### 第6章 体系结构

#### 6.1 引言

前面章节介绍了数字电路设计的原理，并设计了一些模块。在本章中，我们将进入到新的抽象层次来定义计算机的体系结构（Architecture）。体系结构是程序员所见到的计算机，它由指令集（汇编语言）和操作空间（寄存器和存储器）来定义。现在有不同类型的体系结构，例如ARM、MIPS、SPARC和PowerPC等。

理解任何计算机体系结构的第一步是学习它的语言。计算机语言中的单词叫做指令（Instruction）。计算机的词汇表叫做指令集（Instruction Set）。所有在同一计算机上运行的程序使用相同的指令集。即使是非常复杂的软件应用程序（如字处理软件或电子表格软件）也最终编译为一系列诸如加法、减法或跳转的简单指令。计算机指令包含需要完成的操作（Operation）和需要使用的操作数（Operand）两个部分，其中操作数来自于存储器、寄存器或者指令自身。

计算机硬件仅仅能理解二进制信息，所以指令也将被编码为二进制数，其格式称为机器语言（Machine Language）。正如使用字母来编码人类的语言，计算机使用二进制数编码机器语言。ARM体系结构将每条指令表示为一个32位的字。微处理器是一个可以读入并执行机器语言指令的数字电路系统。因为人类直接阅读二进制格式的机器语言会非常枯燥而乏味，所以使用符号形式来表示指令，称为汇编语言（Assemble Language）。

不同体系结构的指令集更像不同的方言，而不是不同的语言。几乎所有的体系结构都定义了基本指令，例如加法、减法和跳转，操作数也都来源于存储器或寄存器。一旦学习了一个指令集，理解其它指令集就将相当简单。

计算机体系结构并未确定底层的硬件实现。对于一个计算机体系结构，往往会有不同的硬件实现。例如，Intel公司和AMD公司销售的不同处理器都属于相同的X86体系结构。他们可以运行相同的程序，但是他们使用了不同的底层硬件实现，可以提供在价格、性能和功耗等方面的各种折衷。一些处理器专门对高性能服务器进行优化，而其它处理器可能为了延长笔记本计算机的电池寿命而针对功耗进行优化。寄存器、存储器、ALU和其它模块组织成微处理器的方式称为微结构（Microarchitecture），我们将在第7章中讨论这个主题。经常会有针对同一体系结构的不同微结构设计。

在本文中，我们将介绍ARM体系结构。这种架构最早是在20世纪80年代由Acorn Computer Group开发的，该公司独立出Advanced RISC Machines Ltd.，现在称为ARM。每年销售超过100亿个ARM处理器。几乎所有手机和平板电脑都包含多个ARM处理器。该架构适用于从弹球机到相机，机器人，汽车，机架式服务器等各种应用。 ARM不同寻常之处在于它不直接销售处理器，而是授权其他公司构建其处理器，通常作为更大的片上系统的一部分。例如，三星，Altera，Apple和Qualcomm都使用从ARM购买的微架构或内部由ARM授权开发的微架构构建ARM处理器。我们选择专注于ARM，因为它是商业领导者，也因为它的架构很干净，几乎没有什么特异性。我们首先介绍汇编语言指令，操作数位置和常见的编程结构，例如分支，循环，数组操作和函数调用。然后，我们将描述汇编语言如何转换为机器语言，并显示程序如何加载到内存中并执行。

在本章中，我们使用David Patterson和John Hennessy在他们的书《计算机组织和设计》中阐述的四个原则来促进ARM体系结构的设计：（1）规整性支持简单性；（2）加快常见功能；（3）越小的设计越快；（4）好的设计需要好的折衷方法。

#### 6.2 汇编语言

汇编语言是计算机机器语言的人类可阅读表示。每条汇编语言指令都指明了需要完成的操作和操作所处理的操作数。我们通过介绍简单的算术指令来体现如何用汇编语言来写这些操作，接着将定义ARM指令系统中的操作数：寄存器、存储器和常数。

我们假设读者已经熟悉一种高级程序语言，如C、C++或Java。（实际上这些语言在本章的例子中都几乎差不多，在有所不同的时候，我们将使用C语言。）附录C为那些具有较少或者没有编程经验的读者提供C语言介绍。

##### 6.2.1 指令

最常见的计算操作是加法操作。代码示例6.1给出了如何将两个变量b和c相加将结果写入a中。在左边的是高级语言（C、C++或Java），在右边使用ARM汇编语言重写。注意C语言程序语句的最后是一个分号。

代码示例6.1 加法

高级语言代码

{代码}

ARM汇编代码

{代码}

汇编指令的第一部分（add）是助记符（Mnemonic），指明了需要执行的操作。该操作基于源操作数（Source Operands）b和c，结果将写入目的操作数（Destination Operand）a。

代码示例6.2 减法

高级语言代码

{代码}

ARM汇编代码

{代码}

代码示例6.2显示减法指令类似于加法。除了操作码sub以外，指令格式完全与加法指令相同。这种一致的指令格式很好地证明了第一个设计准则：

**设计准则1**：规整性支持简单性

指令中包含固定数目的操作数（在本例中，有2个源操作数和一个目的操作数）将易于编码和硬件处理。更复杂的高级语言代码可以转化为多条ARM指令，如代码示例6.3所示。

代码示例6.3 复杂代码

高级语言代码

{代码}

ARM汇编代码

{代码}

在高级语言例子中，单行注释以//开始直到一行结束，多行注释则由/\*开始，由\*/结束。在ARM汇编语言中仅仅支持单行注释，由分号（;）开始直到一行结束。代码示例6.3中的汇编语言程序需要一个临时变量t来存储中间结果。使用多条汇编指令执行复杂的操作体现了计算机体系结构的第二个设计准则：

**设计准则2**：加快常见功能。

ARM指令集通过仅仅包含简单、常用指令使得常见的情况能较快执行。指令的数目比较少，这使得用于指令操作和操作数译码的硬件比较简单、精练和快捷。更复杂但不常见的操作由多条简单指令序列执行。因此，ARM属于精简指令集计算机（Reduced Instruction Set Computer, RISC）体系结构。具有复杂指令的体系结构，例如Intel的x86，称为复杂指令集计算机（Complex Instruction Set Computer, CISC）。例如，x86中定义的“字符串移动”指令将字符串从内存的一个位置拷贝到另外一个位置。这样的操作需要很多条（很可能上百条）RISC处理器的指令。然而，CISC体系结构中实现复杂指令的缺点在于增加了硬件而且降低了简单指令的执行速度。

RISC体系结构在简化硬件复杂性的同时，还使得指令编码中区分不同指令操作的位数比较少。例如，有64种简单操作的指令集需要log264=6位来编码各种操作。有256种复杂指令的指令集需要log2256=8位编码各种指令。在CISC处理器中，即使复杂指令使用的频率非常小，它们也将增加包括简单指令在内的所有指令的开销，。

##### 6.2.2 操作数：寄存器、存储器和常数

一条指令的操作需要基于操作数。在代码示例6.1中变量a, b, c都是操作数。但是计算机只能处理二进制，而不能处理变量名称。指令需要从一个物理位置中取出二进制数据。操作数可以存放在寄存器或存储器中，也可以作为常数存储在指令自身中。计算机使用不同的位置存放操作数以优化性能和存储容量。存放在指令中的常数或者在寄存器中的操作数访问非常快，但是它们只能包含少量数据。更多的数据需要从存储器访问得到。存储器虽然容量大但是访问速度比较慢。因为ARM（ARMv8之前）体系结构操作32位数据，所以被称为32位体系结构。

**寄存器**

只有快速访问到操作数，指令才能执行比较快。但是存放在存储器中的操作数需要较长时间才能访问到。因此，大多数体系结构定义了一组容量比较小的寄存器用于存放常用的操作数。MIPS体系结构有16个寄存器，称为寄存器集（Register Set）或寄存器文件（Register File）。越少的寄存器访问速度越快，这正验证了第三个设计准则：

**设计准则3**：越小的设计越快。

从书桌上少量相关的书籍中查找信息要远比从图书馆的书库中找要快很多。同样道理，从一个比较少数量的寄存器文件中读取数据要快于从大的存储器中读取。小的寄存器文件往往利用小SRAM阵列（参见5.5.3节）构造。

代码示例6.4显示了带寄存器操作数的ADD指令。 ARM寄存器名称前面带有字母“R”。 变量a，b和c随意放置在R0，R1和R2中。 名称R1发音为“寄存器1”或“R1”或“寄存器R1”。 该指令将R1（b）和R2（c）中包含的32位值相加，并将32位结果写入R0（a）。 代码示例6.5显示了使用寄存器R4来存储b + c的中间计算结果的ARM汇编代码：

