第1章　欢迎阅读本书

1.1　我们的目标

欢迎阅读本书。我们的任务是向你介绍“计算世界”(the world of computing)。本书的主要目标之一，是希望你能认识到计算世界并非如此神秘。相反，计算机（computer）是非常“确定”的一个系统，即在任何时候，在相同的方法、相同的状态下（当然还包括相同的起始条件），同样的问题必然获得相同的结果。其实，计算机并不是什么电子天才，相反，它只是一个电子傻瓜，只会精确地按照我们的要求去执行任务，它本身是没有心智的。

事实上，计算机这样一个复杂的机体（organism），是由一堆简单的部件，经过精心的系统组合而成的。本书首先将介绍那些简单部件的原理和机制，然后一步一步地搭建出一个互连结构，即所谓的“计算机”。这如同一幢房子的建造过程，先是从最底层的“地基”开始，自底向上一层层地“添砖加瓦”，最后形成一个功能完整的计算机。在逐层的讲述过程中，每增加一层，我们都将解释在做什么，以及新出现的概念与其底层组织之间的关系等。我们的目标是，一旦你完成了这本教材的学习，就能够自然地操纵一种语言（如C语言）来编写程序了，并能使用其中的一些高级功能，同时也能理解在程序执行过程中，计算机底层所发生的相应运作。

1.2　怎么才能做到

从第2章开始，是基于这样一种理念，即计算机不过是个电子设备，它由许多电子部件组成，而这些部件又通过导线相连。在任何一个时刻，这些导线要么是高电平、要么是低电平。但是在理解计算机这种电子设备的时候，我们并不关心具体的电压值是多少。换句话说，电压是115V还是118V并不重要，我们所关心的只是“相对于0V电压，它是否足够大”。如果该电压与0V电压相差很小，则在逻辑标识上将它定义为“0”；而如果电压与0V相差很大，则将它定义为逻辑“1”。

同时，通过0和1序列的组合，我们可以表示任何信息。例如，可以将字母a表示为01100001，十进制数35则表示为00100011。后面将详细介绍这种编码体系的原理。

当我们开始习惯了通过0和1编码来表示信息，以及基于这种编码的操作（如“加”操作）之后，就会进入下一个问题：“计算机是怎么工作的？”在第3章中，将介绍怎样用晶体管构建现代微处理器，具体地说，就是怎样使用晶体管来构建能够运算的部件（如“加法器”）和存储信息的记忆部件（如“内存”）；随后，第4章介绍“冯·诺伊曼（Von Neumann）机器”，它是一个描述计算机应该怎样工作的模型；第5章介绍一个简单的机器，即LC-3（Little Computer 3）。从LC-1开始，LC-3已经历了两版修改，它具备现代微处理器应具备的所有特性。所谓现代微处理器，如Intel 8088（用于1981年的IBM PC）、Motorola 68000（用于经典的1984年的Macintosh），以及奔腾IV（它是2003年高性能PC的首选处理器之一）等微处理器产品，而LC-3具备这些真实微处理器产品所具备的所有重要特性，但又不像这些“真家伙”那样复杂，因而很容易理解和掌握。

在理解了LC-3的工作原理之后，下一步就是要对它编程。开始是采用其自身语言即LC-3机器语言编程（第6章）；然后采用汇编语言编程（第7章），当然，汇编语言相对人的自然语言来说，多少还是有点儿让人不习惯；第8章介绍有关LC-3是怎样输入/输出信息的问题；第9章涉及两个很

重要的LC-3机制，即TRAP和子程序调用；第10章将对LC-3编程做全面总结，同时引入两个重要的概念，即“栈”（stack）和“数据转换”（data conversion）。最后，将给出一个稍微复杂的例子，它是一个基于LC-3编程实现的手持计算器。

本书的第二部分（第11~19章）将注意力转向高级编程语言——C语言。我们将深入介绍C语言的很多实现机制，这些内容在一般的入门教材中是不会介绍的。在几乎所有的例程中，我们都会将高层的C结构（construct）和底层的LC-3实现联系起来，以使你明白在使用C程序中的特定结构时，它对底层计算机存在什么要求。

我们讲解C语言的方法是，先介绍基本概念如变量和操作符（第12章）、控制结构（第13章）、函数（第14章）等；之后，是高级话题如C语言调试技术（第15章）、递归（第16章）、指针和数组（第17章）等。

C语言部分的结束篇是两个常见的高层结构，即C语言的输入/输出（第18章）和链表（第19章）。

1.3 两个反复出现的理念

在本书中，有两个理念将反复出现并反复强调：一是“抽象”，二是“在脑子里不要对硬件和软件做任何区分”。这两点非常重要！我们希望每个人都能认识到其中的价值，这也正是我们不断向工程专业和计算机科学专业的学生所灌输的。如果你想成为一个高级工程师或计算机科学家，领会这两点远比你理解计算机是怎样工作或怎样对它编程更为重要。随着对全书学习的不断深入，相信你对它们的理解会越来越清晰。

“抽象”理念（notion of abstraction）非常重要，它是学习的重点，也是在实践中要把握的核心理念。不管你未来是要做数学家、物理学家、工程专家，还是要做商业人士，抽象理念都非常有用，很难想像有哪个学科或知识体系中不需要“抽象”。同样，将硬件和软件做明显的区分也是错误的，这对未来的工作和学习都会造成误导。下面我们就来阐述这两个永恒的理念。

