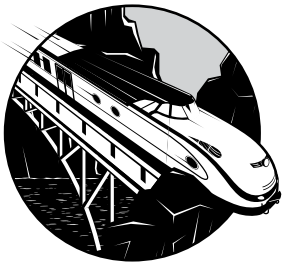
C

ARM ASSEMBLY FOR THE HLL PROGRAMMER



A basic understanding of ARM assembly language will enable you to read the ARM output produced by compilers on machines like most mobile phones or tablets, the Raspberry Pi, and even some higher-end Arduino-class single-board computers (such as the Teensy 3.6). Thus, this appendix provides an overview of:

- The basic ARM machine architecture
- ARM assembly language (so that you'll be able to read the ARM output produced by the GCC compiler)
- The memory addressing modes of the ARM CPU
- The syntax used by the ARM Gas assembler
- How to use constants and declare data in assembly language programs

In addition, the resources at *www.writegreatcode.com* describe a minimal ARM instruction set that you'll need when examining compiler output.

C.1 Assembly Syntaxes

As Chapter 3 explained, there are significant syntax differences in the code generated by various assemblers for the 80x86. ARM assemblers, by contrast, use a much more uniform syntax, so this book uses the syntax employed by the Gnu assembler (Gas) as provided on the Raspberry Pi. You should have no trouble reading ARM assembly listings produced for other assemblers if you learn the Gas syntax used in this book.

C.2 Basic ARM Architecture

The ARM, Inc.,[1] ARM CPU family is classified as a *Von Neumann machine*. Von Neumann computer systems contain three main building blocks: the *central processing unit (CPU)*, *memory*, and *input/output (I/O) devices*. These three components are connected together via the *system bus* (consisting of the address, data, and control buses). Figure 3-1 showed this relationship.

The CPU communicates with memory and I/O devices by placing a numeric value on the *address bus* to select one of the memory locations or I/O device port locations, each of which has a unique binary numeric address. Then the CPU, I/O, and memory devices pass data among themselves by placing the data on the *data bus*. The *control bus* provides signals that determine the direction of the data transfer (to/from memory and to/from an I/O device).

The registers are the most prominent feature within the CPU. The ARM CPU registers are categorized as general-purpose integer registers, floating-point/SIMD (single-instruction, multiple data) registers, special-purpose application-accessible registers, and special-purpose kernel-mode registers. Special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools. That topic is well beyond the scope of this book, so they will not be discussed further.

C.2.1 32- and 64-Bit Variants

The ARM architecture describes a family of CPUs. The original stand-alone ARM processor (ARMv2) was a relatively simple 32-bit processor with a 26-bit address space. As time passed, different variants emerged, and the ARMv7 architecture (with a 32-bit physical address bus) became the predominant architecture in smartphones throughout the 2000s. The ARMv8 64-bit variant (with a 48-bit physical address bus) appeared in 2011 and quickly replaced the ARMv7 in smartphones and tablets as well as in later Raspberry Pi models.

1. ARM used to be ARM Holdings, Inc. Before that, it was Advanced RISC Machines, and originally it was Acorn RISC Machines. Today, the company simply goes by ARM, Inc.

Although the 32-bit ARMv7 and earlier variants (we'll call these the "A32 CPUs") are still used in microcontrollers and embedded devices, most modern ARM computing devices use ARMv8 or later versions (the "A64 CPUs"). Alas, the Raspberry Pi family contains a mixture of ARMv7 and ARMv8 processors. Therefore, the Raspberry Pi Foundation (the outfit that created and sells the Raspberry Pi) currently provides only 32-bit support in their version of Linux (Raspbian), even on 64-bit CPUs. Suse Linux (by Suse, LLC) and OpenSUSE, however, do provide 64-bit variants of Linux running on Raspberry Pi computers. This book typically presents A32 examples running on the standard Raspbian releases for the Raspberry Pi. Where 64-bit examples are appropriate, though, it presents A64 examples running under Apple's iOS and OpenSuse (on the Raspberry Pi).

C.2.2 General-Purpose Integer Registers

C.2.2.1 ARMv7 (A32) Registers

The 32-bit variants of the ARM provide 13 general-purpose 32-bit registers (R0 through R12). There are three additional 32-bit registers with special names: SP (stack pointer), LR (link register), and PC (program counter). Note that R11 is commonly used as a frame-pointer register in compiler-generated code on A32, and you'll often see FP equated to R11 in source code.

C.2.2.2 ARMv8 (A64) Registers

The A64 CPUs provide 31 general-purpose 64-bit registers for application use. Most compilers refer to these registers as X0 through X30. There is an additional register, corresponding to X31, that holds the stack pointer value (which most programs refer to as SP). The stack pointer serves double-duty: for stack-based instructions, the SP register behaves like a normal stack pointer register on other CPUs; for non-stack-related operations, reading this register (named XZR) returns the value `0`, while data written to it is ignored.

The A64 also supports 32-bit operations on the register set using the register names W0 through W30. The 32-bit registers comprise the LO 32-bits of the 64-bit register set.

The A64 PC register is not mapped to a general-purpose register. Therefore, you cannot use the `mov` instruction to create an indirect jump by moving some other register into the PC. There is, however, a new branch instruction, `br`, on the 64-bit ARM that accepts a general-purpose register as the operand; this instruction copies the register operand's value into the PC register.

X30 on the A64 is the link register (LR) on the A64. Executing the branch and link (`bl`) instruction copies the address of the instruction following the `bl` into the X30 register.

C.2.3 Floating-Point and SIMD Registers

Floating-point support is optional on ARM processors. That being said, almost any A64 CPU you encounter will likely provide full floating-point support. However, A32 CPUs on various devices may have differing FPU capabilities. It's always important, then, to determine exactly which A32 you're using before assuming any particular floating-point capabilities. For example, the ARM Cortex M4 CPU on the Teensy 3.6 device supports only single-precision arithmetic, whereas the ARM Cortex A7 in the Raspberry Pi 2 Model B supports single-precision, double-precision, and vector operations.[2] This book assumes that an A32 supports the ARM VFPv4-D32 floating-point extensions. Again, if you're using a specific device, be sure to check its architecture manual to determine its floating-point capabilities.

C.2.3.1 A32 Floating-Point Registers

An A32 CPU with VFPv4 support provides the following floating-point registers:

- 32 single-precision (32-bit) floating-point registers, named S0 through S31
- 16 or 32 double-precision (64-bit) floating-point registers, named D0 through D15 or D0 through D31 (the number of registers depends on the floating-point architecture: VFPv4-D16 or VFPv4-D32)
- One floating-point status and control register (FPSCR; see Figure C-1)

Note that the S0 through S31 and D0 through D15/D31 registers are not necessarily independent of one another. On some ARM architectures the single- and double-precision floating-point registers share the same bits inside the FPU. See your particular ARM manual for details. On the VFPv4-D32 (with NEON extensions), the S0 through S31 single-precision floating-point registers overlap the first 16 double-precision registers. The 32 double-precision registers (D0 through D31) overlap the 16 quad-precision (128-bit) registers (Q0 through Q15). See Figure C-2 for details.

2. The Raspberry Pi 3 uses an ARM Cortex A53 CPU (A64), which provides a full-featured CPU. Most Apple CPUs (iPhone, iPad, and the like) provide outstanding FPU capabilities, as do most modern Android phones, tablets, and other systems.

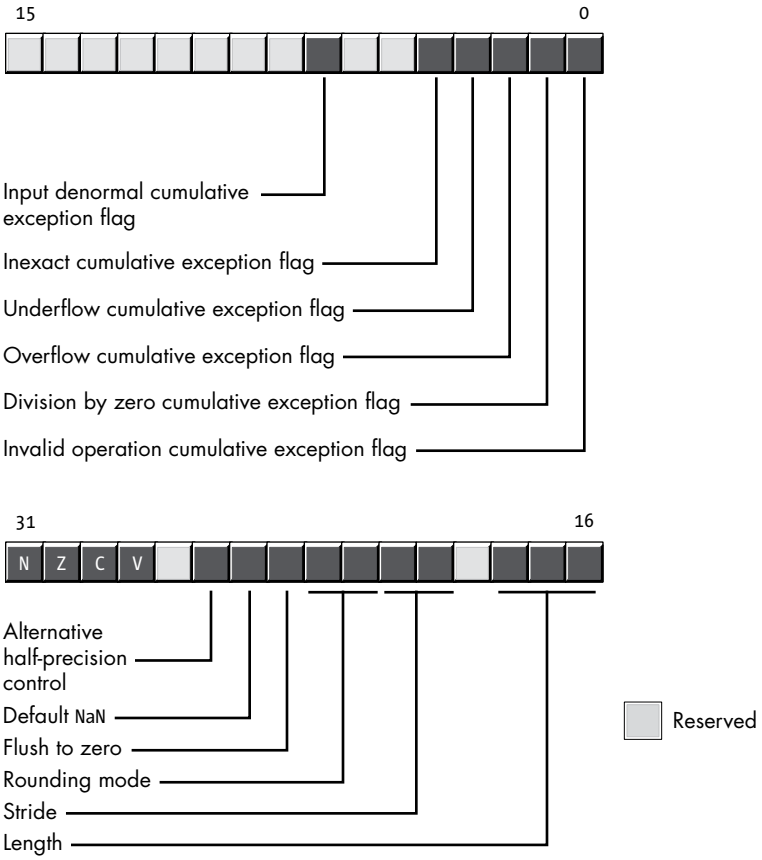


Figure C-1: ARM FPSCR register

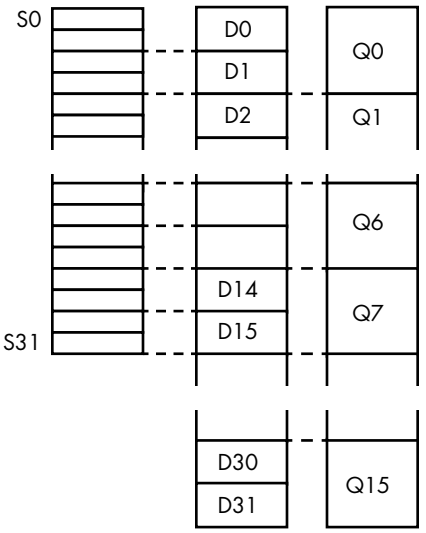


Figure C-2: A32/VFPv4 floating-point registers

C.2.3.2 A64 Floating-Point Registers

The A64 CPUs provide the same set of floating-point registers as A32 (single-, double-, and quad-precision). However, there are a couple of differences. First, the A64 FPU implementation provides 32 quad-precision registers (Q0 through Q31). Second, although the registers still overlap, S0 through S31 comprise the LO 32 bits of D0 through D31 (rather than packing S0 through S31 into D0 through D15). Likewise, D0 through D31 comprise the LO 64 bits of Q0 through Q31 (see Figure C-3).

