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.4. 时序逻辑电路设计基础

本章学习导言

- 前一章讨论了组合逻辑电路的设计，组合逻辑电路的输出只跟当前的输入有关。然而，大量常用逻辑电路的输出不仅跟当前的输入有关，而且跟过去的输入也有关。这就要求在电路中必须包含一些存储元件来记住这些输入的过去值。这种包括锁存器和触发器的电路，我们称之为时序电路。时序电路是一种能够记忆电路内部状态的电路。**与组合逻辑电路不同，时序逻辑电路的输出不仅取决于当前输入，还与其当前内部状态有关。**
- 本章的教学目的是介绍一些常用的时序电路元件模块的Verilog HDL语言描述，并对其设计进行分析，给出时序电路设计的一般方法。

主要内容

- 触发器和锁存器
- 寄存器
- 移位寄存器
- 计数器
- 常用时序逻辑电路设计实例
- 练习题

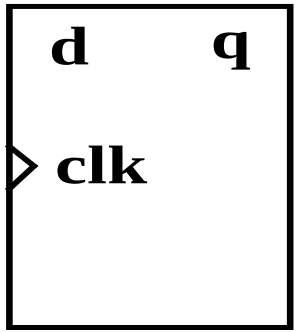
4.1 触发器和锁存器

- 不同结构功能和不同用途的触发器和锁存器，是基本的时序电路元件，是时序逻辑电路设计的基础；

- 掌握这些基础时序逻辑单元的Verilog HDL描述方法，有助于深入了解和掌握时序数字系统的设计方法。

4.1.1 基本D触发器

- D触发器（DFF）是逻辑电路中最基本的存储元件。
- 上升沿触发的D触发器是最简单的D触发器。



clk	q*
0	q
1	q
↑	d

图4-4.1 基本D触发器的符号及真值表

4.1.1 基本D触发器

- **从时序波形可以看出，输出信号d的值只在clk的上升沿到达时变化，并存储在触发器中。**

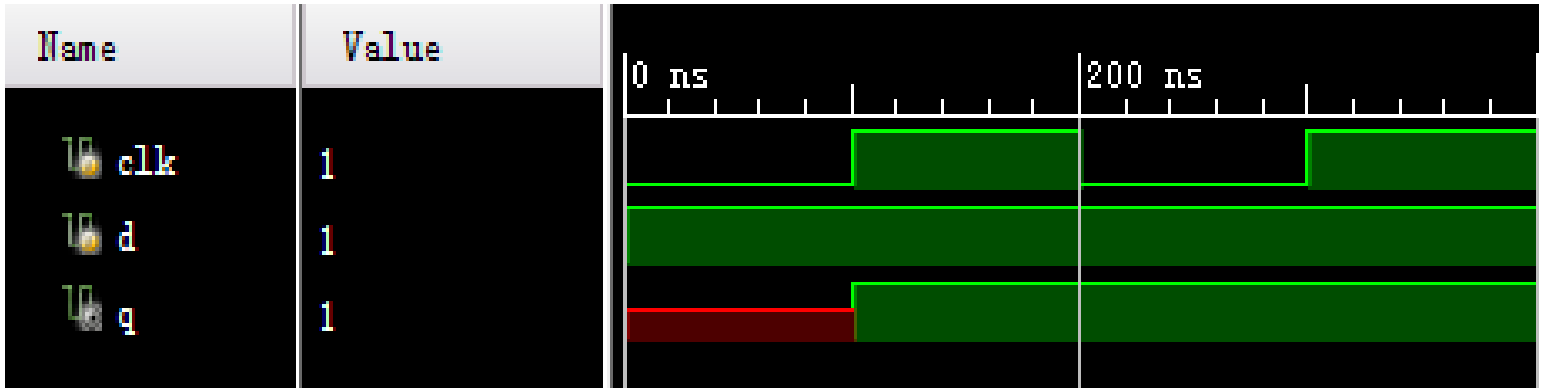


图4-4.2 基本D触发器的时序波形

4.1.1 基本D触发器

例4.1使用了过程语句，时序电路通常都是用过程语句来进行描述。

例 **4.1** 无异步复位D触发器

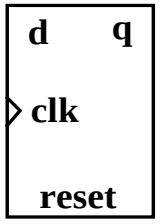
```
module dff
 (
   input  clk,
    input  d,
   output  reg  q
 );
 always @(posedge clk)
   q <= d;
endmodule
```

- 敏感列表中的posedge clk是时钟的上升沿检测函数，posedge(positive edge)指定时钟信号的变化方向为由0变为1。
- 这表明状态变化总是在clk信号的上升沿触发，反应出触发器边沿触发的特性。
- **注意**：输入信号d不包含在敏感列表中。这就验证了输入信号d只在时钟信号的上升沿进行采样,其值的改变并不会立即改变输出信号。

注：与posedge clk对应的还有negsedge clk，它是时钟下降沿敏感的描述。

4.1.2 含异步复位的D触发器

D触发器可以包含异步复位信号，从时序波形可以看出，reset脚的高电平能够在任意时刻复位D触发器，而不受时钟信号控制，它实际上比定期采样输入优先级更高。



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↑	d

图4-4.3含异步复位的D触发器的符号及真值表

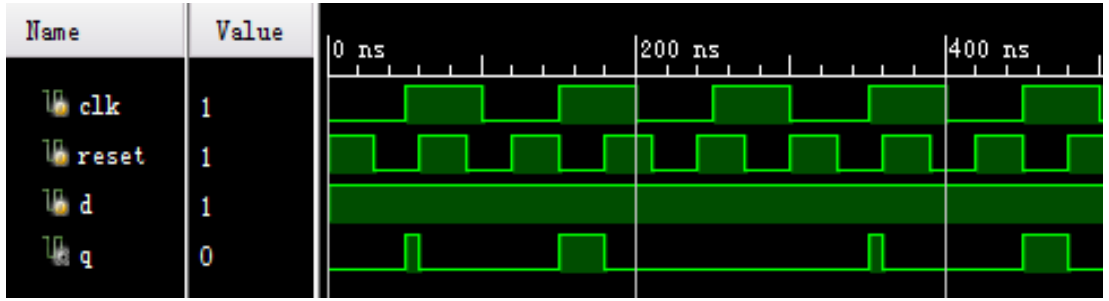


图4-4.4 含异步复位的D触发器的时序波形

注：使用异步复位信号违反了同步设计方法，因此应该在正常操作中避免。其主要应用于执行系统初始化。例如,在打开系统电源之后，我们可以生成一个短的复位脉冲迫使系统进入初始状态。

4.1.2 含异步复位的D触发器

例 **4.2** 有异步复位D触发器

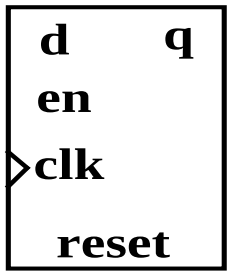
```
module dff_reset
  (
    input  clk,reset,
     input  d,
    output  reg  q
  );
  always @(posedge clk,posedge reset)
  begin
       if (reset)
           q <= 1'b0;
       else
           q <= d;
  end
endmodule
```

注意：

- reset信号的上升沿也包括在敏感列表中，同时在if语句中要首先检查其值。
- 如果reset信号为1，则将q信号置为0。
- 这里所谓的“异步”是指独立于时钟控制器。
- 在任何时刻，只要reset是高电平，触发器输出端q的输出即为低电平。

