图 A-9-3 中的设计消除了对巨型多选器的需求。但是它仍然需要非常大的译码器和大量的字线。例如，在 4M × 8 SRAM 中，需要一个 22-4M 的译码器和 4M 条字线（用于各个触发器的使能）。为了避免这个问题，大型存储器被做成矩形阵列并使用二级译码装置。图 A-9-4 显示了如何使用二级译码实现 4M × 8 SRAM。二级译码对于理解 DRAM 的运行方式非常重要。

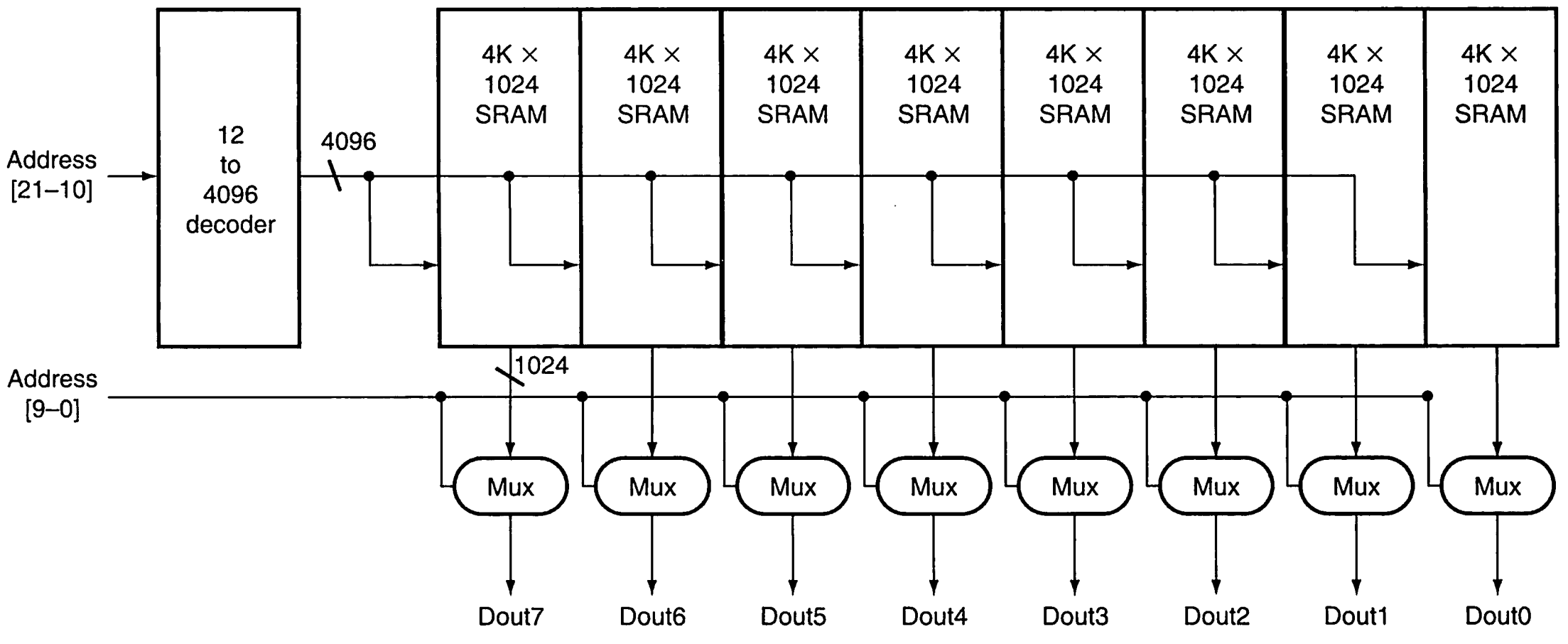



图 A-9-4 利用 4K × 1024 阵列组成 4M × 8 SRAM。第一个译码器生成 8 个 4K × 1024 阵列的地址，然后使用一组多选器从每个 1024 位宽的阵列中选择 1 位。这比需要巨型译码器或巨型多选器的单级译码更容易设计。实际上，现在这种大小的 SRAM 可能会使用更大数量的模块，每个块都更小

最近，我们见证了同步 SRAM（SSRAM）和同步 DRAM（SDRAM）的发展。同步 RAM 提供的关键功能是能够从阵列或行中的一系列有序地址中实现数据的簇发式传输。簇发由起始地址和簇发长度定义。同步 RAM 的速度优势来自于能够在簇发中传输多位数据而无须指定额外的地址位。但在簇发中传输连续位会受到时钟控制。簇发方式中，对于额外地址位的消除显著提高了传输数据块的速率。由于这种能力，同步 SRAM 和 DRAM 正迅速成为在计算机中构建存储系统的首选 RAM。我们在下一节和第 5 章中更详细地讨论了存储系统中同步 DRAM 的使用。

## A.9.2 DRAM

在静态 RAM(SRAM) 中，存储在单元中的值保存在一对反相门上，并且只要给它供电，该值就可以无限期地保持。在动态 RAM（DRAM）中，保存在单元中的值作为电荷存储在电容器中。然后使用一个晶体管来访问该存储的电荷，以读取该值或覆盖掉存储的电荷。由于 DRAM 存储一位仅使用一个晶体管，因此每位的密度更高，也更便宜。相比之下，SRAM 每位需要四到六个晶体管。因为 DRAM 将电荷存储在电容器上，所以不能无限期地保存，必须要定期刷新。这就是为什么这种存储结构被称为动态，而不是 SRAM 单元中的静态存储。

要刷新存储单元，我们只需读取其内容并将其写回。电荷可以保持几毫秒，这对应于近一百万个时钟周期。如今，单芯片存储控制器通常独立于处理器处理刷新功能。如果必须从 DRAM 中读出每个位然后再单独写回，一旦使用了包含几兆字节的 DRAM，则需要不断刷新 DRAM 而没有时间访问它。好在 DRAM 也使用了二级译码结构，这就可以在读周期后紧跟一个写周期来刷新整个行（共享一个字线）。通常，刷新操作占 DRAM 有效时间的

1%~2%，剩余的 98% ～ 99% 的时间可用于读和写数据。

**详细阐述** DRAM 如何读写存储在单元中的信号？单元内的晶体管是一个开关，称为通道晶体管，允许存储在电容上的值被读取或写入。图 A-9-5 显示了单个晶体管单元的外观。传输晶体管的作用类似于开关：当字线上的信号有效时，开关闭合，将电容器连接到位线。如果操作是写操作，则将要写入的值放在位线上。如果该值为 1，则电容器将被充电；如果该值为 0，则电容器将放电。因为 DRAM 必须检测存储在电容器中的非常小的电荷，所以读取稍微复杂一些。在激活用于读取的字线之前，将位线充电到低电压和高电压之间一半的电压。然后，通过激活字线，电容器上的电荷被读出到位线上。这导致位线稍微向高或低电压方向移动，而这种变化可以由能够检测电压微小变化的传感放大器检测到。

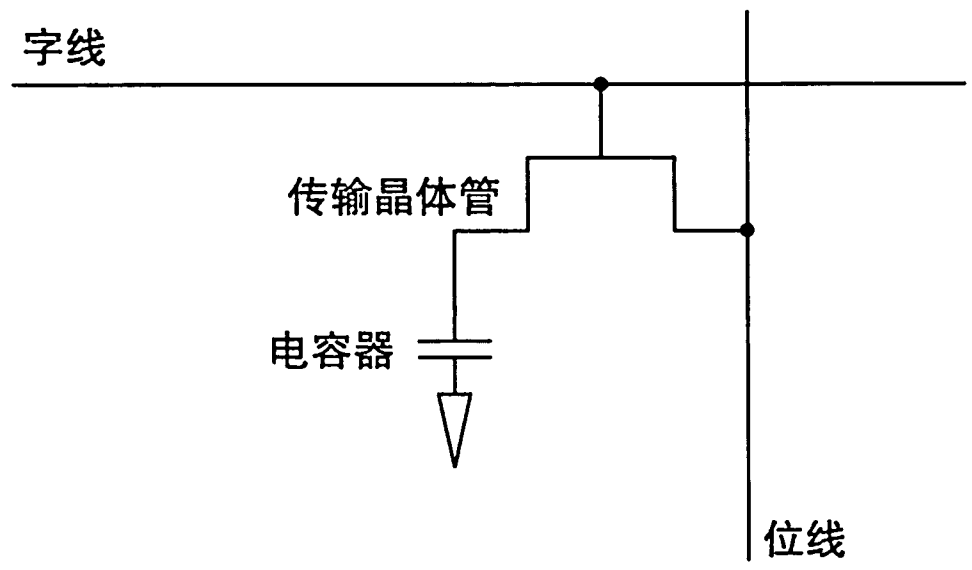



图 A-9-5　单晶体管的 DRAM 单元，包含存储单元内容的电容器和用于访问单元的晶体管

如图 A-9-6 所示，DRAM 使用二级译码器，分别实现行访问和列访问。行访问选择多行中的一行并激活相应的字线，被激活行中所有列的内容存储在一组锁存器中。然后，列访问从列锁存器中选择数据。为了节省引脚并降低封装成本，行地址和列地址共享地址线。一对称为 RAS（行访问选通）和 CAS（列访问选通）的信号用于表示正在提供行或列地址。刷新是指将列读入到列锁存器后，再将相同的值写回。因此，整个行可在一个周期内完成刷新。和内部电路结合的两级寻址方案，使得 DRAM 的访问时间比 SRAM 访问时间长得多（5 ～ 10 倍）。2004 年，典型的 DRAM 访问时间为 45 ～ 65ns，256Mbit DRAM 已量产，1GB DRAM 的首批客户样片于 2004 年第一季度生产出来。由于每位成本更低，DRAM 成为主存的首选，而更快的访问时间使 SRAM 成为 cache 的首选。

