试。如果采用队列的方式，请求可以按照收到的顺序依次处理。存在的问题是即使请求排在队列的最前面，我们也不能保证总是能为其提供服务（因为可能缺乏所需的资源）。描述使用一个队列为5个哲学家服务的情景，即使有的哲学家有两把叉子都可用，但仍然不能为其服务（因为他的请求排在队列的后部）。

6.8.4 [10] <6.7> 如果我们让请求周期性地重试直到资源变为可用，这样是否就解决了6.8.3中的问题？请给出原因。

6.9 考虑下面的3种CPU结构：

- CPU SS：一个双核超标量微处理器，支持在两个功能单元（FU）上的乱序发射。每个核只能运行单线程。
- CPU MT：一个细粒度的多线程处理器，支持来自两个线程中指令的并发执行（例如，有两个功能单元），但是每个周期只能从一个线程发射一条指令。
- CPU SMT：SMT处理器支持来自两个线程的指令并发执行（例如，有两个功能单元），并且发射的指令可来自任意一个线程或者两个线程。
- 假定我们在这些CPU上运行线程X和线程Y，具体操作如下：

线程A	线程B
A1：需三个周期执行	B1：需两个周期执行
A2：无相关性	B2：与B1使用的一个功能单元冲突
A3：与A1使用的一个功能单元冲突	B3：需要B2的结果
A4：需要A3的结果	B4：无相关性，需要两个周期执行

除非特别标记或者遇到冒险，假定所有指令的执行都需要一个周期。

6.9.1 [10] <6.4> 如果使用一个SS CPU，执行这两个线程需要多少个周期？冒险浪费了多少发射槽？

6.9.2 [10] <6.4> 如果使用两个SS CPU，执行这两个线程需要多少个周期？冒险浪费了多少发射槽？

6.9.3 [10] <6.4> 如果使用一个MT CPU，执行这两个线程需要多少个周期？冒险浪费了多少发射槽？

6.9.4 [10] <6.4> 如果使用一个SMT CPU，执行这两个线程需要多少个周期？冒险浪费了多少发射槽？

6.10 虚拟化软件正在用于降低管理高性能服务器的成本，包括VMWare、Microsoft和IBM公司在内的很多公司正在开发一系列的虚拟化产品。第5章中介绍的管理程序层（hypervisor layer）位于硬件和操作系统之间，使多个操作系统可以共享同一物理硬件。管理程序层负责分配CPU和存储资源，同时处理原本由操作系统完成的服务（如I/O）。

虚拟化为宿主操作系统和应用软件提供了底层硬件的一个抽象，使得若干操作系统可并行运行在共享的CPU和存储上。我们需要重新考虑未来如何设计多核和多处理器系统以对此进行支持。

6.10.1 [30] <6.4> 选择现在市场上的两种管理程序，比较它们虚拟化和管理底层硬件（CPU和存储）的方式。

6.10.2 [15] <6.4> 为了更好地满足资源需求，未来的多核CPU平台需要做出哪些改变？例如，多线程技术是否可以减轻计算资源间的竞争？

6.11 我们将讨论如何高效地执行下面的代码。假设有两个不同的机器，一个是MIMD，另一个是SIMD。

```
for (i=0; i<2000; i++)
  for (j=0; j<3000; j++)
        X_array[i][j] = Y_array[j][i] + 200;
```

6.11.1 [10] <6.3> 对于一个包含 4 个 CPU 的 MIMD 机器，请给出每个 CPU 上执行的 RISC-V 指令序列。此 MIMD 机器的加速比是多少？

6.11.2 [20] <6.3> 对一个宽度为 8 的 SIMD 机器（例如，包含 8 个并行的 SIMD 功能单元），使用你自己的对 RISC-V 的 SIMD 扩展，编写一个执行该循环的汇编程序，并比较 SIMD 和 MIMD 机器上执行指令的数量。

6.12 MISD 机器的一个例子是脉动阵列（systolic array）。它是一个由数据处理单元构成的流水线网络或“波阵面”。这些单元都不需要程序计数器，因为执行是由数据的到达触发的。时钟脉动阵列通过与每个处理器相“锁步”的方式进行计算，而这些处理器承担了交替的计算和通信。

6.12.1 [10] <6.3> 分析脉动阵列的各种实现机制（可以在互联网或出版物中查找相关资料），然后使用 MISD 模型对 6.11 中的循环进行编程，并对遇到的问题进行讨论。

6.12.2 [10] <6.3> 使用数据级并行的术语，讨论 MISD 和 SIMD 之间的相似点和不同点。

6.13 假定我们要使用 NVIDIA 8800 GTX GPU 上的 RISC-V 向量指令执行 DAXPY 循环（见 6.3.2 节）。在这一问题中，假定所有算术操作是单精度浮点数运算（因此我们将其重新命名为 SAXPY）。假定指令执行的周期数如下所示。

Loads	Stores	Adds	Multiplies
5	2	3	4

6.13.1 [20] <6.6> 请描述在一个 8 核处理器中如何构建 warp 来完成 SAXPY 循环？

6.14 从 https://developer.nvidia.com/cuda-toolki 下载 CUDA Toolkit 和 SDK。注意使用代码的“emure-lease”（Emulation Mode）版本（此版本可在没有 NVIDIA 硬件的情况下运行）。编译 SDK 中提供的示例程序，并确认它们运行在仿真器上。

6.14.1 [90] <6.6> 以 SDK 的示例程序为起点，编写一个完成如下向量操作的 CUDA 程序：

1. $a-b$（向量减法）

2. $a \cdot b$（向量点积）

向量 $a=[a_1, a_2, \cdots, a_n]$ 和 $b=[b_1, b_2, \cdots, b_n]$ 的点积定义如下：

$$a \cdot b=\sum_{i=1}^{n} a_i b_i=a_1 b_1+a_2 b_2+\cdots+a_n b_n$$

运行编写的程序并验证结果是否正确。

6.14.2 [90] <6.6> 如果你有可用的 GPU 硬件，请完成对程序的性能分析。并且对于不同的向量规模，查看在 GPU 和一个 CPU 版本上的计算时间，并解释其中的原因。

6.15 AMD 最近宣布将把 GPU 与 x86 核集成到一个封装中（尽管两者的时钟不同）。这是一个异构多核的实例。设计的关键之一是如何支持 CPU 和 GPU 之间的高速数据通信。在 AMD 的 Fusion 体系结构之前，分散的 CPU 和 GPU 芯片之间需要通信。目前的计划是采用多个（至少 16 个）PCI express 通道来实现高速通信。

6.15.1 [25] <6.6> 比较这两种互连技术的带宽和延迟。

6.16 参照图 6-14b 中给出的 3 阶 n 维立方体互连拓扑结构，它将 8 个节点进行了互连。n 维立方体互连拓扑的一个优势是在部分互连损坏的情况下依然可以保持连接性。