1.3.1 抽象之理念

抽象（abstraction）在生活中普遍存在。当我们搭乘出租车的时候，如果我说“去飞机场”，那么我使用的就是抽象的表达方式。为什么呢？因为我还可以用另一种表述方式，详细告诉他到达目的路线的每一个步骤：“顺这条街道向前过10个街区，左转”，然后当他到达这里之后，我又告诉他“现在顺着这条街道继续5个街区，右转”，如此继续。显然，你知道其中的细节，但这远不如告诉司机你要去机场来得简洁。如果还想进一步细化，你甚至可以将“顺这条路向前10个街区……”这句话分解为“踩油门”、“转方向盘”、“注意过往车辆和人行道”等这样的动作细节，但显然没有这个必要。

学会“抽象”是个重要的进步，它让我们学会站在更高的层次看问题，从而将事物的本质表现出来，而将其中的细节隐藏起来；它让我们学会更有效地使用时间和大脑；它让我们在分析问题时不至于陷入泥潭。

当然，其中存在这样一个假设，即“假设各个方面的细节都是运转正常的”。但是，如果底层细节的工作并不是完全正常呢？这是一个挑战，在这种情况下，要求我们不仅要具备抽象的能力，还要具备“分解抽象”的能力，这样才能保证问题的顺利解决。有人又称之为“解析”过程，即从抽象回到具体的过程。

此刻，让我想起两个小故事：

第一个故事是我很久以前穿越亚利桑那州的一次旅行。那是一个炎热的夏天，我当时的家是在常年温和的帕洛阿尔托（属加利福尼亚州）。为了应对亚利桑那州的炎热天气，我在出发之前去

机械师那里改装车子的制冷系统。注意，我在这里用了一个很抽象的说法——“制冷系统”。然而，我忽略了一个细节，应对帕洛阿尔托的天气所需要的制冷系统远不足以对付亚利桑那州沙漠的炎热。结果你一定猜到了，制冷系统在到达目的地之前出问题了，结果我被迫在Deer Lodge（亚利桑那州人口第三大的城市）待了两天，等待维修所需要的盖板密封圈到货。

第二个故事（可能是杜撰的传闻）发生在电力发电时代的早期。通用电气公司的一个大发电机出故障了，但面对发电机前板上一大堆的仪表盘和旋钮，所有的人都束手无策。大家都知道，调整其中的某些旋钮就可能解决问题，但谁也无法确定是其中的哪些旋钮，以及应该是顺时针还是逆时针旋转、转多少角度？正在这时，请来了电力厂创建初期的一个大师级人物。他看了一眼仪表盘，又仔细地听了一会儿电机的声音。然后，他从口袋里拿出一个螺丝刀，将其中的一个旋钮逆时针旋转了35°；机器正常了！随后，他为自己这两分钟的工作开出了一张1000美元的收费单（这在当时是一笔巨款）。控制中心接到这份账单时很不情愿，于是请求他开一个具体的明细账单，以说明收费理由，新账单的明细如下：

（1）将旋钮逆时针旋转35°：　　$0.75
（2）知道旋转哪个旋钮以及旋转多少度：　$999.25

两个故事所表达的信息是相同的，即“抽象”能提高我们的效率，从而摆脱细节的纠缠。如果事情不存在什么意外，就会一切OK！即如果我不是去亚利桑那州旅游的话，抽象词“制冷系统”就足够了，而当时我却忽略了告诉机械师要穿越亚利桑那州沙漠这个“细节”。同样，如果不是电机发生了意外故障，大师对电机的深刻理解也就派不上用场了。

从这两个故事中，我们获得的启示是：当设计一个由各种门电路组成的逻辑电路时，千万不要深陷门电路的内部原理，因为这会大大拖延设计进度。你应该将其中的每个门电路都看做是现成的、可靠的；而仅当电路不工作的时候，才去研究门电路的内部结构，也只有这样，才能发现问题的症结所在。

再如，当你设计一个复杂的计算机应用程序，如电子表格处理系统、字处理系统或计算机游戏时，你可以将其使用到的每个组件都看做是一个“抽象”。此时，探究每个组件的细节是毫无意义的，那只会让你的工作永远无法结束。但当系统出现问题时，要想发现问题所在，就必须深入到每个组件的实现机制。

抽象技能相当重要。我们的观点是，抽象的层次越高越好，而且它与工作效率成正比。本书的做法是逐步提高抽象层次，我们先是基于晶体管描述逻辑门的实现机制，但一旦你领会了逻辑门的抽象，晶体管将永不再提；随后，就是基于逻辑门来构建更高层次的结构，而一旦我们理解了这些结构，逻辑门又将被丢弃。

结论

“抽象”能提高我们的思考效率。换句话说，忽略抽象之下的细节，会让我们更有效率。希望不要有人一听到“抽象”就反感哦！相反，你应该感谢它，抽象确实能提高我们的效率。

如果我们不需要将一个组件（component）和其他的东西相结合（以构建更大的系统），并且组件内部也不会出问题，那么，将认识停留在抽象层面就万事大吉了。但实际情况是，我们肯定会需要将这些组件拼装成更大的系统，而这些组件结合在一起工作的时候，也难免会出错误。这就意味着，我们既要不断地提高抽象层次，又要注意细节的深入。

