Structured computer organization

数字计算要是通过执行人们给出的指令解决问题的机器。描述如何完成一个确定任务的指令序列称为**程序**。计算机的电子电路都只能识别和直接执行有限的简单指令，所有程序都必须在执行前转换成这些指令。这些基本的指令几乎都不会比下面的指令更复杂：

两个数相加。

检查某个数是否为零。

将一些数据从计算机内存的某些单元复制到另外的单元中。

计算机的这些原始指令共同组成一种可供人和计算机进行交流的放言，我们称之为**机器语言**。设计一种新的计算机时，人们必须首先解决它的机器语言中包含哪些指令。通常，原始指令应尽量简单，兼顾计算机的使用要求和性能要求，目的是降低实现电路的成本和复杂度。正因为大多数计算机机器语言如此简单，所以使用起来就显得十分困难和乏味。

通过对计算机的这些简单描述，我们可以将计算机结构化为一系列抽象机，每台抽象机都建立在下层抽象机的基础上。这样，计算机的复杂性就在可控范围内，计算机的设计也可在有组织的和系统的状态下进行。我们可以把这种方法称为**结构化计算机组成**。

**1.1 结构化计算机组成**

正如前面提到的，在方便人使用和方便计算机实现之间存在着巨大的差距。人可能要做X，而计算机只会做Y，这就有问题了。本书的目的就是解释如何解决这个问题。

* + 1. **语言、层次和虚拟机**

这个问题可通过两种途径解决，两者都需要设计一个比内含的机器指令更便于人们使用的新的指令集合。这些新的指令集合也构成了一种语言，我们称不L1，对应地把机器中内含的机器语言指令的语言称为L0。这两种途径的不同之处在于，采取了不同的办法让只能执行用L0写的程序的计算机执行用L1写的程序。

一种途径是在执行用L1写的程序之前生成一个等价的L0指令序列来替换它，生成的程序全部由L0指令组成。计算机执行等效的L0程序来代替原来的L1程序，这种技术称为**翻译**。

另一种途径是用L0写一个程序，将L1的程序作为输入数据，按顺序检查它的每条指令，然后直接执行等效的L0指令序列，计算出结果。它不需要事先生成一个L0语言的新程序。我们把这种方法称为**解释**，把执行它的程序称为**解释器**。

翻译和解释其实是类似的。在这两种方法中，L1的指令最终都是通过执行等效的L0指令序列来实现的。区别在于，翻译时整个L1程序都先转换为L0程序，然后L1程序就被抛弃，新的L0程序被装入计算机中执行。在执行过程中，运行的都是新生成的L0程序，控制计算机的也是L0程序。

而解释时，每条L1指令被检查和译码之后将立即执行，不生成翻译后的程序。这里，控制计算机的是解释器，对它来说，L1程序只是数据。这两种方法都得到了广泛的应用，而且日益多用二者的结合。

与其顾名思义地去理解翻译和解释这两个概念，不如想象存在一种以L1为机器语言的假想的计算机（**虚拟机**）。让我们把这种虚拟机定义为M1（相应地，把原来的以L0为机器语言的实际的计算机定义为M0）；如果这种计算机的成本足够低廉，那就根本不需要L0语言，也不需要执行L0程序的机器了。人们可以单纯用L1写程序并让计算机直接执行。即使使用L1语言的虚拟机太贵或电子电路太复杂而不能实际使用，大家还是可以写L1语言的程序，这些程序可以用能直接被现有计算机执行的L0语言程序翻译或解释。换句话说，大家完全可以像虚拟机真正存在一样用它们的语言写程序。

为了使翻译和解释现实可行，两种语言L0和L1的差别不能“太”大。这条限制往往意味着，虽然L1比L0好一些，但对多数应用来说还是不够理想。这也许不能达到提出L1的最初目的，即减轻程序员不得不用更适合计算机（而不是人类）的语言来描述算法的负担。不过，还没到无可挽回的地步。

显然，解决问题的办法是发明一种比L1更面向人的指令集来取代它，这个指令集也使用一种语言，我们可以称为L2（对应的虚拟机称为M2）。人们可以像用L2作为机器语言的虚拟机真正存在一样，用L2写程序，然后将这种程序翻译成L1或由用L1写成的解释器来执行。