代码示例6.4 寄存器操作数

高级语言代码

{代码}

ARM汇编代码

{代码}

代码示例6.5 临时寄存器

高级语言代码

{代码}

ARM汇编代码

{代码}

例6.1 高级语言代码转换为汇编语言代码

转换以下高级语言代码为ARM汇编语言代码。假设变量a—c储存在寄存器R0–R2中，而变量f—j则储存在寄存器R3–R7中。

{代码}

解：该程序使用4句汇编语言指令。

{代码}

**寄存器集**

表6.1列出了16个ARM寄存器中每个寄存器的名称和用法。R0-R12用于存储变量; 在程序调用期间，R0-R3也有特殊用途。R13-R15也称为SP，LR和PC，它们将在本章后面介绍。

常数/立即数

除寄存器操作外，ARM指令还可以使用常量或立即操作数。这些常量称为立即数，因为它们的值可以立即从指令中获得，并且不需要寄存器或存储器访问。代码示例6.6显示了向寄存器添加立即数的ADD指令。在汇编代码中，立即数前面带有＃符号，可以用十进制或十六进制编写。ARM汇编语言中的十六进制常量以0x开头，与C中一样。立即数是无符号的8到12位数字，具有6.4节中描述的特殊编码。

表6.1 ARM寄存器集

|  |  |
| --- | --- |
| 名称 | 用途 |
| R0 | 参数/返回值/临时变量 |
| R1–R3 | 参数/临时变量 |
| R4–R11 | 保存的变量 |
| R12 | 临时变量 |
| R13 (SP) | 堆栈指针 |
| R14 (LR) | 链接指针 |
| R15 (PC) | 程序计数器 |

代码示例6.6 立即操作数

高级语言代码

{代码}

ARM汇编代码

{代码}

代码示例6.7 使用立即数初始化值

高级语言代码

{代码}

ARM汇编代码

{代码}

移动指令（MOV）是初始化寄存器值的有用方法。 代码示例6.7分别将变量i和x初始化为0和4080。 MOV也可以采用寄存器源操作数。 例如，MOV R1，R7将寄存器R7的内容复制到R1中。

**存储器**

如果仅仅以寄存器作为操作数的存储空间，将限制程序中的变量不能超过15个。但是，数据也可以存储在存储器中。与寄存器文件相比，寄存器容量小速度快，而存储器容量大速度慢。所以，频繁使用的变量保存在寄存器中。在ARM体系结构中，指令仅在寄存器上运行，因此存储在存储器中的数据必须先移入寄存器才能进行处理。通过综合使用寄存器和存储器，程序可以以相对较快的速度访问大量的数据。如5.5节所述，存储器组织为数据字的阵列。ARM体系结构采用32位存储器地址和32位数据字长。

ARM使用字节可寻址的内存。 也就是说，内存中的每个字节都有一个唯一的地址，如图6.1（a）所示。 32位字由4个8位字节组成，因此每个字的地址是4的倍数。最高有效字节（MSB）位于左侧，最低有效字节（LSB）位于右侧。 32位字地址和图6.1（b）中的数据值都以十六进制给出。例如，数据0XF2F1AC07存储在存储器第4号地址空间中。按照惯例，存储器图中的低地址在下，高地址在上。

ARM提供加载寄存器指令，LDR，从存储器中读出数据字到寄存器，代码示例6.8表明了如何将内存中的第2号字读入到a（R7）中。在C中，方括号内的数字是索引或字编号，我们将在6.3.6节进一步讨论。LDR指令使用基址寄存器（R5）和偏移量（8）指定存储器地址。 回想一下，每个数据字是4个字节，因此字号1在地址4，字号2在地址8，依此类推。 字地址是字号的四倍。 存储器地址由基址寄存器（R5）的内容和偏移量相加来形成。 ARM提供了几种访问内存的模式，如6.3.6节所述。

图6.1 ARM字节可寻址存储器展示：（a）字节地址和（b）数据

Byte address：字节地址

Word address：字地址

Data：数据

Word number：字号

Word：字

Width：宽度

Bytes：字节

代码示例6.8 读取存储器

高级语言代码

{代码}

ARM汇编代码

{代码}

在代码示例6.8中执行加载寄存器指令（LDR）后，R7保存了值0x01EE2842，这是图6.1中存储在存储器地址8的数值。

ARM使用存储寄存器指令，STR，将寄存器中的数据字写入存储器。 代码示例6.9将来自寄存器R9的值42写入内存字5。

如图6.2所示，字节寻址存储器的组织方式有大端（big-endian）和小端（little-endian）两种形式。两种格式中，一个32位字的最高字节（most significant byte：MSB）在左，最低字节（least significant byte：LSB）在右。两种格式的字地址相同，而且并指向相同的4个字节，唯一不同的只是一个字中不同字节的地址。在大端形式的机器中，第0个字节是从最高字节开始；在小端形式的机器中，第0个字节是从最低字节开始。

代码示例6.9 写入存储器

高级语言代码

{代码}

ARM汇编代码

{代码}

图6.2 大端（big-endian）和小端（little-endian）存储器寻址

Big-endian：大端

Little-endian：小端

Word address：字地址

Byte address：字节地址

IBM公司的PowerPC（在Macintosh计算机中采用）使用大端方式寻址。PC机中使用的Intel公司x86体系结构使用小端方式编址。ARM更喜欢little-endian，但在某些版本中支持双端数据寻址，这允许以任一格式加载和存储数据。选择大端或小端方式完全是任意的，却因此引起了使用大端与小端方式的计算机之间共享数据的麻烦。在本文的例子中，我们在涉及字节顺序时都使用小端方式。

#### 6.3 编程

软件编程语言（如C或Java）称为高级编程语言，因为它们比汇编语言的抽象层次更高。许多高级语言使用算术和逻辑操作、if/else语句、for和while循环、数组索引和过程调用等常见的软件结构。本节将会探讨如何将这些高级语言结构翻译成ARM汇编代码。

##### 6.3.1 数据处理指令

ARM体系结构定义了大量的数据处理指令（在其他架构中通常称为逻辑和算术指令）。这些指令对实现高级语言结构是必须的，因此首先简要介绍它们。附录B提供了ARM指令的总结。

**逻辑指令**

ARM逻辑运算包括AND，ORR（OR），EOR（XOR）和BIC（位清除）。 它们各自在两个源上按位运行，并将结果写入目标寄存器。 第一个源始终是寄存器，第二个源是立即数或另一个寄存器。 另一个逻辑操作MVN（MoVe和Not）在第二个源（立即数或寄存器）上执行按位NOT，并将结果写入目标寄存器。 图6.3显示了对两个源值0x46A1F1B7和0xFFFF0000执行这些操作的示例。 该图显示了指令执行后存储在目标寄存器中的值。

位清除（BIC）指令对于屏蔽位（即，将不用的位清除为0）非常有用。 BIC R6，R1，R2计算R1 AND NOT R2。 换句话说，BIC清除R2中断言的位。 在这种情况下，R1的前两个字节被清除或屏蔽，R1的未屏蔽的底部两个字节0xF1B7被置于R6中。 寄存器位的任何子集都可被屏蔽。

ORR指令可用于组合两个寄存器位。例如，0x347A0000 ORR 0x00072FC=0x347A72FC，即将两个值组合起来。

图6.3 逻辑操作

Assembly code：汇编代码

Source registers：源寄存器

Result：结果

**移位指令**

移位指令左移或右移寄存器中的值，从末尾开始丢弃位。 rotate指令将寄存器中的值旋转最多31位。 我们一般将移位和旋转一起称为移位操作。 ARM移位操作有LSL（逻辑左移），LSR（逻辑右移），ASR（算术右移）和ROR（右旋）。 没有ROL指令是因为可以通过右旋互补量来执行左旋。

在5.2.5节的讨论中， 左移一般将低位补0。但右移可以是逻辑右移（高位补0）或算术右移（高位补符号位）。移位的位数可以是立即数或寄存器。

图6.4显示了当移位的位数为一个立即数时LSL，LSR，ASR和ROR的汇编代码和结果寄存器的值。 R5移动的位数为一个立即数，结果放在目标寄存器中。如5.2.5节所讨论，将一个值左移N位相当于乘以2N。 同理，算术右移N位，相当于除以2N。逻辑移位也用于提取或组合位域。

图6.5显示了移位操作的汇编代码和结果寄存器值，其中移位量保存在寄存器R6中。 该指令使用寄存器移位寄存器寻址模式，其中一个寄存器（R8）移位的位数为第二个寄存器（R6）中保存的量（20）。

图6.4 移位指令，移位量为立即数

Source register：源寄存器

Assembly code：汇编代码

Result：结果

图6.5 移位指令，移位量为寄存器

Source registers：源寄存器

Assembly code：汇编代码

Result：结果

**乘法指令\***

乘法指令有些不同于其他算术指令。两个32位数相乘，产生一个64位乘积。ARM体系结构提供乘法指令，产生32位或64位乘积。 乘法（MUL）将两个32位数相乘，产生32位结果。 MUL R1，R2，R3将R2和R3中的值相乘，并将乘积的最低有效位置于R1中; 乘积的最高有效的32位被丢弃。 该指令对于值小的两个数相乘，且其结果不长于32位非常有用。 UMULL（无符号相乘长整型）和SMULL（有符号相乘长整型）将两个32位数字相乘并生成64位乘积。 例如，UMULL R1，R2，R3，R4执行R3和R4的无符号乘法。乘积的最低有效32位放在R1中，最高有效32位放在R2中。

