第 1 章
概述

1.1 记法

本书既不是普通的数学算式教程，也不是单纯的电脑程序算法手册，它讲的是“计算机算术”（computer arithmetic），参与运算的数是长度固定的位串与位元数组[㊀]。计算机算术中的表达式与普通的数学表达式近似，不同之处在于其中的变量指代的是 CPU 寄存器中的内容，而且计算机算术表达式的值是一串不具备特定符号性的位元。在此类表达式中，操作符可能会以不同方式解读其操作数，例如“比较操作符”（comparison operator）有时会将其操作数当成有正负号的二进制整数，有时却把它视为无符号的二进制整数。为免混淆，本书所列计算机算式以不同的符号来区分上述两种情况。

计算机算式与数学算式的主要差别在于：无论是加减法还是乘法，计算机算式的结果总是要跟 2 的 n 次方取模，n 是指当前机器的字长[㊁]。此外，计算机算式的运算种类也远多于数学算式：除了基本的四则运算外，还有逻辑与、异或、比较、左移，等等。

如未特别指明，则默认字长为 32 位，且带符号的整数均以“2 补码”[㊂]形式表示。

㊀ bit vector，即 bit array，也写作 bitmap、bitset、bit string 等，是由若干位元排列而成的数组，又叫“位向量”、“位矢量”等，详情参见：https://en.wikipedia.org/wiki/Bit_array。译文在不致混淆时，均以“位元数组”称之。——译者注

㊁ word size，就是字组中所含的位元个数，也称 word length、word width。在不致混淆的情况下，“字长”、“字宽”均指这一概念。——译者注

㊂ two's complement，也叫“2 的补码”、“二补数”，是一种用二进制表示带符号数字的方式。整数和零的补码表示法与其二进制写法相同，只是左方要补足 0，而要表示负数，则需先将其绝对值按位取反，也就是求绝对值的一补数（也称反码），然后再加 1。例如用 8 位二进制表示数字时，5 可以表示为 0000 0101，而要表示 −5，则需先求其绝对值 5，写成二进制形式 0000 0101，然后对其按位取反得到 1111 1010，再加 1。最终的结果 1111 1011就是 −5 的补码表示。详情参见：https://zh.wikipedia.org/wiki/补码。带符号整数与无符号整数的英文分别为 signed integer 与 unsigned integer，其中“带符号”与“无符号”是计算机处理二进制数的两种不同方式。前者会根据最高有效位来判定数字的正负，0 为非负，1 为负，而后者则一律将其视为非负数。以上面的 1111 1011 来说，若视为带符号整数，则其值为 −5（因最高位为 1，故是负数。先将其按位取反得 0000 0100，再加 1 得 0000 0101，即十进制的 5，再添上负号得 −5），若视为无符号整数，则其值为 251（最高位的 1 不再表示负数，而当做 128 来算）。详情参见：https://zh.wikipedia.org/wiki/有符号数处理。——译者注

计算机算式与数学算式的书写方式相同，只是其中指代CPU寄存器中内容的变量会以粗体标出。为了和位元数组运算的通则一致，我们把电脑中的字组也视为由一串位元构成的数组。若某常量表示CPU寄存器中的值，那么它也会以粗体字出现。（这种情况在位元数组运算中找不到对应物，如果要在位元数组算式里书写常量，只能逐个列出其中的每个位元。）然而常量如果作为shift等指令的操作数，则不加粗。

对于“+”这样的操作符，若其操作数为粗体，则表明执行的是计算机加法，也就是“向量加法”，否则意味着执行的是纯数字加法。如果操作数未加粗，则其对应的操作符就是纯数学运算中的含义。要是我们想拿原来做计算机运算的粗体 $\boldsymbol{x}$ 值做数学运算的话，那就会把它写成不加粗的 x，其符号性应该能够从上下文中推出。假如 $\boldsymbol{x}=$ **0x8000 0000**，$\boldsymbol{y}=$**0x8000 0000**，那么在做带符号的整数运算时，$\boldsymbol{x}=\boldsymbol{y}=-2^{31}$，$x+y=-2^{32}$，而 $\boldsymbol{x}+\boldsymbol{y}=\boldsymbol{0}$[㊀]。此处的 **0x8000 0000** 是用十六进制表示的位串，它最左边的位元是“1”，后面跟着31个“0”。

