if you declare a byte variable, it will consume 1 byte of storage, and the next variable you declare in that declaration section will have the address of that byte object plus 1. If the byte variable's address happens to be an even address, the variable following that byte will start at an odd address. If that following variable is a half-word, word, or dword object, its starting address will not be optimal.

In this section, we'll explore ways to ensure that a variable is aligned at an appropriate starting address based on its size. Consider the following Gas variable declarations:

```
    .data
w:  .word  0
bb: .byte  0
s:  .hword 0
w2: .word  0
s2: .hword 0
b2: .byte  0
dw: .dword 0
```

The first `.data` declaration in a program places its variables at an address that is an even multiple of 4,096 bytes. Whatever variable first appears in that `.data` declaration is guaranteed to be aligned on a reasonable address. Each successive variable is allocated at an address that is the sum of the sizes of all the preceding variables, plus the starting address of that `.data` section.

Therefore, assuming Gas allocates the variables in the previous example at a starting address of `4096`, it will allocate them at the following addresses:

```
                    // Start Adrs          Length
w:    .word   0     //     4096              4
bb:   .byte   0     //     4100              1
s:    .hword  0     //     4101              2
w2:   .word   0     //     4103              4
s2:   .hword  0     //     4107              2
b2:   .byte   0     //     4109              1
dw:   .dword  0     //     4110              8
```

With the exception of the first variable (which is aligned on a 4KB boundary) and the byte variables (whose alignment doesn't matter), all these variables are misaligned. The `s`, `s2`, and `w2` variables start at odd addresses, and the `dw` variable is aligned on an even address that is not a multiple of 8 (word-aligned but not dword-aligned).

An easy way to guarantee that your variables are aligned properly is to put all the dword variables first, the word variables second, the half-word variables third, and the byte variables last in the declaration, as shown here:

```
      .data
dw:   .dword  0
w:    .word   0
w2:   .word   0
s:    .hword  0
s2:   .hword  0
bb:   .byte   0
b2:   .byte   0
```

This organization produces the following addresses in memory:

```
                  //  Start Adrs         Length
dw:    .dword  0  //     4096              8
w2:    .word   0  //     4104              4
```

```
w3:     .word   0   //      4108            4
s:      .hword  0   //      4112            2
s2:     .hword  0   //      4114            2
bb:     .byte   0   //      4116            1
b2:     .byte   0   //      4117            1
```

As you can see, these variables are all aligned at reasonable addresses.

Unfortunately, it is rarely possible for you to arrange your variables in this manner. While many technical reasons make this alignment impossible, a good practical reason for not doing this is that it doesn't let you organize your variable declarations by logical function (that is, you probably want to keep related variables next to one another, regardless of their size).

To resolve this problem, Gas provides the `.align` and `.balign` directives. As noted in "The Anatomy of an Assembly Language Program" on page XX, the `.align` argument is a value that will be raised to that power of 2, and the `.balign`'s operand is an integer that must be a power of 2 (1, 2, 4, 8, 16, and so on). These directives ensure that the next memory object will be aligned to the specified size.

By default, these directives will pad the data bytes they skip with 0s; in a `.text` section, Gas aligns the code by using `nop` (no-operation) instructions. If you would like to use a different padding value, these two directives allow a second operand:

```
.align  pwr2Alignment, padValue
.balign alignment, padValue
```

Here, `padValue` must be an 8-bit constant, which these directives will use as the padding value. Gas also allows a third argument, which is the maximum allowable padding; see the Gas documentation for more details.

The previous example could be rewritten, using the `.align` directive, as follows:

```
     .data
     .align  2   // Align on 4-byte boundary.
w:   .word   0
bb:  .byte   0
     .align  1   // Align on 2-byte boundary.
s:   .hword  0
     .align  2   // Align on 4-byte boundary.
w2:  .word   0
s2:  .hword  0
b2:  .byte   0
     .align  3   // Align on 8-byte boundary.
dw:  .dword  0
```

If Gas determines that an `.align` directive's current address (location counter value) is not an integral multiple of the specified value, Gas will quietly emit extra bytes of padding after the previous variable declaration until the current address in the `.data` section is a multiple of the specified value. This makes your data larger by a few bytes, in exchange for faster

access to it. Since your data will grow only slightly larger when you use this feature, this is probably a good trade-off.

As a general rule, if you want the fastest possible access, choose an alignment value equal to the size of the object you want to align. That is, align half words to even boundaries with an `.align 1` statement, words to 4-byte boundaries with `.align 2`, double words to 8-byte boundaries with `.align 3`, and so on. If the object's size is not a power of 2, align it to the next higher power of 2.

Data alignment isn't always necessary, since the cache architecture of modern ARM CPUs handles most misaligned data. Use the alignment directives only with variables for which speedy access is absolutely critical.

3.6 The ARM Memory Addressing Modes

For the most part, the ARM uses a very standard RISC *load/store architecture*. This means that it accomplishes almost all memory access by using instructions that load registers from memory or store the value held in registers to memory. The load and store instructions access memory by using memory *addressing modes*, mechanisms the CPU uses to determine the address of a memory location. The ARM memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, structs, pointers, and other complex data types. Mastering ARM addressing modes is an important step toward mastering ARM assembly language.

In addition to loads and stores, ARM uses *atomic instructions*. For the most part, these are variations of the load and store instructions, with a few extra bells and whistles needed for multiprocessing applications. Atomic instructions are beyond the scope of this text; for more information, see the ARM V8 reference manual.

Until now, this book has presented only two mechanisms for accessing memory: the register-indirect addressing mode (for example, `[X0]`) introduced in Chapter 1, and the PC-relative addressing mode discussed in "PIE and ASLR" on page XX. However, the ARM provides more than half a dozen modes (depending on how you count them) for accessing data in memory. The following sections describe each of these modes.

3.6.1 PC-Relative

The PC-relative addressing mode is useful only for fetching values from the `.text` section, as the other sections will likely fall out of the ±1MB range of this addressing mode. Therefore, it is much easier to directly access constant data in the `.text` section than it would be in the `.rodata` section (or another read-only section).

A couple of issues arise when using the PC-relative addressing mode in the `.text` section. First, because the 19-bit offset buried in the 32-bit instruction encoding is shifted left 2 bits to produce a word offset (as discussed earlier), you can load only word and double-word values when using this addressing mode—no bytes or half words. For example, you can access byte

and half-word values in the .text section with the register-indirect addressing mode, but not with the PC-relative addressing mode.

When accessing data in the .text section by using the PC-relative addressing mode, keep the following points in mind:

- Under macOS, all labels in the .text section must be aligned on a 4-byte boundary, even if the data associated with that label doesn't require such alignment (such as bytes and half words).
- Data values in the .text section cannot refer to other sections (for example, pointer constants, discussed in Chapter 4). However, such objects can refer to data within the .text section itself (this is important for *jump tables*, covered in Chapter 7).
- The data must reside within ±1MB of the instruction(s) that reference it. For example, you cannot create an array of data that exceeds 1MB.
- Only word and dword accesses are allowed when using the PC-relative addressing mode.
- As the data resides in the .text section, it is read-only; you cannot put variables in the .text section.

To use the PC-relative addressing mode, just reference the label you used to declare the object in the .text section:

```
        ldr w0, wordVar
          .
          .
          .
wordVar: .word 12345
```

Don't forget that all data declarations you put in the .text section need to be out of the execution path, preferably in the .pool section. (You'll see an exception to this rule in Chapter 5 when I discuss passing parameters in the code stream.)