1.3.2　硬件与软件

许多计算机科学家或工程师称他们自己是搞硬件的或是搞软件的。硬件通常指一个“物理的”计算机以及和它相关的方方面面；而软件通常指程序，如操作系统UNIX或Windows、数据库系统

Oracle或DB-terrific、应用程序Excel或Word等。他们的这种说法是暗示他们对其中的某一方面相当精通，而对另一方面知之甚少。听起来好像在软件和硬件之间存在一堵很高的墙，硬件是描述有关计算机怎样工作的，而软件的主导则是程序，你要做的就是选择待在墙的哪一边。

当你开始学习和接触计算机的时候，我们希望你抛弃这种观点。因为在我们看来，硬件和软件只是计算机系统中两个组成部分的名称而已；对设计者来说，具体将计算机的某个功能划分给哪部分来实现，以及它们之间如何协同工作，原则只有一个：让计算机工作得最棒（而不是刻意要区分它们）！

处理器的设计者如果懂得运行在处理器之上的程序需求，那么所设计的处理器必然比那些不懂的人所设计的处理器要快。例如，Intel、Motorola等大牌处理器设计厂家，在许多年前就意识到，未来的程序如E-mail、视频游戏、视频电影等，将大量包含视频信息（video clip），未来的处理器必须保证它们的执行性能。结果是，在他们所设计的处理器中，大都内嵌了专用视频处理硬件。如Intel为此提出的MMX指令集及MMX专用执行硬件，而Motorola和Apple也做了类似工作，如AltaVec指令集及其硬件。

软件设计中也有类似的故事。懂硬件特性的软件设计师所设计的程序，其运行性能远高于那些不懂硬件的人所设计的程序。“排序”是一个经典的计算任务，几乎在所有的大型软件中都不可或缺。我们需要将一系列的条目（item）按照一定的顺序排列，如字典中的单词需要按字母排序，学生成绩单是按数字排序的。存在太多有关排序的编程方法（又称算法）。但是，Donald Knuth在他的传世巨著《计算机程序设计艺术》（第3卷）中，竟然花了391页的篇幅专门讲述排序，因为要想做到排序最快，在很大程度上取决于软件设计者对硬件特性的了解。

结论

我们相信，不管你未来的职业取向是计算机软件还是硬件，两者都懂必然会使你更强。本书的宗旨就是让你两者都掌握。有时我们在讲述一个概念的时候，并未特意强调是关于软件或是关于硬件的，但通常是两者都相关的。

当你在学习数据类型（data type）这个软件概念时（第12章），你将理解硬件中字（word）的有限长度，是怎样影响软件中数据类型表示的。

当你学习函数（function）时（第14章），你会联想起硬件的知识，从而明白“函数调用规则”的含义和意义。

当你学习递归（recursion）——一个强大的算法工具时（第16章），结合硬件知识，你将明白为什么花些时间递归执行过程（procedure）是值得的。

当你学习指针（pointer）变量时（第17章），有关计算机内存的知识将更有助于深入理解指针，从而知道什么时候适合使用它，什么时候不适合使用它。

当你学习数据结构（data structure）时（第19章），有关计算机内存的知识，将帮助你理解数据结构在内存中的具体实现，以及有效操作数据结构的窍门。

我们知道，前面的内容中所出现的很多名词让你感到迷惑，不要紧，在本章结束的时候再重读一遍以上内容即可。目前你仅仅需要认识到：软件中的许多重要话题是和硬件中的话题紧密交织的。我们的观点是：无论你更倾向于其中的哪一面，从两方面思考必然会使其更容易。

更重要的是，面对大多数的计算问题，如果解题者具备软、硬件两方面的知识，那么他或她给出的答案会更漂亮。

1.4 计算机系统简述

在前面的章节中，我们已多次使用了“计算机”这个词，但并未直接解释过其定义。它是指这样一种机制，即同时在做着两方面的事情：既控制着信息的处理过程，同时也是信息处理过程

的具体执行者。所谓“控制着信息的处理过程”(directs the processing of information)，指的是它必须决策下一个执行任务是什么，而“处理过程的具体执行者”，意味着它必须具备“加”、“乘”等运算能力以产生执行结果。该机制更准确、更合适的称谓应该是“中央处理器”(Central Processing Unit, CPU)。本书的重点也是围绕CPU及其之上的程序运行而展开的。

在20年前，一个处理器由10个或更多的18英寸电路板组成，每个电路板上包含了大约50个部件（采用集成电路封装，如图1-1所示)。而今天的处理器通常由一个微处理器芯片组成，其大小仅为一片1英寸左右的硅材料，其中包含了大约几百万个晶体管（如图1-2所示)。

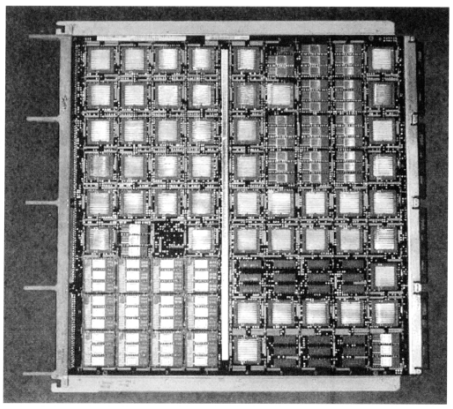


图1-1 处理器板，1980年产

资料提供：Emilio Salgueiro, Unisys Corp.

