

逻辑设计基础

我一直很喜欢这个词：布尔。

——Claude Shannon, IEEE Spectrum, April 1992

(Shannon 的硕士论文表明，由 George Boole 在 19 世纪初发明的代数可以代表电器开关的工作原理。)

B. 1 引言

本附录仅对逻辑设计的基本原理进行了讨论，无法替代逻辑设计的课程，也不能保证你可以设计出可以很好工作的逻辑系统。如果你很少接触，甚至从没接触过逻辑设计，本附录将提供足够的背景知识，让你了解本书中提到的内容。另外，本附录将帮助你了解计算机内部的实现机制。如果你对该部分内容感兴趣，附录后面的参考文献还可以为你提供更多的信息。

B. 2 节介绍了逻辑块中的基本单元：门。B. 3 节中使用这些逻辑块来构建一个不含存储器的简单组合逻辑。如果你对逻辑电路或数字电路有所了解，前两部分将不会感到陌生。B. 5 节讲述了怎样利用 B. 2 节和 B. 3 节的概念设计一个 MIPS 处理器的 ALU。B. 6 节讲述了怎样设计一个快速加法器，如果对该部分不感兴趣，直接跳过即可。B. 7 节简单介绍了时钟，如果想知道存储器如何工作，必须对时钟有所了解。B. 8 节介绍了存储器的基本单元，B. 9 节介绍了随机存取存储器，这两部分不仅介绍了存储器的特点，而且讲述了构建多层次存储体系的背景知识。其中了解存储器的特点对如何使用存储器有很大帮助，详细内容在第 4 章介绍，构建存储体系在第 5 章进行介绍。B. 10 节介绍了如何设计和使用时序逻辑块——有限状态机。如果你要掌握附录 D 的内容，那么你需要了解 B. 2 ~ B. 10 节所有的内容。如果你只要掌握第 4 章的知识，则可以直接跳到 B. 11 节。B. 11 节是为需要深入了解时钟方法和时序的人准备的，这一部分介绍了边缘触发时钟的工作原理，引入了另一种时钟策略，并且介绍了异步输入的同步问题。

在附录 B. 4 节中，介绍了怎样用 Verilog 来描述逻辑电路。同时，在 CD 中可以找到更加广泛和完整的 Verilog 教程。

B. 2 门、真值表和逻辑方程式

现代计算机的内部电路为数字电路。数字电路仅工作在两个电压：高电压和低电压。其他所有的电压值均为瞬时值，出现在电压值间的过渡阶段。（正如后面要讲到的，数字电路设计中可能存在一个陷阱，当无法确定电压值属于高电压还是低电压时，则对该电压进行采样。）计算机采用二进制数也是促使其采用数字电路进行设计的一个重要原因，因为二进制可以和数字电路中的底层抽象相匹配。在不同的逻辑大家庭中，两个电压值间的关系和对应的值均有所不同。因此，我们不去谈论电压值的高低，而只需考虑电压值对应的逻辑值信号 0 和 1。其中逻辑 1 也称为“真”或有效信号（asserted signal），逻辑 0 也称为“假”或无效信号（deasserted signal）。逻辑 0/1 也称为 1/0 的补逻辑值或反逻辑值。

- 有效信号：信号为逻辑 1 或真。
- 无效信号：信号为逻辑 0 或假。

B-4

根据是否包含存储器件，逻辑电路被分为两大类。不包含存储器件的逻辑电路称为组合逻辑，组合逻辑的输出只取决于当前的输入。而包含有存储器件的电路中，输出不仅与当前的输入有关，而且与存储器件中存储的值有关，将存储器件中存储器的值称为逻辑电路的状态。在 B.2 节和 B.3 节中，我们只介绍组合逻辑（combinational logic）。在 B.8 节中介绍完各种存储元件后，我们再介绍包含电路状态的时序逻辑（sequential logic）。

- 组合逻辑：组合逻辑不包含存储元件，因此当输入相同的数据时，将得到相同的输出。
- 时序逻辑：时序逻辑包含有存储元件，因此输出取决于输入和当前存储元件的内容。

B.2.1 真值表

由于组合逻辑不包含存储元件，因此可以对每一个可能的输入集，定义对应的输出值，通过这种方法就可以指定一个逻辑电路。通常我们用真值表来描述组合逻辑。对一个包含 n 个输入的组合电路来说，有 2^n 种可能的输入组合，因此真值表中有 2^n 项。真值表中的每一项都指定了特定输入对应的所有输出的数值。

01 例题·真值表

假设一个逻辑函数包含三个输入 A 、 B 、 C 和三个输出 D 、 E 、 F 。函数的定义如下：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；如果三个输入都为真，则 F 为真。请写出该函数的真值表。

01 答案

真值表包含 $2^3 = 8$ 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

□

真值表可以描述任意的组合逻辑函数，但是真值表中的项数随着输入的增加将变得很大，同时真值表也不容易理解。有时，我们需要构造一个逻辑函数，其中很多输入组合均为 0，此时我们可以只描述非 0 的输出组合。这种方法在第 4 章和附录 D 中使用。

B-5

B.2.2 布尔代数

描述组合逻辑函数的另一种方法是使用逻辑方程式，这可以通过使用布尔代数（以 19 世纪数学家布尔的名字命名）来完成。在布尔代数中，所有的变量均取值为 0 或 1，在典型的表达式中，包含如下三种操作符：

- 或操作可以记为 $+$ ，例如 $A + B$ 。或操作的结果如下：如果任意一个变量为 1，则或操作的结果为 1。由于任一变量为 1，或操作结果都为 1，因此或操作也被称为逻辑和。

- 与操作可以记为 \cdot ，例如 $A \cdot B$ 。只有当所有输入均为 1 时，与操作的结果才为 1。由于所有输入为 1 时，与操作结果才为 1，因此与操作也称为逻辑乘。
- 非操作可以记为 \bar{A} ，如果输入为 0，则非操作的结果为 1。非操作将会对逻辑值进行取反操作（如果输入为 0，则输出为 1，反之亦然）。

布尔代数中有几条定律，对逻辑方程式的操作很有帮助。

- 同一定律： $A + 0 = A, A \cdot 1 = A$
- 0 和 1 定律： $A + 1 = 1, A \cdot 0 = 0$
- 互补定律： $A + \bar{A} = 1, A \cdot \bar{A} = 0$
- 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： $A \cdot (B + C) = (A \cdot B) + (A \cdot C), A + (B \cdot C) = (A + B) \cdot (A + C)$

另外，还有两条很有用的定律，称为德·摩根定律，德·摩根定律将在练习题中进行深入介绍。

任何逻辑方程式组都可以写成一系列的逻辑等式，其中等式的左边为输出，等式右边为变量及上述三种操作符的组合。

B-6

01 例题·逻辑等式

请写出上个例题中逻辑函数 D 、 E 、 F 的逻辑等式。

01 答案

D 的逻辑等式为：

$$D = A + B + C$$

F 的逻辑等式为：

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

逻辑函数 E 需要一点技巧。将其分为两部分： E 肯定为真的情况（三个输入中的两个必须为真）， E 肯定不会为真的情况（三个输入都不能为真）。由此 E 的逻辑等式可以描述为：

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

我们也可以通过另一种方法得到 E 的逻辑等式。考虑到只有当两个输入为真时， E 才为真，因此我们可以将 E 写成三个式子的或操作，其中每个式子为两个输入为真，一个输入为假的与操作，如下所示：

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

可以在练习题中验证两个逻辑等式是等价的。

在 Verilog 中，我们通过赋值声明来描述组合逻辑，这部分将在 B.4 节中进行描述。我们可以通过 Verilog 中的异或操作来定义 E ：assign $E = (A \wedge B \wedge C) * (A + B + C) * (A * B * C)$ ，这也是一种表示逻辑函数的方法。 D 和 F 的定义就更加简单了，与 C 语言的差别不大，如下所示：D = A | B | C, F = A & B & C。

□

B-7

B.2.3 门

逻辑块是由可以实现基本逻辑功能的门（gate）构成的。例如，一个与门可以实现与操作，或门可以实现或操作。因为与和或操作是可交换、可结合的操作，因此与门、或门可以有多种输入，输出为所有输入的与、或操作。非操作通过一个反向器实现，反向器只有一个输入。这三种逻辑门的标准表示形式如图 B-2-1 所示。

- 门：实现基本逻辑功能的硬件设备，比如与门、或门。

在描述非门时，更常见的形式并不是明确的画出反向器，而是在需要取反的输入或输出中加一个“气泡”（即小圆圈）。如图 B-2-2 所示，对于逻辑操作 $\overline{A + B}$ ，左侧为使用反向器的表示形式，右侧为使用“气泡”的表示形式。

任何逻辑函数都可以通过与门、或门和非门来实现，有几个练习题要求使用门来实现一些通用逻辑函数。下一节中，我们将介绍如何通过这些门来实现任意的逻辑函数。



图 B-2-1 从左到右，依次为与门、或门、非门的标准表示形式。每个门的左侧信号为输入信号，右侧信号为输出信号。与门和或门有两个输入信号，非门只有一个输入信号

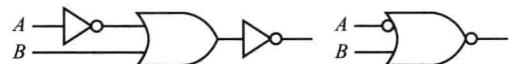


图 B-2-2 用逻辑门实现 $\overline{A + B}$ ，左侧的输入输出均明确地画出了反向器，右侧则使用了“气泡”。该逻辑函数可以简化为 $A \cdot \bar{B}$ ，或使用 Verilog 来表示 $A \& \sim B$

事实上，所有逻辑函数都可以通过单一的门来实现，只要这种门是反相的。两种常见的门为或非门（NOR gate）和与非门（NAND gate），其中或非门是对或门的输出进行取反操作，与非门是对与门的输出进行取反操作。或非门和与非门称为万能门，因为任何逻辑函数都可以通过其中的一种门来实现。下面的练习题将进一步探索这种观点。

- 或非门：或门的输出进行取反。
- 与非门：与门的输出进行取反。

01 小测验

下面的两个逻辑表达式等价吗？如果不等价，证明它们不等价。

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

B-8

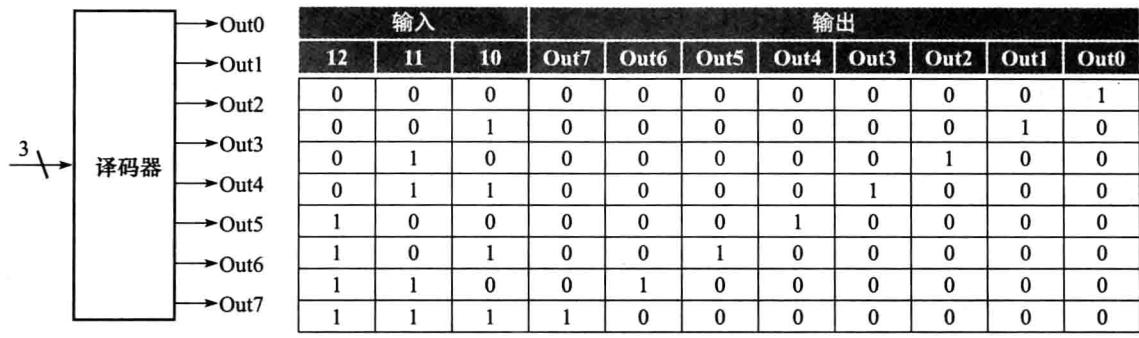
B.3 组合逻辑

本节将介绍经常用到的较大的逻辑块。同时，我们将讨论结构化逻辑块的设计，这些逻辑块可以通过一种翻译程序，自动由逻辑等式或真值表来实现。最后，我们将讨论逻辑块组成的阵列。

B.3.1 译码器

在设计大型逻辑单元中，译码器（decoder）是经常用到的一种逻辑块。最常见的译码器有 n 个输入， 2^n 个输出，对每一种输入组合，只有一个输出信号置为 1。译码器将输入的 n 位数据转化为该数据对应的二进制形式。译码器的输出常通过数字来标记，如 Out0、Out1…Out $2^n - 1$ 。如果输入数据对应的值为 i ，则 Out i 被置为 1，其他所有的输出信号均为 0。图 B-3-1 为一个 3 位的译码器及对应的真值表。由于这种译码器有 3 个输入和 8 个输出，因此也成为 3-8 译码器。相对于译码器，编码器的功能正好相反，编码器有 2^n 个输入和 n 个输出。

- 译码器：拥有 n 位输入和 2^n 输出的逻辑块。对每一种输入组合，只有一个输出信号为真。



a) 3位译码器

b) 3位译码器的真值表

图 B-3-1 一个 3 位译码器包含 3 个输入 (12、11、10) 和 8 个输出 (Out0 ~ Out7)。正如真值表所示，只有与输入信号对应的二进制数据相关的输出被置为 1。译码器输入端的 3 表示输入信号为 3 位位宽

B-9

B.3.2 多路选择器

第 4 章中，我们经常用到一个逻辑块：多路选择器。多路选择器之所以称为选择器，是因为它的输出来自输入信号中的某一个，而到底来自哪一个输入信号由控制信号决定。下面考虑两输入多路选择器。图 B-3-2 中，左侧包含三个输入：两个数据信号和一个选择（控制）信号 (selector (control) value)。其中控制信号决定哪一个输入信号将成为输出信号。图 B-3-2 右侧的两输入多路选择器对应的逻辑函数为 $C = (A \cdot S) + (B \cdot \bar{S})$ 。

② 选择信号：也称为控制信号。控制信号用来选择某一个输入信号，来作为多路选择器的输出信号。

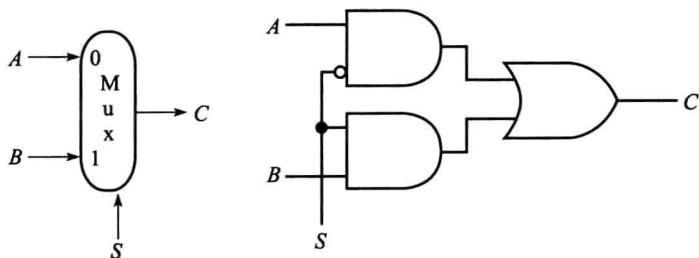


图 B-3-2 左侧为两输入的多路选择器，右侧为对应的实现。多路选择器包含两个输入 (A 和 B)，分别标记为 0 和 1，并且包含一个选择输入信号 (S) 和一个输出信号 (C)。用 Verilog 来实现多路选择器需要较多的工作量，尤其是当输入信号数量大于 2 时，我们将在 B.4 节中进行介绍

多路选择器可以有任意数量的输入信号。如果多路选择器只有两个输入信号，则当选择信号为 1 时，选择其中的一个输入作为输出；如果选择信号为 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，此时的多路选择器包含三个部分：

- 1) 生成 n 个信号的译码器，每一个信号代表一个不同的输入信号。
- 2) n 个与门阵列，每个与门都包含 n 个信号中的一个。
- 3) 一个较大的或门，用来将与门的输出进行合并。

为了将输入信号与控制信号联系起来，我们经常将输入数据用数字进行标记（如 0，

1, 2, …, $n - 1$), 同时将控制信号转化为二进制形式。有时, 我们也使用未解码的选择信号。

B-10 在 Verilog 中, 通过 if 语句可以很简单地实现多路选择器。对于大型的多路选择器, 使用 case 语句将更加方便, 但是在对组合逻辑进行综合的时候, 需要十分小心。

B.3.3 两级逻辑和 PLA

如上一节所述, 任何逻辑函数都可以通过与、或和非门实现。事实上, 还有更加规整的实现方法。任意的逻辑函数都可以描述成规范形式, 即输入信号要么为真, 要么为假, 并且只有两级门: 与门和或门, 如果需要, 可以在最后的输出中加一个反向器。这类表示法称为两级表示法, 它有两种形式: 乘积和 (sum of products)、和项积。乘积和表示所有乘积 (即与操作) 的逻辑和 (即或操作); 和项积正好相反。在前面的例子中, 输出 E 有两种形式:

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

和

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

其中第二个表达形式即为乘积和: 它包含两级逻辑, 并且非操作只发生在单个变量上面。第一个表达形式包含三级逻辑。

◎ 乘积和: 一种逻辑表达形式, 即对所有乘积 (由与操作实现) 进行逻辑求和 (或操作)。

01 精解 我们也可以将 E 写成和项积的形式:

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

为了得到这种表达形式, 需要使用德·摩根定律, 德·摩根定律在练习题中讨论。

在这本书里, 我们使用乘积和的形式。显而易见, 对于任何逻辑函数来说, 我们都可以从它的真值表中构造出乘积和的形式。真值表中该函数为 1 的表项对应一个乘积项。乘积项为所有输入的乘积或输入取反后的乘积, 是否取反取决于真值表中该变量对应的信号是 1 还是 0。通过一个例子可以更容易理解。

01 例题·乘积和

请写出下面真值表中 D 的乘积和的表达式。

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

01 答案

由于真值表中有 4 个表项对应的 D 为 1, 因此总共有 4 个乘积项, 如下:

$$\begin{aligned} & \bar{A} \cdot \bar{B} \cdot C \\ & \bar{A} \cdot B \cdot \bar{C} \\ & A \cdot \bar{B} \cdot \bar{C} \\ & A \cdot B \cdot C \end{aligned}$$

由此，我们可以写出 D 的乘积项和的形式：

$$D = \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot C$$

需要注意到真值表中只有 D 为 1 的项，才能生成对应的乘积项。

可以利用真值表和两级门表示方法之间的关系，为任何逻辑函数生成一个门级的实现。一个真值表对应的逻辑函数集包含多个输出列，正如在 B.2 节中看到的一样。每一个输出列都对应一个不同的逻辑函数，都可以直接从真值表中构造出来。

乘积和的表示方法对应一种常见的称为可编程逻辑阵列（Programmable Logic Array, PLA）的结构化逻辑实现方法。PLA 包含一组输入、输入取反的信号（通过反向器来实现）和两级逻辑。第一个逻辑是一个与门阵列，用来生成乘积项（product term）（也称为最小项（minterm）），每一个乘积项都由输入信号或对应的反向信号构成。第二级为一个或门阵列，每一个或门都生成任意数量的乘积项对应的和。图 B-3-3 显示了 PLA 的基本构成。 □

B-12

- ① 可编程逻辑阵列（PLA）：是一种结构化逻辑单元。PLA 由一组输入信号及反向信号和一个两级逻辑构成。其中第一级逻辑用来生成输入信号和反向信号的乘积项，第二级逻辑用来生成这些乘积项的和。因此，PLA 的逻辑功能为实现乘积项的和。
- ② 最小项：也称为乘积项。由一组输入信号通过与操作形成。乘积项形成了 PLA 的第一级逻辑。

通过多个输入和多个输出，一个 PLA 可以直接实现真值表的功能，其中真值表看作一组逻辑函数。真值表中输出为真时，就产生一个乘积项，PLA 中就需要对应地生成一行。真值表中每一个输出都与或门阵列中潜在的某一行对应。或门的数量与真值表中输出为真的数量相对应。如图 B-3-3 所示，PLA 的大小等于与门阵列和或门阵列的大小之和。通过观察图 B-3-3 可以发现，与门阵列的大小等于输入信号的数量乘以不同乘积项的数量，或门阵列的大小等于输出信号的数量乘以乘积项的数量。

PLA 有两个特点，这两个特点使 PLA 成为实现逻辑函数组的有效方法。首先，真值表的每一项中，至少一个输出为真时，才需要对应的逻辑门。其次，在 PLA 中，不同的乘积项只对应一个输入，即使该乘积项被多个输出使用也不例外。下面让我们来看一个例子。

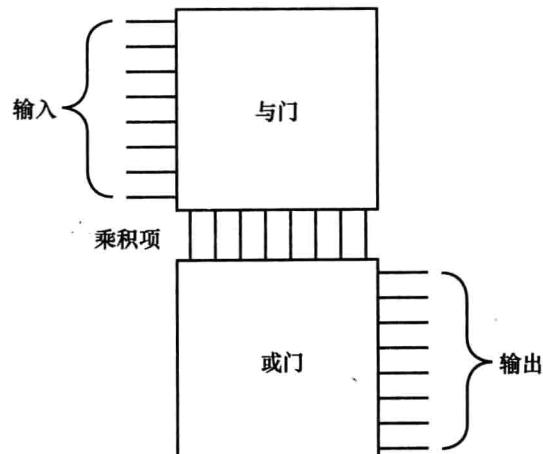


图 B-3-3 PLA 由一个与门阵列和紧跟的或门阵列构成。与门阵列的每一个输入都是若干输入信号或其反向信号的乘积。或门阵列的每一个输入为若干数量的乘积项的和

01 例题 • PLA

考虑 B.2 节中定义的一组逻辑函数。写出 D 、 E 、 F 的 PLA 实现方法。

01 答案

这是我们前面构造的真值表。

B-13

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

由于真值表中至少有一个输出为 1 的表项有 7 个，因此与门阵列将有 7 列。与门阵列中行数为 3 (因为共有 3 个输入信号)，同时或门阵列中也将包含 3 行 (因为共有 3 个输出信号)。图 B-3-4 为最终的 PLA，其中的乘积项与真值表中自顶向下的表项相对应。□

图 B-3-4 将所有的门都画了出来，事实上，设计者常常只画出与门和或门的位置。当乘积项对应的信号线与输入信号或输出信号交叉时，需要使用点来标注。图 B-3-4 中的 PLA 使用这种方法时，结果如图 B-3-5 所示。当 PLA 被创建时，PLA 的功能就固定下来了。也存在类似 PLA 结构的逻辑块，称为 PALs，当设计者需要时，可以通过电子编程的方式来使用 PALs。

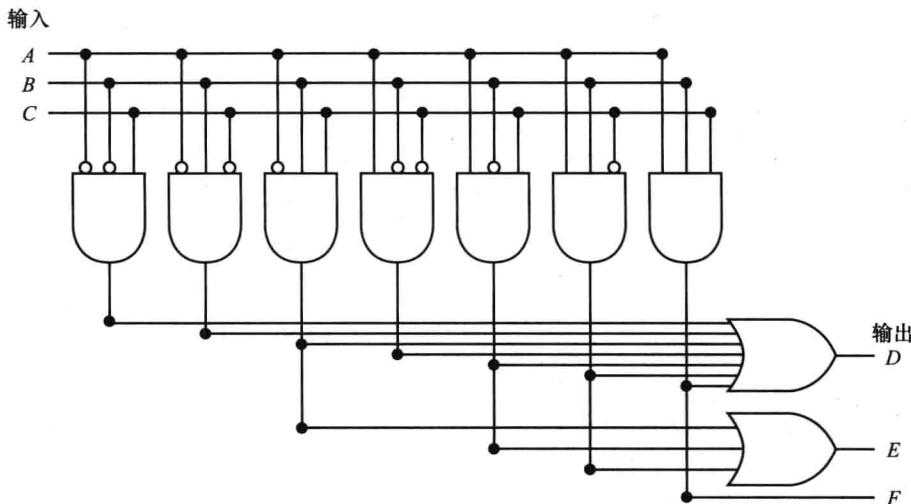


图 B-3-4 例题中逻辑函数对应的 PLA 实现结果

B. 3.4 ROM

另一类可以实现一组逻辑函数的结构化逻辑叫作只读存储器 (Read-Only Memory, ROM)。ROM 被称为存储器是因为它包含一组可以进行读操作的位置，然而，这些位置的内容是固定的，一般在制造的时候就固定下来。除此之外，还有一种可编程只读存储器 (Programmable ROM, PROM)，这类存储器可以电写入。还存在可擦除可编程只读存储器，这类设备需要一个缓慢的擦除过程，过程中需要使用紫外线，因此除了设计和调试外，这类设备只用做只读存储器。