6.16.1 [10] <6.8> n 维立方体中最多有多少互连损坏时还能保证任何节点依然能够连接？请写出计算公式。

6.16.2 [10] <6.8> 比较 n 维立方体和全互连网络的可靠性。随着损坏的连接增加，画图比较两种拓扑何时会连接失效。

6.17 基准测试程序用于在指定的计算平台上运行有代表性的工作负载，从而比较不同系统之间的性能。在本题中，我们将比较两种基准测试程序：Whetstone CPU 基准测试程序和 PARSEC 基准测试集。从 PARSEC 中选择一个程序。所有程序都可从网上免费下载。考虑将 Whetstone 的多份拷贝或 PARSEC 运行在 6.11 节描述的系统上。

6.17.1 [60] < 6.10 > 两种工作负载运行在这些多核系统上时，本质区别是什么？

6.17.2 [60] < 6.10 > 使用 Roofline 模型的相关术语，分析在运行这些基准测试程序时，运行结果与工作负载的共享和同步数量的相关性。

6.18 在计算稀疏矩阵时，存储的延迟至关重要。由于稀疏矩阵缺乏矩阵操作中常见的空间局部性，所以需要研究新的矩阵表示方法。

最早的稀疏矩阵表示方法之一是 Yale 稀疏矩阵格式。它使用 3 个一维数组存储 $m \times n$ 的矩阵 M。令 R 代表 M 中的非零项数目。我们构造一个长度为 R 的数组 A 存储 M 中的所有非零项（按照从左到右、从上到下的顺序）。我们再构造一个长度为 m+1 的数组 IA（每行一项，再加一）。$IA(i)$ 包含第 i 行中第一个非零项在 A 中的索引号。原矩阵中第 i 行的元素可从 $A(IA(i))$ 到 $A(IA(i+1)-1)$ 得到。第三个数组 JA 包含 A 中每个元素的列号，因此它的长度也为 R。

6.18.1 [15] < 6.10 > 分析下面的稀疏矩阵 X，并编写 C 程序将其存储为 Yale 稀疏矩阵格式。

```
Row 1 [1, 2, 0, 0, 0, 0]
Row 2 [0, 0, 1, 1, 0, 0]
Row 3 [0, 0, 0, 0, 9, 0]
Row 4 [2, 0, 0, 0, 0, 2]
Row 5 [0, 0, 3, 3, 0, 7]
Row 6 [1, 3, 0, 0, 0, 1]
```

6.18.2 [10] < 6.10 > 在存储空间方面，假定矩阵 X 中的每个元素都是单精度浮点格式，如果用 Yale 稀疏矩阵格式存储上面的矩阵，请计算共需多少存储空间。

6.18.3 [10] < 6.10 > 执行下面给出的矩阵 X 和矩阵 Y 的矩阵乘。

```
[2, 4, 1, 99, 7, 2]
```

将该计算放入循环中，并对执行过程进行计时。确保增加循环执行的次数，以在时间测量中获得较好的分辨率。比较原始表示方法的矩阵运行时间和使用 Yale 稀疏矩阵格式的运行时间。

6.18.4 [15] < 6.10 > 你是否能够找到更加有效的稀疏矩阵表示方法（考虑空间和计算开销）？

6.19 在未来的系统中，我们期待能够看到由异构 CPU 构成的异构计算平台。在嵌入式处理相关市场，一些同时包含浮点 DSP 和微控制 CPU 的多芯片模块包的系统已经开始出现。

假定你有三类 CPU：

- CPU A：每周期可执行多条指令的中速多核 CPU（有一个浮点单元）。
- CPU B：每周期可执行单条指令的快速单核定点 CPU（无浮点单元）。
- CPU C：每周期可执行相同指令的多个副本的慢速向量 CPU（具备浮点运算能力）。

假定处理器在下面的频率运行：

CPU A	CPU B	CPU C
1 GHz	3 GHz	250 MHz

在每个时钟周期，CPU A 可以执行 2 条指令，CPU B 可以执行 1 条指令，CPU C 可以执行 8 条指令（来自相同指令）。假定所有的操作在单周期延迟中完成执行，且没有任何冒险。

三个 CPU 均可执行定点算术运算，但是 CPU B 不能执行浮点算术运算。CPU A 和 B 具有与

RISC-V 处理器相似的指令系统。CPU C 仅能执行浮点加减以及存储器存取操作。假定所有 CPU 均可访问共享存储，并且同步的开销为零。

我们的任务是比较两个矩阵 X 和 Y，它们每个都包含 1024×1024 个浮点元素。输出结果应是指示矩阵 X 中某处的值比矩阵 Y 中的值大或相等的个数。

6.19.1 [10] < 6.11 > 请描述如何划分该问题到 3 个不同的 CPU 上，以获得最佳性能。

6.19.2 [10] < 6.11 > 为向量 CPU C 中增加哪类指令，以获得更好的性能？

6.20 本题着眼于给定最大事务处理速率的系统中发生的排队量，以及事务观察到的平均延迟。延迟包括服务时间（根据最大速率计算）和排队时间。

假定一个四核计算机系统可以在稳定状态处理最大请求速率的数据库查询，同时假定每个事务平均花费固定的时间来处理。下表给出了几对事务延迟和处理速率。对于表中的每一对数据，回答如下问题：

平均事务延迟	最大事务处理速率
1ms	5000/sec
2ms	5000/sec
1ms	10 000/sec
2ms	10 000/sec

6.20.1 [10] < 6.11 > 在任意时刻，平均有多少请求被处理？

6.20.1 [10] < 6.11 > 如果移到 8 核的系统中，理想情况下，系统的吞吐量将发生什么变化（计算机每秒处理多少请求）？

6.20.1 [10] < 6.11 > 讨论为什么通过简单地增加核的数量，我们很少获得这种加速？

自我检测答案

6.1 节　错误。任务级并行可以帮助串行应用程序，可以使串行应用在并行硬件上运行，尽管会有很多挑战。

6.2 节　错误。弱比例缩放可以补偿程序的串行部分，强比例缩放的缩放性会被串行部分所限制。

6.3 节　正确。但是它们缺少可以提升向量体系结构性能的特性，如聚集 – 分散和向量长度寄存器。（就像这节的“详细阐述”中提到的，AVX2 SIMD 扩展通过聚集操作提供了变址加载，但不通过分散操作提供变址存储。Haswell x86 微处理器是第一个支持 AVX2 的微处理器。）

6.4 节　1. 正确。2. 正确。

6.5 节　错误。由于共享地址是物理地址，且多任务中的每个任务都在它们自己的虚拟地址空间中，因而可在共享存储的多处理器上良好运行。

6.6 节　错误。图形 DRAM 因其更高的带宽而被赞扬。

6.7 节　1. 错误。发送和接收消息是一个隐式的同步，同样也是一种共享数据的方式。2. 正确。

6.8 节　正确。

6.10 节　正确。我们或许需要在硬件的所有层次和软件栈上进行革新，以使并行计算获得成功。

附录 A

Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition

逻辑设计基础

A.1 引言

本附录简要讨论了逻辑设计的基础知识，但无法取代逻辑设计课程，也不能保证让你设计出重要的逻辑系统。不过，如果你很少或根本没有接触过逻辑设计，本附录将为你理解本书中的材料提供足够的背景知识。此外，本附录还能帮助你了解计算机是如何实现的。如果仍想进一步了解有关逻辑设计的知识，可以参阅书末参考文献提供的信息。

