搭建你的数字积木
—数字电路与逻辑设计
（Verilog HDL&Vivado版）
— 参考课件PPT

东南大学 & Xilinx大学计划部

Xilinx 大学合作部 & 东南大学

Chap.3. 组合逻辑电路设计基础

本章学习导言

- 上一章已经介绍了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。**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，而与电路的以前状态无关。**电路只是由门电路组成，没有记忆单元，也没有反馈电路。本章主要介绍常用中等规模组合电路的HDL描述，例如：加法器、比较器和多路复用器等。
- 本章的教学目的是给出组合逻辑电路基本概念，内容安排先介绍用于组合逻辑设计的条件语句和循环语句等基本Verilog HDL语言要素，之后分别给出比较器、多路选择器等常见组合逻辑电路单元的介绍及其Verilog HDL实现。

主要内容

- 组合逻辑中的always块
- 条件语句和循环语句
- always块的一般编码原则
- 参数和常数
- 常用组合逻辑电路设计实例

3.1 组合电路中的always块

- 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，但只有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；
- 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，但是很多都没有对应的硬件，编码不佳的always块通常会导致不必要的复杂实施或者根本无法综合。

3.1.1 基本语法格式

- 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]
        [可选的本地变量声明];
        [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。

3.1.1 基本语法格式

- 如有两个或者两个以上的信号时：在Verilog-1995中，它们之间可以用关键字or来连接，如：

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

- Verilog HDL 2001规范中，可以使用“，”来区分，如：

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

- 在本书中，我们都使用Verilog HDL 2001规范。

3.1.1 基本语法格式

- @[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。模块体包含任意数目的过程语句，当模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，一种是边缘敏感型。对于组合电路，一般采用电平触发；对于时序电路，一般由时钟边沿触发，Verilog HDL提供了**posedge**和**negedge**两个关键词来分别描述上升沿和下降沿。

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：

  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，一个表达式只能赋给一个数据类型的值，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。

3.1.2 过程赋值

- 在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）组合电路建议使用阻塞赋值；

  （2）时序电路建议使用非阻塞赋值。

- 因为本章我们关注的是组合电路，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，但用于过程输出；
- **integer**数据类型表示固定大小（通常是**32**位）有符号二进制补码格式，由于大小固定，我们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----1位比较器

- 1位比较器
- 代码如例3.1所示：
- 因为eq，p0和p1信号在always块内赋值，它们都声明为reg型数据类型，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。

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

例3.1

3.1.4 简单实例----1位比较器

- 三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 2001**中，我们可以用下面的符号：

```
always  @*
```

来隐式地包含所有输入信号，在本书中，我们在组合电路描述中使用这种结构。

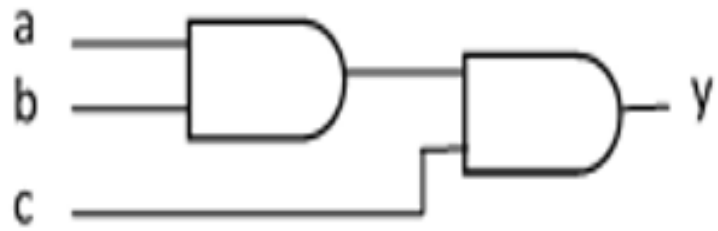
3.1.4 简单实例----过程赋值语句和持续赋值语句

- **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
```

图3.1(a) 综合的电路



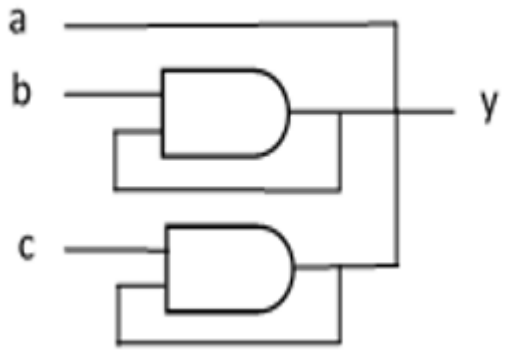
3.1.4 简单实例----过程赋值语句和持续赋值语句

- 如果我们采用类似如例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，每条程序赋值综合一部分电路，左端3次出现的y表明三个输出捆绑在一起，对应的电路如图3.1(b)所示，这显然不是我们想要的设计结果。

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（b） 不正确的电路图



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----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者关系表达式，也可以是一位的变量。语句先对表达式判断，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。

3.2.1 if-else语句----语法

- **多条if语句可以“级联”，以执行多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
```

3.2.1 if-else语句----实例

- **4位优先编码器：**
- **4位优先编码器有四个优先级：r[3]、r[2]、r[1]和r[0]作为一组4位输入信号，r[3]优先级最高，输出是最高优先级的二进制代码，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（r[3] r[2] r[1] r[0]）	输出（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

3.2.1 if-else语句----实例

- 代码首先检查信号r[3]请求，如果为1则将100赋给y，如果r[3]为0，则继续检查信号r[2]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]）。

例 3.4 使用if语句的优先编码器

```
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
```

3.2.1 if-else语句----实例

- 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，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

