第一章　硬件基础知识

1.1　引　言

一个可供使用的计算机系统是由硬件和软件组成的。硬件是计算机系统中的实际装置，是系统的基础和核心（称为硬核），一般由中央处理机、存储器、输入／输出设备等组成，它以机器语言（即指令系统）提供给程序员使用。软件指的是操作系统、文本编辑程序、调试程序、汇编程序、编译程序、数据库管理系统、文字处理系统以及各种应用程序等。其中较低层次的程序与硬件密切相关，而用户在使用高级语言以及第四代语言编写程序时基本上已与硬件的实现无关，但是硬件的结构与性能对程序处理的速度影响极大。本章主要从程序员角度来观察与分析计算机的硬件结构及其对程序运行的影响。

另一方面，软件和硬件在逻辑功能上是等效的，即某些操作可以由软件，也可以用硬件来实现。所以软、硬件之间没有固定不变的分界面，主要受实际应用的需要及系统性能价格比所支配。从使用者来看，机器的速度、可靠性、可维护性是主要的硬件技术指标。具有相同功能的计算机系统，其软、硬件之间的功能分配可以有很大差异。回顾计算机的发展历史，在其早期，由于硬件昂贵，所以计算机的硬件比较简单，尽可能让软件完成更多的工作。但是随着组成计算机的基本元器件的发展，其价格不断下降，性能不断提高，因而造成硬件成本下降。与此同时，随着应用的不断发展，软件成本在计算机系统中所占的比例不断上升，这就造成了软、硬件之间的分界面的推移，即将某些由软件完成的工作交给硬件去完成，同时还提高了计算机实际运行速度。因此程序员，尤其是高级程序员具有硬件的基础知识是很必要的。

冯·诺曼型计算机的发展，按其所使用的元器件分成了四个阶段（称为四代）。第一代计算机的基本元器件是电子管，由于受电子管的寿命短、价格贵、体积大、能耗大等因素的影响，决定了当时计算机的硬件功能比较简单。第二代计算机的基本元器件是晶体管。第三代为集成电路。第四代为中、大规模及超大规模集成电路。由于其价格不断地大幅度下降，而可靠性却不断提高，因此计算机的硬件功能逐步增强，硬件体系不断完善。除了软、硬件的分界面发生变化外，大、中、小型计算机的界线也发生了变化。大、中型计算机的系统结构不断“下移”到小、微型机中去，高速缓冲存储器及虚拟存储器的普遍采用即为一例。

本章是在《软件人员水平考试辅导（程序员级）》一书的基础上编写的，凡是该书第一章“硬件基础知识”已有的内容在此不再重复。

本章的重点是计算机的组成及程序员能见到的体系结构。

1.2　指令系统

指令系统是计算机所有指令的集合，程序员用各种语言编写的程序都要翻译成（编译或解释）以指令形式表示的机器语言后才能运行，所以它反映了计算机的基本功能，是硬件

设计人员和程序员都能见到的机器的主要属性。早期的计算机的指令系统是很简单的，一些比较复杂的操作由子程序实现，因此计算机的处理速度比较慢。随着硬件价格的下降，指令系统逐步扩充，指令的功能也逐步增强。指令系统的改进途径是围绕着缩小与高级语言的语义差异以及有利于操作系统的优化而进行的。例如，对高级语言中的实数计算是通过浮点运算进行的，在计算机中设置浮点运算指令能明显地提高速度；另外在高级语言程序中经常用到IF语句、DO语句等，为此设置功能较强的“条件转移”指令是有好处的；为了便于实现程序嵌套，而设置了Call及Return指令……上述这些措施都是针对高级语言的优化而进行的，最终使生成的目标程序短而且运行速度快，同时也便于编译。至于为操作系统的实现或优化而设置的特殊指令，除了各种控制系统状态的特权指令外，还有为多道程序公用数据管理和多处理机系统信息管理用的“测试与置定”和“比较与交换”等指令。除此以外，中断系统和存储体系对操作系统的支持与影响也很大，这将在后面讨论。

1.2.1 指令系统的分类

计算机的指令系统没有统一的分类标准，大体上可分成以下几类：

（1）算术逻辑运算指令。

这里讲的算术运算一般指的是对定点数进行计算，即相当于高级语言中对整数（Integer）的处理。逻辑运算一般指的是对字或字节进行几种逻辑操作处理及移位处理。

（2）传送指令。

一般指的是在寄存器与存储器之间、寄存器与输入输出端口之间、存储器与存储器之间的各种传送操作。其中大部分有时还称为访问存储器指令、存取（Load/Store）指令。

（3）程序转移指令。

一般指的是控制程序流的无条件转移指令、条件转移指令、转子程序指令及返回指令等。在某些计算机中还有功能较强的调用(Call)指令及循环控制指令。

（4）控制指令。

这组指令允许程序去控制处理机与外部事件同步、进行存储管理以及对程序状态字或标志寄存器(或其中某几位)进行设置与修改等。在多道程序和多处理机系统中，有相当一部分控制指令被称为特权指令。在这种系统中，程序被区分成运行在操作系统环境和用户环境中两种情况，前者称为管态（即管理状态），后者称为目态（即用户状态）。在管态情况下，可运行全部指令，而在目态情况下，不允许运行特权指令。若不慎误用，处理机将自动按出错处理，其目的是防止多个用户之间或多个处理机之间的相互干扰而破坏对方（其他用户）的正常工作。

（5）外部指令（I/O）指令。

某些计算机有专设的I/O指令，对外部设备的起动、停止、数据传送等进行控制。但也有不少计算机没有专设的外部指令，而是将外部设备的数据寄存器、控制寄存器等与存储器统一编址。

（6）浮点运算指令。

高级语言中的实数（Real）经常是先转换成浮点数的形式而后再进行处理。某些机器没有设置浮点运算指令而用子程序实现，其速度较低。因此主要用于科学计算的计算机应该设置浮点运算指令，能对单精度（32位）、双精度（64位）数据进行处理。

（7）十进制运算指令。

在人机交互作用时，输入输出的数据都是以十进制形式表示的。在某些数据处理系统中输入输出的数据很多，但对数据本身的处理却很简单。在不具有十进制运算指令的计算机中，首先将十进制数据转换成二进制数，再在机器内运算；尔后又转换成十进制数据输出。因此，在输入输出数据频繁的计算机系统中设置十进制运算指令能提高数据处理的速度。

（8）字符串处理指令。

这类指令包括字符串传送、字符串比较、字符串转换等指令。其中“字符串传送”指的是数据块从主存储器的某区域移动到另一区域；“字符串比较”可以是一个字符串与另一字符串进行比较，也可以是从某一字符串中寻找某一指定的符号（字符或字符串）；“字符串转换”指的是从一种数据表达形式转换成另一种形式，例如从ASCII码转换成EBCDIC码（扩充的二—十进制交换码）。

