这些方程仅表示一般形式：如果某个前面的加法器生成进位且所有中间的加法器都传播进位，则 CarryIn*i* 为 1。图 A-6-1 使用管道来解释超前进位。

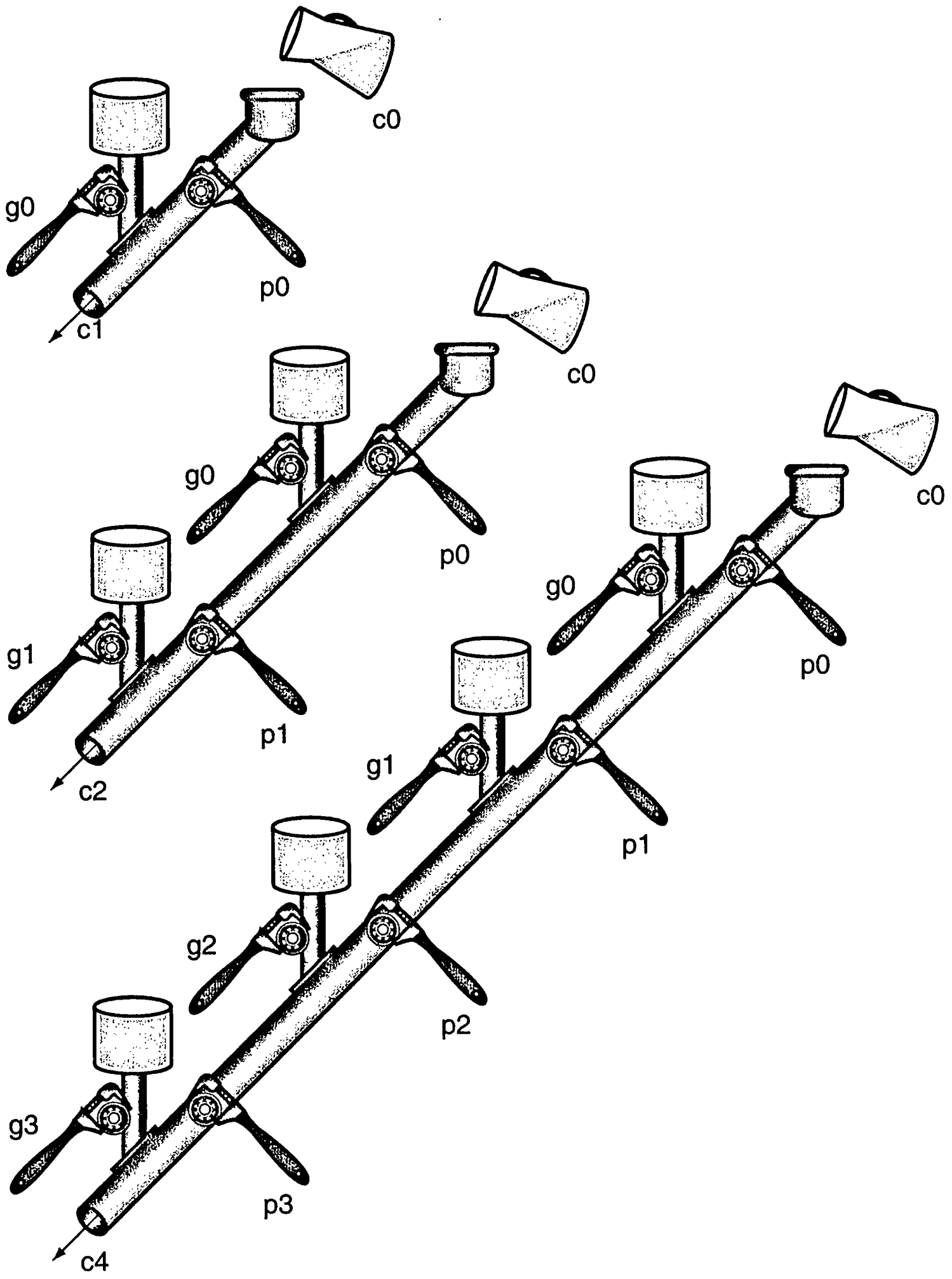

图 A-6-1　管道系统类比，使用水管和阀门类比 1 位、2 位和 4 位超前进位。转动扳手打开和关闭阀门。水用灰色标出。如果最近的（进位）生成值（g*i*）打开，或者第 *i* 个（进位）传播值（p*i*）打开且上游有水，或者有先前的（进位）生成或后面（进位）传播来的水，则管道（c_{i+1}）的输出将是满的。进位输入（c0）可以导致进位，即使没有任何（进位）生成的帮助，但是需要所有（进位）传播的帮助

即使是这种简化形式也会产生长长的方程式，因此即使是 16 位加法器也是如此。接下来尝试使用两级抽象。

A.6.3　使用第二级抽象的快速进位

首先，将这个 4 位加法器与其超前进位逻辑视为单个构建块。如果用行波进位机制将其连接得到一个 16 位加法器，只需要增加少量硬件，就可以使加法操作比原来更快。

为了更快，超前进位需要在更高层次上进行。为了实现 4 位加法器的超前进位，传播和生成信号也需要在这个更高层次上进行。以下是四个 4 位加法器块：

$$P0 = p3 \cdot p2 \cdot p1 \cdot p0$$
$$P1 = p7 \cdot p6 \cdot p5 \cdot p4$$
$$P2 = p11 \cdot p10 \cdot p9 \cdot p8$$
$$P3 = p15 \cdot p14 \cdot p13 \cdot p12$$

也就是说，只有当这一组中每一位都传播进位时，用于 4 位抽象（P*i*）的“超级”传播信号（P*i*）才为真。

对于“超级”生成信号（G*i*），只关心 4 位组的最高有效位是否有进位。如果最高有效位的（进位）生成为真，则上述情况显然会发生；如果之前的（进位）生成为真且所有中间传播（包括最高有效位的传播）也都为真，上述情况也会发生：

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$
$$G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$
$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$
$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$

图 A-6-2 更新了管道类比，以描述 P0 和 G0。

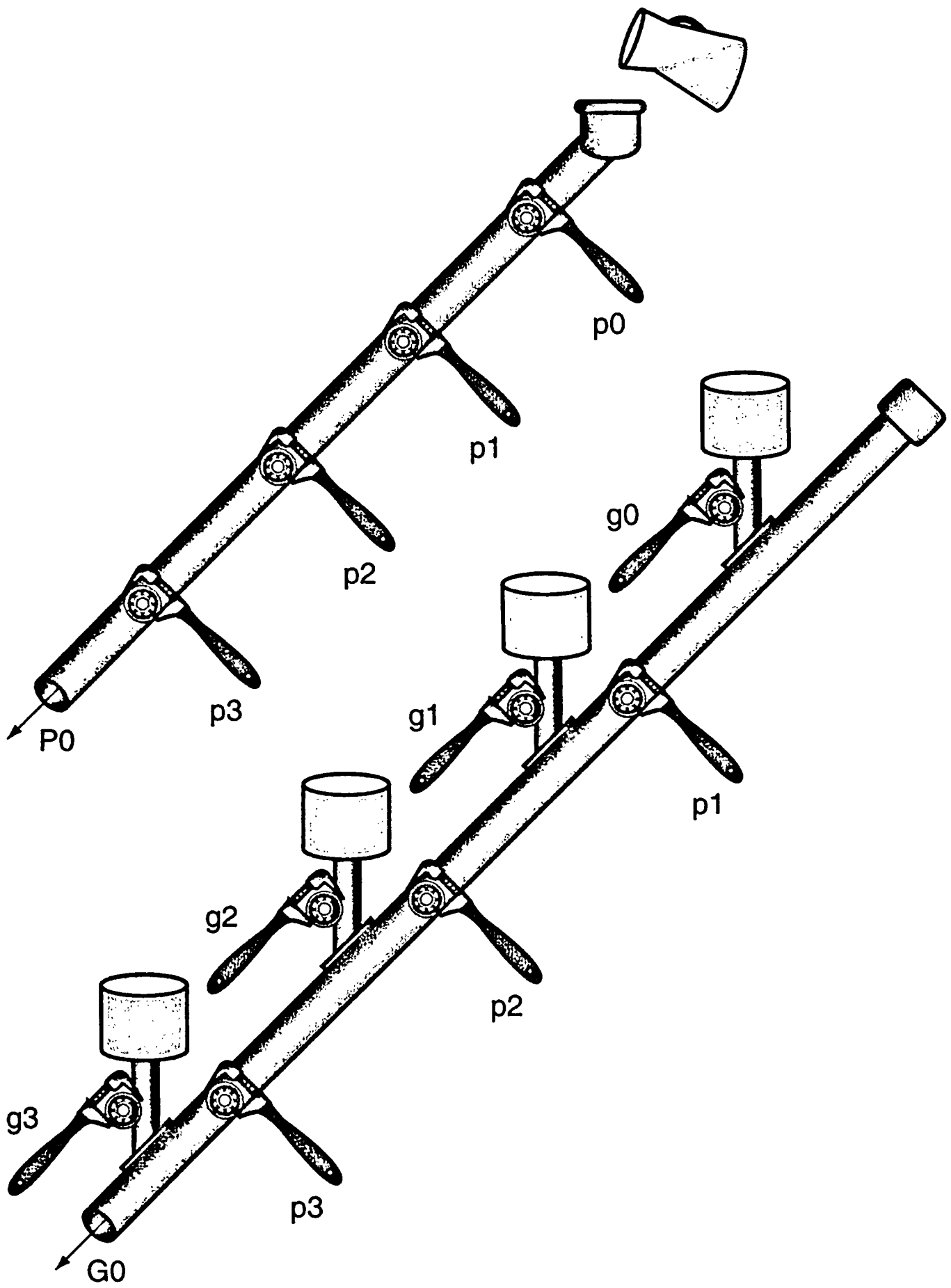

图 A-6-2　管道系统类比下一级超前进位信号 P0 和 G0。如果 4 个（进位）传播（p*i*）打开，则 P0 打开，仅当至少 1 个（进位）生成（g*i*）打开且下游所有（进位）传播都打开，G0 才有水流