3.2.1 if-else语句----实例

- 4位优先编码器代码首先检查使能位en是否是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 ）。

例3.5 使用if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
```

3.2.2 case语句

- 相对于if语句只有两个分支而言，csae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、状态机及微处理器指令译码等。
- case语句有**case**、**casez**和**casex**三种表示方式，以下分别进行介绍。

3.2.2.1 case语句语法

- case语句的简化语法如右所示：
- case语句是一条多路决策语句，其将case[表达式]和多个表达式[分支项]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。
- 最后一条分支为可选的，关键词是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。

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

3.2.2.2 case语句实例

- **我们采用和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.**
- **2-4译码器使用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
```

3.2.2.2 case语句实例

- **4位优先编码器的功能表如表3.1所示，HDL代码如例3.7所示。**
- **例3.7使用case语句的优先编码器**

例3.7 使用case语句的优先编码器

```
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
                      y = 3'b000;
           endcase
endmodule
```

3.2.2.3 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**（即对应为无需匹配）。
- 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，我们更倾向于采用？。

3.2.2.3 casex 和casez 语句

- **例 3.8 使用casez语句的优先编码器**

```
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;    // 这里可以使用default
        endcase
endmodule
```

3.2.2.4 full case和parallel case

- **在Verilog HDL中，项目表达式不需要包括case表达式的所有值，一些值也可以匹配不只一次，考察下面的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
```

- 注意该语句中**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，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。

3.2.2.4 full case和parallel case

- 当分支项表达式包括case表达式的**所有**二进制值，则语句称为**full case**语句，未匹配的值我们可以用**default**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），语句则称为**parallel case**语句，例如，之前的**casez**语句不是**parallel case**语句，因为**3'b111**出现了两次，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。

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。它们分别是：**
  - **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。
  - **repeat**：连续执行一条语句n次。
  - **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。
  - **for**：有条件的循环语句。

3.3.1 for 语句

- **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
```

- **它的执行过程是：**

(1) 先求解表达式1；
(2) 再求解表达式2，如果其值为真（非0），则执行下面的第(3)步；如果为假（0），则结束循环，转到第(5)步；
(3) 执行begin/end块中的顺序执行语句，然后求解表达式3；
(4) 转回第(2)步继续执行；
(5) 执行for语句下面的语句。

- 当顺序执行语句只有一条时，定界符begin和end可以省略。

3.3.1.2 for语句实例

- **for循环实现两个8位二进制数的乘法操作**

例 3.9：2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-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。

3.3.2 repeat语句-实例

- **下面的例子用repeat循环实现两个8位数的二进制数的乘法操作，实现代码参见例3.10所示。**

例3.10

```
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---语法

- **while语句的使用格式如右所示：**

```
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，为真的话，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

3.3.3 while语句---实例

- **下例用while循环实现两个8位二进制数的乘法操作，实现代码见例3.11所示。**

例 3.11：2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，当顺序执行语句只有一条时，定界符begin和end可以省略。**

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

- 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，我们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；
- 我们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，做不到这些会经常导致无法综合的代码，不必要的复杂实施，或者在仿真和综合之间产生差异。
- 在本部分，我们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：**

> **① 变量在多个always块中赋值**
>
> **② 不完整的敏感信号列表及不完整分支**
>
> **③ 输出赋值**

- 下边分别讨论这些错误

3.4.1组合电路代码中的常见错误

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y变量：

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

3.4.1组合电路代码中的常见错误

- **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。**
  - 以上代码表示y是两块电路的输出，并可以通过每个部分进行更新，没有物理电路可以表示这种行为，因此代码不能综合，我们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;
```

3.4.1组合电路代码中的常见错误

不完整的敏感列表

- 对于组合电路，要求所有的输入信号都应该包含在敏感信号列表中
- 例如2输入与门可以写成:

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

- **如果不包含b，代码变成：**

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

- 由于always块对b“不敏感”，当b发生变化时，y仍保持之前的值不变。
- 导致仿真与综合的结果不一致。

3.4.1组合电路代码中的常见错误

不完整分支和不完整输出赋值

- 组合电路always模块的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

3.4.1组合电路代码中的常见错误

有两种方法来解决错误

- 加上else分支并明确给所有输出变量赋值，代码如下所示：

```
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
```

- 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，代码如下所示，如果gt和eq之后未赋值，则认为是0。

```
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
```

3.4.1组合电路代码中的常见错误

- **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，这会综合出意外的锁存器。

3.4.1组合电路代码中的常见错误

- 必须保证任何时候y都被赋值
- 采用关键字default以包含所有未指定值，如（1）。
- 也可以用以下代码替代最后一 条分支项表达式，如（2）。
- 在always块的开始部分赋给一个默认值，如（3）。

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

（1）

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

（2）

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

（3）

3.4.2 组合电路中always块的使用原则

- **以下原则需要遵守：**

(1) 只在一个always模块中对变量进行赋值。
(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) 一种满足以上两条原则的方法是在always块开始时给输出赋默认值。
(7) 用代码描述full case和parallel case，而不用软件指令和属性。
(8) 了解不同控制结构综合出的电路类型。
(9) 思考硬件而不是C代码。

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量代替“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，可以用关键词localparam（局部参数）声明常量，例如，我们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;
```