4.1.3 含异步复位和同步使能的D触发器

更加实用的D触发器包含一个额外的控制信号en，能够控制触发器进行输入值采样。注意，使能信号en只有在时钟上升沿来临时才会生效，所以它是同步信号。如果en没有置1，触发器将保持先前的值。



reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↑	0	q
0	↑	1	d

图5 含异步复位和同步使能的D触发器的符号及真值表

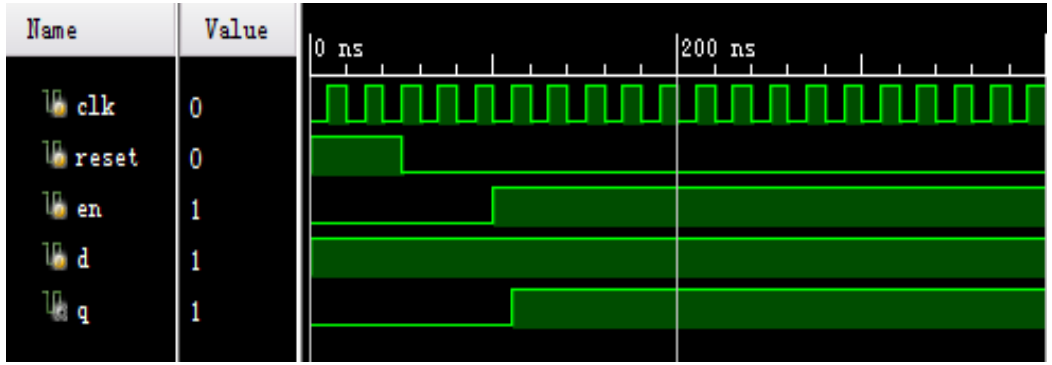


图6 含异步复位和同步使能的D触发器的时序波形

4.1.3 含异步复位和同步使能的D触发器

例 4.3 一段式含异步复位和同步使能的D触发器

```
module dff_reset_en_1seg
    (
     input  clk,reset,
     input  en,
     input  d,
     output reg  q
    );
  always @(posedge clk,posedge reset)
     begin
        if (reset)
           q <= 1'b0;
        else if(en)
           q <= d;
     end
endmodule
```

- 注意，第二个if语句后没有else分支。
- 根据Verilog HDL语法,变量如果没有被赋新值保持其先前的值。
- 如果en等于0，q将保持原值。
- 因此，省略的else分支描述了这个触发器的预期行为。

4.1.3 含异步复位和同步使能的D触发器

D触发器的使能特性在同步快子系统和慢子系统时是非常有用的。

例如,假设快子系统和慢子系统的时钟频率分别为50 MHz和1 MHz。我们可以生成一个周期性的使能信号，每50个时钟周期使能一个时钟周期，而不是另外派生出一个1 MHz的时钟信号来驱动慢子系统。慢子系统是其余49个时钟周期中是保持原来状态的。

- 这种方法同样可应用于消除门控时钟信号。

由于使能信号是同步的,该电路可以由一个常规的D触发器和简单下一状态逻辑电路构成。

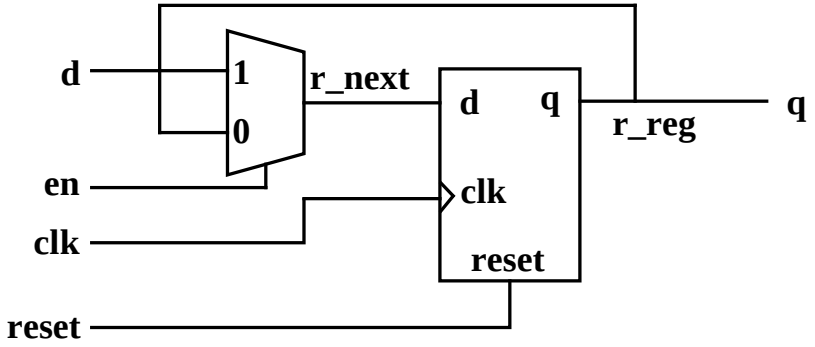


图4.7 同步使的D触发器逻辑图

4.1.3 含异步复位和同步使能的D触发器

```
module dff_reset_en_2seg
   (
     input  clk，reset,
     input  en,
     input  d,
     output  reg  q
   );
   reg r_reg，r_next;
   always @(posedge clk， posedge reset)
    begin
        if (reset)
            r_reg <= 1’b0;
        else
            r_reg <= r_next;
    end
```

为了清晰起见,代码使用了后缀_next和_reg强调下一个输入值和触发器的输出。它们分别与D触发器的d和q信号连接。

```
          //next-state logic
       always @*
          begin
              if (en)
                  r_next = d;
              else
                  r_next = r_reg;
          end
              //output logic
        always @*
            q = r_reg;
endmodule
```

例 4.4 两段段式含异步复位和同步使能的D触发器

4.1.4 基本锁存器

基本锁存器电路模块如图4.8所示，这是一个电平触发型锁存器。

它的工作时序如图4.9所示，从波形显示可以看出，当时钟clk为高电平的时，其输出q的值才会随输入d的数据变化而更新；当clk为低电平时，锁存器将保持原来高电平时是锁存的值。

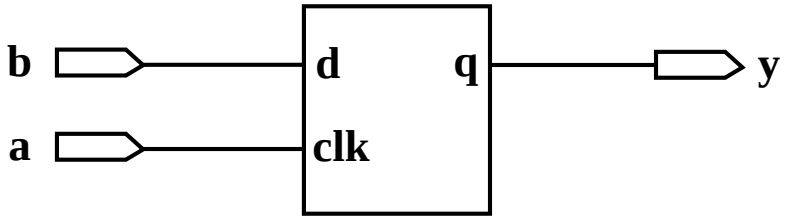


图4.8 基本锁存器的电路模块图

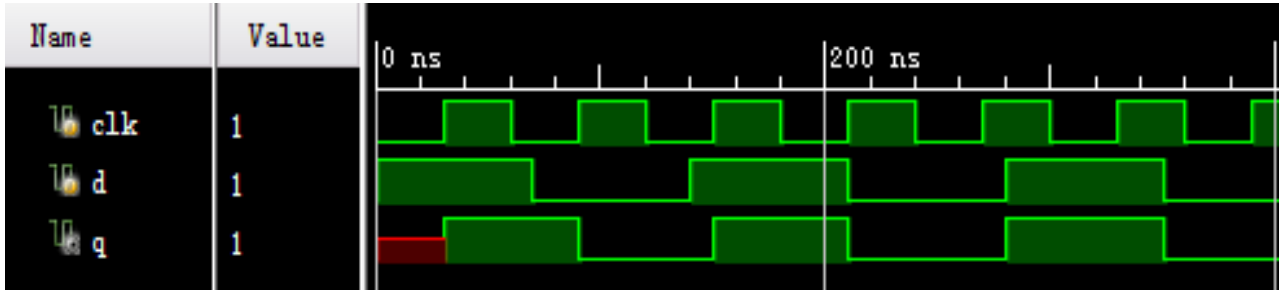


图4.9 基本锁存器的时序波形图

4.1.4 基本锁存器

例 4.5 基本锁存器的一种描述

```
module latch_1
 (
  input  clk,
   input  d,
  output  reg  q
 );
 always @(clk, d)
     if(clk)
         q <= d;
endmodule
```

- 当敏感信号clk电平从低变为高时，过程语句被启动，顺序执行if语句，此时clk为1，于是执行q<=d，把d的数据更新至q，然后结束if语句。
- 当clk从1变为0或者保持低电平都不会触发过程，锁存器的输出q将保持原来的状态，这就意味着在设计模块中引入了存储元件。而此处也是省略的else分支描述了这个触发器的预期行为。