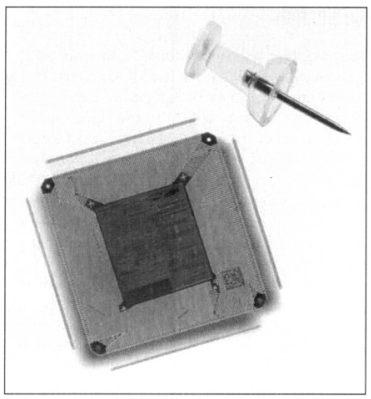


图1-2 微处理器，1998年产

资料提供：Intel Corp.

然而大多数人更熟悉“计算机”这个词，它包含了比处理器更多的意思。一个计算机系统(computer system）由更多的部件组成（如图1-3所示)，除了处理器之外，还包括键盘（用来输入命令)、鼠标（用来点击菜单)、显示器（用来显示计算机系统产生的信息)、打印机（用来打印信息的拷贝)、内存（用来临时存储信息)、磁盘和CD-ROM（用来永久存储信息及很多可以执行的程序或软件)。

这些附加的部件更方便了计算机最终用户的使用。例如，如果没有打印机，就只能手抄屏幕上显示出来的信息；而没有鼠标，你将永远手工输入各种命令，不像现在这样，轻轻点击鼠标按键，就可以启动命令。

图1-3 个人计算机（Personal Computer, PC)

资料提供：Dell Corp.

因而，在我们开始本书的旅途之际，需要声明的是，本书的重点落在那1英寸的空间内部，即CPU。但若没有其他这些部件（虽然它们不是本书的重点内容)，计算机使用起来就不会那么方便了。

1.5 两个非常重要的思想

在结束第1章之前，我们还将介绍两个非常重要的思想。它们非常重要，论述了计算（computing）的全部内涵，因而我们希望你能了解它们。

第一：所有的计算机（不管是最大的还是最小的、最快的还是最慢的、最贵的还是最廉价的），只要给予足够的时间和内存，它们所能完成的计算任务是相同的。换句话说，最快的计算机能够完成的事情，最慢的计算机也一样能够完成，只是更慢一些而已；而一个便宜的计算机所不能完成的事情（如果有足够内存的话），对于一个更昂贵的计算机来说，同样也是无法完成的。总之，所有的计算机能够完成完全相同的事情。只是有些计算机可能做得更快些，但绝不会做得更多。

第二：我们用英语或其他语言给出了一个问题，然而计算机却能通过电子运转（运行程序）来解决这个问题，太奇妙了！至于怎样把用人类语言描述的问题转换成能够影响电子运转的电压，需要一系列的、系统的转换过程。在计算机的50年历史里，这一转换问题竟然被成功解决了，而且这一复杂的转换任务竟然是由计算机本身完成的。看起来不可思议，但确实如此。

本章后面的内容将详细阐述这两个思想。

1.6 计算机：通用计算设备

一本入门性的教科书先描述计算机是怎么工作的，感觉有些奇怪吧？机械专业的学生是先学物理，然后才是汽车发动机的工作原理；化学工程专业的学生是先学化学，而不是石油提炼。那为什么计算机专业的学生要先学计算机设备呢？

答案是：计算机是特别的。要学习计算机的基本原理，必须先了解计算机是怎样工作的。其原因在于计算机被称为“通用计算设备”（universal computational device），不理解吗？请看下面的解释。

在现代计算机出现之前，曾经出现了很多能够计算的机器。其中，有些是模拟机（analog machine）——即机器产生的结果是用可测量的模拟量来表示的（如电压、距离等）。例如，滑动计算尺计算乘法的机制，就是滑动其中的一个对数等分尺，然后从第二个尺子上读出其对数“距离”。还有些早期的加法器，其工作原理是采用在秤盘上加重量的方式。模拟机器的缺陷主要是难以提高其精度。

同模拟机相比，数字机（digital machine）通过一组固定的、有限的数字和字符来完成操作，这就是为什么数字机最终主宰计算世界的原因。你应该很熟悉模拟手表和数字手表之间的区别[⊖]。模拟手表有时针、分针，以及秒针，它通过这些针的位置来表达具体的时间；而数字手表通过数字来表达时间。对于数字手表，提高其精度的办法是增加数字数目，如10 : 35 : 16（而不是10 : 35）。但对于机械手表，该怎样表达百分之一秒这样的精度呢？可以的，但你或许需要再加一个更长的秒针（应该比已有的秒针更长些）。本书在提到计算机的时候，讲述的对象是数字计算机。

在现代数字计算机出现之前，在西方最常见的数字机是加法器（adding machine），而在东方用的则是神奇的算盘。加法器是一种能执行特定“加”功能的机械或机电设备。另外，还有各种各样的其他数字机，有的能够执行整数乘法，有的能对一堆有打孔标识的卡片做字母排序。所有这些机器的局限性在于：它们都只能做一件特定的工作。例如，如果你有一台只能做加法的机器，那么做两个数的乘法运算时，还得用纸和铅笔。