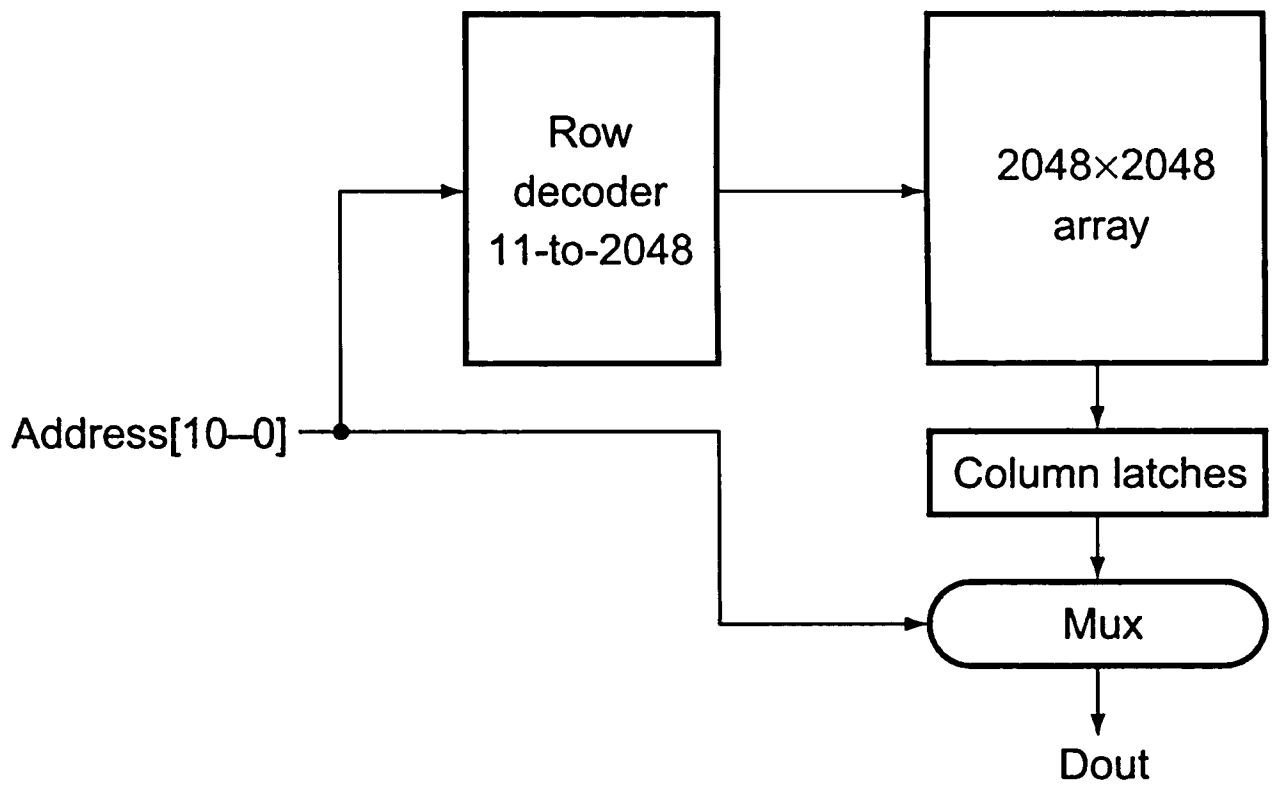



图 A-9-6　利用 2048 × 2048 阵列实现的 4M × 1DRAM。行访问使用 11 位来选择一行，然后将其锁存在 2048 个 1 位锁存器中。多选器选择来自这 2048 个锁存器的输出位。RAS 和 CAS 信号控制地址线是否被发送到行译码器或列多选器

读者可能会发现，64M × 4 DRAM 实际上在每行访问时能访问 8K 位，然后在列访问时丢弃得只剩下 4 位。DRAM 设计人员早已使用 DRAM 的内部结构实现了更高的带宽。这是通过允许在不改变行地址的情况下改变列地址来完成的，从而可以访问列锁存器中的其他

位。为了使这个过程更快更精确，地址输入被时钟同步，这产生了当今使用的主流 DRAM 形式：同步 DRAM 或 SDRAM。

自 1999 年以来，对于大多数基于 cache 的主存系统来说，SDRAM 已成为首选的存储器芯片。SDRAM 通过在时钟信号的控制下顺序簇发传输一行中的所有位提供行内快速访问。2004 年，DDR RAM（双倍数据传输率 RAM）是最常用的 SDRAM 类型，之所以被称为双倍数据传输率，是因为它可以在外部提供时钟的上升沿和下降沿都传输数据。正如在第 5 章中所讨论的，这些高速传输可用于增加主存可用的带宽，以满足处理器和 cache 的需求。

## A.9.3 错误修正

由于大容量存储器中存在数据损坏的可能性，因此大多数计算机系统使用某种校验码来检测可能存在的数据损坏。一种普遍使用且足够简单的是奇偶校验码。在奇偶校验码中，计算一个字中的 1 的数量；如果 1 的数量是奇数，则该字具有奇校验，否则为偶校验。当一个字写入存储器时，也会写入奇偶校验位（1 表示奇数，0 表示偶数）。然后，当读出该字时，读取并检查奇偶校验位。如果存储器字的奇偶校验和存储的奇偶校验位不匹配，则说明发生错误。

1 位的奇偶校验方案可以检测数据项中最多 1 位的错误；如果存在 2 位错误，则 1 位奇偶校验方案将不会检测到任何错误，因为发生 2 位错误数据的奇偶校验位不发生变化，仍然满足要求。（实际上，1 位奇偶校验方案可以检测到任何奇数个错误；但是，具有 3 个错误的概率远低于具有 2 个错误的概率，因此，实际上，1 位奇偶校验码仅限于检测 1 位错误。）当然，奇偶校验码无法判断数据项中的哪个位出错。

1 位奇偶校验方案是一种**检错码**，还有一种纠错码（ECC），它能够检测并纠正错误。对于大容量主存，许多系统使用的纠错码允许检测最多 2 位的错误并纠正单个错误位。这些校验码使用更多位来编码数据的方式，例如，用于主存的常用校验码每 128 位数据需要 7 或 8 位纠错码。

**检错码**：一种校验码，可以检测到数据中的错误，但不能检测到精确的位置，从而无法纠正错误。

**详细阐述** 1 位奇偶校验码是距离为 2 的编码方法，这意味着对于数据位和校验位来说，任何 1 位的变化都可以被检测出来。例如，如果更改数据中的某个位，则检验位将出错，反之亦然。当然，如果改变 2 位（任何 2 个数据位或 1 个数据位和 1 个校验位），奇偶校验位将依然和数据匹配，从而将无法检测到错误。因此，这是一种距离为 2 的校验码。

为了检测多个错误或纠正错误，需要一种距离为 3 的编码，该编码具有这个特性：纠错码和数据的任意组合之间至少有 3 位不同。假设存在这样的编码方法，并且在数据中有一个错误。在这种情况下，我们可以检测出里面有 1 位错误，并对其进行纠正。如果有两个错误，则可以识别出存在错误，但无法纠正错误。我们来看一个例子，以下是 4 位数据项的数据字和距离为 3 的纠错码。

| 数据字 | 纠错码 | 数据 | 纠错码 |
|---|---|---|---|
| 0000 | 000 | 1000 | 111 |
| 0001 | 011 | 1001 | 100 |
| 0010 | 101 | 1010 | 010 |
| 0011 | 110 | 1011 | 001 |
| 0100 | 110 | 1100 | 001 |
| 0101 | 101 | 1101 | 010 |
| 0110 | 011 | 1110 | 100 |
| 0111 | 000 | 1111 | 111 |

为了了解其工作原理，让我们选择一个数据字，比如 0110，其纠错码为 011。以下是该数据发生 1 位错误的四种情况：1110，0010，0100，0111。请注意，011 既是 0110 的纠错码，也是 0001 的纠错码。如果纠错译码器接收到具有错误的四个可能数据字之一，则必须在校正到 0110 或 0001 之间进行选择。而这四个字如果相对于 0110 只有 1 位错误，则它们每个都有 2 位与 0001 不同。因此，由于 1 位错误的概率更高，纠错机制可以很容易地选择纠正到 0110。要检测到 2 位错误，只需注意所有 2 位错误的组合具有不同的编码。相同编码的一次重用与 3 位不同，但如果我们纠正 2 位错误，将纠正成错误的值，因为译码器将假定只发生了 1 位错误。如果我们想要对 1 位、2 位错误都有纠错功能，则需要一个距离为 4 的校验码。

虽然我们在解释中区分了校验码和数据，但事实上，纠错码将校验码和数据的组合视为更大编码中的单个字（在本例中为 7 位）。因此，它以与数据位中的错误相同的方式处理校验码中的错误。