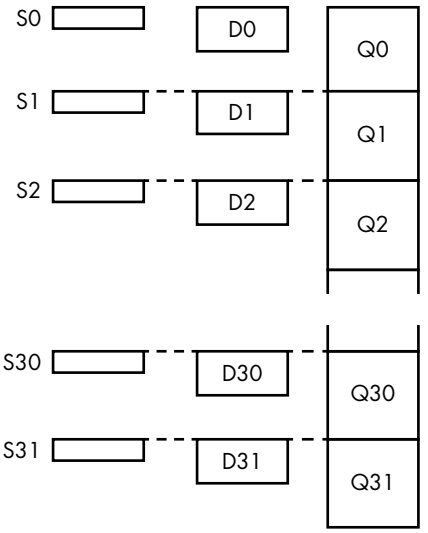


Figure C-3: A64 floating-point registers

C.2.3.3 Condition Code Bits

The ARM CPUs provide four separate condition code bits: `N` (sign), `Z` (zero), `C` (carry), and `V` (overflow). From a historical perspective, these are the same condition codes found on the MOS Technology 6502 processor—the CPU used in the Acorn machine (Acorn Computers invented the original ARM processor). The condition code bits on the ARM provide the typical signed, unsigned, and floating-point equal, not equal, greater than, greater than or equal, less than, less than or equal, negative, positive (or zero), and overflow (signed and unsigned) tests.

Most ARM instructions don't normally affect the condition code bits. Instead, there are special instructions (mostly with an `S` suffix) that explicitly set the condition codes. Typically, these are arithmetic and logical instructions.

The ARM processors maintain the condition codes in a special *Application Program Status Register (APSR)*. This register contains the `N`, `V`, `C`, and `Z` condition code bits along with a saturation (`Q`) status bit indicating whether arithmetic saturation occurs.

The floating-point compare instruction (`fcmp`) updates a separate set of condition code flags in the FPSCR. You must copy these flags into their corresponding flags in the APSR using the `fmstat` instruction prior to testing them.

C.2.3.4 The Link Register

The ARM LR (link register; X30 on A64, and R14 on A32) holds a *return address* after the execution of a branch and link instruction. Executing the `bl` instruction leaves the address of the instruction following the branch in the LR. ARM applications use this register to implement returns from subroutine operations as well as to compute program-counter-relative addresses for various operations.

C.3 Literal Constants

Like most assemblers, Gas supports literal numeric, character, and string constants. This section describes their syntax.

C.3.1 Binary Literal Constants

Binary literal constants in Gas begin with the special `0b` prefix followed by one or more binary digits (`0` or `1`). Examples:

```
0b1011
0b10101111
0b0011111100011001
0b1011001010010101
```

C.3.2 Decimal Literal Constants

Decimal literal constants in Gas take the standard form—a sequence of one or more decimal digits without any special prefix or suffix. Examples:

```
123
1209345
```

C.3.3 Hexadecimal Literal Constants

Hexadecimal literal constants in Gas consist of a string of hexadecimal digits (`0..9`, `a..f`, or `A..F`) with a `0x` prefix. Examples:

```
0x1AB0
0x1234ABCD
0xdead
```

C.3.4 Character and String Literal Constants

Character literal constants in Gas consist of an apostrophe followed by a single character. Examples:

```
'a
''
'!
```

String literal constants in Gas consist of a sequence of zero or more characters surrounded by quotes. They use the same syntax as C strings. You use the \ escape sequence to embed special characters in a Gas string. Examples:

```
"Hello World"
"" -- The empty string
"He said \"Hello\" to them"
"\"" -- string containing a single quote character
```

C.3.5 Floating-Point Literal Constants

Floating-point literal constants in assembly language typically take the same form you'll find in HLLs—a sequence of digits, possibly containing a decimal point, optionally followed by a signed exponent. Examples:

```
3.14159
2.71e+2
1.0e-5
5e1
```

C.4 Manifest (Symbolic) Constants in Assembly Language

Almost every assembler provides a mechanism for declaring symbolic (named) constants. Gas uses the `.equ` ("equate") statement to define a symbolic constant in the source file. This statement uses the following syntax:

```
.equ        symbolName, value
```

Here are some examples within a Gas source file:

```
.equ        false, 0
.equ        true, 1
.equ        anIntConst, 12345
```

C.5 ARM Addressing Modes

ARM instructions can access three types of operands: register operands, immediate constants, and memory operands.

C.5.1 ARM Register Access

Gas allows assembly programmers and compiler writers to access the ARM general-purpose integer registers by name:

- A32: R0; R1 through R12; LR; SP; and PC
- A64: X0; X1 through X30; W0; W1 through W30; LR; SP; and PC

Floating-point instructions access the floating-point registers by their name (S0 through S31 or D0 through D31). Note that floating-point registers are legal only as floating-point instruction operands (just as integer instructions are accessible only within integer instructions).

C.5.2 The Immediate Addressing Mode

Many integer instructions allow a programmer to specify an immediate constant as a source operand. However, as all ARM instructions are exactly 32 bits in size, a single instruction cannot load a 32-bit (or larger) constant into an ARM register. The ARM's instruction set does support immediate constants that are 8 bits in size (or smaller), possibly rotated in a 32-bit word by an even multiple of bits (0, 2, 4, 6, 8, . . . , 30 bits). The ARM CPU uses an additional 4 bits to encode these 16 different rotate positions (see Table C-1).

Table C-1: Immediate Constant Encoding for 8-Bit Binary Values

Rotate encoding	Value (in binary)
0	00000000000000000000000ABCDEFGH
1	GH0000000000000000000000ABCDEF
2	EFGH0000000000000000000000ABCD
3	CDEFGH0000000000000000000000AB
4	ABCDEFGH0000000000000000000000
5	00ABCDEFGH0000000000000000000000
6	0000ABCDEFGH0000000000000000000000
7	000000ABCDEFGH000000000000000000
8	00000000ABCDEFGH0000000000000000
9	0000000000ABCDEFGH00000000000000
10	000000000000ABCDEFGH000000000000
11	00000000000000ABCDEFGH0000000000
12	0000000000000000ABCDEFGH00000000
13	000000000000000000ABCDEFGH000000
14	00000000000000000000ABCDEFGH0000
15	0000000000000000000000ABCDEFGH00

Although it's impossible to encode all possible 32-bit values with only 12 bits, the 12-bit encoding does cover a wide variety of useful constants. In particular, you can create a constant that has a single 1 bit set in each of the 32 different bit positions (that is, by setting the G and H bits in Table C-1 or 01 or 10 and employing all 16 different rotate encodings). One drawback to this scheme is that you can't create 4,096 different constant encodings because it's possible for certain (different) encodings to produce the same constant. For example, you can produce the value 4 by encoding (4 rotate 0) and (1 rotate 15).[3]

3. (4 rotate 0) has a 1 bit in the F position, while (1 rotate 15) has a 1 bit in the H position. All other bits are 0.

For immediate values that the ARM cannot encode into the instruction, the ARM requires that you load the constant into a register from a memory location.[4] The most obvious downside to this is that the code is larger and slower, but another problem is that you must dedicate a precious register to hold the immediate value.

C.5.3 ARM Memory Addressing Modes

The ARM CPU is a *load/store architecture*, meaning that it can only access (data) memory using load and store instructions. All other instructions operate on registers (or small immediate constants). With a load/store architecture, for example, you cannot directly add the contents of some memory location to a register value—you must first load the memory data into a register and then add that register to the destination register's value.

Arguably, the ARM CPU supports a single addressing mode: *auto-increment/decrement scaled indexed with offset plus write-back*. Technically, though, there are 4 bits of encoding within the load/store instruction to control this "single" addressing mode, so in reality the ARM is very un-RISC-like and supports 16 memory addressing modes (in the most complex case). However, not all variations of the ARM load and store instructions support all 16 addressing modes.

C.5.3.1 Register Plus Displacement Addressing Mode

The ARM register plus displacement addressing mode adds a signed 12-bit displacement value, sign-extended to 32 bits, with the value from a general-purpose integer register to compute the effective memory address. The Gas syntax for this addressing mode is as follows:

```
[Rn, #displacementValue] // A32
[Xn, #displacementValue] // A64
```

where *displacementValue* is a signed 8-bit expression and *Rn* represents one of the A32's 32-bit general-purpose registers (R0 through R12); *Xn* represents one of the A64's 64-bit general-purpose registers (X0 through X30).

The `ldr` (load register) instruction is a typical load instruction that allows the register plus displacement addressing mode. It fetches a 32-bit word from memory and then copies the result into a destination register. For example, this particular instruction loads R3 with the (32-bit) word found in memory at the address held in R5 plus 4:

```
ldr R3, [R5, #4] // A32 instruction
```

4. You could also logically OR multiple immediate constants together using two to four different instructions. This might be faster than loading a constant from memory if memory access is sufficiently slow. Keep in mind, however, that those instructions each consume 4 bytes of memory, which the CPU must fetch from memory to execute (though they'll likely be in the instruction cache/pipeline).