注：与对D触发器的描述不同，基本锁存器的Verilog HDL描述中，没有使用时钟边沿敏感的关键词posedge。

4.1.5 含清0控制的锁存器

含异步清0控制的锁存器电路模块如图4.10所示，它的工作时序如图4-4.11所示。例4.6和例4.7分别是它的两种不同风格的Verilog HDL描述。

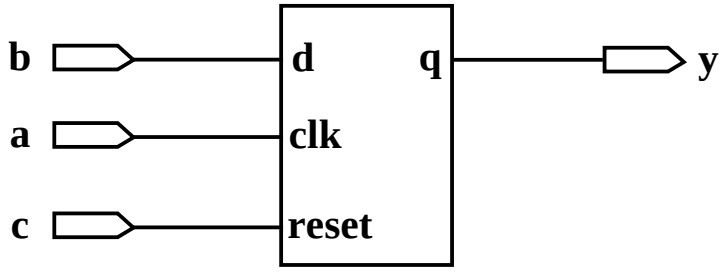


图4.10 含异步清0控制的锁存器模块图

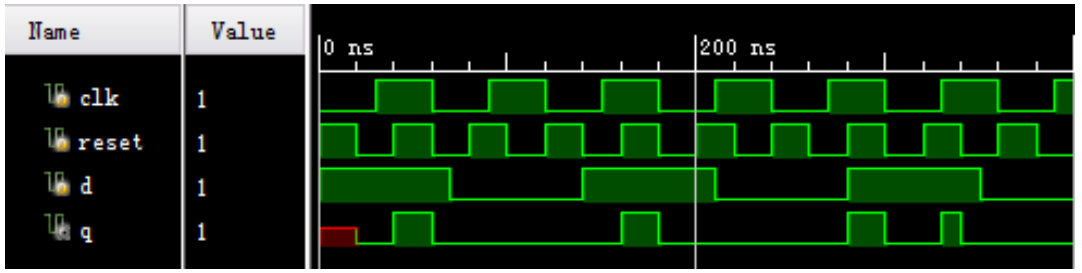


图4.11含异步清0控制的锁存器的时序波形图

4.1.5 含清0控制的锁存器

例 4.6 含异步清0控制锁存器的一种描述

```
module latch_reset_1
 (
  input  clk,reset,
   input  d,
  output q
 );
     assign q = (!reset)? 0:(clk?d:q);
endmodule
```

- 例4.6中的描述使用了具有并行语句特色的连续赋值语句，其中使用了条件操作。
- 例4.7使用的是过程语句，把数据信号d，复位信号reset和时钟信号clk都列在敏感信号列表中，从而实现例reset的异步特性和clk的电平触发特性。

例 4.7 含异步清0控制锁存器的另外一种描述

```
module latch_reset_2
 (
  input  clk,reset,
   input  d,
  output reg  q
 );
 always @(clk, d, reset)
   if(!reset)
        q <= 0;
   else if(clk)
        q <= d;
endmodule
```

4.2.1 1位寄存器

在4.1节的讨论中，我们知道D触发器可以用于比特信号的存储。如果d为1，那么在时钟的上升沿，D触发器的输出q将变为1。如果Dd为0，那么在时钟的上升沿，D触发器的输出q将为0。

在实际的数字系统中，一般D触发器的时钟输入端始终都有时钟信号输入。这就意味着在每个时钟的上升沿，当前的d值都将被锁存在q中，而时钟的变化频率通常是每秒几百万次。

4.2.1 1位寄存器

为了设计一个1位寄存器，它可以在需要时从输入线in_data加载一个值，我们给D触发器增加一根输入线load，当我们想要从in_data加载一个值时，就把load设置为1，那么在下一个时钟上升沿，in_data的值将被存储在q中。

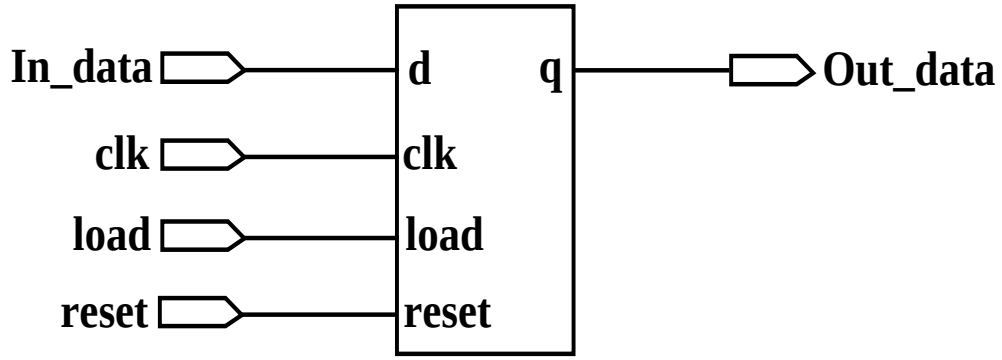


图4.12 1位寄存器逻辑符号

```
module reg_1
  (
    input  clk,reset,
    input  in_data,
    input  load,
    output reg  out_data
  );
  always @(posedge clk， posedge reset)
     if(reset)
          out_data <= 0;
     else if(load == 1)
          out_data <= in_data;
endmodule
```

例 4.8 1位寄存器的Verilog描述

4.2.1 1位寄存器

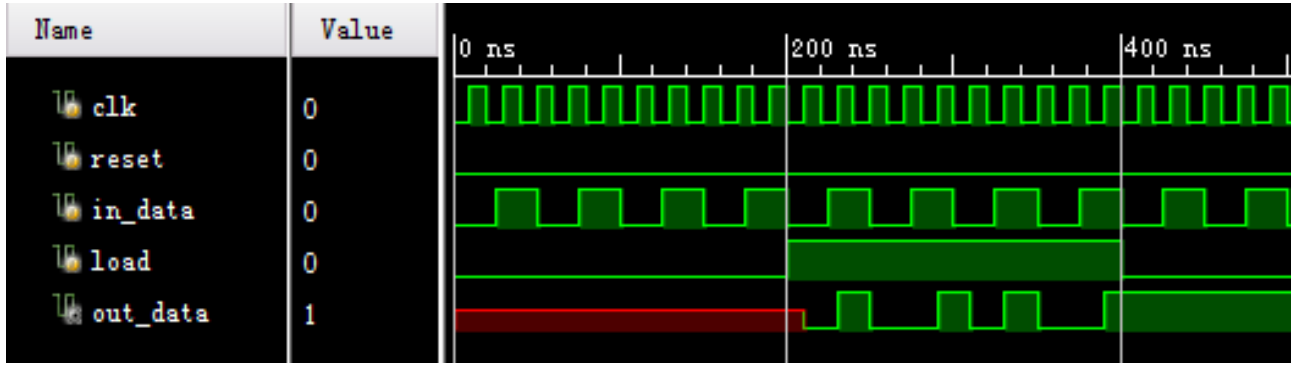


图4.13 1位寄存器的仿真时序图

从仿真结果我们可以看出:

- 当load信号为0时，输入线上的数据in_data就不会在每个时钟clk的上升沿被不断地重新加载到out_data，寄存器的输入out_data保持不变；

- 当load信号为1时，在下一个时钟clk的上升沿，out_data就变为in_data的值。

4.2.2 N位寄存器

- 如果我们把N个1位寄存器模块组合起来，就可以构成一个N位寄存器。
- 和1位寄存器不同之处在于，我们把输入in_data和out_data定义为一个N位的数组。

N位寄存器的逻辑符号如图4.14所示:

