第5章

典型电路的VHDL设计

前面各章介绍了 VHDL 语言的基本的结构、语句及语法，同时介绍了常用的设计软件 Quartus Ⅱ的基本操作，也接触了一些利用 VHDL 语言设计硬件电路的基本方法。本章重点介绍利用 VHDL 语言进行一些典型电路的设计，包括常用的组合逻辑电路、计数器电路、各种形式的分频电路、LED 显示电路、LCD 显示电路、键盘扫描电路以及常用的三态缓冲器和总线缓冲器等接口电路的设计。

5.1 组合逻辑电路的设计

组合逻辑电路的特点：在组合电路中，任意时刻的输出信号仅取决于该时刻的输入信号，与信号作用前电路原来的状态无关。组合逻辑表示形式包括逻辑函数式、逻辑图、真值表。下面分别介绍用 VHDL 描述各种表示形式的组合逻辑电路。

1. 逻辑函数式使用直接赋值语句实现

例 5-1：全加器的 VHDL 描述。

全加器的逻辑函数式：

$$S = A \oplus B \oplus C$$

$$CY = AB + AC + BC$$

```
library IEEE;
entity fullAdder is
port(A,B,C: in std_logic;
     CY : out std_logic;
     S : out std_logic);
end fulladder;
architecture a of fullAdder is
begin
S <= A xor B xor C;
```

```
  CY <= (A and B) or (A and C) or (B and C);
end a;
```

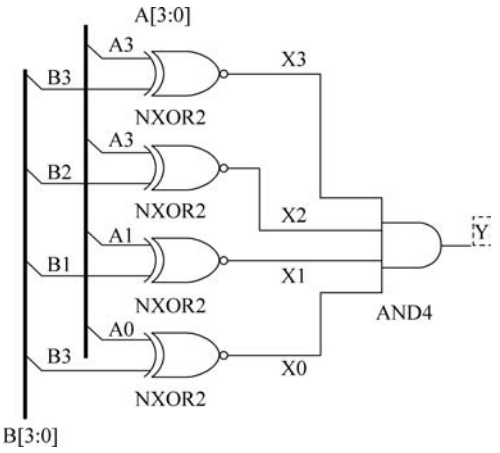

图 5-1　一个组合逻辑电路图

2. 逻辑图使用元件例化语句、生成语句实现

例 5-2：图 5-1 逻辑图的 VHDL 描述。

```
architecture A of eqcomp4 is
 component and4
 port (A, B, C, D: instd_logic;
       Y: out std_logic);
 end component;
 component xnor2
 port (M, N :in std_logic;
    P :out std_logic );
 end component;
 signal X: std_logic_vector(3 to 0);
begin
   U0: xnor2 port map (A(0),B(0),X(0));
   U1: xnor2 port map (A(1),B(1),X(1));
   U2: xnor2 port map (A(2),B(2),X(2));
   U3: xnor2 port map (A(3),B(3),X(3));
   U4: and4  port map (X(0),X(1),X(2),X(3), Y );
end A;
```

3. 真值表使用 with…select 语句或 case…when 语句或 when…else 语句设计实现

1）编码器

编码器是能够完成编码功能的电路。其功能是把 2^N 个分离的信息输入代码转化为 N 位二进制编码输出。目前使用的编码器有普通编码器和优先编码器两类。

编码器常常用于影音压缩或通信方面，可以达到精简传输量的目的。如常见的键盘里就有大家天天打交道的编码器，当你敲击按键时，被敲击的按键被键盘里的编码器编码成计算机能够识别的 ASCII 码。

例如，常见的 8 线-3 线编码器输入 I7～I0 八路信号，输出是 Y2、Y1、Y0 三位二进制代码。EN 是控制输入端，当 EN＝1 时，编码器工作；当 EN＝0 时，编码器输出“000”。8 线-3 线编码器的真值表如表 5-1 所示。

表 5-1　8 线-3 线编码器真值表

输 入 信 号									输出信号		
EN	I7	I6	I5	I4	I3	I2	I1	I0	Y2	Y1	Y0
0	X	X	X	X	X	X	X	X	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	1	0	0	0	1	0
1	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	1	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	0	0	1	1	0
1	1	0	0	0	0	0	0	0	1	1	1

例 5-3：8 线-3 线编码器的 VHDL 程序。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity coder is
  port (
       I : in std_logic_vector(7 downto 0);
       EN : in std_logic;
       Y : out std_logic_vector( 2 downto 0));
end coder;

architecture a of coder is
signal sel: std_logic_vector(8 downto 0);
begin
  sel <= EN & I;                    -- 将 EN,I7,I6,…,I0 合并以简化程序
process (I,EN)
     begin
case sel is
when "110000000" => Y <= "111";
when "101000000" => Y <= "110";
when "100100000" => Y <= "101";
when "100010000" => Y <= "100";
when "100001000" => Y <= "011";
when "100000100" => Y <= "010";
when "100000010" => Y <= "001";
when "100000001" => Y <= "000";
when others => Y <= "000";          -- 包含 EN = 0 的情况
end a;
```

其仿真图如图 5-2 所示。

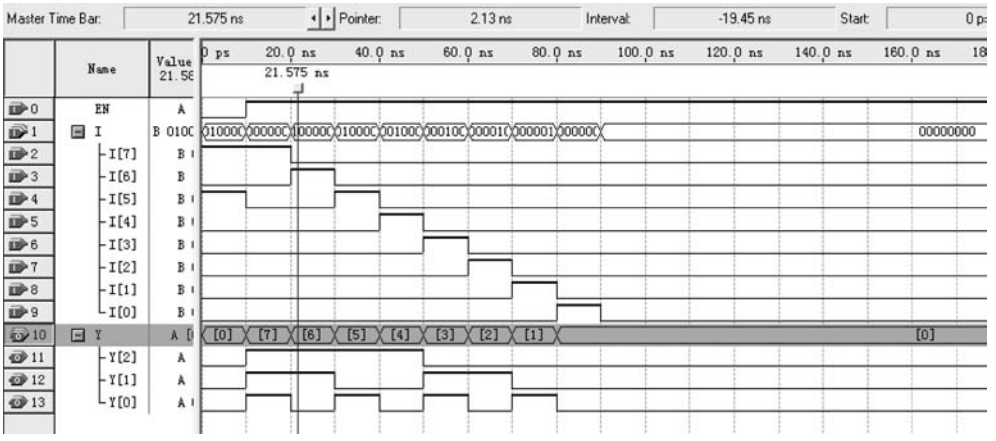

图 5-2　编码器仿真图

2）译码器

译码器是实现译码功能的数字电路。其逻辑功能是将输入的每个代码分别译成高电平(或低电平)。实现微机系统中存储器或输入/输出接口芯片的地址译码是译码器的一个典型用途。

例如,常见的 3 线-8 线译码器,输入为 A2A1A0 三位二进制代码,输出 Y7～Y0 八个输出信号,EN 是控制输入端,当 EN＝1 时,译码器工作；当 EN＝0 时,译码器输出全部是特定电平,本例为低电平。其真值表如表 5-2 所示。

表 5-2　3 线-8 线译码器真值表

输入信号				输 出 信 号							
EN	A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	1	0	0	1	0	0	0
1	1	0	0	1	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

例 5-4：3 线-8 线译码器的 VHDL 程序。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity encoder is
```

```
  port (
      A: in std_logic_vector( 2 downto 0);
      EN: in std_logic;
      Y: out std_logic_vector( 7 downto 0));
end encoder;

architecture a of encoder is
signal sel: std_logic_vector( 3 downto 0);
begin
sel <= EN & A;                          -- 将 EN、A2、A1、A0 合并以简化程序
with  sel  select
    Y <= "00000001" when "1000",
         "00000010" when "1001",
         "00000100" when "1010",
         "00001000" when "1011",
         "00010000" when "1100",
         "00100000" when "1101",
         "01000000" when "1110",
         "10000000" when "1111",
         "00000000" when others;   -- 包含 EN = 0 的情况
end a;
```

注意："&"是并置运算符，实现将多个信号合并成总线形式。

```
sel(3) <= EN;
sel(2) <= A(2);
sel(1) <= A(1);
sel(0) <= A(0);
```

其仿真图如图 5-3 所示。

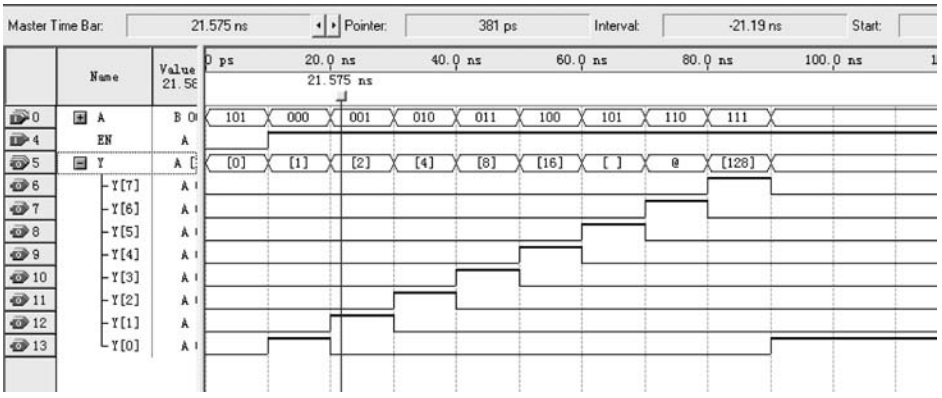

图 5-3　译码器仿真图

5.2　计数器的 VHDL 设计

计数器是在数字系统中使用最多的时序电路，不仅能用于对时钟脉冲计数，还可以用于分频、定时，产生节拍脉冲和脉冲序列，以及进行数字运算等。计数器的原理：采用几个触

发器的状态，按照一定规律随时钟变化记忆时钟的个数。一个计数器所能记忆时钟脉冲的最大数目称为计数器的模。计数器根据不同的分类依据，可以分为同步计数器、异步计数器，或者分为加法计数器、减法计数器和可逆计数器。