3.6.2 Register-Indirect

Up to this point, most examples in this book have used the register-indirect addressing mode. *Indirect* means that the operand is not the actual address, but that the operand's value specifies the memory address to use. In a register-indirect addressing mode, the value held in the register is the address of the memory location to access. For example, the instruction

```
ldr x0, [x1]
```

tells the CPU to load X0's value from the location whose address is currently in X1. The square brackets around X1 tell Gas to use the register-indirect addressing mode.

The ARM has 32 forms of this addressing mode, one for each of the 32 general-purpose 64-bit registers (though X31 is not legal; use SP instead).

You cannot specify a 32-bit register in the square brackets when using an indirect addressing mode.

Technically, you could load a 64-bit register with an arbitrary numeric value and access that location indirectly by using the register-indirect addressing mode:

```
ldr x1, =12345678
ldr x0, [x1]    // Attempts to access location 12345678
```

Unfortunately (or fortunately, depending on how you look at it), this will probably cause the OS to generate a segmentation fault because it's not always legal to access arbitrary memory locations. There are better ways to load the address of an object into a register, as you'll see shortly.

You can use the register-indirect addressing modes to access data referenced by a pointer, to step through array data, and, in general, whenever you need to modify an object's address while your program is running.

When using a register-indirect addressing mode, you refer to the value of a variable by its numeric memory address (the value you load into a register) rather than by the name of the variable. This is an example of using an *anonymous variable*.

The *aoaa.inc* include file provides the `lea` macro, which you can use to take the address of a variable and put it into a 64-bit register:

```
lea x1, j
```

After executing this `lea` instruction, you can use the `[x1]` register-indirect addressing mode to indirectly access the value of `j` (which is how almost every example up to this point has accessed memory). In "Getting the Address of a Memory Object" on page XX, you'll see how the `lea` macro works.

3.6.3 Indirect-Plus-Offset

Consider the following data declaration, similar to other examples given in this book:

```
bVar:  .byte 0, 1, 2, 3
```

If you load X1 with the address of `bVar`, you can access that byte (`0`) by using an instruction such as this:

```
ldrb w1, [x1]   // Load byte at bVar (0) into w1.
```

To access the other 3 bytes following that `0` in memory, you can use the *indirect-plus-offset* addressing mode. Here is the mode's syntax:

```
[Xn|SP, #signed_expression]
```

`Xn|SP` means X0 to X30 or SP, and *`signed_expression`* is a small integer expression in the range –256 to +255. This particular addressing mode will compute the sum of the address in X*n* (*n* = 0 to 30, or SP) with the signed constant and use that as the *effective memory address* (the memory address to access).

For example, if X1 contains the address of `bVar` from the previous example, the following instruction will fetch the byte just beyond `bVar` (that is, the byte containing `1` in that example):

```
ldrb  w0, [x1, #1] // Fetch byte at address X1 + 1.
```

Once again, the 32-bit instruction size severely limits the range of this addressing mode (only 9 bits are available for the signed offset). If you need a greater offset, you must explicitly add a value to the address in X1 (perhaps using a different register if you need to maintain the base address in X1). For example, the following code does this using X2 to hold the effective address:

```
add  x2, x1, #2000  // Access location X1 + 2000.
ldrb w2, [x2]
```

This computes `X2 = X1 + 2000` and loads W2 with the word at that address.

3.6.4 Scaled Indirect-Plus-Offset

The *scaled indirect-plus-offset* addressing mode is a somewhat more complex variant of the indirect-plus-offset mode. It incorporates a 12-bit unsigned constant into the instruction encoding that is scaled (multiplied) by 1, 2, 4, or 8, depending on the size of the data transfer. This provides a range extension to the 9-bit signed offset of the indirect-plus-offset mode.

This addressing mode uses the same syntax as the indirect-plus-offset addressing mode, except that it doesn't allow signed offsets:

```
[Xn|SP, #unsigned_expression]
```

For byte transfers (`ldrb`), the unsigned expression can be a value in the range 0 to 0xFFF (4,095). For half-word transfers (`ldrh`), the unsigned expression can be a value in the range 0 to 0x1FFE, but the offset must be even. For word transfers (`ldr`), the unsigned expression must be in the range 0 to 0x3FFC and must also be divisible by 4. For dword transfers, the unsigned expression must be in the range 0 to 0x7FF8 and must be divisible by 8. As you'll see in Chapter 4, these numbers work great for accessing elements of a byte, half-word, word, or double-word array.

Generally, the assembler will automatically select between the indirect-plus-offset and scaled indirect-plus-offset addressing modes, based on the value of the offset appearing in the addressing mode. Sometimes the choice might be ambiguous. For example:

```
ldr  w0, [X2, #16]
```

Here, the assembler could choose the scaled or unscaled versions of the addressing mode. Typically, it would choose the scaled form. Its decision shouldn't matter to your code; either form will load the appropriate word in memory into the W0 register.

If, for some reason, you wish to explicitly specify the unscaled addressing mode, you can do so using the `ldur` and `stur` instructions (load or store register unscaled).

3.6.5 Pre-indexed

The *pre-indexed* addressing mode is very similar to the indirect-plus-offset addressing mode, insofar as it combines a 64-bit register and a signed 9-bit offset. However, this addressing mode copies the sum of the register and offset into the register before accessing memory. In the end, it accesses the same address as the indirect-plus-offset mode, but once the instruction finishes, the index register points into memory at the indexed location. This mode is useful for stepping through arrays and other data structures by incrementing the register after each access in a loop.

Here's the syntax for the pre-indexed addressing mode:

```
[Xn|SP, #signed_expression]!  // Xn|SP has the usual meaning.
```

The `!` at the end of this sequence differentiates the pre-indexed addressing mode. As with the indirect-plus-offset mode, the *signed_expression* value is limited—in this case, to 9 bits (–256 to +255).

The following code fragment uses this addressing mode:

```
bVar:  .byte 0, 1, 2, 3
         .
         .
         .
       lea  x0, bVar-1  // Initialize with adrs of bVar - 1.
       Mov  x1, 4
loop:  ldrb w2, [x0, #1]!

       Do something with the byte in w2.

       Subs x1, x1, #1
       bne  loop
```

On the first iteration of this loop, the addressing mode adds `1` to X0 so that it points at the first byte in the `bVar` array of 4 bytes. This also leaves X0 pointing at that first byte. On each successful iteration of the loop, X0 is incremented by 1, accessing the next byte in the `bVar` array.

The `subs` instruction will set the Z flag when it decrements X1 down to 0. When that happens, the `bne` (branch if Z = 0) instruction will fall through, terminating the loop.

3.6.6 Post-Indexed

The post-indexed addressing mode is very similar to the pre-indexed addressing mode, except it uses the value of the register as the memory address *before* updating the register with the signed immediate value. Here's the syntax for the post-indexed addressing mode:

```
[Xn|SP], #signed_expression  // Xn|SP has the usual meaning.
```

Again, the *signed_expression* is limited to 9 bits (–256 to +255).

The example of the previous section can be rewritten and slightly improved by using the post-indexed addressing mode:

```
bVar:  .byte 0, 1, 2, 3
         .
         .
         .
       lea  x0, bVar
       mov  x1, 4
loop:  ldrb w2, [x0], #1

       Do something with the byte in w2.

       Subs x1, x1, #1
       bne  loop
```

This example starts with X0 pointing at `bVar` and ends with X0 pointing at the first byte beyond the (four-element) `bVar` array. On the first iteration of this loop, the `ldr` instruction first uses the value in X0, pointing at `bVar`, then increments X0 after fetching the byte where X0 points.