我一如既往地爱着这个词——布尔。
Claude Shannon, IEEE Spectrum, 1992.4
(Shannon 的硕士论文表明，George Boole 在 19 世纪发明的代数可以表示电子开关的运作。)

A.2 节介绍了逻辑电路的基本组成单元——门。A.3 节使用这些门来构建不包含存储器件的简单组合逻辑系统。如果接触过逻辑或数字系统，会比较熟悉前两部分内容。A.5 节介绍了如何使用 A.2 节和 A.3 节的思想来设计 RISC-V 处理器的 ALU。A.6 节介绍了如何构建快速加法器，如果你对此不感兴趣，可以放心跳过。A.7 节简要介绍了“时钟”，这是讨论存储元件如何工作所必需的。A.8 节介绍了存储器件，A.9 节进一步扩展并集中介绍随机访问存储器；这两节描述了存储器件的特点和背景，这些特点对于理解如何使用存储器件来说至关重要（如第 4 章所述），该背景引发了存储器层次结构设计的多个方面（如第 5 章所讨论）。A.10 节描述了有限状态自动机的设计和使用，有限状态自动机又被称为时序逻辑单元。如果打算阅读附录 C，需要完全理解 A.2 ～ A.10 节的内容。如果只阅读第 4 章中控制相关的内容，可以仅浏览附录，但也应该大致熟悉除 A.11 节以外的所有内容。A.11 节面向那些希望更加深入理解时钟同步方法和时序的读者，解释了边沿触发的时钟同步逻辑工作的基本原理，介绍了另一种时钟同步方案，并简要描述了异步输入的同步问题。

本附录中，在适当的地方增加了描述逻辑的 Verilog 代码段（将在 A.4 节中介绍）。完整的 Verilog 教程收录在本书的网站中。

A.2 门、真值表和逻辑方程

现代计算机的内部电子元件是数字电路。数字电子元件仅在两个电压水平下运行：高电压和低电压。所有其他电压值均为瞬时值，且出现在两个电压值转换过程中。（正如本节后面所讨论的，数字电路设计中可能存在的缺陷是：当信号不是明确的高电压或低电压时就对信号进行采样。）数字式的计算机也是其使用二进制数的一个重要原因，因为二进制系统可以匹配电子元件中的底层抽象。在不同的逻辑系列中，两个电压值以及它们之间的关系是不同的。因此，我们不参考电压水平的高低，而是谈论（逻辑上）为真、为 1 或**有效**（asserted）的信号，或者（逻辑上）为假、为 0 或**无效**（deasserted）的信号。称值 0 和 1 彼此*互补*或*反转*。

有效信号：（逻辑上）为真或 1 的信号。

根据逻辑块是否包含存储器，将其分为两类。不包含存储器的块称为组合电路，组合电路的输出仅取决于当前输入。在含有存

无效信号：（逻辑上）为假或 0 的信号。

储器的块中，输出可以由输入和存储在存储器中的值（称该值为逻辑块的状态）共同决定。在本节和下一节中，我们仅关注**组合逻辑**（combinational logic）。在 A.8 节介绍完不同的存储元件之后，再描述如何设计包含状态的**时序逻辑**（sequential logic）。

组合逻辑：块中不包含存储器件且因此给定相同输入时计算相同输出的一个逻辑系统。

时序逻辑：块中包含存储器件且因此输出值同时取决于输入和当前存储器中内容的一组逻辑元件。

A.2.1 真值表

由于组合逻辑块不包含存储器，因此通过为每个可能的输入值集合定义对应的输出值就可以完全指定一个组合逻辑电路。这种确定的对应关系通常用真值表给出。对于一个包含 n 个输入的逻辑块，存在许多可能的输入值组合，因此真值表含有 2^n 个表项。每个表项为特定输入组合指定所有的输出值。

例题 | 真值表

假设一个具有 A、B、C 三个输入和 D、E、F 三个输出的逻辑函数。函数定义如下：如果有一个输入为真，则 D 为真；如果有两个输入为真，则 E 为真；仅当三个输入都为真时，F 才为真。写出该函数的真值表。

答案 | 该真值表应包含 $2^3=8$ 个表项。如下所示：

输入			输出		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

真值表可以完整地描述任何组合逻辑函数，但其表项增长很快，且可能不容易理解。有时我们想构造一个对于许多输入组合均为 0 的逻辑函数，此时可以使用仅列举非零输出表项的简化真值表。这种方法在第 4 章和附录 C 中用到。

A.2.2 布尔代数

另一种方法是用逻辑方程表示逻辑函数。这可以通过使用布尔代数（以 19 世纪数学家布尔的名字来命名）来完成。在布尔代数中，所有变量的值非 0 即 1，在典型的表达式中有三个运算符：

- 或操作记作 +，如 $A + B$。如果任一变量为 1，则或操作的结果为 1。因为任一操作数为 1，则结果为 1，所以或操作也称为逻辑和。
- 与操作记作 ·，如 $A \cdot B$。只有当两个输入都为 1 时，与操作的结果才为 1。因为仅当两个操作数均为 1 时，结果才为 1，所以与操作也称为逻辑积。

- 一元非操作写为 $\overline{A}$。仅当输入为 0 时，非操作的结果才为 1。我们将非操作应用于逻辑取反（例如，如果输入为 0 则输出为 1，反之亦然）。

布尔代数中的某些定律有助于操作逻辑方程。

- 恒等定律：$A+0=A$，$A \cdot 1=A$
- 0/1 定律：$A+1=1$，$A \cdot 0=0$
- 互补律：$A+\overline{A}=1$，$A \cdot \overline{A}=0$
- 交换律：$A+B=B+A$，$A \cdot B=B \cdot A$
- 结合律：$A+(B+C)=(A+B)+C$，$A \cdot (B \cdot C)=(A \cdot B) \cdot C$
- 分配律：$A \cdot (B+C)=(A \cdot B)+(A \cdot C)$，$A+(B \cdot C)=(A+B) \cdot (A+C)$

除此之外，还有两个有用的定律，称为德摩根（DeMorgan）定律，在练习中将进行深入讨论。

任何一组逻辑函数都可以写成一系列方程式，每个方程式的左侧为输出，右侧为变量和上述三种操作符组成的式子。

例题 | 逻辑方程式

写出上一示例中描述的逻辑函数 D、E、F 的逻辑方程式。

答案 | D 的方程式为：

$$D=A+B+C$$

F 的方程式同样简单，为：

$$F=A \cdot B \cdot C$$

E 的方程式稍显复杂。可以将其分为两部分：当 E 为真时，哪些输入组合必须为真（三个输入中的两个必须为真），而哪些输入组合不可能为真（所有三个都不为真）。因此 E 可以写成：

$$E=((A \cdot B)+(A \cdot C)+(B \cdot C)) \cdot (\overline{A} \cdot \overline{B} \cdot \overline{C})$$