位元的序号从右侧算起，最右方的位元（也就是最低有效位）叫做0号位元。术语“位”、“半字节”、“字节”、“半字组”、“字组”、“双字”[㊁]所对应的位元数量分别是1、4、8、16、32、64。

简短的代码段用的都是计算机算式，并以左箭头表示赋值操作，偶尔还会用if语句。在这种情况下，它只是以一种与电脑平台无关的方式编写汇编语言代码罢了。

过长或过于复杂的计算机算式则用C语言来写，其代码遵循ISO 1999标准[㊂]。

完整描述C语言不是本书该做的事，不过书中用到的大部分C语言基本表达式[H & S]都总结到表1.1中了，用过程序语言编程但是不熟悉C语言的读者应该看看。表中也列出了笔者在计算机算式中用到的对应操作符，它们按照优先级从高到低的顺序排列（高者先算），在优先级这一列中，L表示左结合，比如乘号就是左结合的运算符：a·b·c=(a·b)·c，而R则表示右结合[㊃]。本书计算机算式里的运算符，其优先级与结合性同C语言一致。

表1.1 C语言与计算机算术表达式对照表

优先级	C	计算机算式	含义
	0x...	**0x...**，**0b...**	十六进制常数，二进制常数

㊀ 不加粗的 $x+y$ 是普通的数学运算，故结果为 -2^{32}，而粗体的 $\boldsymbol{x}+\boldsymbol{y}$ 则是按照计算机算术规则做二进制加法，其结果为1 00000000 00000000 00000000 00000000，也就是1后面32个零。宽度为32的字组无法容纳这个33位元的数，故而最高有效位的“1”会被舍去，留下32个0，因此最终的运算结果就是0。——译者注

㊁ 对应英文写法分别为：bit、nibble、byte、halfword、word、doubleword。——译者注

㊂ 该标准的正式名称是ISO/IEC 9899：1999，俗称C99，详情参见：https://zh.wikipedia.org/wiki/C语言#C99。C语言的最新标准是2011年发布的ISO/IEC 9899：2011，俗称C11。——译者注

㊃ 左结合与右结合的英文分别是left-associative与right-associative，结合性指的是遇到两个相邻的同优先级运算符时，要从左先算还是从右先算。详情参见：https://en.wikipedia.org/wiki/Operator_associativity。——译者注

（续）

优先级	C	计算机算式	含义
16	a［k］		数组 a 中索引为 k 的元素
16		x_0，x_1，…	若干个变量或位元（具体含义会在文中说明）
16	f（x，…）	$f(x,\ldots)$	求函数值
16		abs（x）	求绝对值（例外：abs（-2^{31}）$=-2^{31}$）
16		nabs（x）	绝对值对应的负数
15	x++，x--		后置自增与自减①
14	++x，--x		前置自增与自减②
14	（数据类型名）x		类型转换
14R		x^k	x 的 k 次方
14	~x	$\neg x$，$\overline{x}$	按位取反（也就是求 x 的反码）③
14	！x		逻辑非（若 x 是 **0** 则结果为 **1**，若 x 非 **0** 则结果为 **0**）
14	-x	$-x$	取相反数
13L	x*y	$x*y$	乘法，根据字组长度裁剪其运算结果④
13L	x/y	$x\div y$	带符号整数的除法
13L	x/y	$x\overset{u}{\div}y$	无符号整数的除法
13L	x%y	rem（x，y）	已知 x 与 y 为带符号的数，求 $x\div y$ 的余数。结果有可能是负数⑤
13L	x%y	remu（x，y）	已知 x 与 y 为无符号的数，求 $x\div y$ 的余数
		mod（x，y）	已知 x 与 y 为带符号的数，求 x 除以 y 的余数，并将结果调整到［**0**，abs(y)−**1**］之间
12L	x+y，x-y	$x+y$，$x-y$	加法与减法
11L	x≪y，x≫y	$x\ll y$，$x\overset{u}{\gg}y$	左移位，右移位（以 0 填补空缺位元，又名逻辑移位，logical shift）
11L	x≫y	$x\overset{s}{\gg}y$	右移位（根据 x 的符号来填补空缺位元，又名算术移位或数学移位）
11L		$x\overset{rot}{\ll}y$，$x\overset{rot}{\gg}y$	循环左移，循环右移
10L	x<y，x<=y，x>y，x>=y	$x<y$，$x\leqslant y$，$x>y$，$x\geqslant y$	带符号数的关系比较表达式
10L	x<y，x<=y，x>y，x>=y	$x\overset{u}{<}y$，$x\overset{u}{\leqslant}y$，$x\overset{u}{>}y$，$x\overset{u}{\geqslant}y$	无符号数的关系比较表达式
9L	x==y，x!=y	$x=y$，$x\neq y$	比较两数是否相等，比较两数是否不等
8L	x&y	$x\ \&y$	按位与
7L	x^y	$x\oplus y$	按位异或
7L		$x\equiv y$	按位等值，结果与$\neg(x\oplus y)$相同
6L	x|y	$x\mid y$	按位或
5L	x&&y	$x\ \vec{\&}\ y$	条件与（x 与 y 均不为 **0** 时，结果是 **1**，否则是 **0**）
4L	x||y	$x\ \vec{\mid}\ y$	条件或（x 与 y 均为 **0** 时，结果是 **0**，否则是 **1**）