3.6.7 Scaled-Indexed

The *scaled-indexed* addressing mode contains two register components (rather than a register and an immediate constant) that form the effective address. The syntax for this mode is the following:

```
[Xn|SP, Xi]
[Xn|SP, Wi, extend]
[Xn|SP, Xi, extend]
```

The first form is the easiest to understand: it computes the effective address (EA) by adding the values in X*n* (or SP) and X*i*. Generally, X*n* (or SP) is known as the *base address*, and the value in X*i* is the index (which must be X0 to X30 or XZR). The base address is the lowest memory address of an object, and the index is an offset from that base address (much like the immediate constants in the indirect-plus-offset addressing mode). This is just a simple *base + index* addressing mode: no scaling takes place.

WHY XN|SP, NOT X31?

As noted in "The ARM64 CPU Architecture" on page XX, the stack pointer register, SP, is the same as X31. However, if you try to use X31 as the base register in an addressing mode, Gas will report an error. This is because the ARM64 CPU actually maps two separate registers to X31: SP and XZR (the zero register). You use one of those register names rather than X31.

In addressing modes, the ARM does not allow you to use XZR as a base register. You can, however, use SP as the base register. Conversely, XZR *is* allowed as an index register (though it's somewhat redundant to do so), and SP is not allowed there.

The base + index form is useful in these situations:

- You have a pointer to an array object in a register (X*n*, the base address), and you want to access an element of that array by using an integer index (typically in a memory variable). In this case, you would load the index into the index register (X*i*) and use the base + index mode to access the actual element.
- You want to use the indirect-plus-offset addressing mode, but the offset is outside the range –256 to +255. In this case, you can load the larger offset into X*i* and use the base + index addressing mode to access the memory location regardless of the offset.

The second and third forms of the scaled-indexed addressing mode provide an extension/scaling operation, which is quite useful for indexing into arrays whose element size is larger than a byte. Of these two scaled-indexed modes, one uses a 32-bit register as the index register, and the other uses a 64-bit register.

The 32-bit form is convenient because most of the time indexes into an array are held in a 32-bit integer variable. If you load that 32-bit integer into a 32-bit register (W*i*), you can easily use it as an index into an array with the

```
[Xn, Wi, extend]
```

form of the scaled-indexed addressing mode.

Ultimately, all effective addresses turn out to be 64 bits. In particular, when the CPU adds X*n* and W*i* together, it must somehow extend the W*i* index value to 64 bits prior to adding them. The *extend* operator tells Gas how to extend W*i* to 64 bits.

The simplest forms of *extend* are the following:

```
[Xn|SP, Wi, uxtw]
[Xn|SP, Wi, sxtw]
```

The [X*n*|SP, W*i*, uxtw] form zero-extends W*i* to 64 bits before adding it to X*n*, while the [X*n*|SP, W*i*, sxtw] form sign-extends W*i* to 64 bits before the addition.

Another form of the scaled-indexed addressing mode introduces the *scaled* component. This form allows you to load elements from an array of bytes, half words, words, or dwords scaled by the size of the array element (1, 2, 4, or 8 bytes). These particular forms are not stand-alone addressing modes that can be used with an arbitrary ldr or str instruction. Instead, each addressing mode form is tied to a specific instruction size. The following is the allowable syntax for the ldrb/ldrsb and strb instructions (W*d* is a 32-bit destination register, and W*s* is a 32-bit source register):

```
ldrb  Wd, [Xn|SP, Wi, sxtw #0]  // #0 is optional,
ldrb  Wd, [Xn|SP, Wi, uxtw #0]  // 0 is default shift.
ldrb  Wd, [Xn|SP, Xi, lsl #0]

ldrsb Wd, [Xn|SP, Wi, sxtw #0]
ldrsb Wd, [Xn|SP, Wi, uxtw #0]
ldrsb Wd, [Xn|SP, Xi, lsl #0]

strb  Ws, [Xn|SP, Wi, sxtw #0]
strb  Ws, [Xn|SP, Wi, uxtw #0]
strb  Ws, [Xn|SP, Xi, lsl #0]
```

These forms zero- or sign-extend W*i* (or X*i*) and add the result with X*n* to produce the EA. The previous instructions are equivalent to the following (because the #0 is optional):

```
ldrb  Wd, [Xn|SP, Wi, sxtw]
ldrb  Wd, [Xn|SP, Wi, uxtw]
ldrb  Wd, [Xn|SP, Xi]

ldrsb Wd, [Xn|SP, Wi, sxtw]
ldrsb Wd, [Xn|SP, Wi, uxtw]
ldrsb Wd, [Xn|SP, Xi]

strb  Ws, [Xn|SP, Wi, sxtw]
strb  Ws, [Xn|SP, Wi, uxtw]
strb  Ws, [Xn|SP, Xi]
```

For the ldrh/ldrsh and strh instructions, you can specify either the 0 (×1) or 1 (×2) scale factor:

```
ldrh  Wd, [Xn|SP, Wi, sxtw #1]  // #0 is also legal, or
ldrh  Wd, [Xn|SP, Wi, uxtw #1]  // no immediate value (which
ldrh  Wd, [Xn|SP, Xi, lsl #1]   // defaults to 0).

ldrsh Wd, [Xn|SP, Wi, sxtw #1]
ldrsh Wd, [Xn|SP, Wi, uxtw #1]
ldrsh Wd, [Xn|SP, Xi, lsl #1]
```

```
strh  Ws, [Xn|SP, Wi, sxtw #1]
strh  Ws, [Xn|SP, Wi, uxtw #1]
strh  Ws, [Xn|SP, Xi, lsl #1]
```

With a scaling factor of `#1`, these addressing modes compute W*i* × 2 or X*i* × 2 (after any zero or sign extension) and then add the result with the value in X*n* to produce the EA. This scales the EA to access half-word values (2 bytes per array element). If the scaling factor is `#0`, no scaling occurs, as the scaling factor is 2^0. The preceding code must multiply W*i* or X*i* by an appropriate scaling factor, if needed. Loading or storing half words allows a scaling factor of only 0 or 1.

For the 32-bit `ldr` instruction (W*d* is the destination register) and `str` instruction (W*s* is the 32-bit source register), the allowable scaling factors are 0 (×1) or 2 (×4):

```
ldr  Wd, [Xn|SP, Wi, sxtw #2]  // #0 is also legal, or
ldr  Wd, [Xn|SP, Wi, uxtw #2]  // no immediate value (which
ldr  Wd, [Xn|SP, Xi, lsl #2]   // defaults to 0).

str  Ws, [Xn|SP, Wi, sxtw #2]
str  Ws, [Xn|SP, Wi, uxtw #2]
str  Ws, [Xn|SP, Xi, lsl #2]
```

Finally, for the 64-bit `ldr` and `str` instructions, the allowable scaling factors are 0 (×1) and 3 (×8):

```
ldr  Xd, [Xn|SP, Wi, sxtw #3]  // #0 is also legal, or
ldr  Xd, [Xn|SP, Wi, uxtw #3]  // no immediate value (which
ldr  Xd, [Xn|SP, Xi, lsl #3]   // defaults to 0).

str  Xs, [Xn|SP, Wi, sxtw #3]
str  Xs, [Xn|SP, Wi, uxtw #3]
str  Xs, [Xn|SP, Xi, lsl #3]
```

You'll see the main uses for the scaled-indexed addressing modes in the next chapter, when it discusses accessing elements of arrays.