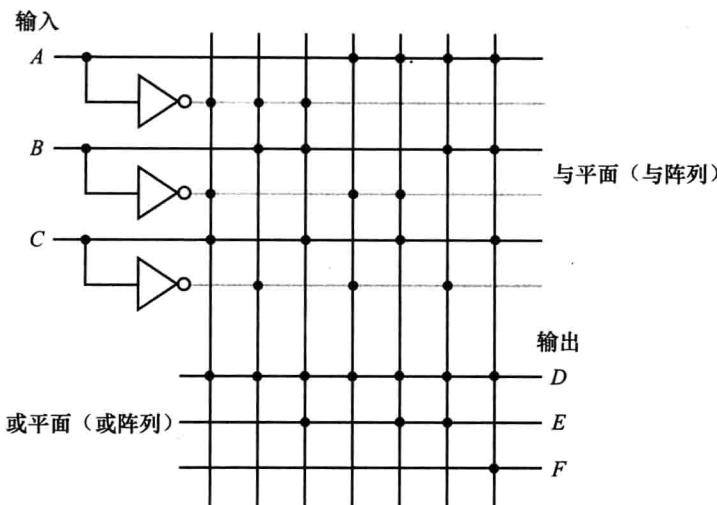


图 B-3-5 在矩阵中，用点来表示乘积项及这些乘积项之和的 PLA 结构。这里，在门的选择上，不是用反相器，而是所有输入信号以实际值及其补码形式连接到与阵列的每个输入上。与阵列中的一个点表示该输入，或其相反值在该乘积项中存在。或阵列中的一个点表示相应的乘积项出现在相应的输出上

- ② 只读存储器 (ROM)：一类存储器，它的数据在制造时就固定下来，之后其数据只能被读。ROM 作为结构化逻辑，可以将逻辑函数组中的项作为输入地址、将输出作为存储器中的一个字，以此来实现逻辑函数组。
- ③ 可编程 ROM (PROM)：一类只读存储器，但是当设计者知道其中的数据时，可以对其进行编程。

ROM 包含一组地址输入线和一组输出。ROM 可寻址的入口数量决定了地址线的数量：如果 ROM 包含 2^m 个可寻址入口（称为高度），则需要 m 条地址线。每一个可寻址入口包含的二进制数量等于输出信号数量，有时也成为 ROM 的宽度。ROM 中总的二进制数量等于高度乘以宽度。有时将高度和宽度统称为 ROM 的形状。B-14

ROM 可以直接通过真值表对逻辑方程式组进行编码。例如，对于有 m 个输入的 n 个方程组来说，ROM 需要 m 条地址线 (2^m 个人口)，其中每一个入口都为 n 位宽。真值表中的输入代表着 ROM 中地址的入口，同时，真值表中的输出代表着 ROM 中存储的内容。当真值表中的输入序列构成一个二进制序列时（正如我们展示的所有真值表一样），ROM 的输出也将是有序的。在 B. 3 节的例子中，共有 3 个输入和 3 个输出，因此 ROM 有 $2^3 = 8$ 个人口地址，每一个入口包含 3 位二进制数据。ROM 中入口对应的数据量可直接由上面例子中的真值表中的输出得到。

ROM 和 PLA 间联系很密切。ROM 是完全译码的：对每一个可能的输入组合，都会输出一个字。而 PLA 是部分译码的。这意味着 ROM 将比 PLA 包含更多的人口。如前面的真值表所示，ROM 包含了所有可能的 8 个输入入口，而 PLA 只包含了 7 个乘积项。随着输入数量的增加，ROM 中的人口数量呈指数增长。与此相反，对实际的逻辑函数来说，乘积项数量的增长很缓慢（参考附录 D 中的例子）。ROM 和 PLA 间的这种差异，使得 PLA 成为实现组合逻辑函数更有效的方法。ROM 的优势在于，当输入、输出数量匹配时，ROM 可以实现任意的逻辑函数。这种优势使得当逻辑函数发生变化时，ROM 中的内容很容易就随之变化，原因在于 ROM 的大小不需要改变。B-15

除了 ROM 和 PLA 外，现代的逻辑综合系统也将小的组合逻辑块转化为一系列门的组合，

自动完成布局布线。尽管这些门的组合占面积较大，但面积的开销仍然比呆板的结构化 ROM 和 PLA 的面积开销要小，因此成为逻辑实现的首选方法。

B-16 对于设计全定制或半定制的集成电路来说，更常用的方法是使用现场可编程器件，我们将在 B. 12 节中讨论这些器件。

B. 3. 5 无关项

在实现组合逻辑时，有时我们并不在乎某些输出的值，其原因可能是另一个输出为真，或者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。因为可以简化逻辑函数的实现，因此无关项很重要。

无关项包含两种类型：无关项的输出和无关项的输入，两者都在真值表中体现出来。当我们对一些输入组合产生的输出不太关心时，就产生了无关项的输出。这类输出在真值表中以 X 代替。当一个输出对于一些输入的组合来说属于无关项时，设计者或逻辑优化程序就可以自由地对这些输入产生的输出赋值为 1 或 0。当输出只取决于一部分输入时，就产生了无关项的输入，在真值表中也记为 X。

01 例题·无关项

考虑一个包含 A、B、C 三个输入的逻辑函数，其定义如下：

- 不管 B 的值为多少，只要 A 或 C 为真，则输出 D 为真。
- 不管 C 的值为多少，只要 A 或 B 为真，则输出 E 为真。
- 虽然 D 和 E 都为真时，我们不关心 F 的值，但是如果三个输入中一个为真，则输出 F 为真。

请写出这个逻辑函数完整的真值表和带有无关项时的真值表。对每一个真值表，PLA 各需要多少个乘积项？

01 答案

下面是不带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

B-17

这个真值表对应的 PLA 需要 7 个乘积项。带有无关项的输出时的真值表如下：

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

(续)

输入			输出		
A	B	C	D	E	F
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

当加入无关项的输入时，真值表可以被进一步简化，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	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 个最小项，或者可以采用一个两输入的与门和三个或门来实现（其中两个或门包含三个输入，另一个包含两个输入）。而原始的真值表需要 7 个最小项，可能需要 4 个与门。 □

逻辑最小化对获得高效的逻辑实现很重要。对任一逻辑进行手工最小化的一个有效工具是卡诺图。卡诺图将真值表以图的形式表示出来，因此可以很容易看出哪些乘积项可以进行合并。但是，由于卡诺图的尺寸和其复杂性，对实际逻辑函数进行手工最小化是不太可能的。幸运的是，逻辑最小化的过程已经高度机械化，可以通过设计工具来完成。在最小化的过程中，设计工具利用了无关项的这个优势，因此，识别出哪些是无关项很重要。附录最后的参考文献中提供了更多的内容，包括逻辑最小化、卡诺图和逻辑最小化算法背后的原理。

B.3.6 逻辑单元阵列

对数据的组合逻辑操作中，很多操作一次需要处理整个字（32 位二进制数）。因此我们常常构建一个逻辑单元的阵列，这可以通过将一个操作作用在整个输入的集合中来实现。大多数时候，我们在机器内部需要在一对总线中进行选择。总线（bus）是若干数据线的集合，这些数据线被当做单一的逻辑信号对待。（名词“总线”也用来表示一组由多个信号源共享使用的信号线。）

● 总线：在逻辑设计中，由多个数据线构成的一条逻辑线，这些数据线被同时执行；被多个源使用的一组信号线也可以称为总线。

例如，在 MIPS 指令集中，指令运行的结果被写入寄存器中，而寄存器中的数据可能有一个或两个来源。此时，需要用一个多路选择器来决定哪一个总线上的数据（32 位）将被写入寄存器中。前面提到的 1 位多路选择器，在这里需要被复制 32 次。

在画图时，为了区分信号线是总线还是一条 1 位的信号线，我们在信号线上画一条较粗的

B-18

线来表示总线。大多数总线都是 32 位宽，如果不是 32 位宽，就明确地写出其位宽。当一个逻辑单元的输入和输出为总线时，意味着逻辑单元必须被复制足够的次数来满足输入的位宽。图 B-3-6 显示了一个多路选择器，这个多路选择器在一对 32 位宽的总线间进行选择。同时，图中也显示了该多路选择器是如何通过 1 位多路选择器实现的。有时，我们需要构造逻辑单元的阵列，其中有些元件的输入来自于前面元件的输出。例如，多位宽的 ALU 就是这样构造的。在这一类例子中，我们必须明确地显示出如何构造更宽的阵列，因为此时阵列中的单个元件并不是独立存在的。正如 32 位宽多路选择器的例子中显示的那样。

B-19

01 小测验

对于奇偶性校验函数来说，它的输出取决于输入中 1 的数量。对于偶校验函数来说，如果输入中 1 的数量为偶数，则输出 1。假设用 ROM 来实现包含 4 位输入的偶校验函数， A 、 B 、 C 、 D 中哪一个可以表示 ROM 中的内容？

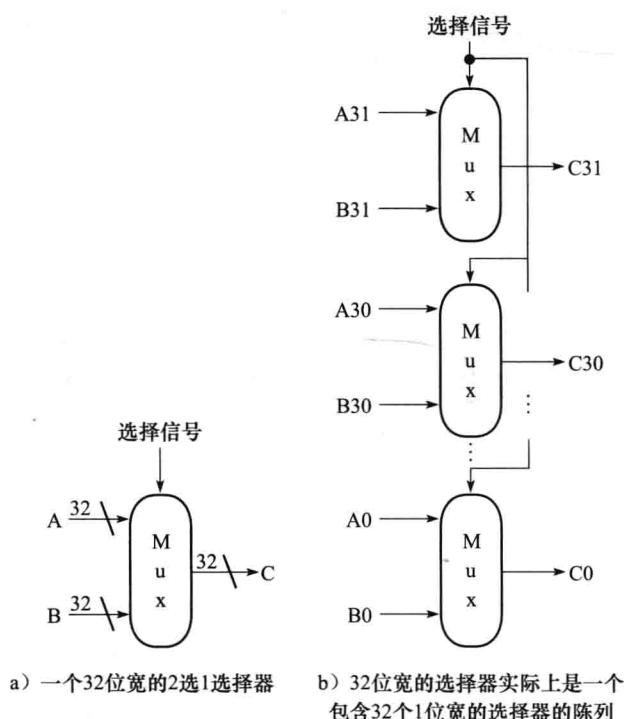


图 B-3-6 为了在两个 32 位宽的输入中进行选择，多路选择器需要被复制 32 次。需要注意，对所有 32 个 1 位多路选择器来说，只使用一位的数据选择信号

地址	A	B	C	D
0	0	1	0	1
1	0	1	1	0
2	0	1	0	1
3	0	1	1	0
4	0	1	0	1
5	0	1	1	0
6	0	1	0	1
7	0	1	1	0
8	1	0	0	1
9	1	0	1	0
10	1	0	0	1
11	1	0	1	0
12	1	0	0	1
13	1	0	1	0
14	1	0	0	1
15	1	0	1	0

B. 4 使用硬件描述语言

当前，处理器和相关硬件系统的设计都是通过硬件描述语言（hardware description language）完成的。硬件描述语言有两个作用。首先，它提供了对硬件的一种抽象描述，通过这种描述可以对设计进行模拟和调试。其次，借助于综合工具和硬件汇编工具，硬件描述语言可以被编译成硬件的实现方法。

- ② 硬件描述语言：一种描述硬件的编程语言，硬件描述语言用来模拟硬件设计，同时也作为综合工具的输入来生成具体的硬件。

本节中，我们将介绍硬件描述语言 Verilog，并介绍如何使用 Verilog 来进行组合逻辑的设计。在附录的其他部分中，我们将 Verilog 的使用扩展到时序逻辑的设计上。在网站上第 4 章选读部分中，我们使用 Verilog 来描述处理器的实现。在网站上第 5 章的选读部分中，我们使用 System Verilog 来描述 cache 控制器的实现。System Verilog 为 Verilog 增加了一些结构和其他有用的特征。

Verilog 是两种基本硬件描述语言中的一种，另一种是 VHDL。Verilog 基于 C 语言，相对基于 Ada 的 VHDL 在工业界的使用更加频繁。对 C 语言比较熟悉的读者会发现，附录中我们用到的 Verilog 基本原理会很容易理解。如果对 C 语言的语法有所了解的话，对 VHDL 比较熟悉的读者将会发现 Verilog 的概念很简单。

- ② Verilog：两种常用硬件描述语言中的一种。
- ② VHDL：两种常用硬件描述语言中的一种。

Verilog 可以将数字系统在行为级和结构级进行描述。行为级（behavioral specification）描述方法描述了数字系统的功能特性。结构级（structural specification）描述方法描述了数字系统的详细组织，并且结构级描述常采用层次描述。结构级描述可以在基本元件的层次结构方面描述硬件系统，比如在门级和开关级。因此，我们可以使用 Verilog 来描述真值表的具体内容和最后章节的数据通路。

- ② 行为级描述：描述一个数字系统在功能方面是怎么工作的。
- ② 结构级描述：描述一个数字系统是如何通过基本元件的层次化连接进行组织的。

随着硬件综合工具（hardware synthesis tool）的出现，大多数设计者都使用 Verilog 或 VHDL，只对数据通路进行结构级描述，之后通过逻辑综合从行为级描述中生成控制系统。另外，大多数 CAD 系统都提供了广泛的标准元件库，如 ALU、多路选择器、寄存器、存储器和可编程逻辑块，当然也包含基本的门电路。

- ② 硬件综合工具：一种计算机辅助设计软件，该软件可以通过数字系统的行为级描述来生成门级的设计结果。

利用库和逻辑综合进行设计时，如果想得到可接受的结果，需要着眼于最终的综合及所需的输出，并据此来写描述语言。对于我们简单的设计来说，需要考虑的就是搞清楚哪些需要用组合逻辑来实现、哪些需要用时序逻辑来实现。在本节及剩余附录中的大部分例子中，写 Verilog 代码时，我们需要将最终的综合放在脑海里。

B-20

B. 4.1 Verilog 的数据类型和操作类型

Verilog 包含两种基本数据类型：

1) **wire** 表示一个组合信号。

2) **reg** (寄存器) 存储一个数据，该数据随着时间的推移而变化。尽管实际实现中 reg 常常与一个寄存器相关连，但并不一定必须这样做。

② wire：在 Verilog 中表示一个组合逻辑信号。

③ reg：在 Verilog 中表示一个寄存器。

假设有一个 wire 或 reg，命名为 X，当 X 为 32 位宽时，可以这样声明：`reg[31:0] X` 或 `wire[31:0] X`，通过最后的索引 0 来划定最低有效位。由于经常访问 reg 或 wire 的子字段，我们可以通过 [起始位: 结束位] 访问 reg 或 wire 的一段连续位，其中的起始位和结束位必须为常数。

reg 的一组阵列可以用来表示寄存器文件或存储器。声明语句

```
reg [31:0] registerfile[0:31]
```

声明了一个类似于 MIPS 的寄存器文件，其中寄存器 0 是第一个寄存器。当访问存储阵列时，我们可以使用 `registerfile[regnum]` 访问一个数据，与 C 语言一样。

Verilog 中 reg 或 wire 型数据可能的取值有：

- 0 或 1，表示逻辑假或真。
- X，表示取值未知，所有寄存器初始化数据、未被连接的 wire 数据均为 X。
- Z，表示三态门处于高阻态，在该附录中不对其进行讨论。

常量可以被指定为十进制、二进制、八进制或十六进制。通常我们需要确切地知道一个常量包含多少二进制位。我们通过在常量前面加一个前缀来表示该常量包含多少二进制位，例如：

- `4'b0100` 表示包含 4 位二进制常量的数据 4，等价于 `4'd4`。
- `{A[31:16],B[15:0]}` 创建了一个数值，其中高 16 位来自 A，低 16 位来自 B。

Verilog 从 C 语言中继承了一元组和二进制操作符，包括算术运算符 (+, -, *, /)、逻辑运算符 (&, |, ~)、比较运算符 (=, !=, >, <, <=, >=)、移位运算符 (<<, >>) 和 C 语言的条件运算符（使用格式为 `condition? expr1:expr2`，当 condition 为真时返回 expr1，否则返回 expr2）。Verilog 中增加了一组逻辑运算符 (&, |, ^)，这类运算符对操作数的所有位均进行逻辑操作。例如，`&A` 返回 A 中所有位进行与操作的结果。`^A` 返回 A 中所有位异或的结果。

01 小测验

下面的定义中，哪些定义了相同的数值？

1. `8'bimoooo`
2. `8'hF0`
3. `8'd240`
4. `{(4{1'b1}),{(4{1'b0})}}`
5. `{4'b1,4'b0}`

B. 4.2 Verilog 程序的结构

Verilog 程序是由模块的组合构成的。这些模块最小可以是一个逻辑门，最大可以是一个完

整的系统。Verilog 中的模块与 C++ 中的类类似，但没有类那样强大的功能。一个模块定义了它的输入和输出，输入和输出分别对应了模块与外部进行连接时的输入接口和输出接口。模块也可能声明一些附加的变量。一个模块的主体有以下几个部分构成：

- initial 结构，该结构对 reg 型变量进行初始化。
- 连续赋值语句，这类语句只出现在组合逻辑中。
- always 结构，该结构既可以用在组合逻辑中，也可以用在时序逻辑中。
- 模块实例化，该结构用来对已经定义的模块进行实例化操作。

B. 4.3 Verilog 构造复杂的组合逻辑

关键字 assign 表示连续赋值语句，连续赋值语句对应的组合逻辑函数为：输出被连续地赋值，并且只要输入的值发生变化，输出的值也马上发生变化。wire 型变量只能通过连续赋值语句进行赋值。通过连续赋值语句，我们可以定义一个模块来实现半加器，如图 B-4-1 所示。

使用 Verilog 来构造组合逻辑时，推荐使用连续赋值语句。但是，当需要构造更复杂的结构时，连续赋值语句将变得很笨拙和乏味。另一种可行方法是使用模块中的 always 语句，来描述组合逻辑单元，但是需要小心使用。always 语句中允许使用控制语句，比如 if-then-else、case 语句、for 语句和 repeat 语句。这些语句与 C 语言中的类似，只有少许变化。

always 语句块指定了一个信号列表，语句块对这些信号敏感（信号列表以 @ 开始）。如果敏感信号列表中任一信号发生变化，always 语句块都将重新执行。如果省略了敏感信号列表，则 always 语句块将一直被不停地重新执行。当 always 语句块表示组合逻辑时，**敏感信号列表 (sensitivity list)** 需要包含所有的输入信号。如果 always 语句块中包含多条 Verilog 语句，这些语句将被关键字 begin 和 end 环绕，就像 C 语言中的 { 和 }。一个 always 块如下所示：

```
always @(list of signals that cause reevaluation) begin
    Verilog statements including assignments and other
    control statements end
```

⑤ 敏感信号列表：一些信号的列表，当这些信号中任一信号发生变化时，always 块都将重新执行。

reg 型变量只能在 always 块内部进行赋值，需要使用过程性赋值语句（与前面介绍的连续赋值不同）。总共有两种不同的过程赋值语句。其中操作符 = 与 C 语言中的类似，右侧语句计算出结果，并赋值给左侧变量。而且与 C 语言的执行一致，在下一个赋值语句执行前，该赋值语句完成执行。因此，操作符 = 被称为**阻塞性赋值 (blocking assignment)**。阻塞性赋值对构造时序逻辑来说很有用，我们很快就会再次介绍它。另一种过程赋值语句为**非阻塞赋值 (nonblocking assignment)**，记为 <=。在 always 块中的非阻塞赋值语句，当所有非阻塞赋值语句计算出右侧的结果时，就立即赋值给对应的左侧变量。图 B-4-2 为 4 选 1 多路选择器的实现，该例子为使用 always 语句块实现的组合逻辑，为了简单化，我们使用了 case 语句。case 语句与 C 语言中的 switch 语句类似。图 B-4-3 显示了 MIPS 中 ALU 的实现，其中也使用了 case 语句。

- ⑤ 阻塞赋值：Verilog 中，在下一个赋值语句执行前，阻塞赋值完成执行。
 ⑤ 非阻塞赋值：一种赋值语句，只有计算出所有非阻塞语句右侧结果时，才进行赋值。

```
module half_adder (A,B,Sum,Carry);
    input A,B; //two 1-bit inputs
    output Sum, Carry; //two 1-bit outputs
    assign Sum = A ^ B; //sum is A xor B
    assign Carry = A & B; //Carry is A and B
endmodule
```

图 B-4-1 使用连续赋值语句定义的一个半加器 Verilog 模块

B-23

由于在 always 块中只能给 reg 型变量赋值，如果希望使用一个 always 块描述组合逻辑，则必须非常小心，以确保这个 reg 变量不被综合为一个寄存器，下面的“精解”中给出了许多“陷阱”。

```
module Mult4to1 (In1, In2, In3, In4, Sel, Out);
    input [31:0] In1, In2, In3, In4; //four 32-bit inputs
    input [1:0] Sel; //selector signal
    output reg [31:0] Out;// 32-bit output
    always @(In1, In2, In3, In4, Sel)
        case (Sel) //a 4->1 multiplexor
            0: Out <= In1;
            1: Out <= In2;
            2: Out <= In3;
            default: Out <= In4;
        endcase
    endmodule
```

图 B-4-2 使用 case 语句实现的 4 选 1 多路选择器，该多路选择器包含 32 位输入。case 语句与 C 语言中的 switch 语句类似，不同之处在于，在 Verilog 中，只有被 case 选择到的语句会被执行（就好像每一个 case 状态后面都加了 break 一样），并且不能转到下一个分支去

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
    input [3:0] ALUctl;
    input [31:0] A,B;
    output reg [31:0] ALUOut;
    output Zero;
    assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
    always @(ALUctl, A, B) //reevaluate if these change
        case (ALUctl)
            0: ALUOut <= A & B;
            1: ALUOut <= A | B;
            2: ALUOut <= A + B;
            6: ALUOut <= A - B;
            7: ALUOut <= A < B ? 1:0;
            12: ALUOut <= ~(A | B); // result is nor
            default: ALUOut <= 0; //default to 0, should not happen;
        endcase
    endmodule
```

图 B-4-3 MIPS 中 ALU 的 Verilog 行为级定义。通过使用包含基本算术和逻辑操作的模块库，就可以对其进行综合

01 精解 组合逻辑中经常使用连续赋值语句，但是在其他一些 Verilog 结构中，逻辑综合中可能会产生想不到的结果，即使是 always 块也有这种问题。最常见的问题是，使用已经存在的锁存器或寄存器来实现时序逻辑，这将导致生成的结果比预期的要慢，并且开销要大。为了保证你设计的组合逻辑可以按这种方式被综合，你需要按下面的注意事项操作：

- 1) 将所有的组合逻辑放在连续赋值语句或 always 块中。
- 2) 保证作为输入的所有信号都出现在 always 块的敏感信号列表中。
- 3) 保证每一个通过 always 块的数据通路，都将值赋给同一位组。

最后一点是最容易被忽略的。请读一下图 B-5-15 中的例子，来说服自己为何要坚持最后这条准则。

01 小测验

假设所有变量都初始化为 0，下面的 Verilog 包含在 always 块中，则执行完下面的语句后，A、B 的值分别为多少？