我们也可以通过另一种方法得到 E 的逻辑方程式：当且仅当两个输入均为真时，E 才为真。因此，可以将 E 写为具有两个输入为真和一个输入为假的输入组合的或操作：

$$E=(A \cdot B \cdot \overline{C})+(A \cdot C \cdot \overline{B})+(B \cdot C \cdot \overline{A})$$

章末练习中将验证两个逻辑等式是等价的。

在 Verilog 中，我们尽可能使用赋值语句来描述组合逻辑，该语句将从 A.4 节开始介绍。我们可以使用 Verilog 的异或操作来定义 E：`assign E =(A&(B^C))|(B&C&~A)`，这是该函数的另一种描述方式。D 和 F 的表示就更简单了（就像对应的 C 代码一样）：`assign D = A|B|C,assign F = A&B&C`。

A.2.3 门

逻辑块由实现基本逻辑功能的门（gate）构成。例如，与门实现与操作，或门实现或操作。由于与和或操作都是可交换、可结合的，因此与门和或门可以有多个输入，输出等于所有输入的与操作或者或操作。逻辑非操作通过一个始终具有单个输入的反相器实现。这三种逻辑构建块的标准表示如图 A-2-1 所示。

门：实现基本逻辑函数的设备，例如与门、或门。

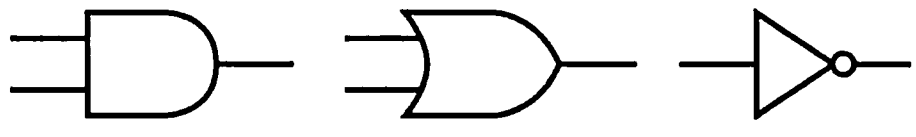


图 A-2-1 与门、或门、非门的标准表示形式。每个门的左侧为输入信号，右侧为输出信号。与门和或门均为两个输入信号，非门仅有单个输入信号

通常不直接绘制反相器，而是在门的输入 $\overline{A+B}$ 或输出端添加“气泡”，以表示该输入线或输出线上的逻辑值取反。例如，图 A-2-2 展示了函数的逻辑图，左侧使用显式反相器表示，右侧使用带“气泡”的输入和输出表示。

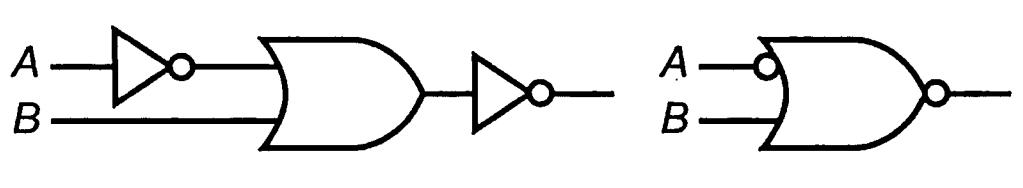


图 A-2-2 用逻辑门实现 $\overline{A+B}$，左侧使用显式反相器，右侧使用带气泡的输入和输出。该逻辑函数可以被简化为 $A \cdot \overline{B}$ 或用 Verilog 表示为 A&~B

与、或、非门可以构造任意逻辑函数；可以通过章末练习来尝试使用门电路实现一些常见的逻辑函数。在下一节中，我们将介绍如何使用这些门电路来构建任意逻辑函数。

事实上，所有逻辑函数都能用单一门电路构建，只要该门电路是反相门电路。两种常见的反相门电路为**或非门**（NOR）和**与非门**（NAND），分别对应于或（OR）门的反相操作和与（AND）门的反相操作。或非门和与非门被称为*万能门电路*，因为任何逻辑函数都可以使用这种类型的门电路构建。练习中将进一步探索这一概念。

或非门：或门取反。

与非门：与门取反。

自我检测 以下两个逻辑表达式是否等价？如果不等价，找出使其不相等的变量值。

- $(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$
- $B \cdot (A \cdot \overline{C} + C \cdot \overline{A})$

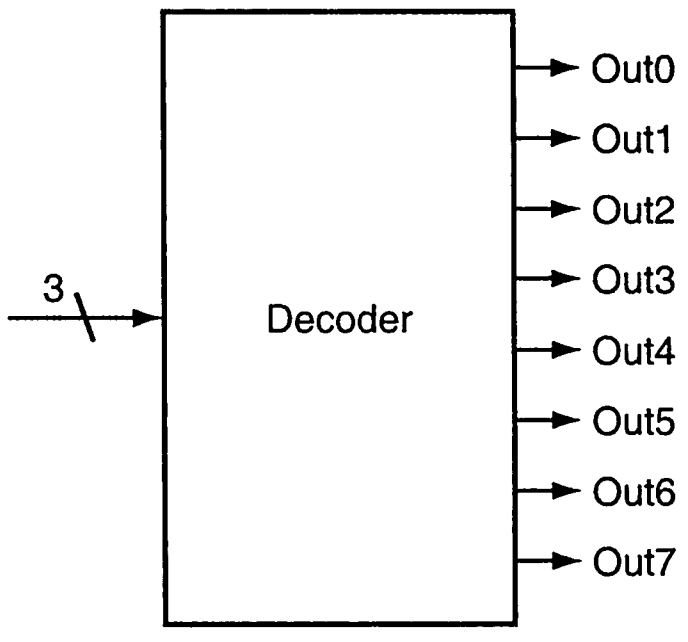
A.3 组合逻辑电路

在本节中，我们将介绍一些常用的较大的逻辑单元，并讨论结构化逻辑设计，这些结构化逻辑可以通过一个根据逻辑方程或真值表的转换程序自动实现。最后，我们将讨论逻辑阵列的概念。

A.3.1 译码器

译码器（decoder）是用于构造更大组件的一种逻辑单元。最常见的译码器有 n 位输入和 2^n 个输出，其中每种输入组合仅对应一个输出。该译码器将 n 位输入转化为对应于 n 位二进制值的信号。因此 n 个输出通常被标作 Out0，Out1，…，Out2^n-1。如果输入的值是 i，那么 Out$i$ 为真，其他所有输出均为假。图 A-3-1 给出了一个 3 位译码器及其对应的真值表。该译码器有 3 位输入和 8（2^3）个输出，因此称为 3-8 译码器。此外，还有一种称作编码器的逻辑元件，它与译码器的功能正好相反。编码器有 2^n 个输入并产生 n 位输出。

译码器：具有 n 位输入和 2^n 个输出的逻辑单元，其中每种输入组合仅对应于一个输出。



a）3位译码器

输入			输出							
I2	I1	I0	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b）3位译码器对应的直值表

图 A-3-1　3 位译码器，包含 3 位输入（I2、I1、I0）和 2^3=8（Out0~Out7）个输出。只有与输入的二进制值相对应的输出为真（如真值表所示）。输入处的标号 3 表示输入信号为 3 位宽