（续）

优先级	C	计算机算式	含义
3L		$x \mid\mid y$	连接
2R	x = y	$x \leftarrow y$	赋值

① postincrement 与 postdecrement，意为先求表达式的值，然后再对其增减。假设 x 是 5，那么 x＋＋的值还是 5，求完值之后，x 才变成 6。——译者注

② preincrement 与 predecrement，意为先对其增减，然后再求表达式的值。假设 x 是 5，那么先将 x 加 1，然后再判定＋＋x 的值为 6。——译者注

③ one's-complement，又叫一补数，是将二进制数中的每个位元反转之后得到的数。例如 10010 的一补数为 01101。详情参见：https：//zh. wikipedia. org/wiki/一补数。——译者注

④ modulo word size，意思是如果运算结果位数太多，超过了字组长度，那么会丢弃超出的部分。此概念直译为“与字长取模”或“模字长”，为了不使人误认为是和 32（字组长度值）取模，故译文用了意译，下文皆同。——译者注

⑤ 在 C99 标准中，模除的结果与被除数的正负号相同，详情参见：https：//en. wikipedia. org/wiki/Modulo _ operation。——译者注

除了表 1.1 列出的写法之外，书中也将借用一些布尔代数运算符与标准数学符号，在用到它们时会给出解释。

计算机算术中除了“abs”、“rem”之外，还会用到其他一些函数，等讲到那些函数时再给出其定义。

C 语言中，x＜y＜z 这个表达式的意思是：先判断 x＜y 是否成立，若成立，则这个子表达式的值就是 1，否则为 0，然后，再将刚才的结果与 z 比较[㊀]。而在本书所讲的计算机算术中，该表达式的意思就是想判断 $x<y$ 与 $y<z$ 是否同时成立而已。

C 语言中有三种循环控制语句：while、do、for。while 语句的语法[㊁]是：

while(*expression*)*statement*

首先评估 expression 的值，若为**真**（也就是非 0），那么就执行 statement 对应的语句，然后再次判断 expression。一旦 expression 为**假**（也就是 0），while 循环就终止了。

do 语句与之相似，只是判断放在了循环尾部。它的语法是：

do *statement* while(*expression*)

先执行 statement 中的语句，然后再判断 expression。若为**真**，则继续执行循环，若为**假**，则循环终止。

for 语句的格式为：

for(e_1;e_2;e_3)*statement*

首先执行 e_1，这部分通常是赋值表达式。然后判断 e_2，它一般是个关系表达式，如果

㊀ 例如 C 语言中表达式－5＜－4＜－3 就不成立，因为必须先求－5＜－4 这个子表达式的值，它成立，故被视为 1，然后再判断 1 是否小于－3，此判断不成立，故整个表达式不成立。——译者注

