第 1 章
计算机的组成

物有本末，事有终始，知其先后，则近道矣。一件事物的产生并非一蹴而就，无论是宏观的世界，还是微观的杯子。我们所能看到的，都是经历了漫长发展过程最终形成的成果，杯子是这样，计算机亦是如此。本章将介绍计算机演变过程及其证明，并从数学与物理两个方面证明它的可行性。由于内容非常广泛，本章不会深入某些具体的内容。因为计算机发展至今，它的知识体系已经非常庞大了，并非一书能解决。本章遵从书名，主要讲述计算机发展的主要过程，读者根据此过程可以找到自己的发展方向。

1.1　一颗计算机种子

1936 年，英国数学家艾伦·图灵提出了一种将人的计算行为抽象化的数学逻辑机器，这种机器在更抽象的层面上被视为一种计算机模型，也被称为**确定型图灵机**，如图 1.1 所示。

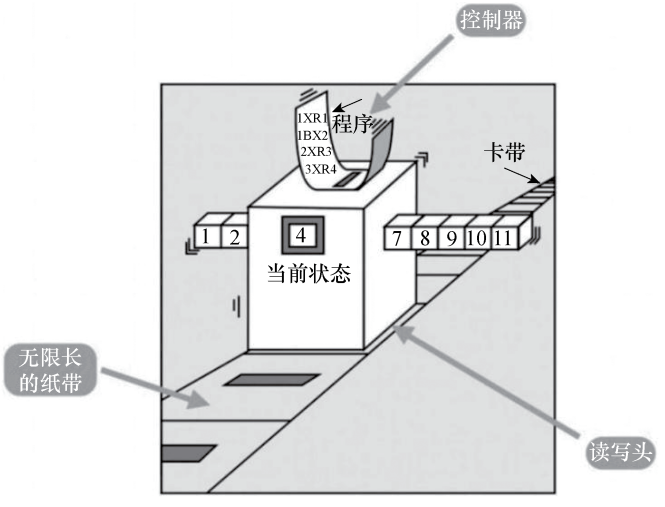

图 1.1　确定型图灵机示意图

图灵的基本思想是用机器模拟人用纸和笔进行数学运算的过程。图灵将计算过程看作两种简单动作的组合，即在纸上标记或擦除符号，以及把注意力从纸的一处移动到另一处。

在每个阶段，人依据当前关注的纸上某个位置的符号和当前思维的状态，决定下一步动作。

为了模拟人的运算过程，图灵构造了一台假想的机器，该机器由纸带、读写头、控制规则、寄存器四部分组成。

无限长的纸带被划分为一个接一个的小格子，每个格子上包含一个来自有限字母表的符号，并使用特殊符号（□）表示空白。纸带上的格子从左到右依次被编号为 0、1、2 等数字。纸带的右端可以无限伸展。

读写头可以在纸带上左右移动，它不仅能读出当前所指的格子上的符号，还能对符号进行修改。

控制规则根据当前机器所处的状态，以及当前读写头所指格子上的符号，确定读写头下一步的动作，并改变状态寄存器的值，使机器进入新状态。控制规则按照写入、移动、保持的顺序告知图灵机命令。其中：写入命令用于替换或擦除当前符号；移动命令则指挥读写头向左（L）、向右（R）移动，或不移动（N）；保持命令则用于维持当前状态或者切换到另一状态。

状态寄存器负责保存图灵机当前所处的状态。图灵机的所有可能状态的数目是有限的，并且其中有一个特殊的状态，称为停机状态（即无法通过一个程序正确地判断另一个程序是停机还是死循环，另一个程序也包含它自己）。

注意：

图灵机的每一部分都是有限的，但它有一个潜在的无限长的纸带，因此这种机器只是一种理想的设备。图灵认为这样的一台机器就能模拟人类所能进行的任何计算过程。由于图灵机是理想化的设备，1946 年美国陆军弹道研究实验室（BRL）公布的**电子数值积分计算机**（ENIAC）才被认为是世界上第一台通用计算机。ENIAC 是图灵完全的电子计算机，能够重新编程，以解决各种计算问题。

1.2 百 花 齐 放

掌握图灵机原理后，我们会发现其设计思想相对直观，尽管它属于高科技产品。本节将介绍两种结构：**冯 • 诺依曼架构**和**哈佛架构**。

1.2.1　冯・诺依曼架构

冯・诺依曼架构（Von Neumann architecture），也称为**冯・诺依曼模型**（Von Neumann model）或**普林斯顿架构**（Princeton architecture），是一种将程序指令存储器和数据存储器合并在一起的计算机设计概念，诞生于 1945 年，由**约翰・冯・诺依曼**（John Von Neumann）和其他人在 EDVAC 报告初稿中提出。它是一种实现通用图灵机的计算设备架构，同时隐约指导了将存储设备与中央处理器分开的概念。因此，基于此架构设计的计算机又被称为**存储程序计算机**。

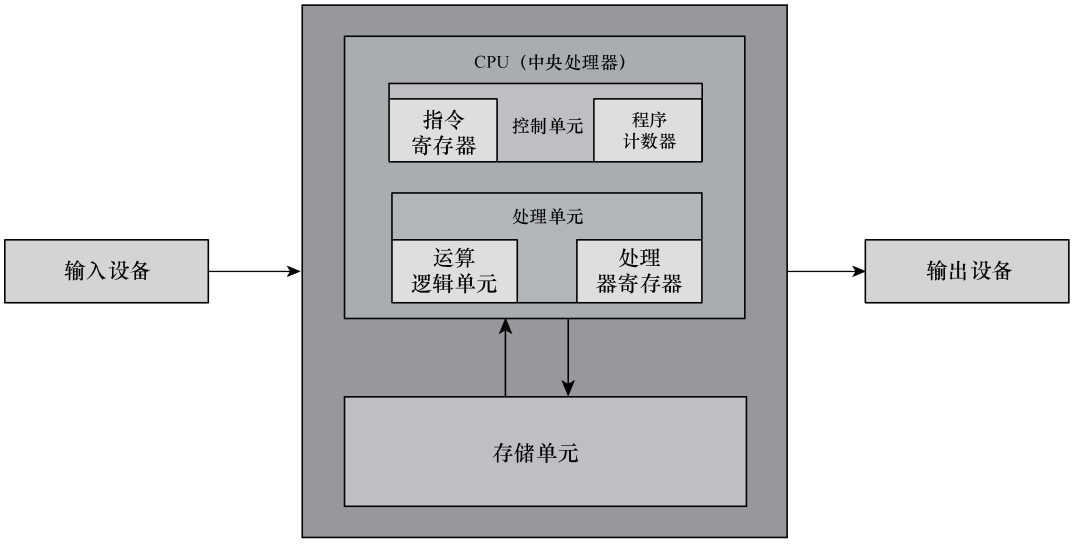

图 1.2　冯・诺伊曼架构的设计概念

图 1.2 展示了冯・诺依曼架构的设计理念，它由处理单元、控制单元、存储单元和输入/输出设备构成。

从图 1.2 中可以观察到，处理单元包含运算逻辑单元和处理器寄存器。其中，处理器寄存器用于存储逻辑运算的中间值。

控制单元包含指令寄存器和程序计数器，指令寄存器负责存储即将执行的指令，而程序计数器则记录下一条指令的位置，即行号。处理单元可以计算出当前指令的执行结果，该结果可能是运算值，也可能是下一条指令的行数（这个行数可以是具体值，也可以是相对于当前计数器值的偏移量）。

存储单元是控制单元用于存储执行结果的部分，同时支持控制器读取数据和指令。程序指令也存储于此。

输入/输出设备在程序和计算机都准备就绪后，允许用户进行操作。用户的任何操作都会转化为指令执行，这构成了输入。计算机的计算结果可以输出到屏幕上，也可以控制其他设备，这个过程称为输出。

扩展存储用于连接更大的外部存储设备，它通常被视为输入/输出设备的一部分。

冯·诺依曼架构这一术语现已扩展至涵盖所有存储程序计算机，在这些计算机中，指令的获取和数据操作无法同时进行（因为它们使用同一条公共总线）。这种现象被称为冯·诺依曼瓶颈，通常会对系统的性能产生限制。该架构的核心是对图灵机概念的进一步具体化，其中一个重要变革是引入了电子元件，而图灵机的时代则主要依赖于机械部件。

1.2.2 哈佛架构

哈佛架构（Harvard architecture）是一种计算机架构，它采用将程序指令存储和数据存储分开的存储器结构，即所谓的分离缓存（split cache）。这种架构最初应用于马克一号（Mark I），该计算机被认为是世界上第一台大型通用电子数字计算机。马克一号由 IBM 的霍华德·艾肯设计，并于 1944 年 8 月 7 日被安装在哈佛大学，因此这种架构被称为哈佛架构，如图 1.3 所示。

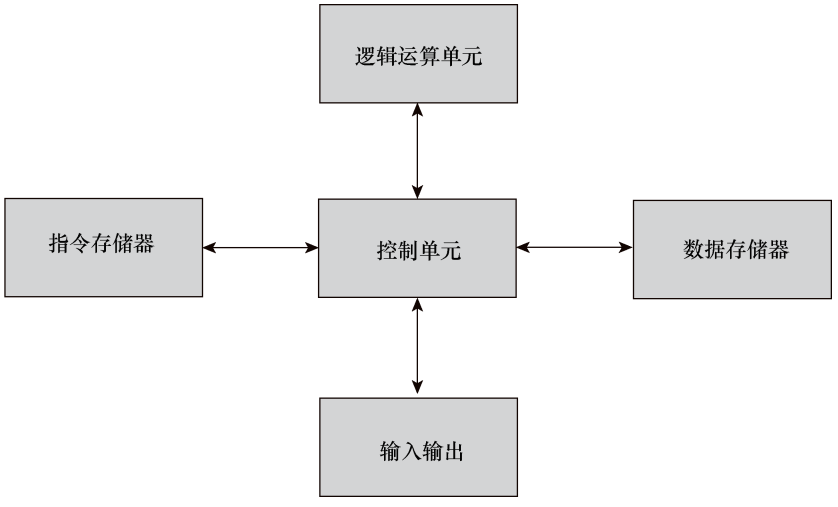

图 1.3　哈佛架构的设计概念

哈佛架构经常被用来与冯·诺依曼架构进行比较。在冯·诺依曼架构中，程序指令和数据共享相同的内存和传输路径。哈佛架构则将存储单元分为指令存储器与数据存储器两部分，这样做有效地解决了冯·诺依曼架构的瓶颈问题。具体对比可参见图 1.4。