5.2.1 同步加法计数器

带异步复位端的同步加法计数器如例 5-5 所示，clk 是时钟输入端，上升沿有效；clr 是异步清零控制端，高电平有效；当 clr＝1 时，允许计数器计数清零；Num 为计数值。

例 5-5：带异步复位端的计数器。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity counter is
     port (clk, clr: in std_logic;
num:buffer integer range 0 to 9);
end counter;
architecture rtl of counter is
begin
process(clr,clk)
begin
    if (clr = '1') then    -- 异步复位
       num <= 0;
    elsif  rising_edge(clk) then
  if num = 9 then
          num <= 0;
        else
           num <= num + 1;
     end if;
    end if;
end process;
end rtl;
```

带异步复位端的计数器仿真结果如图 5-4 所示。

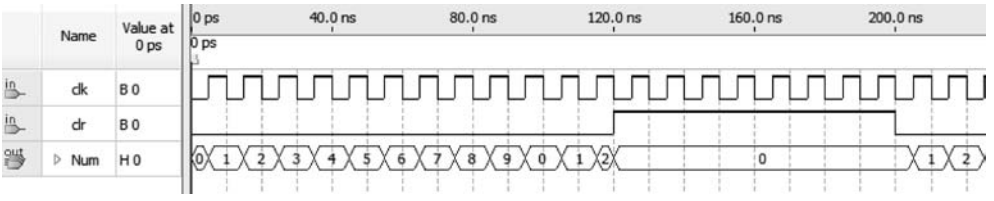

图 5-4　例 5-5 的仿真波形

5.2.2 同步可逆计数器

同步可逆计数器可以用加法计数也可以用减法计数方式。例 5-6 中，用一个控制信号 updn 来控制计数器的计数方式，当 updn＝1 时，计数器实现加法计数；updn＝0 时，计数器实现减法计数功能。clr 信号为异步清零控制端，高电平有效。

例 5-6：可逆计数器(加减计数器)。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
      port (clk,clr,updn: in std_logic;
                        qa,qb,qc,qd,qe,qf: out std_logic);
end updncount64;
architecture rtl of updncount64 is
      signal count_6: std_logic_vector(5 downto 0 );
begin
       qa <= count_6(0);
       qb <= count_6(1);
       qc <= count_6(2);
       qd <= count_6(3);
       qe <= count_6(4);
       qf <= count_6(5);
process(clr,clk)
begin
     if (clr = '0') then
        count_6 <= "000000";
     elsif (clk'event and clk = '1') then
          if (updn = '1' ) then
             count_6 <= count_6 + 1;
          else
              count_6 <= count_6 - 1;
           end if;
      end if;
end process;
end rtl;
```

可逆计数器仿真结果如图 5-5 所示。

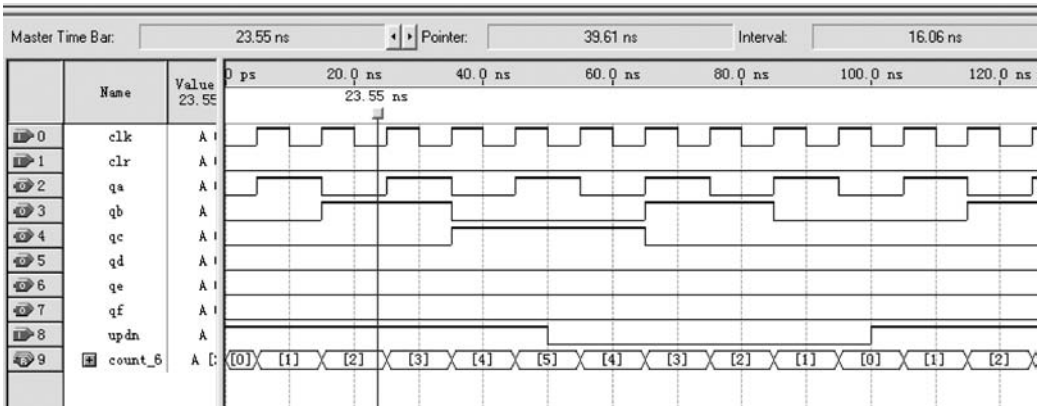

图 5-5　例 5-6 的仿真波形

5.2.3 同步六十进制加法计数器

在数字系统中，常常用 BCD 码来表示十进制数，即用 4 位二进制码表示 1 位十进制数，用 8 位二进制码表示 2 位十进制数。在时间计数电路中常用作秒和分计数。

例 5-7 同步六十进制加法计数器中，ci 为计数控制端，ci＝1 时，计数器开始计数；nreset 为异步复位控制端，nreset＝0 时，计数器复位为零；load 为置数控制端，load＝1 时，同步置数；d 为待预置的数；clk 为时钟信号，上升沿有效；co 为进位输出端；qh 为输出端的高 4 位；ql 为输出端的低 4 位。

例 5-7：用 VHDL 设计一个模为 60，具有异步复位、同步置数功能的 BCD 码计数器。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity cnt60 is
 port(ci:in std_logic;                                  --计数控制
        nreset:in std_logic;                            --异步复位控制
        load:in std_logic;                              --置数控制
        d:in std_logic_vector(7 downto 0);              --待预置的数
        clk:in std_logic;
        co:out std_logic;                               --进位输出
        qh:buffer std_logic_vector(3 downto 0);         --输出高 4 位
        ql:buffer std_logic_vector(3 downto 0));        --输出低 4 位
end entity cnt60;
architecture art of cnt60 is
  begin
  co<='1'when(qh="0101"and ql="1001"and ci='1')else'0';
   --进位输出的产生
  process(clk,nreset) is
    begin
if (nreset='0')then                                     --异步复位
       qh<="0000";
       ql<="0000";
    elsif(clk'event and clk='1')then                    --同步置数
      if (load='1')then
         qh<=d(7 downto 4);
         ql<=d(3 downto 0);
    elsif(ci='1')then                                   --模 60 的实现
        if (ql=9)then
           ql<="0000";
           if(qh=5)then
              qh<="0000";
           else
              qh<=qh+1;                                 --计数功能的实现
           end if;
    else
           ql<=ql+1;
```

```
            end if;
          end if;
        end if;
    end process;
end architecture art;
```

其仿真波形如图 5-6 所示。

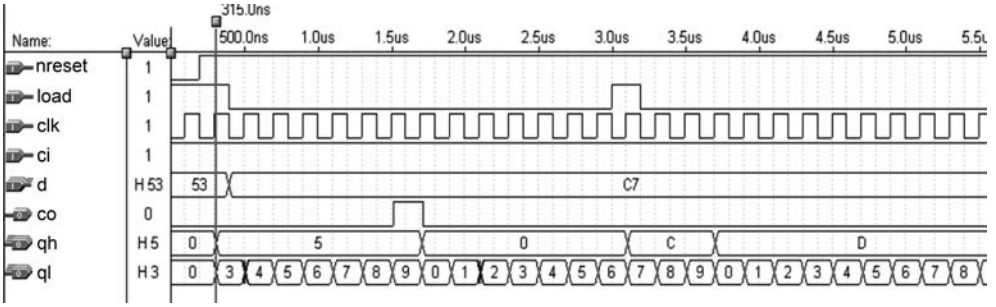

图 5-6　例 5-7 的仿真波形

5.3　分频器的 VHDL 设计

5.3.1　分频器的原理

所谓分频电路，就是将一个给定的频率较高的数字输入信号，经过适当的处理后，产生一个或数个频率较低的数字输出信号。分频电路本质上是加法计数器的变种，其计数值由分频系数 rate＝f_{in}/f_{out} 决定，其输出不是一般计数器的计数结果，而是根据分频常数对输出信号的高、低电平进行控制。

分频系数(倍率)：rate＝f_{in}/f_{out}，即输出的信号频率如果是输入信号频率的 1/2，称为 2 分频率；1/3，称为 3 分频；1/n，称为 n 分频。

占空比(DUTY CYCLE)，占空比在电信领域中有如下含义：在一串理想的脉冲序列中(如方波)，正脉冲的持续时间与脉冲总周期的比值。例如，正脉冲宽度 1ms，信号周期 4ms 的脉冲序列占空比为 0.25 或者为 1∶4。

常见的分频器有偶数分频器、奇数分频器。

5.3.2　偶数分频器的设计

分频系数 rate＝even(偶数)，占空比为 50%。

设计原理：定义一个计数器对输入时钟进行计数，在计数的前一半时间里，输出高电平，在计数的后一半时间里，输出低电平，这样输出的信号就是占空比为 50%的偶数分频信号。例如，6 分频，计数值为 0～2 输出高电平，计数值为 3～5 输出低电平。

例 5-8：偶数分频器的 VHDL 源程序。

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity fdiv is
  generic(n: integer:=6);        --rate=n,n是偶数
  port(
        clkin: in std_logic;
        clkout: out std_logic
        );
end fdiv;
architecture a of fdiv is
  signal cnt: integer range 0 to n-1;
begin
  process(clkin)   --计数
  begin
      if(clkin'event and clkin='1') then
          if(cnt<n-1) then
              cnt<= cnt+1;
          else
              cnt<= 0;
          end if;
      end if;
  end process;

  process(cnt)   --根据计数值,控制输出时钟脉冲的高、低电平
  begin
      if(cnt<n/2) then
          clkout<= '1';
      else
          clkout<= '0';
      end if;
  end process;
end a;
```

当然也可以用一个进程实现偶数分频器,可以使程序更简洁。

例 5-9：常用的偶数分频器的 VHDL 程序。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity fdiv is
  generic(n: integer:=6);        --rate=n,n是偶数
  port(
        clkin: in std_logic;
        clkout: out std_logic
        );
end fdiv;
architecture a of fdiv is
  signal cnt: integer range 0 to n/2-1;
```