㊁ 为了遵循业内惯例，语法格式中的英文单词在译文中保留原样。其中的 expression 意为“表达式”，statement 意为“语句”。——译者注

为假，则终止 for 循环，如果为**真**，那就执行 statement。最后再执行 e_3，这部分通常也是个赋值表达式。执行完毕后，又跳回 e_2 判断。因此，“do i = 1 to n”用 for 语句写出来就是：

```
for (i = 1; i <= n; i++)
```

（书中用到后置自增运算符的场合不多，这是一例。）

ISO C 标准并未规定右移（“≫”操作符）带符号的数时，左边多出来的空位是用 0 填充还是用符号填充[㊀]。在本书的 C 语言代码中，我们假定如果左操作数是带符号的，那么右移操作就使用符号填充，若无符号，则按 ISO 规范以 0 填充。大多数 C 编译器也都这么做。

左移操作符都采用“逻辑”填充。（某些电脑还有“算术”左移操作，也就是左移之后保持符号不变。这通常出现在老式计算机上。）

移位操作还会出现一个问题，那就是 ISO C 标准规定：假如移位的数量等于或大于左操作数的位宽，则结果未定义。虽说如此，但是几乎所有 32 位机在遇到这种情况时，都会把移位数量和 32 或 64 取模[㊁]。书中代码在对待此问题时，上述各种处理方式都会用到。若不同方式之间的差异关乎运算结果，则会给出解释。

1.2 指令集与执行时间模型

我们假定测试书中算法时所用的这台电脑，其 CPU 使用当前通行的 RISC 指令集，这样就好粗略估量其效率了。IBM RS/6000 系列电脑[㊂]的 CPU、Oracle SPARC[㊃]及 ARM 架构[㊄]的处理器都用的是 RISC。这台虚拟电脑的 CPU 使用“三段式地址”表示操作数，其寄存器相当多，至少 16 个。除非另有说明，否则寄存器都是 32 位的。0 号通用寄存器的值永远是 0，其他寄存器作用不限。

有些 CPU 中具备“特殊用途”寄存器[㊅]，专门保存条件比较的结果或“溢出”等状

㊀ 用 0 填充（0-propagating）就是表 1.1 中所说的“逻辑”移位，而用符号填充（sign-propagating）的意思是，如果待移位的数是负值，那么就用 1 来填充，这样移位后的结果还是负的；而如果待移位的数非负，则用 0 填充，移位后的结果仍然非负。这种保持正负性的填充方式也就是上表所说的“算术”移位或“数学”移位。——译者注

㊁ 举例来说，若要将一个 32 位元的数左移 80 位，那么由于 80 大于等于 32，所以必须先求它除以 32 的余数，也就是 16，然后将该数左移 16 位，得出运算结果。——译者注

㊂ RS6000 是 IBM 公司使用其 RISC 架构的 Power 处理器设计生产的小型计算机。详情参见：https://zh.wikipedia.org/wiki/RS/6000。——译者注

㊃ SPARC，全称为“可扩充处理器架构”（**S**calable **P**rocessor **ARC**hitecture），是 RISC 微处理器架构之一。详情参见：https://zh.wikipedia.org/wiki/SPARC。——译者注

㊄ ARM 架构是一种 32 位元精简指令集（RISC）处理器架构，因其具备低成本、高效能、低耗电的特性，故广泛用于嵌入式系统。详情参见：https://zh.wikipedia.org/wiki/ARM 架构。——译者注

㊅ special purpose register，又叫 special function register 或 special register，详情参见：https://en.wikipedia.org/wiki/Special_function_register。——译者注