3.7 Address Expressions

Often, when accessing variables and other objects in memory, you will need to access locations immediately before or after a variable rather than at the address of the variable. For example, when accessing an element of an array, or a field of a struct, the exact element or field is probably not at the address of the variable itself. *Address expressions* provide a mechanism to access memory at an offset from the variable's address.

Consider the following legal Gas syntax for a memory address. This isn't a new addressing mode but simply an extension of the PC-relative addressing mode:

```
varName + offset
```

This form computes its effective address by adding the constant offset to the variable's address. For example, the instruction

```
ldr W0, i + 4
```

loads the W0 register with the word in memory that is 4 bytes beyond the `i` object (which, presumably, is in the `.text` section; see Figure 3-10).

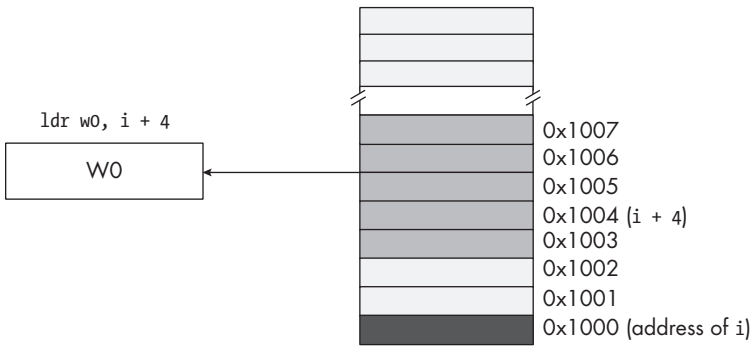

Figure 3-10: Using an address expression to access data beyond a variable

The *offset* value in this example must be a constant (for example, `3`). If *Index* is a word variable, then *varName* + *Index* is not a legal address expression. If you wish to specify an index that varies at runtime, you must use one of the indirect or scaled-indexed addressing modes. Also remember that the offset in *varName* + *offset* is a byte address. This does not properly index into an array of objects unless *varName* is an array of bytes.

NOTE *The ARM CPU does not allow the use of the `ldrb` and `ldrh` instructions when using the PC-relative addressing mode. You can only load words or double words when using this addressing mode. Furthermore, because the instructions don't encode the LO 2 bits of the offset, any offset you specify using an address expression must be a multiple of 4.*

Until this point, the offset in the addressing mode examples has always been a single numeric constant. However, Gas also allows a constant expression anywhere an offset is legal. A *constant expression* consists of one or more constant terms manipulated by operators such as addition, subtraction, multiplication, division, and a wide variety of others, as shown in Table 3-4. Note that operators at the same precedence level are left-associative.

Table 3-4: Gas Constant Expression Operators

Operator	Precedence	Description
+	3	Unary plus (no effect on expression)
-	3	Unary minus (negates expression)
*	2	Multiplication
/	2	Division
<<	2	Shift left
>>	2	Shift right
|	1	Bitwise OR
&	1	Bitwise AND
^	1	Bitwise XOR
!	1	Bitwise AND-NOT
+	0	Addition
-	0	Subtraction

Most address expressions, however, involve only addition, subtraction, multiplication, and sometimes division. Consider the following example:

```
ldr w0, X + 2*4
```

This instruction will move the byte at address `X + 8` into the W0 register.

The value `X + 2*4` is an address expression that is always computed at compile time, never while the program is running. When Gas encounters the preceding instruction, it calculates

```
2 × 4
```

on the spot and adds this result to the base address of X in the `.text` section. Gas encodes this single sum (base address of X plus 8) as part of the instruction; it does not emit extra instructions (that would waste time) to compute this sum for you at runtime. Because Gas computes the value of address expressions at compile time, and therefore Gas cannot know the runtime value of a variable while it is compiling the program, all components of the expression must be constants.

Address expressions are useful for accessing the data in memory beyond a variable, particularly when you've used directives like `.byte`, `.hword`, `.word`, and so on in a `.data` or `.text` section to tack on additional values after a data declaration. For example, consider the program in Listing 3-1 that uses address expressions to access the four consecutive words associated with memory object `i` (each word is 4 bytes apart in memory).

```
// Listing3-1.S
//
// Demonstrates address expressions
```

—-1
—0
—+1

```
#include "aoaa.inc"

            .data
saveLR:     .dword      0
outputVal:  .word       0

ttlStr:     .asciz      "Listing 3-1"
fmtStr1:    .asciz      "i[0]=%d "
fmtStr2:    .asciz      "i[1]=%d "
fmtStr3:    .asciz      "i[2]=%d "
fmtStr4:    .asciz      "i[3]=%d\n"

            .text
            .extern     printf

            .align      2
i:          .word       0, 1, 2, 3

// Return program title to C++ program:

            .global     getTitle
getTitle:
            lea         x0, ttlStr
            ret

// Here is the asmMain function:

            .global     asmMain
asmMain:

// "Magic" instruction offered without
// explanation at this point:

            sub     sp, sp, #256

// Save LR so we can return to the C++
// program later:

            lea     x0, saveLR
            str     lr, [x0]

// Demonstrate the use of address expressions:

            lea     x0, fmtStr1
          ❶ ldr     w1, i + 0
            lea     x2, outputVal
            str     w1, [x2]
            vparm2  outputVal
            bl      printf

            lea     x0, fmtStr2
          ❷ ldr     w1, i + 4
            lea     x2, outputVal
            str     w1, [x2]
```

```
    vparm2  outputVal
    bl      printf

    lea     x0, fmtStr3
❸   ldr     w1, i + 8
    lea     x2, outputVal
    str     w1, [x2]
    vparm2  outputVal
    bl      printf

    lea     x0, fmtStr4
❹   ldr     w1, i + 12
    lea     x2, outputVal
    str     w1, [x2]
    vparm2  outputVal
    bl      printf

    lea     x0, saveLR
    ldr     lr, [x0]
    add     sp, sp, #256
    ret
```

Loading W1 from location `i + 0` fetches `0` from the word array ❶. Loading W1 from location `i + 4` fetches `1` from the second word in the array, located 4 bytes beyond the first element ❷. Loading W1 from location `i + 8` fetches `2` from the third word in the array ❸, located 8 bytes beyond the first element. Loading W1 from location `i + 12` fetches `3` from the fourth word in the array ❹, located 12 bytes beyond the first element.

Here's the program's output:

```
$ ./build Listing3-1
$ ./Listing3-1
Calling Listing 3-1:
i[0]=0 i[1]=1 i[2]=2 i[3]=3
Listing 3-1 terminated
```

Because the value at the address of `i` is `0`, the output displays the four values `0`, `1`, `2`, and `3` as though they were array elements. The address expression `i + 4` tells Gas to fetch the word appearing at `i`'s address plus 4. This is the value `1`, because the `.word` statement in this program emits the value `1` to the `.text` segment immediately after the (word/4-byte) value `0`. Likewise, for `i + 4` and `i + 8`, this program displays the values `2` and `3`.

3.8 Getting the Address of a Memory Object

Up to this point, this book has used the `lea` macro to obtain the address of a memory object. Now that this chapter has provided the necessary prerequisite information, instead of treating `lea` like a black box, it's time to look behind the curtains to see what this macro is doing for you.