这种发明一系列语言，每种都比前一种更方便人们使用的步骤，可以无限地继续下去，直到最后找到一种合适的语言。每种语言都以前一种为基础，因此我们可以把使用这种技术的计算机看成一系列**层**，如图1-1所示，一层在另一层之上。最底部的语言或层最简单，而最上面的语言或层最复杂。

层－虚拟机－语言（指令集）－解释器或翻译器（下层的指令集编写） （操作系统也是由一些高级语言写的）上相对于其下层有更多的指令集，即更多的功能或特性。

第5层

用Ln编写的程序需经运行在

更低级别的机器上的解释器解释执行或

翻译成更低级别机器上的语言后执行

虚拟机Mn

（其机器语言为Ln）

虚拟机M3

（其机器语言为L3）

第4层

用L2编写的程序需经运行在

M1或M0上的解释器解释执行或

翻译成L1或L0语言后执行

虚拟机M2

（其机器语言为L3）

第3层

虚拟机M1

（其机器语言为L1）

用L1编写的程序需经运行在

M0上的解释器解释执行或

翻译成L1或L0语言后执行

第2层

用L0编写的程序

可直接由电子电路执行

实际的计算机M0

（其机器语言为L0）

第0层

图1-1 多层计算机

语言和虚拟机之间存在着重要的对应关系。每种机器都有由它能执行的指令组成的机器语言。也就是说，机器定义了语言。类似地，语言也定义了机器，即机器要能执行用这种语言写的所有程序。当然由某种语言定义的机器真正用电路实现的话，可能会十分复杂和昂贵，但我们依然可以想象它的存在。以C、C++或Java语言作为机器语言的机器确实会很复杂，但用今天的技术实现起来也不难。可是我们有充分的理由不去造这么一台机器：与其他技术比起来，它的成本太高。真正实用的设计应该同时具有成本上的优势，仅仅具有理论上的可行性是不够的。

一般意义上讲，有n层的计算机可被看成n台不同的虚拟机，每一台机器语言都不相同。我们交替使用术语“层”和“虚拟机”。只有用L0语言写的程序可以被电路直接执行，不用进行内部翻译或解释。用L1、L2、…、Ln写的程序必须经低层解释器解释或翻译成对应于低层的另一种语言。

为n层虚拟机写程序的程序员不必关心下层的翻译器或解释器，程序由计算机的结构来保证正确执行，而不必管它是由解释器一步步地执行后交给下一个解释器还是电路直接执行。对每种情况，结果都是一样：程序被执行了。

多数使用n层计算机的程序员只对顶层感兴趣，这一层与底层的机器语言差别最大。然而，那些想了解计算机到底是如何工作的人就必须研究所有的层，而那些对设计新计算机或新的层（新的虚拟机）感兴趣的人也必须熟悉所有的层。将计算机分为一系列层的概念和技术以及这些层的组成细节构成了本书的主题。

**1.1.2 现代多层次计算机**

多数现代计算机包含两层以上。甚至有多达6层的计算机存在，如图1-2所示。位于底部的第0层是机器的真正的硬件，它的电路执行第1层的机器语言程序。

第5层

面向问题语言层

翻译（编译器）

第4层

汇编语言层

翻译（汇编器）

第3层

操作系统机器层

指令系统层

设备层

微体系结构层

部分解释(操作系统)

第2层

解释（微程序）或直接执行

第1层

硬件

第0层

数字逻辑层

晶体管组成的逻辑门的复杂组合

图1-2 6层计算机 每层的支持方法（支持程序）在下部标明

在我们要研究的最底层－**数字逻辑层**，我们感兴趣的对象是门。虽然它们由模拟元件（如晶体管）做成，但也可以精确地建模为数字设备。每个门可以有一个或多个数字输入端（由0或1表示的信号），并可计算这些输入的一些简单逻辑函数（如与和或）并输出结果。门最多由几个晶体管组成，几个门可组成1位存储器，存放一个0或1。1位存储器可以组合成（如16、32或64一组）寄存器。每个**寄存器**可存放一个不大于某个最大值的二进制数。门本身也可组成主要的计算部件。