这些指令中的每一个还具有相乘-累加变体，MLA，SMLAL和UMLAL，它们将乘积添加到正在运行的32位或64位和。 这些指令可以提高矩阵乘法和包含重复乘法和加法的信号处理等应用中的数学性能。

##### 6.3.2 条件标志

如果程序每次只能以相同的顺序运行，那么程序将会很无聊。 ARM指令可选择根据结果是否为零来设置条件标志。然后后续指令有条件地执行，具体取决于条件标志的状态。 ARM条件标志（也称为状态标志）有负（N），零（Z），进位（C）和溢出（V），如表6.2所示。 这些标志由ALU设置（见第5.2.4节），并保存在32位当前程序状态寄存器（CPSR）的前4位，如图6.6所示。

设置状态位的最常用方法是使用比较（CMP）指令，该指令从第一个源操作数中减去第二个源操作数，并根据结果设置条件标志。 例如，如果两个数字相等，则结果为零，并设置Z标志。 如果第一个数字是大于或等于第二个数字的无符号值，则减法将产生进位并设置C标志。

后续指令可以根据标志的状态有条件地执行。 指令助记符后跟一个条件助记符，指示何时执行。 表6.3列出了4位条件字段（cond），条件助记符，名称以及导致指令执行的条件标志的状态（CondEx）。 例如，假设程序执行CMP R4，R5，然后执行ADDEQ R1，R2，R3。 如果R4和R5相等，则该比较设置Z标志，仅当Z标志被设置时，ADDEQ才执行。 cond字段将用于第6.4节中的机器语言编码。

图6.6 当前程序状态寄存器（CPSR）

bits：位

表6.2 条件标志

Flag：标志

Name：名称

Description：描述

Negative：负

Zero：零

Carry：进位

oVerflow：溢出

Instruction result is negative, i.e., bit 31 of the result is 1：指令结果是负的，即结果的第31位是1

Instruction result is zero：指令结果为零

Instruction causes a carry out：指令导致进位

Instruction causes an overflow：指令导致溢出

表6.3 条件助记符

Mnemonic：助记符

Name：名称

Equal：相等

Not equal：不相等

Carry set / unsigned higher or same：设置进位/无符号数更高或相同

Carry clear / unsigned lower：清楚进位/无符号数更低

Minus / negative：减/负

Plus / positive or zero：加/正或零

Overflow / overflow set：溢出/设置溢出

No overflow / overflow clear：没有溢出/清除溢出

Unsigned higher：无符号数更高

Unsigned lower or same：无符号数更低或相同

Signed greater than or equal：有符号数大于或等于

Signed less than：有符号数小于

Signed greater than：有符号数大于

Signed less than or equal：有符号数小于或相等

Always / unconditional：总是/无条件的

or none：或者没有

Ignored：被忽略

当指令助记符后跟“S”时，其他数据处理指令将设置条件标志。例如，SUBS R2，R3，R7将从R3中减去R7，将结果放入R2，并设置条件标志。 附录B中的表B.5总结了每条指令影响哪些条件标志。 所有数据处理指令将根据结果是否是零或者是否设置最高有效位来影响N和Z标志。 ADDS和SUBS也会影响V和C，移位影响C。

代码示例6.10显示了有条件执行的指令。 第一条指令CMP R2，R3无条件执行并设置条件标志。 其余指令有条件地执行，具体取决于条件标志的值。 假设R2和R3包含值0x80000000和0x00000001。 比较指令计算R2-R3 = 0x80000000-0x00000001 = 0x80000000 + 0xFFFFFFFF = 0x7FFFFFFF，产生一个进位（C= 1）。 两个源操作数的符号相反，计算结果的符号与第一个源操作数的符号不同，因此结果溢出（V = 1）。 其余标志（N和Z）为0。ANDHS执行，因为C = 1。EORLT执行，因为N为0且V为1（见表6.3）。 直观上看，ANDHS和EORLT分别执行，因为R2≥R3（无符号）且R2 <R3（有符号）。 ADDEQ和ORRMI不执行，因为R2-R3的结果不为零（即R2≠R3）或负。

图6.7 有符号与无符号比较：HS与GE

Unsigned：无符号

Signed：有符号

代码示例6.10 有条件的执行

ARM汇编代码

{代码}

##### 6.3.3 分支

相对于计算器而言，计算机的优势在于它能做出判断，可以根据不同的输入处理不同的任务。例如，if/else语句，case语句，while循环和for循环，都是根据判断有条件地执行代码。

做出判断的一种方法是使用条件执行来忽略某些指令。 这适用于忽略少量指令的简单if语句，但对于在程序体中包含许多指令的if语句而言，它是浪费的，并且它不足以处理循环。 因此，ARM和大多数其他架构使用分支（branch）指令跳过代码段或重复代码。

程序通常按顺序执行，程序计数器（PC）在每条指令之后递增4以指向下一条指令。 （回想一下，指令长4个字节，ARM是字节寻址架构。）分支指令改变程序计数器。 ARM包括两种类型的分支：简单分支（B）和分支并链接（BL）。 BL用于函数调用，将在6.3.7节中讨论。 与其他ARM指令一样，分支可以是无条件的或有条件的。 在某些体系结构中，分支也称为跳转（jumps）。

代码示例6.11显示了使用分支指令B的无条件分支。当代码到达B TARGET指令时，执行（taken）分支。 也就是说，执行的下一条指令就是名为TARGET的标号（label）之后的SUB指令。

汇编代码使用标号来说明程序中指令的位置。当汇编代码翻译成机器代码时，这些标号将翻译为指令地址（参见6.4.3节）。ARM汇编标号不能是保留字，比如指令助记符。大多数的程序员编程时为了突出标号，只缩进代码而不缩进标号。ARM编译器要求这样做：标号不能缩进，并且指令必须以空格开头。 一些编译器，包括GCC，在标号后要求有冒号。

分支指令可以根据表6.3中列出的条件助记符有条件地执行。 代码示例6.12说明了BEQ的使用，分支取决于相等性（Z = 1）。 当代码到达BEQ指令时，Z条件标志为0（即，R0≠R1），因此不执行分支。 也就是说，执行的下一条指令是ORR指令。

代码示例6.11 无条件分支

ARM汇编代码

{代码}

代码示例6.12 有条件分支

ARM汇编代码

{代码}

##### 6.3.4 条件语句

if语句、if/else语句和case语句是高级程序设计语言常用的条件语句。它们根据一条或多条指令，有条件地执行一块（block）代码。本节将介绍如何把这些高级语言结构翻译成ARM汇编语言。

**if语句**

仅当满足条件时，if语句执行if块（if block）代码。代码示例6.13指出了如何将if语句翻译成ARM汇编代码。

代码示例6.13 IF语句

高级语言代码

{代码}

ARM汇编代码

{代码}

if语句的汇编代码检测与高级语言代码相反的条件。 在代码示例6.13中，高级语言代码检测apples == oranges。汇编代码检测apple！= oranges，如果条件不满足就使用BNE跳过if块。 否则，说明apples == oranges，则不执行分支而执行if块。

因为任何指令都可以有条件地执行，所以代码示例6.13的ARM汇编代码也可以更紧凑地编写，如下所示。

{代码}

条件执行的解决方案更短且更快，因为它涉及少一条指令。 此外，我们将在7.5.3节中看到，分支有时会引入额外的延迟，而条件执行总是很快。 此示例显示了ARM体系结构中条件执行的强大功能。

通常，当一个代码块只有一条指令时，最好使用条件执行而不是分支。随着块变得更长，分支变得有价值，因为它避免了浪费时间获取不会被执行的指令。

**if/else语句**

if/else语句根据条件执行两块代码中的一块。当满足if语句中的条件，执行if块，否则，执行else块。代码示例6.14给出了一个if/else语句的例子。

像if语句一样，if/else语句汇编代码检测的条件与高级语言中相反。在代码示例6.14中，高级语言代码检测apples == oranges。汇编代码检测apples != oranges。如果那个相反的条件为真，BNE跳过if块，执行else块。否则，执行if块，并用一个无条件分支（B）跳过else块。

代码示例6.14 IF/ELSE语句

高级语言代码

{代码}

ARM汇编代码

{代码}

同样，因为任何指令都可以有条件地执行，并且因为if块中的指令不会改变条件标志，所以代码示例6.14的ARM汇编代码也可以更简洁地编写为：

{代码}

**switch/case语句**

switch/case语句根据条件执行多块代码中的一块。如果不能满足条件，则执行default块。一个case语句相当于多个嵌套的if/else语句。代码示例6.15给出了两个同样功能的代码片断。代码根据按下的按钮计算是否从ATM（自动柜员机）取出20美元，50美元或100美元。ARM汇编代码的实现与高级语言代码片断相同。