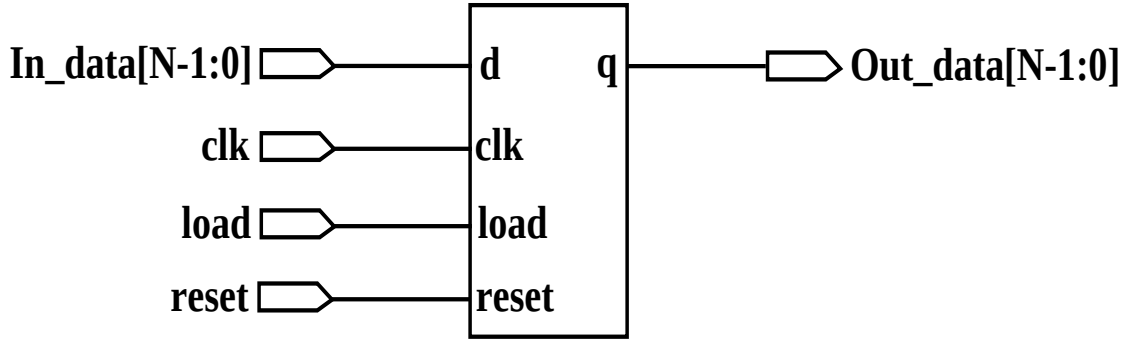


图4.14 1位寄存器逻辑符号

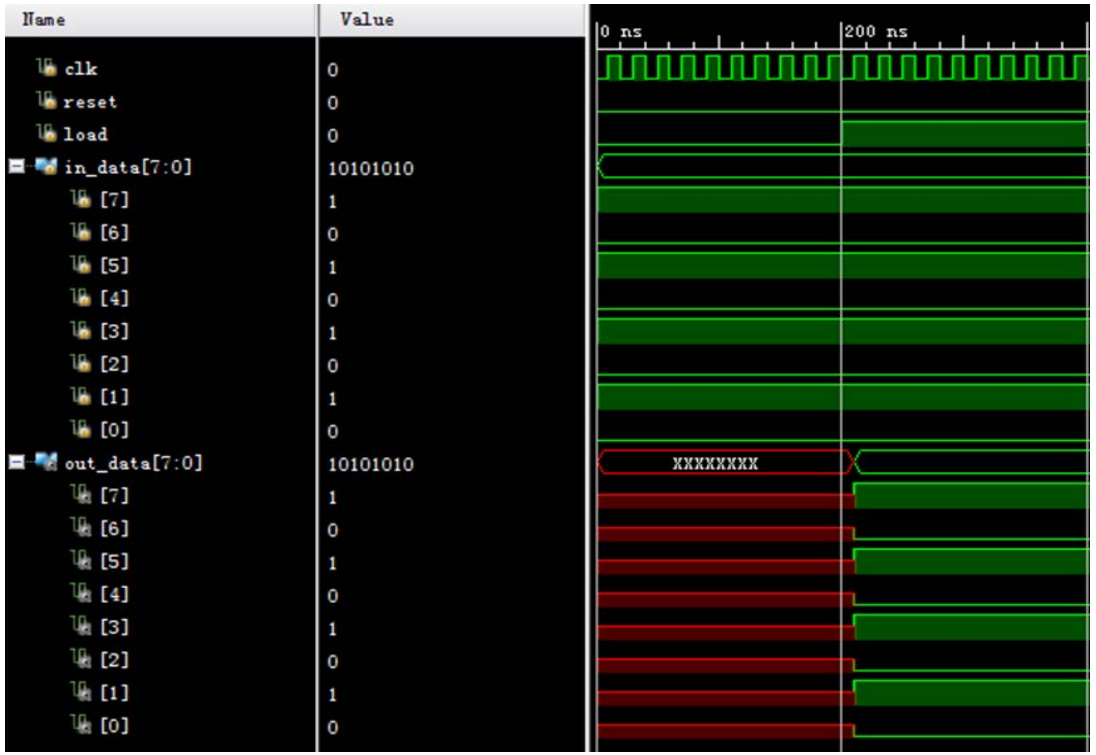


图4.15 8位寄存器的仿真时序图

4.2.2 N位寄存器

```
module reg_N
  #(parameter N = 8)
     (
         input  clk,
         input  reset,
         input [N-1:0] in_data,
         input   load,
         output  reg [N-1:0] out_data
    );
always @(posedge clk, posedge reset)
        if(reset)
              out_data <= 0;
        else if(load == 1)
             out_data <= in_data;
     endmodule
```

例 4.9 N位寄存器的描述

- 在例4.9中使用parameter语句，是为了使总线宽度可调。
- 默认状态下，总线宽度为8。

4.2.2 N位寄存器

```
module reg_N
  #(parameter N = 8)
     (
       input  clk,
       input  reset,
       input [N-1:0] in_data,
       input   load,
       output  reg [N-1:0] out_data
   );
  always @(posedge clk, posedge reset)
       if(reset)
             out_data <= 0;
       else if(load == 1)
             out_data <= in_data;
  endmodule
```

例 4.9 N位寄存器的描述

- 如果我们想修改寄存器的位宽，可以使用Verilog的实例化语句。我们可以实现一个如下所示的16位寄存器，称为fReg。

```
reg_N #(
        .N(16))
       fReg(.clk(clk),
       .reset(reset),
       .load(load),
       .in_data(indata),
       .out_data(out_data)
);
```

4.2.3 寄存器组

- 寄存器组是由一组拥有同一个输入端口和一个或多个输出端口的寄存器组成。写地址信号w_addr指定了数据存储位置，读取地址信号r_addr指定数据检索位置。
- 寄存器组通常用于快速、临时存储。

4.2.3 寄存器组

例4.10给出了一个参数化的寄存器组的实现代码。参数W指定了地址线的位数，表明在这个寄存器组中有2^W个字。参数B指定了一个字的位数。

```
module reg_file
  #(
parameter N = 8，//比特数
W = 2) //地址比特数
      (
         input  clk,
         input  wr_en,
         input [W-1:0] w_addr,r_addr,
         input [BN-1:0]  w_data,
         output  [BN-1:0]  r_data
    );
      reg [BN-1:0] array_reg[2**W-1:0];
```

```
always @(posedge clk)
        if(wr_en)
          array_reg[w_addr] <= w_data;
          assign r_data = array_reg[r_addr];
    endmodule
```

一个信号被用作索引来访问数组中元素

二维数组的数据类型：表示array_reg变量是一个含有[2**W-1:0]个元素的数组，每个元素的数据类型是reg [B-1:0]

4.2.3 寄存器组

虽然描述非常抽象，Xilinx公司的软件能够识别出这种语言构造，并正确地执行。

array_reg[…] = …和… = array_reg[…]语句分别表明解码和多路复用的逻辑。

一些应用程序可能需要同时检索多路数据，可以通过添加一个额外的读取端口来解决

。

例如:

```
r_data2 = array_reg[r_addr_2];
```

4.3 移位寄存器

一个N位的移位寄存器包含N个触发器。在每个时钟脉冲作用下，数据从一个触发器移到另一个触发器。

本节我们将介绍移位寄存器的几种不同的描述和设计方法。

4.3.1 具有同步预置功能的8位移位寄存器

具有同步预置功能的8位移位寄存器的描述如例4.11所示：

- clk是移位时钟信号，load是并行数据预置使能信号，din是8位并行预置数据端口，qb是串行输出端口。

工作原理：

- 当clk的上升沿到来时，过程被启动，如果此时预置使能端口load为高电平，则输入端口din的8位二进制数被同步并行移入移位寄存器，用作串行右移的初始值；
- 如果此时预置使能端口load为低电平，则执行赋值语句：

```
reg8[6:0] <= reg8[7:1];
```