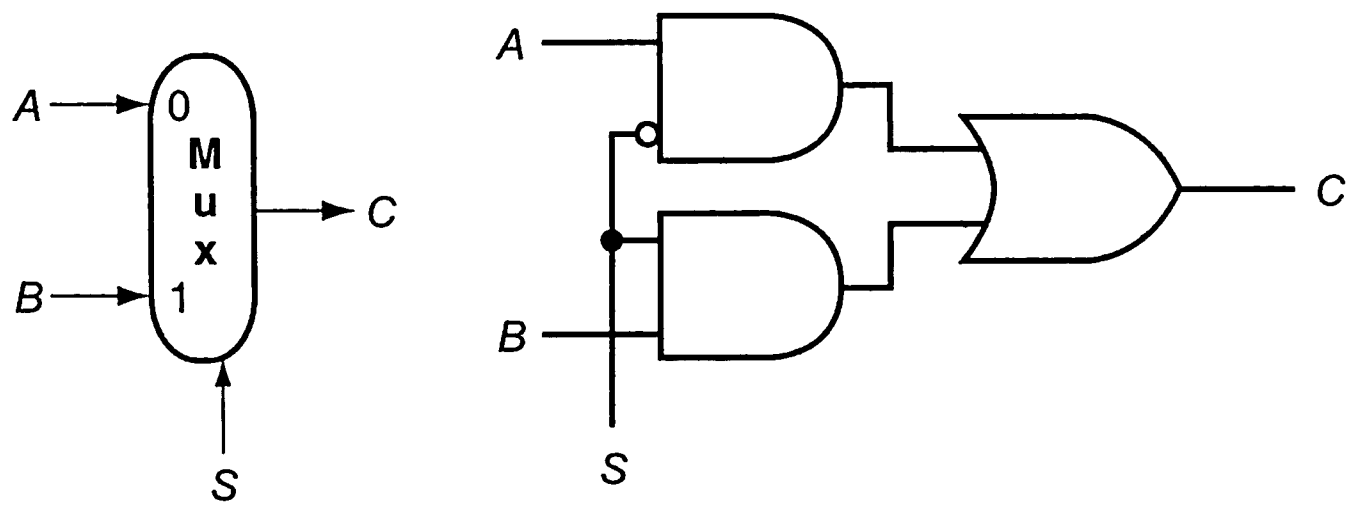


图 A-3-2　左侧为两输入多选器，右侧是其门电路实现。多选器有两个输入端（*A* 和 *B*），分别标记为 0 和 1，以及一个选择信号（*S*）和一个输出端（*C*）。用 Verilog 实现多选器需要相对较多的工作量，尤其是当输入端大于两位宽时。我们将从 A.4 节开始讲解如何实现

A.3.2　多选器

在第 4 章中经常用到的一个基本逻辑功能单元就是多选器。称其为选择器可能比多选器更为合适，因为它的输出是控制元件从输入中选出的一个。考虑双输入多选器。图 A-3-2 的左侧给出了该多选器的三个输入：两个数据值和一个**选择器（控制）值**（selector（control）value）。选择器值确定哪个输入信号将成为输出信号。图 A-3-2 右侧的门电路表示由双输入多选器计算的逻辑函数：$C=(A\cdot S)+(B\cdot S)$。

选择器值：也称作控制值。用于从多选器的输入信号中选出一个作为其输出信号的控制信号。

多选器可以有任意数量的输入信号。当只有两个输入时，选择器只需要单个信号，如果选择信号为真（1），则选择其中一个输入作为输出；如果选择信号为假（0），则选择另一个输入作为输出。如果有 n 个数据输入，则需要$\lceil \log_2 n \rceil$个选择信号。此时的多选器包含以下三个部分：

1. 产生 n 个信号的译码器，每个信号指示一个不同的输入信号值
2. n 个与门组成的阵列，每个与门将一个输入信号和对应于译码器的一个信号相结合。
3. 单个大的或门，用来合并与门的输出。

为了将输入信号与选择器值相关联，我们经常用数字来标记数据输入信号（如 0，1，2，3，…，$n-1$），并将数据选择器输入信号转化为二进制数。有时，也使用具有未解码选择信

号的多选器。

在 Verilog 中，通过使用 if 语句，可以很容易地以组合方式表示多选器。case 语句对于更大的多选器来说更方便，但必须注意组合逻辑合成。

A.3.3 两级逻辑和 PLA

如上节所述，任何逻辑函数都只能用与、或、非功能实现。事实上，存在更强大的实现方式。任何逻辑函数都可以写成一种规范形式，其中每个输入非真即假，且只有两级门电路（与门和或门），最终输出可以取非。这种表示方法称为两级表示，它有两种形式：**析取范式**（sum of products）和合取范式（product of sums）。合取范式表示的是对逻辑积（与操作）结果求逻辑和（或操作），析取范式则恰好相反。在前面的示例中，有两个输出 E 的等式：

析取范式：一种用来对逻辑积（与操作）求逻辑和（或操作）的逻辑表达式。

$$E=((A\cdot B)+(A\cdot C)+(B\cdot C))\cdot(\overline{A\cdot B\cdot C})$$

和

$$E=(A\cdot B\cdot\overline{C})+(A\cdot C\cdot\overline{B})+(B\cdot C\cdot\overline{A})$$

其中第二个表达式即为析取范式表示形式：具有两级逻辑且只对单个变量取非。第一个表达式包含三级逻辑。

详细阐述 E 也可以写作合取范式表示形式：

$$E=(\overline{A}+\overline{B}+C)\cdot(\overline{A}+\overline{C}+B)\cdot(\overline{B}+C+A)$$

为了得到这种表示形式，需要使用德摩根定律，章末练习中将对其进行讨论。

在本章中，我们使用析取范式形式。显而易见，任何逻辑函数都可以通过其真值表来构造出析取范式表示形式。使函数为真的每个真值表项对应于一个乘积项。乘积项由所有输入或输入取反后的逻辑积组成，是否取反取决于真值表中变量对应的表项是 0 还是 1。逻辑函数就是使函数为真的逻辑积的逻辑求和。通过示例可以更容易地理解这一点。

例题 | 析取范式

写出以下真值表中 D 的析取范式表达式。

输入			输出
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

答案 | 真值表中，在 4 种不同的输入组合下 D 为真（1），因此有 4 个乘积项。分别如下：

$$\overline{A}\cdot\overline{B}\cdot C$$

$$\overline{A}\cdot B\cdot C$$

$$A \cdot \overline{B} \cdot \overline{C}$$

$$A \cdot B \cdot C$$

由此，可以将以上乘积项逻辑相加，得到 D 的函数：

$$D = (\overline{A} \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C)$$

注意：只有使函数为真的真值表表项，才能在等式中生成对应的乘积项。

可以利用真值表和两级表示之间的关系来生成任何逻辑函数集的门级实现。一组逻辑函数对应于一个含有多个输出列的真值表，正如 A.2.1 节的例题所示。每个输出列表示一个不同的逻辑函数，都可以直接根据真值表构造出来。