```
C=1;
A <= C;
B = C;
```

B.5 构建基本的算术逻辑单元

ALU：被所有计算机系统作为标准来使用的一个随机数生成器。

——Stan Kelly-Bootle, 魔鬼词典, 1981

算术逻辑单元 (arithmetic logic unit, ALU) 是计算机的核心，ALU 用来执行算术运算，比如加法和减法，也可以用来执行逻辑运算，比如与操作和或操作。本节中，我们将借助 4 个硬件块（与门、或门、反向器和多路选择器）来构造一个 ALU，并演示组合逻辑是如何工作的。下一节中，我们将展示如何通过更加聪明的设计来加速加法器的操作。

因为 MIPS 中一个字为 32 位宽，因此我们需要一个 32 位宽的 ALU。假设我们使用 32 个 1 位宽 ALU 来构建所需的 ALU。我们将从如何构建 1 位宽 ALU 开始。

B.5.1 1 位 ALU

逻辑操作是最简单的，因为它们直接映射为图 B-2-1 中的硬件元件。

与和或对应的 1 位逻辑单元如图 B-5-1 所示。多路选择器选择是进行 $a \text{ AND } b$ 操作还是 $a \text{ OR } b$ 操作，如何选择取决于 Operation 的值为 0 还是 1。为了与数据信号线进行区分，多路选择器的控制信号画了颜色。需要注意的是，我们需要为多路选择器的控制线和输出线进行重命名，以便反映它们在 ALU 中的功能。

下一个需要加入的函数是加法。一个加法器必须包含两个输入操作数，并输出一位和。同时，需要另外一个输出来传递进位，称为 CarryOut。因为来自相邻加法器的进位是作为输入对待的，因此加法器需要第三个输入。这个输入称为 CarryIn。图 B-5-2 显示了一位加法器的输入和输出。我们知道加法操作的作用是什么，因此我们可以通过输入来指定对应的输出，如图 B-5-3 所示。

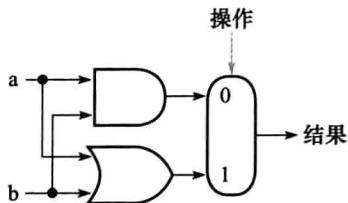


图 B-5-1 与和或的 1 位逻辑单元

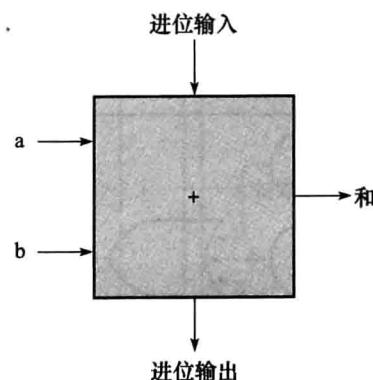


图 B-5-2 1 位加法器。该加法器称为全加器，它也称为 (3, 2) 加法器，因为它有 3 个输入端和 2 个输出端。如果一个加法器只有 a 和 b 两个输入，则称为 (2, 2) 加法器或半加器

输入			输出		注解
a	b	进位输入	进位输出	和	
0	0	0	0	0	$0+0+0=0_2$
0	0	1	0	1	$0+0+1=01_2$
0	1	0	0	1	$0+1+0=01_2$
0	1	1	1	0	$0+1+1=10_2$
1	0	0	0	1	$1+0+0=01_2$
1	0	1	1	0	$1+0+1=10_2$
1	1	0	1	0	$1+1+0=10_2$
1	1	1	1	1	$1+1+1=11_2$

图 B-5-3 1 位加法器输入输出定义

我们可以用逻辑等式的方式来表示输出信号 CarryOut 和 Sum，这些逻辑等式又可以通过逻辑门来实现。以 CarryOut 为例。图 B-5-4 显示了当 CarryOut 为 1 时，对应输入的值。

我们可以将真值表转化为逻辑等式：

$$\begin{aligned} \text{CarryOut} = & (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) \\ & + (a \cdot b) + (a \cdot b \cdot \text{CarryIn}) \end{aligned}$$

如果 $a \cdot b \cdot \text{CarryIn}$ 为真，则剩余的三个乘积项也必然为真，因此我们可以根据真值表的第 4 行将最后一项省略掉。简化后的等式为：

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

图 B-5-5 显示了加法器黑盒子内部的硬件，其中 CarryOut 由 3 个与门和一个或门组成。三个与门分别对应上式中括号内的乘积项，或门用来得到三个乘积项的和。

当有一个输入为 1 或三个输入都为 1 时，Sum 设置为 1。Sum 对应的布尔等式较为复杂（回忆一下， \bar{a} 表示 $\text{Not } a$ ），如下所示：

$$\text{Sum} = (a \cdot b \cdot \text{CarryIn}) + (\bar{a} \cdot b \cdot \text{CarryIn}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

如何画出加法器中 Sum 对应的逻辑，留给读者作为练习。

图 B-5-6 所示的是用之前的部件组成的加法器得到的 1 位 ALU。有时设计人员也希望 ALU 能完成再多一些简单的操作，比如生成 0。最简单增加一个操作的方法是扩大由操作线控制的多路选择器，例如，为了将 0 直接连到扩展的多路选择器的新输入端。

输入		
a	b	进位输入
0	1	1
1	0	1
1	1	0
1	1	1

图 B-5-4 当 CarryOut 为 1 时，各个输入的值

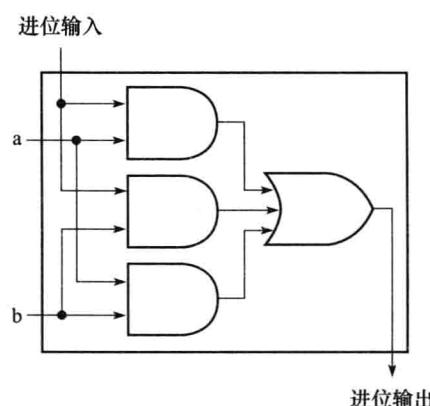


图 B-5-5 加法器中产生 CarryOut 信号所需的硬件。加法器硬件的剩余部分是本页等式中和 (Sum) 的输出逻辑

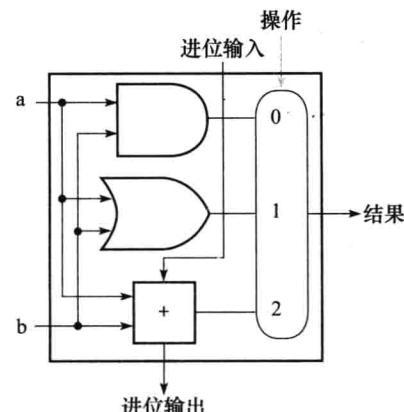


图 B-5-6 完成“与”、“或”和“加法”运算的一个 1 位 ALU (见图 B-5-5)

B. 5.2 32 位的算术运算单元 (ALU)

现在我们已经实现了 1 位的 ALU，由临近的“黑盒子”连接成 32 位的 ALU。用 x_i 表示 x 的第 i 位，图 B-5-7 所示的是一个 32 位的 ALU。犹如一块石头能使一个平静的湖激起涟漪，一个最不重要的进位（结果 0）能通过所有的加法器带来显著的进位（结果 31）。因此，通过直接连接 1 位进位的加法器称为行波进位加法器。从 B.6 节开始我们将看到一种更快连接 1 位加法器的方法。

减法就是将第二个源操作数变相取反后用加法实现。快速求一个二进制数补码的方法是，将这个数按位取反，然后加 1。为了反转每一位，我们只需在 b 和 \bar{b} 之间添加用来选择的 2:1 多路选择器，如图 B-5-8 所示。

假设将 32 个 1 位的 ALU 连接到一起，如图 B-5-7 所示。所添加的多路转换器给出 b 的选项或它的反码，这取决于 Binvert，但是这仅是求二进制数补码的一个步骤。注意到，最低位仍然有一个 CarryIn 信号，即使它对加法是不必要的。如果我们用 1 替代 0 来设置 CarryIn 信号，将会发生什么？加法器会计算 $a + b + 1$ 。通过将 b 取反，就能得到我们想要的结果：

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

一个二进制补码加法器的简单设计有助于解释为什么二进制的补码表示已经成为整数计算机运算的通用标准。

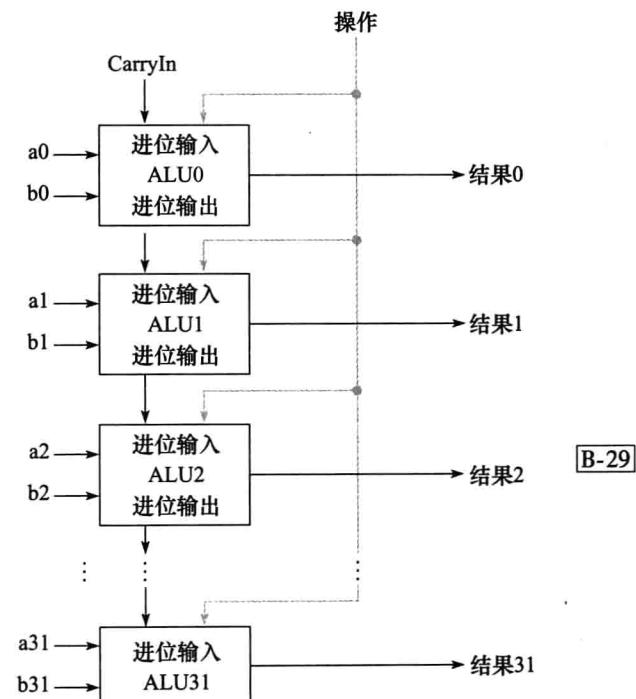


图 B-5-7 由 32 个 1 位 ALU 构成的 32 位 ALU。
不太重要的位的 CarryOut 信号连接到较重要的位的 CarryIn 信号上，这种组成方式称为行波进位

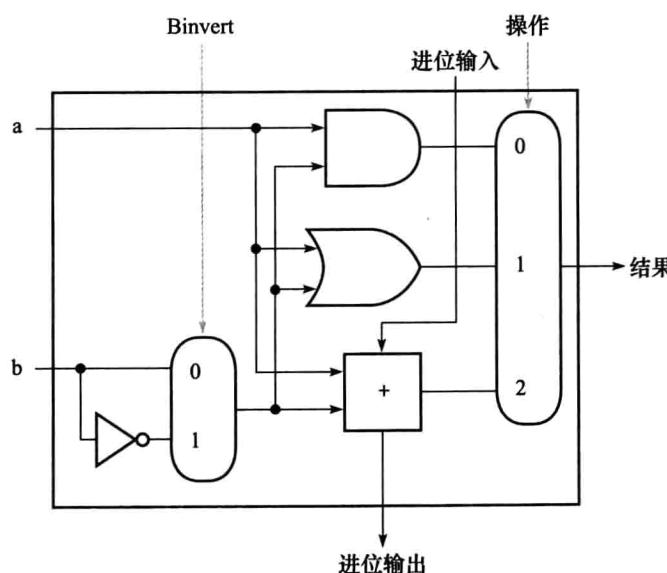


图 B-5-8 一个 1 位的 ALU 执行“与”、“或”以及加法 $a + b$ 和 $a + \bar{b}$ 。通过选择 \bar{b} ($\text{Binvert} = 1$)，并在影响不显著的位上将 CarryIn 设置为 1，得到 b 的二进制补码减法，而不是 b 到 a 的加法

一个 MIPS ALU 需要或非 (NOR) 功能，我们可以通过重复使用 ALU 内部已经有的硬件来实现这种功能，而不是单独增加一个或非门。或非表达式表示如下：

$$(a + b) = \overline{a} \cdot \overline{b}$$

即 a 或 b 的非和非 a 与非 b 是相等的，这也被称为德·摩根定律，在练习题中进行更加深入的探究。

ALU 上已经有了与门和非 b ，只需要再增加非 a ，图 B-5-9 所示的是改变后的结构。

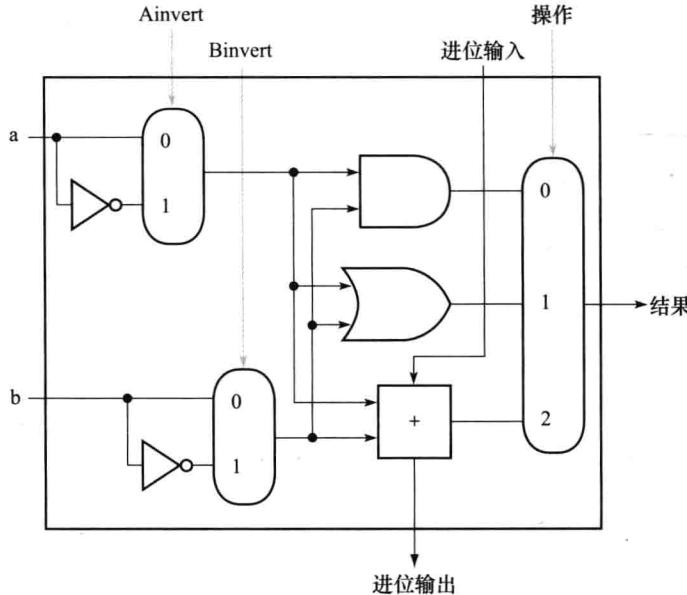


图 B-5-9 1 位 ALU 实现“与”、“或”以及 $a + b$ 或者 $\overline{a} + \overline{b}$ 。通过选择 \overline{a} ($Ainvert = 1$) 和 \overline{b} ($Binvert = 1$)，能得到 $\overline{a} + \overline{b}$ 而不是 $a + b$

B.5.3 将 32 位的 ALU 添加至 MIPS

加法、减法、与、或 4 个操作几乎在每一台计算机里面都能找到，并且大多数的 MIPS 指令都能由 ALU 实现，但是 ALU 的设计不是完美的。

仍然需要支持的一条指令是 `slt` 指令，回想前面的内容，如果 $rs < rt$ ，操作过的记为 1，反之，结果为 0。因此，`slt` 指令根据比较将所有不显著的位都进行设置，但是都将它们设置为 0。为了让 ALU 执行 `slt` 指令，首先需要扩大三输入的多路选择器，在图 B-5-8 中为 `slt` 结果增加一个输入。我们称之为 `Less`，仅在 `slt` 中使用。

图 B-5-10 顶部的图所示的是有扩展的多路选择器的 1 位 ALU。从以上对 `slt` 的描述中，我们得知必须将上面 31 位 ALU 的输入端 `Less` 连接到 0，因为这些位经常被置为 0。剩下要考虑的是在 `slt` 指令中如何比较和设置最不显著的位。

如果用 a 减去 b 会发生什么？如果结果为负值，那么 $a < b$ ，因为

$$\begin{aligned} (a - b) < 0 &\Rightarrow ((a - b) + b) < (0 + b) \\ &\Rightarrow a < b \end{aligned}$$

如果 $a < b$ ，我们就把 `slt` 操作中最最低位设置为 1；也就是说， $a - b$ 为负数时结果为 1，为正时结果为 0。期望值完全对应的标志位置为：1 代表负值，0 代表正值。遵循这个论点，仅需要将加法器输出的标志位连接到最低位上，进而得到 `slt`。

不幸的是，图 B-5-10 顶部的图的 `slt` 操作中，ALU 输出的 Result 的最高位不是加法器的

输出；加法器 slt 操作输出明显是输入值 Less。

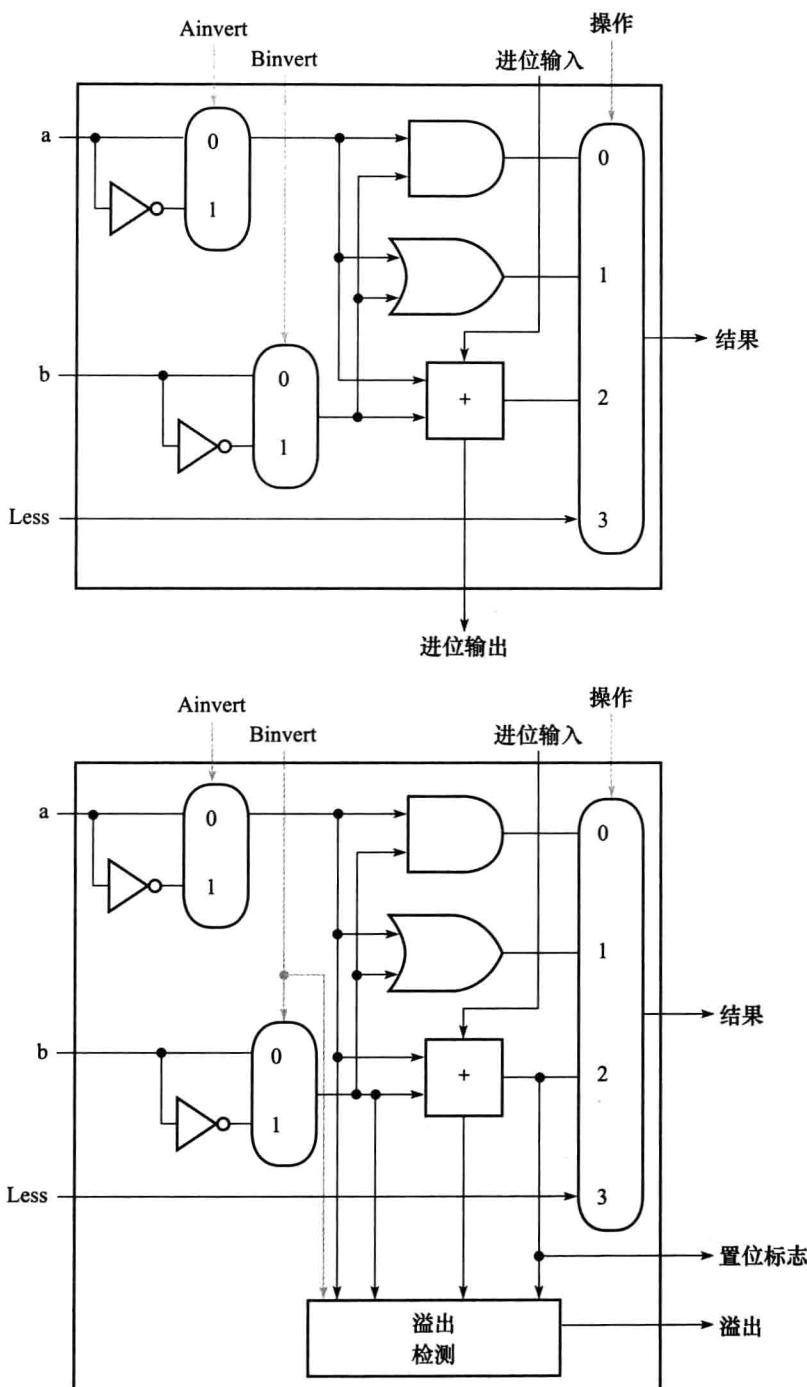


图 B-5-10 (顶部) 1位 ALU 执行“与”、“或”以及加法 $a + b$ 或者 $a + \bar{b}$, (底部) 一个 1 位 ALU 最显著位。顶部的图包括连接到 slt 操作上的直接输入 (见图 B-5-11); 底部的图有 Less 操作的直接输入, 称为 Set (见附录末的练习题 B. 24, 了解如何用较少的输入计算溢出)

因此, 最高位需要一个新的 1 位的 ALU, 它有额外的输出位: 加法器的输出。图 B-5-10 底部的图所示的设计是有新输出 Set 的加法器, 并且仅用在 slt 中。我们只需要一个特殊的最高位, 于是增加了溢出检测逻辑, 因为它和最高位是关联的。

由于溢出问题的存在, slt 的检测比之前描述的要更加复杂, 如在练习题中探究的。

图 B-5-11 所示的是 32 位的 ALU。

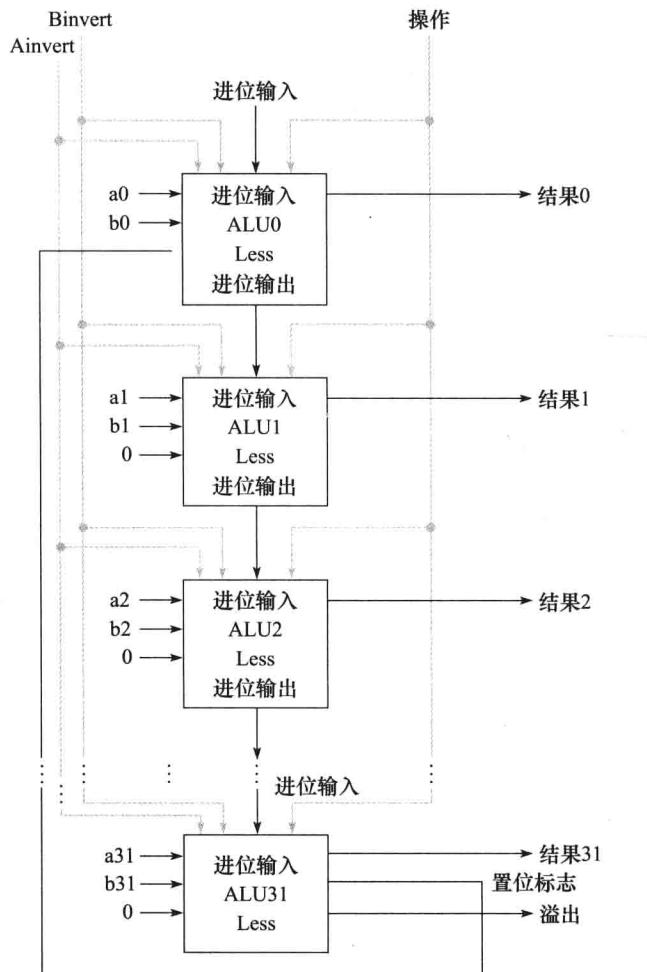


图 B-5-11 一个复制 31 个图 B-5-10 顶部结构和一个图 B-5-10 底部结构组成的 32 位 ALU。除了最低位，Less 输入都连接到 0，最低位连接至 Set 输出端的最高位。如果 ALU 执行 $a - b$ ，我们选择在图 B-5-10 中多路选择器的输入端为 3，如果 $a < b$ ，那么 $Result = 0\cdots001$ ，其他 $Result = 0\cdots000$

注意每次要 ALU 做减法运算，都要将 CarryIn 和 Binvert 信号置为 1。对于加法或者逻辑运算，我们想使控制线均为 0。因此可以通过将 CarryIn 和 Binvert 结合为一根控制线，称为 Bnegate，从而简化了 ALU 的控制。

为了使 ALU 更进一步适合 MIPS 指令集，必须支持条件分支指令。这些指令分支或者是两个寄存器相等，或者它们不相等。最简单的方法是用 ALU 测试 $a - b$ ，并测试其结果是否为 0，因为

$$(a - b = 0) \Rightarrow a = b$$

因此，如果增加硬件来测试结果是否为 0，就能测试是否相等。最简单测试方法是将所有的输出取“或”，然后将结果连接至反相器：

$$\text{Zero} = (\text{Result}31 + \text{Result}30 + \cdots + \text{Result}2 + \text{Result}1 + \text{Result}0)$$

图 B-5-12 所示的是修改后的 32 位 ALU。考虑将 1 位 $Ainvert$ 线、1 位 $Binvert$ 线以及 2 位的 Operation 线作为 4 位的 ALU 控制线结合在一起，执行加法、减法、与、或，或者 `slt` 指令。图 B-5-13 所示的为 ALU 控制线以及相应的 ALU 操作。

最后，我们已经看到一个 32 位的 ALU 的内部结构，我们将用通用的符号表示一个完整的 ALU，如图 B-5-14 所示。

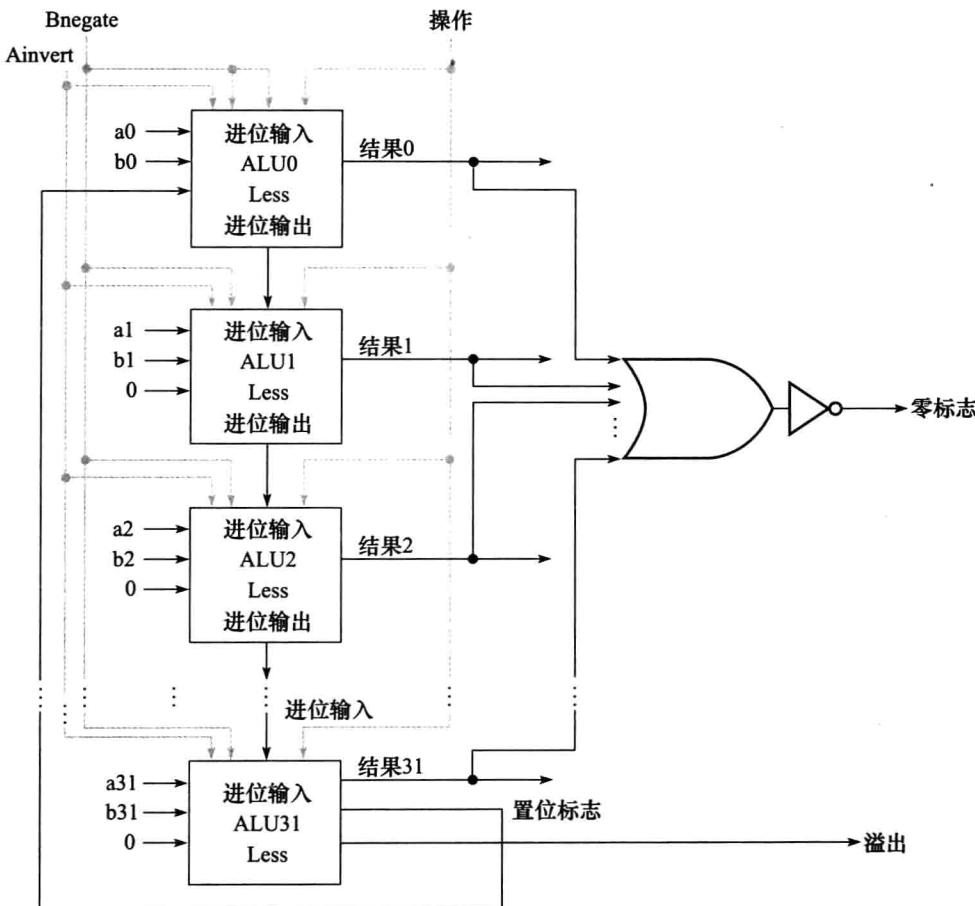


图 B-5-12 最终的 32 位 ALU。在图 B-5-11 的结构上增加了一个 0 检测器

ALU 控制线	功能
0000	与
0001	或
0010	加
0110	减
0111	小于则置位
1100	或非

图 B-5-13 ALU 三个控制线、Bnegate 和 Operation 的值以及对应的操作

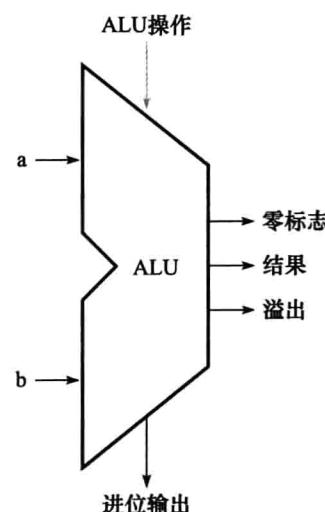


图 B-5-14 图 B-5-12 所示 ALU 的符号图。这个符号也被用来表示加法器，因此通常使用 ALU 或 Adder 标记

B.5.4 用 Verilog 定义 MIPS ALU

图 B-5-15 所示的是如何用 Verilog 定义一个组合 MIPS ALU 的结合；这样的规范可能会用一个加法器的被实例化的标准库编译。为了完整性，我们在图 B-5-16 中展示了 MIPS 的 ALU 控制器（在第 4 章使用过），在这个控制器上我们建立了一个 Verilog 版本的 MIPS 数据通路。