但计算机就不一样了。你需要告诉计算机的是“方法”和“任务”，即怎样做加法，怎样做乘法，怎样对一堆表单做排序或任何其他计算任务。而假若你又想出了新的计算方法，就不需要再

⊖ 生活中我们俗称它们为机械表和电子表，但这种说法已不太准确，如石英表就是机械和电子混合的，但石英表属于模拟表。——译者注

购买或设计新机器了，所要做的惟一事情，就是在原来的计算机上增加一些新的指令（instruction）或程序（program）即可。这就是我们称计算机为“通用计算设备”的原因。计算机科学家相信“任何事情都是可计算的”，换句话说，也就意味着任何事情都是可被计算机所运算的（只要给它足够的时间和内存）。当我们学习计算机的时候，我们要学习计算的基本原理，即计算是什么？怎样实施计算？

通用计算思想的产生要归功于Alan Turing（阿兰·图灵）。1937年，图灵提出一个大胆的假设：任何计算都可以由这样一台机器来完成。这个机器就是图灵机（Turing machine）。他为这类机器给出了一个清晰的数学描述，但没有为之建造一台真正的机器。世界上真正可运行的数字计算机是在1946年才出现的。图灵的兴趣在于解决一个哲学问题：计算的可定义性。于是，他开始观察人们在计算时所采用的各种方法和行为，其中包括：在纸上做标记（mark），按照一定的规则记录符号（symbol）等行为。并将这些行为抽象出来，定义了一种能够表达它们的机制；同时，他还给出了一些例子，来解释基于这种机制是怎样完成一些特定任务的，例如，用图灵机来完成两个整数的加法及乘法。

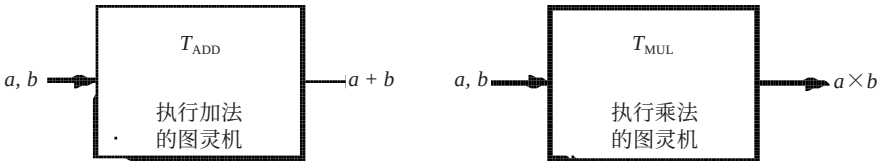


图1-4　图灵机的黑箱模型

图1-4分别是图灵机加法和乘法的“黑箱”模型，要完成的任务描述如箱体内文字所示，被操作数据从箱体输入，操作结果从箱体输出。黑箱模型并不具体说明任务是怎样完成的，但实际上有很多种方法都可以实现两个数的加法或乘法。

图灵提出，任何计算都可以通过某种图灵机来完成，我们称这个理论为“图灵论题”（Turing's thesis）。虽然图灵论题从来没有被严格证明，但众多的证据都表明它是正确的。同时我们发现，任何试图对图灵机做出的改进尝试，最终都可以用图灵机本身来实现。

有关图灵机理论的最好阐述，还要数图灵本人的那份论文。他这样说道：要构建一个比任何形式的图灵机都要强大的机器，则这个机器U必须能仿真所有的图灵机。假设，我们希望机器U能仿真图灵机中两个整数相加的模型，则给定输入，U应该能输出对应的求和结果。随后图灵证明，图灵机也可以做到这点。由此证明了，任何试图发现图灵机所不能做的事情的企图都是失败的。

图1-5进一步阐明了这一点。假设要计算“$g\times(e+f)$”，很简单，你只需要向U提供“加法”图灵机和“乘法”图灵机的描述，以及三个输入参数e、f和g，剩下的事情交给U来做即可。

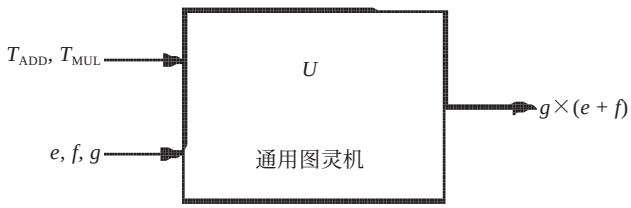


图1-5　通用图灵机的黑箱模型

在通用图灵机U的描述中，图灵第1次深入地阐述了一个论题，即计算机能做什么。换句话说，一台计算机（拥有足够的内存）和一个通用图灵机之间，它们所能完成的事情是一样的。它们中的任意一个，只要给定计算任务的描述及相关数据，都能计算出结果。计算机或图灵机能够计算任何可计算的任务，因为它们都是可编程的。

因此我们说，即使是一台昂贵的大型计算机，它所能完成的事情并不比一台廉价的小计算机多多少。多花的钱只是使你拥有的计算机速度更快、显示器的分辨率更高、声卡系统音质更好。如果你的计算机是一台廉价的个人计算机，你就已经拥有了一台通用（Universal）计算机。

1.7 从问题描述到电子运转

图1-6描述的是要控制电子（器件）按我们的意图工作所经历的整个过程。我们称这个过程的每个步骤为“转换层次”，其中的每一层都有多种实现选择，如果我们忽视其中的任一层，设计完美计算机系统的愿望都会失败。