```
  signal temp: std_logic;
begin
  process(clkin)
  begin
      if(clkin'event and clkin = '1') then
          if(cnt = n/2 - 1) then
              cnt <= 0;
              temp <= not temp;
          else
              cnt <= cnt + 1;
          end if;
      end if;
  end process;
  clkout <= temp;
end a;
```

其仿真波形如图 5-7 所示。

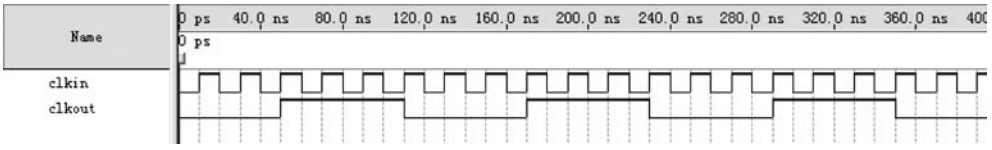

图 5-7　例 5-9 的仿真波形

说明：

(1) 从波形图可以看到，clkout 是 clkin 的 6 分频。

(2) 如果要产生其他分频，直接修改 generic 类属变量参数即可。

5.3.3　奇数分频器的设计

分频系数 rate＝odd(奇数)，占空比为 50％。

设计原理：错位法。定义两个计数器，分别对输入时钟的上升沿和下降沿进行计数，然后把这两个计数值输入一个组合逻辑，用其控制输出时钟的电平。这是因为计数值为奇数，占空比为 50％，前半个和后半个周期所包含的不是整数个 clkin 的周期。例如，5 分频，前半个周期包含 2.5 个 clkin 周期，后半个周期包含 2.5 个 clkin 周期。

例 5-10：奇数分频器的 VHDL 源程序。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity fdiv is
  generic(n: integer: = 5);                    -- rate = n,n 是奇数
  port(
        clkin: in std_logic;
        clkout: out std_logic
        );
```

```
end fdiv;
architecture a of fdiv is
  signal cnt1, cnt2: integer range 0 to n - 1;
begin
  process(clkin)
  begin
      if(clkin'event and clkin = '1') then   -- 上升沿计数
          if(cnt1 < n - 1) then
              cnt1 <= cnt1 + 1;
          else
           cnt1 <= 0;
       end if;
      end if;
  end process;
  process(clkin)
  begin
    if(clkin'event and clkin = '0') then   -- 下降沿计数
        if(cnt2 < n - 1) then
            cnt2 <= cnt2 + 1;
        else
         cnt2 <= 0;
        end if;
    end if;
  end process;
    clkout <= '1' when cnt1 <(n - 1)/2 or cnt2 <(n - 1)/2 else '0';
end a;
```

其仿真波形如图 5-8 所示。

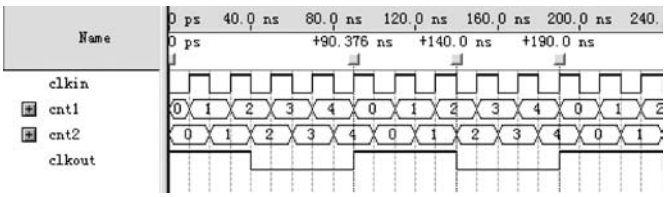

图 5-8　例 5-10 的仿真波形

从波形图可以看到，clkout 是 clkin 的 5 分频。如果要产生其他分频，直接修改 generic 类属变量参数即可。

5.3.4　占空比可调的分频器的设计

分频比为 n，占空比可调的分频器设计，要求占空比为 $m:n$，$m<n$。

设计原理：定义一个计数器，对输入时钟脉冲进行计数，根据计数值判断输出高电平还是低电平。例如，占空比为 3:10 的偶数分频器，当计数值为 0～2 时，输出高电平；当计数值为 3～9 时，输出低电平。

例 5-11：占空比可调的分频器设计，clkin 为输入信号，clkout 为输出信号，其占空比为 $m:n$，分频比为 n，VHDL 语言描述的程序如下。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity fdiv is
  generic(
         n: integer: = 10;
         m: integer: = 3          -- 占空比 m:n,rate = n
         );
  port(
        clkin: in std_logic;
        clkout: out std_logic
        );
end fdiv;

architecture a of fdiv is
  signal cnt: integer range 0 to n - 1;
begin
  process(clkin)
  begin
      if(clkin'event and clkin = '1') then
        if(cnt < n - 1) then
              cnt < = cnt + 1;
          else
              cnt < = 0;
          end if;
      end if;
  end process;
    clkout < = '1' when cnt < m else '0';
end a;
```

其仿真图如图 5-9 所示。

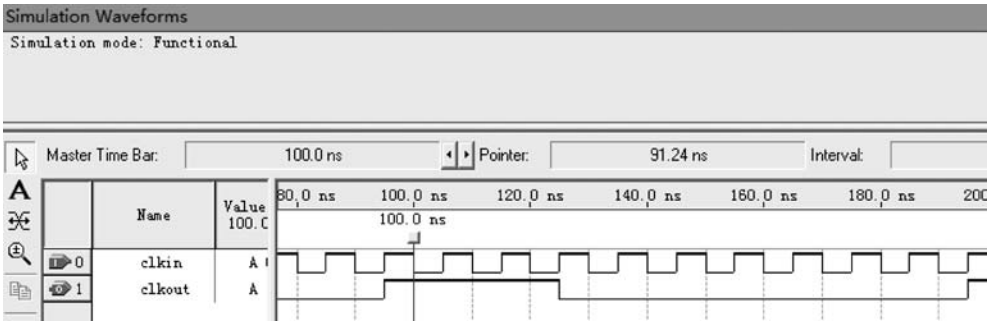

图 5-9　占空比可调的分频器仿真图

从波形图可以看到，clkout 是 clkin 的 10 分频，且占空比为 3:10。如果要产生其他分频，直接修改 generic 类属变量参数即可。

5.4 数码管显示电路 VHDL 设计

译码器除了应用在地址总线进行地址选址和芯片的选通控制外，还可以用作显示译码，将 4 位 BCD 码转换成七段码输出，以便在数码管上显示。七段显示译码器是对一个 4 位二进制数进行译码，并在七段数码管上显示出相应的十进制数或十六进制数。

5.4.1 七段数码管显示译码器

七段数码管有共阳极、共阴极之分。图 5-10(a)是共阴极七段数码管的原理图，图 5-10(b)是其表示符号。使用时，数码管公共阴极接地，7 个阳极 a～g 由相应的 BCD 七段译码器来驱动(控制)。

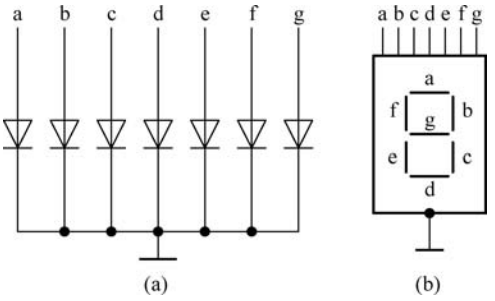

图 5-10 共阴极七段数码管原理图及符号

以输出并显示 BCD 码(十进制码)为例，数码管显示电路分为两个部分，一个是七段译码部分，将 4 位二进制码转换为 BCD 码；另一个是显示驱动部分，实现数码管的驱动。如图 5-11 所示，BCD 七段译码器的输入是一位 BCD 码(以 D、C、B、A 表示)，输出是数码管各段的驱动信号(以 a～g 表示)，也称 4 线-7 线译码器。如果用它驱动共阴极七段数码管，则输出应为高有效，即输出为高(1)时，相应显示段发光。每个相应的显示段也称为段选端或

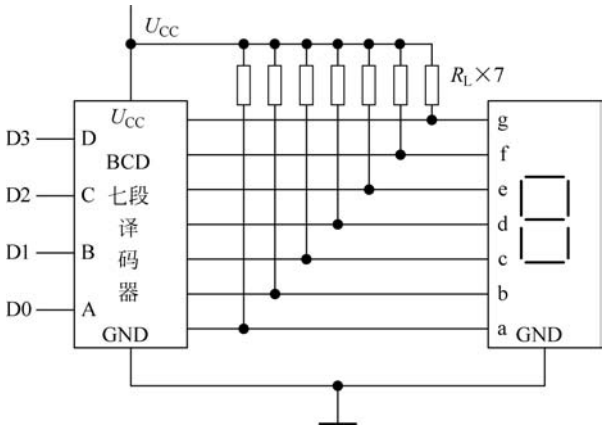

图 5-11 七段数码管显示电路

段选信号。例如，当输入 8421 码 DCBA＝0100 时，应显示的数据为 4，即要求同时点亮 b、c、f、g 段，熄灭 a、d、e 段，则译码器的输出应为 g～a＝1100110，这也是一组代码，常称为段码。同理，根据组成 0～9 这 10 个字形的要求可以列出 8421BCD 七段译码器的真值表，见表 5-3。

表 5-3　七段数码管显示译码电路的真值表

D3	D2	D1	D0	a	b	c	d	e	f	g	显示数字
0	0	0	0	1	1	1	1	1	1	0	**0**
0	0	0	1	0	1	1	0	0	0	0	**1**
0	0	1	0	1	1	0	1	1	0	1	**2**
0	0	1	1	1	1	1	1	0	0	1	**3**
0	1	0	0	0	1	1	0	0	1	1	**4**
0	1	0	1	1	0	1	1	0	1	1	**5**
0	1	1	0	1	0	1	1	1	1	1	**6**
0	1	1	1	1	1	1	0	0	0	0	**7**
1	0	0	0	1	1	1	1	1	1	1	**8**
1	0	0	1	1	1	1	1	0	1	1	**9**

例 5-12：七段数码管显示译码器 VHDL 描述。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity led_encode is
port
(d: in std_logic_vector(3 downto 0);
led7s: out std_logic_vector(6 downto 0));
end led_encode;

architecture one of led_encode is
begin
  process( d )
  begin
   case  d  is
   when "0000" =>  led7s <= "0111111";
   when "0001" =>  led7s <= "0000110";
   when "0010" =>  led7s <= "1011011";
   when "0011" =>  led7s <= "1001111";
   when "0100" =>  led7s <= "1100110";
   when "0101" =>  led7s <= "1101101";
   when "0110" =>  led7s <= "1111101";
   when "0111" =>  led7s <= "0000111";
   when "1000" =>  led7s <= "1111111";
   when "1001" =>  led7s <= "1101111";
   when others   =>  led7s <= "0000000";
  end case;
  end process;
end one;
```