```
module MIPSAU(ALUct1, A, B, ALUOut, Zero);
    input [3:0] ALUct1;
    input [31:0] A,B;
    output reg [31:0] ALUOut;
    output Zero;
    assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
    always @ (ALUct1, A, B) begin //reevaluate if these change
        case (ALUct1)
            0: ALUOut <= A & B;
            1: ALUOut <= A | B;
            2: ALUOut <= A + B;
            6: ALUOut <= A - B;
            7: ALUOut <= A < B ? 1 : 0;
            12: ALUOut <= ~(A | B); // result is nor
            default: ALUOut <= 0;
        endcase
    end
endmodule
```

图 B-5-15 一个 MIPS ALU 的 Verilog 行为定义

下一个问题是，“ALU 将两个 32 位的操作数相加能有多快？”我们能决定 a 和 b 的输入，但是 CarryIn 的输入取决于相邻的 1 位加法器的操作。如果我们通过依赖关系跟踪所有的路径，将最高位连接到最低位上，所有和的最高位必须等待所有 32 个 1 位加法器来依次计算。这种顺序链的反应太慢以至于不能在时间关键的硬件电路中使用。下一节将探究如何加快加法的速度，这个论题对于理解附录的其余部分不是至关重要的，可以跳过。

```
module ALUControl (ALUOp, FuncCode, ALUct1);
    input [1:0] ALUOp;
    input [5:0] FuncCode;
    output reg [3:0] ALUct1;
    always case (FuncCode)
        32: ALUOp<=2; // add
        34: ALUOp<=6; //subtract
        36: ALUOp<=0; // and
        37: ALUOp<=1; // or
        39: ALUOp<=12; // nor
        42: ALUOp<=7; // slt
        default: ALUOp<=15; // should not happen
    endcase
endmodule
```

图 B-5-16 MIPS ALU 控制：一个简单的组合逻辑控制

01 小测验

假设想增加 NOT ($a \text{ AND } b$) 操作，称为与非 (NAND)，应如何修改 ALU？

1. 没有改变。你可以用当前的 ALU 快速计算出 NAND，因为 $(a \cdot b) = \overline{a} + \overline{b}$ ，而且已经有 \overline{a} 、 \overline{b} 以及或门。
2. 必须扩大多路选择器以增加另外的输入，然后增加新的逻辑电路来计算 NAND。

B.6 快速加法：超前进位

提高加法器速度的关键是提高高阶进位的速度。有各种各样的方案来预测进位，最坏的情况是加法器中所有位数的对数的函数。由于进位经过的逻辑门较少，所以这些预期信号执行得

比较快，但是要预测到更准确的进位，需要增加更多的逻辑门。

理解快速进位的关键是要理解：无论输入何时改变硬件都是并行执行的，这一点与软件不同。

B. 6.1 使用“无限”硬件的快速进位

正如前面提到的，任何一个等式都能用两个逻辑层表示。因为只有外部输入是两个操作数和 CarryIn 到加法器的最低位，在理论上我们可以仅在两个逻辑层计算出到加法器所有剩余位的 CarryIn 值。

例如，CarryIn 为 2，CarryOut 恰好为 1，因此公式为

$$\text{CarryIn2} = (b_1 \cdot \text{CarryIn1}) + (a_1 \cdot \text{CarryIn1}) + (a_1 \cdot b_1)$$

类似地，CarryIn1 可以定义为

$$\text{CarryIn1} = (b_0 \cdot \text{CarryIn0}) + (a_0 \cdot \text{CarryIn0}) + (a_0 \cdot b_0)$$

用 c_i 代替 CarryIn i ，上式改写为

$$c_2 = (b_1 \cdot c_1) + (a_1 \cdot c_1) + (a_1 \cdot b_1)$$

$$c_1 = (b_0 \cdot c_0) + (a_0 \cdot c_0) + (a_0 \cdot b_0)$$

将表达式 c_1 带入第一次公式 c_2 ，可得：

$$\begin{aligned} c_2 &= (a_1 \cdot a_0 \cdot b_0) + (a_1 \cdot a_0 \cdot c_0) + (a_1 \cdot b_0 \cdot c_0) \\ &\quad + (b_1 \cdot a_0 \cdot b_0) + (b_1 \cdot a_0 \cdot c_0) + (b_1 \cdot b_0 \cdot c_0) + (a_1 \cdot b_1) \end{aligned}$$

可以想象一下，当加法器得到更高的位时，方程式会如何扩大？它将随着位数的增加而快速增长。这一复杂性可以反应在快速进位的硬件开销上，把这个简单结构广泛用于加法器过于昂贵。

B. 6.2 用第一级抽象快速进位：进位传播函数和进位产生函数

大多数的快速进位方法使用限制方程式的复杂性来简化硬件，同时与行波进位相比速度得到了大幅度提高。超前进位加法器（carry-lookahead adder）就是这样一种结构。在第 1 章中，已经介绍了计算机系统是用不同的抽象级别来配合其复杂性的。一个超前进位加法器依赖于不同的抽象级在其内部的实现。

首先考虑原始方程的各个因子：

$$\begin{aligned} c_i + 1 &= (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i) \\ &= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i \end{aligned}$$

如果用这个公式重写 c_2 的方程，我们将会看到一些重复的部分：

$$c_2 = (a_1 \cdot b_1) + (a_1 \cdot b_1) \cdot ((a_0 \cdot b_0) + (a_0 + b_0) \cdot c_0)$$

注意到 $(a_i \cdot b_i)$ 和 $(a_i + b_i)$ 在上面的公式中重复出现，这两个重要函数通常称为进位产生函数 (g_i) 和进位传输函数 (p_i)：

$$g_i = a_i \cdot b_i$$

$$p_i = a_i + b_i$$

用它们来定义 $c_i + 1$ ，可得：

$$c_i + 1 = g_i + p_i \cdot c_i$$

为了理解信号是从哪里得到的，假设 $g_i = 1$ ，即

$$c_i + 1 = g_i + p_i \cdot c_i = 1 + p_i \cdot c_i = 1$$

也就是说，加法器生成的进位输出 ($c_i + 1$) 和进位输入 (c_i) 是独立的。假设 $g_i = 0$, $p_i = 1$ ，则

$$c_i + 1 = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i$$

也就是说，加法器将进位输入传递到进位输出。将以上二者放在一起可得， $g_i = 1$ 或者 $p_i = 1$ 且 $\text{CarryIn}_i = 1$ ，可得 $\text{CarryIn}_i + 1 = 1$ 。

B-36
l
B-38

B-39

作为比喻，想象一排多米诺骨牌，通过推倒远处的一张牌而将最后一张推倒，假设两张牌之间没有间隙。类似地，一个进位可以通过生成因子而使其为真，只要它们之间所有的传递因子均为真。

根据传递因子和生成因子的定义，我们将其作为第一抽象级，能更加经济地描述进位输入信号。下面所示的是 4 位的：

$$c1 = g0 + (p0 \cdot c0)$$

$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

这些公式只代表一般情况：如果之前的加法器生成了一个进位，并且所有的中间的传递因子传输了这个进位，那么 $\text{CarryIni} = 1$ 。图 B-6-1 就是用这种方法解释超前进位。

但是这种简化的形式使方程变得很长，因此考虑一个 16 位的加法器的逻辑，试着转到两个抽象级上实现。

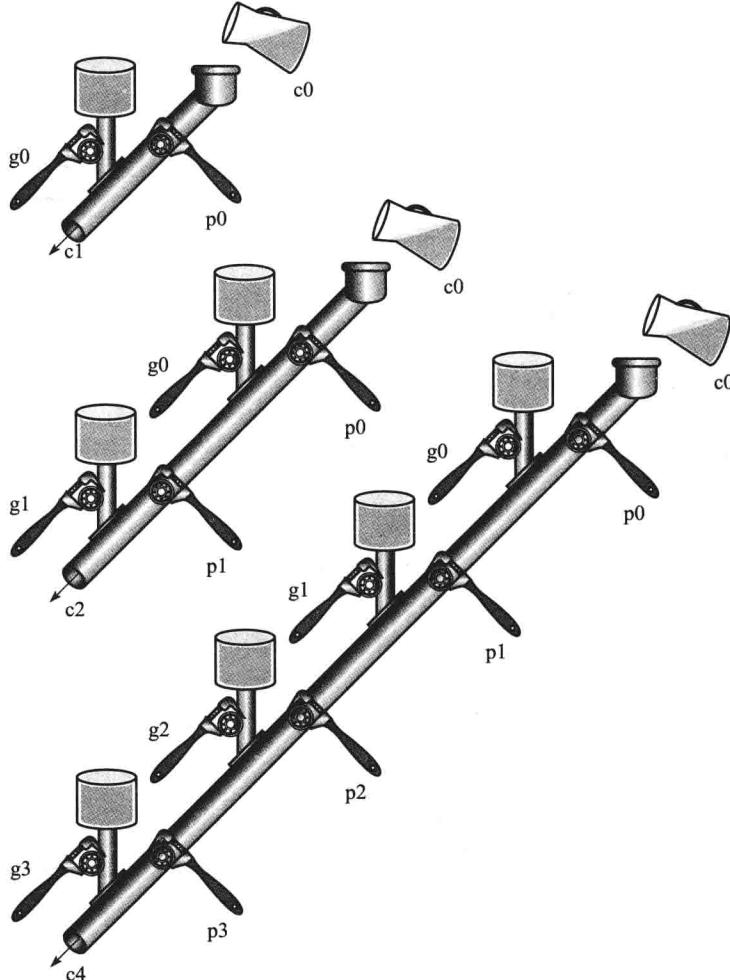


图 B-6-1 1 位、2 位、4 位超前进位的管道表示以及对应的值。扳手有开/关两个值，水用灰色部分表示，管道的输出 (c_{i+1}) 会满，如果最近的进位产生函数的值 (g_i) 处于打开状态或者第 i 个进位传输函数 (p_i) 也是打开的，这样上游会有水，或者从之前生成的或者是从后面传过来的水。进位输入 ($c0$) 能在没有任何进位产生函数的情况下输出一个进位，但是需要所有的进位传输函数

B. 6.3 用第二级抽象快速进位

首先，我们考虑 4 位的加法器，它的超前进位逻辑作为一个单独块。如果将 4 位加法器的行波进位相连接，形成一个 16 位的加法器，这将比原始有更多硬件的加法器速度快。

为了执行得更快，需要将超前进位放置在更高层上。为了执行 4 位加法器的超前进位，需要将传递因子和生成因子也置于较高的层次。下面是 4 位加法器的块：

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

即“超”传递信号的 4 位抽象 (P_i) 为真，当且仅当组中的每一位都将传递一个进位。

对于“超”生成信号 (G_i)，我们只关心 4 位的组中最显著的位中是否有一个进位。如果对于大部分最显著位的生成因子为真，这些情况是显而易见的。如果较早的一个生成因子为真，而且包括大多数的最显著位的中间所有的传递因子也为真，以上情况也是会出现的。

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

图 B-6-2 用管道作为类比，以证明 $P0$ 和 $G0$ 。

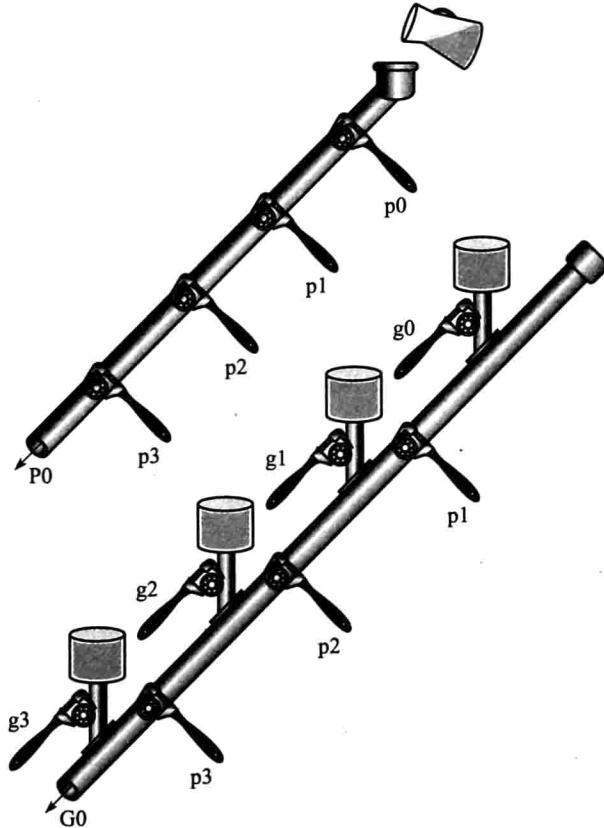


图 B-6-2 下一级超前进位信号 $P0$ 和 $G0$ 的管道分析。仅当所有 4 个进位传输函数 (p_i) 都打开时 $P0$ 是打开的， $G0$ 里是有水流的，仅当至少有一个进位产生函数 (g_i) 是打开的，并且从该生成因子所有的进位传输函数的下游是开放的

图 B-6-3 中 16 位加法器的 C1、C2、C3、C4 与 B.6.2 节的 4 位加法器的 c1、c2、c3、c4 的很相似：

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

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

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

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

图 B-6-3 所示的是 4 位的加法器连接到一个超前进位单元。练习题中会探究这些进位方案

[B-41] 速度的差异，对位传递因子和生成因子信号的不同表示，以及 64 位加法器的设计。

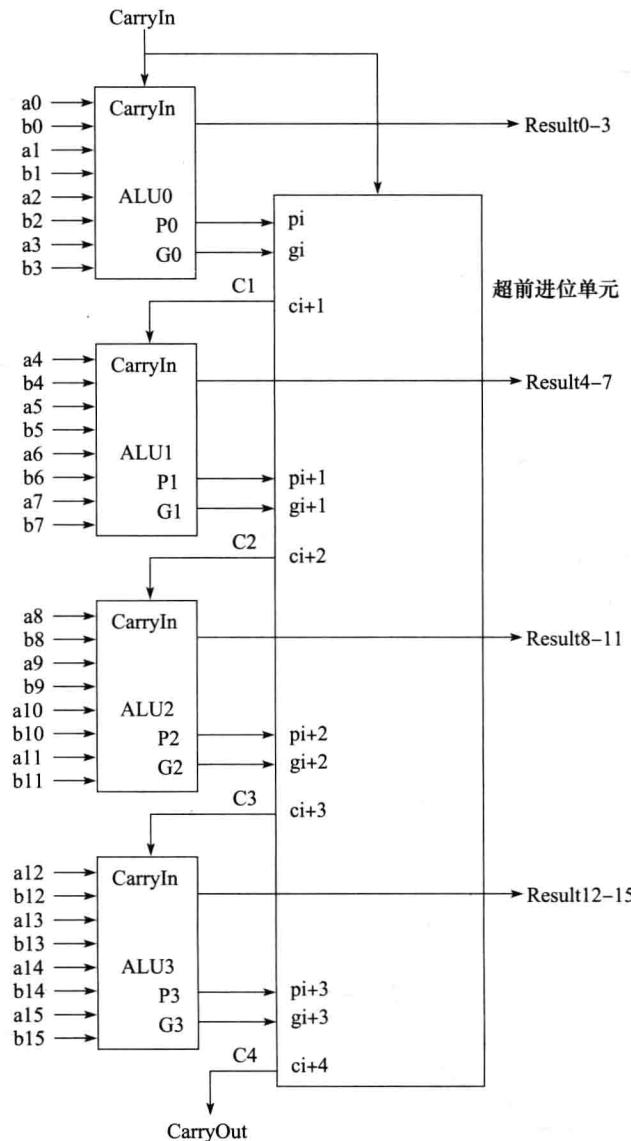


图 B-6-3 用 4 个 4 位超前进位单元串接形成 16 位加法器。注意，进位均来自超前进位单元，而不是 4 位的 ALU

01 例题·不同级的进位传输函数和进位产生函数

确定两个 16 位数的 gi 、 pi 、 Pi 以及 Gi 值：

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

同样的，CarryOut15 (C4) 的值是多少？

01 答案

将各位对准，很容易得到进位产生函数 $gi(ai \cdot bi)$ 和进位传输函数 $pi(ai + bi)$ 的值：

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”进位传输函数 (P3、P2、P1、P0) 是低级进位传输函数简单相与。

$$\begin{aligned} P3 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P2 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P1 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P0 &= 1 \cdot 0 \cdot 1 \cdot 1 = 0 \end{aligned}$$

“超”进位产生函数较复杂一些，用下式表示：

$$\begin{aligned} G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \\ G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \\ G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\ G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \end{aligned}$$

最后，CarryOut15 为：

$$\begin{aligned} C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\ &= 0 + 0 + 1 + 0 + 1 = 1 \end{aligned}$$

因此，当加上这些 16 位的数之后会有一个进位输出。 □

B-42
B-44

超前进位能快速进位的原因是，当时钟周期开始时所有的逻辑单元同时开始计算，并且每个门停止变化时输出不会改变。通过利用更少的门发送进位信号这种快捷方式，门的输出将很快停止变化，因此加法器延迟时间就少了。

为了更好地理解超前进位的重要性，我们需要计算它与行波进位加法器之间的相对性能。

01 例题·行波进位加法器和超前进位加法器速度的比较

一个建立时间逻辑模型的简单方法是假设通过每个与门或者或门需要的时间是相同的。通过简单计算通过逻辑路径上门的数量估计时间，比较 16 位加法器路径上门延迟的数量，一个用行波进位，另一个用的是两级的超前进位。

01 答案

B.5 节中的图 B-5-5 所示的每个进位输出信号需要两个门延迟，最低位上的进位输入和最高位上的进位输出之间的门延迟为 $16 \times 2 = 32$ 。

对超前进位加法器来说，最高位的进位输出正是例子中定义的 C4。用 P_i 和 G_i 两个层次的逻辑上定义 C4 (几个 AND 组成的 OR 式)。 P_i 是用 p_i (与门) 定义的， G_i 是用 p_i 和 g_i 共同来定义

的，所以，下一抽象级最差的情况是两级逻辑。 p_i 和 g_i 都是用 a_i 和 b_i 定义的一级逻辑。在这个方程中，如果假设每个逻辑级都是一个门延迟，那么最坏的情况是 $2 + 2 + 1 = 5$ 个门延迟。

因此，对于一个进位输入到进位输出的通路，16 位超前进位加法器的速度是行波进位加法器的 6 倍，用这种方法可以简单估计硬件的速度。□

B-45
B-46

小结

超前进位加法器比 32 个 1 位组成的 32 位行波进位加法器的速度快，这个快速通路的两个主要信号是进位产生函数和进位传播函数。

前者忽略了进位输入，后者是沿着进位传播的。超前进位加法器的抽象概念在计算机设计解决复杂化问题时的重要性方面也是很好的示例。

01 小测验

用门延迟对硬件执行速度进行简单评估，一个 8 位的行波进位加法器和一个 64 位的超前进位加法器的相对性能如何？

1. 一个 64 位超前进位加法器能快 3 倍：8 位加法器有 16 个门延迟，64 位的有 7 个门延迟。
2. 它们的速度大约相等，因为 64 位加法器需要 16 位加法器有更多的逻辑层。
3. 8 位加法器比 64 位的快，即使有超前进位。

01 精解 除了一个算术逻辑操作之外，我们已经描述了核心 MIPS 指令集的全部操作：图 B-5-14 忽略了移位指令，这可能会扩大 ALU 多路选择器，包括左移一位和右移一位。但硬件设计人员设计了一种电路，称为桶形移位器（barrel shifter），它可以从 1 移到 31 位，消耗的时间和将两个 32 位的数字相加的时间相差不大，所以移位操作通常是在 ALU 外部完成的。

01 精解 B.5 节中，全加器和的输出的逻辑方程可以用一个比与门和或门能力更强的门来简单表示。如果两个操作数不同，异或门输出为真，即

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

在一些技术中，异或门比与门和或门的执行效率更高，用 \oplus 来表示异或运算，则等式可以重新表达为：

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

同样，我们用这种传统的门级表示方法来表示 ALU 电路。当今的计算机都是用 CMOS 晶体管设计的，CMOS ALU 以及桶形移位器利用了开关的优点，而且比文中的设计使用得多路选择器少，但是设计原则是相似的。

01 精解 当有两个以上的级别的层次结构时，用小写和大写字母来区分进位生成和进位传播符号。 $g_{i..j}$ 和 $p_{i..j}$ 代表从 i 位到 j 位的进位产生函数和进位传播函数，因此， $g_{1..1}$ 表示产生 1， $g_{4..1}$ 表示产生 4 到 1，同样， $g_{16..1}$ 表示从 16 到 1。

B-47

B.7 时钟

在我们讨论存储元件和时序电路之前有必要简要地讨论一下时钟。这一节主要讨论这一主题，并且本节内容同 4.2 节的讨论类似。更多的关于时钟和时序策略的细节将在 B.11 节讨论。

时钟在时序电路中非常重要，它决定了处于某一状态的存储元件何时被更新。时钟是一个具有固定周期时间的不停运转的信号，时钟频率是周期时间的倒数。如图 B-7-1 所示，时钟周期时间或者说时钟周期被分割为两部分，高电平时钟和低电平时钟。在此我们只使用边沿触发时钟（edge-triggered clocking）。这意味着所有的状态改变都将发生在时钟边沿。我们之所以使用一种基于边沿触发的时钟策略，是因为它易于解释。从工艺学的角度来看，很难说依赖于边沿触发的时钟策略的时钟同步方法（clocking methodology）是否是最好的选择。

- ⌚ 边沿触发时钟：一种时钟机制，在这种机制下所有的状态改变都发生在时钟边沿。
- ⌚ 时钟同步方法：一种根据时钟来决定数据何时有效和稳定的方法。

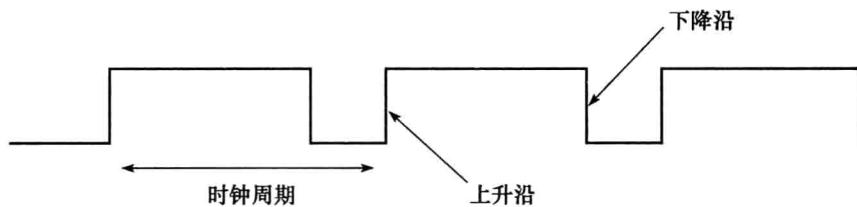


图 B-7-1 时钟信号在高电平和低电平之间震荡。时钟周期是一个完整的最小间隔的时间。
在边沿触发的设计中，可使用有效的上升沿或者下降沿来改变状态

在一种边沿触发的方法中，可使用有效的上升沿或者下降沿来改变状态。在下一节中，**状态单元**（state element）在一个边沿触发的设计中实现，且状态单元的内容只有在一个有效的时钟沿才能改变。选择哪一个时钟边沿作为有效边沿受设计策略的影响，且不会影响逻辑设计中所涉及的概念。

时钟边沿作为一个采样信号，会导致状态单元的输入值被采样且存储在状态单元中。使用一个边沿触发器意味着采样过程实际上是瞬时的，可以消除信号在很小的时间差内被采样可能导致的问题。

时钟系统即同步系统（synchronous system），最主要的限制是被写入状态单元的信号在有效时钟边沿必须是有效的。一个信号在它是稳定（不改变）时才是有效信号，并且在输入不变时，信号值不会改变。因为组合电路没有反馈结果，在组合逻辑单元的输入没有改变的情况下，组合逻辑单元的输出将最终有效。

- ⌚ 状态单元：存储元件。
- ⌚ 同步系统：一个使用时钟的存储系统，且数据信号只有在当前时钟下是处于稳定状态时才可被读取。

图 B-7-2 表示处于一个同步时序逻辑设计中的状态单元和组合逻辑单元的关系。状态单元的输出只在时钟边沿之后改变，且状态单元组合逻辑块提供有效的输入。为了保证在有效时钟边沿写入状态单元的数据是有效的，时钟周期必须足够长才能保证所有在组合逻辑块中的信号都稳定后，在时钟边沿采样这些数据，将其存储在状态单元中。这个限制将时钟周期的长度限制在一个较小的范围，即时钟周期必须足够长以满足所有的状态单元的输入都是有效的。

在附录的其他部分和第 4 章，我们通常忽略时钟信号，因为我们假设所有的状态单元都会在同一时钟边沿更新。一些状态单元会在所有的时钟边沿被写入，而其他一些仅仅在确定的条件下被写入（如某个寄存器被更新）。在这种情况下，我们会使用一个显式的写信号来控制这个状态单元。写信号同时钟信号一起控制状态单元的更新，只有在时钟边沿且写信号有效时状态单元才会被更新。我们将在下一节学习和使用这一机制。

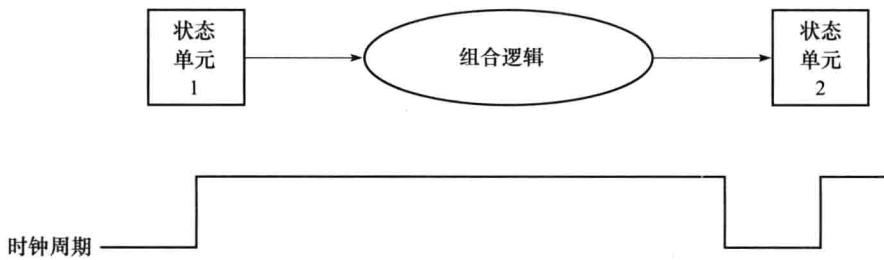


图 B-7-2 组合逻辑块的输入来自状态单元，同时其输出也将被写入状态单元。时钟边沿决定了状态单元的内容何时被更新

边沿触发机制的另一优势是可以将一个状态单元同时作为同一组合逻辑块的输入和输出，如图 B-7-3 所示。实际上，在这种情况下必须要防止竞争，同时要保证时钟周期足够长。这一问题将在 B.11 节讨论。

B-49 现在我们已经讨论了时钟是如何用来更新状态单元的，我们将讨论如何构建这种状态单元。

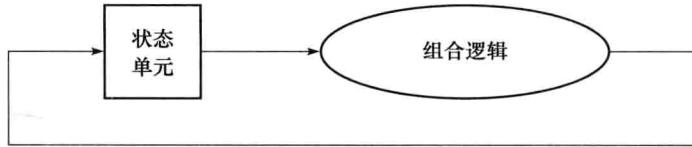


图 B-7-3 边沿触发策略允许一个状态单元在一个时钟周期内被读写，前提是在这个时钟周期内没有会导致不确定值的竞争。因此时钟周期必须足够长，在有效时钟沿到来前所有状态单元的输入都是有效的

01 精解 设计者经常发现，在大多数状态单元中，使得少量的状态单元在相反的时钟沿进行状态变化非常有用。但是在使用这种方法时需要十分小心，因为会影响到状态单元的输入和输出。为什么设计者还要这么做呢？考虑这样的情况，一部分作为状态单元输入或者输出的组合逻辑十分小以至于它们可以在半个周期内完成，而不是通常的完整时钟周期。因此状态单元可以在半个周期的时钟边沿被写入，因为它的输入和输出在半个时钟后期都是有效的。这种技术经常被用在寄存器文件（register file）中，在寄存器文件中，简单的寄存器文件读写通常发生在半个周期。第 4 章使用这种策略来减少流水线开销。

② 寄存器文件：包含一系列寄存器的状态单元，通过提供一个访问的寄存器号可以读写这些寄存器。

B.8 存储元件：触发器、锁存器和寄存器

在本节及下一节中，我们将讨论存储元件的基本原理，从触发器、锁存器开始，再介绍寄存器文件，最后介绍存储器。所有的存储元件都存储着一些状态：存储元件的输出不仅取决于当前的输入，而且与当前存储的数据有关。因此所有包含存储元件的逻辑块都包含有状态信息，属于时序逻辑。

B-50 最简单的存储元件类型是无时钟的，即这些元件都没有任何的时钟输入。因为无时钟的锁存器是最简单的存储元件，所以我们将先讨论这种元件，尽管在本节中我们只使用带时钟的存储元件。图 B-8-1 为一个 S-R 锁存器（set-reset 锁存器），该锁存器由一对或非门构成（或非

门即一个或门加一个反向器)。输出信号 Q 和 \bar{Q} 表示存储的数据及其反向数据。当 S 和 R 都没有被激无效时, 交叉耦合的或非门就是一个反向器, 存储先前的 Q 和 \bar{Q} 的值。

例如, 如果输出 Q 为真, 下面的反向器将产生一个值为假的输出(也就是 \bar{Q}), 这个输出又成为上面反向器的输入, 上面的反向器产生一个值为真的输出, 即 Q , 之后一直循环下去。如果 S 为有效信号, 输出 Q 的值拉低, \bar{Q} 的值被拉高; 如果信号 R 有效, 则输出 \bar{Q} 为假, 输出信号 Q 被拉高。如果 S 和 R 都有效, 则 Q 和 \bar{Q} 最后的数值将被存储在交叉耦合结构内。同时将 S 和 R 置为有效信号, 可能会导致错误的操作: 这取决于 S 和 R 是如何被拉高的, 对锁存器来说, 结果可能会不停地摆动, 也可能处于亚稳态(这部分将在 B.11 节中详细介绍)。

这种交叉耦合结构是我们构造复杂存储元件的基本结构, 构造出来的复杂存储元件可以存储数据。这些存储元件包含额外的门用来存储信号, 并且在包含时钟时对存储的数据状态进行更新。下一节将讲述如何构建这些存储元件。

B.8.1 触发器和锁存器

触发器 (flip-flop) 和**锁存器 (latch)** 是最简单的存储元件。在触发器和锁存器中, 输出信号的值都与存储元件中存储的状态一致。而且, 与上面提到的 S-R 锁存器不同, 从现在开始, 我们使用的所有触发器和锁存器都是带时钟的, 这意味着这些存储元件将包含时钟输入信号, 并且时钟将触发存储状态的变化。触发器与锁存器间的差别在于, 引起存储状态发生变化的时钟位置不同。在一个包含时钟的锁存器中, 只要时钟信号有效, 输入信号发生变化就会引起存储状态的变化。然而在触发器中, 只有在时钟信号的边沿, 存储元件的状态才会变化。因为本文中, 我们使用边沿触发的时钟方法, 即存储状态只在时钟边沿发生变化, 因此我们只使用触发器。触发器大都由锁存器构成, 因此我们先介绍简单的带时钟的锁存器, 介绍完它的一些操作后, 再介绍由这些锁存器构成的触发器的一些操作。

- ② **触发器:** 一种存储元件, 它的输出与内部存储的状态一致, 并且内部状态只在时钟的边沿发生变化。
- ② **锁存器:** 一种存储元件, 它的输出与内部存储的状态一致, 并且当时钟有效时, 只要输入发生变化, 存储状态就会随之发生变化。

对计算机应用程序来说, 触发器和锁存器的功能都是存储信号。一个 D 锁存器或 D 触发器 (D flip-flop) 将输入的数据信号存储在内部元件中。尽管有很多类型的锁存器和触发器, 但我们只需要 D 型的。一个 D 锁存器包含两个输入和两个输出。两个输入中一个是要存储的数据 (D), 一个是时钟信号 (C), 时钟信号用来指示锁存器什么时候读取输入 D 的值并进行存储。输出信号就是内部状态 Q 和其反向信号 \bar{Q} 。当输入时钟 C 有效时, 锁存器称为打开状态, 此时输出信号 Q 的值为输入信号 D 的值。当输入时钟无效时, 锁存器处于关闭状态, 此时锁存器的输出 Q , 等于锁存器最后一次打开时所存储的数据。

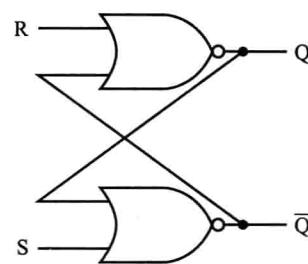


图 B-8-1 一对交叉耦合的 NOR (或非门) 可以存储一位数据。输出信号 Q 取反后得到 \bar{Q} , 再次 \bar{Q} 取反后得到 Q , 因此 Q 是可再生的。如果 R 或 \bar{Q} 中的一个为有效信号时, Q 就会被拉低, 反之亦然

- ② D 触发器：一个包含一个输入数据的触发器，这类触发器只在时钟信号的边沿，才将输入信号存储到内部元件中。

图 B-8-2 显示了如何用交叉耦合的或非门、两个额外的门来构造 D 锁存器。由于当锁存器处于打开状态时，输出 Q 的值随输入 D 的改变而变化，因此这种结构有时也称为透明锁存器。图 B-8-3 显示了 D 锁存器是如何工作的，图中假设输出 Q 初始化为假，并且 D 最先变化。

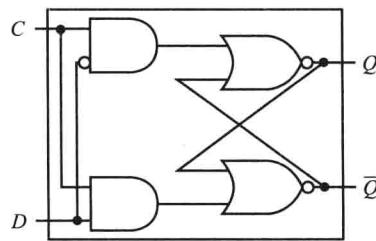


图 B-8-2 一个用或非门实现的 D 锁存器。如果其他的输入为 0，则或非门只作为反向器使用。因此，交叉耦合的或非门只有在输入时钟 C 有效时才进行存储，此时，输入 D 将替代 Q ，并被存储起来。在时钟信号 C 由有效变为无效时，必须保证输入信号 D 的稳定

正如前面提到的那样，我们使用触发器作为基本构造单元，而不是使用锁存器。触发器不是透明的：它们的输出只在时钟边沿发生变化。一个触发器可以设计成在时钟上升沿或下降沿进行触发，在本书的设计中我们可以使用任意一种类型。图 B-8-4 显示了如何用一对 D 锁存器来构造下降沿触发的 D 触发器。在 D 触发器中，输出在时钟边沿存储。图 B-8-5 显示了这个 D 触发器是如何操作的。

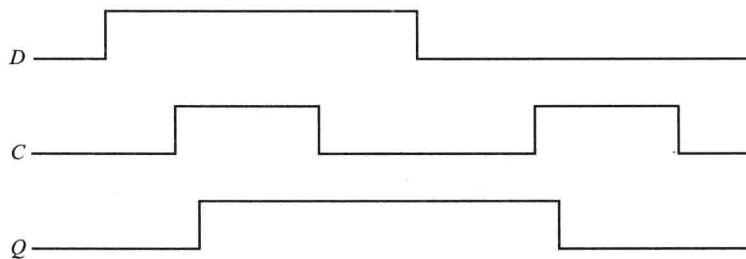


图 B-8-3 D 锁存器的操作，假设输出信号被初始化为无效信号。当时钟 C 有效时，锁存器被打开，输出信号 Q 的值被赋为输入 D 的值

B-52

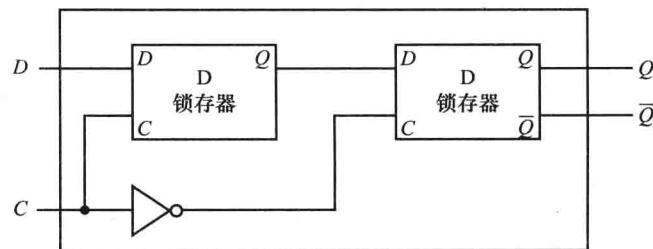


图 B-8-4 下降沿触发的 D 触发器。第一个锁存器称为主锁存器。当输入时钟 C 有效时，主锁存器打开，输入数据 D 。当输入时钟 C 被拉低时，主锁存器关闭，但第二个锁存器打开，并且主锁存器的输出作为第二个锁存器的输入信号。第二个锁存器称为从锁存器

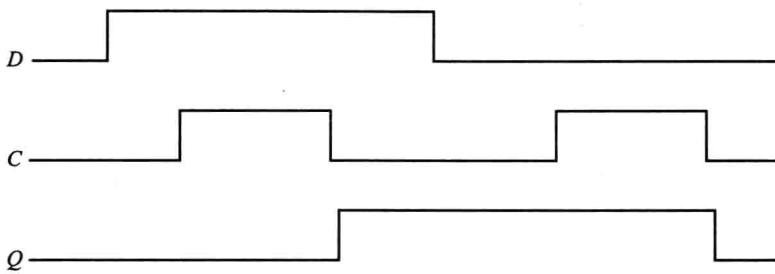


图 B-8-5 下降沿触发的 D 触发器的操作，假设其输出被初始化为无效。当时钟 C 从有效变为无效时，输出 Q 将输入信号 D 的值进行存储。与图 B-8-3 中的 D 锁存器相比，在带时钟的锁存器中，只要时钟 C 为高电平，存储的数据和输出 Q 就发生变化，相反触发器只在时钟发生转变时才发生变化

下面是上升沿触发的 D 触发器的 Verilog 代码，假设信号 C 为输入时钟，D 为输入数据：