对于 16 位加法器的每个 4 位组（图 A-6-3 中的 C1、C2、C3、C4）的进位，这个更高抽象层次的方程式类似于上述 4 位加法器每一位（c1、c2、c3、c4）的进位方程式：

$$C1 = G0 + (P0 \cdot c0)$$

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

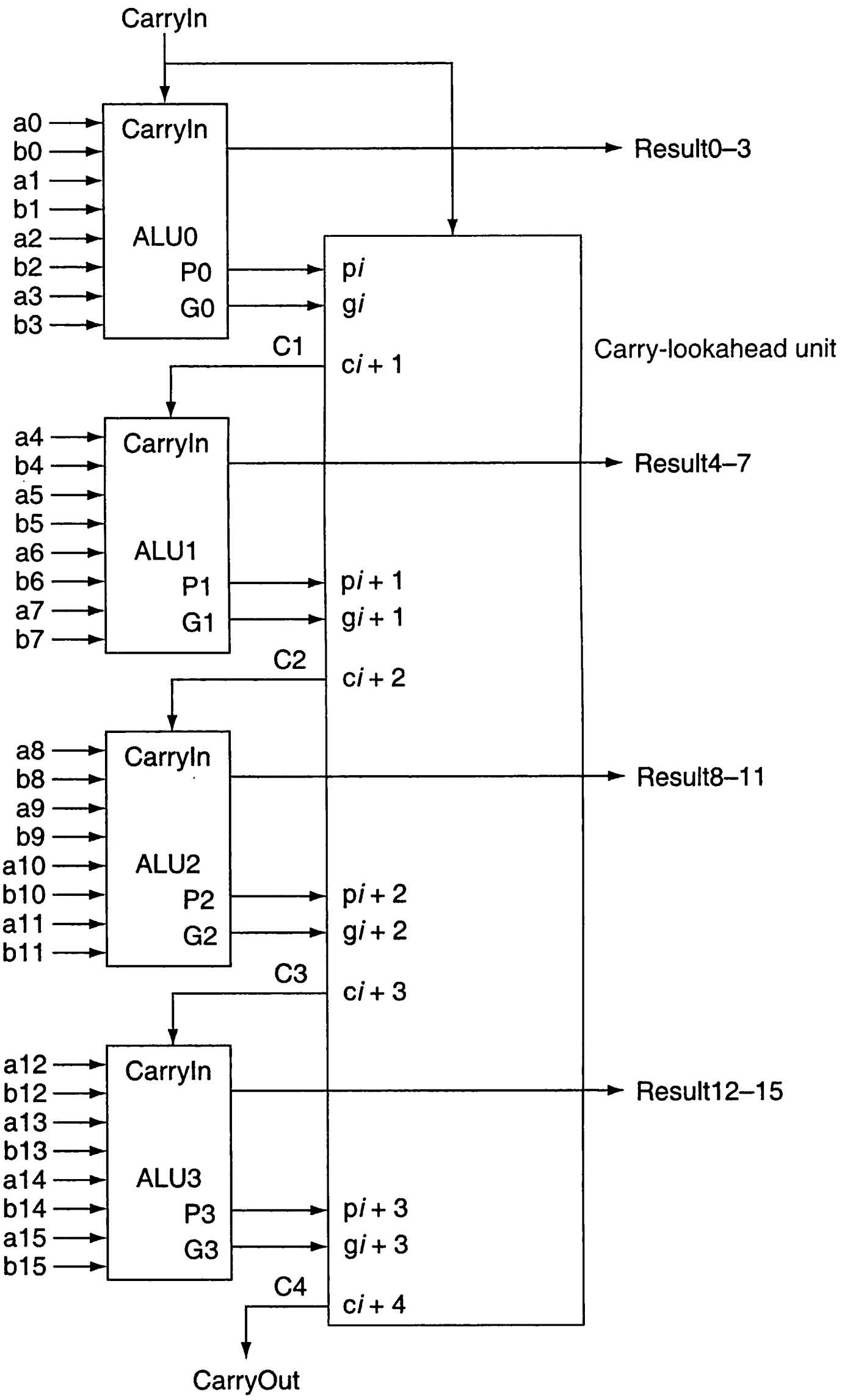

图 A-6-3　4 个 4 位超前进位 ALU 组成的 1 个 16 位加法器。注意进位来源于超前进位单元，而不是 4 位 ALU

例题 | 两级传播和生成

确定两个 16 位数的 g*i*、p*i*、P*i* 和 G*i* 值：

```
a:      0001  1010  0011  0011₂
b:      1110  0101  1110  1011₂
```

CarryOut15（C4）的值是什么？

答案 | 位对齐以便看出（进位）生成值 g*i*（a*i* · b*i*）和（进位）传播值（a*i*+b*i*）：

```
a:      0001  1010  0011  0011
b:      1110  0101  1110  1011
gi:     0000  0000  0010  0011
pi:     1111  1111  1111  1011
```

从左到右依次标记为 15 ～ 0，“超级”（进位）传播（P3、P2、P1、P0）是低级（进位）传播的简单相与。

$$P3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$

“超级”（进位）生成更复杂，因为使用以下等式：

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0$$

$$G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1$$

$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0$$

$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0$$

最后，CarryOut15（C4）为

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0)$$
$$+ (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$
$$= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0)$$
$$= 0 + 0 + 1 + 0 + 0 = 1$$

因此，当这两个 16 位数相加时有一个进位输出。

超前进位之所以进位更快，是因为所有逻辑在时钟周期开始的同时开始进行计算，且一旦每个门的输出停止变化，结果也不再改变。通过用较少的门来发送进位信号，门的输出能更快地停止变化，因此加法器用时更短。

为了理解超前进位的重要性，需要计算它与行波进位加法器的相对性能。

例题 | 行波进位和超前进位的速度比较

为逻辑时间建模的一种简单方法是假设信号通过每个与门、或门时花费的时间相同。通

过简单计算通过逻辑通路的门的数量来估计时间。比较两个 16 位加法器通路的门延迟数，一个使用行波进位，另一个使用两级超前进位。

答案 图 A-5-5 展示了每一位进位输出信号需要两个门延迟。最低有效位的进位输入与最高有效位进位输出之间的门延迟数是 $32 \times 2 = 64$。

对于超前进位，最高有效位的进位输出只有例题中定义的 C4。根据 P*i* 和 G*i*（几个与项的或）需要两级逻辑来定义 C4。用 p*i* 在一级逻辑（与）中定义 P*i*，且用 p*i* 和 g*i* 在两个逻辑层中指定 G*i*，因此下一级抽象的最坏情况也就两级逻辑。p*i* 和 g*i* 都是用 a*i* 和 b*i* 定义的一级逻辑。假设这些方程式中每级逻辑都有一个门延迟，则最坏情况下有 2 + 2 + 1 = 5 个门延迟。

因此，对于从进位输入到进位输出的通路，使用这种简单的硬件速度估算（门延迟），16 位加法超前进位的速度要比行波进位快 6 倍。

A.6.4　总结

超前进位提供比行波进位更快的通路，不需要像行波进位那样等待经过 32 个 1 位加法器的进位。这个快速通路的两个主要信号是（进位）生成和（进位）传播。

前者是不考虑进位输入的进位，后者传递进位。超前进位提供了另一个示例，即抽象对于应对计算机设计的复杂性有多重要。

详细阐述 除一个算术和逻辑操作外，现在已经包含了核心 RISC-V 指令系统中的所有指令：图 A-5-14 中的 ALU 忽略了对移位指令的支持。可以加宽 ALU 多选器来支持左移 1 位或右移 1 位。但硬件设计者设计了一种称为桶式移位器（barrel shifter）的电路，它可以从 1 位移到 63 位，且不超过两个 64 位数相加的时间，因此通常在 ALU 外部进行移位。

详细阐述 通过使用比与门、或门更强大的门，可以更简单地表示 A.5.1 节全加器和的输出逻辑方程式。当两个操作数不同时，异或门为真，即，

$$x \neq y \Rightarrow 1 \text{ 且 } x == y \Rightarrow 0$$

在某些技术中，异或比两级与门、或门更高效。用符号⊕表示异或，新等式为：

$$\text{Sum} = a \oplus b \oplus \text{CarryIn}$$

同时，我们使用门这种传统方式来绘制 ALU。当今的计算机设计使用 CMOS 晶体管作为基本开关电路。CMOS ALU 和桶式移位器利用这些开关的优势，比我们的设计中使用的多选器少得多，但设计原理是类似的。

详细阐述 当使用超过两级的逻辑层次时，使用大小写来区分（进位）生成和（进位）传播信号的层次就不行了。用 g*i*...*j* 和 p*i*...*j* 表示从第 *i* 位到第 *j* 位的生成和传播信号。因此，g1...1 表示位 1 的（进位）生成，g4...1 表示位 4 到位 1 的（进位）生成，g16...1 表示位 16 到位 1 的（进位）生成。

自我检测 使用上述门延迟来简单估算硬件速度，行波进位的 8 位加法相比于超前进位逻辑的 64 位加法的相对性能如何？

1. 64 位超前进位加法器的速度提高了 3 倍：8 位加有 16 个门延迟，64 位加有 7 个门延迟。
2. 速度相当，因为 64 位加法需要 16 位加法器有更多层逻辑。
3. 8 位加也比 64 位更快，即使有超前进位。