其仿真图如图 5-12 所示。

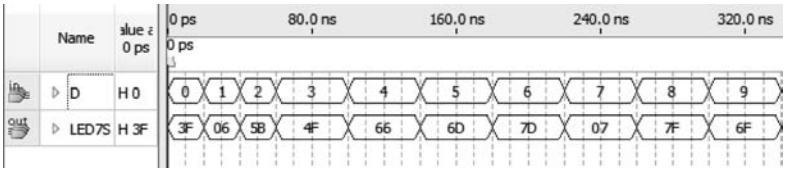

图 5-12　七段数码管显示译码电路仿真图

5.4.2　数码管静态显示

多位七段数码管可以显示多位十进制(或十六进制)数字,数码管显示分为静态显示和动态显示。静态显示同时显示各个字符,位码始终有效,显示内容完全与数据线上的值一致。静态驱动的优点是编程简单,缺点是占用硬件资源 I/O 较多。8 个数码管静态显示需要 8×8=64 根 I/O 端口来驱动,并且增加相应的驱动电路,这样就增加了电路的复杂度。其原理图如图 5-13 所示。由于静态显示编程比较简单,所以这里不做详细介绍。

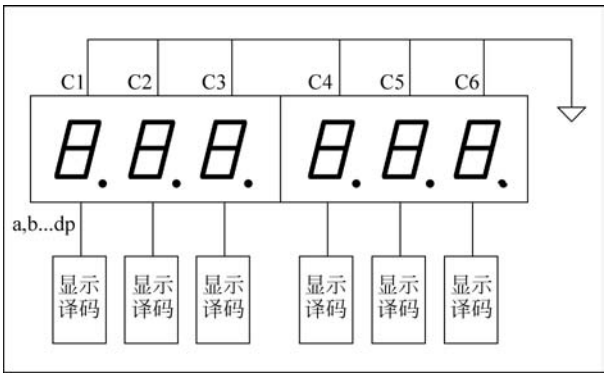

图 5-13　数码管静态显示原理图

5.4.3　数码管动态显示

动态显示是轮流显示各个字符。利用人眼视觉暂留的特点,循环顺序变更位码,同时数据线上发送相对应的显示内容。在设计多位七段数码管显示驱动电路时,为了简化硬件电路,通常将所有位的各个相同段选线对应并接在一起,形成段选线的多路复用。而各位数码管的共阳极或共阴极分别由各自独立的位选信号控制,顺序循环地选通(即点亮)每位数码管,这样的数码管驱动方式就称为“动态扫描”。在这种方式中,虽然每一短暂时间段只选通一位数码管,但由于人眼具有一定的视觉残留,只要延时时间设置恰当,实际感觉到的会是多位数码管同时被点亮。数码管动态显示及位码顺序脉冲如图 5-14 所示。

FPGA 实现动态显示接口电路如图 5-15 所示。其中段选线(a～g,dp)占用 8 位 I/O 口,位选线(Y0～Y7)占用 8 位 I/O 口。由于各位的段选线并联,段选码的输出对各位来说都是相同的。因此,同一时刻,如果各位位选线都处于选通状态,8 位 LED 将显示相同的字符。若要各位 LED 能够显示出与本位相对应的字符,就必须采用扫描显示方式,即在某一位的位选线处于选通状态时,其他各位的位选线处于关闭状态,这样,8 位 LED 中只有选通

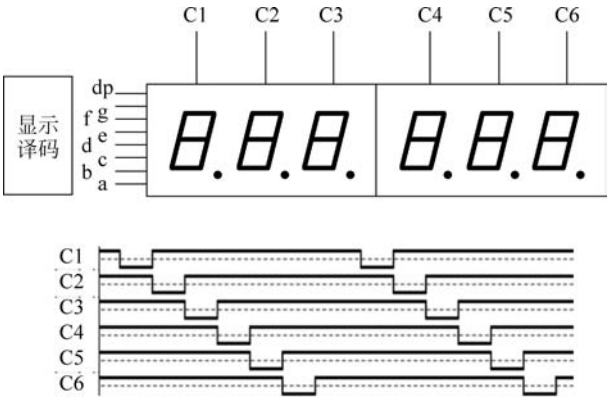

图 5-14　数码管动态显示及位码顺序脉冲

的那一位显示出字符，而其他位则是熄灭的。同样，在下一时刻，只让下一位的位选线处于选通状态，而其他的位选线处于关闭状态。如此循环下去，就可以使各位“同时”显示出将要显示的字符。由于人眼有视觉暂留现象，只要每位显示间隔足够短，则可造成多位同时亮的“景”象，达到完整显示的目的。

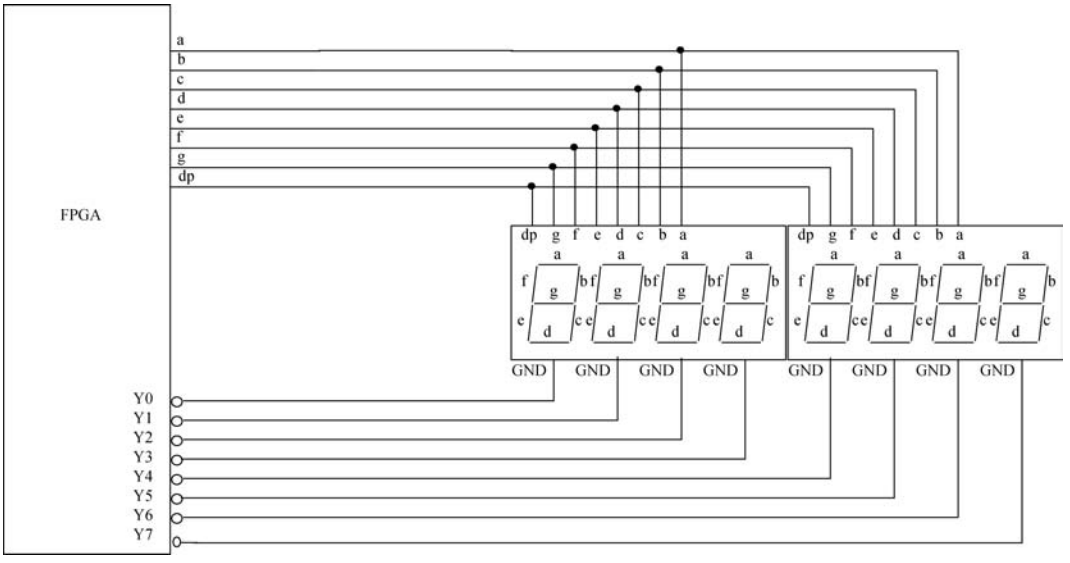

图 5-15　FPGA 实现动态显示接口电路

动态显示 VHDL 设计要点：

(1) 位选信号应为八进制计数器的状态输出；

(2) 八进制计数器计数脉冲的周期是位选信号的负脉冲的宽度；

(3) 位选信号脉冲的频率应大于 50Hz；

(4) 每位预显示内容应与位选信号同步。

例 5-13：七段数码管动态显示 VHDL 程序。

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity seg7_dsp is
port
(clk: in std_logic;                          -- clock
segout: out std_logic_vector(0 to 6);        --段码(dp…)
selout: out std_logic_vector(0 to 5);        --6个数码管片选端
numa:in integer range 0 to 9);
end seg7_dsp;
architecture a of seg7_dsp is
signal counter: integer range 0 to 5;
begin
    process (clk)                            -- 计数器计数
    variable num: integer range 0 to 9;
          begin
    if rising_edge(clk) then
       if counter = 5 then
          counter <= 0;
       else
          counter <= counter + 1;
    end if;
case counter is
   when 0 => selout <= "011111";
   num:= numa;
   when 1 => selout <= "101111";
   num:= numa;
   when 2 => selout <= "110111";
   num:= numa;
   when 3 => selout <= "111011";
   num:= numa;
   when 4 => selout <= "111101";
   num:= numa;
   when 5 => selout <= "111110";
   num:= numa;
   when others => selout <= "000000";
   num:= 0;
 end case;
case  num  is
   when 0 => segout <= "0111111";
   when 1 => segout <= "0000110";
   when 2 => segout <= "1011011";
   when 3 => segout <= "1001111";
   when 4 => segout <= "1100110";
   when 5 => segout <= "1101101";
   when 6 => segout <= "1111101";
   when 7 => segout <= "0000111";
   when 8 => segout <= "1111111";
   when 9 => segout <= "1101111";
   when others => segout <= "0000000";
```