```
module DFF(clock,D,Q,Qbar);
    input clock, D;
    output reg Q; // Q is a reg since it is assigned in an
                  // always block
    output Qbar;
    assign Qbar = ~ Q; // Qbar is always just the inverse
                      // of Q
    always @(posedge clock) // perform actions whenever the
                           // clock rises
        Q = D;
endmodule
```

由于输入 D 在时钟边沿被取样，因此在时钟边沿之前和之后的这段时间内，D 必须保持有效。在时钟发生跳变前，输入信号需要保持有效的最短时间，称为建立时间（setup time）；在时钟跳变后，输入信号需要保持有效的最短时间，称为保持时间（hold time）。因此任何触发器（或任何由触发器构造的设备）的输入必须在一个时间窗口内保持有效，这个时间窗口开始于时钟跳变前 t_{setup} 时间，结束于时钟跳变后的 t_{hold} 时间，如图 B-8-6 所示。B.11 节中，详细介绍了时钟和时序约束，包括触发器的传播延时。

B-53

- ② 建立时间：在时钟发生跳变前，输入信号必须保持有效的最短时间。
- ② 保持时间：在时钟跳变后，输入信号需要保持有效的最短时间。

我们可以使用 D 触发器的阵列来构建一个寄存器，构建的寄存器可以存储多位数据，比如一个字节或一个字。在第 4 章中，我们将在数据通路中使用寄存器。

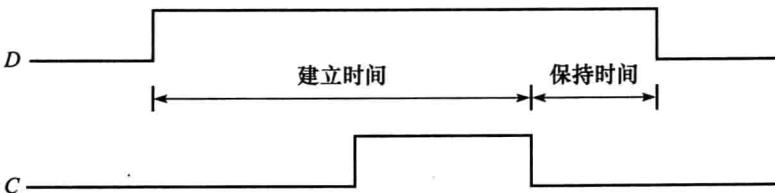


图 B-8-6 下降沿触发的 D 触发器的建立时间和保持时间。输入信号在时钟跳变前和跳变后都需要保持一段时间的有效值。时钟跳变前，输入信号需要保持有效的最短时间称为建立时间；时钟跳变后，输入信号需要保持有效的最短时间称为保持时间。如果违反了最小建立时间和最小保持时间，触发器的输出可能变得不准确，正如 B.11 节中要讲的。保持时间要么为 0，要么是一个很小的值，因此不需要担心建立时间

B.8.2 寄存器文件

寄存器文件是数据通路中一个重要的核心结构。寄存器文件包含一组可读写的寄存器，寄存器的读写是通过指定寄存器号进行的。通过一个由 D 触发器构成的寄存器阵列，并对每一个输入或输出端口添加译码器，就可以构造出一个寄存器文件。如果对寄存器文件只进行读操作，不会改变其状态，因此我们只需提供一个寄存器号作为输入，输出的结果即为该寄存器号对应寄存器中的数据。对于写操作，我们需要三个输入：寄存器号、要写入的数据和一个控制写入的时钟。第 4 章中，我们使用了一个包含两个读端口和一个写端口的寄存器文件。该寄存器文件如图 B-8-7 所示。其中读端口可以通过一对多路选择器来实现，每一个多路选择器的位宽与寄存器文件中单个存储器的位宽相等。

图 B-8-8 为 32 位宽寄存器文件读端口的实现方法。

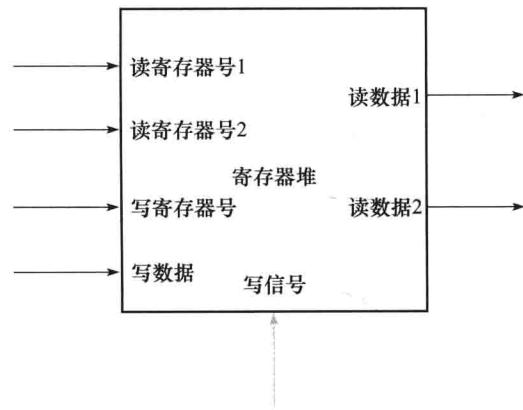


图 B-8-7 一个包含两个读端口和一个写端口的寄存器文件，该寄存器文件包含 5 个输入和 2 个输出。输入控制写信号用灰色表示

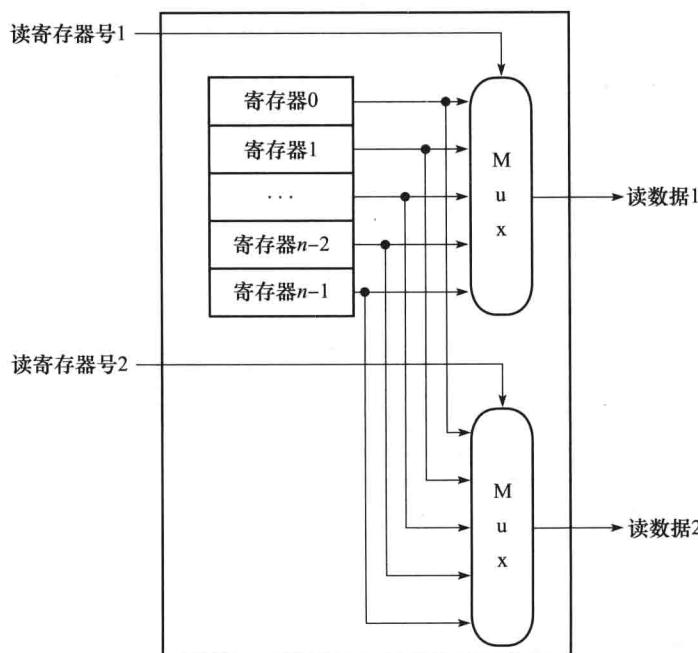


图 B-8-8 对于包含两个读端口、 n 个寄存器的寄存器文件，可以通过一对 n 选 1 多路选择器来实现读端口，每个多路选择器为 32 位宽。读操作寄存器号用来作为多路选择器的选择信号。图 B-8-9 显示了如何实现写端口

实现寄存器写端口有点复杂，因为我们只能更改需要更改的寄存器内容。为了达到这个目的，可以使用一个译码器来生成一个信号，用该信号来决定要对哪个寄存器进行写操作。

图 B-8-9 显示了如何实现一个寄存器文件的写端口。需要注意的是，触发器的状态只在时钟边沿发生变化。在第 4 章中，我们明确地对寄存器文件中的写信号打了勾，并且假设图 B-8-9 中的时钟默认被加入。

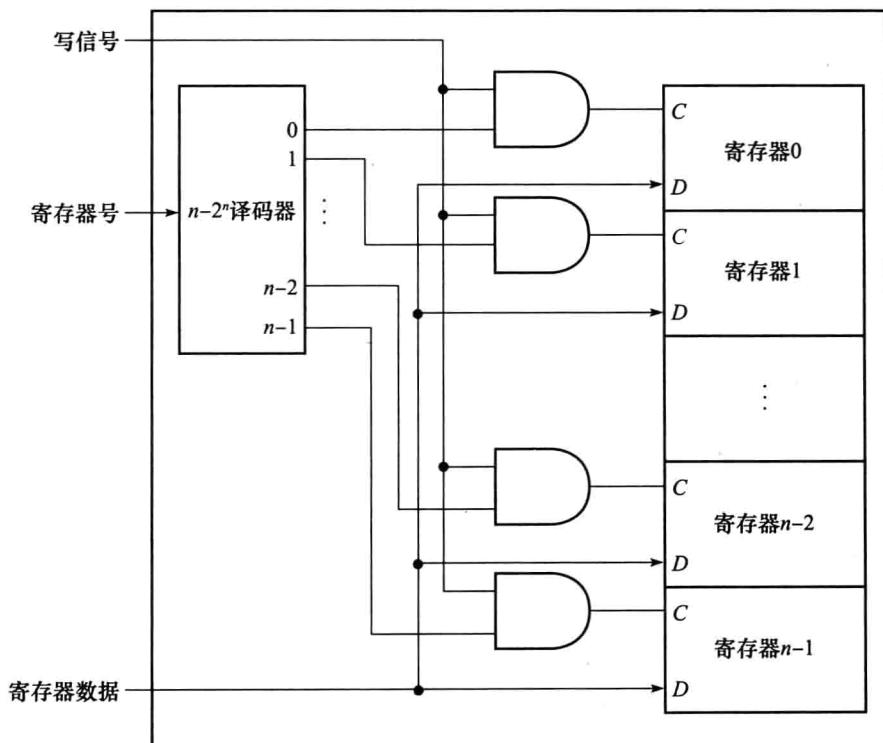


图 B-8-9 寄存器文件的写端口通过一个译码器来实现，译码器与写控制信号一起生成信号 C 输入到寄存器中。所有的三个输入（寄存器号、数据、写控制信号）都存在建立时间和保持时间的约束，以保证正确的数据被写到寄存器文件中

如果在一个时钟周期中，对寄存器同时进行读和写，将会发生什么？因为写寄存器文件出现在时钟边沿，因此在读操作时，寄存器是有效的，正如图 B-7-2 中看到的那样。读出的数据将是上一个时钟周期写入的数据。如果我们想要读出当前正在写入的数据，则需要在寄存器内部或外部添加额外的逻辑。第 4 章广泛使用了这类寄存器。

B.8.3 用 Verilog 描述时序逻辑

用 Verilog 来描述时序逻辑，我们必须知道如何生成一个时钟，如何描述何时将数据写入寄存器中，及如何指定时序控制。我们先来描述一个时钟。时钟不是 Verilog 中预定义的变量，我们可以在一个声明前使用符号 #n 来生成一个时钟，这将导致该声明在 n 个时钟延迟之后被执行。在大多数 Verilog 模拟器中，产生一个时钟来作为外部输入也是可以的，这将允许用户在模拟过程中，指定需要模拟器运行的时钟周期数。

图 B-8-10 中的代码实现了一个简单的时钟，该时钟高电平和低电平都保持一个模拟时间单元，之后就转化为反向状态。为了实现时钟，我们也使用了延迟和阻塞赋值语句。

接下来，我们必须实现边沿触发寄存器的操作。在 Verilog 中，这是通过使用 always 块的敏感信号列表实现的，并且相应的使用 posedge 或 negedge 来指定上升沿触发还是下降沿