这样完成一个时钟周期后，把上一时钟周期的高7位值reg8[7:1]更新至此寄存器的低7位reg8[6:0],实现右移一位的操作。连续赋值语句把移位寄存器最低位通过qb端口输出。

```
module shift_reg8
  (
      input  clk,
      input  load,
      input [7:0]  din,
     output  qb
   );
  reg [7:0] reg8;
always @(posedge clk)
      if(load)
            reg8 <= din;
      else
           reg8[6:0] <= reg8[7:1];
     assign qb = reg8[0];
endmodule
```

例 4.11 具有同步预置功能的8位移位寄存器描述

4.3.1 具有同步预置功能的8位移位寄存器

```
module shift_reg8
  (
     input  clk,
     input  load,
     input [7:0]  din,
    output  qb
   );
   reg [7:0] reg8;
   always @(posedge clk)
     if(load)
       reg8 <= din;
     else
       reg8[6:0] <= reg8[7:1];
    assign qb = reg8[0];
endmodule
```

例 4.11 具有同步预置功能的8位移位寄存器描述

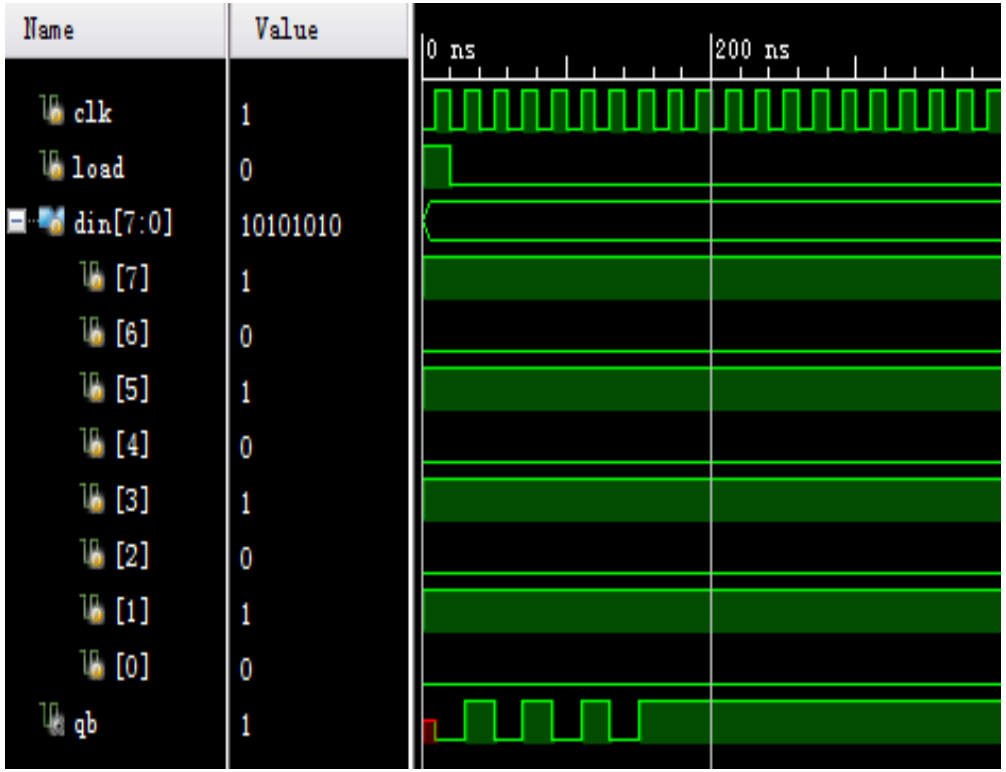


图4.16 8位寄存器的仿真时序图

4.3.2 8位通用移位寄存器

```
module univ_shift_reg
        #(parameter N = 8)
        (
        input clk, reset ;
        input  [1:0] ctrl ;
        input  [N-1:0] d ;
        output [N-1:0] q
        );
        //信号声明
        reg [N-1:0] r_reg, r_next;
        //寄存器
    always @(posegde clk, posedge reset)
        if (reset)
            r_reg <= 0;
        else
            r_reg <= r_next;                        //next-state logic
    always @*
            case (ctrl)
            2'b00: r_next = r_reg;                  //无操作
            2'b01: r_next = {r_reg[N-2:0], d[0]};   //左移
            2'b10: r_next = {d[N-1], r_reg[N-1:1]}; //右移
            default: r_next = d;                    //载入
            endcase
    assign q = r_reg;                               //输出逻辑
endmodule
```

例 4.12 8位通用移位寄存器描述

- 通用移位寄存器可以加载并行数据，将其内容向左移位、向右移位或保持原有状态，也可以实现并转串(第一次加载并行输入，然后移位)或串转并(首先移位，然后进行并行输出)。
- 可以把通用移位寄存器分为组合逻辑和时序逻辑两部分，各用一个always块来进行描述。下一状态逻辑使用4选1的多路选择器来选择寄存器所需下一状态的值。注意，d的最低位和最高位被用来作为串行输入的左移和右移操作。

4.4 计数器

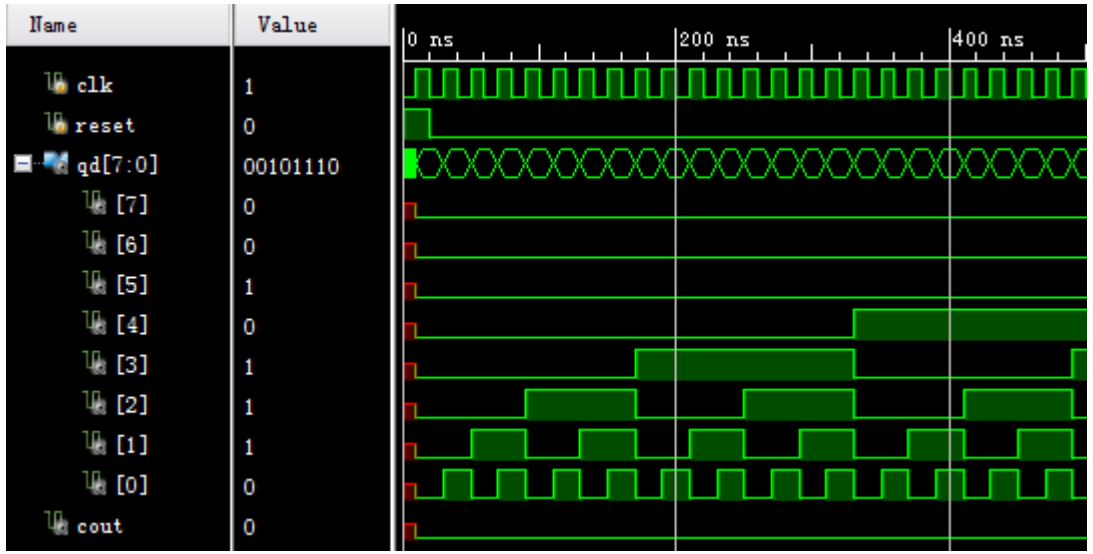
4.4.1 简单的二进制计数器

一个简单的二进制计数器通过二进制序列反复循环。例如一个4位二进制计数器计数，从“0000”，“0001”，…，至“1111”而后循环。N位简单二进制计数器的参数化实现代码如例4.13所示。

```
module counter_sim_bin_N
  #(parameter N = 8)
  (
      input  clk,
      input  reset,
      output [N-1:0] qd,
      output  cout
   );
   reg [N-1:0] regN;
   always @(posedge clk)
      if(reset)
          regN <= 0;
      else
          regN <= regN + 1;
      assign qd = regN;
      assign cout = (regN == 2**N-1)?1'b1:1'b0;
endmodule
```