This particular instruction loads W3 with the (32-bit) word found in memory at the address held in X5 plus 4:

```
ldr W3, [X5, #4] // A64 instruction
```

C.5.3.2 Preindexed Addressing with Write-Back

Most load and store instructions (like `lbr`) on the ARM support a special *update* form. When you're using the register plus displacement addressing mode, these instructions work just like the standard load instructions except that they update the base address register with the final effective address. Such instructions specify the `!` (exclamation point) after the addressing mode. For example, this instruction not only copies the value from memory location [R5 + 4][5] into R3, but also adds 4 to R5:

```
ldr R3, [R5, #4]! // A32 instruction
```

This instruction copies the value from memory location [X5 + 4] into W3 and adds 4 to X5:

```
ldr W3, [X5, #4]! // A64 instruction
```

C.5.3.3 Post-Indexed Addressing with Write-Back

Post-indexed addressing is another update form. When you're using the register plus displacement addressing mode, these instructions work just like the standard load instructions except that they update the base address register with the final effective address after fetching the value from memory. For example, this instruction not only copies the value from memory location [R5] into R3, but also adds 4 to R5 (after copying the value from location [R5] into R3).

```
ldr R3, [R5], #4 // A32 instruction
```

This instruction copies the value from memory location [X5] into W3 and adds 4 to X5 (after copying the value from location [X5] into W3).

```
ldr W3, [X5], #4 // A64 instruction
```

C.5.3.4 Scaled-Index Addressing Mode

The ARM also supports a scaled-index addressing mode, which uses one general-purpose register to hold a base address and a second general-purpose register to hold an index from that base address. In addition, a small immediate constant specifies a left shift factor for the index register

5. The brackets ([]) denote indirection. That is, [R5 + 4] represents the memory at the address specified by the contents of R5 plus 4.

(a maximum of 31 bits). With Gas, you specify the shift factor using an operand of the form `asl #n`, where *n* is the number of bits to shift the index register. The `ldr` instruction, for example, uses the following syntax:

```
ldr Rd, [Rb, Rx, asl #n] // A32 syntax
ldr Wd, [Xb, Xx, asl #n] // A64 syntax (32-bit destination)
ldr Xd, [Xb, Xx, asl #n] // A64 syntax (64-bit destination)
```

Rd, *Wd*, or *Xd* is the destination register, *Rb* or *Xb* is the base register, *Rx* or *Xx* is the index register, and *#n* is the scaling factor.

Examples:

```
ldr R3, [R5, R6, asl #2] // A32 instruction
ldr W3, [X5, X6, asl #2] // A64 32-bit instruction
ldr X3, [X5, X6, asl #2] // A64 64-bit instruction
```

The A32 example loads R3 with the 32-bit word found at the memory address [R5 + R6 × 4]. The A64 examples load W3 or X3 with the 32-bit word or 64-bit double word found at the memory address [X5 + X6 × 4].

There are also pre- and post-update forms of the indexed addressing mode, which update the base register with the sum of the base and index registers after computing the effective memory address. The index register's value is unaffected by the update form of the instruction. Here's the A32 syntax for the pre- and post-update forms:

```
ldr Rd, [Rb, Rx, asl #n]! // Pre-update form
ldr Rd, [Rb], Rx, asl #n  // Post-update form
```

The pre-update form computes *Rb* = *Rb* + *Rx* × 2*n*, stores the result into *Rb*, and then fetches the memory location at the new value of *Rb*. The post-update form fetches the value from the location pointed at by *Rb* and then computes *Rb* = *Rb* + *Rx* × 2*n*.

The A64 syntax for the pre- and post-update forms is:

```
ldr Xd, [Xb, Xx, asl #n]! // Pre-update form (64-bit operation)
ldr Wd, [Xb], Xx, asl #n  // Post-update form (32-bit operation)
```

The pre-update form computes *Xb* = *Xb* + *Xx* × 2*n*, stores the result into *Xb*, and then fetches the memory location at the new value of *Xb*. The post-update form fetches the value from the location pointed at by *Xb* and then computes *Xb* = *Xb* + *Xx* × 2*n*.

C.6 Declaring Data in Assembly Language

The ARM CPU provides only a few low-level machine data types on which individual machine instructions can operate:

- Bytes that hold arbitrary 8-bit values
- Words that hold arbitrary 16-bit values (*halfwords* in ARM terminology)

- Double words that hold arbitrary 32-bit values (*words* in ARM terminology)
- Quad words that hold 64-bit values (*double words* in ARM terminology)
- Single-precision floating-point values (32-bit single floating-point values)
- Double-precision, 64-bit, floating-point values

NOTE *Although the standard ARM terminology is* byte, halfword, word, *and* double word *for 8-, 16-, 32-, and 64-bit integer values, outside of this appendix this book uses the x86 terminology to avoid confusion with the 80x86 code.*

Although the ARM supports 128-bit quad-word floating-point values, Gas does not provide a directive to encode 128-bit floating-point constants. You'll have to convert these values to their equivalent bit patterns and emit them using other directives.

Gas uses the `.byte` directive in a `.data` section to declare a byte variable, like so:

```
variableName: .byte 0
```

Gas doesn't provide an explicit form for creating uninitialized variables, so you just supply a `0` operand for them. Here is an actual byte variable declaration in Gas:

```
IntializedByte: .byte   5
```

Gas also does not provide an explicit directive for declaring an array of byte objects, but you can use the `.rept`/`.endr` directives to create multiple copies of the `.byte` directive as follows:

```
variableName:
        .rept   sizeOfBlock
        .byte   0
        .endr
```

You can also supply a comma-delimited list of values to initialize the array with different values.

Here are a couple of array declaration examples in Gas:

```
            .section    .data ; Variables go in this section
InitializedArray0:            ; Creates an array with elements 5,5,5,5
            .rept       4
            .byte       5
            .endr

InitializedArray1:
            .byte       0,1,2,3,4,5
```

For 16-bit objects, Gas uses the `.short` directive. Other than the size of the object these directives declare, their use is identical to the byte declarations:

```
                    .section    .data
GasWordVar:         .short 0

; Create an array of four words, all initialized to 0:

GasWordArray:
                    .rept   4
                    .short 0
                    .endr

; Create an array of 16-bit words, initialized with
; the values 0, 1, 2, 3, and 4:

GasWordArray2:      .short 0,1,2,3,4
```

For 32-bit objects, Gas uses the `.long` or `.word` directive:

```
                    .section    .data
GasDWordVar:        .long   0

; Create an array with four double-word values
; initialized to 0:

GasDWordArray:
                    .rept   4
                    .long   0
                    .endr

; Create an array of double words initialized with
; the values 0, 1, 2, 3, 4:

GasDWordArray2:     .long   0,1,2,3,4
```

For 64-bit objects, Gas uses the `.quad` or `.xword` directive:

```
                    .section    .data
GasQWordVar:        .xword 0

; Create an array with four xword values
; initialized to 0:

GasQWordArray:
                    .rept   4
                    .xword 0
                    .endr

; Create an array of xwords initialized with
; the values 0, 1, 2, 3, 4:

GasQWordArray2:     .xword 0,1,2,3,4
```

For floating-point values, Gas uses the `.single` and `.double` directives to reserve storage for an IEEE-format floating-point value (32 or 64 bits, respectively). Because the ARM CPU does not support arbitrary immediate floating-point constants, if you need to reference a floating-point constant from a machine instruction, you'll need to place that constant in a memory variable and access that memory variable instead. Here are some examples:

```
                    .section    .data
GasSingleVar:       .single  0.0
GasDoubleVar:       .double  1.0

; Create an array with four single-precision values
; initialized to 2.0:

GasSingleArray:
                    .rept   4
                    .single 2.0
                    .endr

; Create an array of double-precision values initialized with
; the values 0.0, 1.1, 2.2, 3.3, and 4.4:

GasDWordArray2:     .double 0.0,1.1,2.2,3.3,4.4
```

Note that GCC and other compilers often use the `.word` directive to emit floating-point constants. The integer operands that GCC emits will have the same bit patterns as the floating-point constants you specify in the C/C++ (or other HLL) code.

C.7 Specifying Operand Sizes in Assembly Language

ARM instructions generally operate only on 32-bit or 64-bit data. Unlike CISC processors, individual ARM instructions don't operate on various data types. The `add` instruction, for example, operates only on 32-bit values (except on 64-bit implementations of the ARM, where it operates on 64-bit values when in 64-bit mode). Generally, this isn't a problem. If two ARM registers contain 8-bit values, you'll get the same result by adding those two 32-bit registers together that you'd get if they were 8-bit registers, if you consider only the LO 8 bits of the sum.

Memory accesses, however, are a different matter. When reading and (especially) writing data in memory, it's important that the CPU access only the desired data size. Therefore, the ARM provides some size-specific load and store instructions that specify byte, 16-bit halfword, 32-bit word, and (for the A64) 64-bit double word sizes.

C.8 The Minimal Instruction Set

Although the ARM CPU family supports hundreds of instructions, few compilers actually use all of them. This is because many instructions have

become obsolete over time as newer instructions have emerged. Some instructions, such as ARM's NEON instructions, simply don't correspond to functions you'd normally perform in an HLL. As a result, compilers rarely generate these types of machine instructions, which generally appear only in handwritten assembly language programs. Fortunately, this means you don't need to learn the entire ARM instruction set in order to study compiler output, but only the handful that compilers actually emit.

This section describes the A32 32-bit instruction set. The 64-bit instruction set is very similar; the main difference is the absence of condition instruction execution (except for branches), and support for 64-bit operands and a larger register file (32 general-purpose registers rather than 16). If all you need to do is read an assembly language listing rather than write an assembly language program, the relevant differences are mainly in the register sets for the two machines. While there are a few additional instructions on the 64-bit variants (and a few 32-bit instructions that have disappeared in 64-bit mode), if you can read 32-bit ARM assembly you'll largely be able to read 64-bit ARM assembly.