析取范式表示对应于一种称作**可编程逻辑阵列**（Programmable Logic Array，PLA）的常用结构化逻辑实现。PLA 具有一组输入及其输入取反（可以通过一组反相器实现）的逻辑单元集和两级逻辑结构。第一级逻辑结构是与门阵列，用以实现**乘积项**（product term）（有时称为**小项**（minterm）），每个乘积项可以由任意输入或输入取反组成。第二级逻辑结构是或门阵列，每个或门产生任意数量的乘积项的逻辑和。图 A-3-3 给出了 PLA 的基本形式。

可编程逻辑阵列：一种结构化逻辑单元，由一组输入和对应的输入取反，以及两级逻辑组成。第一级逻辑用于生成输入和输入取反的乘积项，第二级用于生成乘积项的逻辑和。因此，PLA 将逻辑功能实现为析取范式。

小项：也称作乘积项，通过连接符（与操作）连接的一组逻辑输入。乘积项形成 PLA 的第一级逻辑。

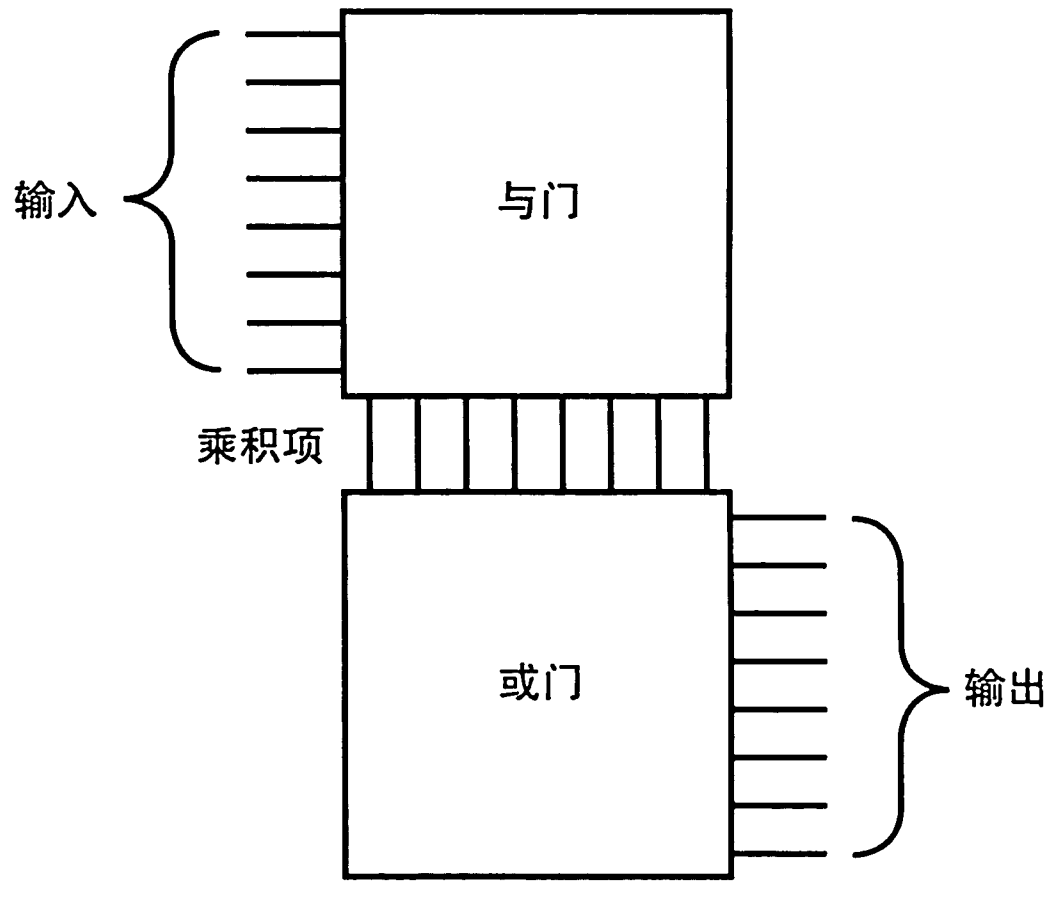


图 A-3-3　PLA 的基本组成形式：与门阵列以及紧随其后的或门阵列。与门阵列的每一项为若干输入或输入取反组成的乘积项。或门阵列的每一项为若干乘积项的逻辑和

PLA 可以根据真值表直接实现多输入 / 输出的一组逻辑函数。输出为真的每个表项都对应一个乘积项，因此 PLA 中需要有相应的行。每个输出对应于第二级逻辑结构中或门阵列的行。或门的数量对应于真值表中输出为真的表项数量。如图 A-3-3 所示，PLA 的总大小等于与门阵列（称为*与平面*）的大小和或门阵列（称为*或平面*）的大小的总和。观察图 A-3-3 可以看到，与门阵列的大小等于输入变量个数乘以不同乘积项个数，或门阵列的大小为输出变量个数乘以乘积项个数。

PLA 的两个特点决定其能有效地实现一组逻辑函数。首先，真值表表项中，至少一个

输出为真时，才具有对应的逻辑门。其次，每个不同的乘积项只对应于 PLA 中的一个输入，即使该乘积项用于多个输出。下面看一个示例。

例题 | PLA

考虑 A.2.1 节例题中定义的逻辑函数集。写出该示例中 D、E、F 的 PLA 实现。

答案 | 这是我们在前面构造的真值表：

输入			输出		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

由于输出部分至少有 1 个真值的独立乘积项有 7 个，因此与门阵列将有 7 列。与门阵列中行数为 3（因为有 3 个输入），或门阵列也是 3 行（因为有 3 个输出）。图 A-3-4 给出了最终的 PLA，乘积项从上到下对应真值表表项。