晶体管－逻辑门－1位存储器（计算单元）－寄存器（存放二进制数）

上面一层是**微体系结构层**。在这一层我们可以看到（一般）由8－32个寄存器组成的寄存器组以及名为**ALU**（Arithmetic Logic Unit，算术逻辑部件）的电路，ALU可以完成一些简单的算术运算。这些寄存器和ALU相连形成**数据通路**－其上是数据流。数据通路的基本操作选择一个或两个寄存器，让ALU对它们进行操作，（如将它们相加），然后将结果存回某个寄存器。

在一些机器上，数据通路的操作是由一个叫做**微程序**的程序控制的，而在另外一些机器上数据通路是直接由硬件控制实现的。本书的前三个版本中，我们都把这层称为“微程序层”，因为过去这几乎都是由软件解释器实现的。由于目前的许多机器经常由硬件直接控制数据通路，从本书第4版开始，我们把这层改名为微体系结构层来反映这种变化。

在由软件控制数据通路的计算机上，微程序可看作是第2层指令的解释器。它通过数据通路逐条对指令进行取指、译码和执行。例如，对ADD指令，将首先取出指令，对操作数寻址并送入寄存器，由ALU求和，最后把结果存回到正确的地方。而在硬件直接控制数据通路的计算机上，执行的步骤与此类似，但不存在一个真正的存储程序来解释第2层的指令。

第2层我们称之为**指令系统层**或**ISA**（Instruction Set Architecture）**层**。每个计算机制造商都会为他们出售的计算机出版一本手册，题为《机器语言参考手册》或《Western Wombat 100x型计算机的操作原理》之类。实际上，这些手册都是关于ISA层的，而不是底下的两层。手册中说的指令集，实际上是由微程序解释或硬件电路直接执行的指令。若某个计算机制造商为一台计算要提供两个解释器，用于解释两种不同的ISA层，那么，这台计算机需要两本“机器语言”参考手册，每个解释器一本。

再往上一层通常是混合层。这一层的大多数指令和ISA层相同。（也没有理由规定某层的指令不允许在其他层出现。）另外，这一层有新的指令集、不同的存储器组成、并发运行两个或多个程序的能力，以及其他一些特性。第3层在设计上存在着比第1层和第2层更多的变化。

第3层增加的新功能是由运行在第2层的解释器来实现的，第2层曾被称为操作系统。那些和第2层指令相同的第3层指令将直接交给微程序（或硬件），而不是由操作系统执行。换句话说，有些第3层指令由操作系统解释，而有些由微程序直接解释。这就是“混合”层的含义。本书从头至尾把这层称为操作系统机器层。

第4层和第3层之间有着根本的区别。下面三层并不是为普通程序员使用而设计的，主要是为支持高层所需的解释器或翻译器运行而设计的。而这些解释器和翻译器，是则专职设计和实现新的虚拟机的**系统程序员**所写的。第4层及以上各层才是供那些解决应用问题的应用程序员使用的。

第4层上的另一个变动是支持上层上方法。第2层和第3层都是解释，而第4层、第5层及以上各层通常（虽然不是全部）用的是翻译。

下面三层和上面各层的另外的不同是高层提供的语言的本质发生了变化。第1、第2、第3层提供的机器语言都是数字串，适合机器执行，但不容易被人理解。从第4层开始，提供的语言包含了人们能理解的单词和缩略语。

第4层，汇编语言层，实际上是某种低层语言的符号表示。本层为程序员写第2、第2、第3层程序提供了一种比直接写这些程序更舒服一些的方法。用汇编语言写的程序首先被翻译成第1、第2、第3层的语言，然后由相应的虚拟机或硬件解释执行。完成翻译过程的程序称为**汇编器**。

第5层的语言通常是供解决现实问题的应用程序员使用的。这些语言常称为**高级语言**。目前存在的高级语言有几百种，比较知名的和C、C++、Java、LISP和Prolog。用这些语言写的程序一般先由**编译器**翻译成第3层或第4层语言。虽然偶尔也解释执行的。例如，用Java语言写的程序通常先被翻译成一种类似于指令系统层的语言，然后被解释执行。