1.2.2 指令系统的兼容性

上述指令系统的分类不是一成不变的，还有不少指令没有包括在内。各计算机公司设计的计算机，其指令的数量以及各条指令的功能也是各异的，即使是一些常用的基本指令，如算术逻辑运算指令、转移指令等也是各不相同的，因此尽管各种型号计算机的高级语言基本相同，但将高级语言程序（例如FORTRAN语言程序）编译成机器语言后，其差别是很大的。因此将用机器语言表示的程序移植到其它机器上去几乎是不可能的。从计算机的发展过程已经看到，由于构成计算机的基本硬件发展迅速，计算机的更新换代是很快的，这就存在软件如何跟上的问题。大家知道，一台新机器推出交付使用时，仅有少量系统软件（如操作系统）可提交用户，大量软件是不断充实的，尤其是应用程序，有相当一部分是用户在使用机器时不断产生的。为了缓解新机器的推出与原有应用程序的继续使用之间的矛盾，1964年在设计IBM360计算机中所采用的系列机思想较好地解决了这一问题。从此以后，各个计算机公司生产的同一系列的计算机尽管其硬件实现方法可以不同，但指令系统、数据格式、I/O系统等保持相同，因而软件完全兼容（在此基础上，产生了兼容机）。当研制该系列计算机的新型号或高档产品时，尽管指令系统可以有较大的扩充，但仍保留原来的全部指令，保持软件向上兼容的特点，即低档机或旧机型上的软件不加修改即可在新机器上运行，以保护用户在软件上的投资。

1.2.3 数据表示

在计算机中的基本数据有逻辑（布尔）数、定点数（整数）、浮点数（实数）、十进制数、字符串、数组等。对这些数据的运算可以设置专门的指令；也可以仅设置最简单的算术逻辑运算指令，而通过程序的执行实现之，但后者的速度下降很多。在机器中若设置能直接对矩阵向量数据（数组）进行运算的指令及硬件，可以大大提高对向量、数组的处理速度，这一般在巨型机中才采用。

目前计算机所用数据字长一般为32位，即使是微型机的字长也从8位、16位发展到32位。在存储器中的地址，一般按字节表示。计算机的指令系统可支持对字节、半字、字、双字的运算，为便于硬件实现，一般要求数据对准边界，如图1.1所示。

1.2.4 寻址方式

程序和数据是存放在存储器中的。所谓寻址指的是如何确定数据地址及转移指令的下一

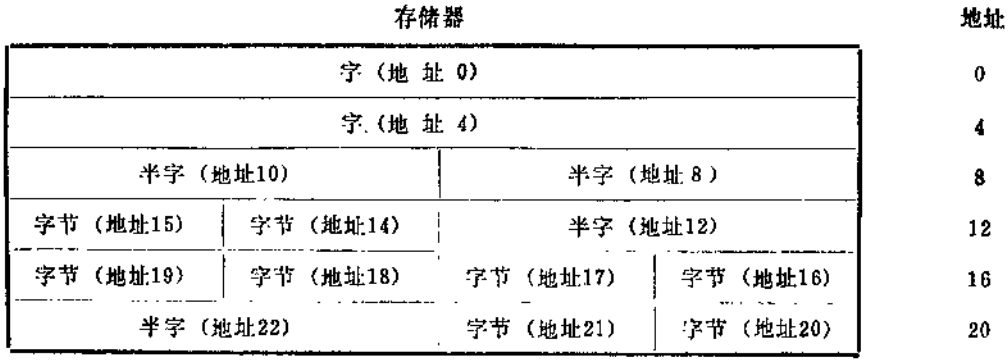


图1.1 存储器中数据的存放

条要执行的指令地址。

每条指令的寻址方式由指令本身确定，或者按某些预先约定的规则进行。可以有按地址访问、按堆栈访问和按内容访问等方式，另外还可以在指令中直接表示出数据，称为立即数。与指令系统一样，各种计算机的寻址方式也是各不相同的。

堆栈是在存储器中的一个先进后出存储区，除了有的机器设置专门的堆栈指令，对存放在堆栈顶部的两个数据进行运算以外，一般用于程序嵌套时保留现场数据及程序地址（PC）用。在堆栈结构中，有一堆栈指针指出栈顶的地址。当往堆栈内送一数时称为压入（PUSH），从堆栈内取出一数时称为弹出（POP）。当进行压入或弹出操作时，栈顶将变化，堆栈指针也跟着变化，始终指向栈顶的地址。

按内容访问适用于查询，在一般计算机中设置这样的指令尚不多见。按内容访问的相联存储器将在“存储体系”这一节中讨论，在这里我们主要讨论按地址访问。

按地址访问对一般指令来讲即是由机器指令指明（或经计算后得到）操作数所在地址；对转移类指令来讲，就是下一条即将执行的指令地址，经常用到的编址方式主要有直接编址、间接编址、基址编址、变址编址、相对编址等。在讨论以前，我们先介绍一下逻辑地址和物理地址的概念。

逻辑地址指的是程序员编程时所指定的程序（指令）地址和数据地址；物理地址指的是机器实际执行时的程序（指令）地址和数据地址。在早期程序员编程时用的程序地址和数据地址就是实际地址，因此当时并不区分逻辑地址和物理地址。但是后来随着多道程序以及汇编语言、高级语言的出现，由于各道程序是独立编写的，因此由程序员确定物理地址存在困难，很容易产生存储冲突，再加上硬件结构上的一些特点，例如存储器容量的可扩充性，采用虚拟存储器方案等，决定了用户的源程序经编译后得到的是逻辑地址，然后在程序链接时或程序运行时转换成物理地址。图1.2表示有A、B两道程序，每道程序的逻辑地址都从零开始，然后由操作系统各自分配到a～a＋l和b～b＋k。在具体实现时，可采用基址编址的方法。

一、基址编址

在执行A道程序之前（图1.2），先在基址寄存器中存放存储器的起始地址a，然后在执行指令时，由硬件将指令中所表示的地址（逻辑地址）和基址值a相加，形成有效地址（物理地址），这就是基址编址的原理（图1.3）

另外也可以用基址扩充寻址范围，如在16位微机中，16位地址码的寻址范围为64k字节，如将基址左移4位与指令地址相加，即可得20位物理地址，寻址范围扩大到1M字节，IBMPC就是这样处理的。