```
end case;
  end if;
  end process;
end;
```

5.5　键盘接口电路的 VHDL 设计

在电子系统中，按键以及键盘是常见的输入装置。例如，按键产生的单脉冲信号作计数脉冲驱动计数器，用数字键盘置数，或用加、减按键置数。在应用设计中，常用的键盘输入电路有独立式键盘输入电路、矩阵式键盘输入电路。独立式键盘输入电路的最大优点是键盘电路结构简单；其缺点是当键数较多时，要占用较多的 I/O 口线。

5.5.1　独立式键盘

独立式键盘十分简单，如图 5-16 所示，按键的一端通过上拉电阻接至 Vcc，一端接 FPGA 的 I/O 口。当按键按下时，此端口为高电平，通过检测 I/O 口的电平就可知该按键是否被按下。其优点是电路结构简单、易行，连接方便；但是由于每个按键要占用 1 个 I/O 口，在系统需要很多按键时，使用这种方法显然会占用大量的 I/O 口，如果所选的芯片 I/O 口数量有限还要进行扩展或是分时复用，增加了整个电路设计的复杂性。

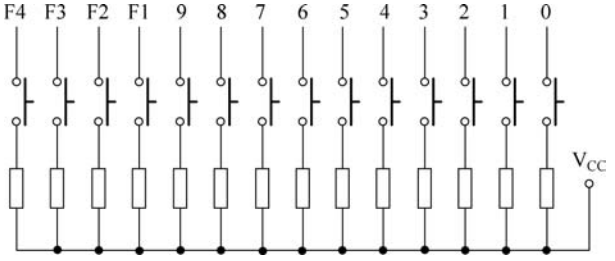

图 5-16　独立式键盘原理图

5.5.2　矩阵式键盘

矩阵式键盘控制比独立式按键要复杂一些，但其优点是节省了 I/O 口。假设矩阵式键盘有 n 行 m 列，则键盘上的按键数目有 $n \times m$ 个，然而这样连接的键盘只需要占用 $n+m$ 个 I/O 口。当设计的系统需要很多按键时，用矩阵式键盘显然比独立式按键要更合理，更节约资源。矩阵式键盘是由若干个按键排列成长方阵而成，如图 5-17 所示。

矩阵键盘工作原理是键盘扫描信号在扫描输出端 KX3、KX2、KX1、KX0 输出的 4 位扫描信号的变化顺序依次为 1110→1101→1011→0111→1110…；当扫描信号为 1110 时，就扫描 KY0 这一排按键，并检查是否有键按下，没有则忽略；反之，进行按键译码，并将译码结果保存在寄存器中。也就是说，当一个按键的行线为 0 时，如果这个键被按下，则列线读到的值为 0，否则为 1。

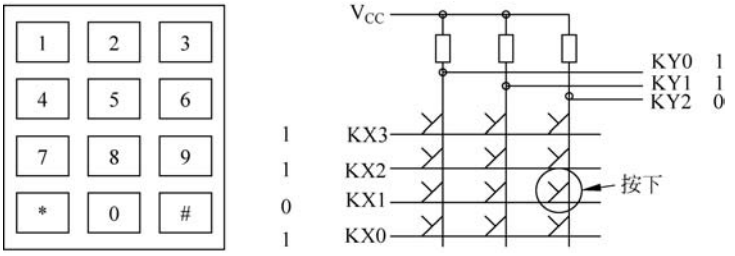

图 5-17　矩阵式键盘原理图

表 5-4　行、列码与按键的对应关系

扫描位置 行码 KX3～KX0	键盘输出 列码 KY2～KY0	对应的按键	按键功能
1110	011	1	数字输入
	101	2	数字输入
	110	3	数字输入
1101	011	4	数字输入
	101	5	数字输入
	110	6	数字输入
1011	011	7	数字输入
	101	8	数字输入
	110	9	数字输入
0111	011	*	清除输入
	101	0	数字输入
	110	#	确认输入
当没有任何按键按下时，译码输出“1111”			

例 5-14：矩阵式键盘扫描程序。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity keyboard is
  port
(clk : in std_logic;                          -- 扫描时钟频率不宜过高,1kHz 以下
kin : in std_logic_vector(0 to 2);            -- 读入列码
scansignal : out std_logic_vector(0 to 3);    -- 输出行码(扫描信号)
num : out integer range 0 to 12               -- 输出键值
);
end;

architecture scan of keyboard is
```

```
signal scans: std_logic_vector(0 to 7);
signal scn: std_logic_vector(0 to 3);
signal counter: integer range 0 to 3;          -- 计数产生扫描信号
begin
    process (clk)
    begin
        if rising_edge(clk) then
             if counter = 3 then
            counter = 0;
           else
         counter <= counter + 1;
         end if;
       case counter is                         -- 产生扫描信号
      when 0 => scn <= "1110";
      when 1 => scn <= "1101";
      when 2 => scn <= "1011";
      when 3 => scn <= "0111";
    end case;
        end if;
    end process;
        process (clk)
        begin
            if falling_edge(clk) then          -- 上升沿产生扫描信号,下降沿读入列码
            case scans is
            when "1110011" => num <= 0;
            when "1110101" => num <= 1;
            when "1110110" => num <= 2;
            when "1101011" => num <= 3;
            when "1101101" => num <= 4;
            when "1101110" => num <= 5;
            when "1011011" => num <= 6;
            when "1011101" => num <= 7;
            when "1011110" => num <= 8;
            when "0111011" => num <= 9;
            when "0111101" => num <= 10;
            when "0111110" => num <= 11;
            when others => null;
           end case;
            end if;
           end process;

        scans <= scn&kin;
        scansignal <= scn;
        end;
```

4×4 矩阵式键盘的 FPGA 连接图如图 5-18 所示。

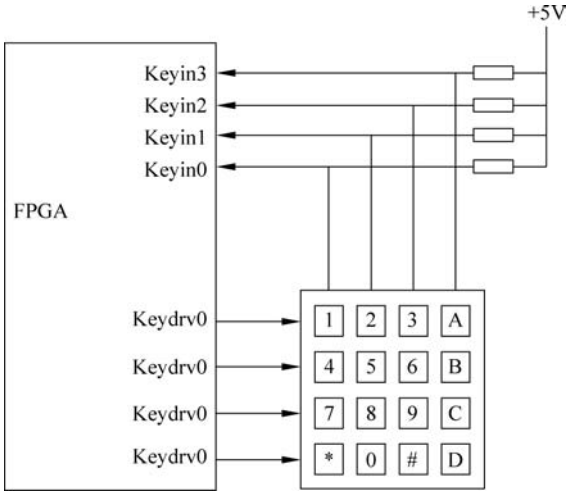

图 5-18　4×4 矩阵式键盘的 FPGA 连接图

5.5.3　键盘的消抖

现在使用的键盘大多是机械式的按键，由于机械式触点触动时的弹性作用，一个按键在闭合及断开的瞬间均伴随有一连串的抖动，如图 5-19 所示。抖动时间的长短由按键的机械特性决定，一般为 5～10ms。

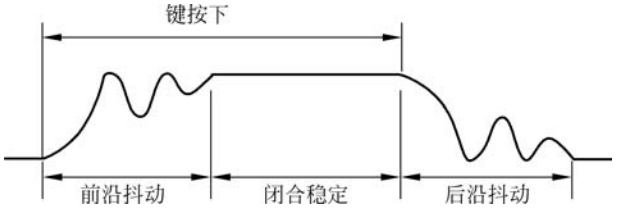

图 5-19　按键抖动示意图

键盘按键按下与释放的瞬间，都会引起输入信号产生毛刺。如果不进行消抖处理，系统会误以为毛刺是另一次输入，导致系统的误操作。最常用、最简单的消抖方法是计数法。其工作原理是对键值进行计数，当计数达到一定值时，延迟约为 10ms，当该按键键值保持一段时间不改变时，才确认它为有效键值；否则将其判为无效值，重新对键值进行计数。

例 5-15：键盘消抖程序。

```
library IEEE;
use IEEE.std_logic_1164.all;

entity antiwrite is
port
(clk : in std_logic;
numin: out integer range 0 to 12;
numout: out integer range 0 to 12
);
```

```
end;

architecture behavior of antiwrite is
signal tempnum: std_logic_vector(0 to 12);
signal counter: integer range 0 to 31;

begin
    process (clk)
    begin
      if rising_edge(clk) then
        tempnum = 12;                -- 上电对输出键值赋予无效值
        numout = 12;
        if numin/ = tempnum then     -- 上一键值与此键值不同
        tempnum = numin;             -- 记录该键值
        counter = 0;                 -- 计数器清零,准备计时
        else
          if counter = 31 then       -- 键值保持 31 个时钟周期不变时
          numout = numin;            -- 确认为有效键值,并且输出
          counter = 0;
          else
          counter < = counter + 1;
          end if;
        end if;
      end if;
    end process;
  end antiwrite;
```

5.6　三态门和总线缓冲器

在电子系统中,三态缓冲器和总线缓冲器是接口电路和总线驱动电路经常用到的器件。

三态门,是指逻辑门的输出除有高、低电平两种状态外,还有第三种状态——高阻状态的门电路,高阻态相当于隔断状态。三态门又称为三态缓冲器,可以使该电路在有其他电路使用总线时处于高阻态,不驱动共享总线(即避免总线竞争);

总线缓冲器使得该器件接口既可以输入信号,也可以输出信号(即 I/O 接口)。

5.6.1　三态门电路

三态门如图 5-20 所示,有三种可能的输出状态:高电平、低电平和高阻态。在设计中必须把数据定义为 std_logic 或 std_logic_vector 数据类型,才能有高阻态的状态,用大写字母 Z 表示高阻态。三态门主要用在总线结构中。通常三态门有一个输入、一个输出和一个控制端。

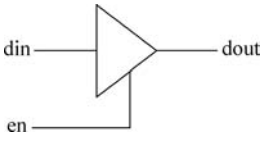

图 5-20　三态门

当 en＝'1'时,dout＝din;

当 en＝'0'时,dout＝'Z'(高阻)。

例 5-16：三态门的描述方法 1。

```
library IEEE;
use IEEE.std_logic_1164.all;
entity tri_gate is
    port (din, en: in std_logic;
                  dout: out std_logic);
end tri_gate;
architecture zas of tri_gate is
begin
tri_gate1: process (din, en)
    begin
        if (en) = '1' then
            dout <= din;
        else
            dout <= 'Z';   -- 注意'Z'要大写
        end if;
    end process;
    end zas;
```

例 5-17：三态门的描述方法 2。

```
library IEEE;
use IEEE.std_logic_1164.all;
entity tri_gate is
    port (din, en: in std_logic;
                  dout: out std_logic);
end tri_gate;
architecture zas of tri_gate is
begin
tri_gate2:
    begin
        dout <= din when en = '1' else 'Z';   -- 用并行信号赋值
    end zas;
```