在某些情况下，第5层由某一特别领域（例如符号数学）的解释器组成，解释器提供该领域专业人员熟悉的去处和数据以解决该领域的问题。

总的来说，应该记住的关键一点是将计算机设计成一系列的层，每层建立在它的前一层之上，由不同的对象和操作表示不同的抽象机。用这种方式设计和分析计算机，我们能暂时忽略一些无关紧要的细节，使复杂的问题更容易理解。

每层的数据类型、操作和特性构成了该层的**体系结构**，它解决的是该层的用户能看到的问题。程序员能看到的特性（如可用的内存有多大）是体系结构的一部分；而具体的实现问题，如用哪种技术实现这种内存，就不属于体系结构的内容。**计算机体系结构**是研究如何设计程序员眼中的计算机系统的学科。一般实践中，计算机体系结构和计算机组成的含义基本相同。

* + 1. **多层次计算机的演化**

为展望多层计算机的未来，我们首先简单回顾一下计算机的发展历史，看一看过去这些年中层数和各层的属性有何变化。用真正的计算机机器语言写的程序（第1层）能直接被计算机的电路（第0层）执行，不需要经过任何的解释或翻译。这些电路和存储器及输入/输出设备一起，构成了计算机的**硬件**。硬件是具体的对象－集成电路、印制电路板、电缆、电源、存储器和打印机等，而不是抽象的概念、算法或指令。

而**软件**则正好相反，它是由**算法**（指明如何做某事的详细指令）及其在计算机中的表示即程序组成。程序可被存储在硬盘、软盘、CD-ROM或其他存储介质上，但软件本质上是构成程序的指令的集合，而不是记录它们的物理介质。

在早期的计算机中，硬件和软件之间的界限十分清楚。然而，随着时间的推移，由于计算机层次的增减和合并，界限变得越来越模糊。到现在，已经很难区分它们了。事实上，本书的一个中心主题就是：

硬件和软件在逻辑上等同的。

任何由软件实现的操作都可直接由硬件来完成，尤其是在操作被人们充分认识之后。正如Daren Panetta Lentz所说的：“硬件就是固化的软件。”当然，这话反过来说也是正确的：任何由硬件实现的指令都可由软件来模拟。将哪些特定的功能由硬件实现，哪些功能由软件实现，是取决于当时的成本、速度、可靠性和预期的修改频率这些因素的。很少能有确定的规则规定X必须由硬件实现而Y必须通过编程实现，而且这些规定也会随着计算机技术发展的趋势和计算机应用范围的变化而改变。

1．微程序技术的出现

回到20世纪40年代，最早的计算机只有两层：编制所有程序的ISA层和执行这些程序的数字逻辑层。数字逻辑层的电路非常复杂难懂，难以生产制造且不可靠。

1951年，剑桥大学的研究员Maurice Wilkes提出设计一个三层计算机来极大地简化硬件设计的主意。这种机器需要内置一个不可修改的解释器（微程序），来解释执行ISA层的程序。这样，硬件就只需执行仅有少数指令集的微程序，而不是执行指令集大得多的ISA层程序，因而只需要较少的电路来实现。由于当时的电路都是由真空管制造的，这种简化减少了真空管的数量，因此也提高了可靠性（即减少了每天系统崩溃的次数）。

20世纪50年代仅仅生产了少量这种三层计算机，60年代就多了一些。到1970年，由微程序代替电路解释ISA层的计算机占据了主导地位。那里所有的主流计算机都用到了这项技术。

2．操作系统的出现

在计算机出现的早期，大多数计算机都是“自选商场”，程序员必须自己操纵计算机。几乎每台计算机都有一个预约单，想上机运行程序的程序员首先要预约机时，比如周三凌晨3点到5点（许多程序员愿意在机房安静时上机）。到了预约时间，程序员一手拿着一叠80列的穿孔卡片（一种早期的输入介质），另一手拿着削好的铅笔动身前往机房，礼貌地请前一个程序员离开，然后接管计算机。