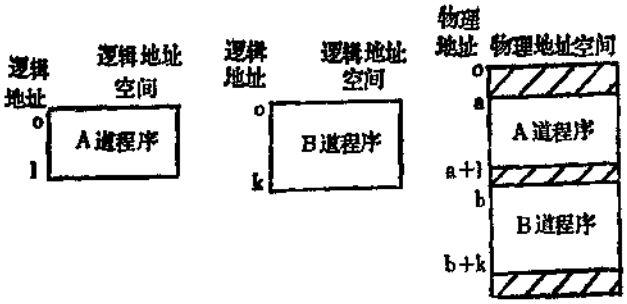


图1.2 逻辑地址空间和物理地址空间

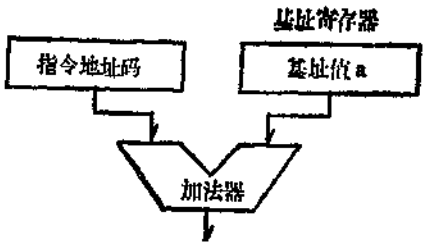


图1.3 基址编址

值得提出的是，在计算机中将逻辑地址转换成物理地址的方法很多，千差万别，在此处不再详细说明。

二、变址编址

这是几乎所有计算机都采用的一种编址方式。变址操作指的是将指令中所指出的地址码与变址寄存器中的内容相加，形成有效地址（图1.4）。

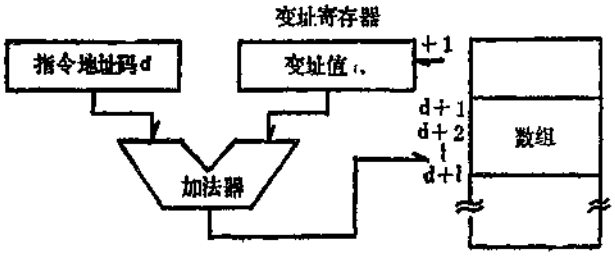


图1.4 变址编址

图1.4表示变址操作对处理一维数组的支持，用户需编写对数组中一个元素进行运算的程序，然后改变变址寄存器的值（从1→l），对程序循环执行l次，就可以对数组中的l个元素逐个进行处理，这就是利用变址操作与循环执行程序的方法对整个数组进行运算的例子，在整个执行过程，不改变原程序，因此对实现程序的重入性是有好处的。至于二维数组，也可用变址操作实现，但需要二个变址寄存器。

三、直接编址、间接编址

在指令中直接指出操作数地址，称为直接编址，根据此地址即可直接取得操作数。

假如按上法取得的不是操作数，而仍是操作数地址，则称为间接编址。有的计算机还存在多级间接编址方式。例如有下述两个程序：

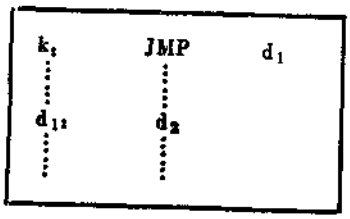


图1.5 直接编址

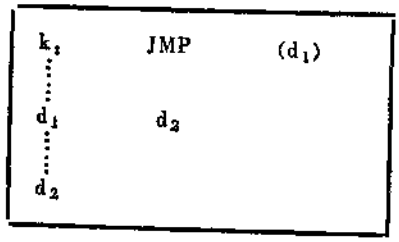


图1.6 间接编址

其中JMP为转移指令的操作符，d_1为地址，执行图1.5的第k条指令后，将转移到d_1，而执行图1.6的第k条指令后，将转移到d_2。假如在某一存储区存放中断向量表（即表内存放的内容是中断程序入口），那么执行间接编址转移指令可直接进入相应的中断程序入口。

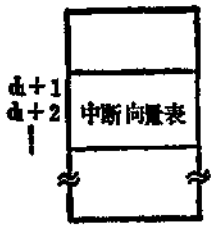


图1.7 存放中断向量表的存储器

四、相对编址

主要用于转移指令，指令中地址码给出的是相对于当前指令的位移值。如有下列指令格式：

操作码	地址码 d

执行本条指令后，将转移到PC＋d。

五、寄存器编址

为提高计算机运算速度，减少访问存储器次数，在计算机内设置多个通用寄存器，参与运算的数存放在寄存器内，指令对这些寄存器直接寻址。

六、立即数

在指令的地址码部分存放的是参与运算的数据。

以上这些编址方式，在计算机内可以组合使用，例如可以在一条指令执行过程中同时实现基址编址和变址编址，其有效地址为：

基址寄存器内容＋变址寄存器内容＋指令地址码d

一台计算机究竟允许用何种编址方式，还需要查阅有关指令系统的说明书，同一种编址方式在不同的计算机中的实现还会有差别。

用户假如用高级语言编程，根本不用考虑寻址方式，因为这是编译程序的事。但若用汇编语言编程，则应对它有确切的了解，才能编出正确而有效的程序。

最后要提醒一下，对虚拟存储器来讲，由于其逻辑空间比实存空间大很多，用上述方法求得的有效地址仍称为逻辑地址，需要通过硬件与软件的配合才能完成逻辑地址到物理地址的转换（见存储体系）。

1.2.5 指令格式

指令由操作码和地址码两部分组成，上述各种编址方式主要反映在指令的地址码部分，地址码可以有0地址、1地址、2地址、3地址等多种形式。随着指令类型的不同，对地址码的长度要求变化很大，例如操作数在寄存器内比之在存储器内，其地址码的长度要短得多。为了缩小程序代码所占的存储容量，在计算机中，各类指令的长度可以不一致，例如在同一计算机中可以有1字节、2字节、3字节、4字节等多种长度的指令。从压缩代码的观点出发，我们希望常用的指令的操作码短些，这样最后使程序的长度也短些。假如某计算机模型有7条指令（I_1～I_7），它们在程序中出现的概率用Pi表示，则可考虑图1.8所示的方案。这就是扩展操作码。使用频率高的指令的操作码为2位，低的用4位。这不是压缩到最小代码的方案，因为在计算机中的操作码还是希望有一定的规整性，否则会引起硬件实现的复杂化。另外在计算机内存放的指令长度应是字节的整数倍，所以操作码与地址码两部分长度之和应是字节的整数倍，在考虑操作码长度时还应考虑地址码的要求。

指令	概率P_i	操作码	操作码长度
I_1	45%	00	2位
I_2	28%	01	2位
I_3	17%	10	2位
I_4	5%	1100	4位
I_5	3%	1101	4位
I_6	1%	1110	4位
I_7	1%	1111	4位

图1.8 指令出现的概率与操作码长度的选择

K+2		K+1	K
空白	K+4	K+3	
K+5			
K+9	K+8	K+7	K+6

图1.9 变长度指令的存放(K，K+1…都是指令)