C.8.1 Data Manipulation Instructions

There are 16 ARM data manipulation instructions:

- `AND` (logical AND)
- `EOR` (exclusive-or)
- `SUB` (subtraction)
- `RSB` (reverse subtraction)
- `ADD` (addition)
- `ADC` (add with carry)
- `SBC` (subtract with carry)
- `RSC` (reverse subtract with carry)
- `TST` (test bits using logical AND)
- `TEQ` (test bits using logical exclusive-OR)
- `CMP` (compare)
- `CMN` (compare negated)
- `ORR` (logical OR)
- `MOV` (copy one operand to another)
- `BIC` (bit clear)
- `MVN` (move, negated/inverted)

Most of these instructions have three or four operands. The generic instruction forms are as follows:

`instr Rd, Rs1, Rs2`

Computes *`Rd = Rs1 op Rs2`*, where *`op`* corresponds to the operation performed by *`instr`*.

instr Rd, *Rs1*, #imm_8

Computes *Rd* = *Rs1 op* imm_8, where *op* corresponds to the operation performed by *instr*. The imm_8 operand is a 32-bit constant that can be encoded using the ARM 12-bit rotated immediate format (see "The Immediate Addressing Mode" on page 9).

instr Rd, *Rs1*, *Rs2*, **ASR** #*n*

Computes *Rd* = *Rs1 op* (*Rs2* ASR *n*), where *op* corresponds to the operation performed by *instr*. ASR is the arithmetic shift right operation (*Rs2*'s value is shifted to the right *n* bits via an ASR operation).

instr Rd, *Rs1*, *Rs2*, **ASR** *Rs3*

Computes *Rd* = *Rs1 op* (*Rs2* ASR *Rs3*), where *op* corresponds to the operation performed by *instr*. ASR is the arithmetic shift right operation (*Rs2*'s value is shifted to the right the number of bit positions specified by *Rs3* via an ASR operation).

instr Rd, *Rs1*, *Rs2*, **LSL** #*n*

Computes *Rd* = *Rs1 op* (*Rs2* LSL *n*), where *op* corresponds to the operation performed by *instr*. LSL is the logical shift left operation (*Rs2*'s value is shifted to the left *n* bits via an LSL operation).

instr Rd, *Rs1*, *Rs2*, **LSL** *Rs3*

Computes *Rd* = *Rs1 op* (*Rs2* LSL *Rs3*), where *op* corresponds to the operation performed by *instr*. LSL is the logical shift left operation (*Rs2*'s value is shifted to the left the number of bit positions specified by *Rs3* via an LSL operation).

instr Rd, *Rs1*, *Rs2*, **LSR** #*n*

Computes *Rd* = *Rs1 op* (*Rs2* LSR *n*), where *op* corresponds to the operation performed by *instr*. LSR is the logical shift right operation (*Rs2*'s value is shifted to the right *n* bits via an LSR operation).

instr Rd, *Rs1*, *Rs2*, **LSR** *Rs3*

Computes *Rd* = *Rs1 op* (*Rs2* LSR *Rs3*), where *op* corresponds to the operation performed by *instr*. LSR is the logical shift right operation (*Rs2*'s value is shifted to the right the number of bit positions specified by *Rs3* via an LSR operation).

instr Rd, *Rs1*, *Rs2*, **ROR** #*n*

Computes *Rd* = *Rs1 op* (*Rs2* ROR *n*), where *op* corresponds to the operation performed by *instr*. ROR is the logical rotate right operation (*Rs2*'s value is rotated to the right *n* bits via an ROR operation).

instr Rd, *Rs1*, *Rs2*, **ROR** *Rs3*

Computes *Rd* = *Rs1 op* (*Rs2* ROR *Rs3*), where *op* corresponds to the operation performed by *instr*. ROR is the logical rotate right operation (*Rs2*'s value is rotated to the right the number of bit positions specified by *Rs3* via an ROR operation).

instr Rd, *Rs1*, *Rs2*, **RRX #1**

Computes *Rd* = *Rs1 op* (*Rs2* RRX 1), where *op* corresponds to the operation performed by *instr*. RRX is the extended logical rotate right operation through carry (*Rs2*'s value is rotated to the right 1 bit with bit 0 going into the carry flag and the previous contents of the carry flag going into bit 31). Note that the RRX operand must always be the immediate constant #1.

In the preceding examples, the instruction computes some value based on the value of *Rs1* (the first operand), and the value of *Rs2* and the optional shift operand (the *immediate value*), which together constitute the second operand. The instruction typically stores the result into the destination operand. The fourth operand is simply a (shift) modifier to the third operand, forming a single value.

To demonstrate, Table C-2 provides explicit examples of all the operand forms for the ARM adc (add with carry) instruction.

Table C-2: Gas Syntax for 32-bit adc

Instruction	Description
adc *Rd*, *Rs1*, *Rs2* adcs *Rd*, *Rs1*, *Rs2*	*Rd* := *Rs1* + *Rs2* + carry *d*, *s1*, and *s2* are register numbers in the range 0..15. The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, #imm_8 adcs *Rd*, *Rs1*, #imm_8	*Rd* := *Rs1* + imm_8 + carry *d* and *s1* are register numbers in the range 0..15. The imm_8 operand is a 32-bit constant that can be encoded using the ARM 12-bit rotated immediate format (see "The Immediate Addressing Mode" on page 9). The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, *Rs2*, ASR *#n* adcs *Rd*, *Rs1*, *Rs2*, ASR *#n*	*Rd* := *Rs1* + (*Rs2* ASR *n*) + carry *d*, *s1*, and *s2* are register numbers in the range 0..15. *n* is a constant in the range 1..32. ASR is the arithmetic shift right operation (*Rs2*'s value is shifted to the right *n* bits using an ASR operation). The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, *Rs2*, ASR *Rs3* adcs *Rd*, *Rs1*, *Rs2*, ASR *Rs3*	*Rd* := *Rs1* + (*Rs2* ASR *Rs3*) + carry *d*, *s1*, *s2*, and *s3* are register numbers in the range 0..15. *Rs3* must contain a value in the range 1..32. ASR is the arithmetic shift right operation (*Rs2*'s value is shifted to the right the number of bits specified by *Rs3* using an ASR operation). The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, *Rs2*, LSL *#n* adcs *Rd*, *Rs1*, *Rs2*, LSL *#n*	*Rd* := *Rs1* + (*Rs2* LSL *n*) + carry *d*, *s1*, and *s2* are register numbers in the range 0..15. *n* is a constant in the range 0..31. LSL is the logical shift left operation (*Rs2*'s value is shifted to the left *n* bits using an LSL operation). The adcs form updates the processor status bits based on the instruction results.

(continued)

Table C-2: Gas Syntax for 32-bit adc (continued)

Instruction	Description
adc *Rd*, *Rs1*, *Rs2*, LSL *Rs3* adcs *Rd*, *Rs1*, *Rs2*, LSL *Rs3*	*Rd* := *Rs1* + (*Rs2* LSL *Rs3*) + carry *d*, *s1*, *s2*, and *s3* are register numbers in the range 0..15. *Rs3* must contain a value in the range 0..31. LSL is the logical shift left operation (*Rs2*'s value is shifted to the left the number of bits specified by *Rs3* using an LSL operation). The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, *Rs2*, LSR #*n* adcs *Rd*, *Rs1*, *Rs2*, LSR #*n*	*Rd* := *Rs1* + (*Rs2* LSR *n*) + carry *d*, *s1*, and *s2* are register numbers in the range 0..15. *n* is a constant in the range 1..32. LSR is the logical shift right operation (*Rs2*'s value is shifted to the right *n* bits using an LSR operation). The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, *Rs2*, LSR *Rs3* adcs *Rd*, *Rs1*, *Rs2*, LSR *Rs3*	*Rd* := *Rs1* + (*Rs2* LSR *Rs3*) + carry *d*, *s1*, *s2*, and *s3* are register numbers in the range 0..15. *Rs3* must contain a value in the range 1..32. LSR is the logical shift right operation (*Rs2*'s value is shifted to the right the number of bits specified by *Rs3* using an LSR operation). The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, *Rs2*, ROR #*n* adcs *Rd*, *Rs1*, *Rs2*, ROR #*n*	*Rd* := *Rs1* + (*Rs2* ROR *n*) + carry *d*, *s1*, and *s2* are register numbers in the range 0..15. *n* is a constant in the range 1..31. ROR is the logical rotate right operation (*Rs2*'s value is rotated to the right *n* bits using an ROR operation). The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, *Rs2*, ROR *Rs3* adcs *Rd*, *Rs1*, *Rs2*, ROR *Rs3*	*Rd* := *Rs1* + (*Rs2* ROR *Rs3*) + carry *d*, *s1*, *s2*, and *s3* are register numbers in the range 0..15. *Rs3* must contain a value in the range 1..32. ROR is the logical rotate right operation (*Rs2*'s value is rotated to the right the number of bits specified by *Rs3* using an ROR operation). The adcs form updates the processor status bits based on the instruction results.
adc *Rd*, *Rs1*, *Rs2*, RRX #1 adcs *Rd*, *Rs1*, *Rs2*, RRX #1	*Rd* := *Rs1* + (*Rs2* RRX 1) + carry *d*, *s1*, and *s2* are register numbers in the range 0..15. *n* is a constant in the range 1..31. RRX is the extended rotate right operation (*Rs2*'s value is rotated to the right 1 bit, with bit 0 going into the carry flag and the old carry flag value going into bit 31). The adcs form updates the processor status bits based on the instruction results.

The and, eor, sub, rsb, add, sbc, rsc, orr, and bic instructions all take the same forms as the adc instruction with the exception of their operations, which are described in Table C-3. As with adc, these instructions allow an s suffix, which updates the condition code bits after the execution of the instruction (without the suffix, these instructions do not affect the condition code flags).

Table C-3: Data Manipulation Instructions: and, eor, sub, rsb, add, sbc, rsc, orr, and bic

Instruction	Description
and, ands	*Rd* is set to bitwise logical AND of *Rs1* and operand 2.
eor, eors	*Rd* is set to bitwise logical exclusive-OR of *Rs1* and operand 2.
sub, subs	*Rd* is set to *Rs1* – operand 2.
rsb, rsbs	*Rd* is set to operand 2 – *Rs1*.
add, adds	*Rd* is set to *Rs1* + operand 2.
sbc, sbcs	*Rd* is set to *Rs1* – operand 2 - not(carry).
rsc, rscs	*Rd* is set to operand 2 – *Rs1* – not(carry).
orr, orrs	*Rd* is set to bitwise logical OR of *Rs1* and operand 2.
bic, bics	*Rd* is set to bitwise logical AND of *Rs1* and not(operand 2).

The ARM cmp, cmn, tst, and teq instructions do not produce a destination value; instead, they update only the condition code flags (listed in Table C-4). Accordingly, these instructions don't support the s suffix. Furthermore, they accept only two source operands, as they don't update a destination register.