虽然上面的例子对 $n$ 位数据需要 $n-1$ 位校验码，但其实所需的位数增长缓慢。对于距离为 3 的校验码，64 位字需要 7 位，128 位字需要 8 位。这种校验码后来被称为汉明码，因为 R. Hamming 描述了创建这种校验码的方法。

## A.10 有限状态自动机

**有限状态自动机**：一种时序逻辑函数。它包括一组输入、输出、将当前状态和输入映射到新状态的下一状态函数，以及将当前状态和输入映射到一组有效输出的输出函数。

如前所述，数字逻辑电路可以分为组合电路和时序电路。时序电路的状态存储在系统内部的存储元件中，它们的行为取决于所提供的一系列输入以及内部存储的数据或系统状态。因此，时序电路不能用真值表来描述。相反，它被描述为**有限状态自动机**（或状态机）。有限状态自动机具有一组状态和两个函数（**下一状态函数**和输出函数）。状态集对应于内部存储的所有可能值。因此，对于 $n$ 位存储，就有 $2^n$ 个状态。下一状态函数是一种组合函数，给定输入和当前状态，就可以确定系统的下一个状态。输出函数根据当前状态和输入产生一组输出。图 A-10-1 给出了有限状态自动机的图示。

**下一状态函数**：一种组合函数，通过给定输入和当前状态，就可以确定有限状态自动机的下一个状态。

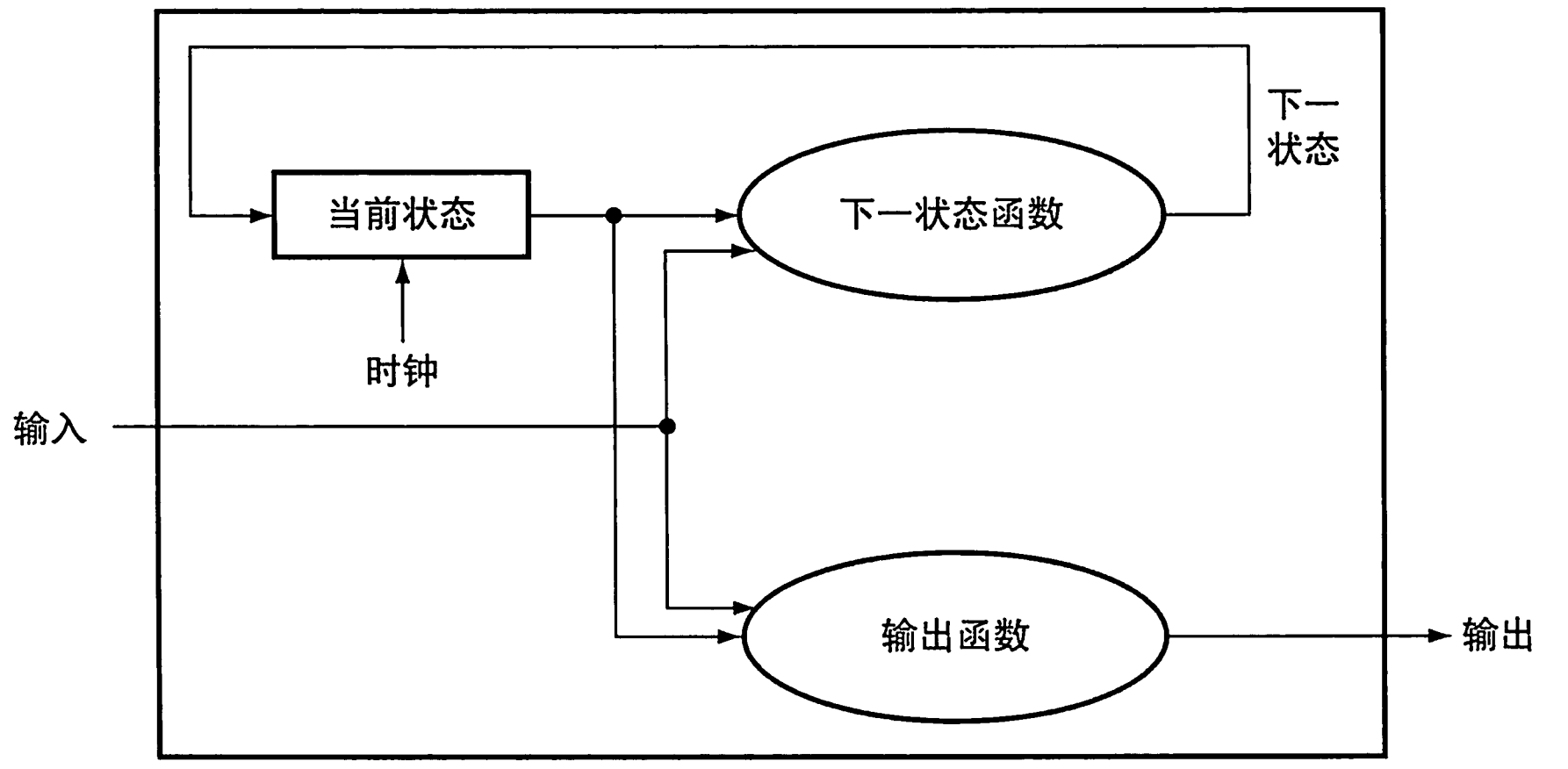



图 A-10-1 状态机由包含状态和两个组合函数（下一状态函数和输出函数）的内部存储器组成。通常，输出函数只会将当前状态作为其输入，这不会改变时序机的能力，但会影响其内部值

在本节和第 4 章讨论的状态机都是同步的。这说明状态随时钟周期变化，并且每个时钟周期计算一次新状态。因此，状态单元仅在时钟边沿更新。在本节和第 4 章中我们使用了这种方法，但通常不显式地表明时钟。在第 4 章中，我们使用状态机来控制处理器的执行和数据通路的操作。

为了说明有限状态自动机的运行和设计，让我们看一个简单的经典示例：控制交通信号灯。(第 4 章和第 5 章包含使用有限状态自动机控制处理器执行的更详细示例。) 当有限状态自动机用作控制器时，输出函数通常仅依赖于当前状态。这种有限状态自动机称为摩尔机。这是我们在本书中所使用的有限状态自动机。如果输出函数依赖于当前状态和当前输入，则该机器称为米利机。这两种机器的功能相同，二者在物理上可相互转换。摩尔机的基本优势在于更快，而米利机则更小巧，因为它比摩尔机需要的状态更少。在第 5 章中，我们更详细地讨论了这些差异，并给出了使用米利机实现的有限状态控制的 Verilog 版本。

我们的示例是交通信号灯，其位于一个南北路线和东西路线相交的位置。为简单起见，我们只考虑绿灯和红灯，黄灯留作练习。我们希望灯在各方向的切换周期不超过 30 秒。因此采用了 0.033Hz 的时钟，以便信号灯在状态之间的控制周期不超过 30 秒。其中有两个输出信号：

- NSlite：当该信号有效时，南北方向上的交通灯为绿色；当该信号无效时，南北方向上的交通灯为红色。
- EWlite：当该信号有效时，东西方向上的交通灯为绿色；当该信号无效时，东西方向上的交通灯为红色。
- 此外，还有两个输入：
- NScar：表示汽车位于探测器处，探测器在南北向道路的交通灯前方（往南或往北）。
- EWcar：表示汽车位于探测器处，探测器在东西向道路的交通灯前方（往东或往西）。

只有当汽车正在等待向另一个方向行驶时，交通灯才会在红绿灯之间切换。否则，交通灯的状态不变，直到该方向上的最后一辆汽车通过交叉路口为止。

为了实现该交通灯，需要两个状态：

- NSgreen：南北向的交通灯为绿色。
- EWgreen：东西向的交通灯为红色。

我们还需要通过状态表来构造下一状态函数：

| | 输入 | | |
|---|---|---|---|
| | NScar | EWcar | 下一状态 |
| NSgreen | 0 | 0 | NSgreen |
| NSgreen | 0 | 1 | EWgreen |
| NSgreen | 1 | 0 | NSgreen |
| NSgreen | 1 | 1 | EWgreen |
| EWgreen | 0 | 0 | EWgreen |
| EWgreen | 0 | 1 | EWgreen |
| EWgreen | 1 | 0 | NSgreen |
| EWgreen | 1 | 1 | NSgreen |