- 声明表达式如2**DATA_WIDTH - 1，在处理前已经执行，因此不综合物理电路。
- 在本书中，我们用大写字母表示常数。

3.5.1 常数

- 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。
- 如果我们想改成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**繁琐且容易出错**的过程。

3.5.1 常数

- 为了增强可读性，我们可以采用符号常数N来表示加法器的位数，修改后代码如例3.13所示，常数使代码更容易理解和维护。

例3.13 使用常数的加法器描述

```
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] ;
 endmudule
```

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，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-2001中，参数声明部分可以在开头，在端口声明之前，其语法如下：

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

3.5.2 参数

- 例3.13可以改成采用参数的加法器，如例3.14所示。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。

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   );
          localparam  N1 = N-1;                    //常数声明
          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] ;
endmudule
```

3.5.2 参数

例3.15 加法器例化的例子

```
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
```

- 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
- 参数提供了一种创建可扩展代码机制，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

- 本节给出了一些常用的组合电路的设计实例，包括：
  - **3.6.1 多路选择器**
  - **3.6.2 比较器**
  - **3.6.3 译码器**
  - **3.6.4 编码器和编码转换器**

3.6.1 多路选择器

- 多路选择器（Multiplexer）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。

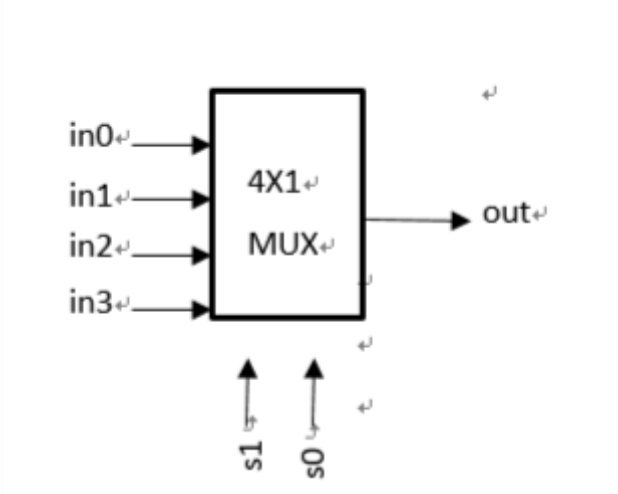


图3.2：4选1多路选择器的电路模型和真值表

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

- in0、in1、in2和in3是4个输入端口
- 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的数据。

3.6.1 多路选择器

- 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。

例3.16： 4选1多路选择器的if-else语句描述

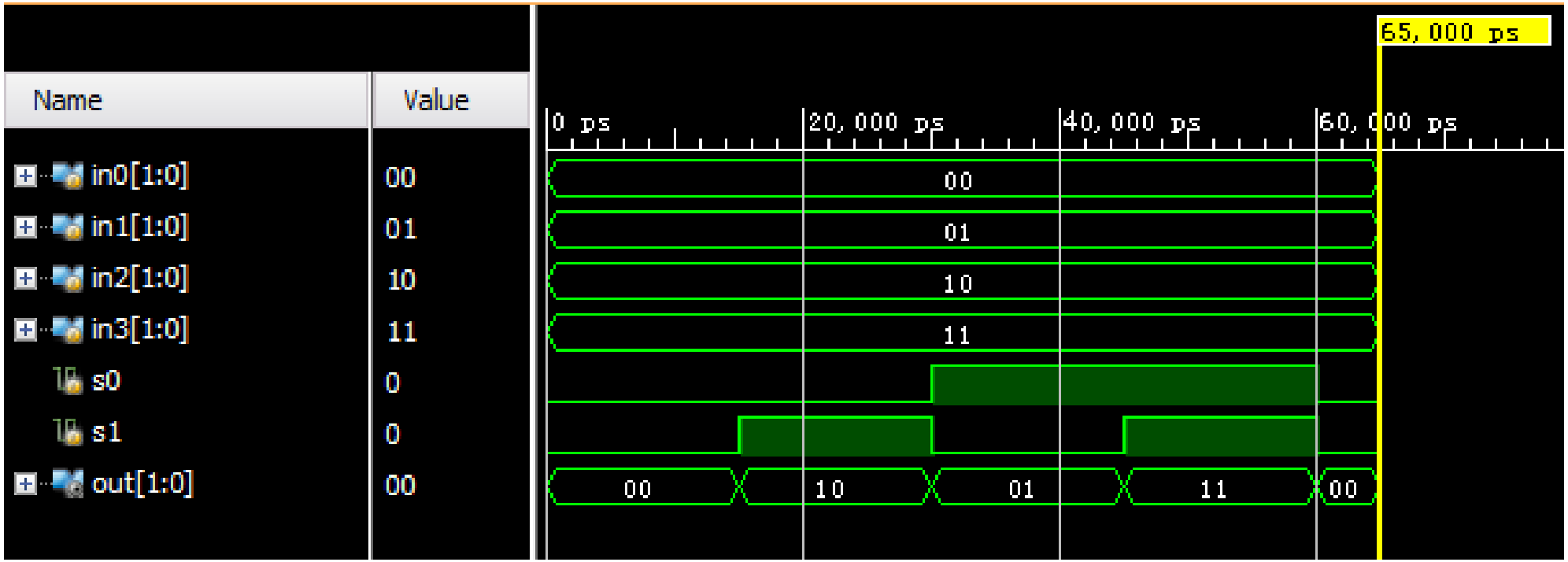
```
module mux41_if
    (    input in0,in1,in2,in3,
         input s0,s1,
         output reg out    //out声明为reg类型
     );
    always @*
         begin
            if ({s1,s0} == 2'b00)
               out = in0;
            else if({s1,s01} == 2'b01)
               out = in1;
            else if({s1,s01}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