例 4.13 简单的N位二进制计数器描述

4.4.1 简单的二进制计数器



例 4.17简单8位二进制计数器的仿真时序图

工作过程：

- 1.当复位端reset为高电平时，在时钟clk的上升沿，完成对计数器的复位；
- 2.在复位端为低电平后，计数器从下一个时钟上升沿从0000开始计数，直至计满1111后，再溢出为0000，此时计数器的进位端cout输入一个clk周期的高电平；
- 3.同时计数器从并行输出端口qd同步输出当前计数器的值

4.4.2 通用二进制计数器

```
module counter_univ_bin_N
   #(parameter N = 8)
   (
        input  clk,reset,load,up_down,
        input [N-1:0] d,
        output [N-1:0] qd
    );
    reg [N-1:0] regN;
    always @(posedge clk)
        if (reset)
            regN <= 0;
        else if (load)
            regN <= d;
        else if (up_down)
            regN <= regN + 1;
        else regN <= regN – 1;
    assign qd = regN;
endmodule
```

例 4.14通用N位二进制计数器描述

通用二进制计数器具有更多功能：

可以实现增/减计数、暂停、预置初值，并有同步清0等功能。参数化的通用二进制计数器实现代码如例4.14所示。

计数器采用同步工作方式，同步时钟输入端口是clk。

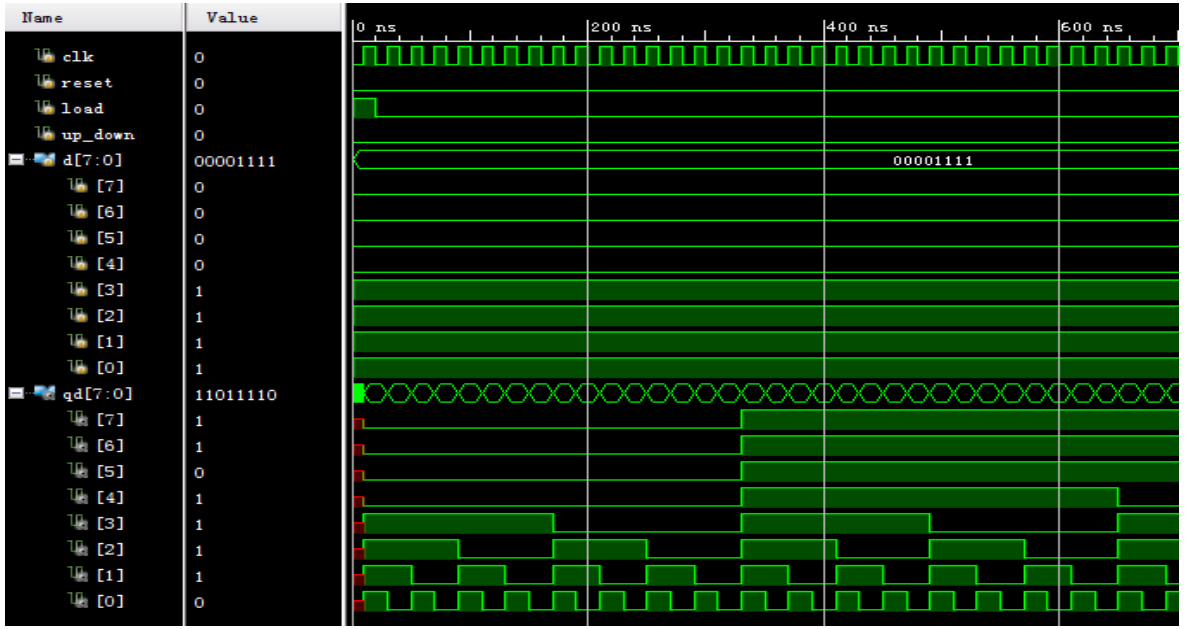


图4.18：简单8位二进制计数器的仿真时序图

4.4.3 模m计数器

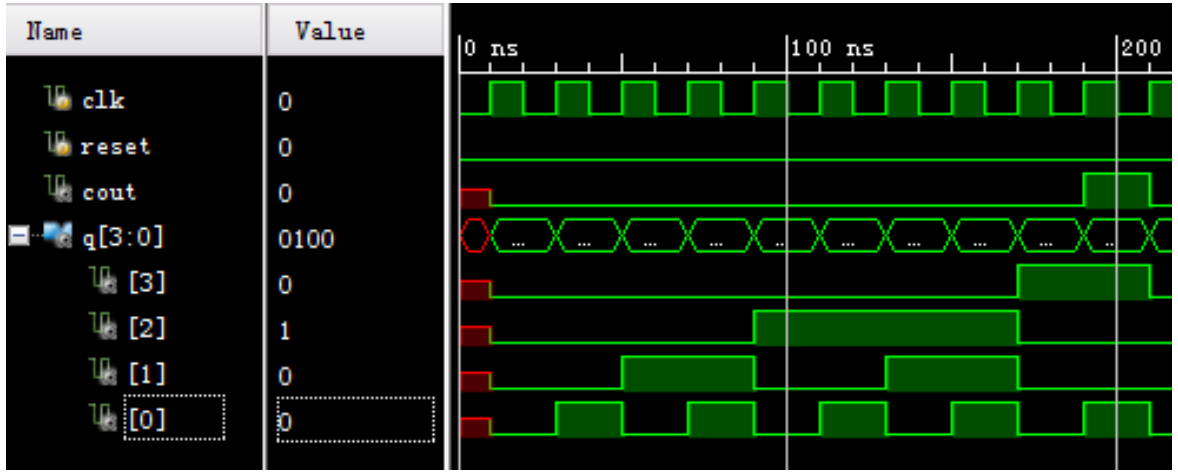
```
module counter_mod_m
  #(parameter N = 4,          //计数器位数
    parameter M = 10)         //模M缺省为10
    (
      input  clk,reset,
      output [N-1:0] qd,
      output  cout
    );
    reg [N-1:0] regN;
    always @(posedge clk)
        if(reset)
            regN <= 0;
        else if (regN < (M-1))
            regN <= regN + 1;
        else
            regN <= 0;
        assign qd = regN;
        assign cout = (regN == (M-1))?1'b1:1'b0;
endmodule
```

例 4.15 模m计数器描述

- 模m计数器的计数值从0增加到m-1，然后循环。参数化的模m计数器实现代码如例4.15所示。
- 它有两个参数：

  参数M，它指定了计数模值m；

  参数N，它指定了计数器所需的位数，等于$\lceil \log_2 M \rceil$。
- 代码例4.15中计数器的默认值是模10，它的仿真工作波形如图4.19所示。



例 4.19 简单8位二进制计数器的仿真时序图

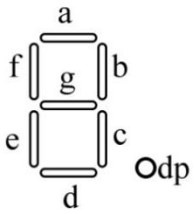
4.5 设计实例

4.5.1 数码管扫描显示电路

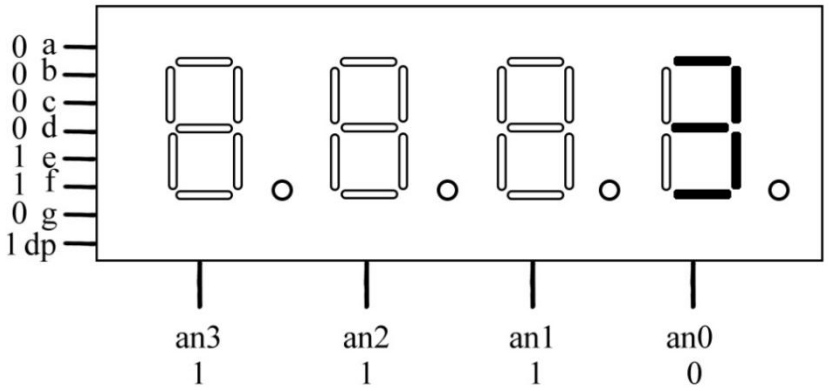
- 在前面章节的组合电路设计中，我们介绍了单个数码管显示电路的设计。每个数码管包括7个笔画和1个小圆点，需要8个IO口来进行控制。采用这种控制方式，当使用多个数码管进行显示时，每个数码管都需要8个IO口。
- 在实际应用中，为了减少FPGA芯片IO口的使用数量，一般会采用分时复用的扫描显示方案进行数码管驱动。
- 以四个数码管显示为例，采用扫描显示方案时，四个数码管的8个段码并接在一起，再用4个IO口分别控制每个数码管的公共端，动态点亮数码管。这样只用12个IO口就可以实现4个数码管的显示控制，比静态显示方式的32个IO口数量大大减少。