A.7 时钟

在讨论存储元件和时序逻辑之前，简要地讨论一下时钟是十分有益的。类似于 4.2 节中的讨论，本节介绍时钟的相关内容。有关时钟和时序的更多细节，请参见 A.11 节。

边沿触发时钟：一种时钟方案，其中所有状态改变都发生在时钟边沿。

时序逻辑中需要时钟来决定何时更新存储元件的状态。时钟只是一个具有固定周期的不停运转的信号，时钟频率是时钟周期的倒数。如图 A-7-1 所示，时钟周期时间或时钟周期分为两部分：高电平和低电平。在本节中仅采用**边沿触发时钟**的同步逻辑。这意味着所有状态更新都发生在时钟边沿。我们使用边缘触发的方法是因为它更容易理解。但从工艺的角度来讲，很难说它是否就是**时钟同步方法**的最佳选择。

时钟同步方法：根据时钟确定数据何时有效和稳定的方法。

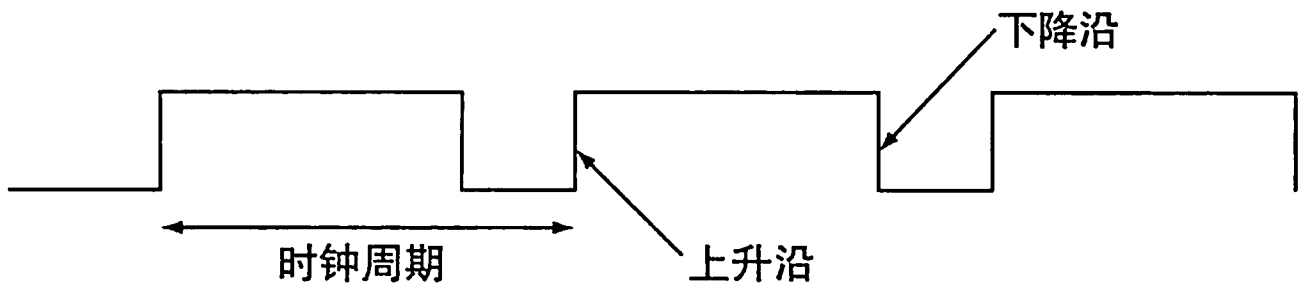

图 A-7-1 时钟信号在高电平和低电平之间振荡。时钟周期是一个完整周期的时间。在边沿触发设计中，时钟的上升沿或下降沿是有效信号并导致状态发生变化

在边沿触发的方法中，时钟的上升沿或下降沿是有效信号并导致状态发生变化。正如将在下一节中所看到的，边缘触发设计中的**状态单元**的改变仅发生在有效时钟的边沿。至于选择哪个时钟边沿作为有效触发信号是受设计技术影响的，但不影响设计逻辑所涉及的概念。

状态单元：一个存储元件。

时钟边沿用作采样信号，使得输入到状态单元的数据值被采样并存储在状态单元中。使用边沿触发意味着采样过程基本上是瞬时的，消除了采样信号时刻不同可能引发的问题。

同步系统：一种采用时钟的存储系统，只有当时钟指示信号值稳定时才读取数据信号。

时钟系统（也称为**同步系统**）的主要约束是，当有效时钟边沿发生时，写入状态单元的信号必须有效。如果信号稳定（即不改变），则该信号有效，并且在输入改变之前该值不会再次改变。由于组合电路无法实现反馈，只要组合逻辑单元的输入不变，输出最终将变为有效。

图 A-7-2 显示了同步时序逻辑设计中状态单元和组合逻辑结构之间的关系。状态单元的输出仅在时钟边沿时刻改变，该输出为组合逻辑提供有效输入。为确保在有效时钟边沿写入的状态单元的值有效，时钟必须具有足够长的周期，从而让组合逻辑中的所有信号稳定，然后在时钟边沿对这些值进行采样以便存储在状态单元。这个约束为时钟周期设置了一个下限，该下限必须足够长，从而保证所有状态单元的输入有效。

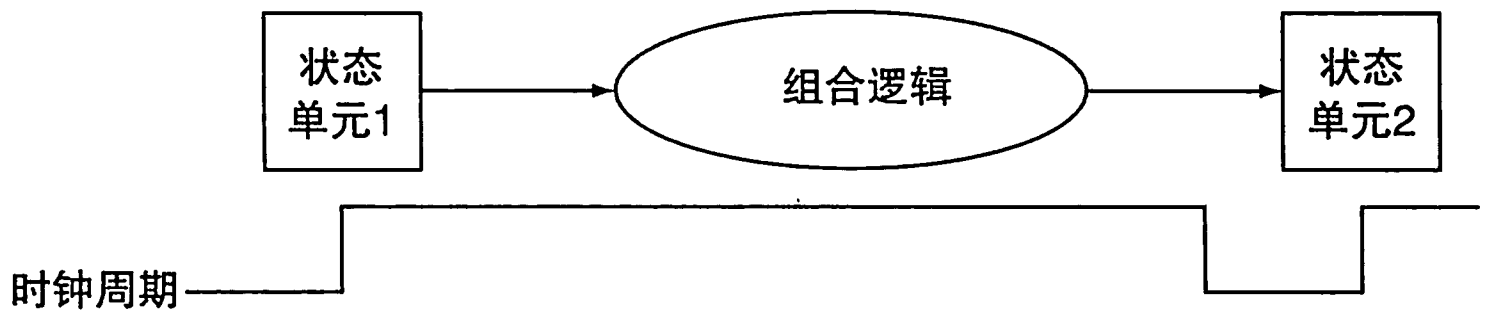

图 A-7-2 组合逻辑的输入来自状态单元，其输出也被写到一个状态单元。时钟边沿决定何时更新状态单元的内容

在本附录的其余部分以及第 4 章中，通常省略时钟信号，因为我们假设所有状态单元都在同一时钟边沿上更新。某些状态单元在每个时钟边沿写入，而其他状态单元仅在某些条件下写入（例如更新寄存器）。在这种情况下，我们将为该状态单元提供显式写信号。该写信号必须和时钟同步，确保在写信号有效时，仅在时钟边沿进行更新。我们将在下一节中看到上述设计是如何实现和使用的。

边沿触发方法的另一个优点是可以使用一个状态单元作为同一组合逻辑的输入和输出，如图 A-7-3 所示。在实际设计中，必须注意防止这种情况下的竞争，并确保时钟周期足够长，A.11 节将进一步讨论该主题。

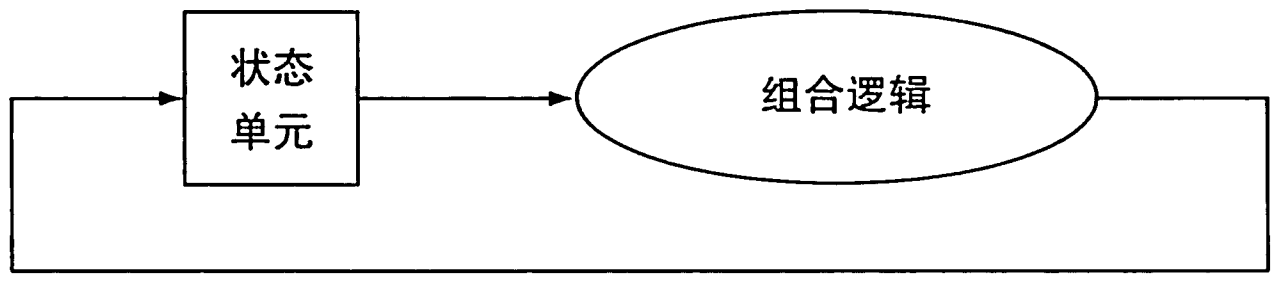

图 A-7-3　边沿触发方法允许在同一时钟周期内读写状态单元，从而不会因产生竞争而导致数据值的不确定。当然，时钟周期必须足够长，以保证在有效时钟边沿到来之前输入值稳定

详细阐述　有时，设计人员发现让一小部分状态单元与大多数状态单元发生改变的时钟边沿相反会很有用。这样做需要非常小心，因为这种方法对状态单元的输入和输出都有影响。那么为什么设计人员会这样做呢？考虑这样一种情况：状态单元之前和之后的组合逻辑的数量足够小，因此每个元件可以在半个时钟周期内执行完，而不是更常见的整个时钟周期。然后，因为输入和输出都将在半个时钟周期之后可用，状态单元可以在对应于半个时钟周期的时钟边沿进行写入。使用这种技术的一个常见实例是寄存器堆，简单地读或写寄存器堆通常可以在正常时钟周期的一半时间内完成。第 4 章利用这种思想来减少流水线开销。

寄存器堆：一个状态单元，由一组寄存器组成，可通过提供要访问的寄存器号来进行读写。

A.8　存储元件：触发器、锁存器和寄存器

在本节和下一节中，我们将讨论存储元件的基本原理。从触发器和锁存器开始，然后是寄存器堆，最后是存储器。所有的存储元件都存储状态：任何存储元件的输出都取决于输入和存储在存储元件内的值。因此，包含存储元件的所有逻辑块都包含状态并且是时序可控的。

最简单的存储元件是无时钟的，也即是说，它们没有任何时钟信号输入。虽然我们在本章中仅使用带时钟的存储元件，但是无时钟的锁存器是最简单的存储元件，所以我们先看看这个电路。图 A-8-1 显示了一个 S-R 锁存器（置位 – 复位锁存器），它由一对 NOR（或非）门（具有反相输出的或门）构成。输出 Q 和 $\overline{Q}$ 表示存储状态及其反相值。当 S 和 R 都没有被设为有效时，交叉耦合的或非门用作反相器并存储 Q 和 $\overline{Q}$ 先前的值。