请注意，我们没有在算法中指定当汽车从两个方向接近时会发生什么。在这种情况下，上面的下一状态函数表需要修改以确保来自一个方向的汽车不会导致另一方向的堵塞。

有限状态自动机可通过指定输出函数来实现。

在研究如何实现这个有限状态自动机之前，我们先看一下有限状态自动机的图形表示。在该表示中，节点表示状态。在节点中，我们放置了一个对该状态有效的输出列表。有向弧用于指出下一状态函数，弧上的标签将输入条件指定为逻辑函数。图 A-10-2 给出了这种有限状态自动机的图形表示。

| | NSlite | EWlite |
|---|---|---|
| NSgreen | 1 | 0 |
| EWgreen | 0 | 1 |

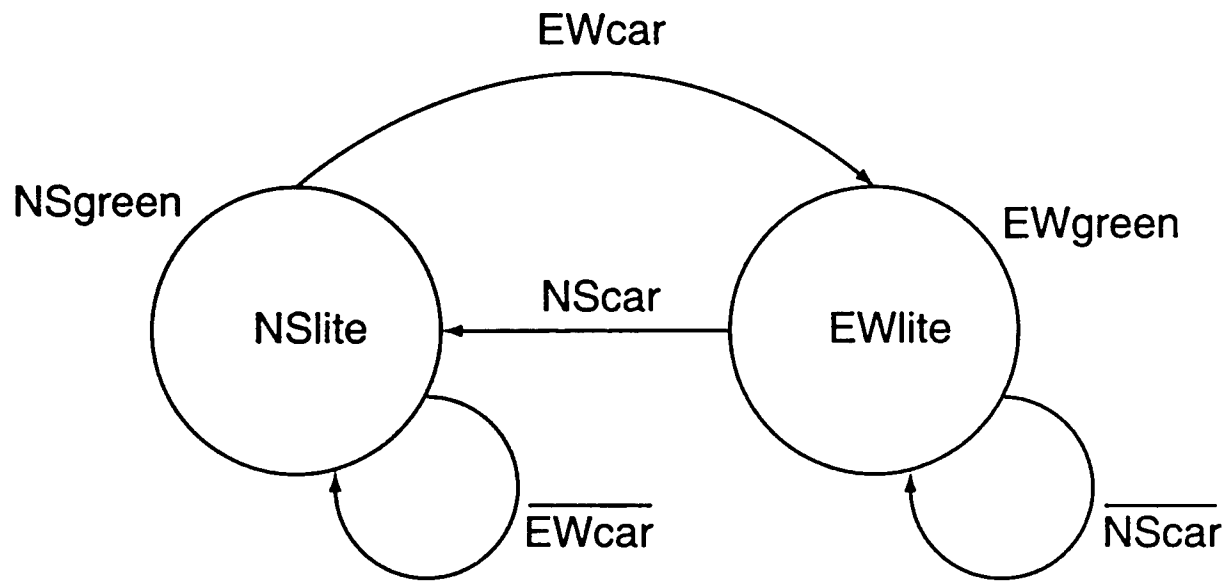



图 A-10-2　2 状态交通灯控制器的图形表示。我们简化了状态转换的逻辑函数。例如，在下一个状态表中从 NSgreen 到 EWgreen 的传输信号是（$\overline{\text{NScar}}$ · EWcar）+（NScar · EWcar），相当于 EWcar

有限状态自动机可以这样实现：利用寄存器保持当前状态，组合逻辑电路计算下一状态函数和输出函数。图 A-10-3 给出了状态量为 4 位的有限状态自动机，它最多可以有 16 个状态。要以这种方式实现有限状态自动机，首先要给状态标号，此过程称为状态分配。例如，我们可以将 NSgreen 指定为状态 0，将 EWgreen 指定为状态 1。状态寄存器保存 1 位数据。下一状态函数由以下公式计算：

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

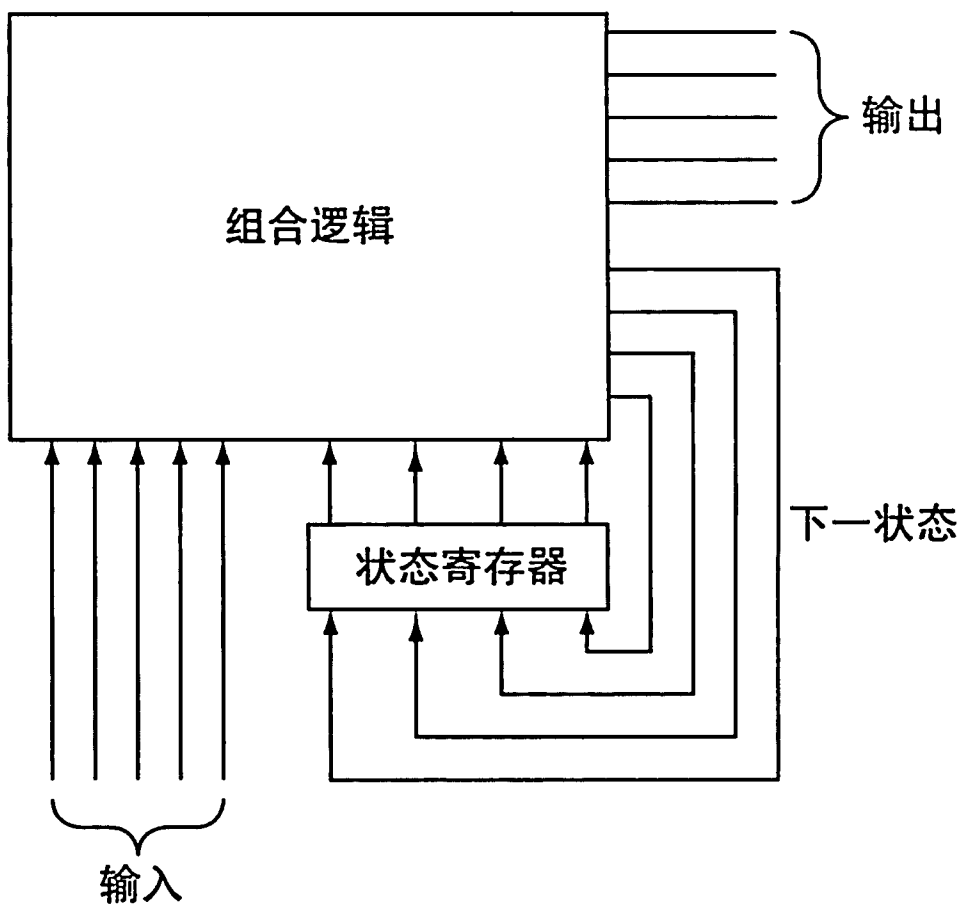



图 A-10-3　有限状态自动机用状态寄存器来实现，它包含一个当前状态以及用于计算下一个状态函数和输出函数的组合逻辑块。后两个函数经常被拆分并用两个独立的逻辑块实现，因为这样做需要的门电路更少

其中，CurrentState 是状态寄存器的值（0 或 1）。NextState 是下一状态函数的输出，会在时钟周期末尾被写入状态寄存器。输出函数也很简单：

$$\text{NSlite} = \overline{\text{CurrentState}}$$
$$\text{EWlite} = \text{CurrentState}$$

组合逻辑电路通常使用结构化逻辑来实现，例如 PLA。PLA 可以通过下一状态表和输出函数表自动构建。事实上，还可以使用计算机辅助设计（CAD）软件，它先把有限状态自动机进行图形或文本表示，然后再自动生成优化电路设计。在第 4 章和第 5 章中，有限状态自动机被用于控制处理器执行。附录 C 详细讨论了利用 PLA 和 ROM 来实现这些控制。

为了展示我们如何在 Verilog 中编写控制逻辑，图 A-10-4 给出了综合的 Verilog 版本。请注意，对于这个简单的控制功能，米利机没有用，但是在第 5 章中使用这种规范实现控制功能的就是米利机，并且比摩尔机拥有的状态更少。

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);

   input EWCar, NSCar,clock;
output EWLite,NSLite;

reg state;

initial state=0;  //set initial state

//following two assignments set the output, which is based
only on the state variable
assign NSLite = ~ state; //NSLite on if state = 0;
assign EWLite = state; //EWLite on if state = 1

always @(posedge clock) // all state updates on a positive
clock edge
   case (state)
      0: state = EWCar; //change state only if EWCar