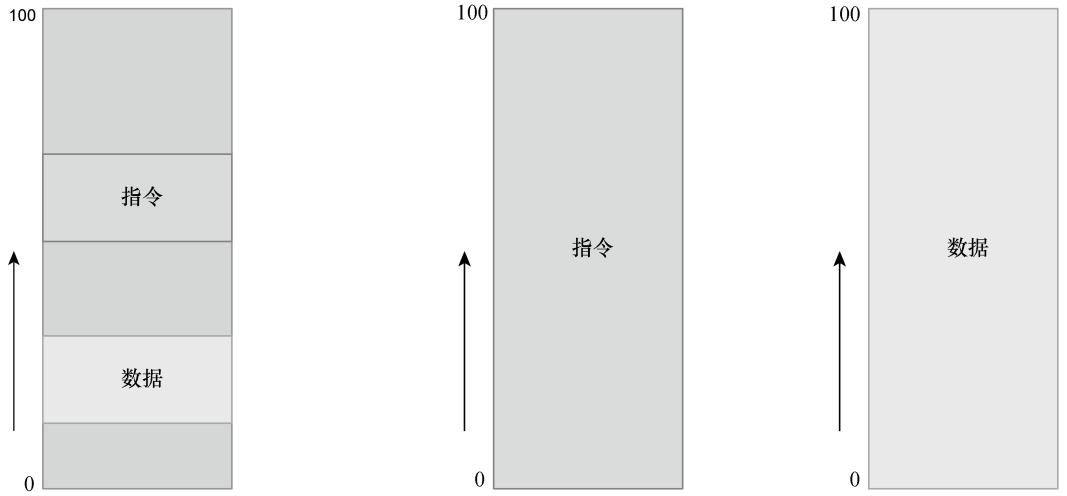

图 1.4　地址编码比较

哈佛架构允许指令地址和数据地址相同，这是因为它拥有独立的指令和数据总线。相较之下，冯·诺依曼架构采用单一总线来同时处理指令和数据，这种设计限制了指令和数据的并行处理，进而影响性能，尽管它简化了设计并降低了成本。目前，许多计算机系统仍然采用冯·诺依曼架构。值得注意的是，第一个在 Mark I 计算机上运行的程序是在冯·诺依曼的领导下，于 1944 年 3 月 29 日开发的。在后续的讲解中，我们将重点介绍冯·诺依曼架构，并对本节提到的术语，如地址，进行详细解释。

1.3　e 进制

前文已提及程序、地址及存储的数据均需要经过编码才能有效存储。本节将详细阐述与编码相关的知识。

1.3.1　进制

进制是一种记数方式，全称为进位制。它是一种记数法，通过使用有限种数学符号来表示所有的数值。在一种进制中，可以使用的数学符号的数目称为该进制的基数或底数。如果一个进制的基数为 n，则可以称其为 n 进制。进制的特征在于每计满 n 个数时，就进一位。

1.3.2 二进制

现代的**二进制**（**binary**）记数系统由戈特弗里德·威廉·莱布尼茨于 1679 年设计，这一系统的详细描述首次出现在他于 1703 年发表的文章《论只使用符号 0 和 1 的二进制算术，兼论其用途及它赋予伏羲所使用的古老图形的意义》[①]中。与二进制数相似的概念在早期文化中也有所体现，包括古埃及、中国、古印度以及太平洋岛屿的土著文明。其中，古代中国的《易经》激起了莱布尼茨的深刻联想。

从对进制的描述中可以推理出，二进制是一种以 2 为基数的计数系统，其特点是逢二进一，因此该系统只有两个数字 0 和 1。例如，当 1+1 时，结果是 10。以下是莱布尼茨的二进制记数系统的一个例子。

- ☑ 0 0 0 1 数值为 2^0。
- ☑ 0 0 1 0 数值为 2^1。
- ☑ 0 1 0 0 数值为 2^2。
- ☑ 1 0 0 0 数值为 2^3。

莱布尼茨认为《易经》中的卦象与二进制算术有着紧密的联系。在解读《易经》中的卦象之后，他坚信这些卦象可以作为二进制算术的证据。他饶有兴致地将《易经》的卦象与从 0 到 111111 的二进制数字进行一一对应，并认为这种对应体现了中国在其重大成就中所展现的、他所崇尚的数学哲学。

1.3.3 三进制

若从常规进制来看，三进制与二进制的基本原理相似，只是多了一个符号 2。因此，这里介绍的是**平衡三进制（balanced ternary）**。平衡三进制是一种非标准的计数系统，其基数为 3，使用的符号为−1、0 和 1。与标准的三进制系统不同，平衡三进制并不是在 1 之后继续延伸数字，而是在 0 之前引入了−1，如表 1.1 所示。

平衡三进制能够表示所有整数，这是因为它引入了−1，从而在表示负数时无须使用额外的负号。这一特性使得平衡三进制在加法、减法和乘法运算的效率上高于二进制。美国著名计算机科学家唐纳德·高德纳在《计算机程序设计艺术》一书中赞叹道：“也许最美的进制是平衡三进制。”

在计算机的早期发展历程中，苏联研制了一些基于平衡三进制的实验性计算机，其中最著名的是尼古拉·布鲁金索夫和谢尔盖·索博列夫共同建造的 Сетунь。与传统的二进制

① 原文完成于 1703 年，并于 1705 年首次发表在巴黎出版的《1703 年皇家科学院年鉴》（Histoire de l’Académie Royale des Sciences , Année 1703 , Paris ,1705 , pp :85—89）

系统相比，平衡三进制的实验性设计在计算科学领域展现了明显的优势。具体来说，其正负对称的特性大大提高了多位乘法运算中的进位效率。此外，平衡三进制在舍入操作中减少了进位的频率。在该系统中，单位数的乘法运算无须进行进位，加法运算最多只产生两个对称进位，而非三个，这简化了计算过程。

表 1.1　平衡三进制

平衡三进制	逻 辑 状 态	标准三进制
1	True	2
0	Unknown	1
T（−1）	False	0

1.3.4　e 进制

数字 *e* 也称为欧拉数（Euler's number），是一个数学常数，其值约为 2.71828。它可以通过多种方式表征，如图 1.5 和图 1.6 所示。*e* 是自然对数的底数。此外，*e* 还可以定义为表达式（1 + 1/*n*)^*n* 的极限，当 *n* 趋向于无穷大时。这个表达式在数学研究中具有重要意义，并且与复利计算的概念密切相关。

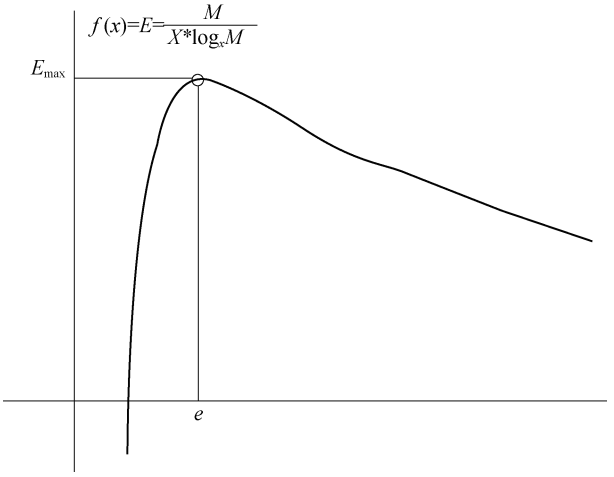

图 1.5　函数图像

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

图 1.6　欧拉数定义公式

当 n 为 1 000 000 时，计算结果约为 2.71828。那么，为何要提出以 e 为基的进制呢？因为欧拉数（即自然对数的底数 e）在底数效率模型中被认为是最高效的进制。以下是一个例子，用以证明其效率：假设我们有一个数 100，我们该如何通过这 100 位数字组合出最大的值？例如，以 2 为底数，计算得到 100/2=50，这意味着有 50 个 2，而组合这些数字意味着将它们相乘，即 2 的 50 次方（2^50）。通过这个例子，我们可以比较不同进制系统的效率差异，具体见表 1.2。

表 1.2　数据对比

进制（底）	每一位（表示存在可能）	位数（约）	总数（约）
2	2	50	2^50≈1 125 899 906 842 624
2.71828	2.71828	36.7879	2.71828^36.7879≈9 479 189 905 491 517
3	3	33	3^33≈5 559 060 566 555 523
8	8	12.5	8^12.5≈194 368 031 998
10	10	10	10^10≈10 000 000 000
16	16	6.25	16^6.25≈33 554 432
32	32	3.125	32^3.125≈50 535.164

根据上一组数据，我们观察到进制底数与 e 的距离越远，可组合的数值就越小，而当底数恰好为 e 时，可组合的数值达到最大。由于 e 进制的组合数较多，因此它能表达的数据量也更大。在成本相同（均为 100）的情况下，不同进制系统能表达的数据量不同，这正是效率差异的体现。因此，我们可以得出以下结论：e 进制是最有效率的进制，其次是三进制，因为它的底数最接近 e，然后是二进制。

1.3.5　其他进制

在日常生活中，我们最常用的是十进制系统，即使用数字 0、1、2…9。接下来，我们将介绍如何将十进制数转换为二进制数。

将十进制数转换为二进制数的方法是：将十进制数不断除以 2，并记录每次除法操作的余数。如果余数为 0，则记为 0；如果余数为 1，则记为 1。然后将这些余数顺序反转，得到的结果就是对应的二进制数。例如，将十进制数 20 转换为二进制的过程如下：

- ☑ 20 ÷ 2 = 10 余 0。
- ☑ 10 ÷ 2 = 5 余 0。
- ☑ 5 ÷2 = 2 余 1。
- ☑ 2 ÷ 2 = 1 余 0。
- ☑ 1 ÷ 2 = 0 余 1。

二进制结果为 10100，这是通过将计算得到的余数 00101 倒序排列而得到的。

将二进制数转换为十进制的方法是：将每个二进制位乘以 2 的幂次，幂次从右边的位开始以 0 计，并随着位数的增加而递增。然后将所有乘积相加，得到十进制结果。例如，将二进制数 10100 转换为十进制的过程如下：

☑ 从右第一位开始 2^0 * 0 = 0。

☑ 第二位，2^1 * 0 = 0

☑ 第三位，2^2 * 1 = 4

☑ 第四位，2^3 * 0 = 0

☑ 第五位，2^4 * 1 = 16

十进制结果为 20，即将这些乘积相加得到的总和。

十六进制是计算机科学中常用的进制系统，它包含数字 0、1、2…9 以及字母 A、B、C、D、E、F。这里将介绍十六进制与二进制之间的转换方法。

十六进制转二进制：这个过程与十进制转二进制的原理相似。

十六进制转十进制：这个过程与十进制转二进制的原理相同，不同之处在于这里除数是 16 而不是 2。

因此，可以得出一个规律：将大进制数转换为小进制数通常使用除法，而将小进制数转换为大进制数则通常使用乘法。相关的进制转换对照表如表 1.3 所示。

表 1.3　进制对照表

十进制	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
十六进制	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
二进制	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

我们已经介绍了计算机常用的进制。可能有读者会好奇，既然三进制在理论上比二进制更高效，那么为什么二进制仍然是计算机的主流实现呢？在历史的长河中，苏联曾尝试研发三进制计算机，美国也对此进行过研究，但这一尝试最终未能普及，这是由多种因素造成的。特别是由于历史的发展路径和技术选择，二进制在计算机科学中确立了其主导地位。当年，二进制的普及得益于半导体的广泛应用。尽管后来出现了支持三进制的新技术，但由于二进制设备已经广泛应用，三进制设备难以与之竞争。至今，如果突然转向三进制，

将会对现有的通信协议、操作系统、应用程序、存储设备以及数据产生颠覆性的影响，因为这一切都是基于二进制设计的。然而，历史总是充满不确定性，未来某一天可能会因为某些原因而转向三进制。既然目前二进制是主要的使用的进制，那么接下来的四节将深入剖析二进制设备的物理实现原理。

1.4 逻辑门与运算单元

本节将深入探讨基本的逻辑电路，并通过这些电路逐步推导出完整的运算单元。需要强调的是，虽然本节所介绍的电路理论上是可行的，但要实现它们，读者需要具备一定的电路基础知识。在某些电路设计中，可能还需要加入电阻等电子元器件来确保电路的正常运行。因此，建议读者在理解这些逻辑电路的基础上，进一步掌握相关的电路知识和技术，以便将这些理论应用于实际电路设计中。