图1-6 转换层次

1.7.1 问题的提出

描述问题的时候，我们采用“自然语言”，即人们所说的语言，如英语、法语、日语、意大利语等。这些语言都经历了几百年的发展，其中包含了太多的不适合作为计算机语言的东西，其中二义性特征最为突出。自然语言中包含了太多的二义性，如不同的说话声调和音量，以及不同的上下文句子内容，都会使得同一段话表现出不同的语义。

英语中有个经典例句：“Time flies like an arrow”（光阴飞似箭）。我们至少可以有三种解释方法，取决于下面三种场景：

（1）一个人正在注意时间的流逝有多快。

（2）一个人正在做昆虫赛跑比赛的相关工作。

（3）一个人正在给昆虫爱好者Abby写信。

其中，第一种情况是一个明喻，表示时间的流逝如箭一般；第二种情况是这个人告诉时间记录员他/她应该像箭一样快速工作；第三种情况是指一种特殊的flies(time flies) like arrow, 而另一种flies(fruit flies) like banana[⊖]。

类似的二义性对于计算机指令来说是不能接受的。计算机是一个电子傻瓜，它只能做你让它做的事情，但如果你给它的命令存在有多种意思（即二义性），它就会茫然不知所措了。

1.7.2 算法

从问题的提出开始，向下转换的第一步是将问题的自然语言描述转换为算法（algorithm）描

⊖ 源自语言学上的一个例子：“Flies like an arrow, fruit flies likes a banana.”（光阴飞似箭，果蝇喜欢香蕉）；外形完全一样的句子，结构完全不同，这里是个笑话。——译者注

述，去除那些无用的特性。算法描述的特点是流程化、步骤清晰，并确保该流程能终止。其中每个步骤的定义描述都足够精确，以保证能在计算机上执行。这些特性可以阐述如下：

- 确定性（definiteness）。表明每个操作步骤的描述是清晰的、可定义的。我们认为“制作烙饼的过程说明”是缺乏定义性的，因为它说“拼命搅动直到成为糊状”（stir until lumpy），其中“糊状”（lumpy）的程度表述就是模糊的。
- 可计算性（effective computability）。表示每一步的描述都可被计算机执行。而“取最大的素数”这样一句描述，我们称其不具备可计算性的，因为根本就不存在“最大”的素数。
- 有限性（finiteness）。即过程是会终止的。

任何一个问题都具有多种求解算法。其中，有的算法步骤最少，有的算法允许个别步骤反复执行，等等。值得注意的是，如果一个算法虽然描述出来的步骤很多，但如果这些步骤在一个时刻可以同时被计算机执行（并行），无疑该算法的执行速度是快的。

1.7.3 程序

下一步就是将算法转换为程序，即用编程语言描述。编程语言属于“机械语言”，与自然语言不同的是，机械语言不是设计出来为人所讲的，相反，它被设计成严格的顺序方式，以便让计算机顺序地执行指令序列。换句话说，它不存在二义性问题。

大约有1000多种程序语言。有的是有“特定用途”的（如Fortran是科学计算语言、COBOL是商业数据处理语言），而本书后半部分介绍的C语言是专门为底层硬件操作而设计的语言。

还有用于其他一些目的的语言，如Prolog语言常用来设计专家系统，LISP语言则被那些研究人工智能的人们所青睐，而Pascal语言则成了学生学习计算机语言的教学语言。

计算机语言可以分为高级语言和低级语言两类。高级语言和底层计算机的相关性很弱（距离很远），或称之为“机器无关”语言。前面提到的各种语言都属于高级语言。低级语言则和执行程序的计算机紧密相关，通常一种低级语言只对应一种计算机，我们称之为“某某机器的汇编语言”。

1.7.4 指令集结构

下一步的任务是将程序转换成特定计算机的指令集（instruction set）。指令集结构（Instruction Set Architecture, ISA）是程序和计算机硬件之间接口的一个完整定义。

ISA定义包括：计算机可以执行的指令集合，即计算机所能执行的操作，以及每个操作所需数据是什么，即操作数（operand）；ISA还定义了可接受的（legitimate）操作数表达方式，即数据类型（data type）；ISA还定义了获取操作数的机制，即定位各种操作数的不同方法，我们称之为“寻址模式”（addressing mode）。

不同的ISA定义的操作类型、数据类型和寻址模式的数目都是不同的。有的ISA只有几个操作类型，有的ISA则有上百个；有的ISA只有一种数据类型，有的则有几十种；有的ISA只有一、两种寻址模式，而有的则有20多种。如x86（PC机中的ISA），有100种操作类型、十几种数据类型、二十多种寻址模式。

ISA的设计中，还要折衷考虑计算机内存的大小及每个存储单元的宽度（即能容纳的0和1的数目）。

许多ISA一直延续至今，典型的例子就是x86。该ISA于1979年由Intel公司设计，目前同时被AMD和其他一些公司所采用。其他一些著名的ISA包括：PowerPC（IBM和Motorola）、PA-RISC（Hewlett Packard）、SPARC（Sun Microsystems）等。

将高级语言（如C）翻译为ISA指令（如X86）的过程，通常是由一个被称为“编译器”

（compiler）[⊖]的程序来完成的。例如，将C语言程序翻译成x86 ISA时，需要一个“x86的C编译器”。就是说，针对不同的高级语言和目标计算机组合，需要一个对应的编译器。