      1: state = ~ NSCar; // change state only if NSCar

   endcase
endmodule
```

图 A-10-4　交通信号灯的 Verilog 版本

**自我检测**　摩尔机最少要有多少个状态数，从而可以有更少状态的米利机？

a. 2，因为 1 状态的米利机可能做相同的事情。

b. 3，因为可能有一个简单的摩尔机，它跳转到两种不同的状态之一，并在此之后总是恢复到原始状态。对于这种简单情形，可以使用 2 状态米利机。

c. 至少需要 4 个状态才能体现米利机相对于摩尔机的优势。

## A.11　定时方法

在本附录和本书剩余内容中，我们均使用边沿触发的定时方法。这种方法的优点在于，与电平触发的方法相比，它更易于解释和理解。在本节中，我们将更详细地阐述这种定时方法，并介绍对电平敏感的时钟控制。在本节末尾，我们会简要讨论异步信号和同步器，这对于数字逻辑设计人员来说也是一个重要问题。

本节的目的是介绍时钟同步方法的主要概念，并假定了一些重要的条件。如果有兴趣更

详细地了解定时方法，请参阅本附录末列出的参考文献。

我们采用边沿触发的定时方法，因为它更容易解释，并且达到正确性几乎不需要什么规则。特别是，如果假设所有时钟同时到达，并且时钟周期足够长，那么可以保证，组合逻辑电路中具有边沿触发寄存器的系统，可以在没有竞争的情况下正确执行操作。当状态单元的值取决于不同逻辑元件的相对速度时，就会发生竞争。在边沿触发设计中，时钟周期必须足够长，以满足从一个触发器通过组合逻辑到另一个触发器的路径时间，另一个触发器还必须满足建立时间要求。图 A-11-1 显示了使用上升沿触发的触发器系统必须满足的时钟条件。在这样的系统中，时钟周期（或周期时间）必须至少与下式一样大：

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}}$$

对于这 3 种延迟的最坏情况值，定义如下：

- $t_{\text{prop}}$ 是信号通过触发器传播的时间，有时也被称为 clock-to-$Q$。
- $t_{\text{combinational}}$ 是对于任何组合逻辑（定义为被两个触发器包围的部分）的最长延时。
- $t_{\text{setup}}$ 是在时钟上升沿到来之前，触发器的输入必须保持有效的时间。

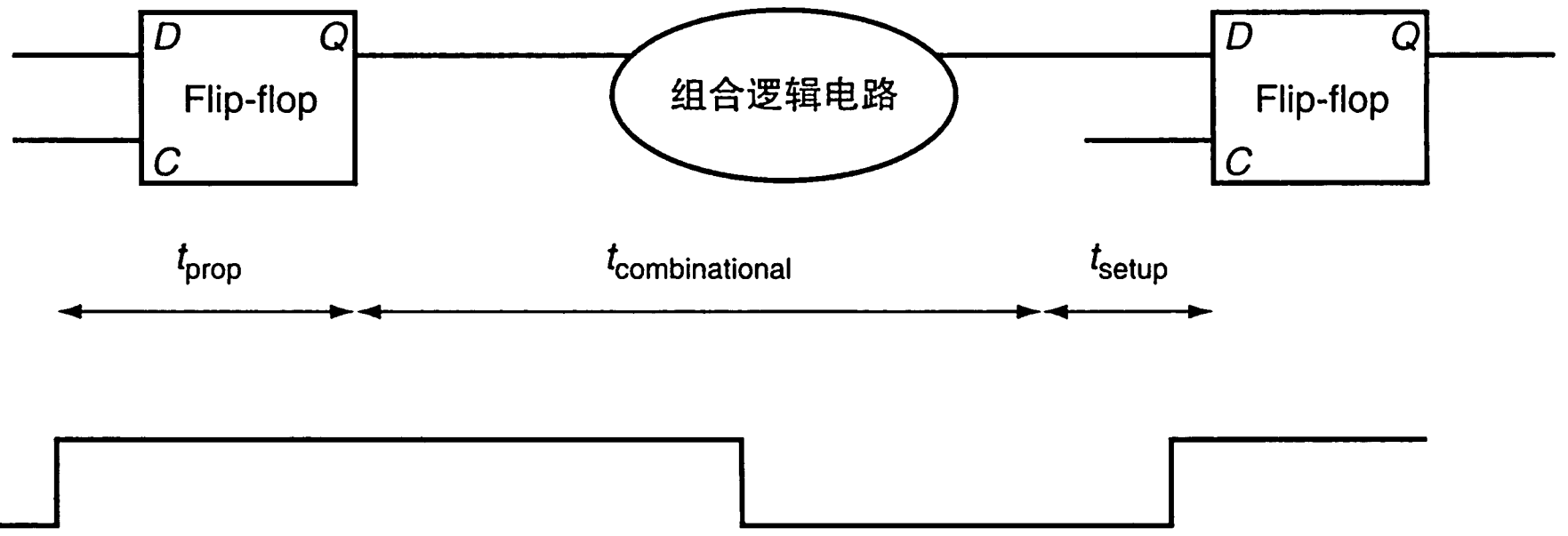



图 A-11-1 在边沿触发设计中，时钟周期必须足够长，以保证下一个时钟沿到来之前的信号在建立时间内有效。信号从触发器输入传播到触发器输出的时间是 $t_{\text{prop}}$，然后，信号需要 $t_{\text{combinational}}$ 的时间通过组合逻辑，并且必须在下一个时钟沿到来之前的 $t_{\text{setup}}$ 时间内为有效

我们做了一个简化的假设：满足触发的保持时间的要求。这对于现代逻辑设计来说几乎不是问题。

边沿触发设计中必须考虑的另一个复杂因素是**时钟扭斜**。时钟扭斜是两个状态元件看到的时钟边沿间的绝对时间差。时钟扭斜产生的原因是时钟信号通常会使用两条不同的路径，经过略微不同的延时，到达两个不同的状态元件。如果时钟扭斜足够大，则状态元件的值可能会变化，从而导致另一个触发器的输入在第二个触发器看到时钟边沿之前发生变化。

**时钟扭斜**：两个状态元件看到时钟边沿的时间之间的绝对时间差。

抛开建立时间和触发器传播延迟，图 A-11-2 说明了这个问题。为避免错误发生，需要增大时钟周期以克服最大时钟扭斜。因此，时钟周期必须大于：

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}} + t_{\text{skew}}$$

给时钟周期加上这个限制条件后，允许两个时钟到达的先后次序颠倒，即第二个时钟提前 $t_{\text{skew}}$ 到达，电路依然会正常工作。设计人员通过仔细设计时钟信号的路由来减少时钟扭斜问题，最大限度地缩短到达时间的差异。此外，聪明的设计人员还通过稍稍增大时钟周期来

减少时钟扭斜，这些变化在逻辑元件和电源上都是允许的。由于时钟扭斜也会影响保持时间的要求，因此最小化时钟扭斜的值非常重要。

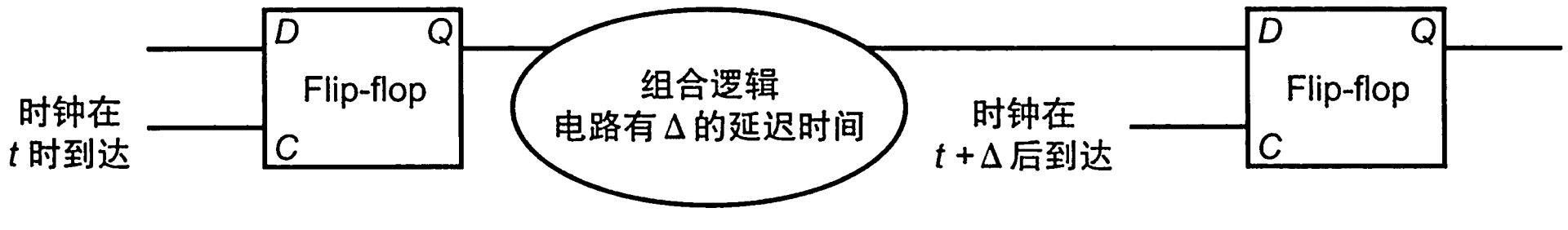



图 A-11-2 时钟扭斜如何导致竞争并导致错误操作。由于两个触发器看到时钟的时间不同，存储在第一个触发器中的信号可以向前传输，并在时钟到达第二个触发器之前改变第二个触发器的输入

边沿触发设计有两个缺点：需要额外的逻辑电路；有时可能会更慢。比较 D 触发器与用于构造触发器的电平敏感锁存器，边沿触发设计需要更多逻辑电路。另一种方法是采用**电平敏感的时钟控制**。由于对电平敏感方法的状态变化不是瞬时的，因此该方法稍微复杂一些，需要考虑额外的因素才能使其正常运行。

**电平敏感的时钟控制**：一种定时方法，其状态变化发生在时钟的高或低电平，和边沿触发设计不同，它不是瞬时的。

## A.11.1 电平敏感的时钟控制

在电平敏感的时钟控制中，状态变化发生在高电平或低电平，但不是瞬时的，因为它们不采用边沿触发。由于状态的非瞬时变化，很容易发生竞争现象。如果时钟足够慢，为确保电平敏感设计也能正常工作，设计人员使用了双向时钟控制。双向时钟控制是一种利用两个非重叠时钟信号的方法。由于两个时钟（通常称为 $\Phi 1$ 和 $\Phi 2$）是非重叠的，因此在任何给定时间，如图 A-11-3 所示，至多有一个时钟信号为高电平。我们可以使用这两个时钟来构建一个包含电平敏感锁存器的系统，且不受任何竞争条件的影响，就如边沿触发设计一样。

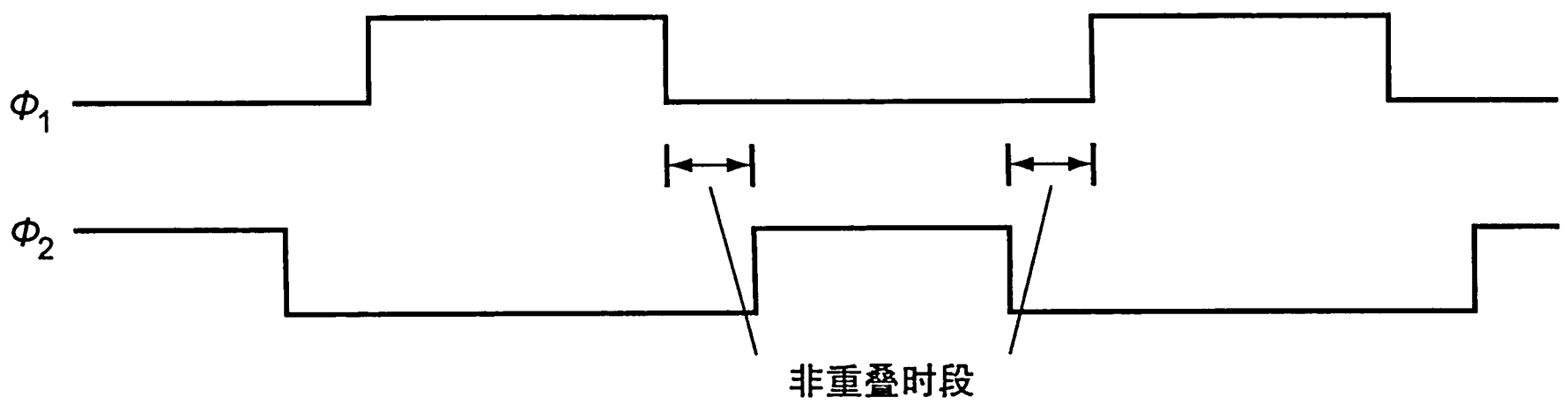



图 A-11-3 双向时钟控制机制，显示了每个时钟的周期和非重叠时段

设计这种系统的一种简单方法是交替使用在 $\Phi 1$ 上处于开状态和在 $\Phi 2$ 上处于开状态的锁存器。因为两个时钟不同时有效，所以不会发生竞争现象。如果组合逻辑的输入是 $\Phi 1$ 时钟，则其输出会在 $\Phi 2$ 时钟被锁存，该输出仅在输入锁存器闭合的 $\Phi 2$ 期间开放，因此输出为有效。 图 A-11-4 显示了具有双向时钟控制和交替锁存器的系统如何工作。和边沿触发设计一样，我们必须注意时钟扭斜问题，特别是在两个时钟相位之间。通过增加两个相位间的非重叠量，就可以减少潜在的误差范围。因此，如果每个相位足够长并且相位间的非重叠量足够大，则能保证系统正确操作。

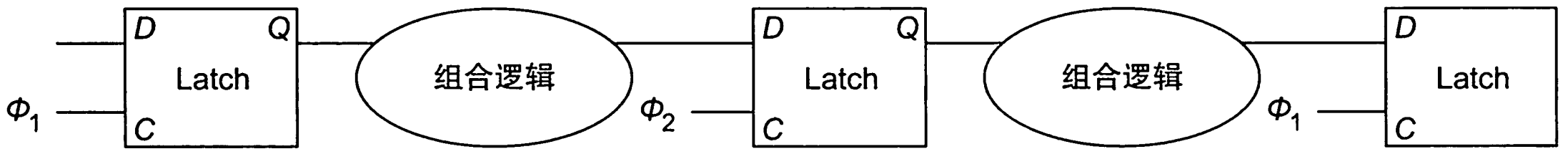