1.4.1 NMOS 与 PMOS

电路中包含两个基本元件：NMOS（N-channel metal-oxide-semiconductor）和 PMOS（P-channel metal-oxide-semiconductor），分别如图 1.7 和图 1.8 所示。后续的逻辑门都是基于这两种元件构建的。其中，NMOS 晶体管在输入高电压时会导通，而 PMOS 晶体管则在输入低电压时导通。在二进制系统中，高电压对应于数字 1，低电压对应于数字 0。也就是说，有电流通过时表示高电压，代表 1；无电流通过时表示低电压，代表 0。在后续的讨论中，我们将使用 1 和 0 来表示这两种状态。

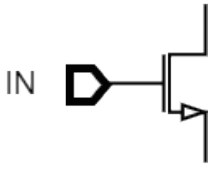

图 1.7 NMOS 晶体管示意图

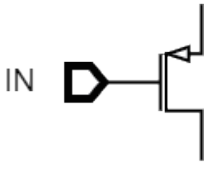

图 1.8 PMOS 晶体管示意图

本文不深入探讨 NMOS 和 PMOS 的工作原理，对此感兴趣的读者可自行查阅相关资料。读者只需知道 NMOS 和 PMOS 是构成整个逻辑电路的基础元件即可。

1.4.2 非门（NOT）

非门，又称为反相器，它能够接收 0 或 1 作为输入，并输出相反的值。如图 1.9 所示，

非门是由一个 PMOS 晶体管和一个 NMOS 晶体管组成的。当输入为 1 时，PMOS 晶体管不导通，而 NMOS 导通，因此输出端 Out 会连接到 NMOS 晶体管的低电压端，从而输出 0。相反，当输入为 0 时，PMOS 晶体管导通，而 NMOS 晶体管不导通，输出端 Out 则会连接到 PMOS 晶体管的高电压端，因此输出为 1。

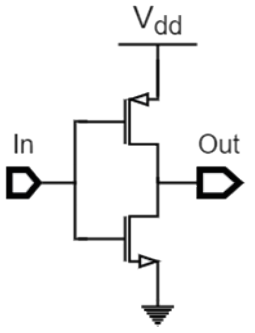

图 1.9　非门电路示意图

1.4.3　与门（AND）

与门具有两个输入端和一个输出端。从图 1.10 中可以观察到，该与门电路是由两个 NMOS 组成。只有当 In1 和 In2 同时输入高电平（即逻辑 1）时，输出端 Out 才会输出高电平（逻辑 1）；如果 In1 和 In2 不同时为高电平，则输出端 Out 将输出低电平（逻辑 0），从而实现二进制逻辑与操作。

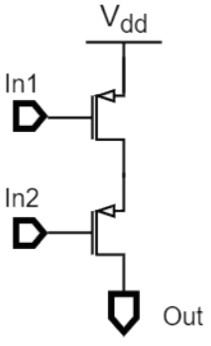

图 1.10　与门电路示意图

1.4.4　与非门（NAND）

与非门具有两个输入端和一个输出端。如图 1.11 所示，该门电路是由一个与门和一个非门组合而成的，其中非门的输入端接收与门的输出信号以产生最终结果。这个电路的一个潜在问题是时延。从该图中可以看出，与门和非门的操作似乎是同时进行的，但实际上，当与门的输出刚刚传递到非门时，非门就已经开始将结果输出到 Out 端，这就产生了时延问题。为了缓解这一时延问题，通常需要在非门之前加入其他元件，以确保非门在接收到与门的确切结果后才进行输出。在此，为了简化电路分析的复杂性，我们不对此进行进一步的扩展讨论。对此感兴趣的读者可以自行深入研究相关内容。

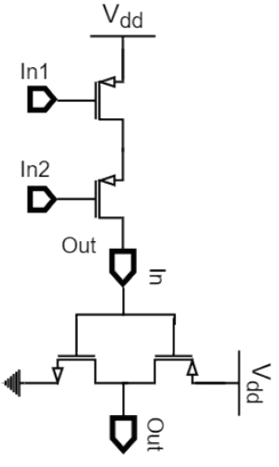

图 1.11　与非门电路示意图

1.4.5　或门（OR）

或门具有两个输入端和一个输出端。如图 1.12 所示，该或门电路由两个 NMOS 晶体管并联连接组成。只要 In1 或 In2 中至少有一个输入为高电平（逻辑 1），电路就会导通，使得输出端 Out 为高电平（逻辑 1）；只有当所有输入端 In1 和 In2 都为低电平（逻辑 0）时，输出端 Out 才会输出低电平（逻辑 0）。这实现了二进

制逻辑或操作。

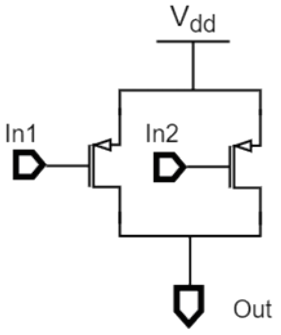

图 1.12　或门电路示意图

1.4.6　解复用器（de-multiplexer，DEMUX）

解复用器具有两个输入和两个输出。如图 1.13 所示，该电路由一个非门和两个与门组成，并通过 sel 信号来控制哪个与门的输出有效。这种电路通常用于选择功能，例如，两个与门分别对应加法电路和减法电路，可以通过 sel 信号来选择使用哪一个电路，并将 in 信号作为所选电路的输入。

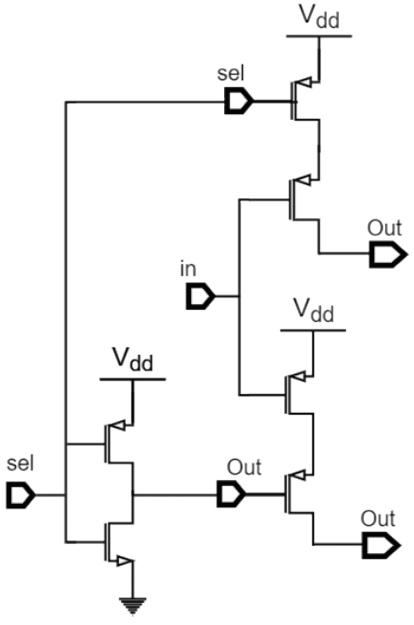

图 1.13　解复用器

由于该电路较为复杂，因此需要使用仿真语言进行描述。后续的电路将更加复杂，主要采用仿真形式展示，以便于读者理解。

使用仿真语言表示解复用器。

```
//存在两个输入，分别是数据 in 与选择信号 sel
IN in, sel;
//使用 sel 将 in 输出到相应的端点，这里有两个输出 a 和 b
OUT a, b;

//先对 sel 取反，得到 n1
Not(in = sel, out = n1);
//如果 in 与 n1 的逻辑与结果为 1，则输出给 a
And(a = in, b = n1, out = a);
//如果 in 与 sel 的逻辑与结果为 1，则输出给 b
And(a = in, b = sel, out = b);
//这两个与门是并行执行的，它们之间没有依赖关系，因此执行顺序不影响结果
```

1.4.7　复用器（multiplexer，MUX）

复用器具有三个输入和一个输出。如图 1.14 所示，该电路由一个非门、两个与门和一个或门组成。它通过 sel 信号从输入 a 和 b 中选择一个结果进行输出：当 sel 为 1 时，选择 a 作为输出；当 sel 为 0 时，选择 b 作为输出。复用器的反向操作是解复用器。

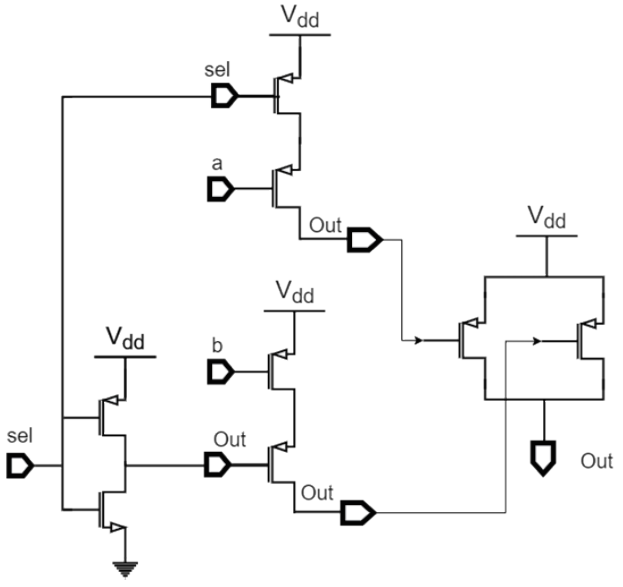

图 1.14　复用器

使用仿真语言表示复用器。

```
IN a, b, sel;
OUT out;

Not(in = sel, out = notSel);
And(a = b, b = sel, out = o1);
And(a = a, b = notSel, out = o2);
//通过或门从 o1 与 o2 中选择一个值作为输出
Or(a = o1, b = o2, out = out);
```

1.4.8 异或门（XOR）

异或门有两个输入和一个输出。通过图 1.15 可知，此门由两个非门、两个与门以及一个或门组成。其功能是实现两个二进制数的无进位加法，即当两个输入不同时，输出为 1；当两个输入相同时，输出为 0。

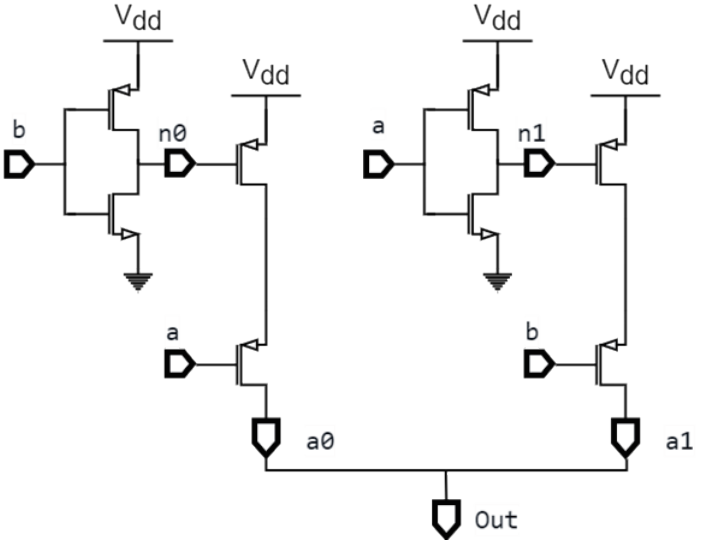

图 1.15　异或门

使用仿真语言表示异或门。

```
IN a, b;
OUT out;

Not(in = b, out = n0);
And(a = a, b = n0, out = a0);
Not(in = a, out = n1);
And(a = n1, b = b, out = a1);
Or(a = a0, b = a1, out = out);
```

1.4.9　多位组合电路

本小节将展示如何使用上述逻辑门来构建支持多位操作的组合电路。由于这些电路较为复杂，我们将提供相应的仿真语言代码。为了更好地理解逻辑计算，建议读者自行绘制电路图。之前的电路都是基于布尔逻辑运算来设计逻辑门的，感兴趣的读者可以尝试使用布尔代数来简化这些电路。