3.6.1 多路选择器

- 4选1多路选择器的仿真波形如图3.3所示。



3.6.2 比较器

- 比较器是用来完成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。
- in0和in1是一位输入比较信号
- lt、eq和gt分别是两个输入信号大小的比较结果

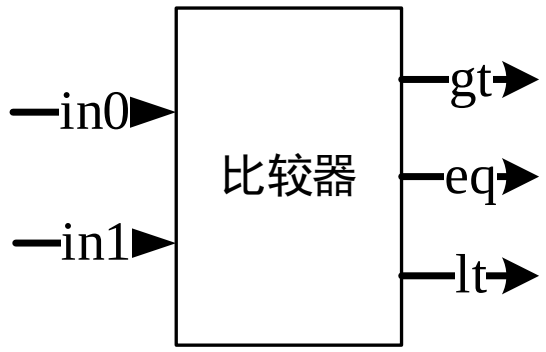


表3.3 一位二进制比较器的真值表

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

3.6.2 比较器

- 例3.18是其Verilog HDL描述。

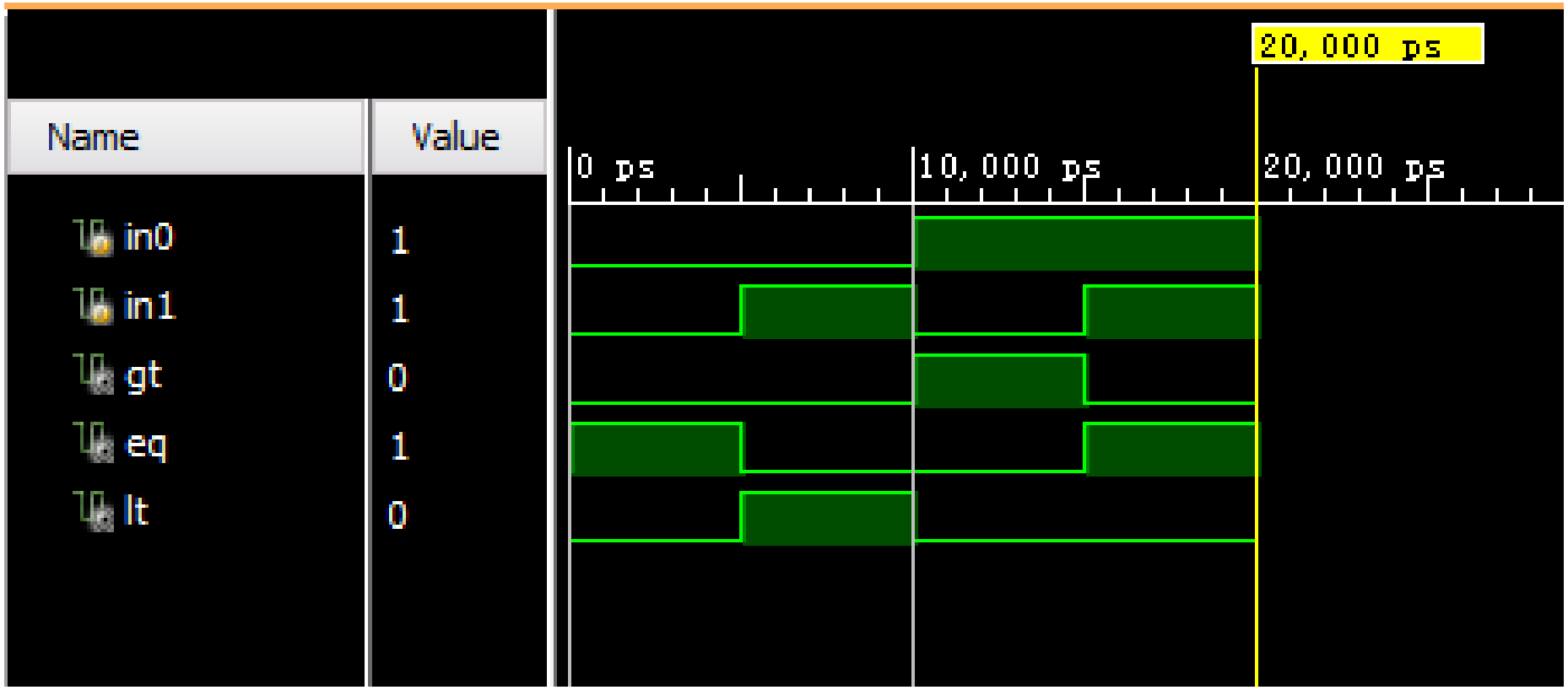
注意：本设计中，在always块内if语句之前，我们对gt，eq和lt都赋值为0。这样做的重要性是保证每个输出都被分配一个值。如果没有这样做，Verilog会认为你不想让它们的值改变，系统将会自动生成一个锁存器，那么得到的电路就不再是一个组合电路了。