例如，如果输出 Q 为真，那么下方的反相器生成假输出（即 $\overline{Q}$），它成为上方反相器的输入，生成一个真输出，即 Q，依此类推。如果 S 被置为有效，那么输出 Q 为真并且 $\overline{Q}$ 为假，而如果 R 被置为有效，则输出 $\overline{Q}$ 为真且 Q 为假。当 S 和 R 都被置为无效时，Q 和 $\overline{Q}$ 的最后的值将继续存储在交叉耦合结构中。同时将 S 和 R 置为有效会导致错误操作：这取决于 S 和 R 如何被置为无效，锁存器可能会振荡或变为亚稳态（这部分将在 A.11 节中详细介绍）。

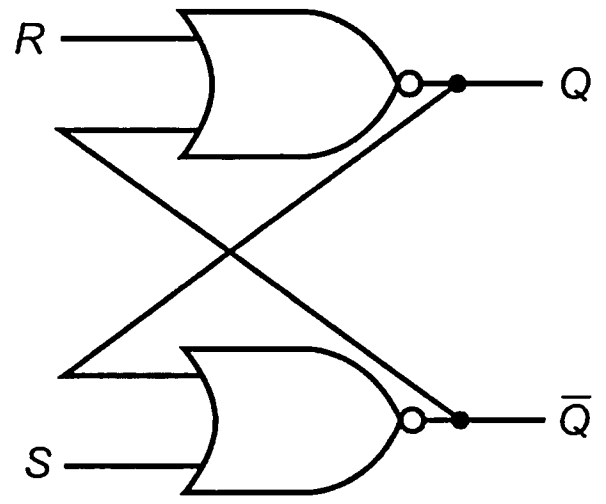

图 A-8-1 一对交叉耦合的 NOR(或非门) 可以存储数据。存储在输出上的值 Q 取反得到 $\overline{Q}$，然后 Q 取反得到 Q，不断循环。如果 R 或 Q 变为有效，则 Q 将被置为无效，反之亦然

这种交叉耦合结构是更复杂的存储元件的基础，这些复杂元件可以用于存储数据信号。这些元件还包含额外的门电路，用于存储信号值和仅与时钟配合的状态更新。下一节将介绍如何构建这些存储元件。

A.8.1 触发器和锁存器

触发器和**锁存器**是最简单的存储元件。在触发器和锁存器中，输出等于元件内存储状态的值。此外，与上述 S-R 锁存器不同，接下来使用的所有锁存器和触发器都是带时钟的，这意味着它们具有时钟输入，并且状态的改变由该时钟触发。触发器和锁存器之间的区别是触发状态实际改变的时钟位置不同。在带时钟的锁存器中，只要时钟信号有效，若输入改变，状态就会随之改变。而在触发器中，状态仅在时钟边沿上改变。从本节开始，我们使用边沿触发的时序控制方法，其中，状态仅在时钟边沿更新，因此使用触发器。触发器通常由锁存器构建，因此我们首先描述一个简单的带时钟的锁存器的一些操作，然后再讨论由该锁存器构成的触发器。

触发器：一种存储元件，其输出等于元件内存储状态的值，并且内部状态仅在时钟边沿上改变。

锁存器：一种存储元件，其输出等于元件内存储状态的值，并且当时钟有效时，只要有适当的输入变化，状态就会改变。

对于计算机应用，触发器和锁存器的功能是存储信号。D 锁存器或 **D 触发器**将其数据输入信号的值存储在内部存储中。虽然还有许多其他类型的锁存器和触发器，但 D 型是我们需要的唯一基本逻辑单元。D 锁存器有两个输入和两个输出。输入是要存储的数据值（D）和时钟信号（C），C 控制锁存器应何时读取 D 输入上的值并存储它。输出就是内部状态（Q）及其反相（$\overline{Q}$）的值。当时钟信号 C 有效时，锁存器处于开状态，输出（Q）的值变为输入 D 的值。当时钟信号 C 无效时，锁存器处于关状态，并且输出（Q）的值是上次锁存器打开时存储的值。

D 触发器：具有单个数据输入的触发器，在时钟边沿将输入信号的值存储在内部存储器中。

图 A-8-2 显示了如何通过为交叉耦合的 NOR（或非）门添加两个额外的门电路来实现 D 锁存器。由于当锁存器打开时，Q 的值随着 D 的变化而变化，所以这种结构有时也被称为透明式锁存器。图 A-8-3 显示了这个 D 锁存器如何工作，假设输出 Q 初始值为假且 D 先改变。

如前所述，我们使用触发器而不是锁存器作为基本逻辑单元。触发器是不透明的，它们的输出仅在时钟边沿发生变化。触发器在上升（正）或下降（负）的时钟边沿触发，我们可以选择其中任何一种方式进行设计。图 A-8-4 显示了如何利用一对 D 锁存器构建一个下降沿的 D 触发器。在 D 触发器中，输出在时钟边沿时存储。图 A-8-5 给出了这个触发器的工作原理。

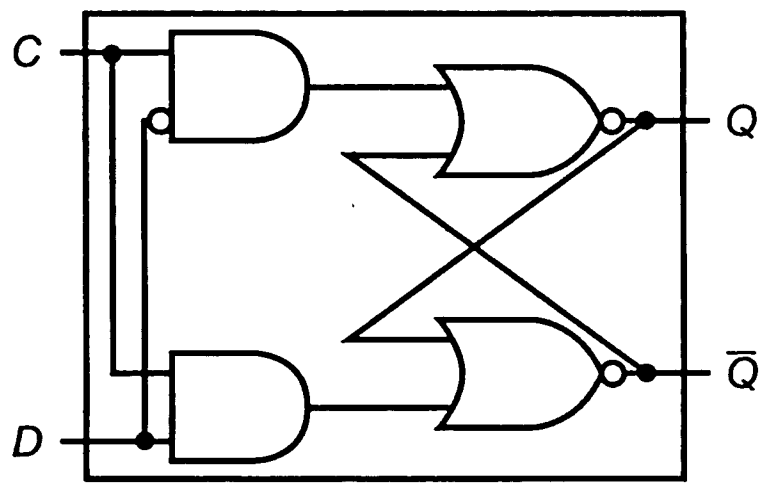

图 A-8-2　利用 NOR（或非）门实现的 D 锁存器。 如果另一个输入为 0，则或非门用作反相器。因此，除非时钟输入 *C* 有效，否则交叉耦合的或非门用于存储状态值。在这种情况下，输入 *D* 的值取代了 *Q* 的值并且被存储。当时钟信号 *C* 从有效变为无效时，输入 *D* 的值必须保持稳定