使用仿真语言表示**十六位与门（And16）**。

```
IN a[16], b[16];
OUT out[16];

And(a = a[0], b = b[0], out = out[0]);
And(a = a[1], b = b[1], out = out[1]);
And(a = a[2], b = b[2], out = out[2]);
And(a = a[3], b = b[3], out = out[3]);
And(a = a[4], b = b[4], out = out[4]);
And(a = a[5], b = b[5], out = out[5]);
And(a = a[6], b = b[6], out = out[6]);
And(a = a[7], b = b[7], out = out[7]);
And(a = a[8], b = b[8], out = out[8]);
And(a = a[9], b = b[9], out = out[9]);
And(a = a[10], b = b[10], out = out[10]);
And(a = a[11], b = b[11], out = out[11]);
And(a = a[12], b = b[12], out = out[12]);
And(a = a[13], b = b[13], out = out[13]);
And(a = a[14], b = b[14], out = out[14]);
And(a = a[15], b = b[15], out = out[15]);
```

使用仿真语言表示**十六位复用器（Mux16）**。

```
IN a[16], b[16], sel;
OUT out[16];

Mux(a = a[0], b = b[0], sel = sel, out = out[0]);
Mux(a = a[1], b = b[1], sel = sel, out = out[1]);
Mux(a = a[2], b = b[2], sel = sel, out = out[2]);
Mux(a = a[3], b = b[3], sel = sel, out = out[3]);
Mux(a = a[4], b = b[4], sel = sel, out = out[4]);
Mux(a = a[5], b = b[5], sel = sel, out = out[5]);
Mux(a = a[6], b = b[6], sel = sel, out = out[6]);
Mux(a = a[7], b = b[7], sel = sel, out = out[7]);
Mux(a = a[8], b = b[8], sel = sel, out = out[8]);
Mux(a = a[9], b = b[9], sel = sel, out = out[9]);
Mux(a = a[10], b = b[10], sel = sel, out = out[10]);
Mux(a = a[11], b = b[11], sel = sel, out = out[11]);
Mux(a = a[12], b = b[12], sel = sel, out = out[12]);
Mux(a = a[13], b = b[13], sel = sel, out = out[13]);
```

```
Mux(a = a[14], b = b[14], sel = sel, out = out[14]);
Mux(a = a[15], b = b[15], sel = sel, out = out[15]);
```

使用仿真语言表示**十六位四路复用器（Mux4Way16）**。

```
IN a[16], b[16], c[16], d[16], sel[2];
OUT out[16];

Mux16(a = a, b = b, sel = sel[0], out = o1);
Mux16(a = c, b = d, sel = sel[0], out = o2);
Mux16(a = o1, b = o2, sel = sel[1], out = out);
```

使用仿真语言表示**十六位八路复用器（Mux8Way16）**。

```
IN a[16], b[16], c[16], d[16],
   e[16], f[16], g[16], h[16],
   sel[3];
OUT out[16];

Mux4Way16(a = a, b = b, c = c, d = d, sel = sel[0..1], out = o1);
Mux4Way16(a = e, b = f, c = g, d = h, sel = sel[0..1], out = o2);
Mux16(a = o1, b = o2, sel = sel[2], out = out);
```

使用仿真语言表示**四路解复用器（DMux4Way）**。

```
IN in, sel[2];
OUT a, b, c, d;

DMux(in = in, sel = sel[1], a = o1, b = o2);
DMux(in = o1, sel = sel[0], a = a, b = b);
DMux(in = o2, sel = sel[0], a = c, b = d);
```

使用仿真语言表示**八路解复用器（DMux8Way）**。

```
IN in, sel[3];
OUT a, b, c, d, e, f, g, h;

DMux4Way(in = in, sel = sel[1..2], a = o1, b = o2, c = o3, d = o4);
DMux(in = o1, sel = sel[0], a = a, b = b);
DMux(in = o2, sel = sel[0], a = c, b = d);
DMux(in = o3, sel = sel[0], a = e, b = f);
DMux(in = o4, sel = sel[0], a = g, b = h);
```

使用仿真语言表示**十六位非门（Not16）**。

```
IN in[16];
OUT out[16];

Not(in = in[0], out = out[0]);
Not(in = in[1], out = out[1]);
Not(in = in[2], out = out[2]);
Not(in = in[3], out = out[3]);
Not(in = in[4], out = out[4]);
```

```
Not(in = in[5], out = out[5]);
Not(in = in[6], out = out[6]);
Not(in = in[7], out = out[7]);
Not(in = in[8], out = out[8]);
Not(in = in[9], out = out[9]);
Not(in = in[10], out = out[10]);
Not(in = in[11], out = out[11]);
Not(in = in[12], out = out[12]);
Not(in = in[13], out = out[13]);
Not(in = in[14], out = out[14]);
Not(in = in[15], out = out[15]);
```

使用仿真语言表示**十六位或门（Or16）**。

对输入的十六位数据进行或操作，最终得到一个结果值。

```
IN in[16];
OUT out;

Or(a = in[0], b = in[1], out = o1);
Or(a = o1, b = in[2], out = o2);
Or(a = o2, b = in[3], out = o3);
Or(a = o3, b = in[4], out = o4);
Or(a = o4, b = in[5], out = o5);
Or(a = o5, b = in[6], out = o6);
Or(a = o6, b = in[7], out = o7);
Or(a = o7, b = in[8], out = o8);
Or(a = o8, b = in[9], out = o9);
Or(a = o9, b = in[10], out = o10);
Or(a = o10, b = in[11], out = o11);
Or(a = o11, b = in[12], out = o12);
Or(a = o12, b = in[13], out = o13);
Or(a = o13, b = in[14], out = o14);
Or(a = o14, b = in[15], out = out);
```

1.4.10　半加器（Half Adder）

半加器有两个输入和两个输出。它通过计算输入的两个值来得到计算结果和进位标志，例如 1+1 的结果值为 0，进位标志为 1。

```
IN a, b;
OUT sum,
    carry;

Xor(a = a, b = b, out = sum);
And(a = a, b = b, out = carry);
```

1.4.11 全加器（Full Adder）

全加器由两个半加器和一个异或门组成，它比半加器多一个进位标志的输入。

```
IN a, b, c;
OUT sum,
    carry;

HalfAdder(a = a, b = b, sum = s1, carry = c1);
HalfAdder(a = s1, b = c, sum = sum, carry = c2);
Xor(a = c1, b = c2, out = carry);
```

1.4.12 十六位负数判断（IsNeg）

该电路判断最高位是否为 1，若为 1 则代表传入的数据小于 0，即负数。

```
IN in[16];
OUT out;

Or(a = in[15], b = false, out = out);
```

1.4.13 十六位加法器（Adder16）

计算两个十六位输入的和。

```
IN a[16], b[16];
OUT out[16];

HalfAdder(a = a[0], b = b[0], sum = out[0], carry = c0);
FullAdder(a = a[1], b = b[1], c = c0, sum = out[1], carry = c1);
FullAdder(a = a[2], b = b[2], c = c1, sum = out[2], carry = c2);
FullAdder(a = a[3], b = b[3], c = c2, sum = out[3], carry = c3);
FullAdder(a = a[4], b = b[4], c = c3, sum = out[4], carry = c4);
FullAdder(a = a[5], b = b[5], c = c4, sum = out[5], carry = c5);
FullAdder(a = a[6], b = b[6], c = c5, sum = out[6], carry = c6);
FullAdder(a = a[7], b = b[7], c = c6, sum = out[7], carry = c7);
FullAdder(a = a[8], b = b[8], c = c7, sum = out[8], carry = c8);
FullAdder(a = a[9], b = b[9], c = c8, sum = out[9], carry = c9);
FullAdder(a = a[10], b = b[10], c = c9, sum = out[10], carry = c10);
FullAdder(a = a[11], b = b[11], c = c10, sum = out[11], carry = c11);
FullAdder(a = a[12], b = b[12], c = c11, sum = out[12], carry = c12);
FullAdder(a = a[13], b = b[13], c = c12, sum = out[13], carry = c13);
FullAdder(a = a[14], b = b[14], c = c13, sum = out[14], carry = c14);
FullAdder(a = a[15], b = b[15], c = c14, sum = out[15], carry = c15);
```

1.4.14　算术逻辑单元（ALU）

此处通过现有的知识，构建一个简易的算术逻辑单元。

```
IN
    x[16], y[16],   //两个十六位输入
    zx,          //设置 x 为全 0 的标志位
    nx,          //取反 x 的输出标志位
    zy,          //设置 y 为全 0 的标志位
    ny,          //取反 y 的输出标志位
    f,           //计算方法选择标志位，1 表示加法（ x + y），0 表示与运算（ x & y）
    no;          //输出取反标志位
OUT
    out[16],     //十六位输出
    zr,          //输出结果是否为 0 的标志位
    ng;          //输出结果是否小于 0（即最高位为 1）的标志位

//当 zx 为 1 时，选择全 0（b:false）作为 x1 的输出
Mux16(a = x, b = false, sel = zx, out = x1);
//当 nx 为 1 时，取反 x1 得到结果 x2
Not16(in = x1, out = x2);
//通过 nx 从 x1 和 x2 中选择一个结果作为 x3
Mux16(a = x1, b = x2, sel = nx, out = x3);
//当 zy 为 1 时，选择全 0 作为输出到 y1
Mux16(a = y, b = false, sel = zy, out = y1);
//当 ny 为 1 时，取反 y1 得到结果 y2
Not16(in = y1, out = y2);
//通过 ny 从 y1 与 y2 中选择一个结果作为 y3
Mux16(a = y1, b = y2, sel = ny, out = y3);
//根据 f 标志位，选择进行加法或与运算
And16(a = x3, b = y3, out = f1);
Add16(a = x3, b = y3, out = f2);
//根据 f 标志位，选择 f1 和 f2 的输出到 o1
Mux16(a = f1, b = f2, sel = f, out = o1);
//当 no 为 1 时，取反 o1 得到 o2
Not16(in = o1, out = o2);
//通过 no 从 no1 与 no2 中选择一个结果作为输出到 out
Mux16(a = o1, b = o2, sel = no, out = out);
//计算输出结果是否为 0，结果输出到 zr1
Or16Way(in = out, out = zr1);
//取反 zr1 得到 zr
Not(in = zr1, out = zr);
//判断输出结果是否小于 0（即最高位为 1），结果输出到 ng
IsNeg(in = o3, out = ng);
```

至此，简单的 ALU 实现已经完成。此节的目的是向读者说明，ALU 是如何通过基础

元器件结合布尔运算来实现的。

1.5 D 触发器与存储单元

在逻辑单元的实现过程中，我们观察到一旦通电，系统就能输出预期的结果。如果需要调整输出值，就必须在程序中设置断点，修改输入参数后重新进行计算。如前文所述，程序是由一系列指令构成的，这些指令在执行时会产生中间计算结果。这些结果需要存储在特定的存储位置上，而存储单元正是负责管理这些位置的组件。本节将详细探讨存储单元的工作原理及其实现细节。

1.5.1 RS 触发器

在详细介绍 D 触发器之前，我们首先需要了解 RS 触发器，因为 D 触发器可以被视为 RS 触发器的升级版，它是在 RS 触发器的基础上增加了时钟脉冲控制。

如图 1.16 所示，RS 触发器是由两个或门（OR）和两个非门（NOT）互联构成的。它具有两个输入端和两个输出端，其中：第一个输入端 R 代表 Reset 的缩写，当 R 为 1 时，输出端 Q 将被置为 0，同时 Q’输出为 1；第二个输入端 S 代表 Set 的缩写，当 S 为 1 时，输出端 Q 将被置为 1，而 Q’输出为 0。在数据存储的需求下，通常只需要一个输出端口，即 Q，就足够了。从 Q 的角度来看，R 的作用是清零，而 S 的作用是置位。