K	操作码	地址码	
K+1	操作码	空白	地址码
K+2	操作码	地址码	
K+3	操作码	地址码	

图1.10 定长度指令(定长度操作码)的存放

K	操作码		地址码	地址码
K+1	操作码	地址码	地址码	地址码
K+2	操作码			地址码
K+3	操作码	地址码	地址码	地址码

图1.11 定长度指令(变长度操作码)的存放

图1.9～图1.11是指令在存储器中存放的例子，假设存储器的字宽为32位。

图1.9为变长度指令的存放，在一个存储器单元中可存放1～4条指令，由于要求指令字边界对齐，因而在存储器中可能有无用的空白。图1.10为定长度指令，由于操作码长度固定，而不同指令对地址码要求不同，甚至有的指令不需要地址码，因而也会出现空白。图1.11的操作码长度主要由地址码决定，这时可能有较多的没有用到的操作码存在。总之指令格式的变化是很多的，其方案也很多，而且总是有冗余情况存在。

1.2.6 指令系统的分析

指令系统从早期的简单形式，逐步发展到多种寻址方式、多种数据格式的复杂指令集。

对已有机器的指令系统进行分析，了解这些指令的实际使用情况，可为下一步的改进以及新型计算机的设计提供资料。图1.12为在IBM370计算机上用不同语言编写的程序所出现的指令的频率。

上述统计数据是从机器运行时测得的动态频率，而若根据程序代码统计则为静态频率，两者虽不完全相同，但有一定关系，从图中可以看到，浮点运算指令对科学计算是需要的。

从分析中还可见到，执行程序的80%是存取、转移、算逻运算等简单指令，复杂指令仅占20%。但为控制20%的复杂指令而设置的微程序控制代码却占控制存储器的80%，这说明编译程序翻译成的目标代码与机器设计人员的设想有一定距离。复杂指令并没有给编译程序的优化带来多大好处。在此基础上提出了精简指令系统计算机，称为 RISC (Reduced Instruction Set Computer)。而把以前传统的计算机称为CISC(复杂机器指令系统)。

指 令	COBOL	FORTRAN	Pascal
转移	24.2%	18.0%	18.4%
逻辑操作	14.6%	8.1%	9.9%
存取	40.2%	48.7%	54.0%
存储器—存储器传送	12.4%	2.1%	3.8%
整数运算	6.4%	11.0%	7.0%
浮点运算	0.0%	11.9%	6.8%
十进制运算	1.6%	0.0%	0.0%
其它	0.6%	0.2%	0.1%

图1.12 各种高级语言中指令出现的频率

RISC采用32位固定长度的指令，指令格式固定，寻址方式少，所以便于编译优化，并减少硬件控制的复杂性。

RISC的机器内有为数较多的寄存器，指定算术逻辑指令的操作数都在寄存器内，访问存储器的只有Load/Store指令，使大多数指令都能在一个机器周期内完成。

执行一个程序所需的时间P可用简单的公式表示：

$P = I \times C \times T$,

其中I为执行的指令数，C为实现每条指令的平均机器周期数，T为每个周期的时间宽度。RISC和CISC的数据如下表：

	I	C	T
RISC	1.2～1.3	1.3～1.7	<1
CISC	1	4～10	1

RISC的程序代码长度大于CISC，其原因是：CISC中采用的复杂指令在RISC中用子程序来实现，统一的32位指令字长度也会使程序的代码增加。

从以上数据可以看到，RISC的程序长度虽有所增加，但增加得并不多，而每条指令执行的平均周期数却减少得很多，因此运算速度可提高几倍，同时因为基本CPU简化了，可以将原来在CPU片外的部分电路集成到片内，于是可进一步减少每个周期的时间宽度，更加提高了机器的速度。

指令系统的发展从简单趋向于复杂主要的出发点是：

（1）尽量缩短指令与高级语言语义的差距；

（2）提高操作系统的效率。

由此希望达到提高速度的目的。但最后发现再进一步增强指令功能已不能取得好的效果，反而带来硬件结构极其复杂的后果，因此再循此道路走下去已没有发展前途，所以回过头来又从精简指令入手寻求出路。但是完全的RISC系统也不理想，所以RISC和CISC的结合是当前新结构的研究课题之一。

1.3 计算机的基本组成及其相互联系

1.3.1 构成计算机系统的基本元器件

随着半导体工艺的发展及大规模、超大规模集成电路的广泛应用，当前世界上大多数计算机的中央处理机是由下述两种方法组成的。

（1）采用半导体公司（或工厂）生产的微处理器芯片构成通用的或专用的计算机系统。当前微处理器芯片已从8位、16位发展成32位结构，诸如Intel公司的80386微处理器，Motorola公司的68020微处理器等均为32位，这些大批量生产的微处理器作为市场商品出售。

（2）一些大计算机公司采用自行设计制造的CPU芯片来构成各种大、中、小型计算机，例如IBM与DEC等公司就是这样做的。由于这些器件的生产量相对来讲比较小，所以设计研制成本由计算机系统来承担，而且这些芯片不在市场上出售，因此仿制（兼容机）就困难了。

除了CPU芯片以外，还有大量的电路与之配合，常用的有浮点处理器、存储器(RAM、ROM、PROM、EPROM)、各种接口电路以及PAL、PLA等芯片，近年来在计算机中还广泛应用门阵列电路（Gate Array），门阵列为用户定制电路。

1.3.2 构成计算机系统的基本组成部件

图1.13为一般计算机的简框图。由CPU、浮点处理器（FPU）、存储器（MEM），输入/输出（I/O）设备等模块组成，各部件之间通过系统总线相连。其中浮点处理器为选件（也有的计算机把浮点运算功能放在CPU中实现），存储器除了基本容量以外，还应该是可扩充的。I/O模块也为选件，但有一基本配置。有的机器在CPU模块内包含有一定容量的存储器及基本I/O设备的控制电路（接口电路）。

