第 3 章

CHAPTER 3

组合逻辑电路设计基础

本章学习导言

组合逻辑电路在任一时刻的输出状态只取决于该时刻的输入状态的组合，而与电路以前的状态无关，即电路只由门电路组成，没有记忆单元，也没有反馈电路。第 2 章介绍的简单逻辑运算符可用于描述基本逻辑单元构成的门级设计，实际已经是基本的组合逻辑电路设计内容。本章主要介绍由中等规模组件构成组合电路的 HDL 描述，例如：加法器、比较器和多路复用器等。本章首先结合实例对 Verilog HDL 行为描述的常用语法进行介绍，包括 always 块、if-else 语句和 case 语句、参数和常数等，并通过一些常见组合逻辑电路实例来展示常用组合电路设计。

本章的目的是给出组合逻辑电路基本概念，在介绍 Verilog HDL 用于组合逻辑设计的条件语句和循环语句等基本要素后，分别给出比较器、多路复用器等常见组合逻辑电路单元的介绍及其 Verilog HDL 实现。

这些常用组合逻辑电路单元在逻辑系统中出现频率高，算得上搭建逻辑系统的最基本积木，熟练掌握这些单元对于复杂逻辑系统设计以及逻辑系统设计优化均有重要作用。

3.1 组合电路中的 always 块

在进行较为复杂的逻辑电路设计时，为了提高设计效率，通常采用较为抽象的行为描述，Verilog HDL 使用一些顺序执行的过程语句来进行行为描述。这些语句封装在一个 always 块或 initial 块中，initial 块仅在仿真开始的时候执行一次，只有 always 块能够进行综合，生成能够执行逻辑运算或控制的电路模块。在本部分中重点介绍 always 块。

always 块可以看成一个包含内部过程描述语句的黑盒子，过程语句包含多种结构，但是很多没有对应的硬件，编码不佳的 always 块通常会导致不必要的复杂实现甚至根本无法综合。

本部分主要关注可综合的组合电路设计，本节将讨论内容限制在 3 种类型的语句：过程赋值、条件语句和循环语句。

3.1.1 基本语法格式

带敏感信号列表的 always 块的简化使用格式如下：

```
always @([敏感信号列表])
begin [可选的模块名]
      [可选的局部变量声明];
      [顺序执行语句];
      [顺序执行语句];
…
end
```

敏感信号列表是 always 块响应的信号和事件列表，对于组合电路，应该包含所有的输入信号。如有两个或者两个以上的信号，在 Verilog HDL 1364-1995 标准中，它们之间可以用关键字 or 来连接，如：

```
always @(a or b or c)
```

Verilog HDL 1364-2001 标准中，可以使用“,”来分隔，如：

```
always @(a, b, c)
```

本书使用 Verilog HDL 1364-2001 标准。

@[敏感信号列表]项实际上是一个时序控制结构，它是可综合 always 块中的唯一时序控制结构。always 块体包含任意数目的过程语句，当 always 块体只有一条语句时，定界符 begin 和 end 可以省略。

敏感信号可分为两种类型：一种是电平敏感型，另一种是边沿敏感型。每个 always 过程一般只由一种类型的敏感信号来触发，而不要混合使用。对于组合电路，一般采用电平敏感方式；对于时序电路，一般由时钟边沿触发，Verilog HDL 提供了 posedge 和 negedge 两个关键字来分别描述上升沿和下降沿。

always 块可以看作一个复杂的电路部分，可以被挂起和激活。当敏感列表中的信号发生变化或某一事件发生时，该部分被激活并执行内部过程语句，由于没有其他时序控制结构，执行会一直持续到 end 模块才会终止，因此，always 块实际上是个“无限循环”，每次执行由敏感信号列表触发。

3.1.2　过程赋值

过程赋值只能用在 always 块或 initial 块中，有两种赋值方式：阻塞赋值和非阻塞赋值。其基本语法格式如下所示。

```
阻塞赋值:[变量名] = [表达式];
非阻塞赋值:[变量名] <= [表达式];
```

在阻塞赋值中，在执行下一条语句前，一个表达式的值只能赋给一个变量，可以理解为赋值阻断了其他语句的执行，与 C 语言中的正常变量赋值行为相似。在非阻塞赋值中，表达式的值在 always 块结束时进行赋值，这种情况下赋值没有阻断其他语句的执行。

Verilog 的初学者经常会混淆阻塞赋值和非阻塞赋值，不能正确理解它们的区别可能会导致意外的行为或竞争条件，它们的基本使用原则是：

(1) 组合电路使用阻塞赋值；

(2) 时序电路使用非阻塞赋值。

本章关注的是组合电路，因此只使用阻塞赋值语句。

3.1.3　变量的数据类型

在过程赋值中，一个表达式只能赋给一种变量数据类型的输出，这些变量数据类型是 reg

型、integer 型、real 型、time 型和 realtime 型。reg 数据类型和 wire 数据类型类似，但用于过程赋值，integer 数据类型表示固定大小（通常是 32 位）的有符号二进制补码格式，由于大小固定，在综合中通常不用该数据类型，其他几种数据类型用于建模和仿真，无法被综合。

3.1.4 简单实例

本节用两个简单例子来说明 always 块和阻塞赋值的用法和行为。

1. 1 位比较器

此处可以用 always 块来设计简单的 1 位比较器电路，代码如例 3.1 所示。

例 3.1 1 位比较器的 always 块实现

```
module eq1_always
  (
    input i0, i1,
    output reg eq          // eq 声明为 reg 类型
  );
  reg p0, p1;              // p0 和 p1 声明为 reg 类型
  // i0 和 i1 必须在敏感信号列表中
  always @(i0, i1) begin
    //语句的顺序是很重要的
    p0 = ~i0 & ~i1;
    p1 = i0 & i1;
    eq = p0 | p1;
  end
endmodule
```

因为 eq、p0 和 p1 信号在 always 块内赋值，它们都声明为 reg 数据类型，敏感列表包括 i0 和 i1，并由逗号隔开，当其中一个发生变化时，always 块则被激活。3 条阻塞赋值语句顺序执行，和 C 程序中的语句非常类似，语句的顺序十分重要，在使用前 p0 和 p1 必须赋值。

要正确建立所需的行为模型，组合电路的敏感列表必须包含所有的输入信号，忽略一个信号就可能导致综合和仿真结果不一致，在 Verilog HDL 1364-2001 中，可以用下面的符号来隐式地包含所有输入信号，在本书中，组合电路描述使用这种结构。

```
always @( * )
```

其实，持续赋值语句和过程赋值语句的行为大不相同。此处以一个三输入的与门来说明它们的区别。

2. 过程赋值语句和持续赋值语句的比较

例 3.2 中的代码，使用的是过程赋值语句，它完成的功能是执行 a、b 和 c 的逻辑与运算（即 a&b&c），综合的电路如图 3.1(a)所示。

例 3.2 使用一个变量的与电路

```
module and_block_assign
 (
  input  a , b , c,
  output reg y
  );
  always @ ( * )begin
    y = a ;
    y = y&b;
    y = y&c;
  end
endmodule
```

如果采用类似例 3.3 中的持续赋值语句，则描述结果不正确。在例 3.3 的代码中，每条持续赋值语句综合一部分电路，左端 3 次出现的 y 表明 3 个输出捆绑在一起，对应的电路如图 3.1(b)所示，这显然不是想要的设计结果。