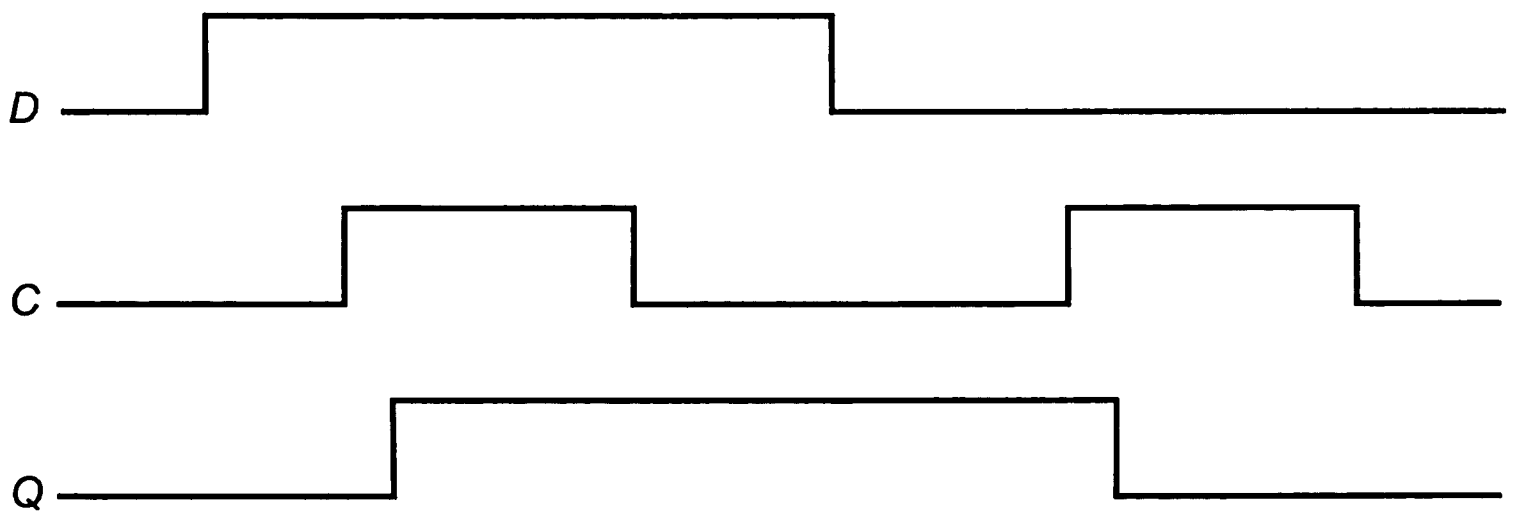

图 A-8-3　D 锁存器的操作，假设输出的初始值无效。当时钟 *C* 有效时，锁存器打开，*Q* 输输出立即变为 *D* 输入的值

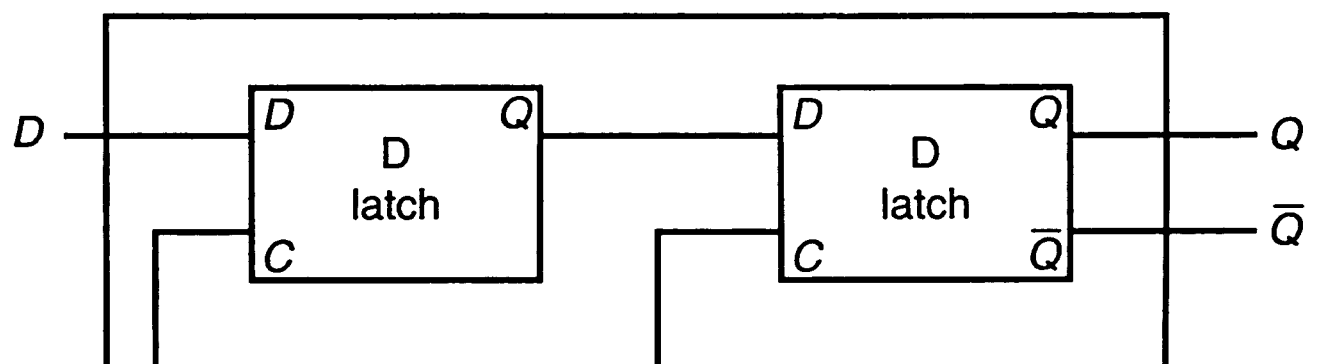

图 A-8-4　下降沿触发的 D 触发器。第一个锁存器（称为主器件）打开，并在时钟输入 *C* 有效时遵循 *D* 的输入。当时钟输入 *C* 下降时，第一个锁存器关闭，但第二个锁存器（称为从器件）打开，并从主锁存器的输出获得其输入

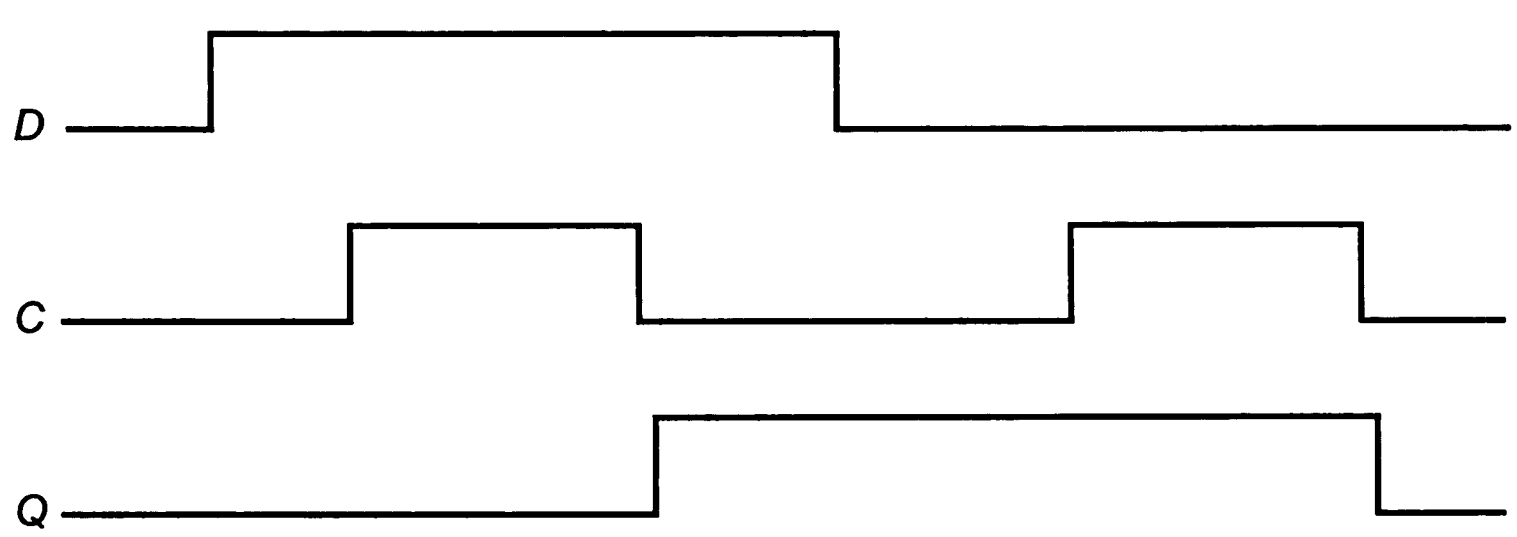

图 A-8-5　下降沿触发的 D 触发器的操作，假设输出的初始值无效。当时钟输入（*C*）从有效变为无效时，*Q* 输出存储 *D* 输入的值。将此行为与图 A-8-3 中所示的时钟 D 锁存器的行为进行比较。在时钟锁存器中，存储值和输出 *Q* 都在 *C* 为高电平时改变，而不是仅在 C 转变时

下面是一段用于上升沿 D 触发器的模块的 Verilog 描述，假设 *C* 是时钟输入而 *D* 是数据输入：

```
module DFF(clock,D,Q,Qbar);
    input clock, D;
    output reg Q;
    output Qbar;
    assign Qbar= ~ Q;
    always @(posedge clock)
       Q=D;
endmodule
```

由于 *D* 输入在时钟边沿采样，因此必须在时钟边沿的之前和之后的一段时间内保持有效。在时钟边沿之前，输入必须保持有效的最短时间称为**建立时间**；在时钟边沿之后，必须保持有效的最短时间称为**保持时间**。因此，如图 A-8-6 所示，任何触发器（或使用触发器构建的任何电路）的输入必须在时间窗口期间有效，该窗口从时钟边沿之前的时间 t_{setup} 开始，并且在时间边沿之后的时间 t_{hold} 结束。A.11 节更详细地讨论了时钟和时序约束，包括触发器的传播延迟。

建立时间：在时钟边沿之前，存储单元的输入必须保持有效的最短时间。

保持时间：在时钟边沿之后，存储单元的输入必须保持有效的最短时间。

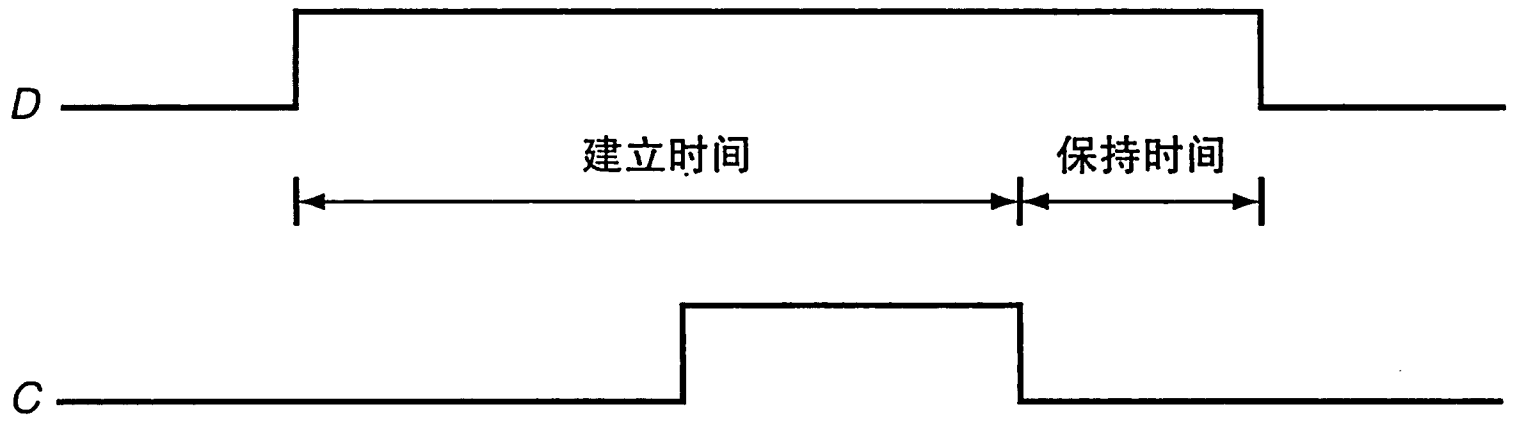

图 A-8-6 下降沿触发的 D 触发器的建立和保持时间要求。输入必须在时钟边沿之前以及时钟边沿之后的一段时间内保持稳定。在时钟边沿之前，信号必须保持稳定的最短时间称为建立时间，而在时钟边沿之后，信号必须保持稳定的最短时间称为保持时间。如果不满足这些最低要求，如 A.11 节所述，可能会导致触发器的输出无法预测。保持时间通常为 0 或非常小，因此无须担心

我们可以使用 D 触发器阵列来构建一个可以保存多位数据（例如字节或字）的寄存器。我们在第 4 章的数据通路中使用了寄存器。

A.8.2 寄存器堆

寄存器堆是一个对数据通路至关重要的结构。它由一组寄存器组成，可通过提供要访问的寄存器号来进行读写。通过对每个读或写端口添加译码器，以及由 D 触发器构建的寄存器阵列，就可以实现寄存器堆。因为读寄存器不会改变任何状态，所以只需要提供一个寄存器号作为输入，唯一的输出就是该寄存器中包含的数据。为了写寄存器，需要三个输入：寄存器号、要写入的数据和控制写寄存器的时钟。在第 4 章中，我们使用了一个寄存器堆，它有两个读端口和一个写端口。该寄存器堆如图 A-8-7 所示。读端口可以用一对多选器实现，每个多选器的宽度与寄存器堆的每个寄存器中的位数一样宽。图 A-8-8 给出了 64 位宽寄存器堆的两个读端口的实现。