代码示例6.15 SWITCH/CASE语句

高级语言代码

{代码}

ARM汇编代码

{代码}

##### 6.3.5 循环

循环根据某一条件重复地执行一块代码语句。for循环和while循环是高级程序语言常用的循环结构。本节将介绍如何利用条件分支把它们翻译为ARM汇编语言。

**while循环**

while循环重复地执行一块代码，直至某一条件不再满足。代码示例6.16中的while循环求出满足2­x=128的x值。它循环7次，直到pow=128。

与if/else语句类似，while循环的汇编代码测试的条件与高级语言代码中的相反。如果那个相反的条件为真（在这个例子中，R0 == 128），那么while循环就停止。如果不是（R0≠128），则不执行分支而循环体执行。

代码示例6.16 WHILE循环

高级语言代码

{代码}

ARM汇编代码

{代码}

代码示例6.16中，while循环将pow值与128做比较，如果相等，就退出。否则，它将pow值乘以2（左移1位），递增x，然后分支回到while循环的开始处。

**for循环**

在while循环之前初始化变量，在循环条件中检查该变量，并且每次通过while循环更改该变量是很常见的。 for循环是一种方便的简写，它将初始化，条件检查和变量更改组合在一个地方。for循环的格式是：

for (initialization; condition; loop operation)

statement

代码示例6.17 FOR循环

高级语言代码

{代码}

ARM汇编代码

{代码}

初始化（initialization）代码在for循环之前执行。每一次循环前检查条件（condition）是否满足，如果不满足条件，则退出循环。循环操作（loop operation）在每次循环后执行。

代码示例6.17将数字0加到9。循环变量i初始化为0.然后在每次循环后自增1。只要i小于10，for循环就会执行。请注意，此示例还说明了相对比较。此循环检查<条件，满足就继续，因此汇编代码检查相反的条件>=，以退出循环。

循环对于访问存储在存储器中的大量类似数据特别有用，这将在下面讨论。

##### 6.3.6 存储器

为了便于存储和访问，可以将类似的数据组合在一起形成数组（array）。 数组将其内容存储在内存中的顺序数据地址中。 每个数组元素由一个称为索引（index）的数字标识。 数组中的元素数称为数组的长度（length）。

图6.8显示了存储在内存中的200个元素的分数（scores）数组。 代码示例6.18是一种等级膨胀算法，它为每个分数增加10个点。 请注意，未显示用于初始化score数组的代码。 数组的索引是变量（i）而不是常量，因此我们必须将它乘以4才能将其添加到基址。

ARM可以在单个指令中缩放（乘）索引，将其添加到基址，并从内存加载。 不使用代码示例6.18中的LSL和LDR指令序列，我们可以使用单个指令：

LDR R3, [R0, R1, LSL #2]

R1被缩放（左移两位）然后被添加到基地址（R0）。 因此，内存地址是R0 +（R1×4）。

图6.8 从基址0x14000000开始保存scores[200]的内存

Address：地址

Data：数据

Main memory：主内存

代码示例6.18 用FOR循环访问数组

高级语言代码

{代码}

ARM汇编代码

{代码}

除了扩展索引寄存器之外，ARM还提供偏移量，预索引和后索引寻址，以便为数组访问和函数调用启用密集而高效的代码。 表6.4给出了每种索引模式的示例。 在每种情况下，基址寄存器为R1，偏移量为R2。 可以通过写-R2来减去偏移量。 偏移量也可以是0-4095范围内的立即数，可以加（例如，＃20）或减（例如，＃-20）。

偏移寻址以基址寄存器±偏移量计算地址; 基址寄存器不变。 预索引寻址将地址计算为基址寄存器±偏移量，并将基址寄存器更新为该新地址。 后索引寻址仅将地址计算为基址寄存器，然后在访问存储器后，将基址寄存器更新为基址寄存器±偏移量。 我们已经看到了许多偏移索引模式的例子。 代码示例6.19显示了代码示例6.18中的重写为后索引寻址，消除增加i的ADD指令的for循环。

表6.4 ARM索引模式

Mode：模式

ARM Assembly：ARM汇编代码

Address：地址

Base Register：基址寄存器

Offset：偏移量

Pre-index：预索引

Post-index：后索引

Unchanged：不改变

代码示例6.19 使用后索引的FOR循环

高级语言代码

{代码}

ARM汇编代码

{代码}

**字节和字符**

范围在[-128,127]之间的数可以存储在一个单独的字节中，而不需要一个完整的字。因为英语键盘上按键的数目远远少于256，所以英语字符可以用字节表示。C语言使用char类型来表示字节或字符。

早期的计算机缺乏字节与英语字符之间的标准映射，所以计算机之间交换文本很困难。1963年，美国标准委员会发表了《用于信息交换的美国标准代码》（ASCII码）。它为每个文本字符确定了一个唯一的字节值。表6.5给出了可印刷字符的编码。ASCII码采用十六进制编码。大写字母与小写字母之间相差0x20（32）。

ARM提供加载字节（LDRB），加载有符号字节（LDRSB）和存储字节（STRB）以访问内存中的各个字节。 LDRB对字节进行零扩展，而LDRSB对字节进行符号扩展以填充整个32位寄存器。 STRB将32位寄存器的最低有效字节存储到存储器中的指定字节地址中。 所有这三个指令都在图6.9中说明，基址R4为0。LDRB将存储器地址2处的字节装入R1的最低有效字节，并将剩余的寄存器位填充为0。LDRSB将该字节装入R2，并将该字节符号扩展到寄存器的高24位。 STRB将R3的最低有效字节（0x9B）存储到存储器字节3中; 它用0x9B替换0xF7。 R3的更高有效字节被忽略。

图6.9 加载和存储字节的指令

Memory：存储器

Byte address: 字节地址

Register: 寄存器

Data：数据

表6.5 ASCII编码

Char：字符

space：空格

一个字符序列称为字符串（string）。字符串的长度可变，因此程序设计语言必须提供一种方法来确定字符串的长度或确定字符串的结尾。在C语言中，空字符（0x00）意味着字符串结束。例如，图6.10给出了字符串“Hello!”（0x48 65 6C 6C 6F 21 00）在内存中的存储方式。这个字符串有7个字节，地址从0x1522FFF0到0x1522FFF6，字符串的第一个字符（H = 0x48）存储于最低字节地址（0x1522FFF0）。

图6.10 存储在内存中的字符串“Hello!”

Memory：内存

Word address：字地址

Data：数据

Byte：字节

例 6.2 使用LDRB和STRB来访问字符数组

下述高级语言代码将大小为10的数组中所有小写字母减去32，从而将其转化为大写字母。将此高级语言代码翻译成ARM的汇编语言。注意数组元素之间的地址变化是1个字节而不是4个字节。假定R0已经保存了chararray的基地址。

//高级语言代码

//chararray[10]之前已声明并初始化

int i;

for （i=0; i!=10; i=i+1）

chararray[i]=chararray[i]-32;

解：

; ARM汇编语言代码

; R0 = chararray的基址（之前已初始化）, R1 = i

MOV R1, #0 ;i = 0

LOOP CMP R1, #10 ; i < 10 ?

BGE DONE ; if (i >=10), exit loop

LDRB R2, [R0, R1] ; R2 = mem[R0+R1] = chararray[i]

SUB R2, R2, #32 ; R2 = chararray[i] − 32

STRB R2, [R0, R1] ; chararray[i] = R2

ADD R1, R1, #1 ;i = i + 1

B LOOP ; repeat loop

DONE

##### 6.3.7 函数调用

高级程序语言支持函数（function，也称为过程，procedure或子程序，subroutines）重用经常使用的代码，并使程序更加模块化和可读。函数的输入和输出分别称为参数（arguments）和返回值（return value）。函数将计算返回值并且不会产生其它非预期的不良影响。

当一个函数去调用其他函数时，调用函数（caller）和被调用函数（callee）必须要在参数和返回值上保持一致。ARM系统的惯例是：调用函数在调用之前要将4个参数分别放在R0–R3中，被调用函数在完成之前将返回值放入R0中。遵循这种惯例，即使由不同人所写的调用函数和被调用函数也均知道参数和返回值存在何处，。

被调用函数不能影响调用函数的行为。这意味着被调用函数必须知道当它完成之后要返回到哪里，而且它不能破坏调用函数用到的寄存器和内存。调用函数将返回地址（return address）存在链接寄存器LR中，与此同时它使用分支和链接指令（BL）跳转到被调用函数入口。被调用函数不能覆盖调用函数所需要的任何体系结构状态和内存。具体来说，被调用函数必须保证已保存寄存器(R4–R11, and LR)以及用于存放临时变量的栈（stack）不被修改。

本节将介绍如何调用一个函数并从此被调用函数中返回，同时还将介绍如何访问输入参数和返回值，如何使用栈来存储临时变量。

**程序调用和返回**

ARM使用分支和链接指令（BL）来调用函数，并将链接寄存器移动到PC（MOV PC，LR）以从函数返回。 代码示例6.20显示了调用simple函数的main函数。 main是调用者，simple是被调用者。 调用simple函数时没有输入参数，并且不生成返回值; 它只是返回给调用者。 在代码示例6.20中，指令地址以十六进制的形式在每个ARM指令的左侧给出。