```
reg clock; // clock is a register
always
#1 clock = 1; #1 clock = 0;
```

图 B-8-10 一个时钟的 Verilog 描述

B-55

B-56

触发。因此，下面的 Verilog 代码中，在时钟上升沿中，寄存器 A 写入数据 b。

通过本章内容及第 4 章的 Verilog 部分，我们将勾画一个上升沿触发的设计。图 B-8-11 显示了一个 MIPS 寄存器文件的 Verilog 代码，代码中包含了两次读操作和一次写操作，其中只有写操作是包含时钟的。

01 小测验

图 B-8-11 中寄存器文件的 Verilog 代码中，正在进行读操作的寄存器对应的输出端口使用的是连续赋值语句。但是寄存器的写入使用的是 always 块。下面哪一项是其原因？

- a. 没有特殊原因，只是为了方便。
- b. 因为 Data1 和 Data2 是输出端口，WriteData 是输入端口。
- c. 因为读操作是一个组合事件，而写操作则是一个时序事件。

```

reg [31:0] A;
wire [31:0] 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 [31:0] WriteData; // data to write
    input RegWrite; // the write control
    clock; // the clock to trigger write
    output [31:0] Data1, Data2; // the register values read
    reg [31:0] 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 B-8-11 用行为级描述的 MIPS 寄存器文件。该寄存器文件在时钟上升沿进行写操作

B.9 存储元件：SRAM 和 DRAM

寄存器和寄存器文件可以作为基本构建单元来构造小容量存储器，但是，大容量存储器要么是用 **SRAM** (static random access memory，静态随机访问存储器)，要么是用 **DRAM** (动态随机访问存储器) 来构建。我们先来介绍比较简单的 SRAM，再来介绍 DRAM。

B.9.1 SRAM

SRAM 是包含存储阵列的集成电路，存储阵列通常包含一个访问端口，该端口可以用来表示读或写。SRAM 对任一单元的访问时间都是固定的，尽管读操作和写操作的特征不同。依据可寻址单元和每个可寻址单元的位宽，SRAM 芯片形成一个特殊的格局。例如，一个 $4M \times 8$ 的 SRAM，可以提供 4M 的入口，每一个入口包含 8 位二进制数据。因此它共有 22 条地址线

($4M = 2^{22}$)、8位宽的输出和8位宽的输入。与 ROM 类似，可寻址单元的个数称为高度，每个可寻址单元的位宽称为宽度。因为多种技术原因，最新最快的 SRAM 常常使用较窄的配置： $\times 1$ 和 $\times 4$ 。图 B-9-1 显示了一个 $2M \times 16$ 的 SRAM 的输入和输出信号。

- SRAM：一种存储器，其中的数据是静态存储的（如触发器），而不是动态存储的（如 DRAM）。SRAM 比 DRAM 快，但是密度较小，每位的价格更高。

为了启动读写操作，片选信号必须处于有效状态。为读取数据，必须激活用来控制被地址选中的数据能否驱动到管脚上的输出使能信号。这样，输出使能信号允许多个存储器连接到单输出总线上，并且用于决定由哪个存储器来驱动总线。SRAM 读取数据所需的时间通常被定义为从输出使能信号有效一直到数据输出总线上为止。2004 年，拥有最快的 CMOS 部件的 SRAM 的读取时间为 $2 \sim 4\text{ns}$ ，但这样的 SRAM 容量较小，数据宽度较窄，更大部件的 SRAM 的读取时间为 $8 \sim 20\text{ns}$ 。2004 年，已经实现的最大的 SRAM 能容纳 32M 位数据。过去 5 年间，消费类产品和数码设备对低功耗 SRAM 的需求增长很快，这些 SRAM 通常具有更低的待机和访问功耗，但通常要比普通的 SRAM 慢 $5 \sim 10$ 倍。最近，类似同步 DRAM（下一节讨论）的同步 SRAM 也开发出来了。

进行写操作时，输入端必须提供要写入的数据、目的地址以及写控制信号。当写使能信号和片选信号为真时，数据线上的值就写入由地址线指定的地址单元。正如我们在介绍 D 触发器和 D 锁存器时所提到的，该操作期间要保证地址线和数据线上的信号量保持的时间足够长。同时，写使能信号不是时钟触发边沿，而是有最小宽度约束的脉冲。写操作完成时间则受到建立时间、保持时间以及写使能脉冲宽度的制约。

大容量的 SRAM 不能通过寄存器文件的方式组建。其根本原因在于寄存器文件中的 32-1 多路选择器是切实可行的，但想把 $64K \times 1$ 多路选择器用于 $64K \times 1$ SRAM 是不现实的。大容量存储器不采用多路选择器，而是通过一条公共信号线（称为位线）来完成，这样存储阵列中有多个存储单元就能被选中。为了满足多个存储单元驱动一条信号线，需要用到三态缓冲装置。三态缓冲器有两个输入：数据信号线和输出使能信号，还有一个输出信号，输出信号有三种状态：有效、无效或高阻。如果输出使能信号有效，其输出值为输入值或输入值取非。否则，其输出为高阻态，这时将由其他的输出使能有效的三态缓冲器共享输出数据。

图 B-9-2 描述了用一组三态缓冲器组成带译码输入的多路选择器。该电路的关键在于需要考虑到任意时刻至多允许一个缓冲器的输出

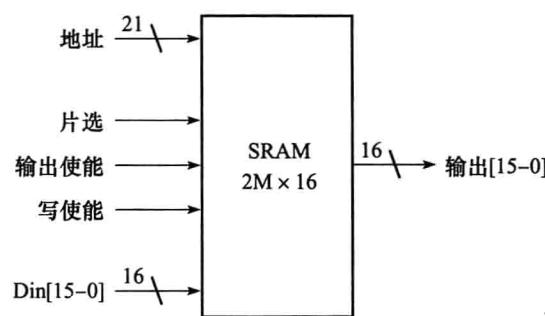


图 B-9-1 一个 $32K \times 8$ 的 SRAM，其中包括 21 位地址线 ($32K = 2^{15}$) 和 16 位输入线，3 条控制线和 16 位输出线

B-57
B-58

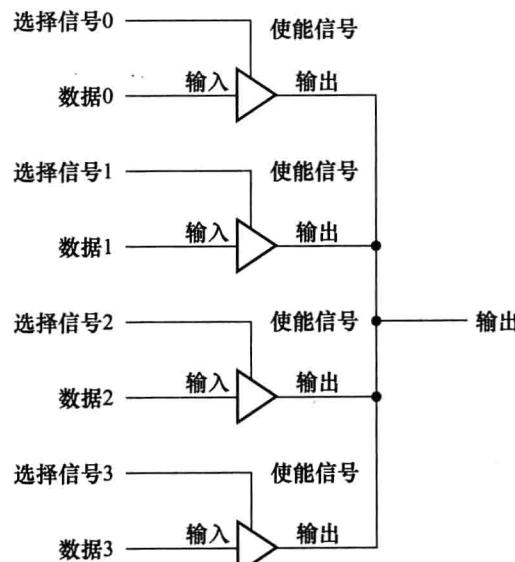


图 B-9-2 用 4 个三态门实现多路选择器。4 个可选择的输入中只有一个有效。三态缓冲器在输出使能信号无效时输出高阻，在输出使能有效时驱动共享的输出线

B-59

使能有效；否则，三态缓冲器将会发生输出线竞争现象。在 SRAM 中，每个存储单元使用三态缓冲器，就能实现存储单元对输出信号线共享。采用分布式的三态缓冲器比大规模集中式的多路选择器效率更高。三态缓冲器通常被嵌入到组成 SRAM 的触发器中。图 B-9-3 描述了小容量的 4×2 SRAM 的实现，其中用到了带有使能输入的 D 锁存器来控制三态输出。

图 B-9-3 的设计电路中没用到多路选择器，但是它还是用到了大容量的译码器和许多字线。例如，在 $4M \times 8$ 型 SRAM 中，我们需要用到 22-4M 的译码器，以及 4M 条字线（用于各触发器使能）！为解决这个问题，大容量的存储器被做成矩阵阵列，并且还用到了二级译码装置。图 B-9-4 表明了 $4M \times 8$ 型 SRAM 是如何利用二级译码来实现的。众所周知，二级译码对于实现 DRAM 的运作至关重要。

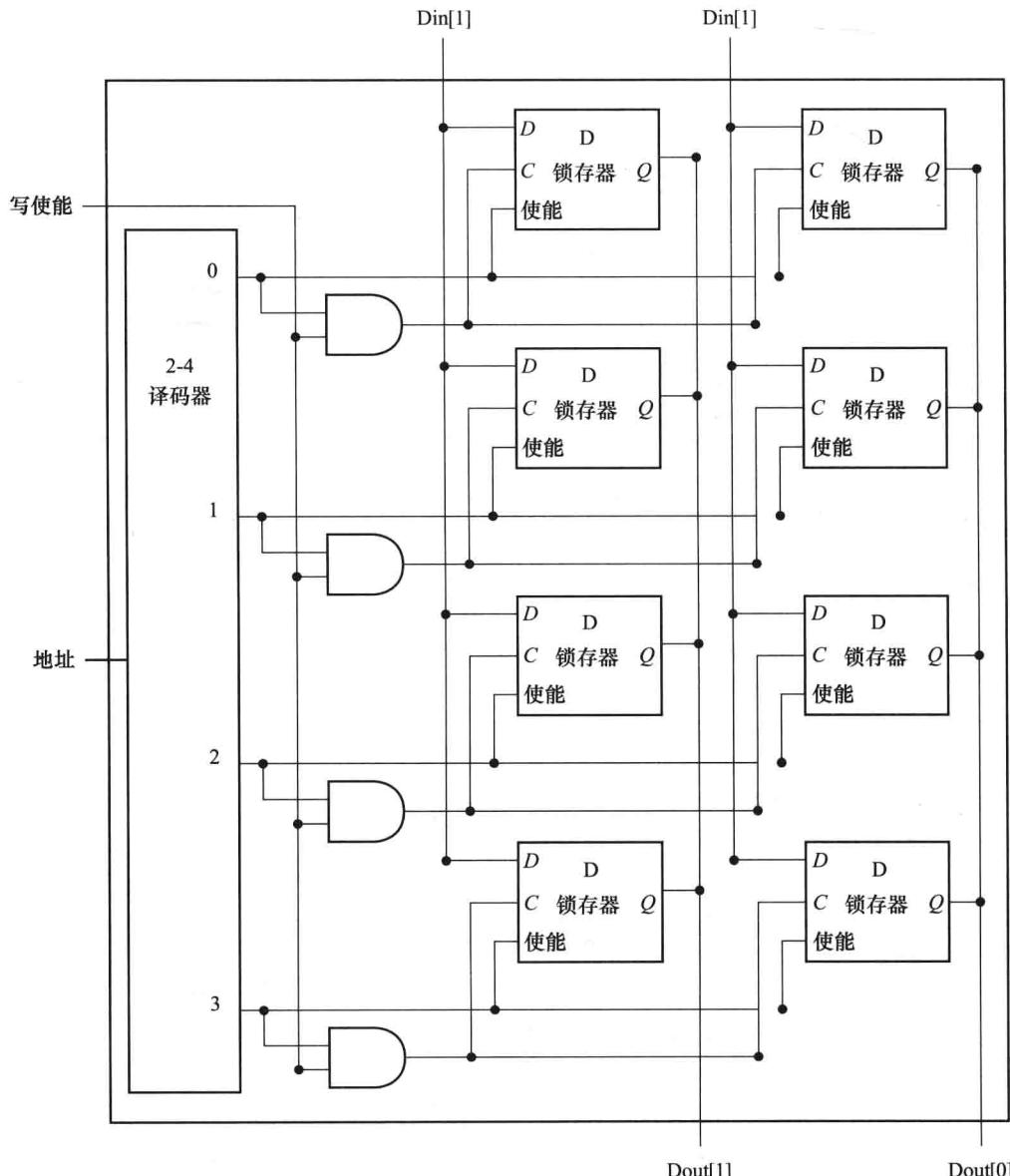


图 B-9-3 4×2 SRAM 的基本结构，其中采用译码器作为片选信号。被激活的单元采用三态输出连接到垂直位数据线，而选择单元的地址信息则通过水平地址线中的某条线（称为字线）传送。为简单起见，此处省略了输出使能信号和片选信号，但它们很容易通过与门接入。

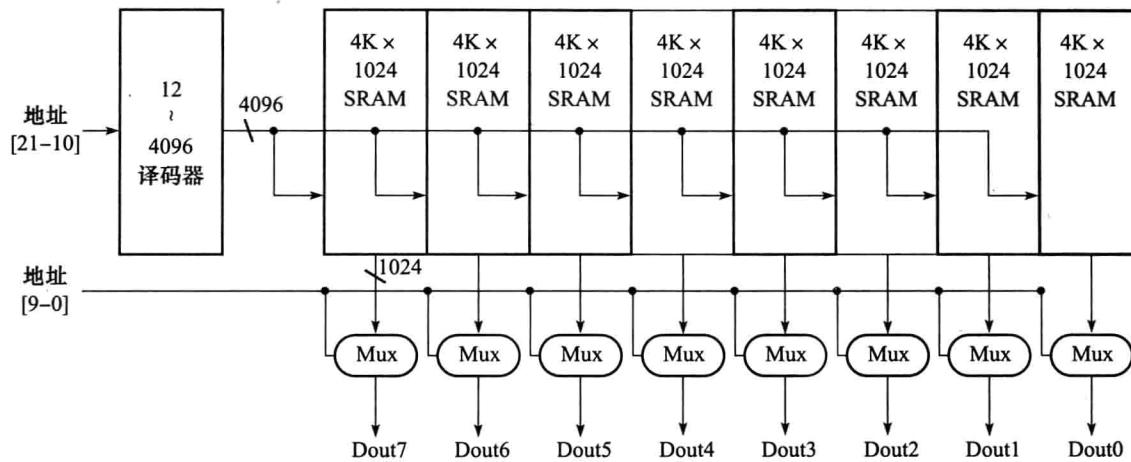


图 B-9-4 用 $4K \times 1024$ 阵列实现 $4M \times 8$ SRAM。译码器 1 产生 8 个 $4K \times 1024$ 阵列的地址，然后由多路选择器（Mux）从每个 1024 位阵列中选出 1 位，该设计远比单级译码器简单，而单级译码器需要一个庞大的多路选择器。实际上，现在这个大小的 SRAM 可能使用更多数量的模块，并且每一块会更小。

近年来，我们看到了同步 SRAM 和同步 DRAM 的发展历程。同步 RAM 的优点在于它能在存储阵列中将顺序地址中的数据以突发方式传输。突发传送的大小同起始地址和突发长度有关。同步 RAM 的卓越之处在于它能将突发数据快速存取，而无需指定额外的地址位。但突发数据中的连续字位受到了时钟的控制。在突发传输模式下，省去指定地址的开销将大大增强传输相邻数据的效率。正因为这个优点，同步 SRAM 和同步 DRAM 在计算机存储系统中大量应用。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。

B-60

B. 9.2 DRAM

在静态 RAM (SRAM) 中，数据保存在一对反向门电路中，所以只要给它供电，其内部数据一直会保留。而在动态 RAM (DRAM) 中，数据是以电荷量的形式被保存在电容中，通过晶体管来存取数据。在动态 RAM 中，对每一位数据的读取只用到了一个晶体管，其密度更高，单位价格更低。相比而言，静态 RAM 中单位比特（每一位）就需要 4~6 个晶体管。由于动态 RAM 中的数值是以电荷量的形式保存在电容中，所以它需要不断刷新来保持数值。这就是该存储结构称作“动态”的缘由。

为实现对存储单元的刷新，我们需要定期读出该内容并且回写到原单元中去。电荷量通常能维持几毫秒，相当于 1 000 000 个时钟周期。目前，单芯片存储控制器能独立完成刷新功能。如果我们只能将动态 RAM 中的内容逐位读出再逐位回写，那么所有时间只能全用来处理几兆字节大容量存储器的刷新了，根本没有其他时间去完成真正意义上的数据存取。幸好在动态 RAM 中也采用了二级译码装置，这就可以在读周期后紧跟一个写周期实现整行刷新（共享一条字线）。通常，刷新工作只占了动态 RAM 的 1%~2% 的时间，剩下的 98%~99% 的时间可以用来处理外界的数据存取。

01 精解 动态 RAM 如何实现读/写存储器单元的信息呢？其关键元件是晶体管，它在存储单元中充当了开关作用，允许存放在电容中的电荷量被读取或写回。图 B-9-5 就是单个晶体管存储单元的外观。电子晶体管的开关作用如下：当字线上的信号有效时，开关处于“关”状态，将电容连到字线上。如果有写请求产生，相应的写入数据就

B-61
B-63

放到该字线上。如果该数值为 1，则电容被充电；否则，数值为 0 时电容放电。由于动态 RAM 必须事先检测电容中的电量，所以读操作显得略为复杂。通常，在激活字线准备读出数据前，该信号线先被充电到一半状态。这样通过激活字线，电容上的电荷可被读出到位线。这时位线只需往高电平或低电平方向稍稍偏移便能满足要求，这种变化能通过敏感放大器检测到。

动态 RAM 有二级译码器，分别实现行存取和列存取，如图 B-9-6 所示。其中行存取选中一行，然后激活对应

的数据字线。于是激活态的行所在的所有列的内容被保存到一组锁存器之中。列存取则是从列锁存器中选取相应的数据。为了节省管脚并进一步减少封装开销，行地址/列地址将共享地址线。其中采用 RAS（行存取选通脉冲）和 CAS（列存取选通脉冲）来决定是行还是列地址。刷新过程只是简单地将列信息读入列锁存器，然后再回写到列单元中去。于是在一个周期之内就可以完成行刷新。二级寻址方法，再加上中间转换电路，会导致 DRAM 的存取时间变长。一般为静态 RAM 的 5~10 倍。2004 年，典型的 DRAM 存取时间为 45~65ns，256Mbit 的 DRAM 已量产，2004 年第一季度第一个 1GB 的 DRAM 样品也生产出来了。因此，单位比特低成本的 DRAM 是实现主存的首选，而高速缓存通常由 SRAM 来完成。

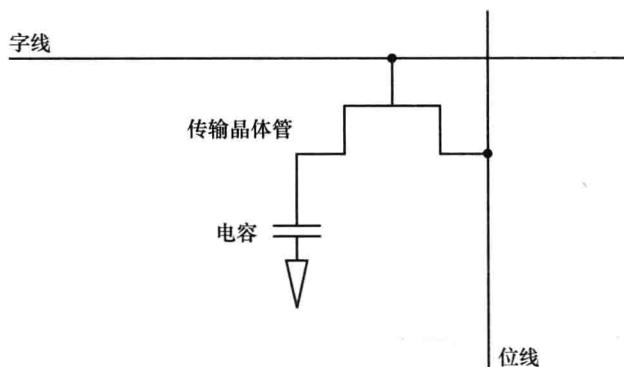


图 B-9-5 用单个晶体管实现的 DRAM 单元。存储单元内容采用电容实现，读写则通过晶体管实现

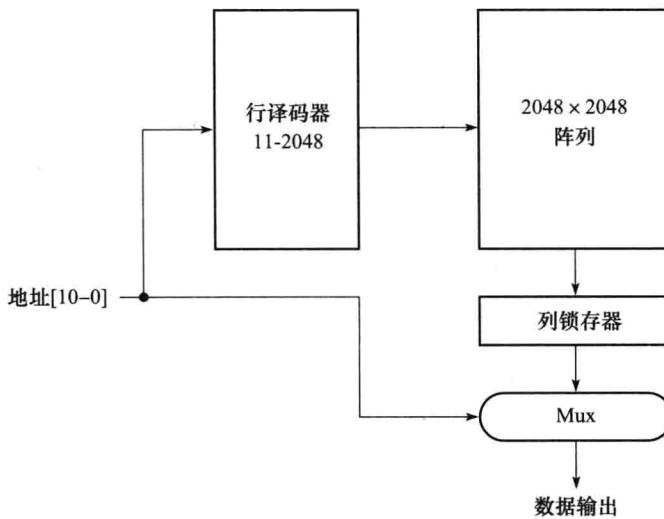


图 B-9-6 用 2048×2048 阵列组成 $4M \times 1$ DRAM。采用 11 位地址选择行，然后再锁存到 2048 个 1 位锁存器中。一个多路选择器从 2048 个锁存器中选择输出。RAS 和 CAS 信号则分别控制地址线是与译码器相连还是与列多选器相连

读者可能会注意到， $64M \times 4$ 的 DRAM 在每次行存取时能同时存取 8K 位，而在列存取时就丢弃了几乎所有位，只剩下 4 位。DRAM 设计师早已通过 DRAM 内部结构实现了更宽的带宽。这个过程可以完成：无需改变行地址，只需能够改变列地址即可，这样就能实现对列锁存

器中的其他位的存取。为了使这个过程更快更精确，地址输入被同步，这样产生了目前主要使用的 DRAM 形式：同步 DRAM 或称为 SDRAM。

1999 年以来，SDRAM 成为大多数基于高速缓存的内存系统中存储芯片的首选。在同步信号控制下，SDRAM 可以快速访问一次突发传输中一行顺序传输的比特流。2004 年，DDRRAM（双倍数据传输率 RAM）是使用最多的 SDRAM 类型。之所以称为双倍数据传输率，是因为在外部时钟的上升沿和下降沿都能传输数据。第 5 章已经提到，这些模式的存储器有利于增加主存的存取带宽，从而进一步匹配 CPU 和高速缓存的速度。

B. 9.3 纠错

因为在大容量存储器中难免发生数据错误，故而大多数系统都会采用各种校验码来检测可能的数据错误。最简单也最常用的是奇偶校验码。在奇偶校验码中，数据中的 1 的个数是有记录的。如果记录数为奇数，则属于奇校验，否则属于偶校验。当数据往内存写入时，校验位也被写上 1 或 0。同时，当数据被读出时，该位也被读出并校验。当内存中的校验位同读出的校验位不一致时，说明数据出错。

1 位奇偶校验能检测出大多数仅有 1 位出错的数据。当某数据有两位出错时，1 位校验法就可能不再奏效。实际上，1 位校验法能测出任何有奇数位出错的数据。但因为出错位数超过 3 位的概率实在很小，所以我们常用 1 位校验码来检测数据中是否有一位出错。遗憾的是，该方法无法确定哪位数字出错。