Table C-4: Gas Syntax for 32-bit cmp Instruction

Instruction	Description
cmp *Rd*, *Rs1*, *Rs2* cmp *Rd*, *Rs1*, *Rs2*	*Rd* := *Rs1* to *Rs2* and updates the condition code flags. *s1* and *s2* are register numbers in the range 0..15.
cmp *Rd*, *Rs1*, #imm_8 cmp *Rd*, *Rs1*, #imm_8	*Rd* := *Rs1* to imm_8 and updates the condition code flags. *s1* is a register number in the range 0..15. The imm_8 operand is a 32-bit constant that can be encoded using the ARM 12-bit rotated immediate format (see ""The Immediate Addressing Mode" on page 9).
cmp *Rs1*, *Rs2*, ASR #*n* cmp *Rs1*, *Rs2*, ASR #*n*	Compares *Rs1* to (*Rs2* ASR *n*) and updates the condition code flags. *s1* and *s2* are register numbers in the range 0..15. *n* is a constant in the range 1..32. ASR is the arithmetic shift right operation (*Rs2*'s value is shifted to the right *n* bits using an ASR operation).
cmp *Rs1*, *Rs2*, ASR *Rs3* cmp *Rs1*, *Rs2*, ASR *Rs3*	Compares *Rs1* to (*Rs2* ASR *Rs3*) and updates the condition code flags. *s1*, *s2*, and *s3* are register numbers in the range 0..15. *Rs3* must contain a value in the range 1..32. ASR is the arithmetic shift right operation (*Rs2*'s value is shifted to the right the number of bits specified by *Rs3* using an ASR operation).
cmp *Rs1*, *Rs2*, LSL #*n* cmp *Rs1*, *Rs2*, LSL #*n*	Compares *Rs1* to (*Rs2* LSL *n*) and updates the condition code flags. *s1* and *s2* are register numbers in the range 0..15. *n* is a constant in the range 0..31. LSL is the logical shift left operation (*Rs2*'s value is shifted to the left *n* bits using an LSL operation).

(continued)

Table C-4: Gas Syntax for 32-bit cmp Instruction (continued)

Instruction	Description
cmp *Rs1*, *Rs2*, LSL *Rs3* cmp *Rs1*, *Rs2*, LSL *Rs3*	Compares *Rs1* to (*Rs2* LSL *Rs3*) and updates the condition code flags. *s1*, *s2*, and *s3* are register numbers in the range 0..15. *Rs3* must contain a value in the range 0..31. LSL is the logical shift left operation (*Rs2*'s value is shifted to the left the number of bits specified by *Rs3* using an LSL operation).
cmp *Rs1*, *Rs2*, LSR #*n* cmp *Rs1*, *Rs2*, LSR #*n*	Compares *Rs1* to (*Rs2* LSR *n*) and updates the condition code flags. *s1* and *s2* are register numbers in the range 0..15. *n* is a constant in the range 1..32. LSR is the logical shift right operation (*Rs2*'s value is shifted to the right *n* bits using an LSR operation).
cmp *Rs1*, *Rs2*, LSR *Rs3* cmp *Rs1*, *Rs2*, LSR *Rs3*	Compares *Rs1* to (*Rs2* LSR *Rs3*) and updates the condition code flags. *s1*, *s2*, and *s3* are register numbers in the range 0..15. *Rs3* must contain a value in the range 1..32. LSR is the logical shift right operation (*Rs2*'s value is shifted to the right the number of bits specified by *Rs3* using an LSR operation).
cmp *Rs1*, *Rs2*, ROR #*n* cmp *Rs1*, *Rs2*, ROR #*n*	Compares *Rs1* to (*Rs2* ROR *n*) and updates the condition code flags. *s1* and *s2* are register numbers in the range 0..15. *n* is a constant in the range 1..31. ROR is the logical rotate right operation (*Rs2*'s value is rotated to the right *n* bits using an ROR operation).
cmp *Rs1*, *Rs2*, ROR *Rs3* cmp *Rs1*, *Rs2*, ROR *Rs3*	Compares *Rs1* to (*Rs2* ROR *Rs3*) and updates the condition code flags. *s1*, *s2*, and *s3* are register numbers in the range 0..15. *Rs3* must contain a value in the range 1..32. ROR is the logical rotate right operation (*Rs2*'s value is rotated to the right the number of bits specified by *Rs3* using an ROR operation).
cmp *Rs1*, *Rs2*, RRX #1 cmp *Rs1*, *Rs2*, RRX #1	Compares *Rs1* to (*Rs2* RRX 1) and updates the condition code flags. *s1* and *s2* are register numbers in the range 0..15. *n* is a constant in the range 1..31. RRX is the extended rotate right operation (*Rs2*'s value is rotated to the right 1 bit, with bit 0 going into the carry flag and the old carry flag value going into bit 31).

The cmp instruction performs the same operation as a sub instruction except that cmp doesn't store a result into a destination register.

Table C-5 describes the cmn, tst, and teq instructions, which use the same syntax as cmp.

Table C-5: Data Manipulation Instructions: `cmn`, `tst`, and `teq`

Instruction	Description
CMN	Compare negated value. Compares *Rs1* to a negated copy of operand 2 and updates the flags. Note that this is equivalent to adding *Rs1* to operand 2 and updating the flags.
TST	Computes the logical AND of *Rs1* and operand 2 and updates the flags.
TEQ	Computes the logical exclusive-OR of *Rs1* and operand 2 and updates the flags.

The final two data manipulation instructions, `mov` and `mvn`, also support only two operands. However, these instructions have a destination register and an operand 2, dropping the *Rs1* operand. These instructions copy data from operand 2 to the destination register. The difference between the two instructions is that the `mvn` (move negated) instruction logically inverts the bits of operand 2's value before copying the value into the destination register. The syntax for these instructions appears in Table C-6.

Table C-6: Gas Syntax for 32-bit `mov` and `mvn` Instructions

Instruction	Description

mov *Rd*, *Rs2*
movs *Rd*, *Rs2*
mvn *Rd*, *Rs2*
mvns *Rd*, *Rs2*
	Rd := *Rs2* // mov
	Rd := not *Rs2* // mvn
	d and *s2* are register numbers in the range 0..15.
	The movs and mvns forms update the processor status bits based on the instruction results.

mov *Rd*, #imm_8
movs *Rd*, #imm_8
mvn *Rd*, #imm_8
mvns *Rd*, #imm_8
	Rd := imm_8 // mov
	Rd := not imm_8 // mvn
	d is a register number in the range 0..15. The imm_8 operand is a 32-bit constant that can be encoded using the ARM 12-bit rotated immediate format (see ""The Immediate Addressing Mode" on page 9).
	The movs and mvns forms update the processor status bits based on the instruction results.

mov *Rd*, *Rs2*, ASR #*n*
movs *Rd*, *Rs2*, ASR #*n*
mvn *Rd*, *Rs2*, ASR #*n*
mvns *Rd*, *Rs2*, ASR #*n*
	Rd := (*Rs2* ASR *n*) // mov
	Rd := not (*Rs2* ASR *n*) // mvn
	d and *s2* are register numbers in the range 0..15.
	n is a constant in the range 1..32.
	ASR is the arithmetic shift right operation (*Rs2*'s value is shifted to the right *n* bits using an ASR operation).
	The movs and mvns forms update the processor status bits based on the instruction results.

mov *Rd*, *Rs2*, ASR *Rs3*
movs *Rd*, *Rs2*, ASR *Rs3*
mvn *Rd*, *Rs2*, ASR *Rs3*
mvns *Rd*, *Rs2*, ASR *Rs3*
	Rd := (*Rs2* ASR *Rs3*) // mov
	Rd := not (*Rs2* ASR *Rs3*) // mvn
	d, *s2*, and *s3* are register numbers in the range 0..15.
	Rs3 must contain a value in the range 1..32.
	ASR is the arithmetic shift right operation (*Rs2*'s value is shifted to the right the number of bits specified by *Rs3* using an ASR operation).
	The movs and mvns forms updates the processor status bits based on the instruction results.

(continued)

Table C-6: Gas Syntax for 32-bit mov and mvn Instructions (continued)

Instruction	Description

mov *Rd*, *Rs2*, LSL *#n*
movs *Rd*, *Rs2*, LSL *#n*
mvn *Rd*, *Rs2*, LSL *#n*
mvns *Rd*, *Rs2*, LSL *#n*
	Rd := (*Rs2* LSL *n*) // mov
	Rd := not (*Rs2* LSL *n*) // mvn
	d and *s2* are register numbers in the range 0..15.
	n is a constant in the range 0..31.
	LSL is the logical shift left operation (*Rs2*'s value is shifted to the left *n* bits using an LSL operation).
	The movs and mvns forms update the processor status bits based on the instruction results.

mov *Rd*, *Rs2*, LSL *Rs3*
movs *Rd*, *Rs2*, LSL *Rs3*
mvn *Rd*, *Rs2*, LSL *Rs3*
mvns *Rd*, *Rs2*, LSL *Rs3*
	Rd := (*Rs2* LSL *Rs3*) // mov
	Rd := not (*Rs2* LSL *Rs3*) // mvn
	d, *s2*, and *s3* are register numbers in the range 0..15.
	Rs3 must contain a value in the range 0..31.
	LSL is the logical shift left operation (*Rs2*'s value is shifted to the left the number of bits specified by *Rs3* using an LSL operation).
	The movs and mvns forms update the processor status bits based on the instruction results.

mov *Rd*, *Rs2*, LSR *#n*
movs *Rd*, *Rs2*, LSR *#n*
mvn *Rd*, *Rs2*, LSR *#n*
mvns *Rd*, *Rs2*, LSR *#n*
	Rd := (*Rs2* LSR *n*) // mov
	Rd := not (*Rs2* LSR *n*) // mvn
	d and *s2* are register numbers in the range 0..15.
	n is a constant in the range 1..32.
	LSR is the logical shift right operation (*Rs2*'s value is shifted to the right *n* bits using an LSR operation).
	The movs and mvns forms update the processor status bits based on the instruction results.

mov *Rd*, *Rs2*, LSR *Rs3*
movs *Rd*, *Rs2*, LSR *Rs3*
mvn *Rd*, *Rs2*, LSR *Rs3*
mvns *Rd*, *Rs2*, LSR *Rs3*
	Rd := (*Rs2* LSR *Rs3*) // mov
	Rd := not (*Rs2* LSR *Rs3*) // mvn
	d, *s2*, and *s3* are register numbers in the range 0..15.
	Rs3 must contain a value in the range 1..32.
	LSR is the logical shift right operation (*Rs2*'s value is shifted to the right the number of bits specified by *Rs3* using an LSR operation).
	The movs and mvns forms update the processor status bits based on the instruction results.

mov *Rd*, *Rs2*, ROR *#n*
movs *Rd*, *Rs2*, ROR *#n*
mvn *Rd*, *Rs2*, ROR *#n*
mvns *Rd*, *Rs2*, ROR *#n*
	Rd := (*Rs2* ROR *n*) // mov
	Rd := not (*Rs2* ROR *n*) // mvn
	d, and *s2* are register numbers in the range 0..15.
	n is a constant in the range 1..31.
	ROR is the logical rotate right operation (*Rs2*'s value is rotated to the right *n* bits using an ROR operation).
	The movs and mvns forms update the processor status bits based on the instruction results.

mov *Rd*, *Rs2*, ROR *Rs3*
movs *Rd*, *Rs2*, ROR *Rs3*
mvn *Rd*, *Rs2*, ROR *Rs3*
mvns *Rd*, *Rs2*, ROR *Rs3*
	Rd := (*Rs2* ROR *Rs3*) // mov
	Rd := not (*Rs2* ROR *Rs3*) // mvn
	d, *s2*, and *s3* are register numbers in the range 0..15.
	Rs3 must contain a value in the range 1..32.
	ROR is the logical rotate right operation (*Rs2*'s value is rotated to the right the number of bits specified by *Rs3* using an ROR operation).
	The movs and mvns forms update the processor status bits based on the instruction results.