BL（分支和链接）和MOV PC，LR是函数调用和返回所需的两个基本指令。 BL执行两个任务：它将下一条指令的返回地址（BL之后的指令）存储在链接寄存器（LR）中，然后它分支跳转到目标指令。

在代码示例6.20中，main函数通过执行分支和链接指令（BL）来调用simple函数。 BL分支到SIMPLE标号并在LR中存储0x00008024。 simple函数通过执行指令MOV PC，LR立即返回，将LR中的返回地址复制回PC。 然后main函数继续在此地址（0x00008024）执行。

代码示例6.20 simple的函数调用

高级语言代码

{代码}

ARM汇编代码

{代码}

**输入参数和返回值**

代码示例6.20中的simple函数没有从调用函数main中获得输入，也没有返回值输出。根据ARM的习惯，程序使用R0–R3保存输入参数，使用R0保存返回值。在代码示例6.21中，将调用有4个参数的函数diffofsums，并返回一个返回值。result是一个局部变量，我们选择保存在R4中。

代码示例6.21 拥有参数和返回值的程序调用

高级语言代码

{代码}

ARM汇编代码

{代码}

根据ARM的习惯，调用程序main将程序参数从左到右放入输入寄存器R0–R3中。被调用程序diffofsums将返回值存储到返回寄存器R0中。当调用多于4个参数的函数时，多出来的输入参数将放入栈中，这个问题我们下面讨论。

**栈**

栈（stack）是用于存储函数中局部变量的存储单元，当处理器需要更多空间时，栈会扩展（使用更多内存）；当处理器不再需要存在栈中的变量时，栈会缩小（使用较少的内存）。在解释程序栈如何存储临时变量前，我们首先解释栈是怎样工作的。

栈是一个后进先出的队列（last-in-first-out queue）。如同一堆盘子，最后入栈的元素（最上面的盘子）首先被推出。每一个函数需要分配栈空间来存储局部变量，并在函数返回前回收空间。栈顶（the top of stack）是最后分配的空间。然而一堆盘子的空间是向上增长的，ARM栈在内存中是向下增长的。当一个程序需要更多的空间时，栈空间向内存中地址较低的方向扩展。

图6.11显示了堆栈的图片。 堆栈指针SP（R13）是一个普通的ARM寄存器，按照惯例，它指向堆栈的顶部。 指针是内存地址的新名称。 SP指向数据（给出地址）。 例如，在图6.11（a）中，堆栈指针SP保存地址值0XBEFFFAE8并指向数据值0xAB000001。

栈指针（SP）开始于一个较高的内存地址，通过地址的递减来扩展栈空间。图6.11（b）描述了栈如何扩展两个临时存储字。要实现扩展，SP减少8而变成0xBEFFFAE0。两个新扩展的数据字，0x12345678和0xFFEEDDCC，临时存储在栈中。

栈的一个重要应用是保存和恢复用于函数的寄存器。一个函数要计算返回值，但不应产生其它负面影响。具体来说，除了包含返回值的R0，其他任何寄存器都不应该被修改。代码实例6.21中的diffofsums程序破坏了这个规则，因为它修改了R4, R8和R9。如果main在调用diffofsums之前使用了这些寄存器，它们的内容会被被调用函数破坏。

为了解决这个问题，在修改寄存器之前，函数要将寄存器保存在栈中，然后在返回之前从栈中恢复这些寄存器。具体来说，将按照以下函数执行：

1. 创建栈空间来存储一个或多个寄存器的值；
2. 将寄存器的值存储在栈中；
3. 使用寄存器执行程序；
4. 从栈中恢复寄存器的原始值；
5. 回收栈空间。

图6.11 扩展前的堆栈（a）和（b）扩展了两个字后的堆栈

Memory：内存

Address：地址

Data：数据

代码示例6.22给出了diffofsums的改进版本，其存储和恢复了R4, R8和R9。图6.12描述了调用diffofsums之前、之中和之后的栈的情况。栈从0xBEF0F0FC开始。diffofsums通过将栈指针SP减少12以得到3个字存储空间，然后在新分配的空间中存储R4, R8和R9的当前值，接着将执行后续函数，并可以改变这3个寄存器的值。在函数执行的末尾，diffofsums从栈中恢复这些寄存器的值，回收栈空间，并返回，当函数返回时，用R0保存结果，但其他寄存器不受影响：R4, R8, R9和SP中的数值与函数调用之前的数值相同。

函数为自己分配的栈空间称为栈帧（stack frame）。diffofsums的栈框架深度为3个字。模块化的原则告诉我们，每个函数只应访问自己的栈框架而不应去访问其它函数的栈框架。

代码示例6.22 拥有参数和返回值的程序调用

ARM汇编代码

;R4 = result

DIFFOFSUMS

SUB SP, SP, #12 ; 在栈上为3个寄存器创建空间

STR R9, [SP, #8] ; 将R9保存在栈上

STR R8, [SP, #4] ; 将R8保存在栈上

STR R4, [SP] ; 将R4保存在栈上

ADD R8, R0, R1 ; R8 = f + g

ADD R9, R2, R3 ; R9 = h + i

SUB R4, R8, R9 ; result = (f + g) − (h + i)

MOV R0, R4 ; 将返回值放在R0中

LDR R4, [SP] ; 从栈恢复R4

LDR R8, [SP, #4] ; 从栈恢复R8

LDR R9, [SP, #8] ; 从栈恢复R9

ADD SP, SP, #12 ; 释放栈空间

MOV PC, LR ; 返回到调用者

图6.12 栈在调用diffofsums之前（a） 之中（b） 之后（c）的情况

Address：地址

Data: 数据

Stack frame: 栈帧

**加载和存储多个寄存器**

保存和恢复栈上的寄存器是一种常见的操作，ARM提供了为此目的而优化的加载多个和存储多个（Load Multiple and Store Multiple）指令（LDM和STM）。 代码示例6.23使用这些指令重写diffofsums。 该栈保存的信息与前一个示例完全相同，但代码要短得多。

代码示例6.23 保存和恢复多个寄存器

ARM汇编代码

;R4 = result

DIFFOFSUMS

STMFD SP!, {R4, R8, R9} ; 将R4/8/9推到满降序的栈上

ADD R8, R0, R1 ; R8 = f + g

ADD R9, R2, R3 ; R9 = h + i

SUB R4, R8, R9 ; result = (f + g) − (h + i)

MOV R0, R4 ; 将返回值放在R0中

LDMFD SP!, {R4, R8, R9} ; 将R4/8/9从满降序的栈中弹出

MOV PC, LR ; 返回到调用者

LDM和STM分为四种形式，分别为满的和空的降序和升序栈（FD，ED，FA，EA）。 SP!在指令中表示存储相对于栈指针的数据并在存储或加载后更新栈指针。 PUSH和POP分别是STMFD SP!，{regs}和LDMFD SP!，{regs}的同义词，并且是在传统的满降序栈上保存寄存器的首选方法。

**受保护的寄存器**

代码实例6.22和6.23假定所有已使用的寄存器(R4, R8, 和 R9)必须被保存和恢复。如果调用函数不用这些寄存器，对它们的保存和恢复就是无用的操作。为了避免这种无用的操作，ARM将寄存器划分为受保护（preserved）类型和不受保护（nonpreserved）类型。受保护寄存器包括 R4–R11，不受保护寄存器有R0–R3和R12。SP 和LR (R13 和R14)也必须是受保护的。一个函数必须保存和恢复任何需要使用的受保护寄存器，但是可以随意改变不受保护寄存器。

代码示例6.24描述了对diffofsums的进一步修改：只将R4保存在栈中。它还介绍了首选的PUSH和POP同义词。 当不再需要这些参数时，代码重用非保留参数寄存器R1和R3来保存中间和。

代码示例6.24 减少受保护寄存器的数量

ARM汇编代码

;R4 = result

DIFFOFSUMS

PUSH {R4} ; 将R4保存在栈上

ADD R1, R0, R1 ; R1 = f + g

ADD R3, R2, R3 ; R3 = h + i

SUB R4, R1, R3 ; result = (f + g) − (h + i)

MOV R0, R4 ; 将返回值放在R0中

POP {R4} ; 将R4从栈中弹出

MOV PC, LR ; 返回到调用者

注意当一个函数调用另一个函数时，前者称为调用函数，后者称为被调用函数。被调用函数必须保存和恢复它要用到的受保护寄存器，被调用函数有可能改变任何不受保护寄存器，因此如果调用函数需要其不受保护寄存器中有效数据不被改变，那么它需要在函数调用之前需要保存不受保护寄存器，而且还需要在调用之后恢复这些寄存器。这种情况下，受保护寄存器也可以称为被调用者保存的（callee-save），不受保护寄存器称为调用者保存的（caller-save）。