其仿真图如图 5-21 所示。

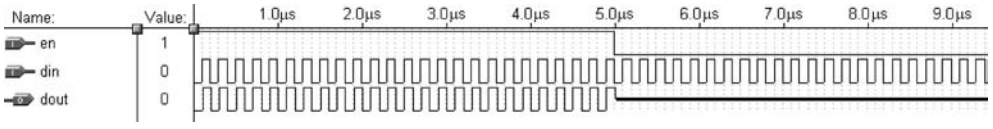

图 5-21　三态门仿真图

5.6.2　单向总线缓冲器

在微型计算机的总线驱动中经常要用到单向总线驱动器，由多个三态门组成，用来驱动地址总线和控制总线，如图 5-22 所示。

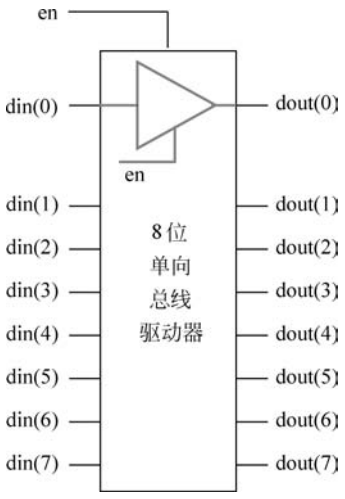

图 5-22　单向总线驱动器

例 5-18：单向总线驱动器的描述方法。

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tri_buf8 is port (
din : in std_logic_vector(7 downto 0);
en : in std_logic;
dout : out std_logic_vector(7 downto 0) bus );
end tri_buf8;

architecture zas of tri_buf8 is
begin
tri_buff: process (din, en)
    begin
        if (en) = '1' then
            dout <= din;
        else
            dout <= "ZZZZZZZZ";
        end if;
    end process;
end zas;
```

注意：

(1) 不能将“Z”值赋予变量，否则不能逻辑综合；

(2) “Z”和“0”或“1”不能混合使用；但可以分开赋值。如：

```
dout <= "Z001ZZZZ"          ←ERROR
dout(7) <= 'Z'              ←RIGHT
dout (6 downto 4) <= "001"  ←RIGHT
dout (3 downto 0) <= "ZZZZ" ←RIGHT
```

5.6.3 双向总线缓冲器

双向总线缓冲器用于在微型计算机的总线驱动中对数据总线的驱动和缓冲。图 5-23 中双向缓冲器有数据输入端 a、b，方向控制端 dir，选通端 en。en＝1 未选中，a、b 都呈现高阻；en＝0 选中，如果 dir＝0，则 a <＝b；如果 dir＝1，则 b <＝a。

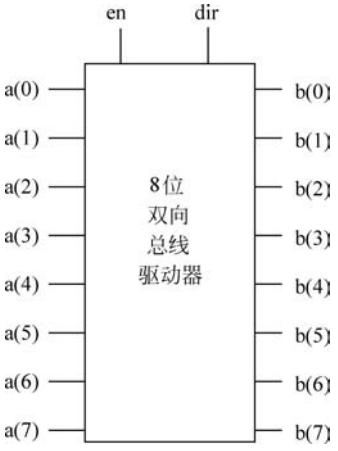

图 5-23 双向总线驱动器

例 5-19：双向总线驱动器的描述方法。

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tri_bigate is port (
a, b: inout std_logic_vector(7 downto 0) bus;
en: in std_logic;
dir: out std_logic);
end tri_bigate;

architecture tri of tri_bigate is
sigal aout, bout: std_logic_vector(7 downto 0);
begin
   process (a, dir, en)
 begin
    if (en = '0') and (dir = '1') then
        bout <= a;
    else
        bout <= "ZZZZZZZZ";
    end if;
    b <= bout;
        end process;

        process (b, dir, en)
        begin
```

```
    if (en = '0') and (dir = '0') then
        aout <= b;
    else
        aout <= "ZZZZZZZZ";
    end if;
    a <= aout;
        end process;
    end tri;
```

5.7　LCD1602 显示电路设计

液晶显示器以其微功耗、体积小、使用灵活等诸多优点在袖珍式仪表和低功耗应用系统中得到越来越广泛的应用。液晶显示器通常可分为两大类，一类是点阵型，另一类是字符型。点阵型液晶显示器通常面积较大，可以显示图形；而一般的字符型液晶显示器只有两行，面积小，能显示字符和一些很简单的图形，简单易控制且成本低。下面以 LCD1602 字符型液晶显示器为例，介绍其使用编程。

5.7.1　LCD1602 内部结构分析

LCD1602 内部主要由 LCD 显示屏(LCD Panel)、控制器 HD44780(Controller)、段码驱动器(Segment Driver)和背光源电路构成。图 5-24 为 LCD1602 的结构图。

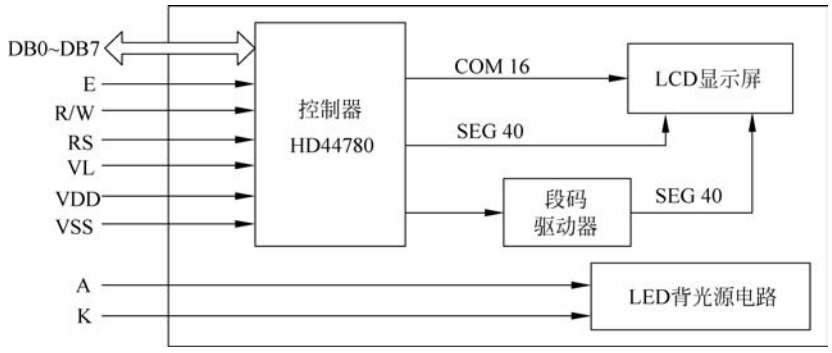

图 5-24　LCD1602 的结构图

控制器 HD44780 主要由指令寄存器 IR、数据寄存器 DR、忙标志 BF、地址计数器 AC、DDRAM(数据存储器)、CGROM(字符发生器)、CGRAM(用户自定义 RAM)以及时序发生电路组成。在控制器 HD44780 的控制下，液晶模块通过数据总线 DB0～DB7 和 RS、R/W、E 三个输入控制端与 FPGA 接口。三根控制线按照规定的时序相互协调作用，使控制器通过数据总线 DB 接收 FPGA 发送来的指令和数据，从 CGROM 中找到欲显示字符的字符码，送入 DDRAM，在 LCD 显示屏上与 DDRAM 存储单元对应的规定位置显示出该字符。DDRAM 地址与 LCD 显示屏上的显示位置对应关系如图 5-25 所示。

显示位置 →		1	2	3	…	14	15	16
DDRAM 地址	Line1	00H	01H	02H	…	0DH	0EH	0FH
	Line2	40H	41H	42H	…	4DH	4EH	4FH

图 5-25　DDRAM 地址与 LCD 显示屏上的显示位置对应关系

5.7.2　LCD1602 控制指令集

LCD1602 液晶模块的读写操作、屏幕和光标的操作都是通过指令编程实现的。LCD1602 液晶模块内部的控制器共有 11 条控制指令，如表 5-5 所示。

表 5-5　LCD1602 液晶模块控制指令

序号	指　　令	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	清显示	0	0	0	0	0	0	0	0	0	1
2	光标返回	0	0	0	0	0	0	0	0	1	*
3	置输入模式	0	0	0	0	0	0	0	1	I/D	S
4	显示开/关控制	0	0	0	0	0	0	1	D	C	B
5	光标或字符移位	0	0	0	0	0	1	S/C	R/L	*	*
6	置功能	0	0	0	0	1	DL	N	F	*	*
7	置字符发生存储器地址	0	0	0	1	字符发生存储器地址					
8	置数据存储器地址	0	0	1	显示数据存储器地址						
9	读忙标志或地址	0	1	BF	计数器地址						
10	写数到 CGRAM 或 DDRAM	1	0	要写的数据内容							
11	从 CGRAM 或 DDRAM 读数	1	1	读出的数据内容							

说明：1 为高电平，0 为低电平，表 5-5 中的 1 为各条指令的标志位，HD44780 通过判断指令从高位到低位第一个 1 出现的位置，识别指令功能。

指令 1：清显示。指令码 01H，光标复位到地址 00H 位置。

指令 2：光标复位。光标返回到地址 00H。

指令 3：光标和显示位置设置 I/D，光标移动方向，高电平右移，低电平左移。S：屏幕上所有文字是否左移或右移，高电平表示有效，低电平表示无效。

指令 4：显示开关控制。D：控制整体显示的开与关，高电平表示开显示，低电平表示关显示。C：控制光标的开与关，高电平表示有光标，低电平表示无光标。B：控制光标是否闪烁，高电平闪烁，低电平不闪烁。

指令 5：光标或显示移位。S/C：高电平时显示移动的文字，低电平时移动光标。

指令 6：功能设置命令。DL：高电平时为 4 位总线，低电平时为 8 位总线。N：低电平时为单行显示，高电平时为双行显示。F：低电平时显示 5×7 的点阵字符，高电平时显示 5×10 的显示字符。

指令 7：字符发生器 RAM 地址设置。

指令 8：DDRAM 地址设置。

指令 9：读忙信号和光标地址。BF：忙标志位，高电平表示忙，此时模块不能接收命令

或数据，如果为低电平则表示不忙。

读写基本操作时序见表 5-6。

表 5-6　读写基本操作时序

操　作	输　入	输　出
读状态	RS＝L，RW＝H，E＝H	DB0～DB7＝状态字
写指令	RS＝L，RW＝L，E＝下降沿脉冲，DB0～DB7＝指令码	无
读数据	RS＝H，RW＝H，E＝H	DB0～DB7＝数据
写数据	RS＝H，RW＝L，E＝下降沿脉冲，DB0～DB7＝数据	无