例 3.3　与电路的不正确代码

```
module and_cont_assign
   (
   input a,b,c,
   output y
  );
   assign y = a;
   assign y = y & b;
   assign y = y & c;
endmodule
```

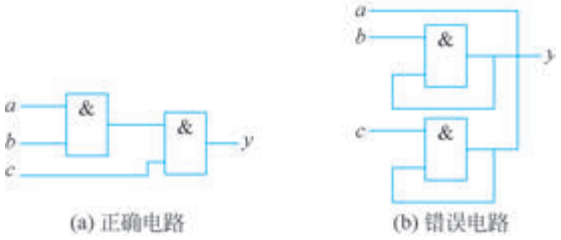


图 3.1　正确和不正确的代码段推导出的电路

3.2　条件语句

条件语句有 if-else 语句和 case 语句两种，它们都是顺序语句，应该放在 always 块内使用。以下分别结合实例对这两种语句进行介绍。

3.2.1　if-else 语句

1. 语法

if-else 语句的简化语法如下：

```
if ([表达式])
  begin
      [顺序执行语句];
      [顺序执行语句];
  end
else
  begin
    [顺序执行语句];
    [顺序执行语句];
  end
```

[表达式]项一般为逻辑表达式或关系表达式，也可以是 1 位的变量。语句先对表达式进行判断，如果表达式为真，则执行下面分支的语句，否则执行 else 分支的语句。else 分支具有选择性，可以省略。如果分支里只有一条语句，则定界符 begin 和 end 可以省略。

多条 if-else 语句可以“级联”，以判断多个布尔条件并建立优先级，如下所示：

```
if  ([表达式 1])
...
else if  [表达式 2]
else if  [表达式 3]
...
else
```

2. 实例

用两个简单例子说明 if-else 语句的用法。

1）四输入优先编码器

四输入优先编码器有 4 个优先级：$r[3]$、$r[2]$、$r[1]$和 $r[0]$。作为一组四输入信号，$r[3]$优先级最高，输出是最高优先级输入的二进制代码，四输入优先编码器的功能表如表 3.1 所示，HDL 代码如例 3.4 所示。

表 3.1 四输入优先编码器功能表

输入				输出		
$r[3]$	$r[2]$	$r[1]$	$r[0]$	$y[2]$	$y[1]$	$y[0]$
1	x	x	x	1	0	0
0	1	x	x	0	1	1
0	0	1	x	0	1	0
0	0	0	1	0	0	1
0	0	0	0	0	0	0

例 3.4 使用 if-else 语句的四输入优先编码器

```
module prio_encoder_if
   (
   input [3:0] r,
   output reg [2:0]  y
   );

   always @( * )
     if (r[3] == 1'b1)          // 可以写成 (r[3])
        y = 3'b100;
     else if (r[2] == 1'b1)     // 可以写成 (r[2])
        y = 3'b011;
     else if (r[1] == 1'b1)     // 可以写成 (r[1])
        y = 3'b010;
     else if (r[0] == 1'b1)     // 可以写成 (r[0])
        y = 3'b001;
     else
        y = 3'b000;
endmodule
```

代码首先检查信号 $r[3]$的请求，如果 $r[3]$为 1 则将 100 赋给 y，如果 $r[3]$为 0，则继续检查信号 $r[2]$的请求并重复以上过程，直至所有请求信号检查完毕。注意当 $r[3]$为 1 时布尔表达式($r[3]$==1'b1)为真，由于在 Verilog 中真值也可表示为 1'b1，因此表达式也可以写成($r[3]$)。

2）2-4 译码器

根据 n 位输入的信号，一个 $n-2^n$ 二进制译码器将 2^n 位输出中的 1 位置位，2-4 译码器的真值表如表 3.2 所示，该电路包括控制信号端 en，用于使能译码功能，HDL 代码如例 3.5 所示。

表 3.2 带使能的 2-4 译码器的真值表

en	$a[1]$	$a[0]$	y
0	x	x	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

例 3.5 使用 if-else 语句实现的二进制译码器

```
module decoder_2_4_if
   (
   input [1:0] a,
   input en,
   output  reg [3:0] y
   );
  always @( * )
     if (en == 1'b0)
       y = 4'b0000;
     else if (a == 2'b00)
       y = 4'b0001;
     else if (a == 2'b01)
       y = 4'b0010;
     else if (a == 2'b10)
       y = 4'b0100;
     else
       y = 4'b1000;
endmodule
```

代码首先检查使能位 en 是不是 1，如果条件为假(即 en 为 0)，则设置输出 y 为无效状态；如果条件为真，则检测 4 个二进制数的组合顺序，设置输出 y 的状态。注意布尔表达式(en==1'b0)也可以写成(~en)。

3.2.2 case 语句

相对于 if-else 语句只有两个分支而言，case 语句是一种多分支语句，故 case 语句可用于描述多条件分支电路，如译码器、数据选择器、状态机、微处理器指令译码等。case 语句有 case、casez 和 casex 这 3 种表示方式，以下分别进行介绍。

1. case 语句语法

case 语句的简化语法如下：

```
case ([表达式])
    [分支项 1]:
       begin
          [顺序执行语句] ;
          [顺序执行语句] ;
       ...
       end
    [分支项 2]:
       begin
          [顺序执行语句] ;
          [顺序执行语句] ;
       ...
       end
     [分支项 3]:
       begin
          [顺序执行语句] ;
          [顺序执行语句];
          ...
        end
       ...
     default:
        begin
           [顺序执行语句] ;
           [顺序执行语句];
        end
endcase
```

case 语句是一条多路决策语句，其将 case([表达式])和多个[分支项]表达式进行比较，程序跳入与当前[表达式]相等的[分支项]对应的分支执行，如果有多条[分支项]匹配，则执行第一条匹配的分支，最后一条分支为可选的，关键字是 default，包含[表达式]未指定的所有值。如果一条分支中只有一条语句，则定界符 begin 和 end 可以省略。

2. case 语句实例

本实例采用和 if-else 语句例子中相同的优先编码器和译码器来说明 case 语句的用法。

1）2-4 译码器的 case 语句描述

2-4 译码器的真值表如表 3.2 所示，使用 case 语句的 HDL 代码如例 3.6 所示。

例 3.6　使用 case 语句实现的二进制译码器

```
module decoder_2_4_case
    (
    input  [1:0] a,
    input  en,
    output  reg  [3:0]  y
    );
    always @( * )
       case  ({en, a})
           3'b000, 3'b001, 3'b010, 3'b011: y = 4'b0000;
           3'b100: y = 4'b0001;
           3'b101: y = 4'b0010;
           3'b110: y = 4'b0100;
           3'b111: y = 4'b1000;          // 也可以使用 default 语句
       endcase
endmodule
```

如果某些值有相同的执行语句，可以将这多个值放入一组表达式中，如同例 3.6 中的“3'b000，3'b001，3'b010，3'b011：”语句，注意分支项表达式包含表达式{en，a}的所有可能值，default 语句省略。