The ARM CPU provides two instructions for computing the effective address of a symbol in an assembly language program. The first is `adr`:

```
adr Xd, label
```

This instruction loads the 64-bit destination register (X*d*) with the address of the specified label. Because instruction encodings (operation codes, or *opcodes*) are limited to 32 bits, a huge caveat is attached to `adr`: it has room for only a 21-bit offset within the opcode, so *`label`* must be a PC-relative address within ±1MB of the `adr` instruction. This effectively limits `adr` to taking the address of symbols within the `.text` section.

To rectify this situation, the ARM CPU also provides the `adrp` (address of a page) instruction. This instruction has roughly the same generic syntax as `adr`:

```
adrp Xd, label
```

The instruction loads the address of the MMU page containing the *`label`* into the destination register. By adding the offset of the label into that page to the value in X*d*, you can obtain the actual address of the memory object, using code that looks something like this:

```
adrp Xd, label
add  Xd, Xd, page_offset_of_label
```

At this point, X*d* will contain the address of *`label`*.

This scheme has a couple of issues: first, computing the page offset of the *`label`* symbol is done differently in macOS versus Linux. Second, when you use the syntax just given to try the `adrp` instruction, you'll find that Gas rejects this on macOS.

Let's first consider the Linux solutions to these problems, as they're a little simpler than those for macOS. If you're not creating a PIE application and the symbol is less than ±1MB away, you don't have to use the `adrp` instruction. Instead, you can get by with the single `adr` instruction. If the data is more than ±1MB from the `adr`, you must use the `adrp` version. If you need to reference a memory object outside the `.text` section, you must use the `adrp/add` sequence. Here's the code to do this:

```
adrp x0, label
add  x0, x0, :lo12:label
```

The `:lo12:` item is a special operator that tells Gas to extract the LO 12 bits of *`label`*'s relocatable address; this value is the index into a 4,096-byte memory management page. For more information on this operator, see "For More Information" on page XX. Unfortunately, the macOS assembler uses a completely different syntax to obtain the LO 12 bits of an address; you must use the following instead:

```
adrp x0, label@PAGE
add  x0, x0, label@PAGEOFF
```

The `lea` macro resolves this issue, automatically expanding into the appropriate sequence for whichever OS you're using.

LINUX VS. MACOS: ABSOLUTE ADDRESSES

Apple's macOS (and presumably, iOS, iPadOS, and so on) is far more restrictive about what you can and cannot do in a PIE program. Specifically, macOS does not allow any absolute pointers in your `.text` section that reference other sections. Linux, on the other hand, doesn't have a problem with this at all, in either PIE or non-PIE mode.

For example, say you're working in Linux and have the following symbol in your `.data` section:

```
var: .word 55
```

You can use the instruction

```
ldr x0, =var
```

to load the address of that symbol into X0. If you try to use this instruction in macOS, however, the program will give the following complaint:

```
ld: Absolute addressing not allowed in arm64 code but used in
          'noPrint' referencing 'var'
```

Likewise, if you put the statement

```
ptrToVar: .dword var
```

in your `.text` section somewhere, Linux is perfectly happy with it, but macOS will reject it, using roughly the same message.

Pointers into the `.text` section from other sections are perfectly acceptable to Gas under macOS. Apparently, Apple thinks that the only way hackers are going to determine your data memory location is by looking for addresses buried in the executable code, while pointers in your `.data`, `.rodata`, and other sections are immune to such attacks.

Ultimately, this means that you'll need to use the `adrp` instruction (or the `lea` macro) to obtain at least your first pointer out of the `.text` section. This makes assembly language programming a touch more difficult under macOS than under Linux. Fortunately, the `lea` macro helps smooth out these issues.

3.9 The Push and Pop Operations

The ARM maintains a hardware stack in the stack segment of memory (for which the OS reserves the storage). The *stack* is a dynamic data structure that grows and shrinks according to certain needs of the program. It also stores important information about the program, including local variables, subroutine information, and temporary data.

The ARM CPU controls its stack via the SP register. When your program begins execution, the OS initializes SP with the address of the last memory location in the stack memory segment. Data is written to the stack segment by *pushing* data onto the stack and *popping* it off the stack.

The ARM stack must always be 16-byte aligned—that is, the SP register must always contain a value that is a multiple of 16. If you load the SP register with a value that is not 16-byte aligned, the application will immediately terminate with a bus error fault. One of the stack's primary purposes is to provide a temporary storage area where you can save things such as register values. You will typically push a register's value onto the stack, do some work (such as calling a function) that uses the register, and then pop that value off the stack and back into the register when you want to restore its value. However, the general-purpose registers are only 64 bits (8 bytes); pushing a dword value on the stack will not leave it 16-byte aligned, which will crash the system.

In this section, I'll describe how to push and pop register values. Then I'll present three solutions to the problem of pushing dword values that don't leave the stack 16-byte aligned: wasting storage; pushing two registers simultaneously; and reserving storage on the stack, then moving the register's data into this reserved area.

3.9.1 Using Double Loads and Stores

The `ldp` instruction will load two registers from memory simultaneously. The generic syntax for this instruction is shown here:

```
ldp  Xd1, Xd2, mem  // mem is any addressing mode
ldp  Wd1, Wd2, mem  // except PC-relative.
```

The first form will load Xd_1 from the memory location specified by *mem* and Xd_2 from the memory location 8 bytes later. The second form will load Wd_1 from the specified memory location and Wd_2 from the location 4 bytes later.

The `stp` instruction has a similar syntax; it stores a pair of registers into adjacent memory locations:

```
stp  Xd1, Xd2, mem  // Store Xd1 to mem, Xd2 to mem + 8.
stp  Wd1, Wd2, mem  // Store Wd1 to mem, Wd2 to mem + 4.
                    // mem is any addressing mode except
                    // PC-relative.
```

These instructions have many uses. With respect to using the stack, however, the forms that load and store a pair of 64-bit registers will manipulate 16 bytes at a time—exactly what you need when pushing and popping data on the stack.

3.9.2 Executing the Basic Push Operation

Many CPUs, such as the Intel x86-64, provide an explicit instruction that will push a register onto the stack. Because of the 16-byte stack alignment requirement, you can't push a single 8-byte register onto the stack (without creating a stack fault). However, if you're willing to use 16 bytes of space on the stack to hold a single register's value, you can push that register's value on the stack with the following instruction:

```
str Xs, [sp, #-16]!
```

Remember, the pre-indexed addressing mode will first add –16 to SP and then store X*s* (the source register) at the new location pointed at by SP. This store operation writes only to the LO 8 bytes of the 16-byte block created by dropping SP down by 16 (wasting the HO 8 bytes). However, this scheme keeps the CPU happy, so you won't get a bus error.

This push operation does the following:

```
SP := SP - 16
[SP] := Xs
```

For example, assuming that SP contains 0x00FF_FFE0, the instruction

```
str x0, [sp, #-16]!
```

will set SP to 0x00FF_FFD0 and store the current value of X0 into memory location 0x00FF_FFD0, as Figures 3-11 and 3-12 show.