将特定计算机的汇编语言程序翻译为其ISA的过程，则是由汇编器（assembly）来完成的。

1.7.5 微结构

下一步的任务是将ISA转换成对应的实现。实现的具体组织（organization）被称为“微结构”（microarchitecture）。例如，近年来许多处理器都实现了X86这种ISA结构，但每个处理器的实现方法不同，即有自己的微结构。最早的实现是8088（1979年），而最新的则是Pentium IV（奔腾4，2001年）；再如Motorola和IBM已实现了几十种基于Power PC ISA的处理器，每一种也都有自己的微结构，其中最新的两种是Motorola的MPC7455和IBM的Power PC 750FX。

对于设计者来说，每次的新设计都是一次机会，设计者可以重新权衡新机器的性价比。设计永远是一个“权衡”的挑战练习，问题在于在高（或低）成本下，能否实现一个高（或低）性能的计算机。

汽车制造业为ISA和微结构（即ISA的实现）的关系提供了一个很好的比喻：ISA描述的是驾车人在车里看到的一切，几乎所有的汽车都提供了相似的接口（但汽车的ISA接口和轮船、飞机的ISA差别很大），所有的汽车中，三个踏板的定义完全相同，即中间的是刹车、右边的是油门、左边是离合器。ISA表达的是基本功能，其定义还包括：所有的汽车都能够从*A*点移动到*B*点，可前进也可后退，还可左右转向等。

而ISA的实现是指车盖板下的“内容”。所有的汽车其制造和模型都不尽相同，这取决于设计者在制造之前所做的权衡决策，如有的制动系统采用刹车片，有的采用制动鼓；有的是八缸发动机，有的是六缸，还有的是四缸；有的有涡轮增压，有的没有。我们称这些差异性的细节为一个特定汽车的“微结构”，它们反映了设计者在成本和性能之间所做的权衡决策。

1.7.6 逻辑电路

微结构最终是由一组简单的逻辑电路实现的，有多种实现方法可供选择，各种方法之间存在着性能和成本上的差异。例如，仅仅一个“加法器”，就存在着多种实现，它们所表现出来的运算速度（性能）及其成本差异非常大。

1.7.7 器件

最后要说明的是，每个基本的逻辑电路，都是按照特定的器件技术（device technology）来实现的。就是说，CMOS电路中所使用的器件与NMOS电路中所使用的器件是不同的，它们与砷化镓电路中所使用的器件也不同。

1.7.8 小结

综上所述，从用自然语言对问题进行描述开始，直到电子（器件）的实际运转，之间要经历很多层次的转换。如果我们能说“电子”语言，或电子能“听懂”我们说的语言，那么我们只需要走到计算机面前，直接向电子发布命令即可。然而，我们不会说电子语言，电子也听不懂我们

⊖ “编译器”是个广义的说法，有时又称为“compiler driver”。事实上，为完成编译工作，需要很多工具（toolkits），如：预处理器（pre-processor）、编译器（compiler）、汇编器（assembler）、链接器（linker），以及一些必须的现成工具（bin-utility）如文档处理（archiver）等。——译者注

的语言，我们所能做的事，只能是进行一系列的转换。在转换过程中，每一层的实现又都存在很多选择方案，面对不同的方案，我们的最终决策和选择决定了系统实现的性能和成本。

本书将详细介绍转换过程中的每一个环节，比如晶体管是怎样实现逻辑电路的，逻辑电路是怎样构成微电路的，以及微电路怎样实现一个特定的ISA（我们的ISA是LC-3）。然后，从某个问题出发，讲述其C语言的描述，以及C语言又怎样转换成LC-3的ISA描述的全过程。

我们衷心希望本书能陪你度过一个愉快的学习过程！

1.8 习题

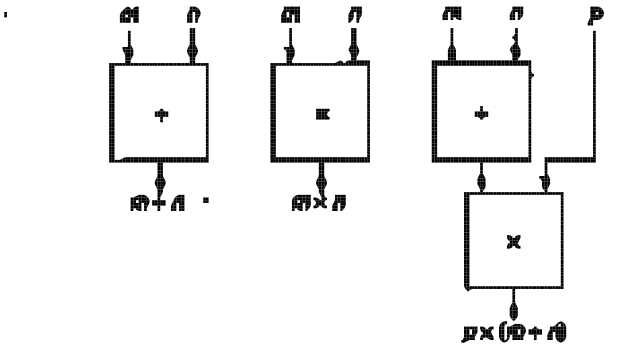
1.1 试解释1.5节中两个重要思想中的第一个。

1.2 试问，同汇编语言相比，高级语言是否能向底层计算机表述更多的计算方式？

1.3 试问，是什么原因使得模拟计算机难以实现，从而使设计者转向采用数字设计？

1.4 列举自然语言的特性之一，说明它为什么不适合直接作为编程语言？

1.5 假设我们有一个“黑箱子”，其输入为两个数字，输出为两数之和，如图1-7a所示；另外还有一个箱子能求两数的乘积，如图1-7b所示。假设我们有无穷多个这样的箱子，通过箱子的互连，我们可以完成如$p\times(m+n)$这样的运算，如图1-7c，请问：怎样通过互连完成如下的计算：



a) 加法　　b) 乘法　　c) 加法和乘法组合

图1-7 黑箱子的功能

a. $ax+b$。

b. 求四个输入w、x、y、z的平均值。

c. $a^2+2ab+b^2$（你能只用一个加法箱和一个乘法箱就完成吗？）。

1.6 试用自然语言写一句话，然后给出这句话可能的两种解释。

1.7 1.3.1节有关抽象的讨论给出的结论是：如果“底层所有一切都很好”，我们是没有必要去理解这些部件的组成的。但如果情况并不良好，我们还是要学会分解部件。在出租车的例子中，你并不知道怎样才能到达机场。采用抽象的方法，你只需要简单地告诉司机“带我去飞机场”即可。解释在什么情况下这种方法是有效的，在什么情况下会起到反作用。