例3.18 一位二进制比较器的描述

```
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
```

3.6.2 比较器

- 仿真波形如图3.4所示。



3.6.2 比较器

- 为了使用方便，我们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。这段代码在仿真时，默认N=8。

例3.19 N位二进制比较器的描述

```
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

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。

8-3编码器

- 编码器是译码器的反向器件
- 它有2n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中的哪一路输入是高电平。

3.6.3译码器和编码器

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
```

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
```

3.6.3译码器和编码器

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

- in[2:0]是编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。

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

- 每一行的输入只有一个为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。
- 为了增加代码的鲁棒性，增加一位valid输出，指示输入了不希望的逻辑组合。

3.6.3译码器和编码器

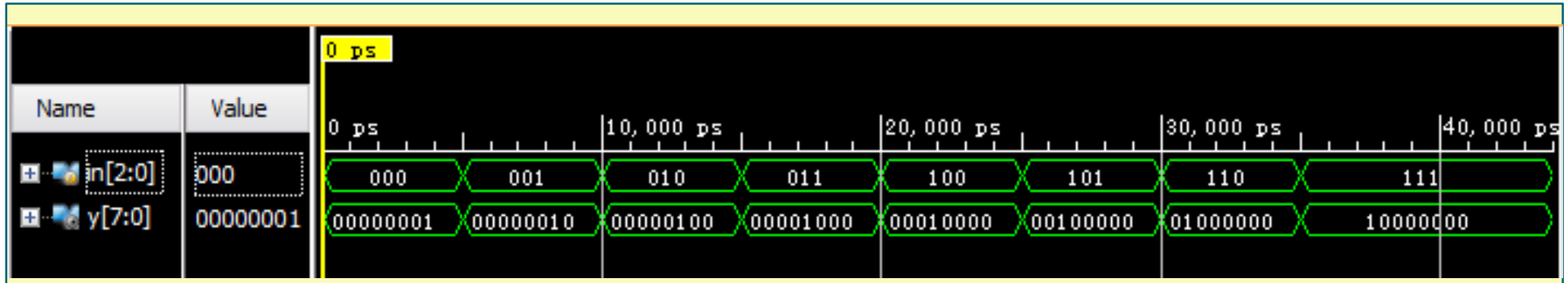


图3.5 3.8译码器的仿真波形图

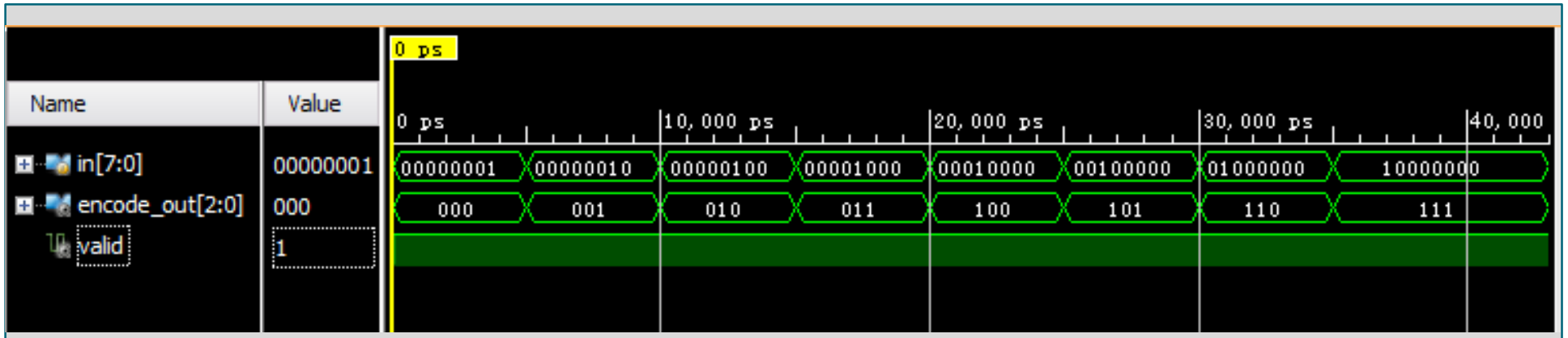


图3.6 8-3编码器的仿真波形图

3、8-3优先编码器

- 上面介绍的8-3编码器，要求在任何时候都只有一个输入为1，如果输入有一个以上的1的时候，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一个以上的1，但输出状态会按照输入的优先级来确定。

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

- 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。8-3优先编码器的仿真结果如图3.7所示，。

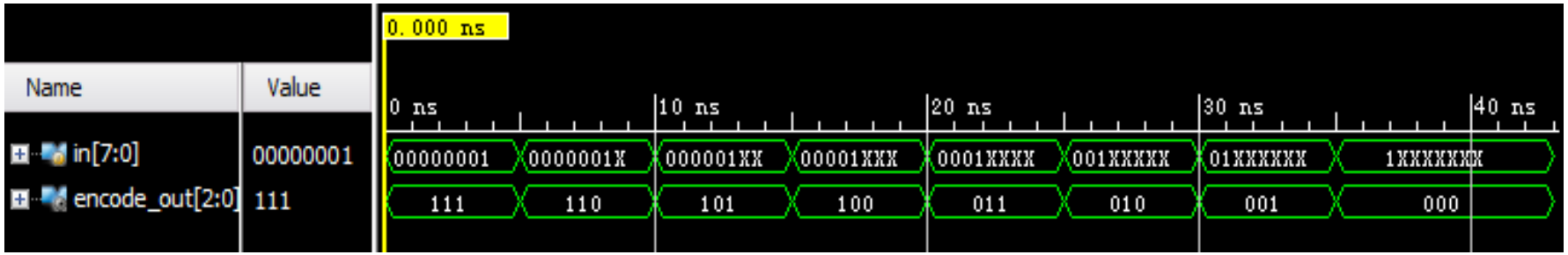
3.6.3译码器和编码器

- 8-3优先编码器实现代码如例3.22所示

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
```