RS 为数据/命令选择端，RS 为高电平对应数据操作，RS 为低电平对应指令操作；R/W 为读写选择端，R/W 为高电平对应读操作，R/W 为低电平对应写操作；E 为使能信号，E 为高时读取信息使能，E 为下降沿执行写入指令或数据操作。读、写操作时序如图 5-26 及图 5-27 所示。

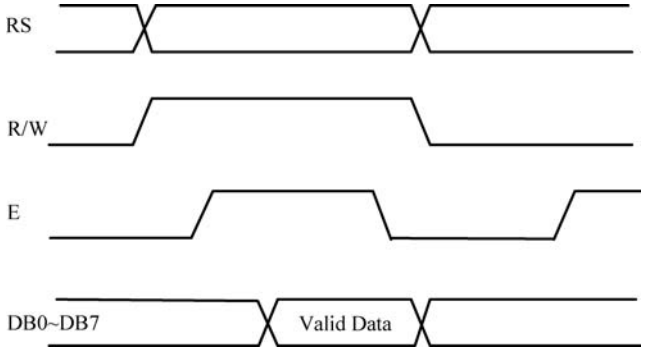

图 5-26　读操作时序

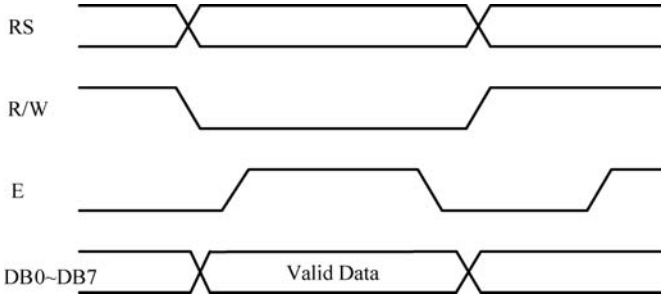

图 5-27　写操作时序

5.7.3　LCD1602 电气连接关系

液晶屏的电气连接如图 5-28 所示，各引脚定义如表 5-7 所示。电阻 R16 和 R17 改变偏压信号，使液晶显示有合适的对比度，也可接一个电位器，通过改变电位器的阻值来改变液晶显示的对比度；LCD_BLON 控制液晶屏背光的开关。

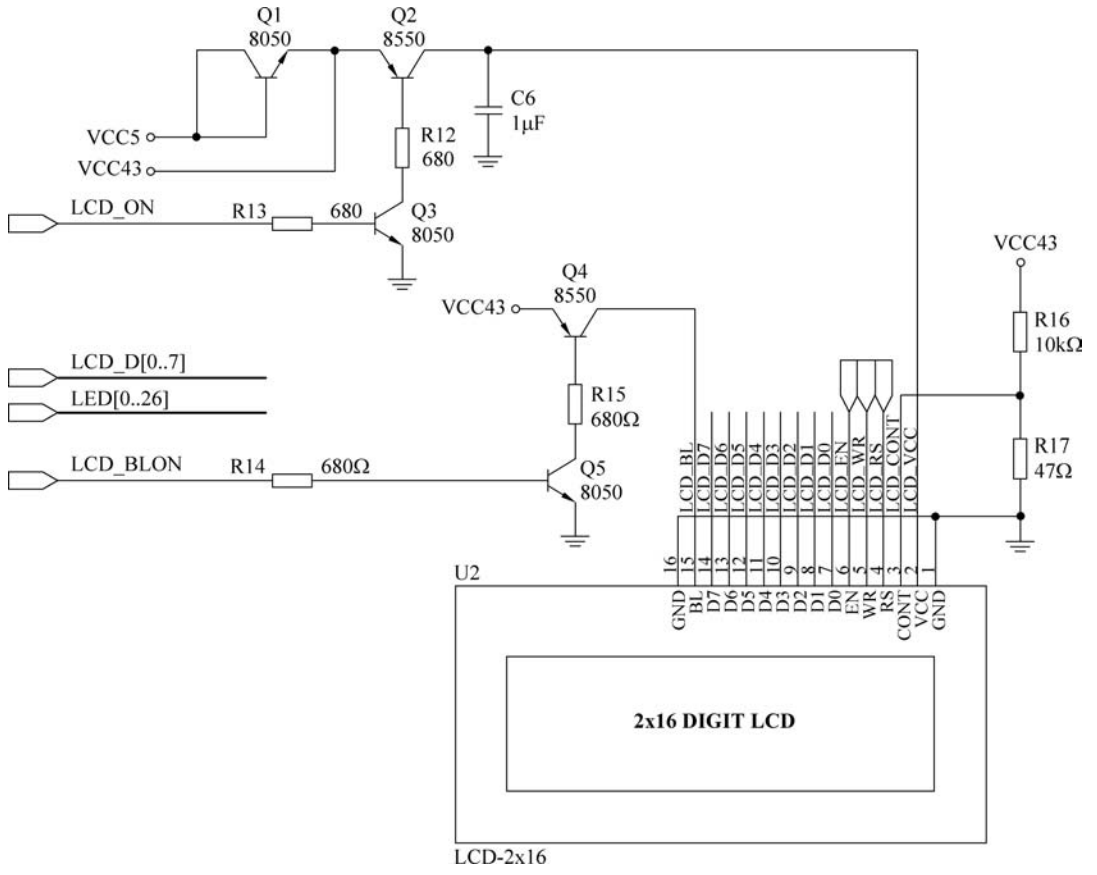

图 5-28　液晶屏的电气连接

表 5-7　字符型 LCD1602 各引脚定义

编号	符号	引 脚 说 明	编号	符号	引 脚 说 明
1	GND	电源地	9	D2	Data I/O
2	VCC	电源正极	10	D3	Data I/O
3	CONT	液晶显示偏压信号	11	D4	Data I/O
4	RS	数据/命令选择端(H/L)	12	D5	Data I/O
5	RW	读/写选择端(H/L)	13	D6	Data I/O
6	EN	使能信号	14	D7	Data I/O
7	D0	Data I/O	15	BL	背光源正极
8	D1	Data I/O	16	GND	电源地(背光源负极)

5.7.4　LCD1602 程序设计

LCD1602 显示控制器由两部分组成,一部分是用于存放待显示字符的 RAM,另一部分是驱动 LCD 的时序状态机。

根据 LCD1602 工作原理编写代码,生成的 LCD1602 驱动模块如图 5-29 所示。LCD 模块输入端口为时钟输入及系统复位信号；输出端口为 LCD1602 的控制线及数据线。

在 LCD1602 显示之前需要对其进行初始化设置。如：设置字符的格式，是一行显示还是两行显示，数据位宽是 8 位还是 4 位，显示点阵的大小；显示开关的控制；输入方式的控制；清屏；DDRAM 及 CGRAM 地址设置等。具体流程图如图 5-30 所示。

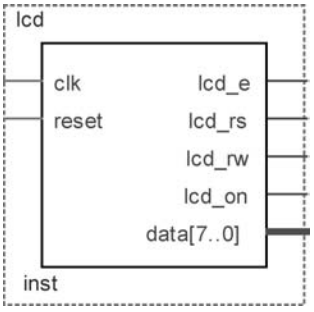

图 5-29　LCD 元件符号

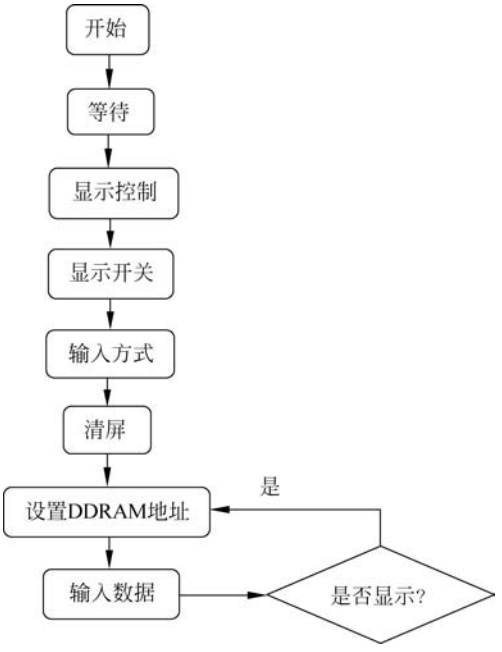

图 5-30　LCD1602 显示流程图

例 5-20：设计程序完成“welcome to the…”字符的显示。输入信号 clk 为 50MHz 时钟信号。

（1）LCD 驱动程序：

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity lcd is
entity lcd is
        port (clk : in std_logic;
            reset : in std_logic;
            lcd_e : buffer std_logic;
            lcd_rs : out std_logic;
            lcd_rw : out std_logic;
            lcd_on : out std_logic;
            data : out std_logic_vector(7 downto 0));
end lcd;

architecture behavioral of lcd is
constant idle: std_logic_vector(10 downto 0) := "00000000000";
constant clear: std_logic_vector(10 downto 0) := "00000000001";
```