(continued)

Table C-6: Gas Syntax for 32-bit mov and mvn Instructions (continued)

Instruction	Description

mov *Rd*, *Rs2*, RRX #1
movs *Rd*, *Rs2*, RRX #1
mvn *Rd*, *Rs2*, RRX #1
mvns *Rd*, *Rs2*, RRX #1
	Rd := (*Rs2* RRX 1) // mov
	Rd := not (*Rs2* RRX 1) // mvn
	d, *s1*, and *s2* are register numbers in the range 0..15.
	n is a constant in the range 1..31.
	RRX is the extended rotate right operation (*Rs2*'s value is rotated to the right 1 bit, with bit 0 going into the carry flag and the old carry flag value going into bit 31).
	The movs and mvns forms update the processor status bits based on the instruction results.

C.8.2 Conditional Suffixes for Instructions

By default, the data manipulation instructions always execute when the CPU encounters them in the code stream. However, every data manipulation instruction supports 16 variants, 15 of which execute only under certain conditions. The CPU condition code flags determine those conditions (see Table C-7).

Table C-7: Conditional Data Manipulation Instruction Suffixes

Condition code setting(s)	Instruction suffix	Meaning
Z=1	eq	Execute if equal (or 0). Instruction executes if zero flag is set.
Z=0	ne	Execute if not equal (or nonzero). Instruction executes if zero flag is clear.
C=1	cs or hs	Execute if carry set/higher or same (unsigned greater than or equal).
C=0	cc or lo	Execute if carry clear/lower (unsigned less than).
N=1	mi	Execute if minus (HO bit of result was set).
N=0	pl	Execute if plus or 0 (HO bit of result was clear).
V=1	vs	Execute if signed overflow.
V=0	vc	Execute if no signed overflow.
C=1 && Z=0	hi	Execute if higher (unsigned greater than).
C=0 || Z=1	ls	Execute if lower or same (unsigned less than or equal).
N==V	ge	Execute if (signed) greater than or equal.
N!=V	lt	Execute if (signed) less than.
Z==0 && N==V	gt	Execute if (signed) greater than.
Z==1 && N!=V	le	Execute if (signed) less than or equal.
n/a	AL (default)	Always execute (AL suffix is optional; always execute is the default condition).

Consider the instruction `moveq`. Upon encountering this instruction, the CPU tests the zero flag. If the zero flag is clear, the CPU skips over this instruction without executing it. If the zero flag is set, the CPU executes the instruction as if the `eq` suffix weren't present.

Note that encoding these 15 different conditions into the data manipulation instructions consumes 4 bits in the instructions' opcodes. In the 64-bit A64 processors, those bits were reclaimed for other purposes (such as 64-bit operations). Therefore, conditionally executed instructions (other than branches) are available only in 32-bit programs.

C.8.3 Multiply Instructions

The ARM instruction set includes several multiply instructions that multiply the values in two registers and store the result in a destination register. These instructions appear in Table C-8.

Table C-8: Gas Syntax for 32-Bit Multiply Instructions

Instruction	Description
`mul` *Rd*`,` *Rs1*`,` *Rs2* `muls` *Rd*`,` *Rs1*`,` *Rs2*	*Rd* := *Rs1* × *Rs2* (signed or unsigned). *d*, *s1*, and *s2* are register numbers in the range `0..15`. The `muls` form updates the processor status bits based on the instruction results.
`mla` *Rd*`,` *Rs1*`,` *Rs2*`,` *Rs3* `mlas` *Rd*`,` *Rs1*`,` *Rs2*`,` *Rs3*	*Rd* := *Rs1* × *Rs2* + *Rs3* (signed or unsigned). *d*, *s1*, *s2*, and *s3* are register numbers in the range `0..15`. The `mlas` form updates the processor status bits based on the instruction results.
`umull` *Rdhi*`,` *Rdlo*`,` *Rs1*`,` *Rs2* `umulls` *Rdhi*`,` *Rdlo*`,` *Rs1*`,` *Rs2*	(*Rdhi*:*Rdlo*) := *Rs1* × *Rs2* (unsigned) *dhi*, *dlo*, *s1*, and *s2* are register numbers in the range `0..15`. *Rdhi* will hold the upper 32 bits of the result; *Rdlo* will hold the lower 32 bits of the result. The `umulls` form updates the processor status bits based on the instruction results.
`umaal` *Rdhi*`,` *Rdlo*`,` *Rs1*`,` *Rs2* `umaals` *Rdhi*`,` *Rdlo*`,` *Rs1*`,` *Rs2*	(*Rdhi*, *Rdlo*) := *Rs1* × *Rs2* + (*Rdhi*:*Rdlo*) (unsigned) *dhi*, *dlo*, *s1*, and *s2* are register numbers in the range `0..15`. *Rdhi* will hold the upper 32 bits of the result; *Rdlo* will hold the lower 32 bits of the result. The `umaals` form updates the processor status bits based on the instruction results.
`smull` *Rdhi*`,` *Rdlo*`,` *Rs1*`,` *Rs2* `smulls` *Rdhi*`,` *Rdlo*`,` *Rs1*`,` *Rs2*	*(Rdhi:Rdlo)* := *Rs1* × *Rs2* (signed) *dhi*, *dlo*, *s1*, and *s2* are register numbers in the range `0..15`. *Rdhi* will hold the upper 32 bits of the result; *Rdlo* will hold the lower 32 bits of the result. The `smulls` form updates the processor status bits based on the instruction results.

(continued)

Table C-8: Gas Syntax for 32-Bit Multiply Instructions (continued)

Instruction	Description
`smaal` *`Rdhi, Rdlo, Rs1, Rs2`* `smaals` *`Rdhi, Rdlo, Rs1, Rs2`*	(*`Rdhi:Rdlo`*) := *`Rs1`* × *`Rs2`* + (*`Rdhi:Rdlo`*) (signed) *`dhi`*, *`dlo`*, *`s1`*, and *`s2`* are register numbers in the range `0..15`. *`Rdhi` will hold the upper 32 bits of the result; `Rdlo` will hold the lower 32 bits of the result.* The `smaals` form updates the processor status bits based on the instruction results.

The 64-bit ARM instruction set provides some additional forms of the multiply instruction. See the ARM documentation for more details.

Note that the base 32-bit ARM instruction set does not include integer divide or remainder instructions. Applications must implement the division/modulo operations using software functions. The 64-bit ARM instruction set does provide signed and unsigned division operations; see the ARM documentation for more details.

C.8.4 Branch Instructions

There are two basic ARM branch instructions: `b` (branch) and `bl` (branch and link). Both instructions have a single label operand that specifies a destination location that must be within about ±32MB from the current instruction (actually, from about 8 to 12 bytes after the current instruction, due to issues with the pipeline on the ARM CPU).

The `b` instruction simply transfers control to the target location by adding the 24-bit offset (shifted to the right by 2 bits) encoded into the instruction to the PC register (R15). This is a standard unconditional branch on the ARM processor.

On both the 32-bit (A32) and 64-bit (A64) architectures, the branch instruction allows conditional suffixes (see "Conditional Suffixes for Instructions" on page 24). For example, the `beq label` instruction transfers control to `label` if and only if the zero flag is set; if the zero flag is clear, control falls through to the next instruction after the `beq` instruction.

The `bl` instruction also transfers control to the target location (the same way as the `b` instruction). However, in addition to transferring control, `bl` also copies the address of the next instruction (that is, the one following it) into the LR. This is the basis for a subroutine call on the ARM processor—the LR (R14 on A32; X30 on A64) will hold the subroutine's return address upon entry to the subroutine. On the A32, you can return from the subroutine by copying the LR into the PC register using the following `mov` instruction:

```
mov PC, LR // same as mov r15, r14 on A32
```

On the 64-bit A64, you can use the `ret` instruction to return from a subroutine. Without an operand, the instruction assumes the return address is

in the LR (X30). You can also specify a different register (X0 through X30) by specifying the register name as an operand to the `ret` instruction. The A64 also has a `br` instruction (branch through register) that functionally does the same thing as `ret.`, except `ret` also tells the ARM branch prediction hardware that you are returning from a subroutine. This helps the branch prediction hardware run the code a little faster.

Note that the execution of the `bl` instruction wipes out the old value in the LR. Nested subroutine calls (that is, calling a subroutine and then having that subroutine's body call a second subroutine) are impossible unless you explicitly save the value in the LR to memory (specifically, to some subroutine return address stack) and return from the subroutine by jumping indirect via that memory location.

The 32-bit ARM instruction set allows conditional suffixes on the `bl` instruction. For example, `blmi` *`func`* calls the specified function (and updates the LR) if and only if the negative flag was set. If the negative flag was clear, control falls through to the instruction following the `blmi` instruction and the LR's contents do not change.

The 32-bit ARM instruction set allows you to treat PC as a general-purpose register. You can move the value of some other register into the PC using the `mov` instruction (see the earlier code example). On the 64-bit ARM, you can do the same thing with the `br` or `ret` instructions. The 32-bit ARM instruction set also allows you to copy the PC register to some other general-purpose register. Alas, this isn't directly possible using 64-bit ARM instructions, though you can use `adr` *`Rd`*`, .` to do the same thing.