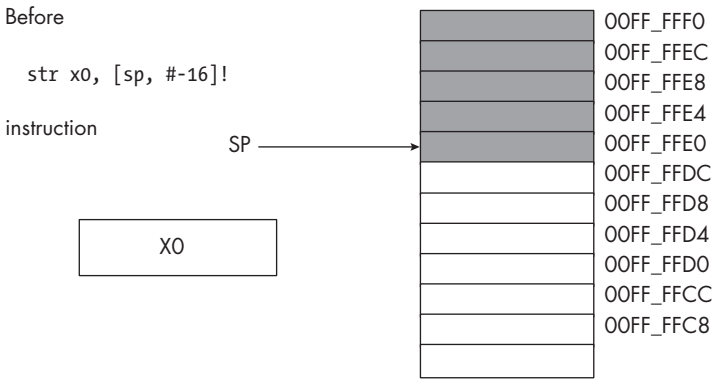

Figure 3-11: The stack segment before the `str x0, [sp, #-16]!` operation

After the `str` instruction, the stack looks like Figure 3-12.

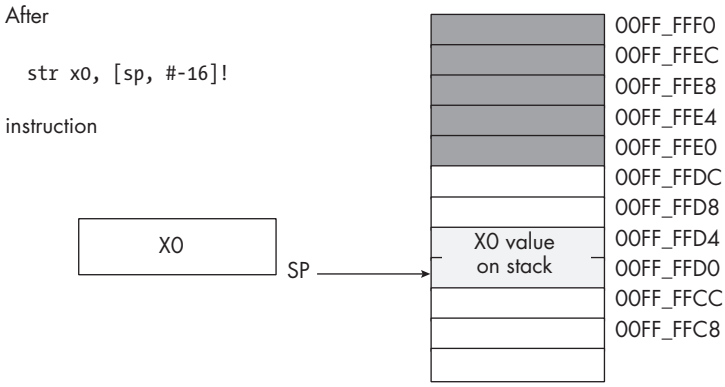

Figure 3-12: The stack segment after the `str x0, [sp, #-16]!` operation

Although this wastes 8 bytes of space on the stack (at addresses 0x00FF_FFD8 through 0x00FF_FFDF), the usage is probably temporary, and the stack space will be reclaimed when the program pops the data off the stack later.

3.9.3 Executing the Basic Pop Operation

The pop operation can be handled using the post-indexed addressing mode and a `ldr` instruction:

```
ldr Xd, [sp], #16
```

This instruction fetches the data from the stack, where SP is pointing, and copies that data into the destination register (X*d*). When the operation is complete, this instruction adjusts SP by 16, restoring it to its original value (its value before the push operation). Figure 3-13 shows the stack before the pop operation.

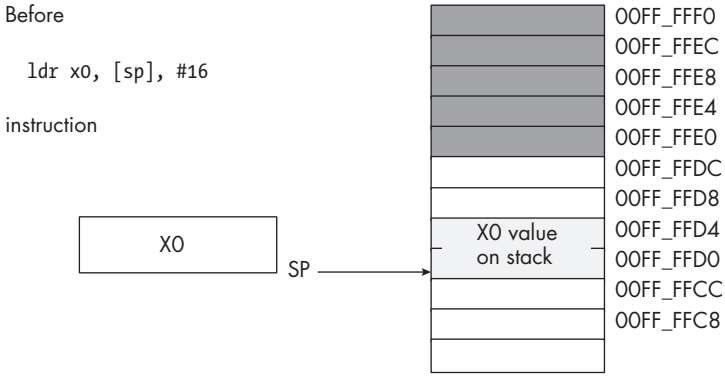

Figure 3-13: Before the pop operation

Figure 3-14 shows the stack organization after executing `ldr`.

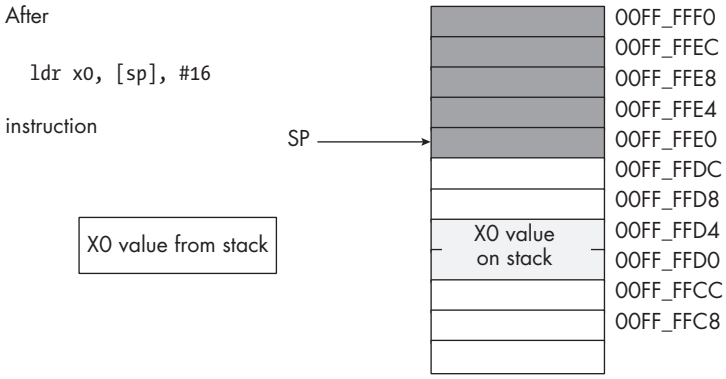

Figure 3-14: After the pop operation

Popping a value does not erase the value in memory; it just adjusts the stack pointer so that it points at the next value above the popped value. However, never attempt to access a value you've popped off the stack. The next time something is pushed onto the stack, the popped value will be obliterated. Because your code isn't the only thing that uses the stack (for example, the OS uses the stack to do subroutines), you cannot rely on data remaining in stack memory once you've popped it off the stack.

3.9.4 Pushing and Popping Registers in Other Ways

If you need to preserve at least two registers, you can reclaim the wasted space shown in Figures 3-11 and 3-12 by using the `stp` instruction rather than `str`. The following code fragment demonstrates how to push and pop both X0 and X7 simultaneously:

```
stp  x0, x7, [sp, #-16]!
 .
 .   // Use X0 and X7 for other purposes.
 .
ldp  x0, x7, [sp], #16  // Restore X0 and X7.
```

The third way to push data on the stack is to drop SP down by a multiple of 16 bytes and then store the value into the stack area by indexing off the SP register. The following code does basically the same thing as the `stp`/`ldp` pair:

```
sub  sp, sp, #16   // Make room for X0 and X7.
stp  x0, x7, [sp]
 .
 .   // Use X0 and X7 for other purposes.
 .
ldp x0, x7, [sp]
add sp, sp, #16
```

While this clearly takes more instructions (and, therefore, takes longer to execute), it's possible to reserve the stack storage only once within a function and reuse that space throughout the execution of the function. You'll see examples of this in Chapter 5.

3.9.5 Preserving Register Values on the Stack

As you've seen in previous examples, the stack is a great place to temporarily preserve registers so they can be used for other purposes. Consider the following program outline:

```
Some instructions that use the X20 register.

Some instructions that need to use X20, for a
different purpose than the above instructions.

Some instructions that need the original value in X20.
```

The push and pop operations are perfect for this situation. By inserting a push sequence before the middle sequence, and a pop sequence after the middle sequence, you can preserve the value in X20 across those calculations:

```
Some instructions that use the X20 register.

     str x20, [sp, #-16]!

Some instructions that need to use X20, for a
different purpose than the above instructions.

     ldr x20, [sp], #16

Some instructions that need the original value in X20.
```

This push sequence copies the data computed in the first sequence of instructions onto the stack. Now the middle sequence of instructions can use X20 for any purpose it chooses. After the middle sequence of instructions finishes, the pop sequence restores the value in X20 so the last sequence of instructions can use the original value in X20.

3.9.6 Saving Function Return Addresses on the Stack

Throughout the example programs up to this point, I've preserved the return address appearing in the link register (LR) by using instructions like the following:

```
lea x0, saveLR
str lr, [x0]
 .
 .
 .
```

```
lea x0, saveLR
ldr lr, [x0]
ret
```

I've also mentioned that this is a *truly horrible* way of preserving the value in LR. It takes six instructions to accomplish (remember, `lea` expands into two instructions), making it slower and bulkier than it needs to be. This scheme also creates problems when you have one user-written function calling another: all of a sudden, you need two separate `saveLR` variables, one for each function. In the presence of recursion (see Chapter 5) or, worse, multithreaded code, this mechanism fails completely.