2）四输入优先编码器的 case 语句描述

四输入优先编码器的功能表如表 3.1 所示，使用 case 语句的 HDL 代码如例 3.7 所示。

例 3.7　使用 case 语句的四输入优先编码器

```
module prio_encoder_case
    (
    input  [3: 0]  r,
    output  reg  [2:0]  y
    );

    always @( * )
       case (r)
             4'b1000, 4'b1001, 4'b1010, 4'b1011, 4'b1100, 4'b1101, 4'b1110, 4'b1111:
                  y = 3'b100;
             4'b0100, 4'b0101, 4'b0110, 4'b0111:
                     y = 3'b011;
             4'b0010, 4'b0011:
                     y = 3'b010;
          4'b0001:
                     y = 3'b001;
             4'b0000:              // 也可以使用 default 语句
                     y = 3'b000;
    endcase

endmodule
```

3. casex 和 casez 语句

除了常规的 case 语句，还有两种变体 casez 和 casex，在 casez 语句中，认为表达式中的 z 值和？是无关值(即对应位无须匹配)。在 casex 语句中，认为表达式中的 z、x 值和？为无关值，由于 z 和 x 可能出现在仿真中，因此设计者更倾向于采用？。

例如，之前的优先编码器可以用 casez 语句描述，如例 3.8 所示。

例 3.8 使用 casez 语句的四输入优先编码器

```
module prio_encoder_casez
    (
    input  [3:0]  r,
    output  reg  [2:0]  y
   );
   always @( * )
         casez (r)
            4'b1???: y = 3'b100;
            4'b01??: y = 3'b011;
            4'b001?: y = 3'b010;
            4'b0001: y = 3'b001;
            4'b0000: y = 3'b000;     // 也可以使用 default 语句
         endcase
endmodule
```

4. full case 和 parallel case

在 Verilog HDL 中，分支项表达式不需要包括 case 表达式的所有值，一些值也可以匹配不止一次，考查下面的 casez 语句：

```
reg [2:0] s
…
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   3'b000: y = 1'b1;
endcase
```

上述语句中，3'b111 在分支项表达式中匹配了两次(一次在 3'b111 中，一次在 3'b1?? 中)，由于第一次匹配先生效，如果 s 是 3'b111，则 y 的值是 1'b1，如果 s 是 3'b001、3'b010 或 3'b011，则没有匹配，y 保持之前的值。

当分支项表达式包括 case 表达式的所有二进制值时，则语句称为 full case 语句，对于组合电路，由于每个输入组合对应一个输出值，因此必须采用 full case 语句，未匹配的值可以用 default 语句表示，例如，之前的语句也可以改为：

```
casez (s)
    3'b111: y = 1'b1;
    3'b1??: y = 1'b0;
    default : y = 1'b1;
endcase
```

或

```
casez (s)
     3'b111: y = 1'b1;
     3'b1??: y = 1'b0;
     3'b000: y = 1'b1;
     default : y = 1'bx;
endcase
```

如果分支项表达式中的值互斥(即一个值只出现在一个项目表达式中)，则语句称为

parallel case 语句，例如，之前的 casez 语句不是 parallel case 语句，因为 3'b111 被匹配了两次，例 3.6 和例 3.7 中的 case 语句是 parallel case 语句。

很多综合软件工具都有 full case 指令和 parallel case 指令，使用这些指令时，所有 case 语句都被当成 full case 语句和 parallel case 语句并进行相应综合，Verilog HDL 1364-2001 也有相同的属性，使用这些指令会本质上覆盖原来的语义并使仿真和综合产生差异。本书用代码来表达这些条件而不是运用这些指令或属性。

3.3 循环语句

在 Verilog HDL 中共有 4 种类型的循环语句，用来控制语句的执行次数，具体如下。

(1) for：有条件的循环语句。

(2) repeat：连续执行一条语句 n 次。

(3) while：执行一条语句直到某个条件不满足，如果一开始条件就不满足，则语句一次也不执行。

(4) forever：连续执行语句；多用在“initial”块中，用来生成时钟等周期性波形。

3.3.1 for 语句

1. 语法

for 语句的一般形式为

```
for(表达式 1;表达式 2;表达式 3)
begin
    [顺序执行语句];
    [顺序执行语句];
    …
end
```

当顺序执行语句只有一条时，定界符 begin 和 end 可以省略。

它的执行过程具体如下。

(1) 先求解表达式 1。

(2) 再求解表达式 2，如果其值为真(非 0)，则执行步骤(3)；如果为假(0)，则结束循环，转到步骤(5)。

(3) 执行 begin/end 块中的顺序执行语句后，求解表达式 3。

(4) 转回步骤(2)继续执行。

(5) 执行 for 语句下面的语句。

2. 实例

用 for 语句实现两个 8 位二进制数的乘法操作，实现代码如例 3.9 所示。

例 3.9　两个 8 位二进制数相乘的 for 语句描述

```
module mult_for
   (
    input [8:1] op0,
    input [8:1] op1,
    output reg [16:1] result
   );
  integer i;
```

```
  always @( * ) begin
    result = 0;
    for(i = 1;i < = 8;i = i + 1)
      if(op1[i])
          result = result + (op0 << (i - 1));
  end
endmodule
```

3.3.2 repeat 语句

1. 语法

repeat 语句的使用格式如下：

```
repeat(表达式)
begin
     [顺序执行语句] ;
     [顺序执行语句] ;
     …
end
```

当顺序执行语句只有一条时，定界符 begin 和 end 可以省略。

在 repeat 语句中，其表达式通常为常量表达式，用来指定循环执行的次数。

2. 实例

用 repeat 语句实现两个 8 位二进制数的乘法操作，实现代码如例 3.10 所示。

例 3.10 两个 8 位二进制数相乘的 repeat 语句描述

```
module mult_repeat
   (
    input [8:1] op0,
    input [8:1] op1,
    output reg [16:1] result
   );

  reg [16:1] tempa;
  reg [8:1] tempb;

  always @( * )begin
     result = 0;
     tempa = op0;
     tempb = op1;
     repeat(8)begin
        if(tempb[1]) result =   result + tempa;
        tempa = tempa << 1;
        tempb = tempb >> 1;
     end
  end
endmodule
```

3.3.3 while 语句

1. 语法

while 语句的使用格式如下：

```
while(表达式)
begin
     [顺序执行语句] ;
     [顺序执行语句] ;
     …
end
```

当顺序执行语句只有一条时，定界符 begin 和 end 可以省略。

while 语句在执行时，首先判断表达式是否为真，如果为真，则执行后面的语句或语句块，然后再回头判断表达式是否为真，若为真，则再执行一遍后面的语句，如此不断，直到表达式不为真。因此，在执行语句中，必须有一条改变表达式值的语句。

2. 实例

用 while 循环实现两个 8 位二进制数的乘法操作，实现代码如例 3.11 所示。

例 3.11　两个 8 位二进制数相乘的 while 语句描述

```
module mult_while
   (
      input [8:1] op0,
      input [8:1] op1,
      output reg [16:1] result
   );
  interger i = 1;

  always @( * ) begin
     result = 0;
     while(i <= 8)
       if(op1[i]) result = result + (op0 << (i - 1));
       i = i + 1;
  end
endmodule
```

3.3.4　forever 语句

forever 语句的使用格式如下：

```
forever
begin
     [顺序执行语句];
     [顺序执行语句];
     …
end
```

当顺序执行语句只有一条时，定界符 begin 和 end 可以省略。

forever 循环语句连续不断地执行后面的语句或语句块，常用来产生周期性的波形，作为仿真激励信号。forever 语句一般用在 initial 过程语句中，如果用它来进行模块描述，可以使用 disable 语句进行中断。