图 A-11-4　具有交替锁存器的双向时钟控制方法，表明了系统如何在两个时钟相位上工作。锁存器的输出在与 *C* 输入相反的相位上是稳定的。因此，第一个组合模块输入在 Φ2 期间具有稳定的输入，并且其输出在 Φ2 时钟锁存。第二个（最右边）组合模块以相反的方式运行，在 Φ1 期间具有稳定的输入。因此，通过组合块的延时确定了各个时钟必须有效的最短时间。非重叠周期的大小由最大时钟扭斜和各逻辑块的最小延时决定

## A.11.2　异步输入和同步器

通过使用单个时钟或双相时钟，如果时钟扭斜问题得到了解决，就可以消除竞争现象。然而，使用单个时钟支持整个系统功能且仍然保持很小的时钟扭斜是不切实际的。即使 CPU 可能使用单个时钟，I/O 设备也可能有自己的时钟。异步设备可以通过一系列握手操作与 CPU 通信。要将异步输入转化为可用于改变系统状态的同步信号，需要使用同步器，其输入是异步信号和时钟，其输出是与输入时钟同步的信号。

构建同步器的第一步是使用边沿触发的 D 触发器，如图 A-11-5 所示，输入信号 D 是异步信号。因为使用握手协议进行通信，且信号将被一直保持有效直到它被确认，所以是在当前时钟还是下一个时钟上检测到异步信号的有效状态并不重要。因此，除了一个小问题之外，这种简单的结构已经足以准确地对信号进行采样。

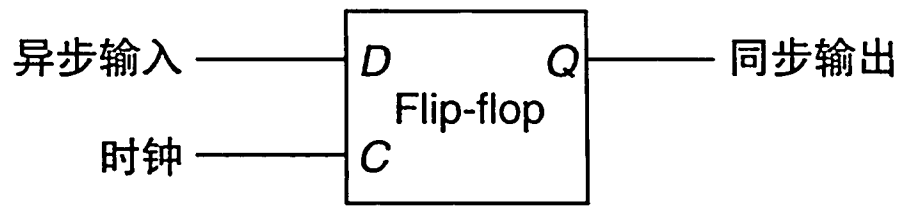



图 A-11-5　由 D 触发器构建的同步器，用于采样异步信号以产生与时钟同步的输出。这个“同步器”无法正常工作

问题在于会出现**亚稳态**的情况。假设当时钟边沿到达时，异步信号在高电平和低电平之间振荡。显然，此时就难以判断信号的高电平还是低电平被锁存。这个问题可以先被搁置，然而更糟的是：当采样的信号在建立时间和保持时间内不稳定时，触发器可能进入亚稳态。在这种状态下，输出将不具有合法的高值或低值，而是介乎二者之间，处于一种不确定状态。此外，触发器不能保证在有限的时间内退出该状态。一些逻辑块看到的触发器的输出可能为 0，而其他逻辑块看到的可能为 1。这种情况被称为**同步失败**。

**亚稳态**：信号在建立时间和保持时间内不稳定时被采样，导致采样值落在高值和低值之间的不确定区域的一种情况。

**同步失败**：触发器进入亚稳态，导致读取触发器输出的某些逻辑块看到的是 0，而其他逻辑块看到的是 1 的情况。

在同步系统中，通过确保始终满足触发器或锁存器的建立时间和保持时间，可以避免同步失败问题，但是当输入是异步时，则无法避免了。这种情况下，唯一可能的解决方案是在查看触发器的输出之前等待足够长的时间以确保其输出稳定或已退出亚稳态。但究竟要等多久呢？由于触发器处于亚稳态的概率呈指数级下降，因此在很短的时间内，触发器处于亚稳态的概率就非常低。但是，概率永远不会变成 0！因此设计人员等待的时间足够长的话，同步失败的概率就会非常低，下一次失败可

能是数年甚至数千年之后了。

对于大多数触发器设计，等待时间会比建立时间设置得长几倍，从而使同步失败的概率非常低。如果时钟周期长于潜在的亚稳态周期（很可能），则可以使用两个 D 触发器构建一个安全的同步器，如图 A-11-6 所示。如果读者有兴趣了解有关这些问题的更多信息，请进一步查阅参考文献。

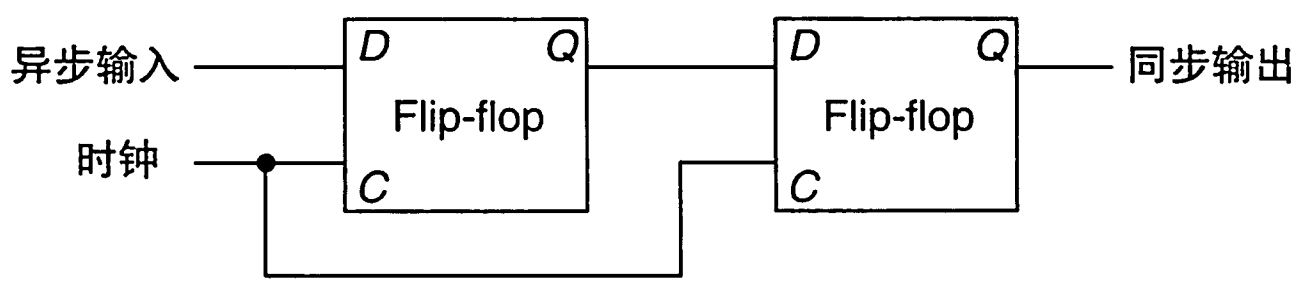



图 A-11-6 如果亚稳态周期小于时钟周期，则此同步器将正常工作。虽然第一个触发器的输出可能是亚稳态的，但是在第二个时钟之前，任何其他逻辑元件都不会看到它。当第二个时钟到来时，第二个 D 触发器对信号进行采样，此时该信号不应再处于亚稳态

**自我检测** 假设我们的设计具有非常大的时钟扭斜，比寄存器的传播时间还长。对于该设计，是否总是可以将时钟减慢，从而保证逻辑的正常运行？

**传播时间**：触发器的输入传播到其输出所需的时间。

a. 可以，即使时钟扭斜很大，如果时钟足够慢的话，信号也总是能正常传播，所以该设计能正常工作。

b. 不可以，因为可能有这样一种情况：对于同一个时钟边沿，两个寄存器看到的时间相差足够大，以至于在同一个时钟边沿，其中一个寄存器会发现另一个寄存器已被触发，且输出已经传播出去了。

## A.12 现场可编程设备

**现场可编程设备**：一种包含组合逻辑的集成电路，也可能包含可由终端用户配置的存储设备。

**可编程逻辑设备**：一种包含组合逻辑的集成电路，组合逻辑的功能由终端用户配置。

在定制或半定制芯片中，设计人员可以利用底层结构的灵活性来轻松实现组合或时序逻辑。对于不想使用定制或半定制 IC 的设计人员来说，如何利用可用的高集成度电路实现复杂的逻辑？除了定制或半定制 IC 之外，时序和组合逻辑设计的最常用组件是**现场可编程设备**（FPD）。FPD 是包含组合逻辑的集成电路，并可能包括可由终端用户配置的存储器设备。

FPD 通常分为两大阵营：**可编程逻辑设备**（PLD），它是纯粹的组合逻辑；**现场可编程门阵列**（FPGA），它提供组合逻辑和触发器。PLD 有两种形式：**简单 PLD**（SPLD），通常是 PLA 或**可编程阵列逻辑**（PAL）；复杂 PLD，允许多个逻辑块以及块之间的可配置互联。当谈到 PLD 中的 PLA 时，指的是具有用户可编程与阵列和或阵列的 PLA。PAL 与 PLA 类似，除了它的或阵列是固定的以外。

**现场可编程门阵列**：一种可配置的集成电路，包括组合逻辑和触发器。

**简单可编程逻辑设备**：可编程逻辑设备，通常包含单个 PAL 或 PLA。

**可编程阵列逻辑**：包含一个可编程与阵列和固定的或阵列。

在讨论 FPGA 之前，讨论如何配置 FPD 很有帮助。配置本质上是一个关于在何处建立或断开连接的问题。门和寄存器结构是静态的，但是连接是可配置的。注意，通过配置连接，用户可以决定实现什么逻辑功能。考虑一个可配置的 PLA：通过确定连接在与

阵列和或阵列中的位置，用户可以指定在 PLA 中运算的逻辑功能。FPD 中的连接是永久性的或可重新配置的。永久连接涉及在两根连线之间建立或破坏连接。目前的 FPLD 都使用**反熔断**技术，允许在编程时建立连接，然后永久保持连接。配置 CMOS FPLD 的另一种方法是通过 SRAM。上电时下载配置信息到 SRAM，其内容控制开关的设置，进而确定连接的金属线。SRAM 控制的优点在于可以通过改变 SRAM 的内容来重新配置 FPD。基于 SRAM 控制的缺点有两个：配置是易失性的，必须在上电时重新加载；并且将有源晶体管用于开关会略微增加这种连接的电阻。

**反熔断**：集成电路中的一种结构，当对其编程后，在两根线路之间形成永久连接。

FPGA 包括逻辑和存储设备，通常采用二维阵列结构，其中划分行和列的通道用于阵列单元之间的全局互连。每个单元都是门和触发器的组合，可以对其进行编程以执行某些特定功能。因为它们基本上都是小型可编程 RAM，所以也被称为**查找表**（LUT）。更新的 FPGA 包含更复杂的构建模块，例如加法器和可用于构建寄存器堆的 RAM 模块。有些 FPGA 甚至包含 64 位 RISC-V 内核！