态码之类的数据，而我们为了简洁起见，假设虚拟电脑的CPU中没有这种寄存器。此外，它还没有浮点指令。浮点数在书中只占少量篇幅，基本上局限于第17章。

现在规定两套RISC指令集：表1.2是“基本RISC指令集”，这些指令再加上表1.3中的那些，就构成“完整RISC指令集”。

表1.2 基本RISC指令集

操作码助记符	操作数	含 义
add，sub，mul，div，divu，rem，remu	RT，RA，RB	RT←RA op RB，其中op可以是加法、减法、乘法、带符号数的除法、无符号数的除法、带符号数的求余、无符号数的求余
addi，muli	RT，RA，I	RT←RA op I，其中op可以是加法或乘法，I是16位元的带符号常量①
addis	RT，RA，I	RT←RA + （I≪16）
and，or，xor	RT，RA，RB	RT←RA op RB，其中op可以是按位与、按位或、按位异或
andi，ori，xori	RT，RA，Iu	同上，但最后一个操作数是16位元的无符号常量
beq，bne，blt，ble，bgt，bge	RT，target	在条件成立时转入target分支。这6个操作符的判断条件分别是：RT＝0，RT≠0，RT＜0，RT≤0，RT＞0，RT≥0
bt，bf	RT，target	当RT是true/false时转入target，与bne/beq等效
cmpeq，cmpne，cmplt，cmple，cmpgt，cmpge，cmpltu，cmpleu，cmpgtu，cmpgeu	RT，RA，RB	将RA与RB比较的结果存入RT。若判断不成立，则RT为0，若成立，为1。cmpeq表示 **comp**are for **eq**uality（判断二者是否相等），cmpne表示 **comp**are for **ine**quality（判断两者是否不等），cmplt表示 **comp**are for **l**ess **t**han（判断前者是否小于后者），以此类推。cmp后面的字母与分支指令中的含义相同。后缀“u”表示无符号数的比较
cmpieq，cmpine，cmpilt，cmpile，cmpigt，cmpige	RT，RA，I	与cmpeq等6个操作符含义相同，只是I为16位元的带符号常量
cmpiequ，cmpineu，cmpiltu，cmpileu，cmpigtu，cmpigeu	RT，RA，Iu	与cmpltu等6个操作符含义相同，只是I为16位元的无符号常量
ldbu，ldh，ldhu，ldw	RT，d（RA）	分别将地址RA＋d中的无符号字节、半字组、无符号半字组、字组载入RT。d为16位元的带符号常量
mulhs，mulhu	RT，RA，RB	将RA与RB之积的高32位存入RT。两指令分别适用于带符号数及无符号数
not	RT，RA	将RA按位取反后的值存入RT
shl，shr，shrs	RT，RA，RB	将RA左移位或右移位后的值存入RT。RB最右方的6个位元代表移位量。shl与shr用0填充，shrs用RA的符号位填充（移位量只取最右方6个位元的原因是，它需要和64取模，以便调整到0至63之间）
shli，shri，shrsi	RT，RA，Iu	将RA左移位或右移位后的值存入RT。移位量是Iu所代表的5位元常数
stb，sth，stw	RS，d（RA）	将RS里的字节、半字、字组存入RA＋d所表示的内存地址中。d是16位元的带符号常量

① 本书多次出现immediate value一词，字面意思为“立即值”、“立即数”，它是谈到计算机指令时的一种术语，在不影响文意的情况下，译文均以“常量”、“常数”称呼之。——译者注

表 1.2、表 1.3 与表 1.4 中的 RA 与 RB，如果作为源操作数使用，就表示这些寄存器本身的内容。㊀

在实际使用的 CPU 中，尚有分支链接跳转指令（branch and link）以及返回到寄存器中所含地址的分支返回指令，前者可用来调用子程序，后者则用于从子程序中返回或实现“switch”功能。此外，可能存在一些处理专用寄存器的指令，特权指令与调用管理服务的指令当然也不少，或许还会有浮点数指令。

表 1.3 列出了 RISC 指令集中可能会用到的其他一些计算类指令，后面的章节中将会讲到它们。

汇编语言中提供了一些“扩展助记符”（Extended Mnemonic），开发人员用起来很方便，它们有点像宏，展开之后通常是一条指令。表 1.4 列举了一些可能会用到的扩展助记符。