3.4　always 块的一般编码原则

Verilog 用于建模和综合，当编写可综合代码时，需要知道不同的语言结构如何与硬件匹配，尤其是 always 块，因为变量和过程语句可以在模块内使用，需要谨记编写代码的目的是综合为硬件电路，而不是用 C 语言描述顺序算法，做不到这些经常会导致无法综合的代码、不必要的复杂实现，或者在仿真和综合之间产生差异。本节讨论一些常见错误并提出一些编码原则。

3.4.1　组合电路代码中的常见错误

组合电路代码中的常见错误主要包括：变量在多个 always 块中赋值、不完整的敏感信号

列表、不完整分支和不完整输出赋值等。以下分别讨论这些错误。

1. 变量在多个 always 块中赋值

在 Verilog 中，变量（出现在左端部分）在多个 always 块中赋值，例如，以下代码段中两个 always 块都含变量 y：

```
reg  y;
reg  a , b , clear ;
...
always @( * )
    if (clear) y = 1'b0 ;
always @( * )
    y = a&b;
```

尽管该代码作为异步电路是正确的，也可以进行仿真，但是无法综合，考虑每个 always 块可以认为是电路的一部分，以上代码表示 y 是两块电路的输出，并可以通过每部分进行更新，没有物理电路可以表示这种行为，因此代码不能综合，必须把赋值语句写入一个 always 块中，例如：

```
always @( * )
    if (clear)y = 1'b0;
    else y = a&b;
```

2. 不完整的敏感信号列表

对于组合电路，输出是输入的函数，因此任何输入信号的变化都会激活电路，这要求所有的输入信号应该包含在敏感信号列表中，例如两输入与门可以写成：

```
always @(a, b)          // a 和 b 都在敏感信号列表中
      y = a&b;
```

如果忘记包含 b，代码变成：

```
always @(a)       // b 不在敏感信号列表中
      y = a&b;
```

尽管代码仍然可以正确综合，但是行为改变了，当 a 发生变化时，always 块被激活，$a\&b$ 的值赋给 y，而当 b 发生变化时，由于 always 块对 b“不敏感”，仍然处于中止状态，y 仍保持之前的值不变，没有物理电路表示这种行为。大部分综合软件会给出警告信息并综合成与门电路，而仿真软件仍按语句表达的行为建模，因此会导致仿真与综合的结果不一致。

在 Verilog HDL 1364-2001 中，采用了特殊符号@(*)表示包含所有相关的输入信号，因此不会出现这种问题，好的做法是在组合电路中使用这个符号。

3. 不完整分支和不完整输出赋值

组合电路的输出是输入的函数，不应该包括任何内部状态（即存储器），always 块的一种常见错误就是综合出组合电路的意外存储器。Verilog 标准指定在 always 块中，变量如果没有赋值则保持原来的值，在综合时，这将导致内部状态（通过闭合反馈回路）或存储元件（锁存器）的产生。

为了防止 always 块中意外的存储器，所有输出信号在任何时候都应该赋予恰当值，不完整分支和不完整输出赋值是两种常见的导致意外存储器的错误。例如，在下面例子中，当表达式 $a>b$ 为真时，eq 没有赋值的话，会保持之前的值，因此会综合出相应的锁存器。

有两种方法来解决错误，第一种是加上 else 分支并明确给所有输出变量赋值，代码变成：

```
always @( * )
   if (a > b) begin
       gt = 1'b1;
       eq = 1'b0;
   end else if (a == b) begin
       gt = 1'b0;
       eq = 1'b1;
   end else begin
       gt = 1'b0;
       eq = 1'b0;
   end
```

另一种方法是在 always 块的起始部分，给每个变量赋默认值，以包含未指定的分支和未赋值变量，代码变成：

```
always @( * )begin
    gt = 1'b0;     // gt 的默认赋值
    eq = 1'b0;     // eq 的默认赋值
    if (a > b)gt = 1'b1;
    else if (a == b) eq = 1'b1;
end
```

如果 gt 和 eq 之后未赋值，则认为是 0。

如果 case 语句表达式的某些值未被分项表达式包含（即不是 full-case 语句），case 语句也会产生相同的错误，思考以下代码：

```
reg  [1:0] s;
...
case (s)
    2'b00: y = 1'b1;
    2'b10: y = 1'b0;
    2'b11: y = 1'b1;
endcase
```

没有任何分支包含 2'b01，如果 s 出现这种组合，y 会保持之前的值，这会综合出意外的锁存器。为了解决这种问题，设计者必须保证任何时候 y 都被赋值。一种方法是在末尾采用关键字 default 以包含所有未指定值，也可以用以下代码替代最后一个分支项表达式：

```
case (s)
    2'b00: y = 1'b1;
    2'b10: y = 1'b0;
    default: y = 1'b1;
endcase
```

或者用无关值增加一个项目表达式：

```
case (s)
    2'b00: y = 1'b1;
    2'b10: y = 1'b0;
    2'b11: y = 1'b1;
    default: y = 1'bx;
endcase
```

另外，也可以在 always 块的开始部分赋给一个默认值：

```
y = 1'b0;
case (s)
    2'b00: y = 1'b1;
    2'b10: y = 1'b0;
    2'b11: y = 1'b1;
endcase
```

3.4.2 组合电路中 always 块的使用原则

always 块是一种灵活强大的语言结构，但是必须谨慎使用，设计正确高效的电路并避免任何综合和仿真的差异性，以下是描述组合电路的编码原则。

(1) 只在一个 always 块中对某个变量进行赋值。

(2) 组合电路采用阻塞赋值。

(3) 在敏感列表中使用@(*)自动包含所有输入信号。

(4) 确保包含 if 和 case 语句的所有分支。

(5) 确保所有分支的输出都被赋值。

(6) 用代码描述 full case 和 parallel case，而不用软件指令和属性。

(7) 了解不同控制结构综合电路的类型。

(8) 思考硬件而不是 C 语言代码。

注：一种满足第(4)条和第(5)条原则的方法是在 always 块开始时给输出赋默认值。

3.5 常数和参数

3.5.1 常数

HDL 代码经常在表达式和数组边界中使用常数值，这些值在模块内固定不变，好的设计是用符号常数代替“固定文本”，这使得代码清晰并有助于以后的维护和修改。在 Verilog HDL 中，可以用关键字 localparam(局部参数)声明常数，例如，可以声明数据总线的位宽和范围：

```
localparam  DATA_WIDTH = 8 ,
            DATA_RANGE = 2 ** DATA_WIDTH - 1;
```

或定义符号端口名：

```
localparam  UART_PORT = 4'b0001,
            LCD_PORT = 4'b0010,
            MOUSE_PORT = 4'b0100;
```

声明中的表达式如 2 ** DATA_WIDTH－1，在处理前已经求值，因此不综合物理电路，本书用大写字母串表示常数。

一个例子可以很好地说明常数的使用，考虑带有进位的加法器，一种方法是手工扩展输入 1 位，执行常规加法，截取和的最高位作为进位，代码如例 3.12 所示。

例 3.12 固定位宽的加法器描述

```
module adder_carry_hard_lit
   (
   input wire [3:0] a, b,
   output wire [3:0] sum,
   output wire cout         // 进位
    );

   wire [4:0] sum_ext ;

   assign sum_ext = {1'b0, a} + {1'b0, b};
   assign sum = sum_ext [3:0] ;
   assign  cout = sum_ext [4] ;
endmodule
```