1 位奇偶校验是一种检错码（error detection code）；另一种称为纠错码（Error Correction Code, ECC）的编码则既能检测错误，还能对错误进行纠正。对于大容量主存，许多系统采用的纠错码不仅能检测出两位之内出错的情况，并且还具有纠正功能。这些方法采用多位编码方式，例如，对主存实施编码的最常用方法是每 128 位数据中加入 7 或 8 位纠错码。

01 精解 1 位奇偶校验码是距离为 2 的编码方法，这就是说，对于数据和校验位而言，任何 1 位数字的改变都会被检测出该数据出错。例如，当改变某个数据位时，校验位就出错；反之亦然。当然，如果我们同时改变两位（两个数据位，或一个数据位和一个校验位），那么奇偶校验位同数据依旧匹配，也就无法检测出错误了。因此，我们将这种校验定义为距离为 2 的校验码。

为了能检测出多于 1 个的错误或纠正一个错误，我们需要距离为 3 的校验码。也就是说，为了判别带校验码的数据是否正确，至少需要有 3 位数字与其他数据不同。假设存在这样的校验码，并且数据中有一位出错，这时，我们就能检测到数据中有一位数字出错，并能纠正；如果有两位数字出错，我们能检测到错误的发生，但无法纠正它。请参考下表的例子，这是对 4 位数据项的距离为 3 的校验码。

B-64
B-65

B-66

数据	校验位	数据	校验位
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

为了说明校验过程，我们不妨以 0110 为例。0110 的纠错码是 011。该数据发生一位错误的可能情况有以下 4 种：1110, 0010, 0100, 0111。请读者注意，011 既是数据 0110 的校验码，也是数据 0001 的校验码。如果校验电路检测到某数值有一位数字出错，它就能肯定该数据是 0110 或 0001 出错。由于 0110 的 4 个 1 位出错数有两位数字同数据 0001 不同，因此该校验码能迅速判断是 0110 数据有误，同时还能加以纠正。为了使两位错误能够被检测出来，简单的方法就是发生两位错的所有组合都有另外一种编码。使用相同编码的方法使码字中有三位不同。但是如果想纠正 2 位错误，就会得到错误的结果，因为该纠正机制仅对 1 位出错有用。如果我们想实现对 1 位、2 位都具有纠错功能，那就需要一个距离为 4 的校验码。

我们在上例中将数据和校验码给区分出来了。但事实上，纠错码把编码和数据看作一个更长的字（在例子中是 7 位）。因此，编码和数据的错误等同对待。

尽管上例中的 n 位数据需 $n - 1$ 位校验码，但随着数据倍数的增加，校验的位数增长较慢。例如，在距离为 3 的校验码中，64 位数据只需 7 位校验码，128 位数据只需 8 位校验码就能实现。这种校验码叫汉明码，是由 R. Hamming 首先发明的。

B.10 有限状态机

前面已经讲过，数字逻辑电路可分为组合电路和时序电路。时序系统的状态存放在存储器中，它们的操作不仅依赖于输入信号，同时也与存储器中的数据、系统的初始状态有关。因此，时序电路无法用真值表加以描述。相反，可以用有限状态机（finite-state machine）来描述时序系统。有限状态机有一组状态量和两个函数（输出函数和下一状态函数（next-state function））。状态集包括在存储单元中可能出现的所有状态量。这样，对于 n 位存储器，就可能有 2^n 个状态量。下一状态函数是这样一种组合函数：通过给定输入值和当前状态量来确定后续状态。输出函数根据当前状态量和输入量产生一组输出。图 B-10-1 是有限状态机的图示。

- ① 有限状态机：一个包含一套输入、输出函数和下一状态函数的时序逻辑函数。其中下一状态函数根据当前状态和输入产生一个新的状态，输出函数根据当前状态和输入（有时不需要输入）确定输出的控制信号。
- ② 下一状态函数：是一个组合函数，根据输入和当前状态确定有限状态机的下一状态。

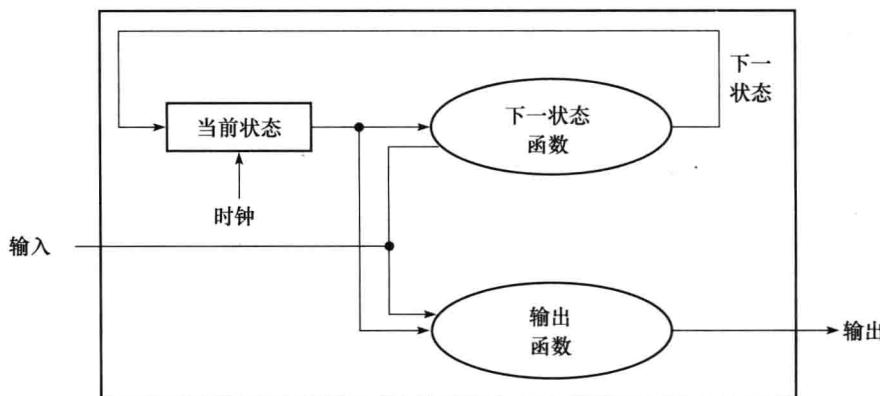


图 B-10-1 一个状态机包含存储状态的内部存储单元和两个组合函数：下一状态函数和输出函数。通常输出函数严格视当前状态量为输入，这样就不会改变时序机的能力，但会对内部值造成影响。

此处和第4章讨论的状态机都是同步的。也就是说，状态量随着时钟周期变化，并且每个时钟周期内都有新的状态量产生。根据前面的约定，状态量的更新发生在时钟触发沿到来时刻。本节和整个第4章都使用了该方法，但通常不显式的支持时钟。在第4章中，我们用到了状态机来控制处理机的执行，并实现了数据路径操作。

B-67

为说明有限状态机的操作和设计过程，我们引用简单经典的“交通灯控制”实例加以说明（第4、5章详细描述了利用有限状态机来控制处理机的执行过程）。若将有限状态机用作控制器，输出函数将仅仅依赖于当前状态，这样的状态机称作摩尔机。在本书中，我们都采用了这种有限状态机。如果输出函数既依赖于当前输入，也依赖于当前状态，这样的状态机称为米利机。这两种状态机在功能上是等价的，二者可以互相转化。摩尔机的优点是快速，而米利机则结构小巧（因为后者的状态量个数比摩尔机少）。第5章已详细讨论了它们之间的差别，并展示了使用米利状态机实现的有限状态机的Verilog版本。

下面的例子是关于交通灯控制逻辑的，背景为东西大道和南北大街相交的十字路口。为简单起见，这里只考虑红灯和绿灯；练习题中有另外加上黄灯的逻辑设计。我们希望灯切换的逻辑周期 ≤ 30 秒。因此采用了频率为0.033Hz的时钟信号，这样就能保证状态间的控制周期 ≤ 30 秒。其中有两个输出信号：

B-68

- NSlite：当信号有效时，南北方向的交通灯为绿色；当信号无效时，南北方向的交通灯为红色。
- EWlite：当信号有效时，东西方向的交通灯为绿色，反之为红色。

另外还有两个输入量：

- NScar：说明有汽车停在监视器处，监视器置于南北方向的交通灯之前。
- EWcar：说明有汽车停在监视器处，监视器置于东西方向的交通灯之前。

只有当其他方向有汽车在等待时，交通灯才会在红绿灯之间切换；否则，交通灯的状态保持为绿色，直到该方向上所有汽车都顺利通过为止。

为完成这个简单的控制，我们还需要两个状态量：

- NSgreen：南北方向的交通灯为绿色。
- EWgreen：东西方向的交通灯为绿色。

下面，我们建立一个下一状态函数表：

	输入		
	NScar	EWcar	Next state
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

注意，算法中我们并没有提及当两个方向同时有汽车通行时该怎么办。如果出现这样的情形，上面的状态表需要修改以保证不会导致某一方向出现交通堵塞。

有限状态机可通过指定输出函数加以实现。

	输出	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

B-69 在考察如何实现这个有限状态机之前，让我们先来看一看用于有限状态机的图形表示。在图解中，节点表示状态量，节点中的一串输出是状态值，有向弧用于指出下一状态函数值，带标记有向弧是将输入条件作为逻辑函数。图 B-10-2 为该有限状态机的图形表示。

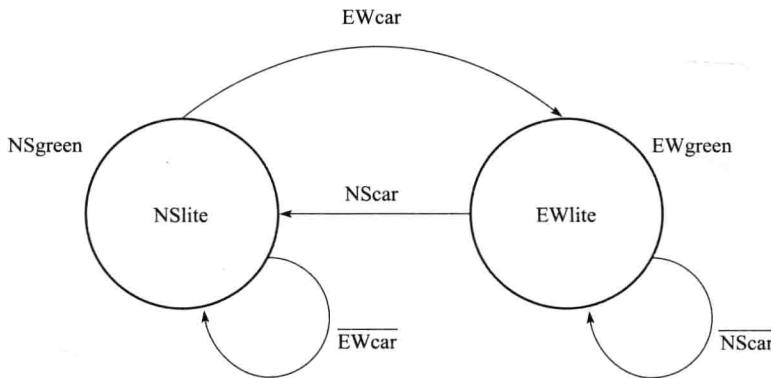


图 B-10-2 2 状态交通信号灯的控制图。其中简化了状态传递的逻辑函数。例如，此状态表中的 NSgreen 到 EWgreen 的传输信号是 $(\overline{NScar} \cdot EWcar) + (NScar \cdot \overline{EWcar})$ ，与 EWcar 相等

有限状态机可这样实现：由寄存器保持当前状态，组合电路计算出下一状态函数和输出函数。图 B-10-3 描述了一个有限状态机的框图：共有 16 个状态量，每个状态量的带宽为 4 位。在实现有限状态机之前，我们先将每个状态标上号，该过程叫状态分配。例如，我们将 NSgreen 标为状态 0，EWgreen 标为状态 1。状态寄存器则只保存 1 位内容。下一状态函数可由以下公式得以计算：

$$\begin{aligned} \text{NextState} = & (\overline{\text{CurrentState}} \cdot EWcar) \\ & + (\text{CurrentState} \cdot \overline{NScar}) \end{aligned}$$

B-70

其中 CurrentState 是状态寄存器的内容（0 或 1）。NextState 是指下一状态量，该值将在时钟周期结束之前写入寄存器。输出函数也很简单：

$$\begin{aligned} \text{NSlite} = & \overline{\text{CurrentState}} \\ \text{EWlite} = & \text{CurrentState} \end{aligned}$$

组合电路则采用结构化的逻辑电路得以实现，譬如采用 PLA。PLA 能自动根据下一状态表和输出函数表实现逻辑设计。其实，也有现成的 CAD 工具，它们先将有限状态机图形化或文本化，然后再自动实现优化电路设计。在第 4、5 章中，有限自动机用于控制处理机的执行。附录 D 则详细讨论了用 PLA 和 ROM 来

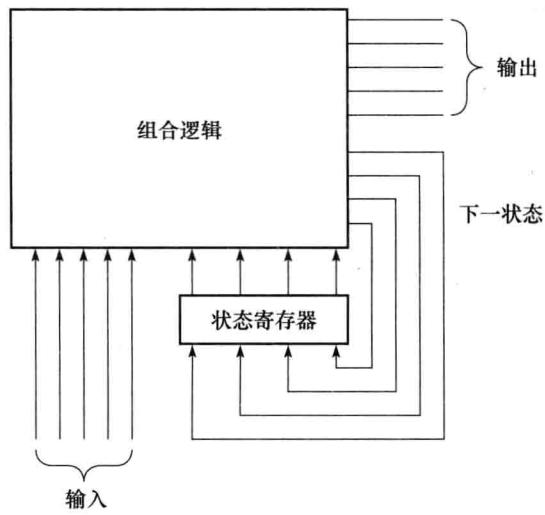


图 B-10-3 有限状态机的实现。有限状态机通过状态寄存器来实现。该寄存器内容包括当前状态和用于计算下一状态和输出函数的组合逻辑块。后面的两个通常采用分离的逻辑模块进行实现，这可能需要更少的门电路

实现这些控制。

为了说明我们如何使用 Verilog 写出控制逻辑，图 B-10-4 给出了一个用于综合的 Verilog 版本。注意，对于这个简单的控制功能，米利机没有用处，但是在第 5 章中为实现控制功能使用的这种类型定义就是米利机，它比摩尔机控制器拥有的状态更少。

B-71

```
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 B-10-4 交通灯控制器的 Verilog 描述

01 小测验

要满足米利机所需的状态数比摩尔机所需的状态数少这个条件，摩尔机最少的状态数是多少？

- 2，因为 1 状态的米利机有可能做相同的事情。
- 3，因为可以构造一个简单的摩尔机，跳转到两个不同状态之一，并且在此之后总是返回先前状态。有可能构造出两状态米利机实现这一功能。
- 需要至少 4 个状态，才能体现出米利机的优越性。

B.11 时钟控制原理

本附录中，我们全部采用时钟边沿作为触发信号。这是因为时钟边沿触发方法比电平触发方法更易于描述和理解。本节将较为详细地阐述时钟控制原理，同时也介绍有关电平触发时钟信号的内容。本节末简单地讨论一下有关异步信号和同步信号的基本原理，这是数字设计中的一个重要问题。

B-72

本节的主要目的是介绍有关时钟控制原理的重要概念。故在章节开始首先假定了一些重要的条件；若有兴趣深入了解时钟控制，可参阅附录末的参考文献。

采用时钟边沿触发方法的优点有两个：易于描述，应用简单。首先，我们假设时钟信号足够长，并且所有时钟都同时到达，那么我们就能保证：对于边沿触发的寄存器位于组合逻辑电路之间的系统，所有的操作都能正确执行，而不会发生竞争现象。如若状态值依赖于不同逻辑单元的相对速度，那么就会发生竞争。在时钟边沿触发电路设计中，时钟周期必须有一定长

度，这样才能满足传输时间（即信号从某个触发器传输到另一个触发器所必须满足的建立时间）。图 B-11-1 描述了采用上升沿触发的系统所必须满足的时钟条件。时钟周期必须至少和下式一样大：

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

3 个分量表示最差情况下的 3 个延时，分别定义如下：

- t_{prop} 是信号通过触发器传播的时间，有时也称为 clock-to-Q。
- $t_{\text{combinational}}$ 是对于任何组合逻辑的最长延时（两个触发器包围的部分）。
- t_{setup} 即在上升沿到来前，触发器输入必须有效的时间。

另一个假设条件是触发的保持时间要求已满足。这在现代逻辑设计中几乎就不是个问题。

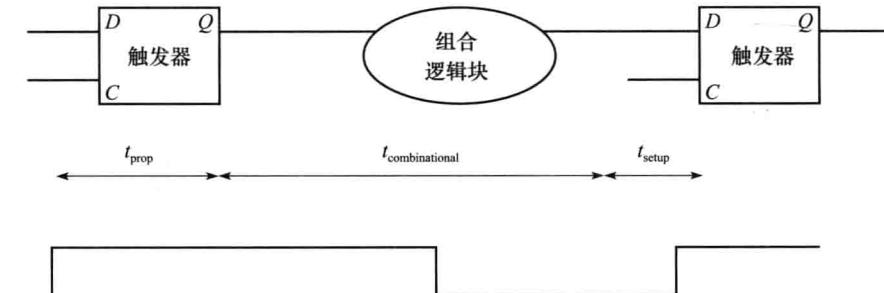


图 B-11-1 在时钟边沿触发的逻辑电路中，需要保证时钟周期足够长，以便保证在下一个时钟触发沿到来之前信号在建立时间内已经有效。信号从触发器输入端传播到触发器输出端的时间为 t_{prop} ，然后经过 $t_{\text{combinational}}$ 的时间通过组合逻辑，并在下一个时钟沿到来之前至少 t_{setup} 时刻有效

B-73

在边沿触发的设计中还必须考虑的一个复杂问题是时钟偏斜（clock skew）。时钟偏斜是指两个状态单元看到同一个时钟沿时的绝对时间差。时钟偏斜产生的原因是时钟信号经常沿两条不同的路径传播，导致到达两个状态单元在时间上有差异。如果时钟偏斜足够大，有可能导致一个状态单元发生变化，从而使得另一个触发器的输入端在该触发器看到时钟沿之前变化。

图 B-11-2 剖析了该问题的产生，这里忽略了建立时间和触发器的传输延时。为避免这种错误，可增大时钟周期以克服最大时钟偏移。这样，时钟周期至少应大于：

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

有了这个前提条件，就算两个时钟的到达先后次序颠倒，即第二个时钟早到了 t_{skew} 整个电路依旧能正常工作。设计人员为减少偏移度，通常需要仔细设计时钟信号经过的路由，争取将偏移减少到最小。另外，聪明的设计师还通过稍稍变大时钟周期的方法来减少时钟偏斜；这些变化在逻辑单元和电源上是允许的。正因为考虑到时钟偏斜会影响保持时间，所以使偏移度尽量小至关重要。

时钟偏斜：两个状态单元看到时钟沿的绝对时间差别。

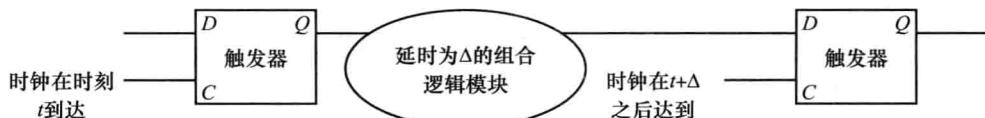


图 B-11-2 时钟偏斜可能引发竞争现象，引起错误操作。由于两个触发器检测时钟信号的时刻存在差异，则可能导致存储在每片的信号都会向前传输，在触发器 2 的时钟有效信号到达之前改变了触发器 2 的输入值

时钟边缘触发有两个缺点：需要额外的逻辑电路，有时会增加时延。比较 D 触发器和电平触发锁存器，我们会发现前者需要更多的逻辑电路。另一种方法是采用电平触发时钟控制（level-sensitive clocking）。因为电平触发机制下状态变化不是瞬间完成的，为使各项操作能正确执行，该方法会更加复杂，需要考虑更多因素。

- 电平触发时钟控制：一种在时钟高电平或低电平期间进行状态改变的时序控制方法，其状态变化没有边沿触发方式快。

B-74

B. 11.1 电平触发时控原理

在电平触发时控机制下，状态量的改变发生在高电平或低电平期间，因为它们不采用时钟边沿触发，所以这些变化并不能瞬间完成，于是就会有竞争现象产生。为了保证在时钟足够慢的情况下，电平触发电路照样能正常工作，设计中用到了双相时钟控制。双相时钟控制中用到了两个互不重叠的时钟信号。参照图 B-11-3，任何时刻至多有一个时钟信号处于高电平状态。这样，我们采用双相时钟（记作 ϕ_1 、 ϕ_2 ）构建的系统就克服了竞争现象，效果与时钟边沿触发电路一样。

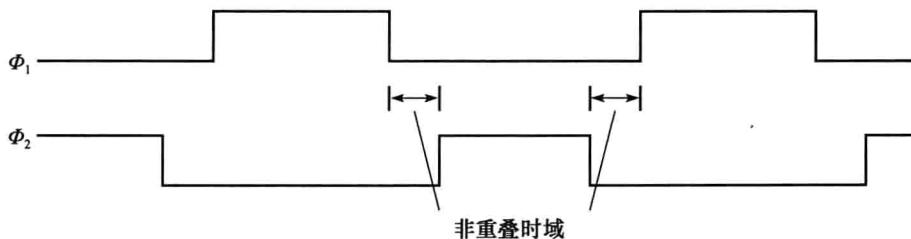


图 B-11-3 双相时钟机制下展示了每个时钟周期和非重叠的阶段

一种简单的设计方案是采用在 ϕ_2 打开的锁存器取代在 ϕ_1 打开的锁存器。因为两个时钟不是同时处于有效状态，于是竞争现象就不会发生。如果组合电路的输入控制信号为 ϕ_1 ，那么它的输出态会在 ϕ_2 时钟周期内被锁存起来。这样当输入锁存器关闭时，其输出值只能在 ϕ_2 的有效信号期间开放，于是能保证输出信号有效。图 B-11-4 描述了双相时钟控制系统和交替锁存器机理是如何实现的。如同在时钟边沿触发电路设计中那样，我们必须考虑并协调好时钟偏斜，尤其是时钟的相位关系。如果能减少两个时钟信号在相位上的重叠分量，就会减少出错的概率。如果每个时钟信号周期足够长，双相不重叠部分足够大，那就可以有效地保证电路运行的正确性。

B-75



图 B-11-4 双相时钟机制下的系统操作和时钟信号。图示表明系统在两个时钟状态下如何运作，在同本身 C 输入反相的时钟状态下，锁存器保持稳定。所以在时域 ϕ_2 中，第一个组合块有一个稳定的输入，其输出则可以被 ϕ_2 锁存。同理，第二个组合块也在其反相时域 ϕ_1 中保持稳定的输入。因此经过组合块的时延决定了各时钟必须有效的最短时间。非重叠时域则由任意逻辑块的最大时钟偏移和最小时延共同决定

B. 11.2 异步输入和同步器

如果时钟偏斜问题得到了解决，只要采用单时钟信号或双相时钟机制，就能完全消除竞争

现象。但是，要想在整个系统中仅使用一个时钟信号并且欲使时钟偏斜很小，这在实际电路中不太现实。实际系统中，CPU有自己的时钟信号，而I/O设备也有自己的时钟信号。一个异步设备可以通过一系列的握手与CPU进行通信。为了将异步输入信号转化为同步信号并用于改变系统的状态量，就需要采用同步器。在同步器中，输入为异步信号和独立的时钟信号，其输出是与此输入时钟同步的信号。

要设计同步器，首先得采用D触发器，如图B-11-5所示，其输入信号D是异步信号。我们还采用了握手通信协议。因为异步信号将一直保持着有效状态直到被确认，所以，无论是在当前周期还是后续时钟周期检测到异步信号并不很重要，于是除了一个小问题之外，就不难理解这种简单设计足以实现正确采样了。

但实际电路设计中还存在一个亚稳态（metastability）的问题。假设在时钟触发沿到来之前，异步信号一直在高低电平间振荡；那么，显然难以判断到底是信号的高电平还是低电平被锁存起来了。虽然这个问题可以克服，但真正头痛的问题在于：如果采样信号的建立时间和保持时间不满足基本要求，触发器可能会进入一种亚稳态。这时，输出信号既不是高电平也不是低电平，而是介于二者之间的电平。并且，还将无法保证触发器在有限的时间内稳定下来。同时，与该触发器相关的逻辑模块测到的输出信号也不一致：有时是0，有时是1，这种现象叫作同步失败（synchronizer failure）。

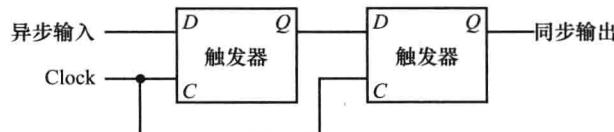
- B-76**
- ① 亚稳态：如果采样时信号不满足建立时间和保持时间的要求，采样所得的数据可能是介于高电平和低电平之间的一个错误值。
 - ② 同步失败：触发器进入亚稳态状态，并且有些逻辑模块读到触发器输出为0，而另外一些模块读到触发器的输出为1。

在一个同步系统中，通过使建立时间和保持时间满足要求，可以确保系统避免同步失败，但是当输入是异步信号时会出现例外。唯一可能的解决方案是当输出信号稳定，或当触发器退出亚稳态后再接收该同步器的输入信号，但这样会增加等待时间。那究竟该等多长时间呢？触发器处于亚稳态的概率是按指数级衰减的，在较短的时间内，此概率就衰减得很小；但永远不会为0！因此要想使同步失败的概率很小，就需经过几年甚至几千年时间。

通常情况下，经过几个建立时间后，失败的概率就衰减得很小了。当时钟周期比亚稳态周期要长时，就需要两个D触发器以保证安全性（见图B-11-6）。有兴趣的读者可进一步阅读参考文献。



图B-11-5 由D触发器组成的同步器。用于采样异步信号，并将产生和时钟保持同步的输出信号。这个同步器不能完全正确工作



图B-11-6 只要亚稳态的时间小于时钟周期，该同步信号器就能正常工作。尽管第一个触发器的输出端可能会是亚稳态，但任何其他的逻辑单元在第二个时钟之前看不到亚稳态。在第二个D触发器在第二个时钟采样信号时，第一个触发器的输出将不再处于亚稳态

01 小测验

假设要设计的电路具有非常大的时钟偏斜，其时钟偏斜程度超过了寄存器的传播时间

(propagation time)。那么是否有可能通过设计来减慢时钟以使得逻辑操作能正确运行?

- a. 可以,即使时钟偏斜非常大,但是只要时钟慢到一定的程度,那么信号总是能正常地传输,整个电路设计就能正确运行。
 - b. 不可以,因为有可能存在这样一种情况:两个寄存器对于时钟边沿的感知时间相差过大,导致在同一个时钟边沿时,其中一个寄存器会发现另一个寄存器已经被触发,并且输出已经传输出去了。
- ② 传播时间:将触发器的输入传播到触发器的输出需要的时间。

B-77

B. 12 现场可编程设备

对于定制或半定制芯片,设计者可以利用底层结构提供的灵活性方便地实现组合或时序逻辑。对于那些不想使用定制或半定制 IC 而又想利用高级集成电路实现复杂逻辑功能的设计者来说,现场可编程器件 (field programmable device, FPD) 是使用最普遍的器件了。FPD 是包括组合逻辑 (可能还有内存设备) 的集成电路, FPD 可由使用者配置。

- ② 现场可编程器件 (FPD):一种包含组合逻辑,可能也包含存储设备的集成电路,最终用户可以对其进行配置。

FPD 主要分为两个阵营:可编程逻辑器件 (programmable logic device, PLD),它是纯粹的组合逻辑;现场可编程门阵列 (field programmable gate array, FPGA),它提供组合逻辑和触发器。PLD 由两种形式组成:简单 PLD (simple PLD, SPLD),就是通常的 PLA 或者可编程阵列逻辑 (programmable array logic, PAL);还有就是复杂 PLD,它包括多于一个的逻辑模块以及模块间的可配置互连线路。当谈到 PLD 中的 PLA 时,通常是指带有用户可编程与阵列和或阵列的 PLA。PAL 类似于 PLA,除了它的或阵列是固定的以外。

- ② 可编程逻辑器件 (PLD):包含最终用户可配置功能的组合逻辑的集成电路。
- ② 现场可编程门阵列 (FPGA):一种可配置的包含组合逻辑模块和触发器的集成电路。
- ② 简单 PLD (SPLD):一种可编程逻辑器件,通常包含一块 PAL 或 PLA。
- ② 可编程阵列逻辑 (PAL):由一个可编程的与阵列后跟一个可编程的或阵列组成的可编程逻辑电路。

在讨论 FPGA 以前,先来看看 FPD 是如何配置的。配置的主要问题就是在何处建立或打破连接。门电路和寄存器是静态的,但是连接是可配置的。注意,通过配置连接,用户可以决定实现何种逻辑功能。考虑一个可配置的 PLA:通过决定与阵列和或阵列在何处连接,用户决定 PLA 中运算的逻辑。FPD 中的连接可以是永久的,也可以是可配置的。永久连接涉及两个连线之间建立或破坏连接。现在的 FPLD 都使用反熔丝 (antifuse) 技术,允许在编程时建立连接然后再永久固定下来。配置 CMOS FPLD 的另外一种方法是使用 SRAM。在上电时,配置信息下载到 SRAM 中,这些内容控制开关设定进而决定哪些金属线连接起来。FPD 使用 SRAM 控制的好处在于可以通过修改 SRAM 的内容进行重新配置。基于 SRAM 控制的两个缺点是:配置信息是易失的,必须在上电时重新加载;为了切换使用主动晶体管会给线路增加一点电阻。

- ② 反熔丝:集成电路中的一种结构,当对其进行编程时,将导致线间的永久性连接。

包含逻辑和存储设备的 FPGA 通常是二维阵列结构,划分行、列的通道用来进行阵列单元间的互连。每个单元是门和触发器的组合,可以编程执行特定功能。因为可编程的 RAM 通常

B-78

很小，它们也被称为查找表（lookup table，LUT）。更新的 FPGA 包括更复杂的构建模块，例如加法器和用来构建寄存器文件的存储模块。一些大型的 FPGA 甚至包含 32 位的 RISC 核心。

② 查找表（LUT）：现场可编程器件中的单元的名称，包含少量的逻辑和 RAM。

除了可以对每个单元进行编程执行特定的功能，单元间的互连也是可编程的，这就使得现在包含上百模块和上千门电路的 FPGA 可以实现复杂的逻辑功能。互连是可定制芯片的最大挑战，对于 FPGA 也是如此，因为单元不能表示结构化设计分解的自然功能单元。许多 FPGA 有 90% 的部分用来实现互连，只有 10% 是逻辑和存储模块。

正如你不可能不使用 CAD 工具来设计定制或半定制芯片，你也需要 CAD 工具来设计 FPD。已经开发出针对特定 FPGA 的逻辑合成工具，帮助从结构或行为 Verilog 描述中使用 FPGA 生成系统。

B.13 结论

本附录介绍了逻辑设计的一些基本概念和原理。在了解了这些内容之后，请参阅第 4、5 章，那些内容是本附录的应用和进一步深入。

拓展阅读

关于逻辑电路设计，有很多好书，以下列出了其中一些。

一本详细介绍用 Verilog 进行逻辑设计的书：

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

一本关于逻辑设计的书：

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

B-79

一本关于逻辑设计的书：

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

B.14 练习题

B.1 [10] < B.2 > 除了这一部分我们讨论过的基本规律之外，还有两个重要的定理，叫作德·摩根定理：

$$\overline{A + B} = \overline{A} \cdot \overline{B} \quad \text{和} \quad \overline{A \cdot B} = \overline{A} + \overline{B}$$

使用下面的真值表对上面的德·摩根定理进行证明：

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

B.2 [15] < B.2 > 用德·摩根定理和 B.2 节中的结合公理证明例题中关于 E 的两个表达式是等价的。

B.3 [10] < B.2 > 证明对于 n 输入的逻辑函数，对应的真值表有 2^n 项。

B.4 [10] < B.2 > 异或函数具有多种用途（可用于加法器或用来计算校验码）。对于二输入的异或函数，当且仅当一个输入值为“真”时输出才为“真”。写出二输入异或函数的真值表，并用与门、或门和反相器实现该函数。

- B. 5** [15] <B. 2> 通过使用二输入的或非门实现与、或、非三种逻辑功能，证明利用或非门可以实现任何逻辑功能。
- B. 6** [15] <B. 2> 通过使用二输入的与非门实现与、或、非三种逻辑功能，证明利用与非门可以实现任何逻辑功能。
- B. 7** [10] <B. 2, B. 3> 写出四输入奇校验函数的真值表（关于错误校验的内容参见 B. 9.3 节）。
- B. 8** [10] <B. 2, B. 3> 用输入端和输出端带有反向小圆圈的与门和或门实现四输入的奇校验函数。
- B. 9** [10] <B. 2, B. 3> 用 PLA 实现四输入的奇校验函数。
- B. 10** [15] <B. 2, B. 3> 通过使用多路选择器实现与非门（或者或非门），证明二输入多路选择器可以实现任何逻辑功能。
- B. 11** [5] <4.2, B. 2, B. 3> 假设 X 由三位 x_2, x_1, x_0 组成。分别写出下列 4 个逻辑表达式（当且仅当满足下面的条件时逻辑表达式为“真”）：

- X 中只有一个 0。
- X 中有偶数个 0。
- 当 X 被当做无符号二进倒数时， X 小于 4。
- 当 X 被当做有符号二进制数时， X 是负数。

- B. 12** [5] <4.2, B. 2, B. 3> 用 PLA 实现练习题 B. 11 的 4 个逻辑函数。
- B. 13** [5] <4.2, B. 2, B. 3> 假设 X 由三位 x_2, x_1, x_0 组成， Y 由三位 y_2, y_1, y_0 组成。写出下列 3 个逻辑表达式（当且仅当满足下面的条件时逻辑表达式为“真”）：
- 当 X, Y 被当做无符号二进制数时， $X < Y$ 。
 - 当 X, Y 被当作有符号（二进制补码）数时， $X < Y$ 。
 - $X = Y$ 。

使用可以扩展到多位的层次表达方法，写出如何扩展为 6 位比较。

- B. 14** [5] <B. 2, B. 3> 用逻辑电路实现开关网络：输入为 A 和 B ；输出为 C 和 D ；控制信号为 S 。当 $S=1$ 时，网络为直通模式，即 $C=A, D=B$ ；当 $S=0$ 时，网络为交叉模式，即 $C=B, D=A$ 。
- B. 15** [15] <B. 2, B. 3> 由 B. 2 节中 E 的“和之积”形式推出其“积之和”形式。你需要使用德·摩根定理。
- B. 16** [30] <B. 2, B. 3> 设计一个算法，该算法能够对任何包含与、或、非逻辑的函数构建其“积之和”形式的表达式。算法应当具有递归性，并且在整个过程中不能产生真值表。
- B. 17** [5] <B. 2, B. 3> 写出多路选择器的真值表（输入为 A, B 和 S ，输出为 C ），通过使用无关项来简化真值表。

- B. 18** [5] <B. 3> 下面的 Verilog 模块实现了何种功能：

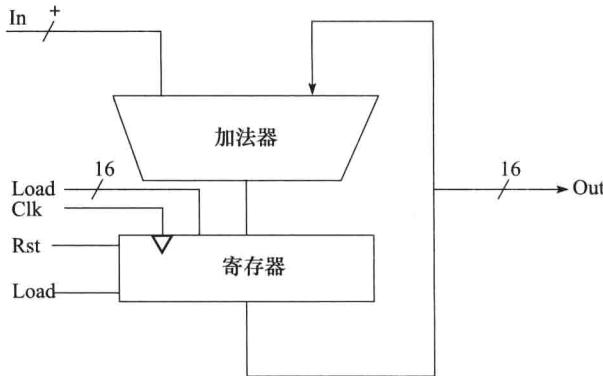
```
module FUNC1 (I0, I1, S, out);
    input I0, I1;
    input S;
    output out;
    out = S? I1: I0;
endmodule