常量加载操作会根据常量 I 的值而展开成一条或两条指令。如果 $0 \leqslant I < 2^{16}$，那么就展开成 R0 与 I 的常量按位或操作（ori 指令），而当 $-2^{15} \leqslant I < 0$ 时，则会展开成 R0 与 I 的常量加法操作（addi 指令）。若 I 最右方的 16 个位元都是 0，那么就视为常量移位加法操作（addis 指令），而在不属于上述三种情况时，则会展开成一条 addis 与一条 ori 指令。㊁（在最后一种情况下，也可以直接使用从内存中加载数值的指令来做。但是为了便于估量算法所需的执行时间和空间，我们假定此种常量加载操作总是相当于两个算术指令。）

表 1.3 “完整 RISC 指令集”中的其余附加指令

操作码助记符	操作数	含 义
abs，nabs	RT，RA	将 RA 的绝对值或“负绝对值”①存入 RT
andc，eqv，nand，nor，orc	RT，RA，RB	将 RB 按位取反后与 RA 按位与，按位比较，按位与后按位取反，按位或后按位取反，将 RB 按位取反后与 RA 按位或
extr	RT，RA，I，L	将 RA 中序号为 I 至 I＋L－1 之间的位元提取出来，靠右存放在 RT 中，余位填 0
extrs	RT，RA，I，L	与 extr 相同，但是用符号填充
ins	RT，RA，I，L	将 RA 中序号为 0 至 L－1 的位元插入 RT 中序号为 I 至 I＋L－1 的位置上
nlz	RT，RA	取得 RA 中前导 0 的个数（该操作的结果在 0 至 32 之间）
pop	RT，RA	RT 计算 RA 中有多少个值为 1 的位元（该操作的结果在 0 至 32 之间）
ldb	RT，d（RA）	从 RA＋d 所在的内存地址处加载一个带符号的字节到 RT 中。d 为 16 位元的带符号常量
moveq，movne，movlt，movle，movgt，movge	RT，RA，RB	如果 RA 与 0 的关系满足判断条件，就将 RB 赋值给 RT，否则 RT 的值不变。eq 与 ne 分别表示 RA＝0、RA≠0，其余依此类推
shlr，shrr	RT，RA，RB	将 RA 循环左移位或循环右移位。RB 最右方的 5 个位元表示移位的数量

㊀ 若 RA、RB 不以带括号形式出现，即表示其本身值，否则视为定位所用的内存地址。——译者注

㊁ 常量加载操作要分成三种情况展开的原因，请参考：http://www.engr.uconn.edu/～jeffm/Classes/CSE240-Spring-2000/Lectures/lecture6/node4.html。——译者注

（续）

操作码助记符	操作数	含　义
shlri，shrri	RT，RA，Iu	将RA循环左移位或者循环右移位，移位的数量在5位立即数字段中给出
trpeq，trpne，trplt，trple，trpgt，trpge，trpltu，trpleu，trpgtu，trpgeu	RA，RB	当两个操作数满足RA = RB、RA≠RB等条件时，令处理器中断（trap）
trpieq，trpine，trpilt，trpile，trpigt，trpige	RA，I	与trpeq等指令含义相同，只是第2个操作数为16位元的带符号常量
trpiequ，trpineu，trpiltu，trpileu，trpigtu，trpigeu	RA，Iu	与trpltu等指令含义相同，只是第2个操作数为16位元的无符号常量

① negative of the absolute value是一种特有的绝对值解读方式，详情参见：http://pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp?topic=%2Fcom.ibm.aix.aixassem%2Fdoc%2Falangref%2Fidalangref_nabs_negabs_instrs.htm。——译者注

表1.4　扩展助记符

扩展助记符	展开式	含　义
b target	beq R0，target	无条件执行分支
li RT，I	详见书中解释	将常量加载至RT中。$-2^{31} \leqslant I < 2^{32}$
mov RT，RA	ori RT，RA，0	将寄存器RA中的值赋给RT
neg RT，RA	sub RT，R0，RA	将RA的相反数（也就是二补码）放入RT
subi RT，RA，I	addi RT，RA，－I	将RA减I的差放入RT（$I \neq -2^{15}$）①