为了更清楚地解释 RS 触发器的工作原理，我们将分别探讨或门和非门的作用，尽管在实际应用中，它们可以被组合成一个单独的门，即 NOR 门。读者如果仔细观察图 1.16 中的电路图，就会发现它看起来与普通逻辑电路没有太大区别，操作完成后似乎就结束了，那么它是如何实现锁存功能的呢？这是因为，或门和非门内部连接有独立的电源 Vdd，这意味着它们内部包含了一个小型的电路。这个小电路的通断状态由 R 和 S 的输入决定，进而影响 Q 和 Q’的输出。值得注意的是，即使 R 和 S 没有输入信号，内部电路仍然保持活跃状态。因此，我们可以确认，RS 触发器的锁存功能是由其内部电路实现的。那么，如何改变 Q 的输出状态（即实现 1 与 0 的切换）呢？这可以通过控制 R 和 S 的输入信号来实现。

图 1.17 清楚地展示了 RS 触发器的工作原理。观察该图可知，当 R 输入为 1 时，该信号作为**步骤一**的输入，经过 NOR1 门处理，使得**步骤二**的输出为 0。接着，这个 0 作为**步骤三**的输入，经过 NOR2 门后，**步骤五**的输出结果变为，这一输出结果实际上替代了 R 的初始输入。因此，无论 R 的输入是 0 还是 1，由于**步骤五**的输出反馈至**步骤一**作为输入，NOR1 门的输入端始终维持为 1。这样，通过**步骤六**持续提供 1 作为输入，可以保持输出结果的稳定，从而实现锁存功能。同时，由于**步骤二**的输出始终为 0，Q 端的输出也一直保

持为 0，实现了 Reset（复位）的功能。

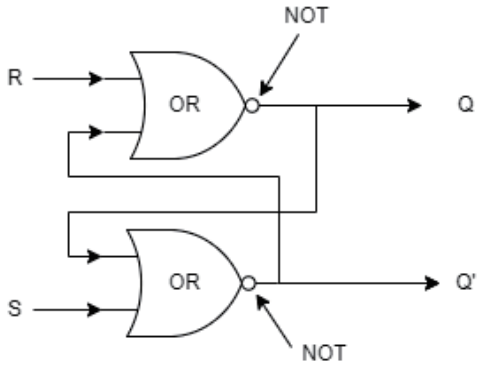

图 1.16　RS 触发器

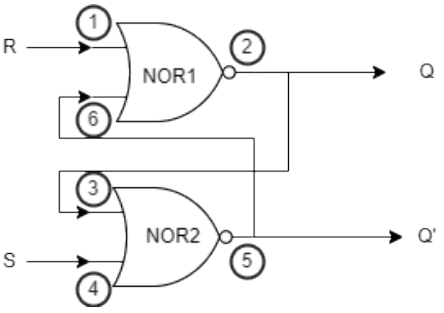

图 1.17　RS 触发器的工作原理

通过图 1.17，可以推导出 Set 原理。当**步骤四**的输入为 1 时，它经过 NOR2 门，导致**步骤五**的结果为 0。这一 0 值随后作为**步骤六**的输入，再经过 NOR1 门后，**步骤二**的输出结果为 1，即 Q 的输出结果为 1。同时，**步骤二**的输出作为**步骤三**的输入，这意味着即使当前的 S 输入为 0，由于**步骤三**已经替代了**步骤四**的原始输入 1，Set 的锁存效果也得以实现。

经过上述推理，发现 R 与 S 不能同时为 1，否则结果会变得不确定。相反，当 R 与 S 同时为 0 时，锁存效果得以实现。基于这些发现，可以进一步推导出真值表 1.4。

表 1.4　真值表

S	R	Q	Q'	动　作
0	0	0	0	保持
1	0	1	0	设置
0	1	0	1	重置
1	1	1	1	不允许

1.5.2　D 触发器

前文已提及，D 触发器作为 RS 触发器的增强版，它的核心改进在于将 RS 输入端合并为一个 D 输入端，并通过 CLK（时钟）信号来控制触发器的状态变化，这样做旨在防止噪声（干扰）导致的错误。如图 1.18 所示，D 触发器是在 RS 触发器的基础上增加了两个 AND 门和一个 NOT 门。当 D 输入为 1 时，表示需要执行 Set 操作，此时需要等待 CLK 信号变为 1，然后通过 AND2 门的作用，将 S 输入端置为 1，以实现 Set 功能。相反，当 D 输入

为 0 时，表示需要执行 Reset 操作，此时信号通过 NOT 门反相后变为 1，并且当 CLK 信号为 1 时，通过 AND1 门来实现 Reset 操作。

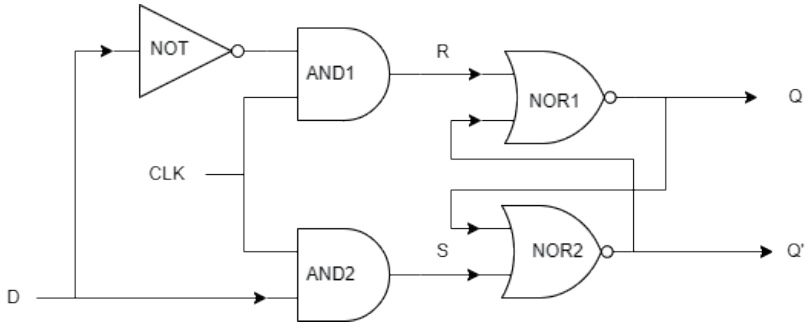

图 1.18　D 触发器

通过一个精巧的 NOT 门，可以灵活地选择是执行 Set 操作还是 Reset 操作。两个 AND 门则用于控制这两种操作（Set 和 Reset）的有效性。CLK 通常是由振荡电路提供的（我们将在后续章节中详细介绍），它为整个电路提供了时序控制。综上所述，我们得到了一个既可控又具备锁存功能的电路，这就是所谓的 D 型触发器（DFF）。

1.5.3　bit——一位存储器

通过整合 DFF（D 型触发器），我们可以构建一个 bit——一位寄存器。在这个寄存器中：只有当 store 信号为 1 时，data 才会被视为有效并被存储；若 store 信号为 0，则寄存器处于读取模式，此时存储的数据会被输出到 out 端，这样的数据单元被称作 bit。

```
IN data, store;
OUT out;

DFF(data = data, clk = store, out = out);
```

1.5.4　十六位寄存器

通过组合多个 bit，可以构建一个十六位的寄存器。该十六位寄存器使用一个统一的 store 信号，确保所有 bit 的步调一致，即同时进行数据的存储或读取操作。这样的设计奠定了基础存储为十六位的寄存器。

```
IN in[16], store;
OUT out[16];

Bit(in = in[0], store = store, out = out[0]);
```

```
    Bit(in = in[1], store = store, out = out[1]);
    Bit(in = in[2], store = store, out = out[2]);
    Bit(in = in[3], store = store, out = out[3]);
    Bit(in = in[4], store = store, out = out[4]);
    Bit(in = in[5], store = store, out = out[5]);
    Bit(in = in[6], store = store, out = out[6]);
    Bit(in = in[7], store = store, out = out[7]);
    Bit(in = in[8], store = store, out = out[8]);
    Bit(in = in[9], store = store, out = out[9]);
    Bit(in = in[10], store = store, out = out[10]);
    Bit(in = in[11], store = store, out = out[11]);
    Bit(in = in[12], store = store, out = out[12]);
    Bit(in = in[13], store = store, out = out[13]);
    Bit(in = in[14], store = store, out = out[14]);
    Bit(in = in[15], store = store, out = out[15]);
```

1.5.5　高位寄存器内存组合

通过对多个寄存器进行组合，我们可以得到更大存储容量的寄存器内存。

1. 128 bit 寄存器内存

使用解码器和复用器，我们可以构建一个由 8 个 16 位寄存器组成的 128 位寄存器内存，记作 RAM8。

```
    IN in[16], store, address[3];
    OUT out[16];

    DMux8Way(in = store, sel = address, a = storeA, b = storeB, c = storeC,
             d = storeD, e = storeE, f = storeF, g = storeG, h = storeH);

    Register(in = in, store = storeA, out = o1);
    Register(in = in, store = storeB, out = o2);
    Register(in = in, store = storeC, out = o3);
    Register(in = in, store = storeD, out = o4);
    Register(in = in, store = storeE, out = o5);
    Register(in = in, store = storeF, out = o6);
    Register(in = in, store = storeG, out = o7);
    Register(in = in, store = storeH, out = o8);

    Mux8Way16(a = o1, b = o2, c = o3, d = o4, e = o5, f = o6, g = o7, h = o8,
              sel = address, out = out);
```

2. 128 byte 寄存器内存

将 RAM8 与解码器和复用器进行组合，可以构建一个更大的内存结构。在这个过程中，注意到地址位的宽度发生了变化，这是因为随着内存容量的增加，地址位的宽度也需要相

应增加。解码器在选择内存单元时，使用了其中的三位地址线作为选择依据。将这个组合后的内存结构记作 RAM64。因为它由 64 个寄存器组成，每个寄存器是 16 位（即 2 字节），所以总的容量是 64×2=128 字节。

```
IN in[16], store, address[6];
OUT out[16];

DMux8Way(in = store, sel = address[3..5], a = storeA, b = storeB, c = storeC,
         d = storeD, e = storeE, f = storeF, g = storeG, h = storeH);
RAM8(in = in, store = storeA, address = address[0..2], out = o1);
RAM8(in = in, store = storeB, address = address[0..2], out = o2);
RAM8(in = in, store = storeC, address = address[0..2], out = o3);
RAM8(in = in, store = storeD, address = address[0..2], out = o4);
RAM8(in = in, store = storeE, address = address[0..2], out = o5);
RAM8(in = in, store = storeF, address = address[0..2], out = o6);
RAM8(in = in, store = storeG, address = address[0..2], out = o7);
RAM8(in = in, store = storeH, address = address[0..2], out = o8);
Mux8Way16(a = o1, b = o2, c = o3, d = o4, e = o5, f = o6, g = o7, h = o8,
          sel = address[3..5], out = out);
```

3. 128 KB 寄存器内存

若按照上述方式通过扩展三位地址来增加容量，中间将会经历多次迭代。为简化流程，此处将一步到位。RAM32K 代表一个包含 32×1024 个寄存器的内存。若存在两个这样的内存，则总容量为 64×1024 个寄存器。考虑到每个寄存器是 16 位，即 2 字节，因此总字节数为(16/8)×64×1024=131072 字节。尽管理论上 16 位地址只能访问 65535 个字节（因为 8 位表示一个字节，16 位表示两个字节的地址范围），但由于每个基础寄存器是 16 位，实际可访问的数据量增加了一倍，因此能够有效利用这 131072 字节的内存空间。这样的设计既确保了内存的高效利用，也提高了访问的便捷性。

```
IN in[16], store, address[16];
OUT out[16];

DMux16(in = store, sel = address[15], a = storeA, b = storeB);
RAM32K(in = in, store = storeA, address = address[0..14], out = o1);
RAM32K(in = in, store = storeB, address = address[0..14], out = o2);
Mux16(a = o1, b = o2, sel = address[15], out = out);
```