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
    end
endmodule
```

[B-80]

[A-81]

- B.19** [5] < B. 4 > B. 8.1 节给出了 D 触发器的 Verilog 代码, 请给出 D 锁存器的 Verilog 代码。
- B.20** [10] < B. 3, B. 4 > 写出 2-4 译码器 (与/或编码器) 的 Verilog 模块实现。
- B.21** [10] < B. 3, B. 4 > 根据下面给出的累加器逻辑图, 写出它的 Verilog 模块实现。假定使用正边沿触发寄存器和异步 Rst。



- B.22** [20] < B. 3, B. 4, B. 5 > 3.3 节介绍了乘法器的基本操作和可能的实现。这个实现的基本单元是一个移位加法单元。给出这个单元的 Verilog 实现, 并说明如何使用这个单元建立 32 位乘法器。
- B.23** [20] < B. 3, B. 4, B. 5 > 根据上一题, 实现无符号除法器。
- B.24** [15] < B. 5 > ALU 支持只使用加法器的符号位设置小于 (`s1t`)。用这种方法比较 -7_{10} 和 6_{10} , 为简单起见, 使用 4 位二进制表示: 1001_2 和 0110_2 。
 $1001_2 - 0110_2 = 1001_2 + 1010_2 = 0011_2$
- 这个结果表示 $-7 > 6$, 这显然是错误的。因此在判断时必须考虑到溢出。修改图 B-5-10 中的 1 位 ALU 来正确处理 `s1t`。为了节省时间可以直接复印图, 在图上改。
- B.25** [20] < B. 6 > 在加法中检查溢出的一个简单方法是看最高有效位的 CarryIn 是否和最高有效位的 CarryOut 相同。证明这个方法和图 3-2 是一样的。
- B.26** [5] < B. 6 > 使用新定义重写 B. 6.3 节中 16 位加法器的超前进位逻辑公式。第一, 使用加法器独立位 CarryIn 信号的名字, 即使用 $c4, c8, c12, \dots$, 而不是使用 $C1, C2, C3, \dots$; 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

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

可改写成

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

这个更通用的定义在建立位数更宽的加法器时有用。

- B.27** [15] < B. 6 > 使用练习题 B. 26 的新定义写出 64 位加法器的超前进位逻辑公式, 使用 16 位加法器作为基础模块。并给出类似图 B-6-3 的图。
- B.28** [10] < B. 6 > 下面计算加法器的相对性能。假定针对某个公式的硬件运行时间为一个时间单位 T , 这个公式只包含与运算或者或运算, 例如 B. 6.2 节的 π 和 gi 公式。由几个与项进行或运算构成的公式运行时间需要 $2T$, 例如 B. 6.2 节中的公式 $c1, c2, c3$ 和 $c4$ 。这个时间包括计算与运算的时间 T 和计算或运算的额外时间 T 。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, 增加中间公式带来的相应时延, 反复迭代直到公式中使用的都是加法器的实际输入为止。并且给每个加法器标出计算时延并标明最坏情况时延。
- B.29** [15] < B. 6 > 类似练习题 B. 28, 不过这次只计算 16 位的相对速度, 加法器的结构分别是: (1) 行波进位加法器; (2) 4 位一组, 组内超前进位, 组间行波进位; (3) 采用 B. 6.2 节所示的超前进位加法器。
- B.30** [15] < B. 6 > 与练习题 B. 28 和 B. 29 类似, 本题计算 64 位加法器的相对速度, 加法器的结构分别是: (1) 行波进位加法器; (2) 4 位一组, 组内超前进位, 组间行波进位; (3) 16 位一组, 组内超前进位, 组间行波进位; (4) 采用练习题 B. 27 中的超前进位加法器。

B.31 [10] < B.6 > 如果我们不把加法器看成一个把两个数相加然后与进位连接到一起的装置，而是将其看成可以把三个数 (a_i, b_i, c_i) 相加，并且产生两个输出 ($s, c_i + 1$) 的硬件装置。当进行两个数的加法时，我们并不能据此做些什么。但是当我们进行两个以上操作数的加法时，就可以通过上述想法降低进位开销。该想法是构造两个独立的“和”，分别叫作 S' （和数位）和 C' （进位）。在这一过程的末尾，我们需要用一个普通的加法器把 S' 和 C' 加到一起。这个把进位传播推迟到加法运算最后阶段的技巧称为进位保留加法。图 B-14-1 右下角的模块图显示了该加法器的结构，该结构中两个进位保留加法器通过一个普通加法器连接到一起。

对于具有 4 个 16 位二进制数的加法运算，分别计算采用完全超前进位加法器和带有超前进位加法器（用来形成最终的累加和）的进位保留加法器的时延。（时间单位 T 与练习题 B.28 相同）

B-84

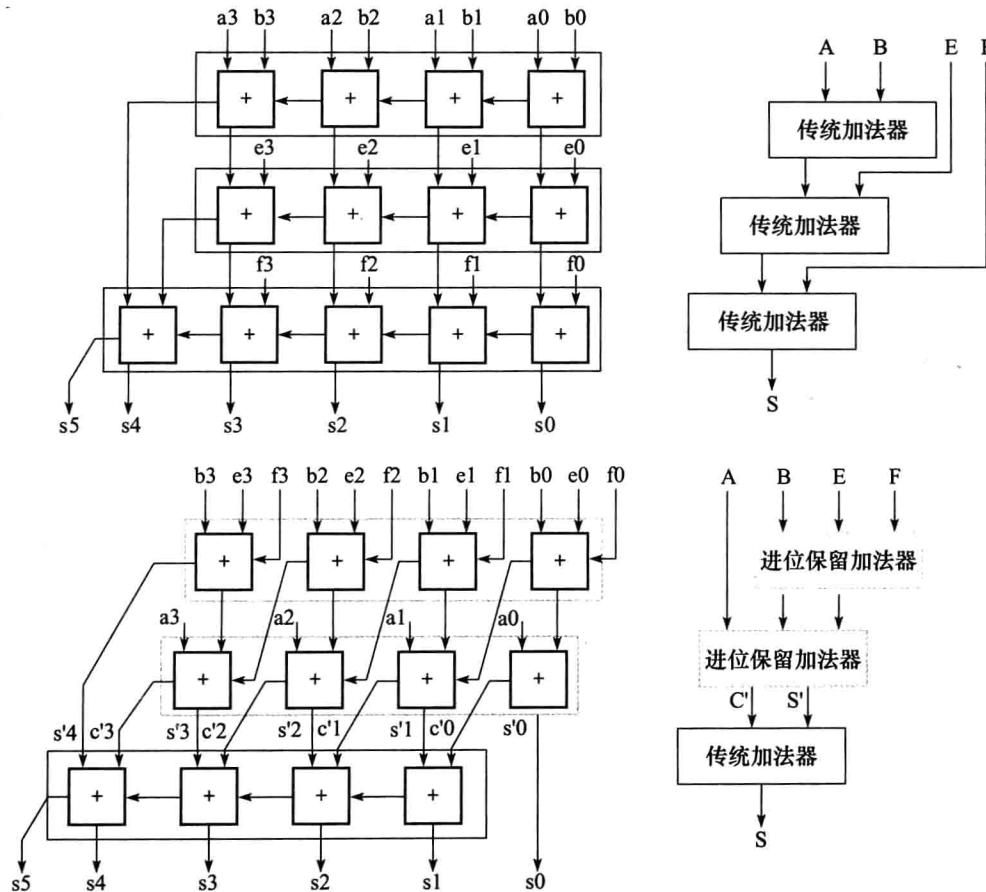


图 B-14-1 4 个 4 位数相加的传统行波进位和进位保留加法器。加法器细节见左边，单独信号小写表示，相应高层模块见右图，组合信号大写表示。注意，4 个 n 位数的和需要 $n+2$ 位

B.32 [20] < B.6 > 在计算机当中最有可能同时把多个数相加到一起的情形恐怕就是当我们试图在一个时钟周期中通过采用多个加法器将多个数相加的办法来加快乘法运算的速度。相比于第 3 章提到的乘法算法，具有多个加法器的进位保留方案可以实现 10 倍以上的乘法运算速度。本习题对采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。假设存在 16 个部分积 $M_{15}, M_{14}, \dots, M_0$ ，这些部分积分别表示被乘数与乘数的每一位 ($m_{15}, m_{14}, \dots, m_0$) 进行“与”运算的结果。我们的想法是用进位保留加法器将 n 个操作数减少到 $2n/3$ 个并行组，每组 3 个，反复迭代直至得到两个大数，最后用普通加法器把二者加到一起。

首先，根据图 B-14-1 右半部分所示，画出 16 位进位保留加法器的组织结构，用来实现 16 个部分积的相加。然后计算把这 16 项加到一起的时延。将计算出的结果与第 3 章中的迭代乘法方案进行

比较，但需要注意的是，这里假定 16 次迭代过程中使用的是具有完全超前进位的 16 位加法器，该加法器的速度在练习题 B.29 中已经计算过。

- B-85**
- B.33** [10] < B.6 > 有时用户想要将一组数一同加起来。例如，使用 1 位完全加法器将 4 个 4 位数 (A, B, E, F) 加起来。现在忽略超前进位。将 1 位加法器按图 B-14-1 的组织形式连接起来。在传统组织形式之下是完全加法器的新组织形式。使用这两种组织结构实现 4 个数的加法，并确保能够得到相同结果。
- B.34** [5] < B.6 > 首先给出如图 B-14-1 所示的 16 位进位保留加法器的模块组织结构。假定通过一个 1 位加法器的时间是 $2T$ ，计算上下两个组织结构进行 4 个 4 位数加法所需的时间。
- B.35** [5] < B.8 > 很多时候你可能希望得到这样的时序图，该时序图包含了对发生在数据输入端 D 和时钟输入端 C (类似于 B.8 节图 B-8-3 和图 B-8-6) 的变化的描述，通常 D 锁存器和 D 触发器的输出端波形 (Q) 是不同的。用一两句话描述使二者输出端波形相同的条件 (即输入信号需要满足的条件)。
- B.36** [5] < B.8 > B.8 节图 B-8-8 描述了 MIPS 数据通路的寄存器文件实现。假设需要建立一个新的寄存器文件，但是只有两个寄存器和一个读端口，并且每个寄存器只有两位数据。重绘图 B-8-8，使得每根连接线仅与一位数据相连 (不像图 B-8-8 中那样，有些连接线为 5 位，有些则为 32 位)。采用 D 触发器重绘图中的寄存器。无需画出 D 触发器或多路选择器的具体实现。
- B.37** [10] < B.10 > 有个朋友想让你帮忙设计一个仿安全装置的“电子眼”。该设备由排成一行的三个灯组成，这三个灯分别受输出 Left、Middle 和 Right 控制，即当这三个信号当中的某一个有效时，对应的灯被点亮。每次仅有一个灯被点亮，并且灯光先从左到右“移动”，然后再从右到左，这样可以吓跑那些误以为该设备正在监控其行踪的小偷。画出用于控制该“电子眼”的有限状态机图示。需要注意的是，“眼睛”的移动速率受时钟速度 (不应当过高) 的控制，并且根本没有输入信号。
- B.38** [10] < B.10 > 为上题中的有限状态机分配状态编码，并写出对应于每个输出信号的包含下一状态位的逻辑表达式。
- B-86**
- B.39** [15] < B.2, B.8, B.10 > 用 3 个 D 触发器和若干逻辑门构造一个 3 位计数器。计数器的输入包括复位信号 reset，计数值增加信号 inc。计数结果作为计数器输出。当计数值为 7 并且继续增加时，计数值应当重新归零。
- B.40** [20] < B.10 > 格雷码是具有这样性质的二进制序列，即该序列当中相邻的两个编码最多有一位不同。例如，下面是一个 3 位格雷码序列：000, 001, 011, 010, 110, 111, 101，和 100。用三个 D 触发器和一个 PLA 实现一个 3 位格雷码计数器，要求该计数器具有两个输入信号：复位信号 reset 和增量信号 inc，其中 reset 信号将计数器设为 000，inc 信号将使计数器进入序列中的下一状态。需要注意的是，该编码序列是循环的，所以 100 的下一个值为 000。
- B.41** [25] < B.10 > 我们希望在 B.10 节的交通灯例子中添加一个黄灯。通过将时钟频率改为 0.25Hz (时钟周期 4 秒，即黄灯的持续时间)。为防止绿灯和红灯循环过快，我们加入了一个 30 秒的计时器。该计时器只有一个输入信号 TimerReset，该信号用于对计时器进行重启；计时器输出信号 TimerSignal 表示 30 秒时间已经过去。而且，为了把黄灯包含进去，我们必须重新定义交通信号。我们通过给每个灯定义两个输出信号 (green 和 yellow) 来实现。如果输出 NSgreen 有效，绿灯被点亮；如果输出 NSyellow 有效，黄灯被点亮。如果两个信号都无效，则红灯被点亮。green 和 yellow 信号不能同时有效，否则美国司机见到之后肯定会感到困惑，即使欧洲司机明白其中的含义！画出上述改进后的控制器对应的有限状态机图示。状态名称不要和输出信号同名。
- B.42** [15] < B.10 > 写出练习题 B.41 中交通灯控制器的下一状态表和输出函数表。
- B.43** [15] < B.2, B.10 > 为练习题 B.41 的交通灯分配状态编码，并根据练习题 B.42 的表格写出每个输出信号的逻辑表达式，包括下一状态的输出。
- B.44** [15] < B.3, B.10 > 用 PLA 实现练习题 B.43 的逻辑表达式。

01 小测验答案

B. 2 否。如果 $A = 1$, $C = 1$, $B = 0$, 那么第一个为真, 第二个为假。

B. 3 C

B. 4 全部相同。

B. 4 $A = 0$, $B = 1$ 。

B. 5 2。

B. 6 1。

B. 8 c。

B. 10 b。

B. 11 b。

[B-87]