① 因为-2^{15}的相反数是2^{15}，该值无法以16个位元容纳，所以在这种特殊情况下，subi与addi展开式不等效。——译者注

一个指令到底应该属于基本RISC指令集还是完整RISC指令集，其实取决于个人的解读。有人可能会觉得无符号数除法与求余操作应该放在完整RISC指令集中，而与此相反，带符号的字节加载（load byte signed，ldb）操作反而应该归入基本RISC指令集。ldb操作之所以会被放在完整指令集中，是因为其使用频率相当低，而且该指令需要将符号位扩展很多次㊀，这样做比较耗费CPU周期。

划分基本RISC指令集与完整RISC指令集时，还有许多可以商榷的地方，此处不再赘言。

书中所用指令最多只会用到两个源寄存器与一个目标寄存器，这样做也简化了计算机的CPU架构（比方说，寄存器堆㊁中最多只需两个读端口与一个写端口即可）。此外也缓解了优化编译㊂的工作量，使它不用再处理那些用到多个目标寄存器的指令了。这样做

㊀ 如果待加载字节的最高有效位是1，则表明其为负数，将它加载到一个32位宽的寄存器时，必须把多出来的24个空位元全部扩展为1，这样才能保证寄存器中的值和正负性与原字节相符。——译者注

㊁ Register File，是CPU中多个寄存器组成的阵列，通常由快速的静态随机读写存储器（SRAM）实现。这种RAM具有专门的读端口与写端口，可以多路并发访问不同的寄存器。详情参见：https://zh.wikipedia.org/wiki/寄存器堆。——译者注

㊂ Optimizing Compiler，是优化程序所用的编译器，可以加快程序执行速度或减低内存占用量等。详情参见：https://en.wikipedia.org/wiki/Optimizing_compiler。——译者注

的代价是，有些本来执行一条指令就能实现的操作现在得分解成两条。比如某些程序想同时知道两个数的商和余数（这种情况很罕见），那么现在就必须用除法和求余两个操作才能完成。在现实环境下的 CPU 中，余数是除法操作的副产品（by-product），很多电脑在执行除法指令时顺便计算好了余数。要获取保存两个字组乘积的双字时，也会遇到这个问题。

条件移动指令（conditional move，例如 moveq）表面上看只有两个源操作数，但换个角度看，也可以说是 3 个。因为该指令的结果取决于 RT、RA 与 RB 的值，所以乱序执行的 CPU 必须把这种指令里的 RT 标注为 use 和 set。也就是说，如果前面一条指令修改了 RT，而后面紧跟一个将要再度修改 RT 的条件移动指令，那么就只能按照这个顺序执行，同时不能丢弃前一条指令的运算结果。有鉴于此，此类处理器的设计者可以选择略过条件移动操作以避免处理这种（从逻辑上讲）需要 3 个源操作数的指令。反之，使用条件移动指令确实能简化分支的实现。

指令格式与本书内容无关，不过上面列出的所有 RISC 指令集，再加上浮点指令和一些管理用的指令，都可以在有 32 个通用寄存器的电脑上用 32 个位元（其中用于指示寄存器序号的字段占 5 个位元[㊀]）加以实现。如果将 compare、load、store、trap 指令所支持的常数字段缩减为 14 位，那么这些指令在具备 64 个通用寄存器的电脑上，也照样能用 32 个位元做出来（其中用于指示寄存器序号的字段占 6 个位元[㊁]）。

执行时间

本书假设所有指令都只需 1 个 CPU 周期即可执行完毕，但是乘法、除法及求余指令例外，我们不去预估这 3 个指令的执行时间。而分支指令不论其执行结果是转入分支还是继续沿主线执行，都只花 1 个周期。

常量加载指令会花费一至两个周期，因为要把常数存入寄存器中，所需的算术指令数可能是 1 个，也可能是两个。

书里用到加载与存储指令的场合不多，我们也假设其只需 1 个周期，而且忽略加载延迟（也就是从算术逻辑单元[㊂]执行完加载指令算起，到待加载的数据真正可以用于后续指令为止，这两者的时间间隔）。