在计算机中，诸如CPU等是必须存在的，而某些模块可由用户自行选择以决定舍取，这些模块就是上面讲到的选件。

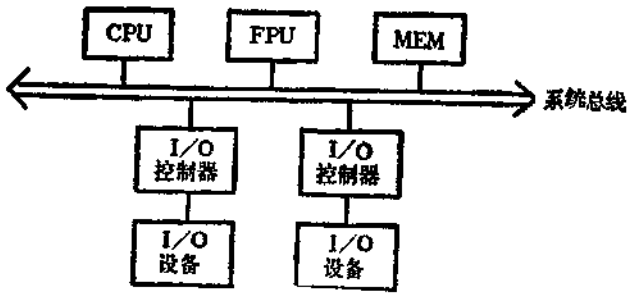


图1.13 计算机简框图

系统总线根据通过该线上的信息的作用而分成三部分：

（1）数据总线。

这是各部件间传送数据的通道，其宽度随字长而定，如为32位结构，则数据总线应是32

根．能向两个方向传送的总线，称为双向总线。

（2）地址总线。

从CPU送来地址的地址线，它可以是存储器的地址，也可以是I/O设备的控制寄存器或数据寄存器地址。

现代计算机，在I/O设备与MEM之间可以直接传送数据，此时在地址线上出现的是I/O设备送来的存储器地址。

（3）控制总线。

为实现各模块之间传送数据所需的一切控制信号是在控制总线上出现的。

在具有多个设备向总线发信号的系统中，在传送数据前，先要监听总线是否有空闲，有空才能占用总线，用毕后要释放总线。

传送数据的双方，处于主导地位的设备（即提出传送要求的设备）称为主模块（Master），处于应答地位的模块，称为从模块（Salve）。

假如各模块之间的所有信息都要通过系统总线相连，那么系统总线可能非常繁忙，以致影响计算机的处理速度，因此在某些计算机中，CPU、FPU与MEM之间的信息通过专门设置的内部总线进行传送。

比较著名的微机系统总线有Intel公司的Multibus，Motorola公司的VME bus，这些总线不仅规定了所有连接到连接器上的信号名称、性能，而且还规定了每根线在连接器上的插针位置。1985年2月，VME bus，作为IEEE P1014及IEC821的最终规范而确定了下来。全世界有不少微机系统及工作站采用VME bus，这种标准化与开放式结构使不同厂家生产的产品很容易组合在一个系统内，因而得到广泛的应用。

1.3.3 中央处理器CPU

CPU包括算术逻辑运算单元、寄存器、控制器、时钟等。一般还包括基本容量的存储器。它的主要功能是控制程序的执行以及完成对数据的处理。

CPU是按一定规则进行工作的，在计算机内必须有一时钟发生器，产生周期性变化的时钟信号，由若干个时钟组成一个机器周期。完成某一项处理工作，例如完成加法运算，或从存储器中取数等，处理器完成不同的操作所需的周期数可以不相等。不同的处理器，组成机器周期的时钟数也是不同的。

程序员编写的汇编语言程序或高级语言程序要编译成机器语言后才能由硬件执行，因此我们在本节讲到的程序都是指用机器语言（即指令系统）来表示的程序，它们与高级语言程序不同，是与机器密切相关的。

程序的执行过程：

用户程序由操作系统将其分配在主存储器中的某一存储区，在该区内是顺序存放的。执行程序是从该程序的入口开始的，先取出程序入口处的指令，完成该指令操作码所规定的操作，然后再取下一条指令，完成规定的操作……，直到取到一条停机指令为止。所谓停机实际是切断时钟信号的输出通路，CPU部件得不到时钟信号，就不再进行工作。在多道程序情况下，不允许停机，因为本道程序虽执行完毕，但其他程序可能还要继续进行，此时可由操作系统将CPU分配给其他程序使用。

图1.14是经过简化后的CPU简框图，该图的右半部为运算器，左半部为控制器。

一、运算器的组成

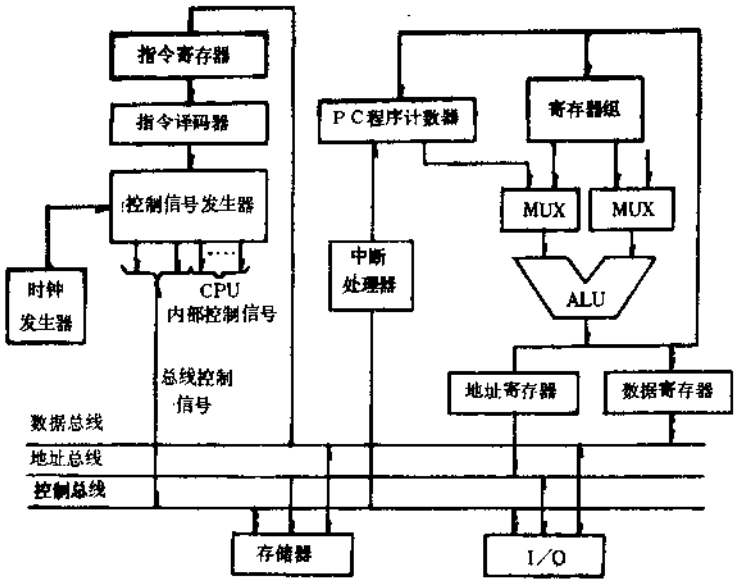


图1.14 CPU简框图

算术逻辑运算单元（ALU）：完成定点数运算。

寄存器组：保存参加运算的操作数和运算结果以及程序调用时的PC当前值等。在运算时，寄存器的内容通过多路转接器MUX送到ALU的两个输入端。有效地址的计算可在ALU中完成，也可另设一个ALU完成。

地址寄存器及数据寄存器：暂存访问存储器的地址及读写数据。

二、控制器的组成

程序计数器（PC）：保存当前正在执行的指令的地址，在一般情况下，下一条指令的地址由PC+1得到。在本条指令完成后将程序计数器的内容送往地址寄存器，从存储器中取出指令，送指令寄存器（IR）。当执行转移指令时，将计算出来的有效地址送地址寄存器取指，同时送PC保存。

指令寄存器（IR）：保存当前正在执行的指令，经过译码后，由控制信号发生器产生执行本条指令的所有控制信号，其中一部分送往CPU内部，完成规定操作，另一部分送往控制总线，控制MEM和I/O的操作。

至于控制信号发生器请阅读“硬布线逻辑与微程序控制”。

中断处理器：在机器运行时，当发生一些诸如出错等异常事件或外部设备请求处理时，要求CPU暂时中止当前正在执行的程序，转而进行异常处理或I/O处理，此项工作由中断处理器协助进行。

三、一条指令执行过程的总结

一条指令的执行步骤：

（1）取指：将当前要执行的指令地址从PC→MEM，取出指令，送IR。

（2）指令译码：产生实现本条指令的操作所需的控制信号。

（3）执行：完成本条指令的操作。除此以外还要确定下一条指令地址，当实现程序转移时，将转移地址送PC；不转移时PC+1送PC。同时要判断是否是停机指令，是则转向(4)，否则返回（1）。

（4）停机或动态停机：停机是使CPU暂停工作；动态停机，则让机器进入无休止的循环程序中（例如执行一条转移到本身的无条件转移指令），待有外来事件引起中断后才脱离动态停机状态。