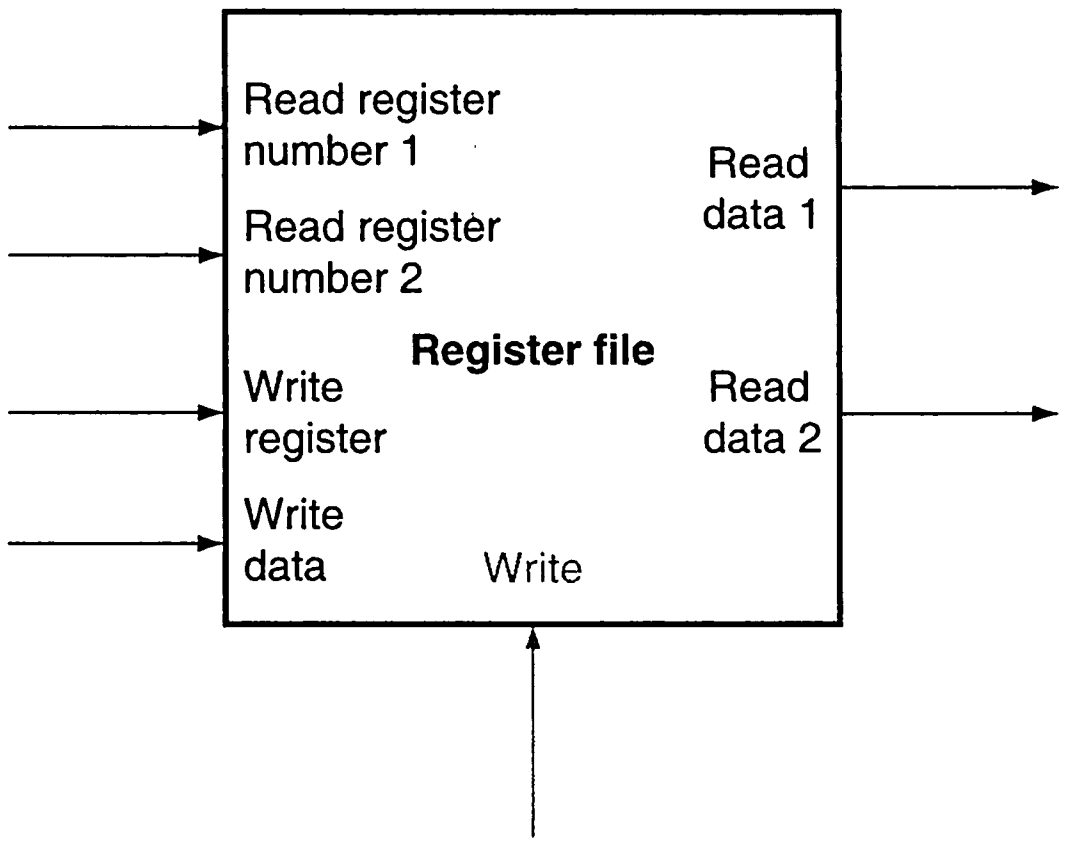

图 A-8-7 具有两个读端口和一个写端口的寄存器堆，有五个输入和两个输出。控制输入 Write 以灰色显示

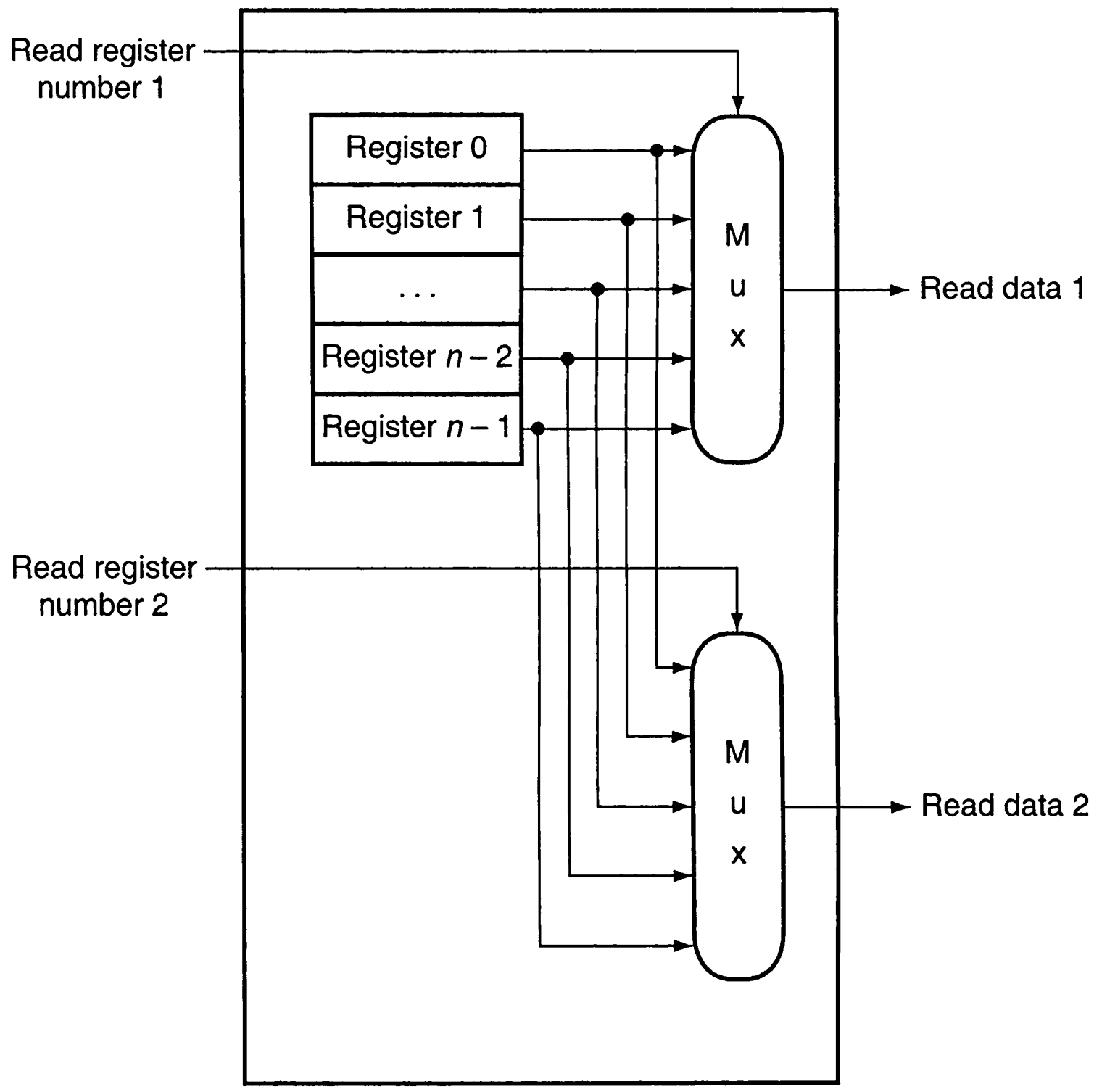

图 A-8-8 具有 n 个寄存器的寄存器堆，两个读端口的实现可以用两个 n 选 1 的多选器完成，每个 64 位宽。寄存器读数字信号用作多选器的选择信号。图 A-8-9 显示了如何实现写端口

实现写端口稍微复杂一些，因为我们只能更改指定寄存器的内容。通过使用译码器生成一个信号，用于确定要写入哪个寄存器，便可以完成此操作。图 A-8-9 给出了如何实现寄存器堆的写端口。需要注意的是要记住触发器仅在时钟边沿上改变状态。在第 4 章中，我们显式地连接了寄存器堆的写信号，并假设图 A-8-9 所示的时钟默认加入了。