4.5.1 数码管扫描显示电路

如图4.20所示，在最右端的数码管上显示“3”时，并接的段码信号为“00001101”，4个公共端的控制信号为“1110”。这种控制方式在同一时间只会点亮一个数码管，采用分时复用的模式轮流点亮数码管，数码管扫描显示电路时序如图4.21所示。

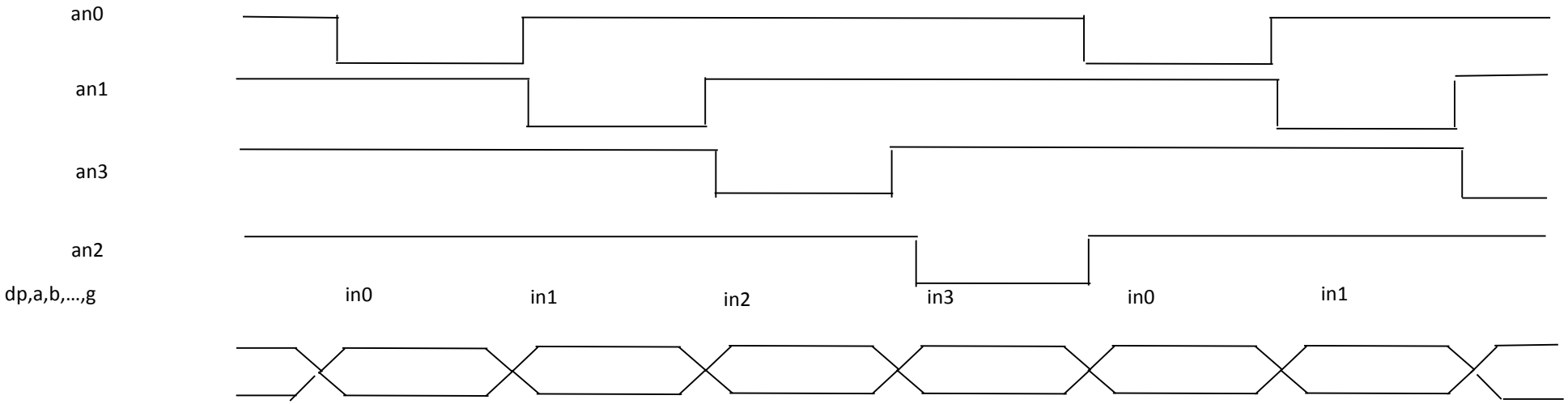


分时复用的扫描显示利用了人眼的视觉暂留特性，如果公共端控制信号的刷新速度足够快，人眼就不会区分出LED的闪烁，认为4个数码管是同时点亮。



例 4.20 数码管扫描显示电路

4.5.1 数码管扫描显示电路



例 4. 21 数码管扫描显示电路时序图

- 分时复用的数码管显示电路模块含有四个控制信号an3、an2、an1和an0，以及与控制信号一致的输出段码信号sseg。
- 控制信号的刷新频率必须足够快才能避免闪烁感，但也不能太快，以免影响数码管的开关切换，最佳工作频率为1000Hz左右。

4.5.1 数码管扫描显示电路

```
module scan_led_disp
    (
     input  clk,reset,
     input [7:0] in3,in2,in1,in0,
     output reg [3:0] an,
     output reg [7:0] sseg
    );
    localparam N = 18; //对输入50MHz时钟进行分频
(50 MHz/2^16)
    reg [N-1:0]  regN;

    always @(posedge clk,posedge reset)
       if (reset)
            regN <= 0;
       else
            regN <= regN + 1;
```

```
    always @*
        case (regN[N-1:N-2])
            2'b00: begin
                        an = 4'b1110;
                        sseg = in0;
                   end
            2'b01: begin
                        an = 4'b1101;
                        sseg = in1;
                   end
            2'b10:  begin
                        an = 4'b1011;
                        sseg = in2;
                   end
            default: begin
                        an = 4'b0111;
                        sseg = in3;
                     end
        endcase
endmodule
```

例 4.16 4位数码管动态显示电路的描述示例

4.5.1 数码管扫描显示电路

- 七段式数码管常用于显示十六进制数字，为了能够利用例4.16介绍的分时复用电路，我们需要四个译码电路。
- 另外一个更好的选择是首先输出多路十六进制数据，然后将其译码。这种方案只需要一个译码电路，使4选1数据选择器的位宽从8位降为了5位（4位16进制数和1位小数点）。
- 实现代码如例4.17所示。除clock和reset信号之外，输入信号包括4个4位十六进制数据：hex3，hex2， hex1，hex0，和dp_in中的4位小数点。

例 4.17 4位16进制数的数码管动态显示电路描述

```
module scan_led_hex_disp
(   input  clk,reset，
    input [3:0] hex3,hex2,hex1,hex0，
    input [3:0] dp_in，
    output reg [3:0] an，
    output reg [7:0] sseg
    );
    localparam N = 18; //对输入50MHz时钟进行分频(50 MHz/2^16)
    reg [N-1:0]  regN;
    reg [3:0] hex_in;
    always @(posedge clk,posedge reset)
       if (reset)
            regN <= 0;
       else
            regN <= regN + 1;
    always @*
        case (regN[N-1:N-2])
                2’b00:
                        begin
                                an = 4’b1110;
                                hex_in = hex0;
                                dp = dp_in[0];
                        end
            2’b01:
                        begin
                                an = 4’b1101;
                                hex_in = hex1;
                                dp = dp_in[1];
                        end
            2’b10:
                        begin
                                an = 4’b1011;
                                hex_in = hex2;
                                dp = dp_in[2];
                        end
            default:
                        begin
                                an = 4’b0111;
                                hex_in = hex3;
                                dp = dp_in[3];
                        end
        endcase
    always @*
    begin
            case (hex_in)
                        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;
                        default: sseg[6:0] = 7’b0111000;  //4’hf
            endcase
            sseg[7] = dp;
     end
endmodule
```

4.5.1 数码管扫描显示电路

我们可以在实际的FPGA电路中验证该设计，把8位开关数据作为了两个4位无符号数据的输入，并使两个数据相加，将其结果显示在四位七段式数码管上。实现代码如例4.18所示。

```
module scan_led_hex_disp_test
   ( input  clk ,
     input [7:0] sw ,
     output [3:0] an ,
     output [7:0] sseg
     );
     wire [3:0] a,b;
     wire [7:0] sum;
```

```
 assign a = sw[3:0];
 assign b = sw[7:4];
 assign sum = {4'b0, a} + {4'b0, b};
      //实例化4位16进制数动态显示模块
scan_led_hex_disp    scan_led_disp_unit
      (.clk(clk), .reset(1'bO),
.hex3(sum[7:4]), .hex2(sum[3:0]), .hex1(b),
.hexO(a),
.dp_in(4’ b1011), .an(an), .sseg(sseg));
      endmodule
```