1.5.6 小结

至此，已成功构建了一块由 16 位寄存器组成的内存。这些寄存器不仅可以作为内存的一部分，还可以单独使用，提供直接的 16 位存储访问。为便于称呼，我们将这组寄存器命名为 ax（后续章节将详细介绍）。为了降低电路的复杂度，本节引入了一系列新的元件符

号。这些符号均基于最初的逻辑电路组合而成，并通过逻辑推理得出。此外，本节还深入阐述了锁存器的工作原理，并利用复用器与解复用器技术，巧妙地构造了一块高效的存储单元。

1.6　振荡器与计时器

在 1.5 节开头，我们提到电路只有导通与截止两种状态。一旦电路通电，电流经过各个组件的结果就已经确定。如果想要改变运算操作，就需要先断电再重新启动，同时修改传入的数据。虽然可以通过计算完成后主动触发重启操作来运行下一条指令，但这仅仅是对于运算而言，这样的操作速度相对较慢，尤其是在涉及访存等操作时。因此，需要一个协调器（即振荡器）来统一元器件间的协调工作。例如，当写入内存时，不必等待内存写入完成，而是将数据写入某个端口。当协调器达到高电平时，数据变得有效。这样，可以根据端口的数据信息来判断是进行读取还是写入操作，从而实现内存的写入（即前文中 CLK）。如果访存操作过于频繁，将出现木桶效应，降低整体性能。为了解决这个问题，可以引入一个特定的存储单元作为缓存。这种缓存能够暂存数据，减少直接访存的次数，从而提升整体性能。

此外，振荡器的第二个功能是作为计时器。假设振荡器每秒产生两千次振荡，也就是高低电平各切换两千次。通过记录振荡次数，我们可以将其转换为时间，从而让电路具备时间的概念。

前面已经简要介绍过与非门，但为了更简洁地描述其实现，此处将使用简化的电路图。注意，本书的主要目的并非深入讲解电路细节，而是通过简化的电路来阐述逻辑概念，帮助读者构建自己的知识体系。因此，电路图可能会省略某些元件（如电阻）以突出核心逻辑。

通过查看图 1.19，可以看到与门和非门的连接方式：与门的输出端连接到非门的输入端，而非门的输出端则连接到与门的另一个输入端。这种配置创建了一个反馈循环，它会影响与门和非门的输出状态。

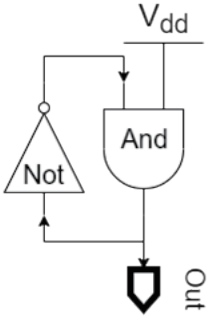

图 1.19　振荡器

反馈循环的工作过程如下。

（1）当系统第一次通电时，V_{dd} 永远是通路的（即为 1），而另一个输入默认是 0。通过与门后，其输出是 0。这个 0 被送到非门，非门将其转换为 1。

（2）与门有一个输入为 1（原始输入），另一个输入也为 1（来自非门的输出）。因此，与门的输出变为 1。

（3）这个 1 再次被送到非门，非门将其转换为 0。

（4）与门的一个输入是 1（原始输入），另一个输入是 0（来自非门的输出）。因此，与门的输出变为 0。

（5）这个过程将不断重复，导致与门和非门的输出在 0 和 1 之间交替切换，形成高电平与低电平之间的振荡。

在 Out 后面添加一个与门，检测为 1 时，使用一个加法器进行计数。这便是振荡器在计数中的应用。

1.7 CPU 的组成

前文介绍了 ALU 的组成，但它并非一个完整的 CPU。首先，它缺少程序计数器，这是用于记录当前代码执行的行数的组件；其次，它没有使用寄存器；另外，它还缺少与内存相关的操作。本节将补充这些内容，使 CPU 更加完整和功能丰富。

1.7.1 PC 计数器

1.6 节在介绍振荡器时，提到了可以通过开关电路来修改输入。然而，由于修改输入的操作不能总是手动进行，因此需要有一个特定的存储空间来记录需要执行的程序。这个存储空间中的程序内容是固定的，但可以通过一个外部可变的存储单元记录访问信息，最终通过读取该单元信息来访问这些程序内容。这个存储单元就是所说的程序计数器，有时也被称为指令指针寄存器（IP，后续将详细介绍）。程序计数器具有多种功能，其中 store 功能用于存储绝对执行地址，inc 功能则使程序能够在当前地址基础上自动跳转到下一条指令，而 reset 功能则可以重置当前的程序计数器。

仿真语言：

```
IN in[16],store,inc,reset;
OUT out[16];

//首先通过 PC 内部的 16 位寄存器（记为 preOut）执行加一操作
Inc16(in = preOut, out = addOut);
//判断当前操作是否需要计数（即是否加一），如果需要，则将 addOut 的计算结果传入 o1
Mux16(a = preOut, b = addOut, sel = inc, out = o1);
//判断当前是否执行加一操作还是 store 操作，将结果存储在 o2
Mux16(a = o1, b = in, sel = store, out = o2);
//判断当前操作是否为 reset（即重置为 0），将结果存储在 o3
Mux16(a = o2, b = false, sel = reset, out = o3);
//将上述操作的结果存储到寄存器中，并在存储的同时将结果输出到 preOut，以便下次使用
Register(in = o3, store = true, out = preOut, out = out);
```

1.7.2　寄存器

CPU 内部配备了多个 16 位寄存器，用于接收用户输入的数据。这些寄存器包括 A 寄存器（ARegister，简称 A）和 D 寄存器（DRegister，简称 D）。A 寄存器主要用于执行算术和逻辑计算、访问存储器地址以及保存指令地址，而 D 寄存器则充当了一个多功能通用寄存器的角色。这样的设计选择对于提升数据处理的高效性和灵活性至关重要。

1.7.3　功能定义

前文多次提及“程序”这一词汇。在设计运算单元时，我们注意到其输入数据均为 16 位。尽管我们在实现过程中已有接触，但该程序并不具备程序计数器操作功能。为了满足 CPU 执行跳转等操作的需求，我们需要在其基础上增加新功能。因此，在此明确具体的功能定义，如图 1.20 所示，该图展示了指令功能定义。

图 1.20　指令功能定义

1. 运算指令定义

在图 1.20 中，第 15 位用于区分当前指令是计算指令还是赋值指令。当第 15 位为 0 时，表示该指令为赋值指令，此时 0～14 位存储的是需要赋值给 ARegister 寄存器的数据。前文已提到，ARegister 不仅用于计算，还包含地址操作，因此有时需要手动指定地址。为了与计算指令区分开，我们采用这一最高位作为赋值指令的标识。然而，这也意味着实际可用的数值位数将减少一位，即只有 0～14 位用于存储数据。在赋值指令下，其他位的定义将不再有效。

运算指令的长度仅为 6 位，这是因为在构建 ALU 时，传入了 6 个标识符。这 6 位与这 6 个标识符一一对应，从而形成了表 1.5，该表格详细定义了这 6 位组合支持的指令。

M 代表当前 ARegister 存储的是地址，也就是需要访存时提供的地址数据。运算指令的 6 位数据与 ALU 对应如表 1.6。

表 1.5　6 位组合支持的指令表

A 为数据含义	11 位	10 位	09 位	08 位	07 位	06 位	A 为地址含义
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
−1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
−D	0	0	1	1	1	1	
−A	1	1	0	0	1	1	−M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

表 1.6　6 位数据与 ALU 对应表

ALU	C 指令	ALU	C 指令
zx	11 位	ny	08 位
nx	10 位	f	07 位
zy	09 位	no	06 位

至此，我们已经验证了上述指令定义与 ALU 对应表的一致性。由于指令数量较多，此处选择其中两个指令进行验证。读者如果对其他指令感兴趣，可以自行进行计算验证。

首先解读第一个指令 0，它表示经过 ALU 计算返回的结果将是 0。

对于指令 0:101010，从最左边第一位开始，各个数位的含义如下。

第 1 位，表示 zx 此处指令生效，而 zx 在 ALU 中代表将 x 设置为 0。

第 2 位，表示 nx 此处指令不生效。

第 3 位，表示 zy 此处指令生效，而 zy 在 ALU 中代表将 y 设置为 0。

第 4 位，表示 ny 此处指令不生效。

第 5 位，表示 f 此处生效，而 f 在 ALU 中代表 x+y。

第 6 位，表示 no 此处指令不生效。

由此得出，0 指令返回结果为 0，这是因为 x 和 y 都被设置为 0，再进行加法计算后结果自然为 0。

对于指令 A+1:110111，从最左边第一位开始，各个数位的含义如下。

第 1 位，表示 zx 此处指令生效，而 zx 在 ALU 中代表将 x 设置为 0。

第 2 位，表示 nx 此处指令生效，而 nx 在 ALU 中代表对 x 取反为−1。

第 3 位，表示 zy 此处指令不生效。

第 4 位，表示 ny 此处指令生效，而 ny 在 ALU 中代表对 y 取反。如果 x 已经取反为−1，那么 y 将是 A；如果 A 为 1，则 y 取反后的值为−2。

第 5 位，表示 f 此处生效，而 f 在 ALU 中代表 x+y。

第 6 位，表示 no 此处指令生效，而 no 在 ALU 中代表 x+y 的结果取反。

由此得出，当 A 为 1 时，A+1 指令的返回结果为 2，因为计算过程是−1+（−2）=−3，再对结果取反得到 2。这样，我们可以看出，虽然单独看指令编码可能不够直观，但是按照 ALU 的操作步骤逐步分析，就能够得到正确的结果。

2. 存储指令定义

在图 1.20 中，3～5 位被定义为存储指令，用于指示 ALU 计算结果应存储的位置。这 3 位表示 8 种不同的存储方式，具体对应如表 1.7 所示。

表 1.7　存储方式对应表

05 位	04 位	03 位	助记符	结　　果
0	0	0	NULL	无须关注结果
0	0	1	M	将数据存储到 A 寄存器指定的内存地址中
0	1	0	D	将数据存储到 D 寄存器中
0	1	1	MD	将数据存储到内存与 D 寄存器中
1	0	0	A	将数据存储到 A 寄存器中
1	0	1	AM	先将数据存储在由 A 寄存器指定的内存地址中，然后将相同的数据存储到 A 寄存器中
1	1	0	AD	将数据存储到 A 寄存器与 D 寄存器中
1	1	1	AMD	将数据存储到 A 寄存器、D 寄存器以及由 A 寄存器指定的内存地址中，需要注意存储顺序，先存储 A 所指定的内存地址，然后修改 A 的值，否则会丢失地址信息。

3. 跳转指令定义

在图 1.20 中，0～2 位被定义为跳转指令，用于在 CPU 内根据计算结果控制程序的执行顺序。这 3 位用于表示不同的跳转条件，具体对应如表 1.8 所示。

表 1.8 跳转条件对应表

02 位	01 位	00 位	助 记 符	结 果
0	0	0	NULL	不跳转
0	0	1	JGT	输出结果大于 0，if out ＞ 0
0	1	0	JEQ	输出结果等于 0，if out = 0
0	1	1	JGE	输出结果大于或等于 0，if out ≥ 0
1	0	0	JLT	输出结果小于 0，if out ＜ 0
1	0	1	JNE	输出结果不等于 0，if out != 0
1	1	0	JLE	输出结果小于或等于 0，if out ≤ 0
1	1	1	JMP	无条件跳转