Fortunately, saving return addresses in the stack is the perfect solution. The stack's LIFO structure (see the next section) completely emulates the way (nested) function calls and returns work, and it takes only a single instruction to push LR onto the stack or pop LR off the stack. The earlier code sequence can be easily replaced by:

```
str lr, [sp, #-16]!
 .
 .
 .
ldr lr, [sp], #16
ret
```

Using the stack to save and restore the LR register is probably the most common use of the stack. Chapter 5 discusses managing return addresses and other function-related values in much greater depth.

3.10 Pushing and Popping Stack Data

You can push more than one value onto the stack without first popping previous values off the stack. However, the stack is a *last-in, first-out (LIFO)* data structure, so you must be careful in the way you push and pop multiple values.

For example, suppose you want to preserve X0 and X1 across a block of instructions. The following code demonstrates the obvious (but incorrect) way to handle this:

```
str  x0, [sp, #-16]!
str  x1, [sp, #-16]!
      Code that uses X0 and X1 goes here.
ldr  x0, [sp], #16
ldr  x1, [sp], #16
```

Unfortunately, this code will not work properly! Figures 3-15 through 3-18 show the problem, with each box in these figures representing 8 bytes (note the addresses). Because this code pushes X0 first and X1 second, the stack pointer is left pointing at X1's value on the stack.

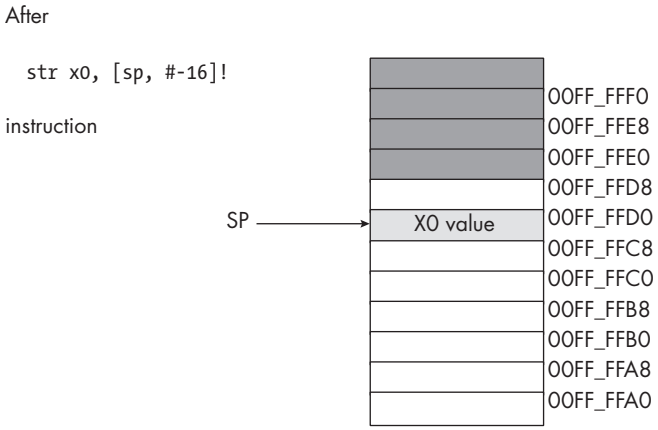

Figure 3-15: The stack after pushing X0

Figure 3-16 shows the stack after pushing the second register (X1).

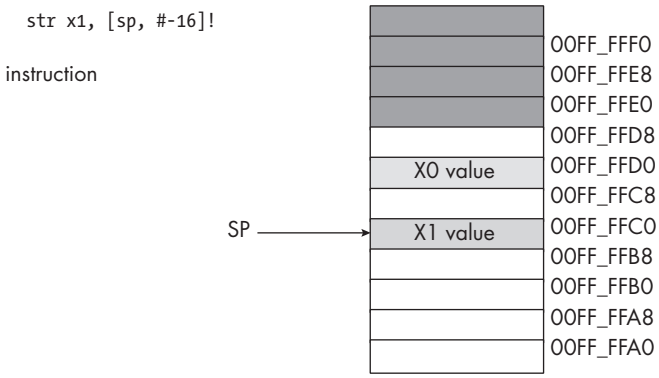

Figure 3-16: The stack after pushing X1

When the `ldr x0, [sp], #16` instruction comes along, it removes the value that was originally in X1 from the stack and places it in X0 (see Figure 3-17).

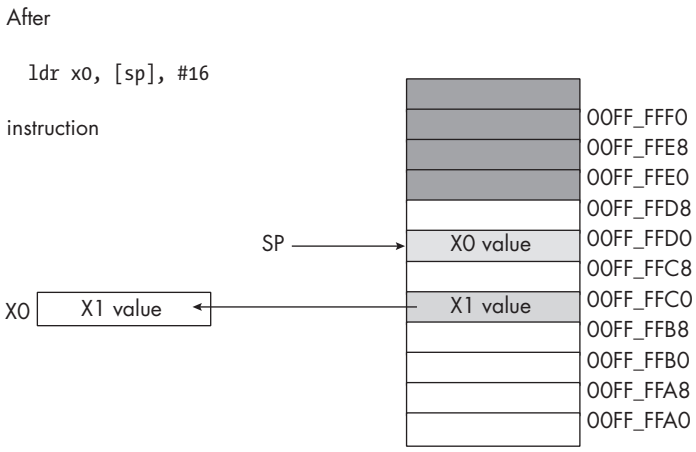

Figure 3-17: The stack after popping X0

Likewise, the `ldr x1, [sp], #16` instruction pops the value that was originally in X0 into the X1 register. In the end, this code manages to swap the values in the registers by popping them in the same order that it pushes them (see Figure 3-18).

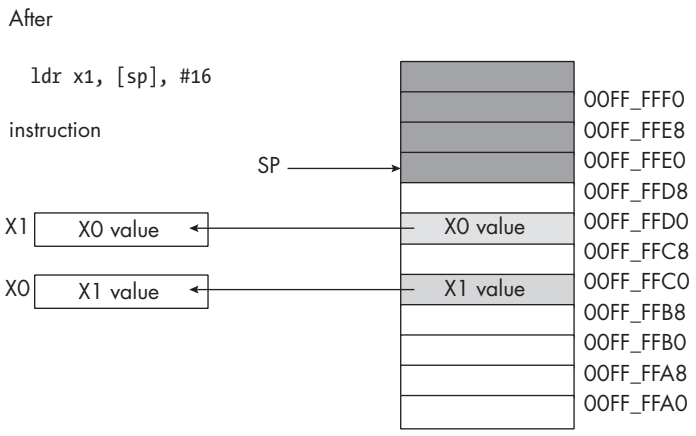

Figure 3-18: The stack after popping X1

To rectify this problem, because the stack is a LIFO data structure, the first thing you must pop is the last thing you push onto the stack. Therefore, *always pop values in the reverse order that you push them.*

The correction to the previous code is shown here:

```
str  x0, [sp, #-16]!
str  x1, [sp, #-16]!
   Code that uses X0 and X1 goes here.
ldr  x1, [sp], #16
ldr  x0, [sp], #16
```

Also remember to *always pop exactly the same number of bytes that you push*. In general, this means you'll need exactly the same the number of pushes and pops. If you have too few pops, you will leave data on the stack, which may confuse the running program. If you have too many pops, you will accidentally remove previously pushed data, often with disastrous results.

As a corollary, *be careful when pushing and popping data within a loop*. It's easy to put the pushes in a loop and leave the pops outside the loop (or vice versa), creating an inconsistent stack. Remember, it's the execution of the push and pop operations that matters, not the number of push and pop operations that appear in your program. At runtime, the number (and order) of the push operations the program executes must match the number (and reverse order) of the pop operations.

Finally, remember that *the ARM requires the stack to be aligned on a 16-byte boundary*. If you push and pop items on the stack (or use any other instructions that manipulate the stack), make sure that the stack is aligned on a 16-byte boundary before calling any functions or procedures that adhere to the ARM requirements.

3.11 Removing Data from the Stack Without Popping It

You may often discover that you've pushed data you no longer need onto the stack. Although you could pop the data into an unused register, there is an easier way to remove unwanted data from the stack: simply adjust the value in the SP register to skip over the unwanted data on the stack.