表6.6总结了哪些寄存器是受保护的，R4–R11常用于保存函数中的局部变量，所以它们必须被保护。LR也是要被保护的，这样才能使得函数知道返回到哪里。

表6.6 受保护和不受包含寄存器

Preserved: 受保护寄存器

Unpreserved: 不受保护寄存器

Saved registers: 保存寄存器

Return Address：返回地址

Stack Pointer：栈指针

Statck above the stack pointer: 栈指针以上的空间

Temporary register: 临时寄存器

Argument register: 参数寄存器

Current Program Status Register: 当前程序状态寄存器

Stack below the stack poiner: 栈指针以下的空间

R0–R3和R12用于在向局部变量赋值前保存临时结果，这些计算结果一般在函数调用之前完成，所以它们不用受保护，调用函数一般不需要保存它们。

R0–R3经常在被调用的函数中被覆盖，因此，如果一个被调用的函数返回之后，调用函数的执行取决于它自身参数时，R0–R3需要由调用函数来保存。R0不用被保护，因为被调用函数将返回结果放入这些寄存器中。回想一下，当前程序状态寄存器（CPSR）保存条件标志。 它在整个函数调用中不是受保护的。

栈指针之上的栈空间要自动保护起来，被调用函数不向$sp之上的内存地址写入数据，这样就使得任何函数不会去修改栈帧。栈指针自身是受保护的，这是因为被调用函数在返回之前需要回收自己的栈空间，栈空间的大小为函数结束时的地址减去函数开始时$sp保存的值。

精明的读者或一个优化的编译器可能会注意到本地变量result直接返回而不用于其他任何地方。 因此，我们可以消除这个变量并将其简单地存储在返回寄存器R0中，从而无需推入和弹出R4并将result从R4移动到R0。 代码示例6.25显示了这个进一步优化的diffofsums。

代码示例6.25 经过优化的diffofsums函数调用

ARM汇编代码

DIFFOFSUMS

ADD R1, R0, R1 ; R1 = f + g

ADD R3, R2, R3 ; R3 = h + i

SUB R0, R1, R3 ; return (f + g) − (h + i)

MOV PC, LR ; 返回到调用者

**非叶子函数调用**

不用调用其他函数的函数称为叶子（leaf）函数，如diffofsums函数。需要调用其他函数的函数叫做非叶子（nonleaf）函数。如同前面提到的，非叶子函数一般会更复杂，因为在调用其他函数之前，它们需要把不受保护的寄存器保存到栈中，调用完之后再恢复这些寄存器。具体来说：

调用函数保存规则：在函数调用之前，调用函数必须保存调用后所需的任何不受保护的寄存器（R0-R3和R12）。 在调用之后，它必须在使用它们之前恢复这些寄存器。

被调用函数保存规则：在被调用函数干扰任何受保护的寄存器（R4-R11和LR）之前，它必须保存这些寄存器。 在它返回之前，它必须恢复这些寄存器。

代码示例6.26演示了一个非叶子函数f1和一个叶子函数f2，它包括所有必要的寄存器保存和保护。假设f1在R4中保存i，在R5中保存x。 f2在R4中保存r。 f1使用受保护的寄存器R4，R5和LR，因此它最初根据被调用函数保存规则将它们压入栈。它使用R12来保存中间结果（a-b），这样就不需要为此计算保护另一个寄存器。在调用f2之前，f1根据调用函数保存规则将R0和R1推入到栈中，因为这些是f2可能会改变的非受保护的寄存器，并且f1在调用后仍然需要它们。虽然R12也是f2可以重写的非受保护的寄存器，但f1不再需要R12因此不必保存它。 f1然后将参数放进R0传递给f2，进行函数调用，并将结果放在R0里。然后f1恢复R0和R1，因为它仍然需要它们。当f1完成时，它将返回值放入R0，恢复受保护的寄存器R4，R5和LR，并返回。 f2根据被调用函数保存规则保存和恢复R4。

代码示例6.26 非叶子函数调用

高级语言代码

{代码}

ARM汇编代码

; R0 = a, R1 = b, R4 = i, R5 = x

F1

PUSH {R4, R5, LR} ; 保存f1使用的受保护寄存器

ADD R5, R0, R1 ; x = (a + b)

SUB R12, R0, R1 ; temp = (a − b)

MUL R5, R5, R12 ; x = x \* temp = (a + b) \* (a − b)

MOV R4, #0 ;i = 0

FOR

CMP R4, R0 ; i < a?

BGE RETURN ; 不满足i<a: 退出循环

PUSH {R0, R1} ; 保存非受保护寄存器

ADD R0, R1, R4 ; 参数是 b + i

BL F2 ; 调用f2(b+i)

ADD R5, R5, R0 ; x = x + f2(b+i)

POP {R0, R1} ; 恢复非受保护寄存器

ADD R4, R4, #1 ; i++

B FOR ; 循环继续

RETURN

MOV R0, R5 ; 返回值是 x

POP {R4, R5, LR} ; 恢复受保护寄存器

MOV PC, LR ; return from f1

; R0 = p, R4 = r

F2

PUSH {R4} ; 保存f2使用的受保护寄存器

ADD R4, R0, 5 ;r = p + 5

ADD R0, R4, R0 ; 返回值是 r + p

POP {R4} ; 恢复受保护寄存器

MOV PC, LR ; 从f2返回

图6.13显示了执行f1期间的栈。 栈指针最初从0xBEF7FF0C开始。

图6.13 栈在函数调用之前（a） 执行f1期间（b） 执行f2期间（c）的情况

Address：地址

Data: 数据

f1’s stack frame: f1的栈帧

f2’s stack frame: f2的栈帧

**递归函数调用**

递归（recursive）函数是调用自己的一个非叶子函数。递归函数同时表现为调用函数和被调用函数，并且必须同时保存受保护和非受保护寄存器。例如，阶乘函数可以使用一个递归函数来描述。回忆阶乘函数为factorial（n）=n×（n-1）×（n-2）×…×2×1。其可以递归地写成factorial（n）= n\*factorial（n-1），如代码示例6.27所示。1的阶乘还是1。代码示例6.27描述了阶乘函数的递归写法。为了方便地标明函数地址，假定函数的开始地址为0x8500。

根据被调用函数保存规则，factorial是一个非叶子函数，必须保存LR。 根据调用函数保存规则，factorial在调用自身后需要n，因此必须保存R0。 因此，它在开始时将两个寄存器压入栈。 然后它检查n≤1。如果是，它将返回值1放在R0中，恢复栈指针，然后返回给调用者。 在这种情况下，它不必重新加载LR和R0，因为它们从未被修改过。 如果n> 1，则函数递归调用factorial（n - 1）。 然后它从栈中恢复n的值和链接寄存器（LR），执行乘法运算，并返回该结果。 请注意，该函数巧妙地将n恢复在R1中，以便不覆盖返回的值。 乘法指令（MUL R0，R1，R0）将n（R1）与返回值（R0）相乘，并将结果存入R0。

代码示例6.27 factorial的递归调用

高级语言代码

{代码}

ARM汇编代码

0x8500 FACTORIAL PUSH {R0, LR} ; 将n和LR推入栈中

0x8504 CMP R0, #1 ; R0 <= 1?

0x8508 BGT ELSE ; 不满足: 分支到 else

0x850C MOV R0, #1 ; 否则, 返回1

0x8510 ADD SP, SP, #8 ; 恢复SP

0x8514 MOV PC, LR ; 返回

0x8518 ELSE SUB R0, R0, #1 ;n = n − 1

0x851C BL FACTORIAL ; 递归调用

0x8520 POP {R1, LR} ; 弹出n (到R1中) 和LR

0x8524 MUL R0, R1, R0 ; R0 = n \* factorial(n − 1)

0x8528 MOV PC, LR ; 返回

图6.14描述了执行factorial（3）时栈的情况。为了方便说明，我们显示SP的初始值指向0xBEFF0FF0，如图6.14（a）所示。函数创建两个字的栈空间来保存n (R0)和LR。在第一次函数调用时，factorial将R0（R0中保存着n=3）保存在0xBEFF0FE8中，将LR保存在0xBEFF0FEC中，如图6.14（b）所示。然后函数将n改为2并且递归调用factotial（2），LR保存着0xBC。在第二次函数调用时，fuctorial将R0（R0中保存着n=2）保存在0xBEFF0FE0中，将LR保存在0xBEFF0FE4中。这次我们知道LR中存储了0x8520。然后函数将n改为1并且递归调用factorial（1）。在第三次函数调用时，fuctorial将R0（R0中保存着n=1）保存在0xBEFF0FD8中，将LR保存在0xBEFF0FDC中。这次LR存储的还是0x8520。fuctorial的第三次递归返回1且保存在R0中，而且在返回第二次调用之前回收栈空间。第二次调用将n恢复为2（到R1中），将LR恢复为0x8520（LR中已经是这个值了），然后回收栈帧，返回R0=2x1=2到第一次调用之处。第一次调用将n恢复为3（到R1中），将LR恢复为调用函数的返回地址，然后回收栈帧，返回R0=3x2=6。图6.14（c）描述了在递归调用函数返回时栈的情况。当factorial返回到调用函数时，栈指针指向它的初始位置（0xBEFF0FF0），指针之上栈空间的内容没有变化，而且所有受保护的寄存器保存的是它们的初始值，R0保存的是返回值6。