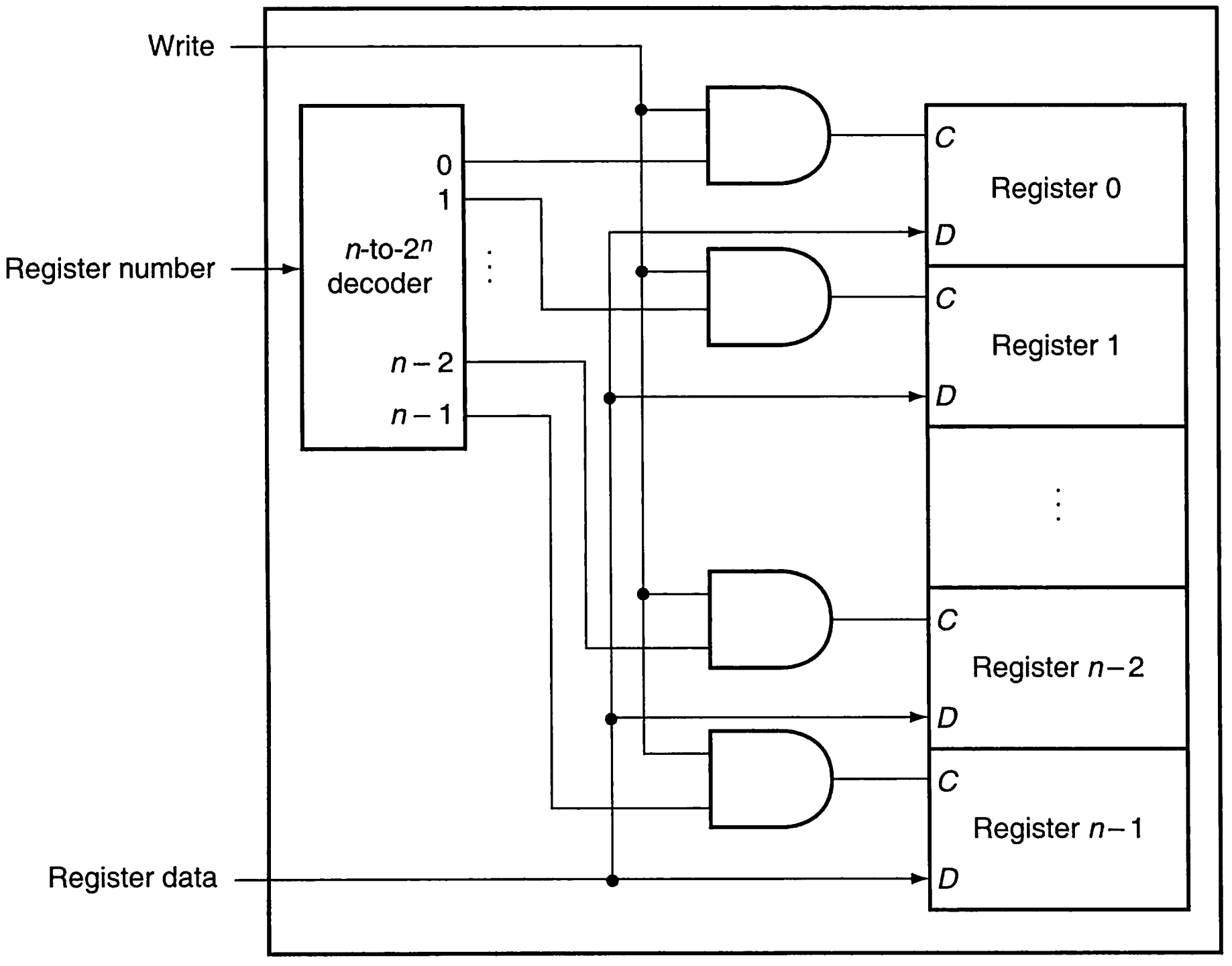

图 A-8-9　寄存器堆的写端口用译码器实现，该译码器与写信号一起使用以产生寄存器的 *C* 输入。所有的三个输入（寄存器号、数据和写信号）都有建立时间和保持时间约束，以确保将正确的数据写入寄存器堆

如果在一个时钟周期内读写相同的寄存器会发生什么？如前面的图 A-7-2 所示，由于寄存器堆的写发生在时钟边沿，寄存器的值在读取期间保持有效，所以返回的值将是在较早的时钟周期中写入的值。如果我们希望读取返回当前正在写入的值，则需要在寄存器堆中或其外部附加逻辑电路。第 4 章广泛使用了这种逻辑电路。

A.8.3　使用 Verilog 描述时序逻辑

要使用 Verilog 描述时序逻辑，我们必须了解如何生成时钟，如何描述何时将值写入寄存器，以及如何指定时序控制。我们从描述时钟开始。时钟不是 Verilog 中的预定义对象，需要在语句之前使用 Verilog 符号 #n 生成时钟。这会导致在执行语句之前延迟 *n* 个模拟时钟步。在大多数 Verilog 模拟器中，还可以生成时钟作为外部输入，允许用户在模拟时指定运行模拟所需的时钟周期数。

图 A-8-10 中的代码实现了一个简单时钟，时钟维持高或低电平一个模拟单元，然后切换状态。我们使用延迟功能和阻塞赋值来实现时钟。

```
reg clock;
always #1 clock = ~clock;
```

图 A-8-10　时钟描述

接下来，我们必须能够指定边沿触发寄存器的操作。在 Verilog 中，这是通过使用 `always` 块上的敏感列表，以及符号 `posedge` 或 `negedge` 指定二进制变量的正边沿或负边沿触发来完成的。因此，下面的 Verilog 代码会令寄存器 `A` 在正边沿时钟写入值 `b`：

在本章和第 4 章的 Verilog 部分中，我们假设都使用正边沿触发设计。图 A-8-11 给出了 RISC-V 寄存器堆的 Verilog 规范，假设有两次读操作和一次写操作，只有写操作需要时钟。

```
    reg [63:0] A;
    wire [63:0] b;

    always @(posedge clock) A <= b;

module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,
Data1,Data2,clock);

   input [5:0] Read1,Read2,WriteReg; // the register numbers
to read or write
   input [63:0] WriteData; // data to write
   input RegWrite, // the write control
     clock; // the clock to trigger write
   output [63:0] Data1, Data2; // the register values read
   reg [63:0] RF [31:0]; // 32 registers each 32 bits long

   assign Data1 = RF[Read1];
   assign Data2 = RF[Read2];

   always begin
      // write the register with new value if Regwrite is
high
      @(posedge clock) if (RegWrite) RF[WriteReg] <=
WriteData;
   end
endmodule
```

图 A-8-11　用行为级 Verilog 描述的 RISC-V 寄存器堆。这个寄存器堆在时钟上升沿写入

自我检测　在图 A-8-11 中寄存器堆的 Verilog 代码中，与正在被读的寄存器对应的输出端口使用的是连续赋值，而正在被写的寄存器在 `always` 块中赋值。以下哪个原因是正确的？

a. 没什么特别原因，只是为了方便。

b. 因为 Data1 和 Data2 是输出端口，而 WriteData 是输入端口。

c. 因为读操作是组合逻辑事件，而写操作是时序逻辑事件。

A.9　存储元件：SRAM 和 DRAM

寄存器和寄存器堆是小型存储器的基本单元，而要组成大型存储器，则需要用到 SRAM（static random access memory，静态随机访问存储器）或 DRAM（动态随机访问存储器）。我们先讨论稍微简单点的 SRAM，然后再讨论 DRAM。

SRAM：一种存储器，其数据是静态存储的（如触发器）而不是动态存储的（如 DRAM）。SRAM 比 DRAM 更快，但存储密度更低且每位的价格更高。

A.9.1　SRAM

SRAM 是一个简单的存储阵列集成电路，其通常只有单个可以提供读或写的访问端口。尽管读写的访问特性通常不同，但 SRAM 对任何数据的访问时间都是固定的。SRAM 芯片根据可寻址空间的大小以及每个可寻址单元的位宽有相应的配置。例如，4M × 8 的 SRAM 可以提供 4M 个表项，每个表项的存储宽度为 8 位。因此，它有 22 条地址线（因为 $4M = 2^{22}$）、8 位的数据输出线和 8 位的单个数据输入线。与 ROM 一样，可寻址范围通常称为高度，每个可寻址单元的位数称为宽度。由于各种技术原因，最新和最快的 SRAM 通常采用“窄”配置：× 1 和 × 4。图 A-9-1 给出了 2M × 16 型 SRAM 的输入和输出信号。

要启动读或写操作，片选（Chip select）信号必须有效。对于读操作，还必须激活输出使

能（Output enable）信号，该信号控制由地址选择的数据是否实际在引脚上驱动。输出使能信号允许将多个存储器连接到单输出总线上，并决定由哪个存储器来驱动总线。SRAM 读数据的访问时间通常被定义为从输出使能信号有效和地址线有效，到数据出现在输出总线上的这段时间。2004 年，拥有最快 CMOS 器件的 SRAM 的读取访问时间大约为 2 ～ 4ns，这些 SRAM 通常容量较小，数据宽度较窄，更大部件的 SRAM 的读取访问时间通常为 8 ～ 20ns。2004 年已经有容量超过 32M 位数据的 SRAM 出现了。在过去 5 年中，消费产品和数码设备对于低功耗 SRAM 的需求大大增长，这些 SRAM 具有更低的待机和访问功率，但通常比普通 SRAM 慢 5 ～ 10 倍。最近，类似于同步 DRAM（下一节讨论）的同步 SRAM 也已经开发出来了。

对于写操作，必须要提供要写的数据和目的地址，以及写控制信号。当写使能（Write enable）信号和片选信号为真时，数据输入线上的数据被写入到地址指定的单元中。和 D 触发器和锁存器一样，对于地址和数据线，有建立时间和保持时间的要求。此外，写使能信号不是时钟边沿，而是具有最小宽度约束的脉冲。完成写操作的时间由建立时间、保持时间和写使能脉冲宽度共同确定。

大容量 SRAM 的构建方式与构建寄存器堆的方式不同，因为对于寄存器堆而言，32-1 多选器是实用的，而对于 64K × 1 SRAM 来讲，使用 64K-1 的多选器就显得不切实际。大容量存储器不使用巨型的多选器，而是使用共享的输出线（称为位线）来实现，存储阵列中的多个存储单元都可以将其置为有效。为了满足多个存储单元驱动信号线，要使用三态缓冲器。三态缓冲器具有两个输入——数据信号和输出使能信号；以及一个输出，它有三种状态——有效、无效或高阻态。如果输出使能信号有效，则三态缓冲器的输出等于输入。否则，输出使能信号无效，输出属于高阻态，由另一个输出使能信号有效的三态缓冲器来决定