Consider the following dilemma (in pseudocode, not actual assembly language):

```
str x0, [sp, #-16]!  // Push X0.
str x1, [sp, #-16]!  // Push X1.

Some code that winds up computing some values we want
to keep in X0 and X1.

if( Calculation_was_performed ) then

      // Whoops, we don't want to pop X0 and X1!
      // What to do here?

else

      // No calculation, so restore X1, X0.

      ldr x1, [sp], #16
      ldr x0, [sp], #16

endif;
```

Within the `then` section of the `if` statement, this code wants to remove the old values of X0 and X1 without otherwise affecting any registers or memory locations. How can you do this?

Because the SP register contains the memory address of the item on the top of the stack, we can remove the item from the top by adding the size of that item to the SP register. In the preceding example, we wanted to remove two dword items from the top. We can easily accomplish this by adding 16 to the stack pointer:

```
str x0, [sp, #-16]!  // Push X0
str x1, [sp, #-16]!  // Push X1

Some code that winds up computing some values we want to keep
into rax and rbx.

if( Calculation_was_performed ) then

     // Remove unneeded X0/X1 values
     // from the stack.

     add sp, sp, #32

else

     // No calculation, so restore X1, X0.

     ldr x1, [sp], #16
     ldr x0, [sp], #16

endif;
```

Effectively, this code pops the data off the stack without moving it anywhere. This code is faster than two dummy pop operations, because it can remove any number of bytes from the stack with a single `add` instruction.

Remember to keep the stack aligned on a quad-word (16-byte) boundary. This means you should always add a constant that is a multiple of 16 to SP when removing data from the stack.

3.12 Accessing Data Pushed onto the Stack Without Popping It

Once in a while, you'll push data onto the stack and will want to get a copy of that data's value, or perhaps you'll want to change that data's value without actually popping the data off the stack (that is, you wish to pop the data off the stack at a later time). The ARM `[SP, #±offset]` addressing mode provides the mechanism for this.

Consider the stack after the execution of the following instruction:

```
stp x0, x1, [sp, #-16]!  // Push X0 and X1
```

This produces the stack result shown in Figure 3-19.

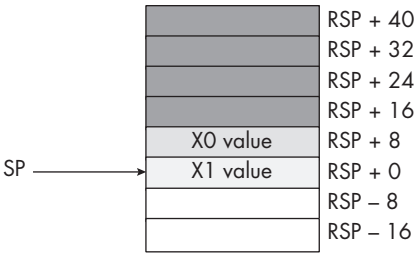

Figure 3-19: The stack after pushing X0 and X1

If you wanted to access the original X1 value without removing it from the stack, you could cheat by popping the value, then immediately pushing it again. Suppose, however, that you wish to access X0's old value or another value even farther up the stack. Popping all the intermediate values and then pushing them back onto the stack is problematic at best, impossible at worst.

However, as Figure 3-19 shows, each value pushed on the stack is at a certain offset from the SP register in memory. Therefore, we can use the `[SP, #±offset]` addressing mode to gain direct access to the value we are interested in. In the preceding example, you can reload X0 with its original value by using this single instruction:

```
ldr x0, [sp, #8]
```

This code copies the 8 bytes starting at memory address SP + 8 into the X0 register. This value just happens to be the previous value of X0 that was pushed onto the stack. You can use this same technique to access other data values you've pushed onto the stack.

Don't forget that the offsets of values from SP into the stack change every time you push or pop data. Abusing this feature can create code that is hard to modify; using this feature throughout your code will make it difficult to push and pop other data items between the point where you first push data onto the stack and the point where you decide to access that data again using the `[SP, #±offset]` memory addressing mode.

The previous section pointed out how to remove data from the stack by adding a constant to the SP register. That pseudocode example could probably be written more safely as this:

```
stp  x0, x1, [sp, #-16]!

Some code that winds up computing some values we want
to keep into X0 and X1.

if( Calculation_was_performed ) then

     // Overwrite saved values on the stack with
     // new X0/X1 values (so the pops that
     // follow won't change the values in X0/X1).
```

```
    stp  x0, x1, [sp, #8]

endif;
ldp  x0, x1, [sp], #16
```

In this code sequence, the calculated result was stored over the top of the values saved on the stack. Later, when the program pops the values, it loads these calculated values into X0 and X1.

THE "MAGIC" INSTRUCTIONS

In most of the example programs in this book so far, the following lines of code have appeared in `asmMain` (and in other functions):

```
// "Magic" instruction offered without
// explanation at this point:

sub     sp, sp, #256
 .
 .
 .
add     sp, sp, #256
```

At this point, it should be clearer what this code is doing: reserving storage on the stack (and removing that storage before returning from the function).

Chapter 5 covers this scheme in greater detail when it discusses local variables and parameter functions. For the time being, just know that the purpose of these statements is to reserve storage on the stack for parameters being passed to the `printf()` function via the `vparm`*n* macros.

3.13 Moving On

This chapter discussed memory organization and access, and how to create and access memory variables on the ARM CPU. It went over problems that can occur when accessing data beyond the end of a data structure that crosses over into a new MMU page, then discussed little- and big-endian memory organizations and how to use the ARM memory addressing modes and address expressions to access those memory objects in multiple ways. You learned how to align data in memory to improve performance, how to obtain the address of a memory object, and the purpose of the ARM stack structure.

Thus far, this book has generally employed only basic data types such as different-sized integers, characters, Boolean objects, and floating-point numbers. Fancier data types, such as pointers, arrays, strings, and structs are the subject of the next chapter.

3.14 For More Information

See *https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_toc.html* for details on the GNU assembler.

Learn more about the GNU linker at *https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html*.

For more about the macOS (LLVM) linker, see *https://lld.llvm.org*.

Visit the ARM developer website at *https://developer.arm.com* for more on ARM CPUs.

Wikipedia offers an explanation of address space layout randomization at *https://en.wikipedia.org/wiki/Address_space_layout_randomization*.

To better understand position-independent executables, see *https://en.wikipedia.org/wiki/Position-independent_code*.

For information on the `:lo12:` operator, see the "Assemb.ly Expressions" section in the document downloadable from *https://developer.arm.com/documentation/100067/0612/armclang-Integrated-Assembler*.

TEST YOURSELF

1. The PC-relative addressing mode indexes off which 64-bit register?
2. What does *opcode* stand for?
3. What type of data is the PC-relative addressing mode typically used for?
4. What is the address range of the PC-relative addressing mode?
5. In a register-indirect addressing mode, what does the register contain?
6. Which of the following registers is valid for use with the register-indirect addressing mode?
   a. W0
   b. X0
   c. XZR
   d. SP
7. What instruction would you normally use to load the address of a memory object into a register?
8. What is an effective address?
9. How would you align a variable in the `.data` section to an 8-byte boundary?
10. What does *MMU* stand for?
11. What is an address expression?
12. What is the difference between a big-endian value and a little-endian value?

13. If W0 contains a 32-bit big-endian value, what instruction could you use to convert it to a little-endian value?
14. If W0 contains a 16-bit little-endian value, what instruction could you use to convert it to a big-endian value?
15. If X0 contains a 64-bit big-endian value, what instruction could you use to convert it to a little-endian value?
16. Explain, step-by-step, what the `str X0, [sp, #-16]!` instruction does.
17. Explain, step-by-step, what the `ldr X0, [sp], #16` instruction does.
18. When using the push and pop operations to preserve registers, you must always pop the registers in the _____ order that you pushed them.
19. What does *LIFO* stand for?