然而，只晓得算术与逻辑指令的执行周期，通常无法估算出程序执行时间，因为数

㊀ 在 CPU 中，除了操作数之外，指令本身也是用二进制来表示的。在由 32 个位元构成的指令中，一般会有指示操作类型的操作码字段、标识寄存器序号的字段，以及容纳常量的字段等。5 个位元能表达值为 0～31 的 32 个数，所以刚好可以对应 32 个通用寄存器的序号。详情参见：https://zh.wikipedia.org/wiki/指令集架构。——译者注

㊁ 一般常量用 16 个位元表示，而指令码本身通常含有两个表示寄存器的字段，现在要将其由 5 位扩展至 6 位，所以原来表示常量用的字段就会缩减为 14 位。——译者注

㊂ 原文为 arithmetic unit，应是 arithmetic logic unit（ALU）的简称，它是中央处理器的执行单元，也是所有 CPU 的核心组成部分，由“And Gate”和“Or Gate”构成，主要用于进行加、减、乘等二进制算术运算。详情参见：https://zh.wikipedia.org/wiki/算术逻辑单元。——译者注

据加载延迟及获取指令延迟可能会大大减缓程序执行速度。尽管近些年人们越来越重视这个问题了，但是本书不打算讨论。另外还有一个因素会改善执行效率，那就是所谓的“指令级并发”（instruction-level parallelism）。很多时下流行的RISC芯片都支持该技术，那些“高端”（high-end）电脑尤其如此。

此类机器的CPU都有多个执行单元，并具备足够的指令派发能力，以便并发执行互不依赖的一组指令（也就是说，这些指令的执行结果都不取决于其他指令，而且不会同时修改某个寄存器或状态位）。由于此技术已非常普遍，所以本书会标明那些相互独立的指令。如此一来，我们就可以说：某某公式用8条指令即可实现，并且在一台具备无限指令级并发能力的电脑上运行，只需5个周期。这也意味着，假使CPU能够合理安排（schedule）各条指令的执行顺序，而且其中加法器[㊀]、移位器（shifter）、逻辑单元及寄存器的数量足够多，那么从理论上讲，只要5个周期就能执行完这段代码了。

然而也不能过分强调这一点，因为不同电脑的指令级并行能力差得很远。比如一台1992年加州产的IBM RS/6000系列电脑[㊁]，其CPU的加法器就有3个输入端，甚至能够并发执行两条首位衔接的加法类指令（比如加法指令的目标寄存器当做比较指令或加载指令的源寄存器用[㊂]）。与之相对，那种在低端嵌入式设备上运行应用程序所用的CPU的构造非常简单，其寄存器堆栈可能只有一个读端口。一般说来，这种机器执行需要两个源寄存器的指令时，会多花一个周期来读取第2个操作数。然而这种CPU也许会有一个旁路（bypass），当某条指令的目标寄存器恰好是下面那条指令的源操作数时，它不用再通过寄存器堆的读端口就可以把寄存器拿来用。在这种设备上以非并行方式执行首位衔接的代码反倒发挥了它的优势。

1.3 习题

1. 将如下循环用while方式改写：

   for（e_1；e_2；e_3）*statement*

   此循环能用do表示吗？
2. 用C语言写一个循环，控制名为i的无符号整数变量，使其值从0开始，递增到32位机所能表示的最大无符号数0xFFFF FFFF（循环范围包含这两个极值）。
3. 此题留给知识丰富的读者：本书所列的基本RISC指令集和完整RISC指令集中，每条指令最多只需读取两次寄存器，写入一次寄存器。请问在常见的RISC指令中，有没有那种貌似简单但实际上却需要操作两个以上源数据，或多次写入寄存器的指令呢？

㊀ 在电子学中，加法器（adder）是一种用于执行加法运算的数字电路部件，是计算机核心微处理器中算术逻辑单元的基础。在这些电子系统中，它主要负责计算地址、索引等数据。此外，它还是二进制数乘法器等其他硬件的重要组成部分。详情参见：https://zh.wikipedia.org/wiki/加法器。——译者注

㊁ IBM RS/6000系列电脑的发展历程可参阅其官方文档：http://www-03.ibm.com/ibm/history/documents/pdf/rs6000.pdf。——译者注

㊂ 原书的说法是one feeds the other，即前一条指令将操作结果作为数据源，“喂”给后一条指令。——译者注