例 4.18 4位16进制数的数码管动态显示测试

4.5.2 秒表

- 秒表显示的时间分为三个十进制数字，从00.0到99.9秒循环计数。它包含一个同步清零信号clr，使秒表返回00.0。还包含一个启动信号go，开始或者暂停计数。
- 本设计是一个BCD（二-十进制代码）编码的计数器。在这种格式中，一个十进制数由4位BCD数字表示。例如139表示为“0001 0011 1001”和下一个数字是140表示为“0001 0100 0000”。
- 计数脉冲可以从50MHz时钟源产生，我们首先需要一个最大值为5,000,000的计数器，每0.1秒生成一个时钟周期的脉冲，用于三位BCD计数器的计数时钟。
- 本设计中，BCD计数器采用同步电路设计方法进行设计。
- 秒表的Verilog HDL描述如例4.19所示。

4.5.2 秒表

```
module stop_watch
   (input  clk ,
    input  go,clr ,
    output [3:0] d2,d1,d0;
    );
    localparam COUNT_VALUE = 5000000;
    reg [22:0] ms_reg;
    reg [3:0] d2_reg, d1_reg, d0_reg;
    reg dp;
    wire ms_tick;
    reg [3:0] d2_next, d1_next, d0_next;
    always @(posedge clk)
       begin
         if (clr==0)
         begin
            ms_reg <=23'b0;
            d2_reg <= 4'b0;
            d1_reg <= 4'b0;
            d0_reg <= 4'b0;
         end
```

```
else if (go == 1)
              begin
                    d2_reg <= d2_next;
                    d1_reg <= d1_next;
                d0_reg <= d0_next;
                if (ms_reg < COUNT_VALUE)
                   ms_reg <= ms_reg + 1;
                else
                   ms_reg <= 23'b0;
              end
          end
assign ms_tick = (ms_reg == COUNT_VALUE) ? 1'b1
: 1'b0;
```

例 4.19 秒表电路的Verilog HDL描述（未完待续）

4.5.2 秒表

```
always @*
begin
    if (ms_tick)
        if (d0_reg != 9)
            d0_next = d0_reg + 1;
        else
            begin
                d0_next = 4'b0;
                if (d1_reg != 9)
                    d1_next = d1_reg + 1;
                else
                    begin
                        d1_next = 4'b0;
                        if (d2_reg != 9)
                            d2_next = d2_reg + 1;
                        else
                            d2_next = 4'b0;
                    end
            end
end
```

```
    assign d2 = d2_reg;

    assign d1 = d1_reg;

    assign d0 = d0_reg;

endmodule
```

例 4.19 秒表电路的Verilog HDL描述（续）

4.5.2 秒表

```
module stop_watch_test
   ( input  clk,
     input [1:0] btn,
     output [3:0] an,
     output [7:0] sseg
     );
     wire [3:0] d2,d1,d0;
```

为了验证例4.19的秒表电路，我们可以把它与前面的十六进制分时复用数码管电路结合，显示出秒表的输出。实现代码如例4.20所示。

注意：数码管的第一位显示0，信号go和clr分别对应两个标号为btn的IO口。

```
scan_led_hex_disp     scan_led_disp_unit
 (.clk(clk),
  .reset(1'bO),
  .hex3(4'b0),
  .hex2(d2),
  .hex1(d1),
  .hexO(d0),
  .dp_in(4'b1011),
  .an(an),
  .sseg(sseg)
);
//实例化秒表
stop_watch   counter_unit
              (.clk(clk), .go(btn[1]), .clr(btn[0]),
.d2(d2), .d1(d1), .d0(d0));

 endmodule
```

例 4.20 秒表电路的测试例程

4.6 练习题

4.6.1 可编程的方波信号发生器

一个可编程的方波发生器是可以产生用变量(逻辑1)和(逻辑0)表示的方波。指定的时间间隔由两个4比特的无符号整数控制信号m和n指定。打开和关闭的时间间隔是m * 100 ns和n * 100 ns。

1. 设计出程序并且进行验证仿真。
2. 下载到FPGA板上，利用按键输入m和n，并且在示波器上显示波形。

4.6.2　pwm和LED调光器

方波的占空比表示在一个周期内高电平（逻辑1）的百分比。PWM(脉冲宽度调制)电路可以输出一个可变占空比的方波。一个4比特分辨率PWM中，4比特控制信号w指定占空比。w信号是一个无符号整数和占空比为 $\frac{w}{16}$.

1．设计一个4比特分辨率的PWM电路，写出程序并且仿真。

2．将程序下载到FPGA板上用示波器进行验证。

3．在分时复用LED电路的基础上增加一个信号，并加入PWM电路。PWM电路指定LED点亮的时间百分比。我们可以通过改变占空比控制LED的亮度。可以通过观察示波器验证电路的操作。

4.6.3　LED循环电路

开发板有四个七段式LED数码管，因此一次只能显示四个符号。我们通过将数据不断旋转和移动就可以显示更多的信息。例如,假设输入信息是10位数(如“0123456789”)，可以将其显示为“0123”“1234”“2345”……“6789”“7890”“0123”。．

1. 电路的输入信号en进行启用或暂停旋转，输入信号dir指定方向（向左或向右)。设计程序并且下载到FPGA板上进行验证。
2. 用按键控制循环，按一下显示下一组数。设计程序并且下载到FPGA板上进行验证。

4.6.4 增强秒表

在4.52的秒表设计基础上进行扩展：

1. 添加一个额外的信号up，来控制计数的方向。当up有效时，秒表进行正方向计时，否则进行倒计时。设计程序并且下载到FPGA开发板上验证。
2. 添加分钟数字显示。LED显示格式为M.SS.D，D代表0.1秒和它的范围是0到9之间。SS表示秒，其范围是00和59之间。M代表分钟，它的范围是0到9之间。设计程序并且下载到FPGA开发板上验证。

XILINX

UNIVERSITY PROGRAM

xup@xilinx.com

4.5.1 数码管扫描显示电路

许多时序逻辑电路一般工作在相对较低的频率，就像分时复用LED电路中的使能脉冲一样。这可以通过使用计数器来产生只有一个时钟周期的使能信号。在这个电路中使用的是18位计数器。

```
module scan_led_hex_disp
 (   input  clk,reset ,
     input [3:0] hex3,hex2,hex1,hex0 ,
     input [3:0] dp_in ,
     output reg [3:0] an ,
     output reg [7:0] sseg
     );
     localparam N = 18; //
     reg [N-1:0]  regN;
     reg [3:0] hex_in;
       always @(posedge clk,posedge
reset)
         if (reset)
           regN <= 0;
         else
            regN <= regN + 1;
```

4.5.1 数码管扫描显示电路

考虑到计数器的位数,仿真这种电路需要消耗大量的计算时间（2^18个时钟周期为一个周期)。因为我们的主要工作在于分时复用那段代码，大部分模拟时间被浪费了。更高效的方法是使用一个较小的计数器进行仿真。我们可以通过修改常量声明

localparam N = 4;

这样就只需要2^4个时钟周期为一个仿真周期，并且我们可以更好地观察关键操作。

最好将定义N作为参数，而不是将其设置为一个常量，在仿真与综合时修改代码。在实例化过程中,我们可以对于仿真和综合设置不同的值。

```
module scan_led_hex_disp
 (   input  clk,reset，
     input [3:0] hex3,hex2,hex1,hex0，
     input [3:0] dp_in，
     output reg [3:0] an，
     output reg [7:0] sseg
     );
     localparam N = 18; //
     reg [N-1:0]  regN;
     reg [3:0] hex_in;
       always @(posedge clk,posedge
reset)
         if (reset)
           regN <= 0;
         else
            regN <= regN + 1;
```