代码描述的是4位加法器，固定值如3和4用以表示范围，例如wire[4:0]、sum_ext[3:0]，和最高位，如sum_ext[4]。如果想改成8位加法器，这些字都要手工修改，如果代码很复杂而且这些文字出现在很多地方，这将是一个烦琐而且容易出错的过程。

为了增强可读性，可以采用符号常数 N 来表示加法器的位数，修改后代码如例3.13所示。

例3.13　使用符号常数的加法器描述

```
module adder_carry_local_par
    (
    input [3:0] a, b,
    output [3:0] sum,
    output cout
    );
    // 常数声明
    localparam  N = 4,
                N1 = N-1;
    wire [N:0] sum_ext;

    assign  sum_ext = {1'b0, a} + {1'b0, b};
    assign  sum = sum_ext [N1:0] ;
    assign  cout = sum_ext [N] ;

endmodule
```

常数使代码更容易理解和维护。

3.5.2　参数

Verilog模块可以实例化为组件并成为更大设计的一部分。Verilog提供一种结构称为parameter的结构，向模块传递信息，这种机制使得模块多功能化并能重复使用。参数在模块内不能改变，因此功能与常数类似。

在Verilog HDL 1364-2001中，参数声明部分可以在开头，在端口声明之前，其简化语法如下。

```
module [模块名]
# (
parameter [参数名] = [默认值],
          [参数名] = [默认值];
  )
  (
    ...
);
```

例如，之前的加法器可以改成采用参数的加法器，如例3.14所示。

例3.14　使用参数的加法器描述

```
module adder_carry_ para
    #(parameter N=4)
    (
    input  [N-1:0] a, b,
    output  [N-1:0] sum,
    output cout
    );
    //常数声明
    localparam  N1 = N-1;
    wire  [N:0] sum_ext;
    assign  sum_ext = {1'b0, a} + {1'b0, b};
```

```
    assign  sum = sum_ext[N1:0];
    assign  cout = sum_ext[N];
endmodule
```

参数 N 声明为默认值 4，N 声明之后，就可以像常数一样在端口声明和模块体中使用。

如果之后加法器在其他代码中作为组件使用，可以在组件实例化中给参数赋所需的值并将原来的默认值覆盖。如果省略了参数赋值，则采用默认参数，组件实例化的用法如例 3.15 所示。

例 3.15　加法器实例化

```
module adder_insta
    (
    input  [3:0] a4, b4,
    output [3:0] sum4,
    output c4,
    input  [7:0] a8, b8,
    output [7:0] sum8,
    output c8
    );

    // 实例化 8 位加法器
    Adder_carry_para #(. N(8)) unit1
       (. a(a8), . b(b8), . sum(sum8), . cout (c8));

    //实例化 4 位加法器
    Adder_carry_para unit2
       (.a(a4), .b(b4),.sum(sum4),.cout(c4));

endmodule
```

参数提供了一种创建可扩展代码的机制，可调整电路的位宽以适应特定的需求，这使代码移植性更好，有利于设计重用。

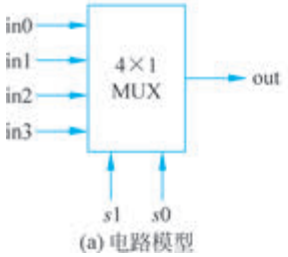
3.6　设计实例

本节给出一些常用的组合电路的设计实例，包括多路选择器、比较器、译码器、编码器和编码器等。

3.6.1　多路选择器

多路选择器是一个多输入、单输出的组合逻辑电路，一个 n 输入的多路选择器就是一个 n 路的数字开关，可以根据通道选择控制信号的不同，从 n 个输入中选取一个输出到公共的输出端。这里以一个 4 选 1 多路选择器为例，介绍多路选择器的 Verilog HDL 描述。

4 选 1 多路选择器的电路模型和真值表如图 3.2 所示。其中，in0、in1、in2 和 in3 是 4 个输



(a) 电路模型

s1	s0	out
0	0	in0
0	1	in1
1	0	in2
1	1	in3

(b) 真值表

图 3.2　4 选 1 多路选择器的电路模型和真值表

入端口，$s1$ 和 $s0$ 是通道选择控制信号端口，out 是输出端口。当 $s1$ 和 $s0$ 取值分别为 00、01、10 和 11 时，输出端 out 将分别输出 in0、in1、in2 和 in3。例 3.16 和例 3.17 分别是 4 选 1 多路选择器的 if-else 语句描述和 case 语句描述。4 选 1 多路选择器的仿真波形如图 3.3 所示。

例 3.16　4 选 1 多路选择器的 if-else 语句描述

```
module mux41_if
  (
    input  in0,in1,in2,in3,
    input  s0,s1,
    output reg out          // out 声明为 reg 类型
   );
  always @( * )begin
     if ({s1,s0} == 2'b00)       out = in0;
     else if({s1,s01} == 2'b01)  out = in1;
     else if({s1,s01} == 2'b10)  out = in2;
     else                        out = in3;
  end
endmodule
```

例 3.17　4 选 1 多路选择器的 case 语句描述

```
module mux41_case
  (
    input  in0,in1,in2,in3,
    input  s0,s1,
    output reg out           // out 声明为 reg 类型
   );
  always @( * )begin
     case ({s1,s0})
        2'b00:  out = in0;
        2'b01:  out = in1;
        2'b10:  out = in2;
        default: out = in0;
     endcase
  end
endmodule
```

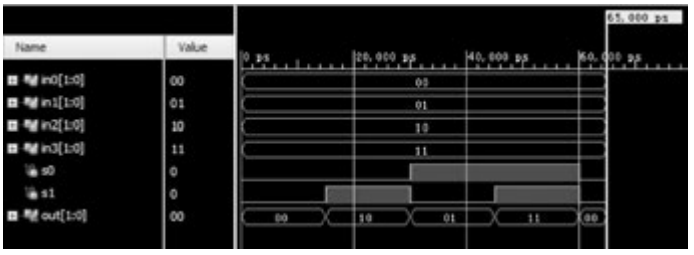


图 3.3　4 选 1 多路选择器的仿真波形

3.6.2　比较器

数值大小比较在计算逻辑中是常用的一种方法，比较器就是用来完成这种数值大小比较逻辑的组合电路。一位二进制数比较器是它的基础，其电路真值表如表 3.3 所示。其中，in0 和 in1 是一位输入比较信号，gt、eq 和 lt 分别是两个输入信号大小的比较结果。例 3.18 是其 Verilog HDL 描述。

注意：在例 3.18 的程序清单中，在 always 块内 if-else 语句之前，对 gt、eq 和 lt 都赋值为 0。这样做的目的是保证每个输出都被分配一个值。如果没有这样做，Verilog 会认为用户不

希望让它们的值改变，系统将会自动生成一个锁存器，那么得到的电路就不再是一个组合电路了。其仿真波形如图 3.4 所示。

表 3.3 一位二进制数比较器的真值表

in0	in1	gt(in0＞in1)	eq(in0＝in1)	lt(in0＜in1)
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

例 3.18 一位二进制数比较器的描述

```
module comp_1
  (
    input  in0, in1,
    output  reg  gt, eq, lt
   );
  always @( * )begin
       gt = 0;eq = 0;lt = 0;
       If (in0 > in1) gt = 1;
       If (in0 == in1) eq = 1;
       If (in0 < in1) lt = 1;
  end
endmodule
```