**查找表**：在现场可编程设备中，单元对应的名字称为 LUT，因为它们由少量逻辑和 RAM 组成。

除了对每个单元进行编程以执行特定功能之外，单元之间的互连也是可编程的，这使得具有数百个模块和数十万个门的现代 FPGA 可用于复杂的逻辑功能。互连是定制芯片的主要挑战，对于 FPGA 来说更是如此，因为阵列单元不代表结构化设计分解后的最小单位。在许多 FPGA 中，90% 的区域用于互连，只有 10% 用于逻辑和存储模块。

正如无法在没有 CAD 工具的情况下设计定制或半定制芯片一样，FPD 同样需要使用 CAD 工具。目前已开发出针对 FPGA 的逻辑综合工具，帮助从结构和行为 Verilog 中使用 FPGA 生成系统。

## A.13 本章小结

本附录介绍了逻辑设计的基础知识。如果你已经消化了本附录中的内容，那么深入阅读第 4 章和第 5 章，这两部分都广泛使用了本附录中讨论的概念。

### 拓展阅读

有许多关于逻辑设计的好书。以下是一些参考。

Ciletti, M. D. [2002]. *Advanced Digital Design with the Verilog HDL*, Englewood Cliffs, NJ: Prentice Hall.

全面介绍使用 Verilog 进行逻辑设计的书。

Katz, R. H. [2004]. *Modern Logic Design*, 2nd ed., Reading, MA: Addison-Wesley.

关于逻辑设计的一本通识书。

Wakerly, J. F. [2000]. *Digital Design: Principles and Practices*, 3rd ed., Englewood Cliffs, NJ: Prentice Hall.

关于逻辑设计的一本通识书。

## A.14 练习

**A.1** ［10］<A.2> 除了在本章中讨论的基本定理之外，还有两个重要的定理，称为德摩根定律：

$$\overline{A+B}=\overline{A}\cdot\overline{B} \qquad \overline{A\cdot B}=\overline{A}+\overline{B}$$

用真值表证明德摩根定律：

| A | B | $\overline{A}$ | $\overline{B}$ | $A+B$ | $\overline{A+B}$ | $\overline{A}+\overline{B}$ | $\overline{A}\cdot\overline{B}$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

**A.2** [ 15 ] <A.2 > 使用德摩根定律和 A.2 节所示的定理，证明 A.2.2 节例题中关于 $E$ 的两个等式是等价的。

**A.3** [ 10 ] <A.2 > 证明，对有 $n$ 个输入的函数，其对应的真值表中有 $2^n$ 项。

**A.4** [ 10 ] <A.2 > 逻辑函数异或具有多种功能（包括用于加法器和计算奇偶校验）。只有当其中一个输入为真时，二输入异或函数的输出才为真。给出二输入异或函数的真值表，并使用与门、或门和反相器实现此函数。

**A.5** [ 15 ] <A.2 > 通过使用二输入或非门实现与、或、非功能，证明利用或非门可以实现各种逻辑功能。

**A.6** [ 15 ] <A.2 > 通过使用二输入与非门实现与、或、非功能，证明利用与非门可以实现各种逻辑功能。

**A.7** [ 10 ] <A.2, A.3 > 构造四输入奇校验函数的真值表（有关奇偶校验的说明，请参阅 A.9.3 节）。

**A.8** [ 10 ] <A.2, A.3 > 使用带有反向输入和输出的与门和或门实现四输入的奇校验函数。

**A.9** [ 10 ] <A.2, A.3 > 使用 PLA 实现四输入的奇校验函数。

**A.10** [ 15 ] <A.2, A.3 > 通过使用多选器构建与非门（或者或非门），证明二输入的多选器同样可以实现各种逻辑功能。

**A.11** [ 5 ] <4.2, A.2, A.3 > 假设 $X$ 由 3 位组成，分别为 $x2$、$x1$、$x0$，写出下面 4 个逻辑表达式：

- $X$ 中只有一个 0。
- $X$ 中有偶数个 0。
- 当 $X$ 被当作无符号二进制数时小于 4。
- 当 $X$ 被当作有符号数（补码）时为负。

**A.12** [ 5 ] <4.2, A.2, A.3 > 使用 PLA 实现上题中的四个逻辑表达式。

**A.13** [ 5 ] <4.2, A.2, A.3 > 假设 $X$ 由 3 位组成，分别为 $x2$、$x1$、$x0$，$Y$ 由 3 位组成，分别为 $y1$、$y2$、$y3$，写出下面 3 个逻辑表达式：

- 当 $X$ 和 $Y$ 被当作无符号二进制数时，$X< Y$。
- 当 $X$ 和 $Y$ 被当作有符号二进制数（补码）时，$X< Y$。
- $X=Y$。

**A.14** [ 5 ] <A.2, A.3 > 实现具有两个数据输入（$A$ 和 $B$）、两个数据输出（$C$ 和 $D$）和控制输入（$S$）的开关网络。如果 $S = 1$，则网络为直通模式，且 $C = A$，$D = B$。如果 $S = 0$，则网络为交叉模式，且 $C = B$，$D = A$。

**A.15** [ 15 ] <A.2, A.3 > 由 A.3.3 节中 $E$ 的“合取范式”推导出“析取范式”形式，需要用到德摩根定律。

**A.16** [ 30 ] <A.2, A.3 > 设计一个算法，对于由与门、或门和非门组成的任意逻辑表达式，可给出其“积的和”表示。算法应该是递归的，且在过程中不能产生真值表。

**A.17** [ 5 ] <A.2, A.3 > 给出一个多选器的真值表（输入为 $A$、$B$ 和 $S$，输出为 $C$），可以使用无关项简化真值表。

**A.18** [ 5 ] < A.3 > 下面的 Verilog 模块实现了什么功能：

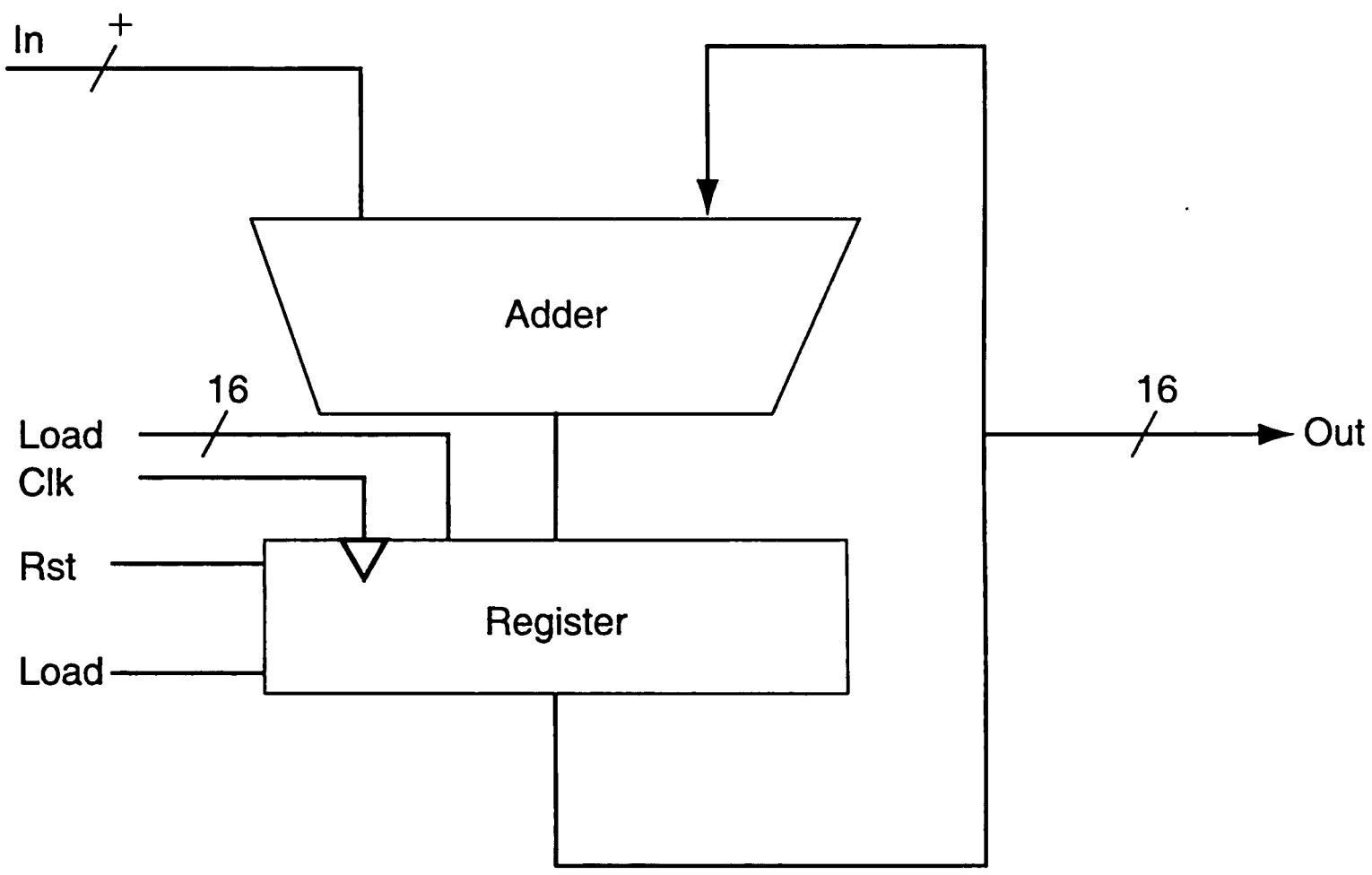
```
module FUNC1 (I0, I1, S, out);
      input I0, I1;
      input S;
      output out;
      out = S? I1: I0;
endmodule