1.7.4 CPU 实现

在了解了 CPU 内部架构之后，我们得知程序计数器用于控制下一条指令的位置，同时 CPU 使用两个寄存器参与计算过程，并通过 ALU 执行计算。为了模拟这一过程，此处将采用仿真语言来实现这一功能。

```
    //inM：当前 A 寄存器中的值对应于内存地址中的数据，因为在此处还未解析指令
    //所以此时并不知道 A 寄存器的含义是数值还是内存地址
IN  inM[16],
    instruction[16], //当前执行的指令
    reset;//是否重置 CPU

    //out：输出结果
OUT out[16],
    //标记是否需要写回内存
    writeM,
    //需要写回的内存地址。由于无法确定当前指令是赋值指令还是计算指令
    //我们使用最高位用于标志，因此内存地址的位数少了一位
    addressM[15],
    //计算下一次需要执行的地址
    pc[16];

    //判断当前指令是否为计算指令
    And(in = instruction[15], out = isC)
    //判断当前是否为操作的内存，即 A 指向内存地址
    And(in = instruction[12], out = isM)
    //判断当前是否为 A 的存储指令
    Not(in = instruction[15], out = isA)
    //若 store 为 1，代表存储指令，则将 0～14 位存储到 A 寄存器中
    //若 store 为 0，代表计算指令，则不会存储，但是会将 A 寄存器的值读取到 addressM 中
    //这代表将指令复制一份，因为后续解析指令时，A 寄存器可能代表的是内存地址
    ARegister(in = instruction[0..14], store = isA, out = outAR,
              out[0..15] = addressM)
```

```
        //选择接下来的计算值是传入的内存值还是 A 寄存器的值
        Mux16(a = outAR, b = inM, sel = isM, out = cdata)
        //读取 D 寄存器的值
        DRegister(in = false, store=0, out = outDR)
        //解析计算指令，前面已推理过
        And(a = isC, b = instruction[6], out = no)
        And(a = isC, b = instruction[7], out = f)
        And(a = isC, b = instruction[8], out = ny)
        And(a = isC, b = instruction[9], out = zy)
        And(a = isC, b = instruction[10], out = nx)
        And(a = isC, b = instruction[11], out = zx)
        //通过 ALU 进行计算时，注意在输出时有两个结果：一个结果输出到中间变量 outALU 中
        //另一个结果输出到 outM 中
        ALU(x = outDR, y = cdata, zx = zx, nx = nx, zy = zy, ny = ny, f = f,
            no = no, out = outALU, out = outM, zr=zr, ng=ng)
        //存储返回值
        And(in = instruction[5], out = storeA)
        And(in = instruction[4], out = storeD)
        //这两步是为了保证读者能阅读清楚而设计的
        //它们可以与上方的读取 A 与 D 寄存器连写，例如
        //Mux16(a = outALU, b = instruction[0..14], sel = storeA, out = storeAD)
        //ARegister(in = storeAD, store = isA, out = outAR, out[0..15] = addressM)
        ARegister(in = outALU, store = storeA, out = nouse)
        DRegister(in = outALU, store = storeD, out = nouse)
        //判断当前是否需要跳转
        And(a = isC, b = instruction[0], out = isGT)
        And(a = isC, b = instruction[1], out = isEQ)
        And(a = isC, b = instruction[2], out = isLT)
        //判断当前是否满足小于 0 跳转
        And(a = ng, b = isLT, out = isLtJump)
        //判断当前是否满足等于 0 跳转
        And(a = zr, b = isEQ, out = isEqJump)
        //将小于或等于 D 的判断结果取反
        Not(in = ng, out = notNg)
        Not(in = zr, out = notZr)
        //然后进行与操作，若二者都是 1，则返回必为 1，代表当前大于 0
        //因此 ALU 计算结果大于 0
        And(a = notNg, b = notZr, out = isOutGt)
        //判断是否需要大于 0 跳转
        And(a = isOutGt, b = isGT, out = isGtJump)
        //组合所有跳转判断结果，确定是否需要跳转
        Or(a = isLtJump, b = isEqJump, out = isJump)
        Or(a = isJump, b = isGtJump, out = jump)
        //若允许跳转，则 A 寄存器的数据便是地址，将跳转地址传入程序计数器中
        //若不满足跳转条件，则 inc 加一
        //并且返会下一条指令 pc
        PC(in = outAR, load = jump, inc = true, reset = reset, out = pc)
```

至此，CPU 的设计已经完成。它的实现过程并不复杂，主要将前文介绍的功能定义与电子元件进行合理组合。这个过程主要依赖于与、或、非等逻辑运算。如果读者在理解这些推理过程时存在困难，建议先学习离散数学中的数理逻辑部分，这将有助于更深入地理解 CPU 的工作原理。

1.8 计算机的组成

前文详细探讨了 CPU 的组成，包括其内部的计算单元（ALU）、存储单元（寄存器）以及输入与输出机制。尽管前文未具体描述输入与输出的具体实现，但通过参数传递和结果返回，可以明确看出它们主要与内存进行交互。那么，如何在不修改 CPU 结构的前提下，使其能够与其他组件进行交互呢？本节将在前文的基础上，进一步推理并探讨计算机的完整组成，同时讨论如何实现 CPU 与其他组件的协同工作。

1.8.1 内存

在实现 CPU 的过程中，我们发现内存地址少了一位，这一位专门用于区分指令，因此在 CPU 内部的实际运算中不会被使用。这种划分方法被称为地址编址。当 CPU 与外部设备进行交互时，也需要定义一个特定的编址方式以允许读写操作。这些地址的编址方式主要分为两大类：统一编址和独立编址。

1. 统一编址

统一编址的方式使得 I/O 设备地址与内存地址相同，其优点在于操作 I/O 设备可以像操作内存一样（即前文提到的 M）。然而，其缺点是，某个地址一旦被指定为 I/O 交互地址，就不能再像普通内存地址那样用于存储和访问数据。这是因为这些地址现在代表的是具体的硬件设备，例如，屏幕（screen）或键盘（keyboard）。统一编址方式允许 CPU 与外部设备直接通信，但相应地，这部分内存地址的用途会受到限制。

2. 独立编址

与统一编址不同，独立编址采用与内存地址空间分离的 I/O 地址空间。其优点在于端口地址与内存地址互不冲突，提供了清晰的界限。然而，独立编址的缺点在于需要一套独立的操作指令集，因为原有的内存操作指令无法区分操作对象是内存还是 I/O 设备。因此，独立编址通常需要专用的指令，例如 x86 架构中的 IN 和 OUT 指令。这种编址方式能够确保对外部设备的精确控制，但同时增加了系统的复杂性和指令集的数量。

基于前文对编址方式的介绍，读者可能已经推测出此处将采用统一编址方式实现内存

映射。这种设计的主要优势在于，无须修改 CPU 架构即可轻松扩展其他组件。统一编址允许内存和 I/O 设备共享相同的地址空间，从而显著简化系统的设计和实现过程。

```
IN in[16], store, address[15];
OUT out[16];
//使用第 13 位和第 14 位作为统一编址的判断。
//如果第 14 位为 0，则代表进行普通内存操作；如果为 1，则表示其他组件端口
//第 13 位用于区分组件，此处只扩展键盘与屏幕，其中 0 代表显卡，1 则代表键盘
//首先将 store 操作分别解码到 storeRam、storeRam1、storeScreen 和 storeKeyboard 中
DMux4Way(in = store, sel = address[13..14], a = storeRam, b = storeRam1,
         c = storeScreen, d = storeKeyboard);
//组合 storeRam 与 storeRam1 的结果，因为它们都代表普通内存操作
Or(a = storeRam, b = storeRam1, out = storeR);
//将数据存储到内存中
RAM16K(in = in, store = storeR, address = address[0..13], out = outRam);
//当键盘被按下时，计算机会触发相应的事件以存储输入的内容，并调用内存操作进行存储
//同时在读取内存时，将使用复用器判断返回的内容，因此需要读取这部分的数据
Keyboard(in = in, store = storeKeyboard, out = outK);
//当屏幕输出时，也需要将数据存储到对应的位置，处理方式与键盘相同
Screen(in = in, store = storeScreen, address = address[0..12], out = outS);
//进行内存操作时，解复用器用于标识操作的组件
//而在返回时，需要根据复用器选择返回哪个组件的结果
Mux4Way16(a = outRam, b = outRam, c = outS,
          d = outK, sel = address[13..14], out = out);
```

1.8.2　其他设备

如果仅执行固定的程序，如解密操作，当前的设计已经足够应对。然而，随着时代的进步，计算机功能日益丰富，这意味着需要更多设备的支持。这些设备的设计，究其根源，都离不开前文所讲述的内容。因为计算机的根基就在于此，正如 1.1 节提到的“计算机种子”。基础于此，我们逐步推出了整个计算机的设计。此时，读者可能会思考：如果本设计完全依赖电路，那么断电后会发生什么？此外，在当时的年代，元器件的成本非常高昂。如果采用这种设计，内存等需要大量元件的组件，其费用远超出普通用户的承受范围。

1. 磁盘

在探讨图灵机时，我们提到穿孔纸带作为最初的存储介质。随着科技的进步，磁盘逐渐取代了纸带成为主流的存储方式，并且这种存储方式一直沿用至今。提及这一历史背景，是因为需要解决一个核心问题：电路构成的存储系统在断电后数据会丢失。与此不同，磁盘虽然依赖电流运行，但其存储机制基于磁性，因此，其读写效率相对电路来说是较慢的。

磁盘的读写速度通常以 min 为单位，例如 7200 转/min。与之对比，电路的计算速度则以 μs 级进行。因此，在大多数情况下，不会让 CPU 直接读取磁盘数据进行计算，而是先将磁盘数据读取到内存中，作为后续 CPU 计算的预加载数据。这样的设计有效地提高了计算效率，同时确保了数据的安全性和稳定性。

2. 内存分类

为了降低成本，内存被划分为两部分。在之前的讨论中，内存主要是由 Register 组合而成的，但由于 Register 的成本较高，因此在 CPU 内部采用了寄存器与缓存相结合的方式，其中寄存器和缓存仍然使用 Register 组成。为了进一步降低成本并满足内存的需求，我们采用了电容和电路的组合方案，这也是目前常见的内存解决方案。

3. 只读存储器（ROM）

执行固定程序意味着数据不会发生变化，这种类型的数据可以安全地存储在 ROM 中。ROM 的一个显著优势是，即使在断电的情况下，它也能保持存储的数据不变。与磁盘相比，ROM 的一个主要限制是不可写（至少在运行期间如此），这意味着一旦数据被写入 ROM，就无法再对其进行修改或删除。然而，不同的 ROM 介质提供了不同的擦除方法。例如，在可擦除的可编程只读存储器（EPROM）中，可以通过紫外线照射来擦除数据，以实现数据的重复利用。

1.8.3 计算机实现

依靠前面的知识，读者应该已经在脑海中构建了自己的计算机模型，从庞大的计算机系统到微小的 CPU，它们都是基于最初讲述的体系结构。从抽象的概念到现实中的电子器件，CPU 的存储功能依赖于寄存器，计算功能则依赖于运算单元；而在更大的计算机系统中，系统存储功能依赖于磁盘和内存，计算功能则依赖于 CPU。尽管从表面上看，CPU 和计算机似乎是两个互补的集合（计算机内部包含 CPU），但它们的核心设计思想是相通的。