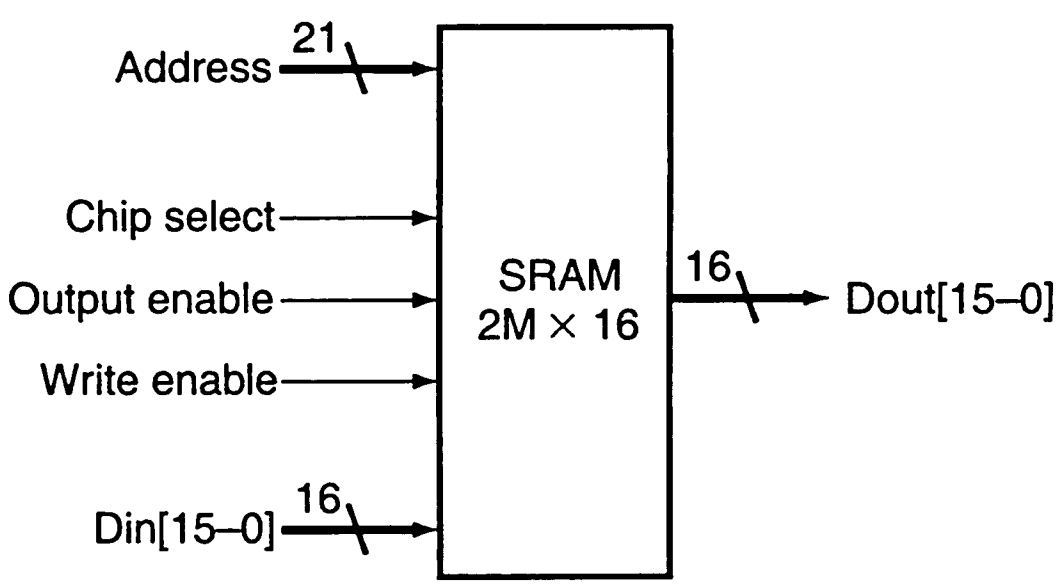

图 A-9-1 一个 32K × 8 的 SRAM，有 21 根地址线（32K=2^{15}）和 16 位数据输入线，3 条控制线以及 16 位数据输出线

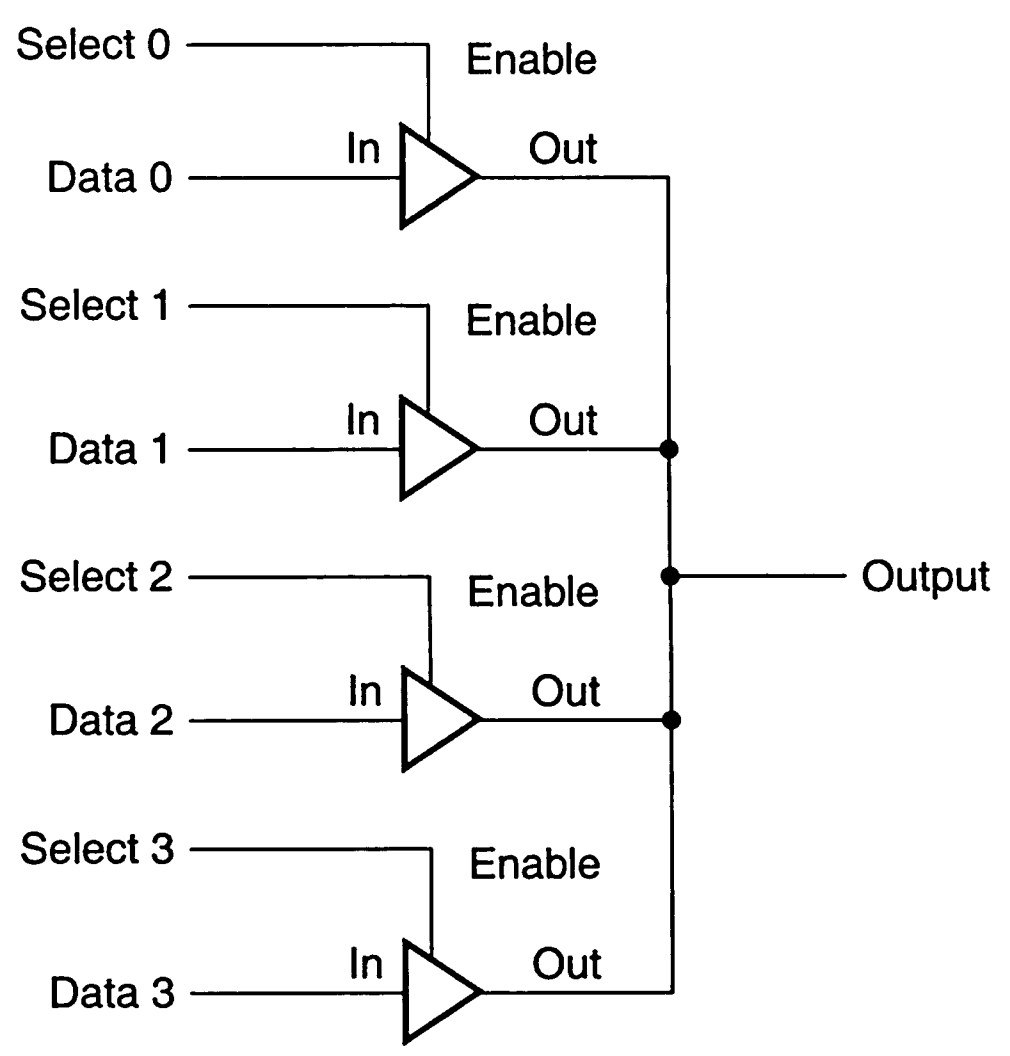

图 A-9-2 利用 4 个三态缓冲器形成多选器。只能识别 4 个可选择输入中的一个。在输出使能无效时，三态缓冲器输出高组态，以允许输出使能有效的其他驱动共享的输出线

共享输出的值。

图 A-9-2 给出了一组三态缓冲器，用于形成具有译码输入的多选器。重要的是，至多只能有一个三态缓冲器的输出使能信号为有效，否则，三态缓冲器会发生竞争输出线的现象。通过在 SRAM 的各个单元中使用三态缓冲器，对应于特定输出的每个单元可以共享相同的输出线。使用一组分布式三态缓冲器比大型集中式的多选器更为有效。将三态缓冲器嵌入触发器，形成了 SRAM 的基本单元。图 A-9-3 显示了如何构建一个小的 4 × 2 SRAM，使用了带有使能输入的 D 锁存器来控制三态输出。

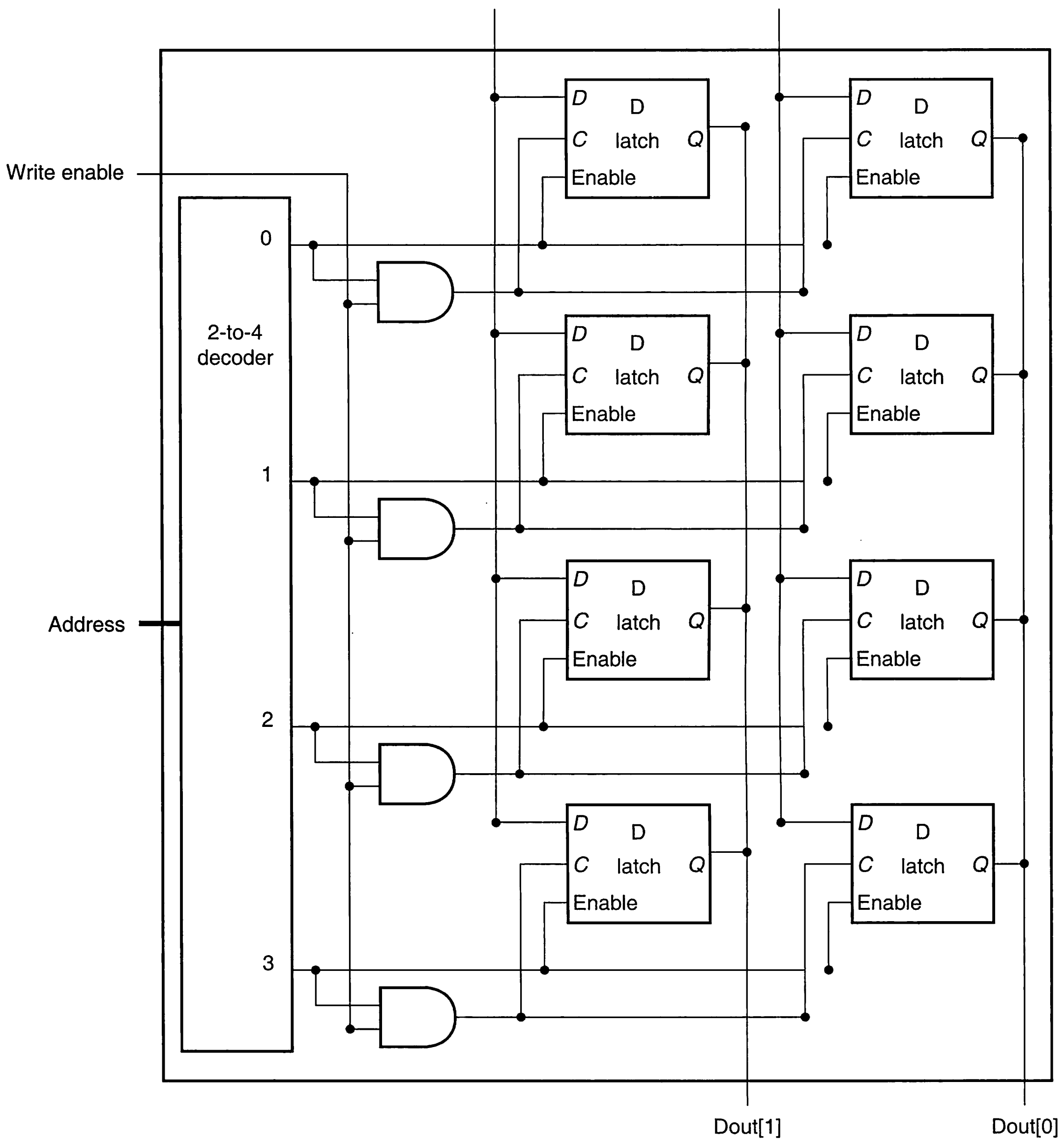

图 A-9-3　4 × 2 SRAM 的基本结构包括一个译码器，用于选择激活哪一对单元。激活的单元使用连接到垂直位线的三态输出，垂直位线提供被请求的数据。选择单元的地址在一组水平地址线的某条线上发送，称为字线。为简单起见，省略了输出使能和片选信号，但它们可以简单地通过一些与门接入进来