module FUNC2 (out,ctl,clk,reset);
      output [7:0] out;
      input ctl, clk, reset;
      reg [7:0] out;
      always @(posedge clk)
      if (reset) begin
                  out <= 8'b0 ;
      end
      else if (ctl) begin
                  out <= out + 1;
      end
      else begin
                  out <= out - 1;
      end
endmodule
```

**A.19** [ 5 ] < A.4 > 根据 A.8.1 节 D 触发器的 Verilog 代码，写出 D 锁存器的 Verilog 代码。

**A.20** [ 10 ] < A.3, A.4 > 写出 2-4 译码器（与 / 或编码器）的 Verilog 模块实现。

**A.21** [ 10 ] < A.3, A.4 > 根据下面给出的累加器逻辑图，写出它的 Verilog 模块实现。假设使用正边沿触发寄存器和异步 Rst。




**A.22** [ 20 ] < 3.3, A.4, A.5 > 3.3 节介绍了乘法器的基本操作及其实现。这种实现的基本单元是移位加法单元。给出此单元的 Verilog 实现，并说明如何使用此单元构建 32 位乘法器。

**A.23** [ 20 ] < 3.3, A.4, A.5 > 根据上一题，实现一个无符号数的除法器。

**A.24** [ 15 ] < A.5 > ALU 支持仅使用加法器的符号位设置小于（slt）。用该方法比较 $7_{10}$ 和 $6_{10}$。简单起见，将二进制表示限制为 4 位：$1001_2$ 和 $0110_2$。

$$1001_2 - 0110_2 = 1001_2 + 1010_2 = 0011_2$$

这个结果表明 $-7 > 6$，这显然是错误的。因此，判断时必须考虑溢出。修改图 A-5-10 中的 1

位 ALU 以正确处理 slt。可在此图的副本上进行更改以节省时间。

**A.25** [20] <A.6> 做加法时检查溢出的简单方法是查看最高有效位的 CarryIn 是否与最高有效位的 CarryOut 不同。证明该方法与图 3-2 所示的相同。

**A.26** [5] <A.6> 使用新表示法改写 16 位加法器的超前进位逻辑公式。首先，使用加法器各个位的 CarryIn 信号的名称。也就是说，使用 c4，c8，c12，…代替 C1，C2，C3，… 另外，$P_{i,j}$ 表示 $i$ 位到 $j$ 位的传播信号，$G_{i,j}$ 表示 $i$ 位到 $j$ 位的生成信号。例如，公式

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

可被改写为

$$c8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c0)$$

这种更通用的表示法在创建位数更宽的加法器时很有用。

**A.27** [15] <A.6> 使用上题中的新表示法写出 64 位加法器的超前进位逻辑表达式，以 16 位加法器作为构建模块。在你的解答中给出类似于图 A-6-3 的图。

**A.28** [10] <A.6> 下面计算加法器的相对性能。假设对应于某个公式的硬件运行时间为一个时间单位 $T$，该公式只包含或运算或与运算，例如 A.6.2 节的 p*i* 和 g*i* 的公式。由若干个与项进行或运算构成的公式运行时间为 $2T$，例如 A.6.2 节的 c1、c2、c3 和 c4。该时间包括需要 $T$ 时间产生与项，以及另一个 $T$ 时间来生成或运算的结果。分别计算 4 位行波进位加法器和超前进位加法器的运算次数和性能的比。如果公式中的项由其他公式定义，则为这些中间公式增加适当的时延，并递归地继续，直到加法器的实际输入位用于公式中。标记每个加法器的计算时延，并标明最坏情况时延的路径。

**A.29** [15] <A.6> 本题与上题类似，但这次只计算使用行波进位的 16 位加法器的相对速度，4 位超前进位组组成行波进位，超前进位使用 A.6.2 节的机制。

**A.30** [15] <A.6> 本题与上两题类似，但这次只计算使用行波进位的 64 位加法器的相对速度，4 位超前进位组组成行波进位，16 位超前进位组组成行波进位，超前进位使用 A.27 中的机制。

**A.31** [10] <A.6> 我们可以将加法器视为可以将三个输入 $(a_i, b_i, c_i)$ 相加并产生两个输出 $(s, c_{i+1})$ 的硬件设备（而不是将两个数相加并将进位连接在一起的设备）。当将两个数相加时，该想法可能没什么作用。但当对两个以上操作数进行相加时，就可以降低进位的开销。这个想法是构造两个独立的和，称为 $S'$（和）和 $C'$（进位）。在过程的末尾，需要使用普通加法器将 $C'$ 和 $S'$ 相加。这种将进位传播延迟到加法最后阶段的技术称为*进位保留加法*。图 A-14-1 右下方的方框图给出了该结构，其中两级的进位保留加法器通过一个普通加法器连接在一起。

对于 4 个 16 位数的加法运算，分别计算利用完全超前进位加法器和带有超前进位加法器（用于形成最终累加和）的进位保留加法器的时延。（时间单位 $T$ 与 A.28 中的相同。）

**A.32** [20] <A.6> 在计算机中最可能一次对多个数相加的情况是，试图在一个时钟周期内通过使用许多加法器将多个数相加来加快乘法操作的速度。与第 3 章中的乘法算法相比，带有许多加法器的进位保留机制可以快 10 倍以上。本题对使用组合逻辑乘法器计算两个 16 位正数乘法的开销和速度进行评估。假设有 16 个中间项 M15，M14，…，M0，称为*部分积*，它们分别表示被乘数与乘数的每一位（m15，m14，…，m0）与运算的结果。我们的想法是使用进位保留加法器将 $n$ 个操作数减少到 $2n / 3$ 并行组，每组 3 个，并重复执行此操作，直到获得两个大数，最后用传统加法器对二者相加。

首先，根据图 A-14-1 右侧所示，画出 16 位进位保留加法器的结构组织，用来实现 16 个部分积相加。然后计算将这 16 个数相加的时延。将该时间与第 3 章中的迭代乘法方案进行比较，但仅假设使用的是具有完全超前进位的 16 位加法器进行 16 次迭代，其速度在 A.29 中已计算过。