上述过程可用图表示（图1.15）

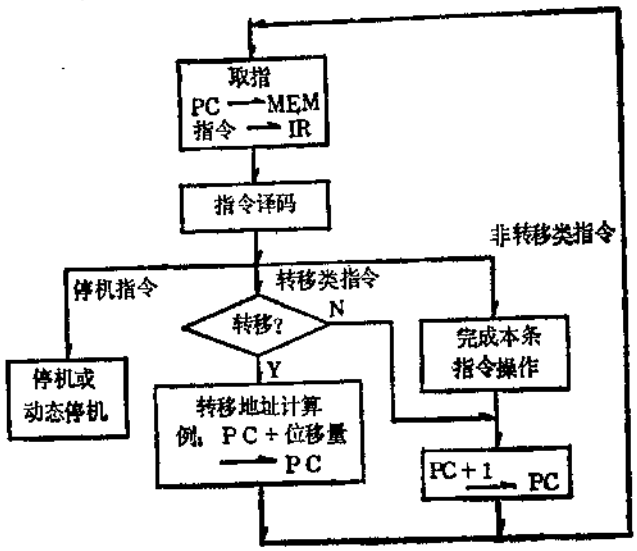


图1.15 指令执行过程

机器完成一条指令的动作所需的时间由3部分组成，其时间关系如下：

取指1	译码1	执行1	取指2	译码2	执行2	……

假如每一部分所需的时间相等，以t表示，那么执行一条指令所需的时间为3t。一般我们称t为机器周期，那么每条指令需要三个机器周期才能完成。实际上此三部分所需的时间是不相等的，取指的时间较译码长，因为它要访问一次存储器，而存储器的速度较之逻辑电路慢，另外不同的指令所需要的执行时间不尽相同，如将执行时间短的指令定为1个机器周期，那么有些指令需要若干个机器周期才能完成。从组成计算机的各个部件来看，在上述三个阶段它们并不是都处于忙碌状态，如在取指阶段，ALU是空闲的，假如能使机器各部分始终处于忙碌状态，其效率是最高的，因而需要考虑时间的合理分配以及某些操作的重叠进行问题。

1.3.4 CPU结构的改进

为了提高机器的计算与处理速度，除了采用高速的集成电路芯片、提高时钟频率以外，主要工作还在于改进硬件结构。

一、流水线结构

流水线结构的主要特点是变顺序串行执行指令为重叠执行。从图1.16中可见，当第一条指令在译码时，就到存储器中去取第二条指令；而在下个周期，三条指令均在操作，即第一条指令在执行，第二条指令在译码，第三条指令正从存储器读出。所以就一条指令来看，需要3t时间完成，但是机器每隔t时间就能完成一条指令，（相当于从工厂的流水线上输出一条指令，流水线的名称由此而来），速度可提高三倍。这是最理想的情况。事实上有很多情

况会引起流水线阻塞，例如当需要从存储器取数据时，就会与从存储器取指令的操作冲突，因此需要将取指操作推迟进行，又如某些指令的执行时间需要几个周期等等，这些都会使流水线的速度减慢。图1.17为第2条指令执行时间较长而引起流水线速度下降的情况。

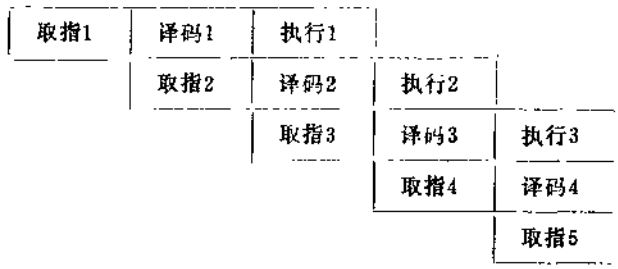


图1.16 指令的重叠执行

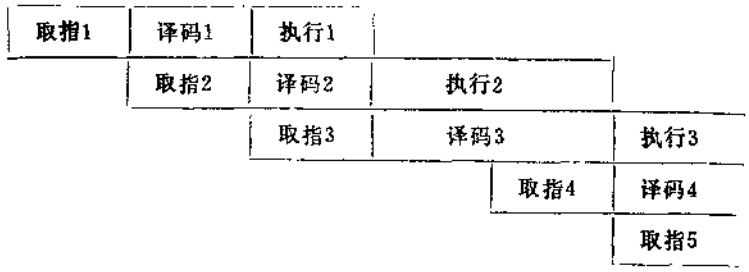


图1.17 流水线速度下降情况

在机器中还可设计多个层次的流水线。上面讲到的是指令级的流水线，假如运算速度慢，还可采用流水线的运算器，例如浮点加法运算可分成对阶、相加及结果规格化三级流水线。

采用流水线后不可避免产生数据相关问题，由于冯·诺曼机器的指导思想是顺序执行指令，而现在几条指令同时处理，因而有可能产生前面指令的运算结果还未送回寄存器或存储器，而后面的指令却要从同一寄存器或存储单元取数，这种现象称为数据相关。

流水线工作的正确性应由硬件保证，但是保持流水线畅通则与编写程序有关，尤其是编译程序要考虑优化问题。

有些计算机考虑从存储器取指令的时间较长，因而采用预取指令、或同时取出几条指令、或采用高速缓冲存储器等方法，这对缩短机器周期t是有效的。另外在对流水线分段时，每一段所需时间应尽量接近，如我们按取指、译码、执行（基本时间）分成三段时，机器周期要取这三段中的最长时间。

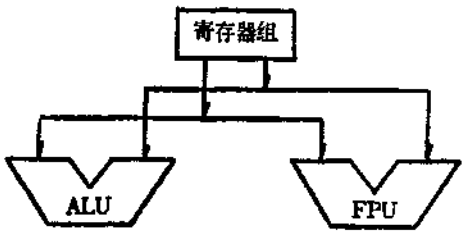


图1.18 具有多个功能部件的运算器

条件转移指令对流水线会产生不好的影响，因为转移是否成功要在执行阶段才知道，而一旦转移成功，预取的指令就要作废，由于不能及时提供指令而使机器处于等待状态。外部事件引起的中断也会产生同样的问题。