本书介绍的正是这种设计思想，它提供了一种全局的视角来理解计算机系统，包括后续的网络协议和 Web 服务等，都体现了相似的设计理念。这种思想贯穿于本书的始终，也是本书名为“混沌”的灵感来源。通过掌握这种思想，读者能够更深入地理解计算机系统的内在逻辑和工作原理。

```
    IN reset;
    //从 ROM 中读取指令
    ROM32K(address = outPC, out = instruction);
    //执行指令，并获取下一条指令地址
    CPU(reset = reset, inM = readMemory, instruction = instruction, outM =
```

```
storeData, writeM = storeM, addressM = storeAddress, pc = outPC);
    //将计算结果存储到内存中
    Memory(in = storeData, store = storeM, address = storeAddress, out =
readMemory);
```

1.9　网络服务的组成

本节将运用混沌的思想，分析单体网络服务和分布式服务与前述知识的关联，以加深读者对计算机世界的理解。你可能会思考：既然已经有了计算机，为何还需要网络服务？同样地，既然已经有了网络服务器，为何还需要分布式架构？当一条请求被发出时，究竟发生了什么？又该如何确保网络服务能返回期望的结果？

1.9.1　单体网络服务

互联网的历史最早可追溯到 20 世纪 60 年代初期，起源于信息论以及当时对计算机网络的设想。这个设想的初衷是建立一个网络，使不同计算机的用户能够互相通信。随着时间的推移，直到 1990 年底，蒂姆 • 伯纳斯-李推出了世界上第一个网页浏览器和网页服务器，标志着万维网的诞生，并推动了互联网应用的迅速发展。从这一点可以得出，计算机网络的出现是为了在计算机之间交换数据，随着时间的推移，它进一步演变为 Web 服务。

将网络的作用与 CPU 内部的组件连线以及计算机内部各设备之间的连线进行类比，尽管网络是虚拟的（这一点将在后续章节中进行详细解释），但它所发挥的作用与这些连线类似，即连接计算机与计算机，实现信息的传输和共享。图 1.21 清晰地展示了网络服务与计算机的对比。

通过图 1.21，可以看到三者的功能虽然大同小异，但在名称和实现上却存在显著差异。它们都以存储作为对比的基础，而服务器在这里则主要承担数据存储的角色，如文件存储、数据库等功能。如果服务器用于加解密操作，那么它的作用就可以与运算单元相对应。从设计的角度来看，这种对应关系依然成立。

那么，当一条请求被发出时，到底发生了什么？我们可以回顾前面的知识。当指令被输入 CPU 时，它会经过复用器的处理，生成对应的标志，然后将解析后的标志传入 ALU 进行计算，最后将结果返回并写回内存。网络服务的工作流程也是类似的。当一条 login 登录请求被发送到服务器时，服务器会解析请求并匹配对应的 login 执行器，最后将执行器的执行结果返回并保存登录信息。这两者的核心思想是完全一致的。

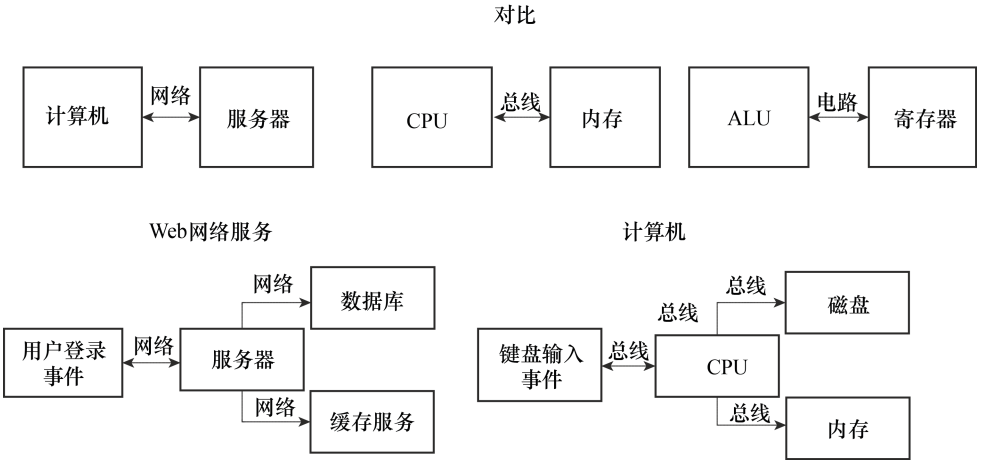

图 1.21　网络服务与计算机的对比

那么，如何让网络服务返回想要的结果呢？在 CPU 中，功能是通过指令集定义的，传入的指令必须与功能定义表中的指令一致，否则结果将无法预料（开发者可以通过传入的指令分析最后的结果，即进行人为处理）。同样，在网络服务中也需要有对应的定义。例如，请求的地址必须是 login 这五个字母，传入的数据也必须符合该功能的要求，否则返回的结果将出错（当然也可以人为地定义错误返回）。因此，我们可以定义单体网络服务的作用是向外提供特定功能的计算机，它使用网络作为传输媒介，通过传输的指令完成对应的功能。从更宏观的角度来看，它就像是一个允许开发者定义功能的 CPU。

1.9.2　分布式服务

在前面的功能定义中，我们了解到每条指令都对应特定的功能，并且功能的执行需要一定的时间。显然，执行一万条指令与一百万条指令所需的时间是不同的。那么，如何提高指令的执行速度呢？一种简单而直接的方法是使用两块或多块 CPU 并行执行指令（此处只考虑指令执行数，并不考虑指令直接的关联关系），这与网络服务中的集群概念相类似。然而，这种方法存在一个问题：如果指令主要是乘法或除法运算（注意，上述的 CPU 设计仍有不足，因为它缺少某些功能，如乘法与除法需要其他独立组件来支持），那么仅仅复制整体功能并不能解决问题。

为了解决这个问题，可以考虑增加多个专门用于乘法或除法运算的组件，以定向增强功能，满足特定指令的需求。在现代 CPU 中，每个核心都包含为提高效率而添加的重复功能运算单元，这正是这种思想的体现。采用这种方式，可以提高指令的执行效率。这种思想，便是分布式思想的核心所在。图 1.22 清晰地展示了网络服务与分布式的对比。

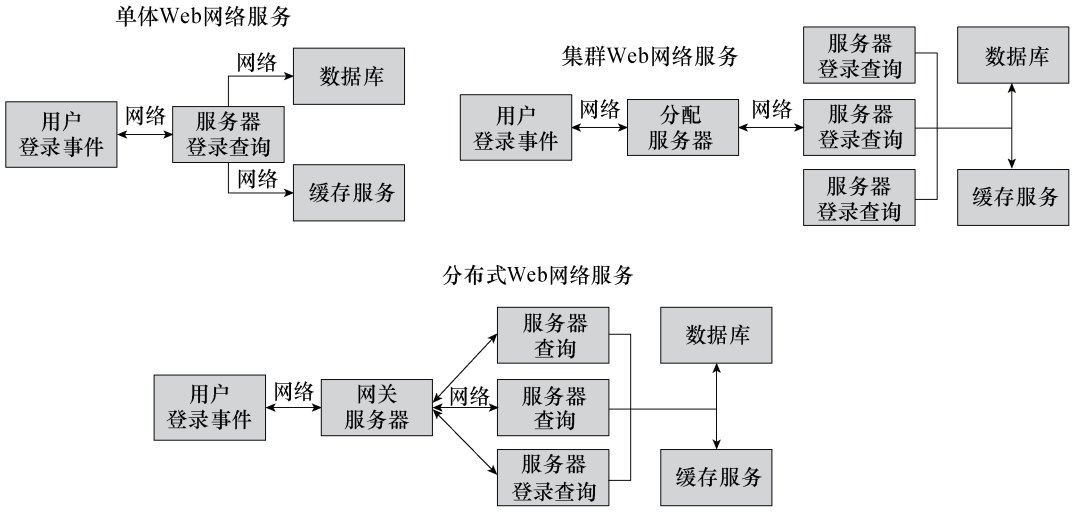

图 1.22　网络服务与分布式的对比

根据对 CPU 扩展的分析，可以对 Web 服务进行相应的扩展，从而得到图 1.22 的设计。在该图中，可以看到存在分配服务器与网关服务器，它们可以被理解为解复用器。这两者的区别在于，分配服务器主要通过轮询等方式根据处理服务器地址进行分配，而网关服务器则需要解析地址并根据指令分配相应的处理服务。实际上，这种设计思想在 CPU 级别已经被广泛应用。因此，学习底层原理对于推理上层设计，是一种非常有效的学习方式。

1.10　小　　结

虽然本章的知识量相当丰富，几乎涵盖计算机系统的各个方面，但这只是构建知识体系的起点。本章真正要表述的信息主要在最后三个小节，它们通过前面的知识积累，逐步进行推理和升华。这个过程不仅深化了对计算机的理解，还展示了知识的连贯性和系统性。接下来，本书将继续深入探索这些概念在现实中的应用，将它们串联起来，最终构建一棵枝繁叶茂的知识树。这棵知识树不仅代表了本书的核心目标，也体现了对计算机科学的全面认识。最后，基于前面的内容，我们绘制了图 1.23，以直观地展示这一知识体系。

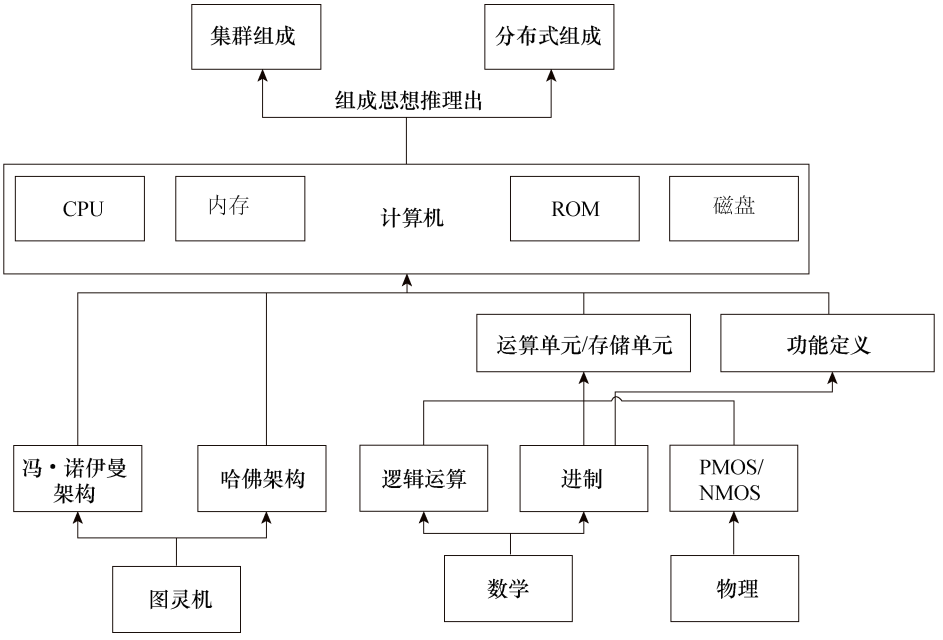
集群组成
分布式组成
组成思想推理出
CPU
内存
计算机
ROM
磁盘
运算单元/存储单元
功能定义
冯·诺伊曼架构
哈佛架构
逻辑运算
进制
PMOS/NMOS
图灵机
数学
物理

图 1.23　本章混沌树