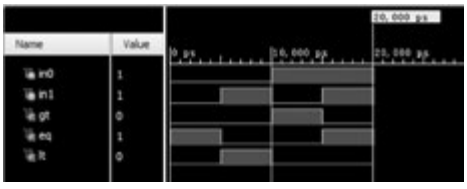


图 3.4 一位二进制数比较器的仿真波形

为了使用方便，可以使用参数来实现一个输入数据位数可变的 N 位二进制数比较器，其 Verilog HDL 描述如例 3.19 所示，这段代码在仿真时，默认 $N=8$。

例 3.19 N 位二进制数比较器的描述

```
module comp_N
  #(parameter N = 8)
  (
      input [N-1:0] in0, in1,
      output  reg  gt, eq, lt
   );

  always @( * )begin
     gt = 0;
     eq = 0;
     lt = 0;
     if(in0 > in1)gt = 1;
     if(in0 == in1)eq = 1;
     if(in0 < in1)lt = 1;
  end
endmodule
```

3.6.3 译码器和编码器

1. 3-8 译码器

译码器电路有 n 个输入和 2^n 个输出，每个输出都对应着一个可能的二进制输入。通常

情况下，一次只能有一个输出有效。3-8 译码器的真值表如表 3.4 所示。其中，in[2:0]是译码器的三位逻辑输入，y[7:0]是其 8 位译码输出，每次只有一位输出为高电平。3-8 译码器的 Verilog HDL 描述如例 3.20 所示，其仿真波形如图 3.5 所示。

表 3.4　3-8 译码器的真值表

in2	in1	in0	y0	y1	y2	y3	y4	y5	y6	y7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

例 3.20　3-8 译码器的描述

```
module decode_3_8
   (
    input [2:0] in,
    output  reg [7:0]  y
   );

   always @( * )begin
      case (in)
         3'b000 :  y = 8'b00000001;
         3'b001 :  y = 8'b00000010;
         3'b010 :  y = 8'b00000100;
         3'b011 :  y = 8'b00001000;
         3'b100 :  y = 8'b00010000;
         3'b101 :  y = 8'b00100000;
         3'b110 :  y = 8'b01000000;
         3'b111 :  y = 8'b10000000;
      endcase
   end
endmodule
```



图 3.5　3-8 译码器的仿真波形

2. 8-3 编码器

编码器是译码器的反向器件，它有 2^n 个输入和 n 个输出，输出的 n 位二进制数代表着输入标号为 2^n 的那一路输入是高电平。8-3 编码器的真值表如表 3.5 所示，在真值表中，每行输入只有一个为 1，其余的输入都为 0。假设剩下的 248 种输入的可能组合值都不会发生，因为它们会产生不希望的输出。不过，为了增加代码的鲁棒性，增加一位 valid 输出，用来指示输入了不希望的逻辑组合。8-3 编码器的 Verilog HDL 描述如例 3.21 所示，其仿真波形如图 3.6 所示。

表 3.5　8-3 编码器的真值表

y0	y1	y2	y3	y4	y5	y6	y7	in2	in1	in0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

例 3.21　8-3 编码器的描述

```
module encode_8_3
   (
    input [7:0] in,
    output  reg [2:0]  encode_out,
    output  reg  valid
   );
  always @( * )begin
      valid = 1;
      case (in)
        8'b00000001 :  encode_out = 3'b000;
        8'b00000010 :  encode_out = 3'b001;
        8'b00000100 :  encode_out = 3'b010;
        8'b00001000 :  encode_out = 3'b011;
        8'b00010000 :  encode_out = 3'b100;
        8'b00100000 :  encode_out = 3'b101;
        8'b01000000 :  encode_out = 3'b110;
        8'b10000000 :  encode_out = 3'b111;
        default     :  valid = 0;
      endcase
  end
endmodule
```



图 3.6　8-3 编码器的仿真波形

3. 8-3 优先编码器

8-3 编码器要求在任何时候都只有一个输入为 1，如果输入有多于一个的 1，编码器只能输出一个无效信号来指示当前的状态。优先编码器的输入可以同时包括多于一个的 1，输出状态会按照输入的优先级来确定。

8-3 优先编码器的真值表如表 3.6 所示，其中每一行 1 的左边的 x 代表不确定的状态，也就是说，不论 x 的值是 1 还是 0，都不会影响优先编码器的输出。从真值表可以看出，输入的优先级顺序依次是 $y7$、$y6$、$y5$、$y4$、$y3$、$y2$、$y1$ 和 $y0$。实现代码如例 3.22 所示，8-3 优先编码器的仿真波形图如图 3.7 所示。

表 3.6　8-3 优先编码器的真值表

y0	y1	y2	y3	y4	y5	y6	y7	in0	in1	in2
1	0	0	0	0	0	0	0	0	0	0
x	1	0	0	0	0	0	0	0	0	1
x	x	1	0	0	0	0	0	0	1	0
x	x	x	1	0	0	0	0	0	1	1
x	x	x	x	1	0	0	0	1	0	0
x	x	x	x	x	1	0	0	1	0	1
x	x	x	x	x	x	1	0	1	1	0
x	x	x	x	x	x	x	1	1	1	1

例 3.22　8-3 优先编码器的描述

```
module prio_encode_8_3 (
    input [7:0] in,
    output  reg [2:0]  encode_out
   );
  always @( * )begin
      casez (in)
        8'b1??????? :  encode_out = 3'b000;
        8'b01?????? :  encode_out = 3'b001;
        8'b001????? :  encode_out = 3'b010;
        8'b0001???? :  encode_out = 3'b011;
        8'b00001??? :  encode_out = 3'b100;
        8'b000001?? :  encode_out = 3'b101;
        8'b0000001? :  encode_out = 3'b110;
        8'b10000001 :  encode_out = 3'b111;
      endcase
  end
endmodule
```

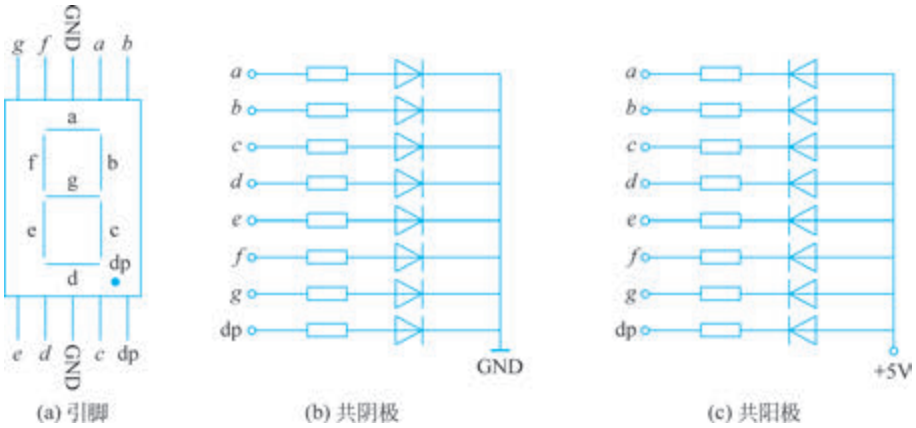


图 3.7　8-3 优先编码器的仿真波形

3.6.4　十六进制数七段 LED 显示译码器

七段 LED 显示器也称为七段数码管，其示意图如图 3.8 所示，包括 7 个 LED 和一个圆形 LED 小数点。按 LED 单元的连接方式可以分为共阳数码管和共阴数码管，共阳数码管指将所有发光二极管的阳极接到一起形成公共阳极(COM)的数码管，共阳数码管在应用时应将公共极接到逻辑高电平，当某一字段发光二极管的阴极为低电平时，相应字段就点亮，当某一字段的阴极为高电平时，相应字段就不亮。共阴数码管指将所有发光二极管的阴极接到一起形成公共阴极(COM)的数码管，共阴数码管在应用时应将公共极接到逻辑低电平，当某一字段发光二极管的阳极为高电平时，相应字段就点亮，当某一字段的阳极为低电平时，相应字段就不亮。