如果程序员要运行Fortran程序，就得经历下面几个步骤：

1. 走向程序库的柜子，找出标有Fortran编译器的绿色卡片，放入读卡器中，按下“启动”按钮。
2. 将自己编写的Fortran程序放入读卡器，按“继续”按钮，计算机读入程序。
3. 计算机停下时，再次将卡片读一遍。虽然有些编译器只要求输入一次程序，但许多是要求读两遍或更多遍的。每一遍都得读入一堆卡片。
4. 终于，翻译过程接近结束了，而程序员的心跳往往也在这时候加快，因为，若编译器发现程序错误，他将不得不将错误修改后再走一遍上述的全部过程。如果没有错误，编译器在卡片上打孔输出翻译好的机器语言程序。
5. 然后，程序员将卡片上的机器语言程序同子程序库卡片一起放入读卡器，将它们一起读入。
6. 开始执行程序。多数情况下计算机会出故障或在中途死机。一般来说，程序员会胡乱拨弄控制台上的开关或对着控制台的指示灯发一会儿呆。运气好的话，可能找到问题，修改程序中的错误，然后从头再来一遍；运气不好时，只能是将内存中的内容打印出来（称为**核心转储**），带回家去研究。

这个场景，多年来在很多计算中心都出现过，只是细节略有不同。它使得程序员不得不学会操纵计算机，并要学会如何处置频繁出现的计算机死机现象。更为可气的是，当人们抱着卡片在机房外等着上机，或正为分析程序不能正常运行的原因而挠头时，计算机却处于空闲状态。

到1960年左右，大家试图通过将这些工作自动化来减少时间的浪费。计算机中出现了一个称为**操作系统**的常驻程序，它一并读入和处理程序员提供的程序和控制卡片。图1-3给出了第一个广为使用的操作系统，即运行在IBM 709上的FMS（Fortran Monitor System,Fortran监控系统）的作业示例。

\*Job,5494,BARBARA

\*XEQ

\*Fortran

Fortran程序

\*DATA

\*END

数据卡

图1-3 FMS操作系统作业示例

操作系统先读入\*JOB卡，用它上面的信息进行作业登记，供记账使用。（星号用来标识控制卡片，使它们不致于和程序及数据卡片混淆。）然后，再读入\*Fortran卡，即从磁带上装入Fortran编译器的指令。装完后，编译器开始读入并编译用户的Fortran语言程序。再将控制返回给操作系统，然后读入\*DATA卡，以\*DATA卡后面的卡片作为输入数据，执行翻译好的程序。

虽然操作系统是为自动执行用户作业而设计的，但这同时也是开发新的虚拟机的第一步。\*Fortran卡可看作是虚拟的“编译程序”指令；类似地，\*DATA卡可被认为是虚拟的“执行程序”指令。只有两条指令对于构成一个新层来说是少了点，但这代表一个新的发展方向。

随后的几年，操作系统变得越来越复杂。新的指令、设施和特性不断地加入到ISA层中，直到形成一个新层。新层中的一些指令和ISA层指令完全相同，但另外一些指令，尤其是输入/输出指令，则是全新的。这些新指令当时被称为**操作系统宏**或**超级用户调用**，现在通称为**系统调用**。

在其他方面，操作系统也得到发展。早期的操作系统先读入卡片，然后在行式打印机上输出结果，这种组织称为**批处理系统**。从提交程序到结果出来通常得等待几个小时，在这种环境下，开发程序十分困难。

20世纪60代，达特茅斯学院、MIT等机构的研究人员开发出允许多个程序员直接和主机通信的操作系统。在这些系统中，远程终端通过电话线和中心计算机相连，计算机由多个用户共享。程序员可在办公室、家中的车库或放入终端的任何地方输入程序，并立即得到结果。这些系统被称为**分时系统**。

我们感兴趣的是操作系统如何解释第3层存在的而ISA层不具体的指令和特性，而不讨论其分时机制。虽然我们没有强调，但你应当记住，操作系统并不是仅仅解释ISA层上的特性。