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(a)所示

- 包括七个LED管和一个圆形LED小数点。
- 按LED单元连接方式可以分为共阳数码管和共阴数码管，共阳数码管是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，COM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。
- 共阴数码管是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，COM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。

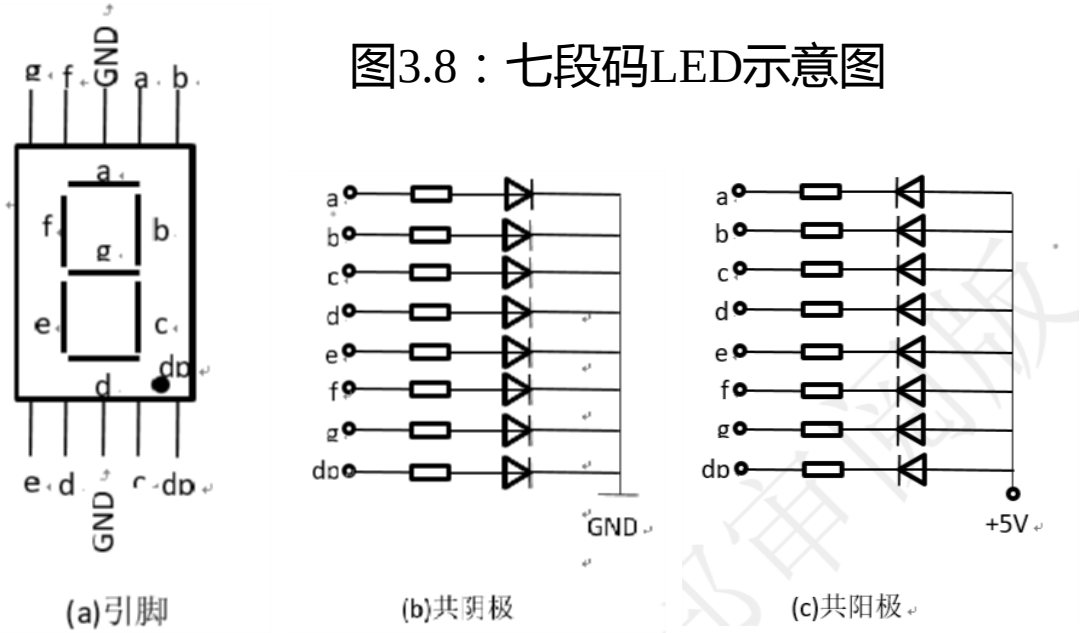


图3.8：七段码LED示意图

3.6.4 十六进制数七段LED显示译码器

- 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
- 把一个4位十六进制数输入，转换为驱动7段LED显示管的控制逻辑；为了完整，我们把1位小数点的控制dp也列出
- Verilog HDL描述如例3.23所示

例3.23十六进制数七段LED显示译码器的描述

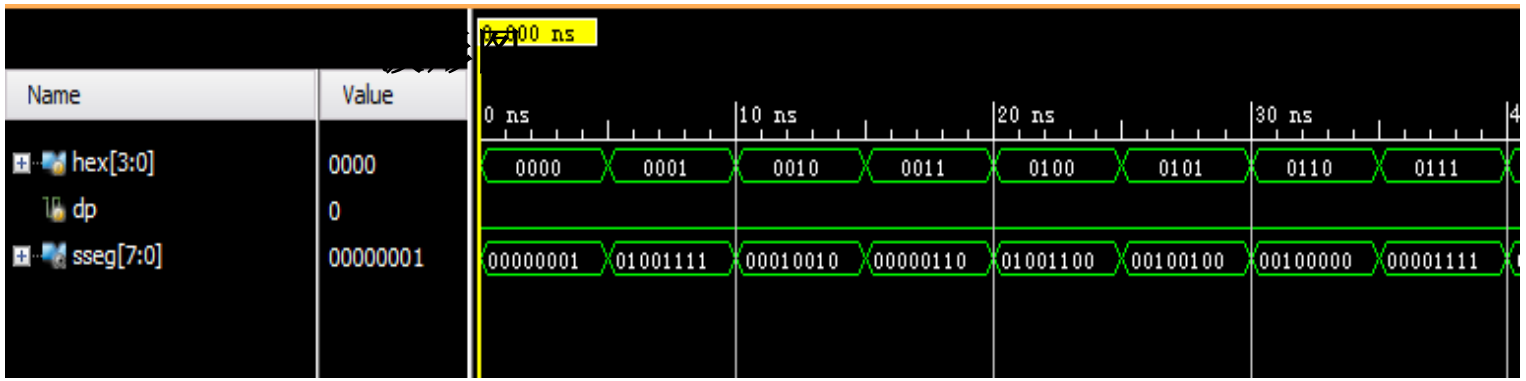
```
module hex_7seg
   (
     input [3:0] hex,
     input  dp,
     output reg [7:0]  sseg
   );
   always @*
      begin
         case (hex)
            4'h0: sseg [6:0] = 7'b0000001;
            4'h1: sseg [6:0] = 7'b1001111;
            4'h2: sseg [6:0] = 7'b0010010;
            4'h3: sseg [6:0] = 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
```