本节设计的十六进制数七段 LED 显示译码器，驱动共阳数码管，将输入的 4 位十六进制数转换为驱动七段 LED 显示管的控制逻辑，为了完整起见，同时列出 1 位小数点的控制 dp，其功能表如表 3.7 所示，Verilog HDL 描述如例 3.23 所示，图 3.9 是其仿真波形图。



(a) 引脚　(b) 共阴极　(c) 共阳极

图 3.8　七段 LED 显示器示意图

表 3.7　十六进制数七段 LED 显示译码器功能表

hex[3:0]	sseg[6:0]
0000	0000001
0001	1001111
0010	0010010
0011	0000110
0100	1001100
0101	0100100
0110	0100000
0111	0001111
1000	0000000
1001	0000100
1010	0001000
1011	1100000
1100	0110001
1101	1000010
1110	0110000
1111	0111000

例 3.23　十六进制数七段 LED 显示译码器的描述

```
module hex_7seg
    (
    input [3:0] hex,
    input  dp,
    output reg [7:0]  sseg
   );
   always @( * )begin
       case (hex)
           4'h0: sseg [6:0] = 7'b0000001;
           4'h1: sseg [6:0] = 7'b1001111;
           4'h2: sseg [6:0] = 7'b0010010;
           4'h3: sseg [6:0] = 7'b0000110;
           4'h4: sseg [6:0] = 7'b1001100;
           4'h5: sseg [6:0] = 7'b0100100;
           4'h6: sseg [6:0] = 7'b0100000;
           4'h7: sseg [6:0] = 7'b0001111;
```

```
            4'h8: sseg[6:0] = 7'b0000000;
            4'h9: sseg[6:0] = 7'b0000100;
            4'ha: sseg[6:0] = 7'b0001000;
            4'hb: sseg[6:0] = 7'b1100000;
            4'hc: sseg[6:0] = 7'b0110001;
            4'hd: sseg[6:0] = 7'b1000010;
            4'he: sseg[6:0] = 7'b0110000;
            4'hf: sseg[6:0] = 7'b0111000;
        endcase
        sseg[7] = dp;
    end
endmodule
```



图 3.9　十六进制数七段 LED 显示译码器仿真波形

3.6.5　BCD 码转换器

BCD 码，即将十进制数 0～9 用二进制码 0000～1001 表示。

例如：十进制数 12 用两个 BCD 码 0001 0010 来表示；十进制数 12 的十六进制（二进制）表示为 C（1100）。

把单个十六进制数（0～F）转换成两个 BCD 码的真值表如表 3.8 所示，这等效于把 4 位二进制数转换为 5 位二进制数，其 Verilog HDL 的描述如例 3.24 所示。

表 3.8　单个十六进制数转换为两个 BCD 码的真值表

bin[3:0]	bcd5[4:0]
0000	00000
0001	00001
0010	00010
0011	00011
0100	00100
0101	00101
0110	00110
0111	00111
1000	01000
1001	01001
1010	10000
1011	10001
1100	10010
1101	10011
1110	10100
1111	10101

例 3.24　单个十六进制数转换为两个 BCD 码的描述

```
module bin_bcd4
   (
    input [3:0] bin4,
    output  reg [4:0]  bcd5
   );
   always @( * )begin
        case (bin4)
            4'h0: bcd5 = 5'b00000;
            4'h1: bcd5 = 5'b00001;
            4'h2: bcd5 = 5'b00010;
            4'h3: bcd5 = 5'b00011;
            4'h4: bcd5 = 5'b00100;
            4'h5: bcd5 = 5'b00101;
            4'h6: bcd5 = 5'b00110;
            4'h7: bcd5 = 5'b00111;
            4'h8: bcd5 = 5'b01000;
            4'h9: bcd5 = 5'b01001;
            4'ha: bcd5 = 5'b10000;
            4'hb: bcd5 = 5'b10001;
            4'hc: bcd5 = 5'b10010;
            4'hd: bcd5 = 5'b10011;
            4'he: bcd5 = 5'b10100;
            4'hf: bcd5 = 5'b10101;
        endcase
   end
endmodule
```

设计任意数目输入的 BCD 码转换器的一种方法是左移加 3 的算法。这里以输入为 8 位二进制码为例来介绍这种算法的基本步骤，具体如下。

（1）左移要转换的二进制码 1 位。

（2）左移之后，BCD 码分别置于百位、十位、个位。

（3）如果移位后所在的 BCD 码列大于或等于 5，则对该值加 3。

（4）继续左移的过程直至全部移位完成。

表 3.9 直观展示了采用左移加 3 算法把十六进制数 0xFF 转换成 BCD 码的过程。8 位二进制数转换为 BCD 码转换器的实现代码如例 3.25 所示，仿真结果如图 3.10 所示。

表 3.9　十六进制数 0xFF 转换为 BCD 码的过程

操作	百位	十位	个位	二进制数	
十六进制数				F	F
开始				1 1 1 1	1 1 1 1
左移 1			1	1 1 1 1	1 1 1
左移 2			1 1	1 1 1 1	1 1
左移 3			1 1 1	1 1 1 1	1
加 3			1 0 1 0	1 1 1 1	1
左移 4		1	0 1 0 1	1 1 1 1	
加 3		1	1 0 0 0	1 1 1 1	
左移 5		1 1	0 0 0 1	1 1 1	
左移 6		1 1 0	0 0 1 1	1 1	
加 3		1 0 0 1	0 0 1 1	1 1	
左移 7	1	0 0 1 0	0 1 1 1	1	

续表

操作	百位	十位	个位	二进制数	
加 3	1	0 0 1 0	1 0 1 0	1	
左移 8	1 0	0 1 0 1	0 1 0 1		
BCD 码	2	5	5		

例 3.25　8 位二进制数转换为 BCD 码的描述

```
module bin_bcd8
   (
    input [7:0] bin4,
    output  reg [9:0]  bcd
   );
  //中间变量
  reg [17:0] x;
  integer i;
  always @( * )begin
      for(i = 0;i < 8;i = i + 1)x[i] = 0;
      x[10:3] = bin4;                                  //左移 3 位
      //重复 5 次
      repeat(5)begin
          //如果个位大于 4
          if(x[11:8]> 4) x[11:8] = x[11:8] + 3;    //加 3
          //如果十位大于 4
          if(x[15:12]> 4)x[15:12] = x[15:12] + 3; //加 3
          x[17:1] = x[16:0];                       //左移 1 位
      end
      bcd = x[17:8];                                   //BCD
  end
endmodule
```