3. 从函数到微码

到1970年，微程序编程已经很普遍，设计者意识到他们可以通过扩充微程序来增加新指令。也就是说，可以通过编程来增加“硬件”（新的机器指令）。这个发现导致机器语言指令集合的“爆炸”，因为设计者会竞相创造出更大和更好的指令集。许多指令并非特别需要，因为它们的功能可以用已有指令很容易地实现，但新指令一般会比执行一系列已有指令快一点。例如，许多机器有往一个整数上加1的INC指令；由于这些机器上也有一条通用的加法指令ADD，应该说再多一条加1（或者加720也一样）指令是没有必要的。然而，INC指令比ADD快一点，也就加进来了。

许多其他的指令也由于同样的理由加入到微程序中。其中包括：

1. 整数乘除指令。
2. 浮点数算术指令。
3. 过程调用和返回指令。
4. 加速循环的指令。
5. 处理字符串的指令。

而且，一旦计算机的设计者认识到增加一条指令是如此容易，他们便开始四处寻找可以加入微程序中的特性。下面就是他们找到的一些：

1. 加速数组（索引和间接寻址）运算的特性。
2. 允许程序启动后在内存中移动的特性（重置设施）。
3. 输入/输出操作完成时通知计算机的中断系统。
4. 用少量指令挂起一个程序并启动另一个程序的能力（进程切换）。
5. 处理音频、图像和多媒体文件的特殊指令。

这些年来，还有数不清的其他特性和功能被加入，一般都是为了提高某一特定操作的速度。

4. 微程序的消失

微程序在它的黄金年代（20年代60和70年代）发展得十分壮大，伴随而来的问题是它也变得越来越慢。终于，一些研究人员认识到，应该取消微程序，大规模地缩小指令集，并将剩下的指令直接由硬件执行（即硬件直接控制数据通路），这样可以提高计算机的速度。大家的共识是，计算机设计的发展画了一个圆圈，又回到Wilkes发明微程序之前的老地方了。

但历史的车轮还在继续转动。Java程序的一般执行过程是：先将它编译成一种中间语言（Java字节码），然后对其解释执行。

说了这么多，关键是想表明计算机硬件和软件的界限完全是人为划定的，并且经常变化。今天的软件也许就是明天的硬件，反过来也是一样。甚至层与层之间的界限也是不断变化的。从程序员的观点看，指令实际上是怎样实现的并不重要（除非是为了提高速度）。ISA层的程序员可以将他的乘法指令当作硬件指令，而不必担心（甚至根本不用去考虑）它是否真正是由硬件实现的。一个人的硬件是另一个人的软件。

第k层的设计很大程序上依赖于第k-1层的属性，如果不了解低层，也就无法理解上一层。

对于指令系统层，多数计算机供应商在广告上把该层叫做机器语言。

计算机是由包含有很多被称为门的微型开关的集成电路制造而成的。

计算机通过加法器来完成算术运算。1位全加器可以由两个半加器构成，多位加法器则是由多个全加器组成的，每个全加器都把自己产生的进位伟给它左边的那个全加器。

（静态）存储器由锁存器和触发器组成，它们都能存入一位信息。可以将它们线性组合成8位锁存器和触发器，或者是2的指数位的实际使用的任何内存储器。内存可以分为RAM、ROM、PROM、EPROM、EEPROM和闪存。静态RAM不需要刷新，只要不断电，它可以一直保存数据。与此相反，动态RAM必须定期刷新，以补偿芯片中小电容的电荷的泄露。

计算机系统中的各个部件通过总线连接在一起。一般CPU的许多（但不是全部）管脚可以直接驱动一个总线信号。总线信号可以分为地址信号、数据信号和控制信号。同步总线由同一个主时钟驱动，异步总线以全握手方式来使从设备和主设备同步。

微体系结构层的作用是实现位于其上的指令系统层。微体系结构层的设计是由其实现的指令系统层决定的。当然计算机的价格和性能也是需要考虑的因素。许多现代的指令系统层，特别是RISC设计中的指令系统层，其指令都很简单，通常可以在一个时钟周期内执行。复杂一些的指令系统层，例如Pentium 4，执行一条指令可能需要多个时钟周期。执行指令的过程可能包括在内在中寻址操作数、读入操作数和把结果存回内存。当执行一条指令需要多个操作步骤时，其控制方式就和简单的指令系统层不同了。