C.8.5 Load and Store Instructions

As mentioned earlier, the ARM processor accesses memory using only load and store instructions. While the ARM CPU supports several, the primary ones are `ldr` (load register) and `str` (store register). These instructions require two operands—a register operand and a memory operand—and have the following syntax (braces indicate optional items and do not appear in the actual instruction):

```
ldr{cond}{B|H|SB|SH}{T} Rd, Address
str{cond}{B}{T} Rd, Address
```

where:

- *`cond`* represents one of the conditional suffixes (see "Conditional Suffixes for Instructions" on page 24). If no suffix is present, the default is *always* `AL`.
- `B|H|SB|SH` means that only one (or none) of the suffixes `B`, `H`, `SB`, or `SH` may appear in the instruction. `B` indicates a byte-sized transfer, with zero extension to 32 bits. `H` indicates a halfword (16-bit) data transfer with zero extension to 32 bits. `SB` indicates a signed byte transfer with sign extension to 32 bits. Finally, `SH` indicates a signed halfword (16-bit) transfer with sign extension to 32 bits.

- The `T` suffix is intended for operating system use only. You probably won't see this suffix in application code. If it is present, you can ignore it.
- *`d`* is a register number in the range `0` through `15`.
- *`Address`* is a memory addressing mode (see the following discussion).

Memory addresses (*`Address`*) can be any of the following:

- The label of a (nearby) variable in memory (for example, `ldr r0, someVar`). The label must reference a memory location that is within ±4,096 bytes of the current instruction (PC-relative addressing mode).
- A preindexed addressing mode (see "Preindexed Addressing with Write-Back" on page 11).
- A post-indexed addressing mode (see "Post-Indexed Addressing with Write-Back" on page 11).
- A scaled-indexed addressing mode (see "Scaled-Index Addressing Mode" on page 11).

The `ldr` instruction copies the (byte or 32-bit word) data value from the specified memory address into register *`Rd`*. The `ldrb` instruction copies the byte at the specified address to the LO 8 bits of *`Rd`* and then zeros out the upper 24 bits of *`Rd`*. The `ldrsb` instruction copies the byte at the specified address to the LO 8 bits of *`Rd`* and then sign-extends this value throughout the upper 24 bits of *`Rd`*. The `ldrh` instruction copies the 16-bit halfword at the specified address to the LO 16 bits of *`Rd`* and then zeros out the upper 16 bits of *`Rd`*. The `ldrsh` instruction copies the 16-bit halfword at the specified address to the LO 16 bits of *`Rd`* and then sign-extends this value throughout the upper 16 bits of *`Rd`*.

If a condition suffix appears immediately after the `ldr` or `str` (and before the `B`, `SB`, `H`, `SH`, and `T` suffixes), the CPU executes only the load or store instruction if the specified condition is `true`. Note that these instructions do not affect the condition code flags during execution (there is no `s` suffix telling the CPU to update the flags after the data movement). If you need to set the condition code flags after a load or store operation, use the `tst` instruction, supplying *`Rd`* as both operands immediately after the execution of the `ldr` or `str` instruction.

The 32-bit ARM instruction set also allows you to load or store multiple general-purpose registers with a single instruction. The `ldm` instruction loads multiple registers from memory, while the `stm` instruction stores multiple registers from memory. The syntax for these two instructions is as follows:

```
ldm{cond}ED Rb {!}, Reglist {^}
ldm{cond}IB Rb {!}, Reglist {^}
ldm{cond}FD Rb {!}, Reglist {^}
ldm{cond}IA Rb {!}, Reglist {^}
ldm{cond}EA Rb {!}, Reglist {^}
ldm{cond}DB Rb {!}, Reglist {^}
ldm{cond}FA Rb {!}, Reglist {^}
ldm{cond}DA Rb {!}, Reglist {^}
```

```
stm{cond}ED Rb {!}, Reglist {^}
stm{cond}IB Rb {!}, Reglist {^}
stm{cond}FD Rb {!}, Reglist {^}
stm{cond}IA Rb {!}, Reglist {^}
stm{cond}EA Rb {!}, Reglist {^}
stm{cond}DB Rb {!}, Reglist {^}
stm{cond}FA Rb {!}, Reglist {^}
stm{cond}DA Rb {!}, Reglist {^}
```

As before, braces surround optional items and the italicized items have the following meanings:

- *cond* represents one of the conditional suffixes (see "Conditional Suffixes for Instructions" on page 24). If no suffix is present, the default is *always* `AL`.
- `!` means that the updated address is written back to *Rd* after the data movement.
- `^` is really for use by operating system code and shouldn't be present in normal compiler-generated code. If you see this, you can ignore it.
- *Rb* is the base register. This points to memory where the registers will be stored to or loaded from (see the following discussion).
- *Reglist* is a list of registers appearing in a pair of braces (note that the braces are actually required here; they do not denote optional items). The register list is a comma-separated list of registers, such as `{R0, R5, R8}`. Optionally, a range of consecutive registers can be specified by a hyphen, such as `{R0, R2-R5, R8}`.

The `ldm` and `stm` instructions will load or store all the registers appearing in the register list. For example, `ldmed R0, {R1, R2}` will load registers R1 and R2 from memory. Similarly, `stmed R11, {R0-R10}` will write 11 registers to memory (R0 through R10).

The `ED`, `FD`, `EA`, and `FA` suffixes are described in Table C-9. The suffixes `IA` (increment after), `IB` (increment before), `DA` (decrement after), and `DB` (decrement before) are synonyms for `FD`, `ED`, `FA`, and `EA`, respectively.

Table C-9: ARM `stm` and `ldm` Instructions

Instruction	Description
`ldmed` *Rb*`, {`*Reglist*`}` `ldmib` *Rb*`, {`*Reglist*`}`	Load multiple, preincrement addressing. *Rb* (base address) register is incremented (by 4) prior to loading each register in *Reglist* from memory (at the incremented address in *Rb*). At the end of the operation, *Rb* will point at the last register value loaded from memory.
`ldmfd` *Rb*`, {`*Reglist*`}` `ldmia` *Rb*`, {`*Reglist*`}`	Load multiple, post-increment addressing. *Rb* (base address) register is incremented (by 4) after loading each register in *Reglist* from memory (at the incremented address in *Rb*). At the end of the operation, *Rb* will point 4 bytes beyond the last register value loaded from memory.

(continued)

Table C-9: ARM `stm` and `ldm` Instructions (continued)

Instruction	Description
`ldmea` *Rb*, {*Reglist*} `ldmdb` *Rb*, {*Reglist*}	Load multiple, predecrement addressing. *Rb* (base address) register is decremented (by 4) prior to loading each register in *Reglist* from memory (at the decremented address in *Rb*). At the end of the operation, *Rb* will point at the last register value loaded from memory.
`ldmfa` *Rb*, {*Reglist*} `ldmda` *Rb*, {*Reglist*}	Load multiple, post-decrement addressing. *Rb* (base address) register is decremented (by 4) after loading each register in *Reglist* to memory (at the decremented address in *Rb*). At the end of the operation, *Rb* will point 4 bytes beyond the last register value loaded from memory.
`stmed` *Rb*, {*Reglist*} `stmib` *Rb*, {*Reglist*}	Store multiple, preincrement addressing. *Rb* (base address) register is incremented (by 4) prior to storing each register in *Reglist* to memory (at the incremented address in *Rb*). At the end of the operation, *Rb* will point at the last register value stored into memory.
`stmfd` *Rb*, {*Reglist*} `stmia` *Rb*, {*Reglist*}	Store multiple, post-increment addressing. *Rb* (base address) register is incremented (by 4) after storing each register in *Reglist* to memory (at the incremented address in *Rb*). At the end of the operation, *Rb* will point 4 bytes beyond the last register value stored into memory.
`stmea` *Rb*, {*Reglist*} `stmdb` *Rb*, {*Reglist*}	Store multiple, predecrement addressing. *Rb* (base address) register is decremented (by 4) prior to storing each register in *Reglist* to memory (at the decremented address in *Rb*). At the end of the operation, *Rb* will point at the last register value stored into memory.
`stmfa` *Rb*, {*Reglist*} `stmda` *Rb*, {*Reglist*}	Store multiple, post-decrement addressing. *Rb* (base address) register is decremented (by 4) after storing each register in *Reglist* to memory (at the decremented address in *Rb*). At the end of the operation, *Rb* will point 4 bytes beyond the last register value stored into memory.

One last load/store instruction of interest is the swap instruction. This instruction actually performs both a load and store operation simultaneously. It has the following syntax:

```
swp{cond}{B} Rd, Rs, [Rb]
```

where:

- *cond* is the usual condition code suffix for conditional execution.
- `B` specifies a byte-sized transfer (if present; 32-bit word transfer if not present). Note that halfword (16-bit) transfers are not possible with this instruction.
- *Rd* is the destination register. The `swp` instruction copies the data originally held at memory location [*Rb*] into this register. For byte transfers, the `swp` instruction zero-extends the byte value into the *Rd* register.
- *Rs* is the source register. The `swp` instruction stores the value held in this register into the memory location specified by [*Rb*] after fetching the

data from that memory location (to store into *Rd*). For byte transfers, the `swp` instruction stores only the LO 8 bits of *Rs* into the memory location pointed at by [*Rb*].

- [*Rb*] is the memory address (held in *Rb*, the base register) whose contents the CPU will swap with *Rd* and *Rs*.

The swap operation is an atomic operation. The CPU will lock the bus to guarantee that the operation completes without interruption (for concurrency locking operations).

C.8.6 Software Interrupt Instruction

The `swi` (software interrupt) instruction provides a mechanism for making a controlled operating system call. The 32-bit instruction code for `swi` includes an 8-bit opcode and a 24-bit *comment* code (a constant). The CPU ignores the comment code, which the OS can use as an OS call number. This instruction has the following syntax:

```
swi constant
```

where *constant* is an expression that evaluates to a 24-bit (or smaller) constant at compile time. The assembler will encode this constant as part of the `swi` instruction code for use by the operating system.

C.8.7 ARM Floating-Point Instructions

The native ARM CPU does not provide support for floating-point arithmetic. Instead, coprocessors (often built onto the same die as the ARM CPU) provide extensions to the ARM instruction set; certain sets of coprocessor extensions include support for floating-point arithmetic. This section discusses the NEON/VFP floating-point extensions to the ARM architecture. NEON is a specialized *single-instruction/multiple-data (SIMD)* extension to the integer instruction set. While some compilers will actually emit SIMD instructions, you'll rarely see them in the type of code this book describes. Therefore, this section concentrates on the ARM VFP (vector floating-point) instructions.