3.6.4 十六进制数七段LED显示译码器

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

图3.9十六进制数七段LED显示译码器仿真



3.6.5 二进制—BCD码转换器

- 二—十进制（Binary Coded Decimal，BCD）码，即把十进制数0~9用二进制码0000~1001表示
  - 例如：十进制数12就用两个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。

表3.8：单个十六进制数转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

3.6.5 二进制—BCD码转换器

例 3.12 单个十六进制数转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
```

3.6.5 二进制—BCD码转换器

- 设计任意数目输入的二进制—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比特二进制码为例来介绍这种算法的基本步骤：
  - 左移要转换的二进制码1位；
  - 左移之后，BCD码分别置于百位、十位、个位；
  - 如果移位后所在的BCD码列大于或等于5，则对该值加3；
  - 继续左移的过程直至全部移位完成。

图3.10十六进制码0xFF转换成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	
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数	2	5	5		

3.6.5 二进制—BCD码转换器

- 8位二进制—BCD码转换器的实现代码见例3.25

例3.25 8位二进制数转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; //左移三位
       repeat(5)              //重复5次
       begin
         if(x[11:8]>4)        //如果个位大于4
           x[11:8] = x[11:8] + 3;     //加3
         if(x[15:12]>4)       //如果十位大于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
```

3.6.5 二进制—BCD码转换器

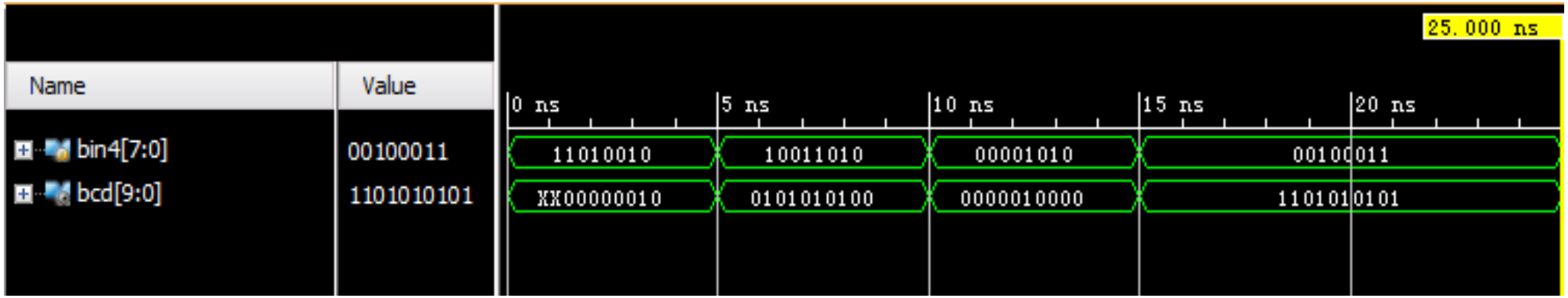


图3.11 8位二进制数转BCD码的仿真波形图

3.7 练习题

- 3.7.1 格雷码转换
- 3.7.2 双优先编码器
- 3.7.3 BCD增量器
- 3.7.4 比较电路

3.7.1 格雷码转换

口格雷码（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：**

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

（G：格雷码，B：二进制码）

- **1. 设计一个4位的二进制格雷码转换电路。**
- **2. 推导代码并且进行验证。**
- **3. 利用FPGA进行验证操作。**

3.7.2 双优先编码器

- 双优先编码器返回最高优先级和次最高优先级请求代码，输入是15位req信号，输出是第一和第二，分别是4位二进制代码的最高优先级和次最高优先级请求。

  - **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.设计测试电路并在原型板上用七段LED数码管显示两个输出信号，并推导代码。**
  - **5.综合电路，编程FPGA并验证操作。**

3.7.3 BCD增量器

- 二进制-十进制（BCD）码用4位二进制数表示1个十进制数。
  - **1.设计一个三位数的12位增量器并推导代码。**
  - **2.推导测试平台并采用仿真验证代码的操作。**
  - **3.用七段LED数码管显示三位数，并推导代码。**
  - **4.综合电路，编程FPGA并验证操作。**

3.7.4 比较电路

- 在设计实例中所举的比较电路是基于if语句的，请思考如何用case语句写出比较电路：
  - **1.推出一个2位较大数判断电路的真值表。**
  - **2.用case语句编写判断电路。**
  - **3.用Vivado查看电路的RTL，思考RTL结果。**
  - **4.仿真并且下载到板上验证**

xup@xilinx.com

行为级建模 (Behavior Modeling)