图 3.10　8 位二进制数转换为 BCD 码的仿真波形

本章习题

1. 格雷码转换电路

格雷码(循环二进制单位距离码)是任意两个相邻数的代码只有一位二进制数不同的编码,它与奇偶校验码同属可靠性编码。

从对应的 n 位二进制码字中直接得到 n 位格雷码码字,需要先对 n 位二进制的码字,从右到左,以 0 到 $n-1$ 编号。如果二进制码字的第 i 位和 $i+1$ 位相同,则对应的格雷码的第 i 位为 0,否则为 1(当 $i+1=n$ 时,二进制码字的第 n 位被认为是 0,即第 $n-1$ 位不变)。

公式表示:

$$G_i = B_i \oplus B_{i+1} \quad (n-1 \geqslant i \geqslant 0)$$

其中,G 表示格雷码,B 表示二进制码。

(1) 设计一个 4 位的二进制格雷码转换电路;

(2) 推导代码并进行仿真验证;

（3）利用 FPGA 开发板进行验证操作。

2. 双优先编码器

双优先编码器返回最高优先级和次高优先级请求代码，输入是 15 位 req 信号，输出是两组 4 位二进制代码，分别对应最高优先级和次高优先级请求。

（1）列出真值表；

（2）设计电路并推导代码；

（3）推导测试平台并采用仿真验证代码的操作；

（4）设计测试电路，在原型板上用多组 LED 灯显示输出信号，并推导代码；

（5）综合电路，编程 FPGA 并验证操作。

3. BCD 增量器

BCD 码用 4 位表示十进制数字 0～9，例如 259_{10} 用 BCD 码表示为“0010 0101 1001”，BCD 增量器用 BCD 码格式加 1，例如加 1 后，“0010 0101 1001”（259_{10}）变为“0010 0110 0000”（即 260_{10}）。

（1）设计一个基于按键选择的三位数的 12 位增量器，BCD 增量计数为初始值加上按键组合值{Key[2], Key[1], Key[0]}，并推导代码；

（2）推导测试平台并采用仿真验证代码的操作；

（3）用 LED 灯显示三位数，并推导代码；

（4）综合电路，编程 FPGA 并验证操作。

4. 比较电路

设计实例中所举的比较电路是基于 if-else 语句的，请思考如何用 case 语句写出比较电路：

（1）写出一个两位较大数判断电路的真值表；

（2）用 case 语句编写该比较电路；

（3）用 Vivado 查看电路的 RTL，对比其与 if-else 语句生成的 RTL 结果；

（4）仿真并将程序下载到 FPGA 开发板上验证。

5. 超前进位加法器

4 位超前进位加法器（Carry Look-Ahead Adder，CLA）的逻辑表达式如下。

（1）中间变量：$G_i=A_iB_i$，$P_i=A_i\oplus B_i$。

（2）计算和：$S_i=P_i\oplus C_{i-1}$。

（3）进位：$C_i=G_i+P_iC_{i-1}$。

请用 Verilog HDL 门级描述上述表达式，实现此 4 位超前进位加法器，接口电路如图 3.11 所示。



图 3.11　接口电路

6. 无符号数值比较器

请用 Verilog HDL 门级描述，实现 4 位数值比较器，输入 A 与 B 均为 4 位无符号数：

（1）当 $A[3:0]>B[3:0]$时，输出 $Z=3'b001$；

（2）当 $A[3:0]=B[3:0]$时，输出 $Z=3'b010$；

（3）当 $A[3:0]<B[3:0]$时，输出 $Z=3'b100$。

7. 有符号数值比较器

请用 Verilog HDL 门级描述，实现 4 位数值比较器，输入 A 与 B 均为 4 位有符号数：

（1）当 $A[3:0]>B[3:0]$时，输出 $Z=3'b001$；

（2）当 $A[3:0]=B[3:0]$时，输出 $Z=3'b010$；

（3）当 $A[3:0]<B[3:0]$时，输出 $Z=3'b100$。

8. BCD 加法器

请用 Verilog HDL，实现 4 位 BCD 加法器，两操作数均为 BCD 码表示的 4 位数，同时接受 1 个进位，输出 BCD 码结果和进位。

接口描述如下：

```
module bcd_adder(
    input [3:0]a,
    input [3:0]b,
    input cin,
    output cout,
    output [3:0]sum
);
    //请输入你的代码

endmodule
```

9. 数据选择器

请用 Verilog HDL，设计一个 2-to-1 选择器。

（1）当 sel=0，选通 a；

（2）当 sel=1，选通 b。

接口描述如下：

```
module sel_2_1(
    input a,
    input b,
    input sel,
    output out
);
    //请输入你的代码

endmodule
```

10. 卡诺图组合逻辑电路

请用 Verilog HDL，描述如图 3.12 所示的卡诺图的组合逻辑电路。

bc \ a	0	1
00	0	1
01	1	1
11	1	1
10	1	1

图 3.12　卡诺图

11. 有符号数加法器

请用 Verilog HDL，设计一个带有溢出标志位的有符号数加法器：

（1）操作数为 8 位有符号数补码；

（2）输出加法结果 sum 和溢出标志 overflow。

接口描述如下：

```
module adder_signed(
    input [7:0]a,
    input [7:0]b,
    output [7:0]sum,
    output overflow
);
    //请输入你的代码

endmodule
```



图 3.13　逻辑门示意图

12. 逻辑门电路

请根据如图 3.13 所示的逻辑门示意图，使用 Verilog HDL 进行组合逻辑电路描述。

13. 逻辑门电路逻辑简化

请根据如图 3.14 所示的逻辑门电路示意图，写出逻辑表达式，简化后，使用 Verilog HDL 重新进行组合逻辑电路描述。

14. 波形图到组合逻辑

请根据如图 3.15 所示的波形图，将 A、B、F 之间的逻辑关系用组合逻辑描述：如 $F=A\&B$。

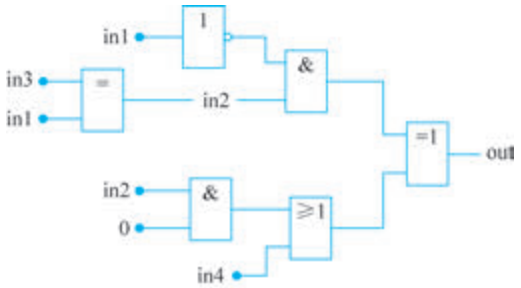


图 3.14　逻辑门电路示意图

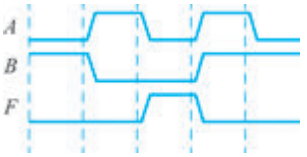


图 3.15　波形图

15. 数字设计语言到组合逻辑

请根据下面的 Verilog HDL 代码，写出组合逻辑，并画出对应的电路图。

```
wire [1:0] a;
reg b;

always @ ( * )begin
    case(a[1:0])
        2'b00: b = 1'b0;
        2'b01: b = 1'b1;
        2'b10: b = 1'b1;
        2'b11: b = 1'b0;
    endcase
end
```

16. 二进制补码

用 Verilog HDL 代码，写一段组合逻辑程序，实现如下功能：

(1) 当 $x>0$ 时，$z=y-|x|$；

(2) 当 $x<0$ 时，$z=y+|x|$；

（3）当 $x=0$ 时，$z=y$。

其中，输入的 x、y 为二进制补码，位宽为 8 位。其中输出的 z 为二进制补码，位宽根据实际需要确定。

17. 赋值语句的电路图绘制

请根据下面的 Verilog HDL 代码，绘制出对应的电路图：

```
assign out = cnt[2:0] < 3'b010;
```