1.8 John说：“I saw the man in the park with a telescope.”。这意味着什么？对这句话有多少种可能的解释？请列出来。这句话具备的什么特性，使得它如果出现在程序中是无法接受的？

1.9 试问，自然语言可以表达算法吗？

1.10 给出算法的三个特性，并给出简单的解释。

1.11 针对算法的每个特性，分别给出一个例子，在例子代码中，都缺少这个特性，说明此时为什么不能说它是一个算法。

1.12 请问：以下从a到e的描述是否是算法？如果不是，那么它缺乏算法的哪些特性？

a. 将下面矩阵中的第一行与其他首列不为0的行相加（注意：列是垂直方向的，行是水平方向的）；

$$\begin{bmatrix} 1 & 2 & 0 & 4 \\ 0 & 3 & 2 & 4 \\ 2 & 3 & 10 & 22 \\ 12 & 4 & 3 & 4 \end{bmatrix}$$

b. 为了证明素数的数目和自然数的数目一样多，创建一系列素数和自然数的配对单元，将第一个素数与1（即第一个自然数）配对，第二个素数与2配对，第三个素数与3配对，依次类推。如果结束的时候，每个素数都找到一个与其配对的自然数，即表明素数的数目和自然数的数目相同；

c. 给定两个矢量，每个矢量20个元素，试执行以下操作：取第一个矢量的第一个元素，与第二个矢量的第一个元素相乘；对第二个元素做相同的操作，依次类推。然后将所有的乘积相加（即点积（dot product））；

d. Lynne和Calvin都想带小狗出去散步，互不相让。Lynne于是拿出一个分币，提议抛硬币决定，但Calvin不相信Lynne（怀疑硬币有重量倾向，即总是某一面向上），于是提出一个新的方案：

1）连续抛两次。

2）如果第一次头像面向上，则下次选背面。

3）如果第一次背面向上，则下次选头像面。

4）如果连续两次都是头像面或背面，则重新开始（再抛两次）。

问：Calvin的方法是否符合算法标准？

e. 给定一个数，执行以下步骤：

1）乘以4

2）加4

3）除以2

4）减2

5）除以2

6）减1

7）此时，给计数器加1（表明刚执行了一遍如1~6的操作步骤）；同时判断刚才的减1结果（第6步）；如果为0，记下此时计数器的值，并停止；如果不为0，则从该减过1的数开始，再次执行以上的1~7步。

1.13 两台计算机*A*和*B*，除了*A*具有减法指令外，其他指令完全相同。两者都具有对一个数求负值的指令。试问：*A*和*B*两台计算机，哪一台能解决的问题更多？证明你的结论。

1.14 假设我们试图对一组名字做字母排序（sorting）。有一种算法为“冒泡排序（bubble sort）”；我们用C语言编写这个算法，对其编译并运行在x86 ISA的机器上；x86 ISA可以实现为Pentium IV微结构的方式。我们称这样一个序列为“冒泡排序、C语言编程、x86 ISA、Pentium IV微结构”的转换过程（transformation process）。再假设我们有四种排序算法，且可以用5种语言编程（C、C++、Pascal、Fortran、COBOL）；有面向x86和SPARC两种ISA的编译器，且有三种x86的微结构实现、3种SPARC的微结构实现。请问：

a. 共有多少种转换过程？

b. 列举其中三种转换过程；

c. 如果x86的微结构是2种（而不是3种），SPARC的微结构是4种（而不是3种），那么共有多少种转换过程？

1.15 将高级语言和底层语言相比较，列举一个优点、一个缺点。

1.16 列举至少三个ISA定义所包含的内容。

1.17 简单描述ISA和微结构之间的区别。

1.18 一种微结构可以实现多少种ISA？再反问，一种ISA可以在多少种微结构上实现？

1.19 列出转换过程的所有层次，并在每层中找一个例子。

1.20 图1-6所示的转换层次通常又被称为是不同的抽象层次。你认为这种说法是否合理？试用例子加以解释。

1.21 假设你去商店购买字处理软件。请问该软件通常以什么方式存在？是高级语言方式或是汇编语言？或是与你的计算机ISA兼容的格式？请回答。

1.22 假设给你一个任务，完成图1-6中某一层的转换工作，且只允许转换为下面的一层（即不允许“跨层转换”）。试问，你觉得哪一层的转换工作难度最大？为什么？

1.23 为什么通常会不断改变微结构实现却不改变ISA？例如，Intel公司为什么要确保Pentium III所实现的ISA一定要与之前的Pentium II一样？提示：当你升级计算机（如更换最新的CPU）之后，你希望丢弃原来的软件吗？