二、在CPU内部采用多个功能模块

有的计算机，在它的运算部件中设置两个或两个以上运算部件。图1.18设置定点运算部件和

浮点运算部件（ALU和FPU），从同一寄存器组取数，定点运算和浮点运算可以并行重叠进行。

三、在系统总线上增加浮点运算部件或协处理器选件

图1.13中的FPU接在系统总线上，当MEM送来的是浮点运算指令由FPU接收处理，FPU与CPU可并行工作。

四、与机器速度关系较大的电路尽可能集成在一个芯片内。芯片内电路、芯片间电路和插件板之间的电路延迟时间之比接近于1：3：10。

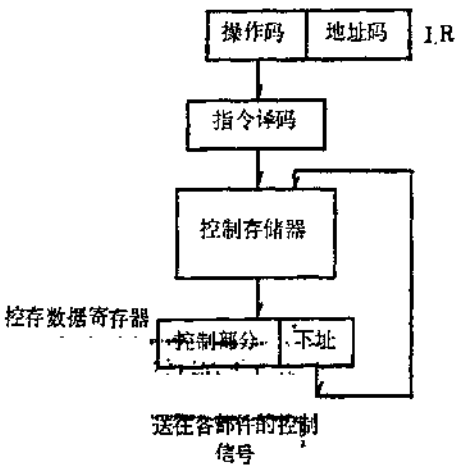
1.3.5 硬布线逻辑与微程序控制

产生控制信号有两种方法，一种是利用逻辑电路实现，它主要由门电路（PLA、PAL、门阵列）及触发器来实现控制，称为硬布线逻辑或组合逻辑。另一种称为微程序控制。

对控制信号发生器（图1.14）的要求是：根据每条指令的功能及其实现步骤，按一定的时序关系发出一定的控制信号（电位或脉冲）到操作部件。我们主要结合微程序控制来讨论。

一、微程序设计的概念

前面已经提到，一条指令的执行过程可以分成几段，每段完成一些操作。微程序设计即是将一条指令分成几条微指令，顺序执行这些微指令，就可以实现指令的功能。这些微指令的集合就叫做微程序。所以微程序的概念与程序设计中的子程序（或宏指令）的概念相当。微程序存放在控制存储器中，每条指令都有相应的一段微程序，所以执行一条指令实际上就成为执行一段微程序。有关的逻辑图如图1.19所示。



当指令送入IR后，在指令译码器中将操作码翻译成该条指令的微程序入口地址，从控存中取出这条微指令，保存在控存数据寄存器中。微指令由两部分组成，其中一部分为控制部分，发出完成本条微指令所需的控制信号，并有连接线与需要这些信号的电路相连接。另一部分为下址部分，指出下一条微指令的地址，称为微地址。假如微程序顺序执行，也可将当前的微程序计数器加1后形成下址。

二、水平型微指令和垂直型微指令

根据控制方式不同，可分为水平型微指令与垂直型微指令两种。

（1）水平型微指令

微指令控制部分的长度是根据控制全机所需的信号而定的，每个信号1位。如整机需要100个控制信号，则需要100位。所以水平型微指令字的长度比指令的长度大。图1.20为控制ALU完成加、减、逻辑加、逻辑乘四种运算的示意图。假如要完成将A、B寄存器的内容相加，并将结果送到C寄存器的操作，那么在微指令中要设置七个控制位，其中A、B、C、+四位的状态为“1”；－、∨、∧三位为“0”。

同样，在这条微指令中还有控制存储器及I/O操作的控制位，当需要这些部件操作时，相应位为“1”，否则为“0”。为缩短微程序长度，提高指令的执行速度，在执行微指令时，应尽量提高操作的并行度。

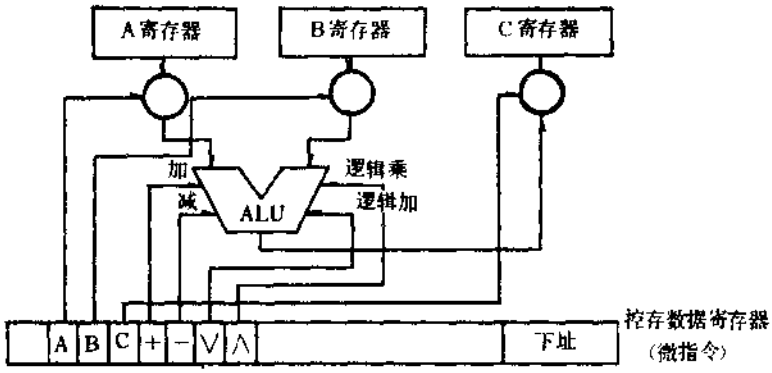


图1.20 水平型微指令控制示意图(举例)

(2)垂直型微指令

垂直型微指令的格式与指令相类似，一条微指令分成微操作码(μOP)和微地址码两部分。例如完成算术逻辑运算的微指令格式如下：

μOP	源寄存器1	源寄存器2	目标寄存器

表示将源寄存器1和源寄存器2的内容按μOP的规定运算，并将结果送往目标寄存器。注意这条微指令与指令不同，它仅完成一个操作，而指令还要完成取指等几个操作。

当微操作码指出要完成取数或其他微指令时，微地址码部分的定义就不同了。

由此可见，垂直型微指令与水平型微指令相比，其指令长度短，但程序长度长，速度慢。在实际应用时，是上述两种型式的综合。

三、微程序控制器

微程序控制的计算机，执行指令的控制逻辑是作为微程序存入控存的，由于一般计算机的指令系统是固定的，所以控制存储器一般采用固定存储器(只读存储器ROM)。

假如用可改写的存储器代替存固，那么就允许用户或程序员设计新的指令，只要编写相应的微程序送入控存就行了，这就叫做动态微程序设计。

微指令从控存取出后，存放在控存数据寄存器中，其目的是让取微指令与执行微指令重叠进行，实现流水线，所以控存数据寄存器一般就叫做流水线寄存器

微程序控制的速度一般比硬布线逻辑低，因为一条指令由若干条微指令实现，往往要多访问控制存储器。

1.4 计算机的存储体系

存储器主要用于存放计算机的程序和数据。存储器系统指的是存储器硬设备以及管理该存储器的软、硬设备。对存储器的基本要求是增大容量、提高速度、降低价格。单一的存储器硬件难以满足要求，所以就提出了多层次的存储体系结构。

1.4.1 存储体系的形成

一、CPU和主存储器的速度匹配问题

第一代冯·诺曼计算机的基本元器件是电子管，由于受电子管的寿命短、价格贵、体积

大、能耗大等因素的影响，因而决定了当时计算机的硬件功能比较简单，程序与数据都存放在同一存储器中，控制器由逻辑电路实现。在这一时期，运算、控制部件的速度与存储器相匹配，因此机器各部分之间的工作是协调的。图1.21（a）是当时的基本结构。应该指出，当时计算机解题的能力和速度都是比较低的。

当集成电路出现后，计算机进入第三代。由于其价格的下降以及可靠性的提高，计算机的硬件功能逐渐增强，CPU和存储器的速度都有明显提高，但CPU的电路速度提高得更快，到了七十年代，在合理的成本与足够的存储容量条件下，CPU的速度约比存储器高一个数量级。基于下述两个原因，由硬件实现的计算机指令系统不断完善和扩大：