Keep in mind that the A64 floating-point instruction set supports twice as many registers as the A32 CPUs. Also, the A64 floating-point instruction set supports additional instructions and data types. This section ignores these extensions; see the ARMv8 (or later) reference manual for details.

C.8.7.1 The vld Instructions

The ARM architecture allows you to load a floating-point register with a constant or from a memory location or to store a floating-point register's value to memory. Except for immediate constants, the syntax is similar to the integer register `ldr` and `str` instructions. The `vldr` and `vstr` instructions appear in Table C-10.

Table C-10: ARM `vldr` and `vstr` Instructions

Instruction	Description
`vld{cond}.f32 Sd, =imm` `vld{cond}.f64 Dd, =imm`	This is actually a pseudo-instruction. The assembler creates an appropriate memory variable initialized with the immediate constant in the instruction and then creates a `vldr` instruction that loads the specified register from this memory location.
`vld{cond}.f32 Sd, [Rn, #offset]` `vld{cond}.f64 Dd, [Rn, #offset]`	Copies the 32-bit (`.f32`) or 64-bit (`.f64`) operand from memory to the specified destination floating-point register. The `#offset` item is optional (`0` is the default offset). If present, `offset` must be a multiple of 4 and in the range `-1020..+1020`.
`vld{cond}.f32 Sd, label` `vld{cond}.f64 Dd, label`	Copies the 32-bit (`.f32`) or 64-bit (`.f64`) operand from memory to the specified destination floating-point register. The memory location specified by `label` must be on a (32-bit) word within ±1,024 bytes of the current instruction.
`vstr{cond}.f32 Ss, [Rn, #offset]` `vstr{cond}.f64 Ds, [Rn, #offset]`	Stores the 32-bit (`.f32`) or 64-bit (`.f64`) source register value into the specified destination memory location. The `#offset` item is optional (`0` is the default offset). If present, `offset` must be a multiple of 4 and in the range `-1020..+1020`.
`vstr{cond}.f32 Ss, label` `vstr{cond}.f64 Ds, label`	Stores the 32-bit (`.f32`) or 64-bit (`.f64`) source register value into the specified destination memory location. The memory location specified by `label` must be on a (32-bit) word within ±1,024 bytes of the current instruction.
`vldmia{cond} Rb, reglist` `vldmia{cond} Rb!, reglist`	`Rb` holds the base address of a block of memory. `reglist` is a list of single- or double-precision registers (you cannot mix them in the same list) surrounded by braces. The `vldmia` (increment after) instruction loads the registers starting at `Rb` and incrementing the address by 4 or 8 (depending on the register size) after each transfer. If the `!` suffix appears on `Rb`, the final address is written back to `Rb` after loading the last register value from memory. Note: `vpop reglist` is a synonym for `vldmia sp!, reglist`.
`vldmdb{cond} Rb, reglist` `vldmdb{cond} Rb!, reglist`	`Rb` holds the base address of a block of memory. `reglist` is a list of single- or double-precision registers (you cannot mix them in the same list) surrounded by braces. The `vldmdb` (decrement before) instruction decrements `Rb` by 4 or 8 (depending on register list size) and then loads the registers from the address starting at `Rb`. If the `!` suffix appears on `Rb`, the final address is written back to `Rb` after loading the last register value from memory.
`vstmia{cond} Rb, reglist` `vstmia{cond} Rb!, reglist`	`Rb` holds the base address of a block of memory. `reglist` is a list of single- or double-precision registers (you cannot mix them in the same list) surrounded by braces. The `vstmia` (increment after) instruction stores the registers starting at `Rb`, incrementing the address by 4 or 8 after each transfer. If the `!` suffix appears on `Rb`, the final address is written back to `Rb` after storing the last register value to memory.

(continued)

Table C-10: ARM `vld` and `vstr` Instructions (continued)

Instruction	Description
`vstmdb{`*`cond`*`} Rb, `*`reglist`* `vstmdb{`*`cond`*`} Rb!, `*`reglist`*	*`Rb`* holds the base address of a block of memory. *`reglist`* is a list of single- or double-precision registers (you cannot mix them in the same list) surrounded by braces. The `vstmdb` (decrement before) instruction decrements *`Rb`* by 4 or 8 (depending on register list size) and then stores the registers to memory starting at the address in *`Rb`*. If the ! suffix appears on *`Rb`*, the final address is written back to *`Rb`* after storing the last register value to memory. Note: `vpush` *`reglist`* is a synonym for `vstmdb sp!,` *`reglist`*.

C.8.7.2 The vmov Instructions

The floating point `vmov` instructions take the forms shown in Table C-11.

Table C-11: ARM `vmov` Instructions

Instruction	Description
`vmov{`*`cond`*`}.f32 `*`Sd`*`, #imm` `vmov{`*`cond`*`}.f64 `*`Dd`*`, #imm`	Initializes a single- or double-precision floating-point register with a floating-point constant (immediate value). Constants are limited to any number that can be expressed as $\pm n \times 2 - r$, where n and r are integers, $16 \leq n \leq 31$; and $0 \leq r \leq 7$. *`Sd`* is a single-precision floating-point register in the range `S0..S15` (A32) or `S0..S31` (A64). *`Dd`* is a double-precision floating-point register in the range `D0..D15` (A32) or `D0..D31` (A64).
`vmov{`*`cond`*`}.f32 `*`Sd, Ss`* `vmov{`*`cond`*`}.f64 `*`Dd, Ds`*	Copies the value held in one floating-point register to another. *`Sd`* is a destination single-precision floating-point register in the range `S0..S15` (A32) or `S0..S31` (A64). *`Ss`* is a source single-precision floating-point register in the range `S0..S15` (A32) or `S0..S31` (A64). *`Dd`* is a destination double-precision floating-point register in the range `D0..D15` (A32) or `D0..D31` (A64). *`Ds`* is a source double-precision floating-point register in the range `D0..D15` (A32) or `D0..D31` (A64).
`vmov{`*`cond`*`}.f32 `*`Sd`*`, Rs` `vmov{`*`cond`*`}.f32 `*`Rd, Ss`*	Copies the value held in one floating-point register and a 32-bit general-purpose register. *`Sd`* is a destination single-precision floating-point register in the range `S0..S15` (A32) or `S0..S31` (A64). *`Ss`* is a source single-precision floating-point register in the range `S0..S15` (A32) or `S0..S31` (A64). *`Rd`* is a destination 32-bit general-purpose register in the range `D0..D15`. *`Rs`* is a source 32-bit general-purpose register in the range `D0..D15`.
`VMRS `*`Rd, vreg`* `VMSR `*`vreg, Rd`*	Copies data between a 32-bit general-purpose register and a special system register. *`Rd`* and *`Rs`* are 32-bit general purpose registers (destination and source). *`vreg`* is a special system register and is usually FPSCR, FPSID, or FPEXC.

C.8.7.3 Floating-Point Arithmetic Instructions

The instructions appearing in Table C-12 are responsible for various floating-point arithmetic operations, and Table C-13 lists the operand types for these instructions.

Table C-12: ARM Floating-Point Arithmetic Instructions

Instruction	Description
`vabs{cond}.f32 Sd, Ss` `vabs{cond}.f64 Dd, Ds`	Computes the absolute value of the source operand and stores the result into the destination operand.
`vneg{cond}.f32 Sd, Ss` `vneg{cond}.f64 Dd, Ds`	Negates the sign of the source operand and stores the result into the destination operand.
`vsqrt{cond}.f32 Sd, Ss` `vsqrt{cond}.f64 Dd, Ds`	Computes the square root of the source operand and stores the result into the destination operand.
`vadd{cond}.f32 Sd, Sl, Sr` `vadd{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = *Sl* + *Sr* Computes *Dd* = *Dl* + *Dr*
`vsub{cond}.f32 Sd, Sl, Sr` `vsub{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = *Sl* – *Sr* Computes *Dd* = *Dl* – *Dr* Note: Unlike in integer arithmetic, there is no floating-point reverse subtraction instruction. Simply swap the left and right registers if you want to compute a reverse subtraction.
`vdiv{cond}.f32 Sd, Sl, Sr` `vdiv{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = *Sl* / *Sr* Computes *Dd* = *Dl* / *Dr*
`vmul{cond}.f32 Sd, Sl, Sr` `vmul{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = *Sl* × *Sr* Computes *Dd* = *Dl* × *Dr*
`vmla{cond}.f32 Sd, Sl, Sr` `vmla{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = *Sd* + *Sl* × *Sr* Computes *Dd* = *Dd* + *Dl* × *Dr*
`vmls{cond}.f32 Sd, Sl, Sr` `vmls{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = *Sd* – *Sl* × *Sr* Computes *Dd* = *Dd* – *Dl* × *Dr*
`vnmul{cond}.f32 Sd, Sl, Sr` `vnmul{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = –(*Sl* × *Sr*) Computes *Dd* = –(*Dl* × *Dr*)
`vnmla{cond}.f32 Sd, Sl, Sr` `vnmla{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = –(*Sd* + *Sl* × *Sr*) Computes *Dd* = –(*Dd* + *Dl* × *Dr*)
`vnmls{cond}.f32 Sd, Sl, Sr` `vnmls{cond}.f64 Dd, Dl, Dr`	Computes *Sd* = –(*Sd* – *Sl* × *Sr*) Computes *Dd* = –(*Dd* – *Dl* × *Dr*)

Table C-13: ARM Floating-Point Instruction Operand Formats

Instruction	Description
Ss, *Sl*, *Sr*	Represents a single-precision floating point register holding the source operand (*l* and *r* stand for left and right source operands, respectively).
Sd	Represents a single-precision floating point register holding the destination operand.
Ds, *Dl*, *Dr*	Represents a double-precision floating point register holding the source operand (*l* and *r* stand for left and right source operands, respectively).
Dd	Represents a double-precision floating point register holding the destination operand.

C.9 For More Information

The list of ARM instructions described in this appendix is by no means comprehensive. If you're interested in additional details about the instruction set, consult ARM Inc.'s ARMv7 instruction set (Cortex A17) or ARMv8 instruction set (for example, Cortex A76) at *https://www.arm.com*. Better yet, follow the links at *https://en.wikipedia.org/wiki/ARM_architecture*.