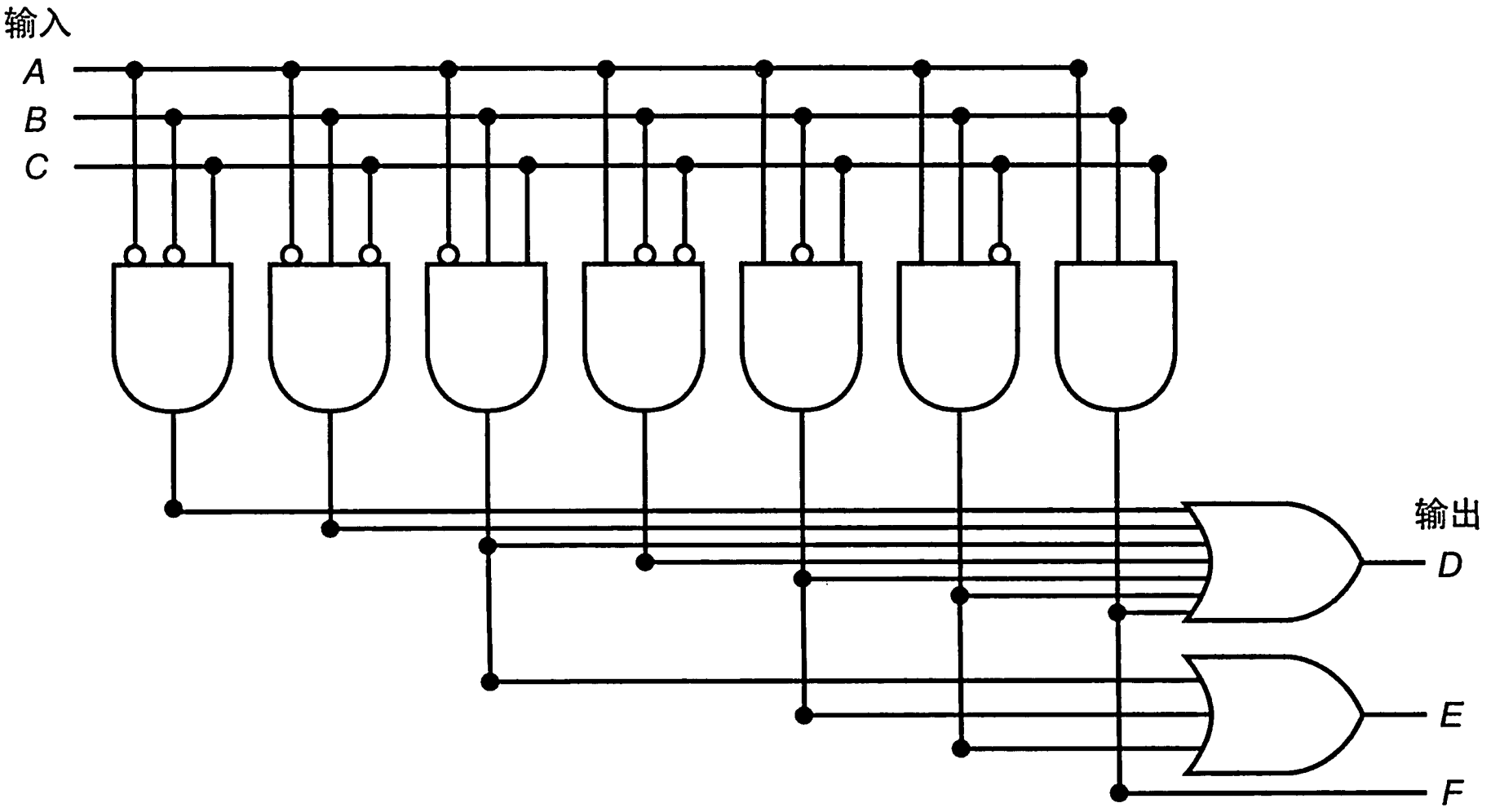


图 A-3-4　例题中描述的逻辑函数的 PLA 实现

与图 A-3-4 所示不同，设计者通常不会绘制所有的门电路，而是指出与门和或门的位置。当需要相应的与门或者或门时，在乘积项信号线和输入 / 输出线的交叉点上用点标示。图 A-3-5 即为图 A-3-4 使用该方法绘制的 PLA 简化图。PLA 的内容在其被创建时就固定了，但也有类似 PLA 的结构，称为 PAL，当设计者准备使用时，可以通过电子方式编程。

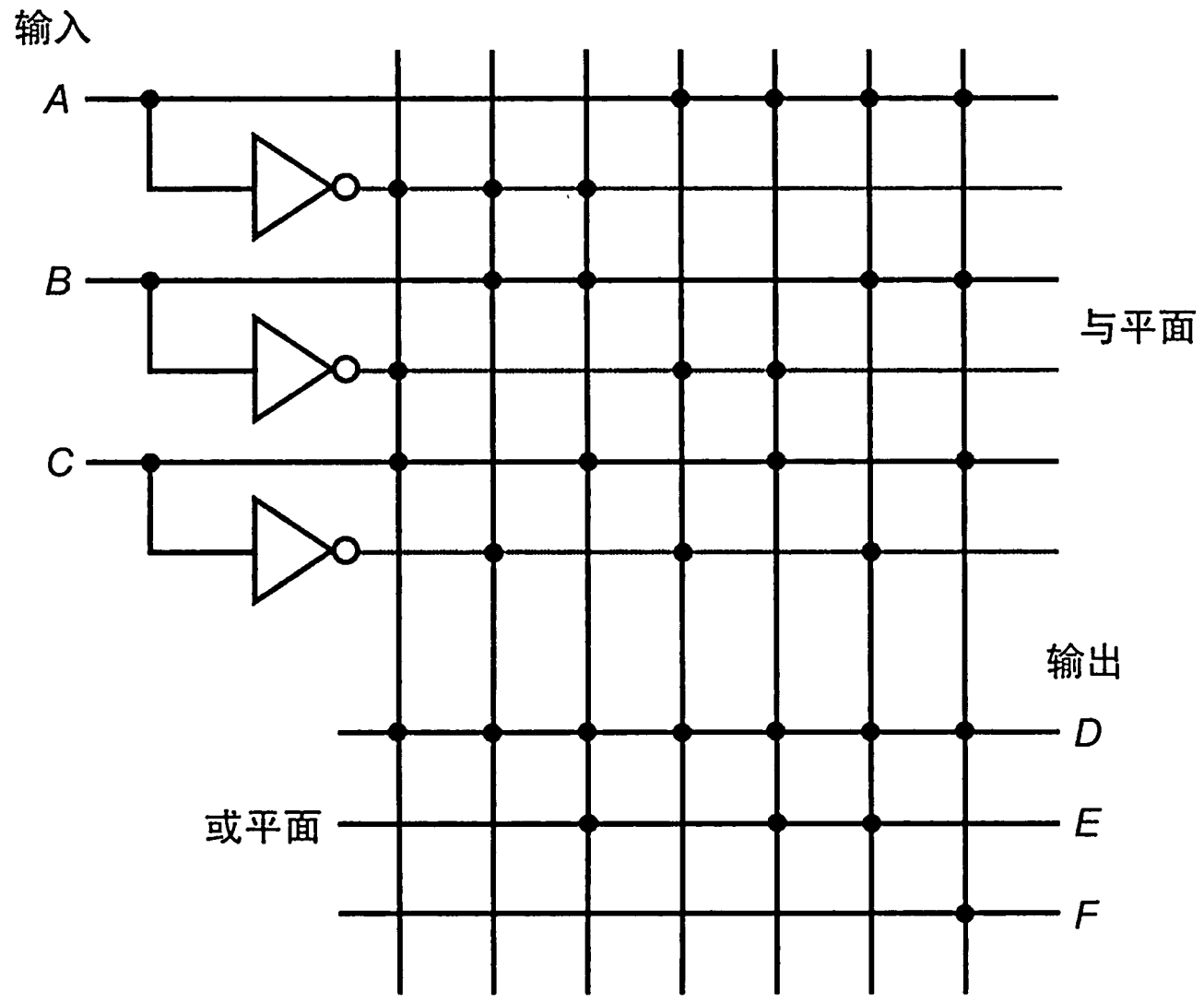


图 A-3-5 在矩阵中用点标示乘积项及其逻辑和的 PLA。通常所有输入都以真或补的形式连接到与平面上，而不是对门电路使用反相器。在与平面上的点表示该输入或其相反值在该乘积项中存在。在或平面上的点表示相应的乘积项出现在相应的输出上