（1）由于硬件成本下降，软件费用在计算机系统中所占的比例逐渐上升，而软件的处理速度明显低于硬件，因此产生功能转移。指令系统是软、硬件的主要交界面。为了便于编写程序，增加了不少复杂指令。

（2）一条复杂指令可以完成若干条简单指令的功能，这样一来可减少程序的长度，而且在程序运行时也可减少访问存储器的次数，以赢得计算机实际速度的提高。

为了尽量利用存储器的空间及时间，在计算机中采用了多种长度的指令字，增加了指令格式及寻址方式的变化，这样便使计算机控制器设计变得极为复杂。因此利用只读存储器（ROM）实现微程序控制的控制器，由于具有规整化、容易设计、容易修改和便于扩充等特点而获得迅速发展，除了速度要求极高的情况外，在大、中、小、微型计算机中得到广泛的应用。其逻辑框图见图1.21（b）。

在解决逻辑电路和存储器速度不平衡问题方面，高速缓冲存储器（Cache）起了很大作用。高速缓冲存储器位于CPU和存储器之间，它的容量比存储器小但存取速度高，其内容为存储器的部分拷贝。在程序运行过程中，当需要取指令或取数时，先检查Cache中是否有此内容，若有就从Cache中取出，若没有再从存储器取出。图1.21（c）为逻辑示意图。为提高计算机操作的并行度及简化设计，近年来将存放指令的Cache和存放数据的Cache分开，分别称为指令Cache（I Cache）和数据Cache（D Cache），见图1.21（d）。Cache的速度已接近于CPU的速度及微程序ROM的速度，因此近年来发展的RISC（精简指令系统计算机），将指令简化，基本上不用微程序控制，图1.21（e）为其示意图。

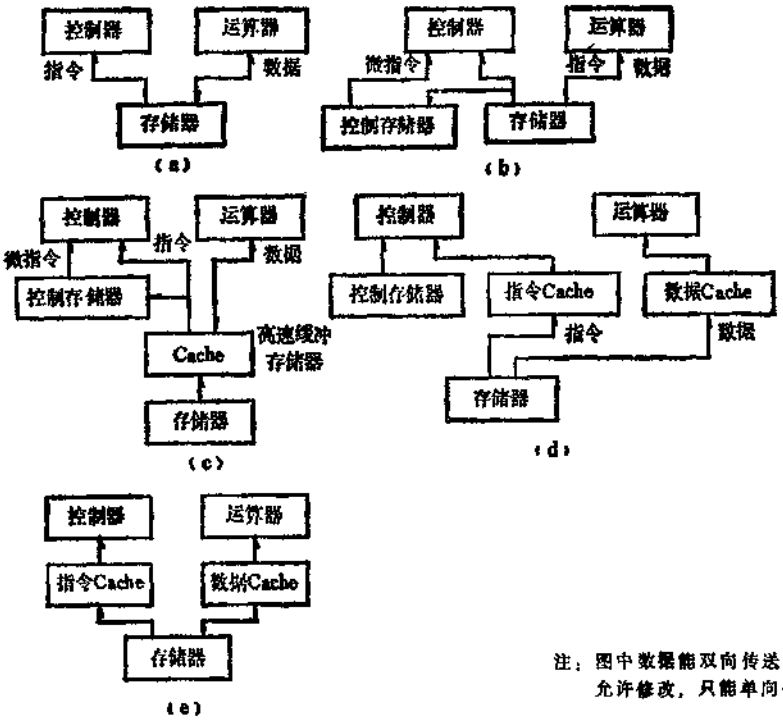
控制存储器虽称为存储器，但它不属于存储器体系，而是CPU的一部分。在CISC（复杂指令系统计算机）中，这部分电路在CPU芯片中所占的面积约占芯片可用的总面积的20%～40%。

当前在CPU芯片中设置了数量较多的寄存器，尤其在RISC中更是这样，其数量可达到100余个到500余个(Parimad计算机有528个寄存器)。大多数指令不必到存储器中去存取数据，给编译程序的优化创造了条件。

至此，我们谈到的存储器已不止一种。为便于讨论，在后面的各节中，我们把一般存放程序和数据的存储器称为主存储器（简称主存）。

二、扩充存储容量

为了程序的需要，主存容量已从几K字节发展到几M～几十M字节。但由于价格的限制还不能充分满足程序的要求，也就是说，系统程序、应用程序以及数据所需要的总存储空间一般超过主存容量，因此只能把当前要用到的或经常要用到的部分放在主存中，而把其它还未用到或不常用到的部分放在低速、大容量的辅存（辅助存储器）中作为后援，当要用到它们时，再把它们从辅存调往主存。这个调动工作是由操作系统中的存储分配管理模块来帮助



注：图中数据能双向传送，而指令不允许修改，只能单向传送。

图1.21 CPU结构的演变

程序员实现的。

多道程序的发展，促进了操作系统的发展。为了使程序员尽可能摆脱主、辅存间调动文件这一复杂工作，逐步形成了支持这些功能的"辅助硬件和软件"，这就是如何从系统结构上通过软硬件结合的方法把主存和辅存统一成整体，使得从整体来看，其速度接近于主存，但从容量来看却等于辅存，这种体系的不断发展，逐步形成了现在广泛使用的虚拟存储系统。对它，应用程序员可用机器指令的地址码对整个程序统一编址。目前新设计的计算机的地址码一般为32位，即使是微型机，最近也已发展到32位，其对应的容量为4000M字节。这当然比主存实际容量大得多，以致可以存得下整个程序。我们把这种指令地址码称为虚拟地址或逻辑地址，其对应的存储容量称为虚存容量或程序空间；而把实际主存的地址称为物理地址或实存地址，其对应的存储容量称为主存容量或实存容量。

但是一般具有辅存的存储系统并不一定是虚拟存储系统，是否是虚拟存储系统的本质区别是：

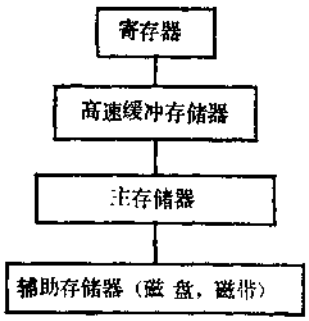


图1.22 多层次存储体系

（1）虚拟存储系统允许用户用比主存容量大得多的地址空间来访问主存，而非虚拟存储系统只允许用户至多用主存容量，而实际上只用比主存容量小的分配给他使用的那部分空间来访问主存。

（2）虚拟存储器每次访问主存时，都必须进行虚、实地址变换，而非虚拟存储系统则不必进行变换。

图1.22所示为多层次存储体系，上三层都是由半导体集成电路实现的，但是其速度与容量是不同的。图中从上到下，速度不断降低而容量不断增加。