微体系结构层中不存在一般性的原理，几乎每种设计都是一个特例。我们以Java虚拟机JVM指令系统层的子集作为指令系统层的例子。这个子集只包括整数指令IJVM。我们的微体系结构层包括一个微程序（存放在ROM中），它的功能是取出、译码并执行IJVM指令。

指令系统层是在所有其他层次之前发展起来的，实际上，最早出现的计算机只有这一层。指令系统层是硬件和软件之间的接口，这一特点使它对于计算机系统设计者来说尤为重要。尽管我们可以硬件直接执行用C、C++、Java或者其他高级语言编写的程序，但这并不是一种好办法。因为这样我们将丧失编译执行相对于解释执行的性能优势。此外，为了介于使用，大多数计算机都应该能够执行用多种语言编写的程序，而不仅仅是一种。

所有的系统设计者都采取本质上一样的策略：把各种不同的高级语言程序转换成一种通用的中间形式－指令系统层程序－再设计能直接执行指令系统层程序的硬件。指令系统层定义了硬件和编译器之间的接口，它是一种硬件和编译器都能理解的语言。编译器、指令系统层和硬件之间的关系如图5-1所示。

C程序编译为

指令系统层程序

C程序

FORTRAN 90程序

FORTRAN 90程序编译为

指令系统层程序

软件

指令系统层

硬件

通过微程序或硬件执行

指令系统层程序

硬件

图5-1 指令系统层是编译器和硬件之间的接口

本书的主要思想是现代计算机是由一系列的层次组成的，每层都在下一层的基础上增加功能。操作系统层是一个在指令系统层提供的指令和特性之上双增加了瓣指令和特性的程序。操作系统层指令集是应用编程人员完全中用的指令集，它包括几乎所有的指令系统层指令和操作系统层增加的新指令。这些新指令称为**系统调用**(system call)。一个系统调用使用一条指令调用一个预先定义好的操作系统服务，这样效率很高。操作系统总是解释执行的。

我们可以把操作系统看成是用于说明指令系统没有的体系结构特性的解释器。其主要部分有虚拟内在、虚拟输入/输出指令和用于并行处理的一些工具。

翻译器translator是这样一种程序，它可以把用某种语言编写的用户程序转换成另一种语言的程序。转换之前的程序使用的语言称为源语言，而转换之后的翻语言称为目标语言。在翻译过程中，使用源语言编写的程序并不直接执行。相反，在翻译结束后，它们将被翻译成可以直接执行的目标程序或可执行的二进制程序。

在计算机的内存中，只存在三种不同的程序：用户的目标程序、操作系统和微程序（如果有）。

根据源语言和目标语言的关系不同，可以把翻译器大致分为两大类。如果源语言基本上是数字型机器语言的符号表示，就把这种翻译器称为**汇编器**assembler,源程序相应地称为汇编语言。如果源语言是如Java或者C语言这样的高级语言，而目标语言是数字型机器语言或者是机器语言的符号表示（汇编语言），那么这种翻译器就称为**编译器**compiler.

在早期设计的机械计算装置中,使用的不是二进制，而是[十进制](http://baike.baidu.com/view/359301.htm)或者其他进制，利用齿轮的不同位置表示不同的数值，这种计算装置更加接近人类的思想方式。

比如说一个计算设备有十个齿轮，它们接连起来，每一个齿轮有十格，小齿轮转一圈大齿轮走一格。这就是一个简单的十位[十进制](http://baike.baidu.com/view/359301.htm)的[数据表示](http://baike.baidu.com/view/1026250.htm)设备了，可以表示0到999999999的数字。 配合其他的一些机械设备，这样一个简单的基于齿轮的装置就可以实现简单的[十进制](http://baike.baidu.com/view/359301.htm)加减法了。

[](http://baike.baidu.com/picview/2418160/2418160/0/fab3ac118b278840ca80c4d7.html)

Eniac采用的是十进制，用10根灯管排列，第n根灯管亮即代表n。