图6.14 栈：（a）调用前 （b）调用中 （c）调用factorial函数（n=3）之后

Address：地址

Data: 数据

**附加参数和局部变量\***

函数的参数可能多于4个并且可能包含太多局部变量以至于无法保存在受保护的寄存器中。可以用栈存储这些信息。依照ARM的惯例，如果一个函数有4个以上的参数，前4个参数像往常一样存储在参数寄存器中，另外的参数使用栈指针上面的空间保存在栈中。调用函数（caller）必须扩展栈空间来满足另外的参数，图6.15（a）描述了调用多于4个参数的函数时栈的情况。

一个函数也可以定义局部变量或数组，局部变量在函数内部定义并且只能在函数内部有效。局部变量存储在R4–R11中。如果有许多局部变量，它们也可以存储在这个函数的栈空间中。另外，局部数组也存储在栈中。

图6.15（b）给出了被调用函数的栈帧结构。栈帧保存了暂时的寄存器和链接寄存器（如果由于后续的函数调用需要保存它们），还有函数要修改的已保存寄存器。此外，它还存储了局部数组和剩余的局部变量。如果被调用函数有4个以上的参数，可以从调用函数的栈帧中找到它们。访问额外的输入参数是一种特殊情况，在这种情况下函数可以访问不属于自己栈帧中的数据。

图6.15 栈的使用情况：（a）调用前（b）调用后

Additional arguments: 额外调用参数

Stack frame: 栈帧

If needed: 如果需要

Local variables or arrays: 局部变量或数组

#### 6.4 机器语言

汇编语言方便人们阅读。然而数字电路只能理解0和1，需要将汇编语言写的程序从助记符号翻译成仅使用0和1表示的机器语言（machine language）。本节介绍ARM机器语言以及在汇编语言和机器语言之间进行转换的繁琐过程。

ARM使用32位指令。这里再次强调，规整性支持简单性，最规整的选择是以将指令编码成储存在存储器中的字。尽管一些指令可能不需要所有32位的编码，但变长度指令将增加太多的复杂性。简单化也鼓励使用单指令格式，但是过于简单化将产生太多限制。但是，这个问题允许我们介绍最后一条设计原则：

**设计准则4**：好的设计需要好的折衷方法

ARM定义了三种主要指令格式：数据处理，存储器和分支。 这种少量的格式允许指令之间的一些规整性，因此允许更简单的解码器硬件，同时还满足不同的指令需求。 数据处理指令具有第一源寄存器，第二源是立即数或寄存器，可能是移位的，以及目标寄存器。 数据处理格式对这些第二源操作数有几种变体。 存储器指令有三个操作数：基址寄存器，偏移量，可以是立即数或可选移位地寄存器的寄存器，以及作为LDR上的标操作数和STR上另一个源操作数的寄存器。 分支指令采用一个24位立即数分支偏移量。 本节讨论这些ARM指令格式，并说明它们如何编码为二进制。 附录B提供了所有ARMv4指令的快速参考。

##### 6.4.1 数据处理指令

数据处理指令格式是最常见的。 第一个源操作数是寄存器。 第二源操作数可以是立即数或可选移位的寄存器。 第三个寄存器是目标。 图6.16显示了数据处理指令格式。 32位指令有六个字段：cond，op，funct，Rn，Rd和Src2。

图6.16数据处理指令格式

Data-processing：数据处理

bits：位

指令执行的操作编码在突出显示的蓝色字段中：op（也称为opcode或操作码）和func或功能代码; cond字段根据第6.3.2节中描述的标志对条件执行进行编码。 回想一下，对于无条件指令，cond = 11102。 数据处理指令的op是002。

操作数在三个字段中编码：Rn，Rd和Src2。 Rn是第一个源寄存器，Src2是第二个源; Rd是目标寄存器。

图6.17显示了funct字段的格式以及数据处理指令的Src2的三种变体。 funct有三个子字段：I，cmd和S。当Src2是立即数时，I位为1。 当指令设置条件标志时，S位为1。 例如，SUBS R1，R9，＃11的S = 1。cmd表示特定的数据处理指令，如附录B中的表B.1所示。例如，对于ADD，cmd为4（01002），对于SUB，cmd为2（00102） 。

Src2编码的三种变体允许第二源操作数是（1）立即数，（2）可选移常数（shamt5）位的寄存器（Rm），或（3）由另一寄存器（Rs）指定移位数的移位寄存器（Rm））。 对于后两种Src2编码，sh编码要执行的移位类型，如表6.8所示。

数据处理指令有一个不寻常的立即数表示方法，涉及一个8位无符号立即数，imm8和4位的旋转，rot。 imm8向右旋转2×rot以创建一个32位常数。 表6.7给出了8位立即0xFF的示例旋转和得到的32位常数。 这种表示是有价值的，因为它允许许多有用的常数，包括任何2的小倍数幂，被打包成少量的比特。 6.6.1节描述了如何生成任意32位常数。

图6.17显示funct字段和Src2变体的数据处理指令格式

Data-processing：数据处理

Immediate：立即数

Register：寄存器

Register-shifted Register：寄存器移位的寄存器

表6.7 立即数旋转和imm8 = 0xFF产生的32位常数

32-bit Constant：32位常数

图6.18显示了当Src2是寄存器时ADD和SUB的机器代码。 从汇编代码转换为机器代码的最简单方法是写出每个字段的值，然后将这些值转换为二进制值。 将位分组为四个块以转换为十六进制以使机器语言表示更紧凑。 请注意目标是汇编语言指令中的第一个寄存器，但它是机器语言指令中的第二个寄存器字段（Rd）。 Rn和Rm分别是第一和第二源操作数。 例如，汇编指令ADD R5，R6，R7有Rn = 6，Rd = 5和Rm = 7。

图6.19显示了带有立即数和两个寄存器操作数的ADD和SUB的机器代码。 同样，目标是汇编语言指令中的第一个寄存器，但它是机器语言指令中的第二个寄存器字段（Rd）。 ADD指令的立即数（42）可以用8位编码，因此不需要旋转（imm8 = 42，rot = 0）。 但是，SUB R2，R3,0xFF0的立即数不能使用imm8的8位直接编码。 而是这样，imm8是255（0xFF），它向右旋转28位（rot = 14）。 最容易的解释就是记住向右旋转28位相当于32-28 = 4位的左旋转。

图6.18有三个寄存器操作数的数据处理指令

Assembly Code：汇编代码

Field Values：字段的值

Machine Code：机器代码

图6.19有一个立即数和两个寄存器操作数的数据处理指令

Assembly Code：汇编代码

Field Values：字段的值

Machine Code：机器代码

移位也是数据处理指令。 回想一下第6.3.1节，可以使用5位立即数或寄存器对移位量进行编码。

图6.20显示了具有立即数移位量的逻辑左移（LSL）和右旋（ROR）的机器代码。 对于所有移位指令，cmd字段是13（11012），并且移位字段（sh）编码要执行的移位类型，如表6.8所示。 Rm（即R5）保存要移位的32位值，shamt5给出要移位的位数。 移位的结果放在Rd中。 Rn未使用且应为0。

图6.21显示了LSR和ASR的机器代码，其移位量以Rs（R6和R12）的最低有效8位进行编码。 如前所述，cmd为13（11012），sh对移位类型进行编码，Rm保存要移位的值，移位后的结果放在Rd中。 该指令使用寄存器移位寄存器寻址模式，其中一个寄存器（Rm）移位第二个寄存器（Rs）中保存的量。 因为Rs的最低有效8位被使用了，所以Rm最多可以移255位。 例如，如果Rs保持值0xF001001C，则移位量为0x1C（28）。 逻辑移位超过31位会将所有位推出并产生全0。 旋转是循环的，因此旋转50位相当于旋转18位。

表6.8 sh字段编码

Instruction：指令

Operation：操作

Logical shift left：逻辑左移

Logical shift right：逻辑右移

Arithmetic shift right：算术右移

Rotate right：右旋

图6.20 具有立即数移位量的移位指令

Assembly Code：汇编代码

Field Values：字段的值

Machine Code：机器代码

图6.21 具有寄存器移位量的移位指令

Assembly Code：汇编代码

Field Values：字段的值

Machine Code：机器代码

##### 6.4.2 存储器指令

存储器指令使用与数据处理指令类似的格式，具有相同的全部六个字段：cond，op，funct，Rn，Rd和Src2，如图6.22所示。 但是，存储器指令使用不同的funct字段编码，有两种Src2变体，而且op为012。Rn是基址寄存器，Src2保存偏移量，Rd是加载操作中的目标寄存器或存储操作中的源寄存器。 偏移量是12位无符号立即数（imm12）或可选移常数位（shamt5）的寄存器（Rm）。 funct由六个控制位组成：I，P，U，B，W和L。I（立即数）和U（加）位根据表6.9确定偏移是立即数还是寄存器以及是否应该加或减。 P（预索引）和W（写回）位根据表6.10指定索引模式。 L（加载）和B（字节）位根据表6.11指定存储器操作的类型。