- 过程语句 (initial, always)
- 块语句 (begin-end, fork-join)
- 赋值语句 (assign, =, <=)
- 条件语句 (if-else, case, casez, casex)
- 循环语句 (for, forever, repeat, while)
- 编译向导语句(‘define, ‘include, ifdef, ’else, ‘endif)

过程语句

- **initial ——用于仿真中的初始化，initial语句中的语句只执行一次**
  - 模板：

```
initial
    begin
        语句1；
        语句2；
        …；
    end
```

- **always ——块内的语句是不断重复执行的**
  - 模板：

```
always @(<敏感信号表达式
    event-expression>)
     begin
          // 过程赋值
          // if-else, case 等选择
          // while, repeat, for 循环
          // task, function调用
       end
```

敏感项说明

- 门控锁存器

```
module D_latch(D, Clk, Q);
    input D, Clk;
    output reg Q;

    always@(D, Clk)
      if (Clk)
        Q = D;
endmodule
```

- D触发器

```
module flipflop(D, Clk, Q);
    input D, Clk;
    output reg Q;

    always@(posedge Clk)
      Q = D;

endmodule
```

赋值语句

连续赋值(Continuous)

- assign为连续赋值语句，主要对wire型变量赋值
- 例：2选1数据选择器

```
module MUX21_1(out, a,
  b, sel);
input a, b, sel;
output  out;
assign out=(sel==0)?a:b;
endmodule
```

过程赋值(Procedural)

- 左边的赋值变量必须是reg型变量
- 阻塞(blocking)

```
b = a;
```

//该语句结束后立刻赋值

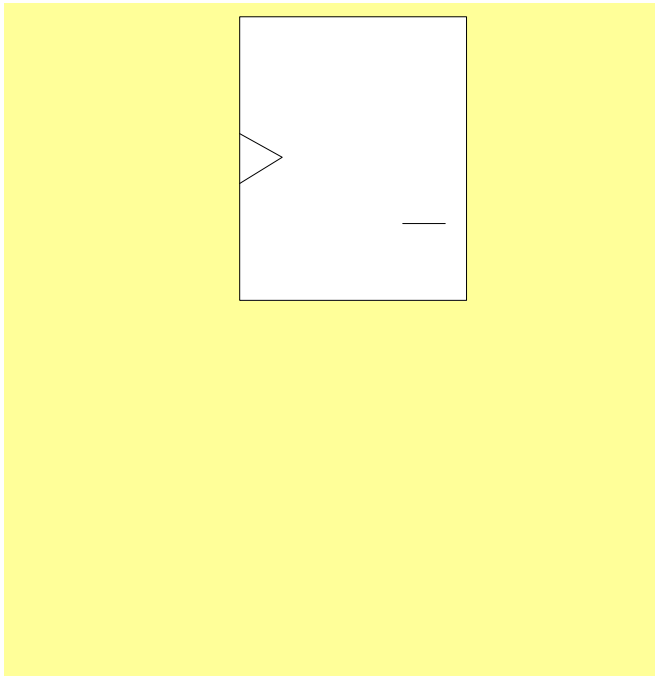
- 非阻塞(non-blocking)

```
b <= a;
```

//整个过程块结束时才执行赋值

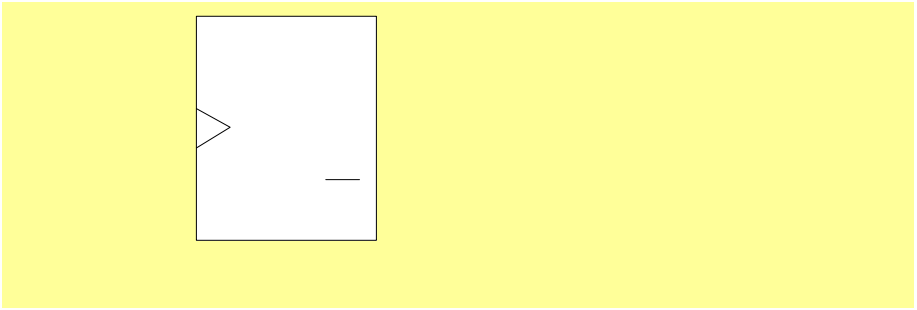
阻塞赋值

```
module example1(D, Clk, Q1, Q2)
    input D, Clk;
    output reg Q1, Q2;
    always@(posedge Clk)
    begin
        Q1 = D;
        Q2 = Q1;
    end
endmodule
```



非阻塞赋值

```
module example2(D, Clk, Q1, Q2)
    input D, Clk;
    output reg Q1, Q2;
    always@(posedge Clk)
    begin
        Q1 <= D;
        Q2 <= Q1;
    end
endmodule
```



- 非阻塞赋值可以使每条赋值语句的结果直到always块的结尾才能看到
- 阻塞赋值语句对语句顺序的依赖可能综合成错误的电路，建议用非阻塞赋值语句描述时序电路

选择语句

- **if-else**

（解释条件语句）

```
if
else if
else if
……
else
```

- **case**

（分支语句）

```
case(敏感表达式)
值1：  语句1;
值2：语句2;
……
值n：  语句n;
default: 语句n+1;
endcase
```