A.3.4 ROM

另一种可用于实现一组逻辑函数的结构化逻辑形式是**只读存储器**（Read-Only Memory，ROM）。ROM 包含一组可以被读取的位置，因此称为存储器；但是，这些位置的内容在制造 ROM 时就固定了。也有**可编程只读存储器**（Programmable ROM，PROM），设计者可以通过电子编程写入内容。还有可擦除的可编程只读存储器，这类设备需要使用紫外线才能缓慢擦除，因此除了设计和调试过程中用到外，它们也只用作只读存储器。

只读存储器：存储内容在制造时就固定的存储器，此后内容仅可读。ROM 作为结构化逻辑，可以将逻辑函数中的项作为地址输入，而输出存储器中对应位的字，以此实现一组逻辑函数。

ROM 具有一组输入地址线和一组输出。ROM 中可寻址单元的数量决定了地址线的数量：如果 ROM 包含 2^m 个可寻址单元（称为高度），则需要 m 条输入线。每个可寻址单元的位数等于输出位数，有时称为 ROM 的宽度。ROM 的总位数等于高度乘以宽度。通常 ROM 的高度和宽度决定了 ROM 的形状。

可编程只读存储器：只读存储器的一种，如果知道其内容，就可对其进行编程。

ROM 可以根据真值表对逻辑函数集进行直接编码。例如，如果有 n 个具有 m 个输入的函数，则需要一个具有 m 条地址线（2^m 个单元，每个单元为 n 位宽）的 ROM。真值表输入部分的表项表示 ROM 单元的地址，而输出部分的内容构成 ROM 的内容。如果真值表输入部分的表项序列被组织成二进制数序列（正如到目前为止所示的所有真值表），则输出部分也按顺序给定 ROM 内容。在之前关于 PLA 的例题中，有 3 个输入和 3 个输出。因此 ROM 具有 $2^3 = 8$ 个入口地址，每个 3 位宽。这些入口的内容按地址递增的顺序直接由例题中真值表的输出部分给出。

ROM 和 PLA 密不可分。ROM 是完全译码的，包含每个可能的输入组合的完整输出字，而 PLA 仅部分解码，这意味着 ROM 始终包含更多入口。对于上道例题中出现的真值表，

ROM 包含所有 8 个可能的输入，而 PLA 仅包含 7 个有效的乘积项。随着输入数量的增加，ROM 的输入单元数将呈指数增长。相反，对于大多数实际逻辑函数，乘积项的数量增长要慢得多（参见附录 C 中的例题）。这种差异使得 PLA 用于实现组合逻辑函数更加有效。ROM 的优势则在于能够实现输入和输出数量相匹配的任何逻辑函数。这个优势使得当逻辑函数发生变化时，ROM 大小无须改变，只需随之改变内容即可。

除 ROM 和 PLA 外，现代逻辑合成系统还将小块组合逻辑转化为可以自动布局和布线的门电路集合。尽管一些小规模门电路集合通常空间利用率较低，但对于简单逻辑函数，它们比 ROM 和 PLA 的严格结构开销更低，因此更具优势。

对于定制或半定制集成电路逻辑设计，常见的选择是现场可编程设备；我们将在 A.12 节中描述该设备。

A.3.5 无关项

通常在实现某些组合逻辑时，有些情况下我们不关心某些输出的值是什么，其原因要么是另一个输出为真，要么是输入组合的子集决定了输出的值。我们称这种情况为*无关项*。由于无关项使得逻辑函数的优化实现更加简单，因此至关重要。

无关项有两种类型：输出无关项和输入无关项，两者都能在真值表中体现出来。当不关心某些输入组合的输出值时，就产生了*输出无关项*。它们在真值表的输出部分表示为 X。当输出对于某些输入组合来说是无关项时，设计者或逻辑优化程序可以自由地对这些输入组合输出真或假。当输出仅依赖于某些输入时，就产生了*输入无关项*，它们在真值表的输入部分也记为 X。

例题 | 无关项

考虑一个输入为 A、B、C 的逻辑函数，其定义如下：

- 如果 A 或 C 为真，无论 B 为何值，输出 D 恒为真。
- 如果 A 或 B 为真，无论 C 为何值，输出 E 恒为真。
- 如果仅有一个输入为真，输出 F 为真。当 D 和 E 同时为真时，F 为无关项。

写出这个逻辑函数完整的真值表和带有无关项的真值表。对于每个真值表，PLA 各有多少个乘积项呢？

答案 | 以下是不考虑无关项的完整的真值表：

输入			输出		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

这个真值表需要 7 个未经优化的乘积项。带有输出无关项的真值表如下：

A			B		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

如果再加入输入无关项，真值表可以被进一步简化为如下形式：

A			B		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

简化后的真值表对应的 PLA 只需要 4 个小项，或者可以用 1 个双输入与门和 3 个或门（2 个有 3 个输入，1 个有 2 个输入）实现。而原始真值表具有 7 个小项且需要 4 个与门。

逻辑最简化对于有效实现逻辑电路至关重要。卡诺图（Karnaugh map）是手工实现任意逻辑最简式的有效工具。卡诺图以图形方式表示真值表，因此很容易看出可以进行组合的乘积项。然而，由于卡诺图的大小及其复杂性，用来手动优化大型逻辑函数是不现实的。幸运的是，逻辑简化过程是高度机械化的，且可以通过设计工具来现实。在最简化过程中，设计工具利用了无关项的特点，因此无关项的识别非常重要。附录末尾的参考文献进一步讨论了逻辑最简化、卡诺图和最简化算法的相关理论。

A.3.6 逻辑单元阵列

由于许多组合操作进行数据处理时不得不对整个数据字（64 位）进行处理。因此，常常构建一个逻辑单元阵列，可以简单地通过给定操作作用于整个输入集合来实现。在机器内部，大多数时候都需要在一对总线之间进行选择。**总线**（bus）是数据线的集合，这些数据线一起被视为单个逻辑信号。（术语总线也用于指示具有多信号源和多设备共享的线路集合。）

总线： 在逻辑设计中的一组数据线集合，它在整体上也可被视为单个逻辑信号；此外，有多个来源共享数据线的集合，也可被共享使用。

例如，在 RISC-V 指令系统中，写入寄存器的指令的结果可以来自两个源中的一个。多选器用于选择两条总线（每个 64 位宽）的哪一条将写入结果寄存器。如果是前面提到的 1 位多选器，则需要重复 64 次（才能将结果写入）。

在图中，我们用加粗的线表示信号量是总线而不是单个的 1 位信号线。大多数总线都是 64 位宽，否则宽度会加以明确标示。当我们指出一个逻辑单元的输入和输出是总线时，意味着该逻辑单元必须反复多次以适应输入线的宽度。图 A-3-6 展示了如何绘制一个多选器，如何在一对 64 位总线之间进行选择，以及如何扩展 1 位宽多选器。有时，我们需要构造一个逻辑单元阵列，该阵列中某些单元的输入是前面单元的输出。例如，这就是多位宽 ALU 的