图6.22 LDR，STR，LDRB和STRB的存储器指令格式

Memory：存储器

Immediate：立即数

Register：寄存器

表6.9 存储器指令的偏移量类型控制位

Bit：位

Meaning：含义

Immediate offset in Src2：在Src2中的立即数偏移量

Register offset in Src2：在Src2中的寄存器偏移量

Subtract offset from base：从基地址中减去偏移量

Add offset to base：基地址加上偏移量

表6.10 存储器指令的索引模式控制位

Index Mode：索引模式

Post-index：后索引

Not supported：不支持

Offset：偏移量

Pre-index：预索引

表6.11 存储器指令的存储器操作类型控制位

Instruction：介绍

例6.3 将存储器指令翻译为机器语言

将以下汇编语言语句翻译为机器语言。

STR R11, [R5], #-26

解：STR是一个存储器指令，因此它的op为012.。根据表6.11，对于STR，L = 0且B = 0。 该指令使用后索引，因此根据表6.10，P = 0和W = 0。从基地址中减去立即数偏移量，因此I = 0且U = 0。图6.23显示了每个字段和机器代码。 因此，机器语言指令是0xE405B01A。

图6.23 例6.3的存储器指令的机器代码

Assembly Code：汇编代码

Field Values：字段的值

Machine Code：机器代码

##### 6.4.3 分支指令

分支指令使用单个24位带符号立即数操作数imm24，如图6.24所示。 与数据处理和存储器指令一样，分支指令以4位条件字段和2位op开始，即102。funct字段仅为2位。 对于分支，funct的高位总是1。 低位L表示分支操作的类型：BL为1，B为0。剩余的24位二进制补码imm24字段用于指定相对于PC + 8的指令地址。

代码示例6.28显示了小于时分支（BLT）指令的使用，图6.25显示了该指令的机器代码。 分支目标地址（BTA）是在执行分支时要执行的下一条指令的地址。 图6.25中的BLT指令的BTA为0x80B4，即THERE标号的指令地址。

24位立即数字段给出了BTA和PC + 8之间的指令数（超过分支两条指令）。 在这种情况下，BLT的立即数字段（imm24）中的值为3，因为BTA（0x80B4）超过PC + 8（0x80A8）三条指令。

图6.24 分支指令格式

Branch：分支

代码示例6.28 计算分支目标地址

ARM汇编代码

0x80A0 BLT THERE

0x80A4 ADD R0, R1, R2

0x80A8 SUB R0, R0, R9

0x80AC ADD SP, SP, #8

0x80B0 MOV PC, LR

0x80B4 THERE SUB R0, R0, #1

0x80B8 ADD R3, R3, #0x5

图6.25 小于时分支（BLT）的机器代码

Assembly Code：汇编代码

Field Values：字段的值

Machine Code：机器代码

处理器通过对24位立即数进行符号扩展，将其向左移2位（为了将字转换为字节）并将其添加到PC + 8来从指令中计算BTA。

例6.4 计算相对PC寻址的立即数字段

计算立即数字段并显示在下面的汇编程序中的分支指令的机器代码。

0x8040 TEST LDRB R5, [R0, R3]

0x8044 STRB R5, [R1, R3]

0x8048 ADD R3, R3, #1

0x8044 MOV PC, LR

0x8050 BL TEST

0x8054 LDR R3, [R1], #4

0x8058 SUB R4, R3, #9

解：图6.26显示了分支和链接指令（BL）的机器代码。 其分支目标地址（0x8040）在PC + 8（0x8058）六条指令之后，因此立即数为-6。

图6.26 BL机器代码

Assembly Code：汇编代码

Field Values：字段的值

Machine Code：机器代码

##### 6.4.4 寻址模式

本节总结了用于寻址指令操作数的模式。 ARM使用四种主要模式：寄存器，立即，基址和PC相对寻址。 大多数其他架构提供类似的寻址模式，因此了解这些模式可以帮助您轻松学习其他汇编语言。 寄存器和基址寻址具有下面描述的几个子模式。 前三种模式（寄存器，立即和基址寻址）定义了读和写操作数的模式。 最后一种模式（PC相对寻址）定义了写程序计数器（PC）的模式。 表6.12总结并给出了每种寻址模式的示例。

表6.12 ARM操作数寻址模式

Operand Addressing Mode：操作数寻址模式

Register：寄存器

Register-only：仅寄存器

Immediate-shifted register：立即数移位的寄存器

Register-shifted register：寄存器移位的寄存器

Immediate：立即

Base：基址

Immediate offset：立即数偏移量

Register offset：寄存器偏移量

Immediate-shifted register offset：立即数移位的寄存器偏移量

PC-Relative：PC相对

Example：示例

Description：描述

Branch to LABEL1：分支到LABEL1

数据处理指令使用寄存器或立即寻址，其中第一个源操作数是寄存器，第二个是寄存器或立即数。 ARM允许第二个寄存器可选地以立即数或第三个寄存器中指定的数量为移位数进行移位。 存储器指令使用基址寻址，其中基址来自寄存器，偏移量来自立即数，寄存器或立即数移位的寄存器。 分支指令使用PC相对寻址，其通过PC + 8加上偏移量来计算分支目标地址。

##### 6.4.5 解释机器语言代码

要解释机器语言，必须解密每个32位指令字的字段。 不同的指令使用不同的格式，但所有格式都以4位条件字段和2位op开始。 最好是从op开始看。 如果是002，则该指令是数据处理指令; 如果是012，则该指令是存储器指令; 如果它是102，那么它是一个分支指令。 基于此，可以解释其余字段。

例6.5 将机器语言翻译为汇编语言

将以下机器语言代码翻译成汇编语言。

0xE0475001

0xE5949010

解：首先，我们用二进制表示每条指令，并查看位27:26以找到每条指令的op，如图6.27所示。 op字段是002和012，分别表示数据处理和存储器指令。 接下来，我们来看看每条指令的funct字段。

数据处理指令的cmd字段是2（00102）并且I位（位25）是0，表示它是具有寄存器Src2的SUB指令。 Rd为5，Rn为7，Rm为1。

存储器指令的funct字段是0110012。B = 0且L = 1，因此这是LDR指令。 P = 1且W = 0，表示偏移寻址。 I = 0，因此偏移量是立即数。 U = 1，因此加上偏移量。 因此，它是一个加载寄存器指令，带立即数偏移量，并将其添加到基址寄存器。 Rd为9，Rn为4，imm12为16。图6.27显示了这两条机器指令的汇编代码。

图6.27 机器代码到汇编代码的转换

Assembly Code：汇编代码

Field Values：字段的值

Machine Code：机器代码

##### 6.4.6 程序存储

用机器语言编写的程序是一个表示指令的32位数序列。如同其他二进制数，这些指令存放在存储器中。这就是程序存储（stored program）的概念，也是计算机如此强大的一个关键原因。运行一个新的程序时，我们不需要花费大量的时间和精力对硬件进行重新装配或重新布线，只需要将一个新的程序写入存储器。程序存储提供了通用（general purpose）计算能力，而不是特定的硬件。在这种方式下，计算机仅仅只是改变存储的程序就可以运行计算器、文字处理程序、影音播放器等多种应用程序。

程序中的指令从存储器中找到或取出（fetch），然后由处理器执行。即使是大型的复杂程序也可以简化为仅仅包括简单的存储器读和指令运行的动作序列。

图6.28 程序存储

Assembly Code：汇编代码

Machine Code：机器代码

Stored program：存储的程序

Address：地址

Instructions：指令

Main memory：主内存

图6.28说明了机器指令是怎样存储在存储器中的。在ARM程序中，指令一般从低地址开始存储，在本例子中从0x00008000开始。记住ARM存储器地址是字节寻址的，所以32位（4字节）指令地址每次增加4个字节而不是1个字节。

运行（execute）程序时，处理器从存储器中顺序地取指令。然后，用数字电路硬件解码和执行这些取回的指令。当前指令地址存储在一个称为程序计数器（program counter, PC）的32位寄存器中，也就是寄存器R15。由于历史原因，对PC的读取将返回当前指令的地址加8。

为了运行图6.28所示的程序，操作系统将PC初始化为地址0x00008000。处理器将这个存储器地址的指令读出，并执行指令0xE3A01064 (MOV R1, #100)。然后，然后处理器将PC增加4，变为0x00008004接着取出并执行该地址的指令，并循环执行上述步骤。

微处理器的体系结构状态（architectural state）保存了程序的状态。对于ARM，体系结构状态包括寄存器文件和状态寄存器。如果操作系统（OS）在程序运行的某时刻保存了其体系结构状态，就可以中断该程序，做些别的事情。然后恢复原先的状态，被打断的程序又能够继续正确执行，而不知道它曾经被打断过。我们在第七章构建一个微处理器时，体系结构状态也十分重要。