```
constant returncursor : std_logic_vector(10 downto 0) : = "00000000010";
constant setmode : std_logic_vector(10 downto 0) : = "00000000100";
constant switchmode : std_logic_vector(10 downto 0) : = "00000001000";
constant shift : std_logic_vector(10 downto 0) : = "00000010000";
constant setfunction : std_logic_vector(10 downto 0) : = "00000100000";
constant setcgram : std_logic_vector(10 downto 0) : = "00001000000";
constant setddram : std_logic_vector(10 downto 0) : = "00010000000";
constant readflag : std_logic_vector(10 downto 0) : = "00100000000";
constant writeram : std_logic_vector(10 downto 0) : = "01000000000";
constant readram : std_logic_vector(10 downto 0) : = "10000000000";

constant increas_decreas : std_logic : = '1';
constant shift_noshift : std_logic : = '0';
constant xianshi : std_logic : = '1';
constant guangbiao : std_logic : = '0';
constant shanshuo : std_logic : = '0';
constant guang_hua_shift : std_logic : = '0';
constant shift_right_lift : std_logic : = '0';
constant datawidth_8_4 : std_logic : = '1';
constant font_5x10_5x7 : std_logic : = '0';
constant line_2_1 : std_logic : = '1';
constant divss : integer : = 16;
constant divcnt : std_logic_vector(20 downto 0): = "00011110100000010010000"; -- 250000
signal flag : std_logic;
signal clkdiv : std_logic;
signal tc_clkcnt : std_logic;
signal char_addr : std_logic_vector(5 downto 0);
signal data_in : std_logic_vector(7 downto 0);
signal clkcnt : std_logic_vector(20 downto 0);
signal state : std_logic_vector(10 downto 0);
signal counter : integer range 0 to 157;
signal div_counter : integer range 0 to 15;

component char_ram
          port (address : in std_logic_vector(5 downto 0);
                data : out std_logic_vector(7 downto 0));
end component;

begin
lcd_on < = '1';
tc_clkcnt < = '1' when clkcnt = divcnt else '0';

process(clk,reset)
begin
    if(reset = '0')then
       clkcnt < = "00000000000000000000";
    elsif(clk'event and clk = '1')
          then
              if(clkcnt = divcnt)
                    then
```

```
                clkcnt <= "00000000000000000000";
            else
                clkcnt <= clkcnt + 1;
            end if;
    end if;
end process;

process(tc_clkcnt,reset)
begin
     if(reset = '0')
          then clkdiv <= '0';
     elsif( tc_clkcnt'event and tc_clkcnt = '1')
          then clkdiv <= not clkdiv;
     end if;
end process;
lcd_e <= clkdiv;

aa:char_ram
port map(address => char_addr,
         data => data_in);

lcd_rs <= '1'
    when state = writeram or state = readram else '0';
lcd_rw <= '0'
    when state = clear or state = returncursor or state = setmode or state = switchmode or state
 = shift or state = setfunction or state = setcgram or state = setddram or state = writeram
else '1';
data <= "00000001" when state = clear else      -- 清屏
    "00000010" when state = returncursor else   -- 归位
    "000001" & increas_decreas & shift_noshift when state = setmode else
                                                -- 输入方式设置,ac 自动增 1,画面不动
    "00001" & xianshi & guangbiao & shanshuo when state = switchmode else
                                                -- 显示开关设置,显示开,光标和闪烁关
    "0001" & guang_hua_shift & shift_right_lift &"00"  when state = shift else
                                                -- 光标画面设置光标平移一个字符,左移
    "001" & datawidth_8_4 & line_2_1 & font_5x10_5x7 & "00" when state = setfunction
                                                -- 功能设置,8 位数据,两行显示,5×7 点阵
    else "01000000" when state = setcgram          -- 设置 cgram 地址
    else "10000000" when state = setddram and counter < 40 -- 设置 ddram 第一行首地址
    else "11000000" when state = setddram and counter >= 40 and counter < 81
                                                -- 设置 ddram 第二行首地址
    else data_in when state = writeram
    else "ZZZZZZZZ";

char_addr <= conv_std_logic_vector(counter,6)
         when state = writeram and counter < 40
             else conv_std_logic_vector( counter - 41 + 16,6)
when state = writeram and counter > 40 and counter < 81
```

```
              else "000000";

process(clkdiv,reset)                    -- 状态机程序
  begin
      if(reset = '0')then                -- reset 为复位信号,复位后重新刷新显示内容
              state <= idle;             -- 复位后,状态回归到等待
              counter <= 0;              -- 计数清零
              flag <= '0';               -- 标志清零
              div_counter <= 0;          -- 分频计数器清零
     elsif(clkdiv'event and clkdiv = '1')then
     case state is
     when idle =>
       if(flag = '0')then
          state <= setfunction; -- flag = '0' -- 表示未完成功能设置,进入工作方式设置状态
                   flag <= '1';
                   counter <= 0;
                   div_counter <= 0;
         else
           if(div_counter < divss )then    -- flag = '1'表示功能设置已经完成, 进入等待状态
                            div_counter <= div_counter + 1;
                            state <= idle;
            else
                            div_counter <= 0;
                            state <= shift;
             end if;
        end if;
         when setfunction =>
              state <= switchmode;
         when switchmode =>
              state <= setmode;
         when setmode =>
              state <= clear;
         when clear =>
              state <= setddram;
         when shift =>
              state <= idle;
         when setddram =>
              state <= writeram;
         when readflag =>
              state <= idle;
         when writeram =>
                   if(counter = 40)then   -- 完成第一行显示后,要写第二行初始地址
                             state <= setddram;
                             counter <= counter + 1;
                       elsif(counter < 81)then
                             state <= writeram;
                             counter <= counter + 1;
                       else
                            state <= setddram;
                            counter <= 0;
```

```
                            state <= shift;
                        end if;
            when readram => 
                state <= idle;
            when others => 
                state <= idle;
    end case;
  end if;
end process;
end behavioral;
```

图 5-31 给出了 LCD 显示驱动程序状态转换图。

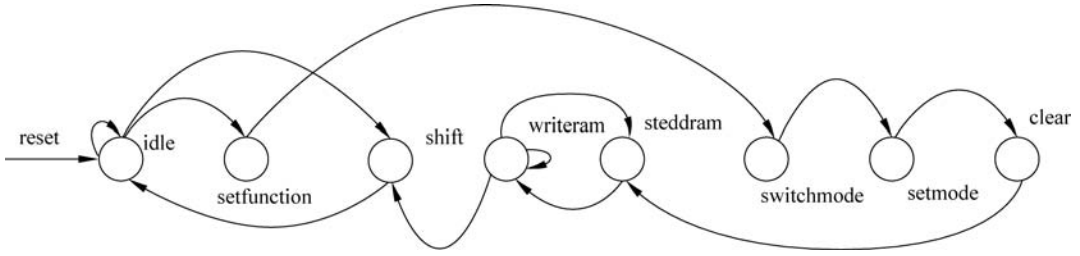

图 5-31　LCD 显示控制状态转换图

状态转换分析：初始时处于等待状态，判断若没有完成刷新任务(flag＝'0')，则首先向 LCD 写设置命令，完成工作方式设置位、显示模式设置、输入方式设置、清屏指令。写完设置命令后，设置光标地址为第一行首地址，开始写入数据，写完第一行 16 个数据后，要设置地址为第二行首地址继续写入第二行 16 个字符，继而进入等待状态，等复位信号到来后，重复上面的过程，刷新液晶屏显示的信息。

(2) 获取字符数据存储器电路：

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity char_ram is
port(address : in std_logic_vector(5 downto 0);
       data : out std_logic_vector(7 downto 0));
end char_ram;
architecture ZXY of char_ram is
function char_to_integer (indata : character) return integer is
variable result : integer range 0 to 16＃7F＃;
begin
    case indata is
    when ' ' => result := 32;
    when '!' => result := 33;
    when '"' => result := 34;
    when '＃' => result := 35;
```

```
        when '$' => result := 36;
        when '%' => result := 37;
        when '&' => result := 38;
        when ''' => result := 39;
        when '(' => result := 40;
        when ')' => result := 41;
        when '*' => result := 42;
        when '+' => result := 43;
        when ',' => result := 44;
        when '-' => result := 45;
        when '.' => result := 46;
        when '/' => result := 47;
        when '0' => result := 48;
        when '1' => result := 49;
        when '2' => result := 50;
        when '3' => result := 51;
        when '4' => result := 52;
        when '5' => result := 53;
        when '6' => result := 54;
        when '7' => result := 55;
        when '8' => result := 56;
        when '9' => result := 57;
        when ':' => result := 58;
        when ';' => result := 59;
        when '<' => result := 60;
        when '=' => result := 61;
        when '>' => result := 62;
        when '?' => result := 63;
        when '@' => result := 64;
        when 'A' => result := 65;
        when 'B' => result := 66;
        when 'C' => result := 67;
        when 'D' => result := 68;
        when 'E' => result := 69;
        when 'F' => result := 70;
        when 'G' => result := 71;
        when 'H' => result := 72;
        when 'I' => result := 73;
        when 'J' => result := 74;
        when 'K' => result := 75;
        when 'L' => result := 76;
        when 'M' => result := 77;
        when 'N' => result := 78;
        when 'O' => result := 79;
        when 'P' => result := 80;
        when 'Q' => result := 81;
        when 'R' => result := 82;
        when 'S' => result := 83;
        when 'T' => result := 84;
        when 'U' => result := 85;
```

```
        when 'V' => result := 86;
        when 'W' => result := 87;
        when 'X' => result := 88;
        when 'Y' => result := 89;
        when 'Z' => result := 90;
        when '[' => result := 91;
        when '\' => result := 92;
        when ']' => result := 93;
        when '^' => result := 94;
        when '_' => result := 95;
        when '~' => result := 96;
        when 'a' => result := 97;
        when 'b' => result := 98;
        when 'c' => result := 99;
        when 'd' => result := 100;
        when 'e' => result := 101;
        when 'f' => result := 102;
        when 'g' => result := 103;
        when 'h' => result := 104;
        when 'i' => result := 105;
        when 'j' => result := 106;
        when 'k' => result := 107;
        when 'l' => result := 108;
        when 'm' => result := 109;
        when 'n' => result := 110;
        when 'o' => result := 111;
        when 'p' => result := 112;
        when 'q' => result := 113;
        when 'r' => result := 114;
        when 's' => result := 115;
        when 't' => result := 116;
        when 'u' => result := 117;
        when 'v' => result := 118;
        when 'w' => result := 119;
        when 'x' => result := 120;
        when 'y' => result := 121;
        when 'z' => result := 122;
        when '{' => result := 123;
        when '|' => result := 124;
        when '}' => result := 125;
        when '～' => result := 126;
        when others => result := 32;
        end case;
        return result;
    end function;
    begin
    process (address)
    begin
        case address is
        when "000001" => data <= conv_std_logic_vector(char_to_integer ('W') ,8);
```

```
        when "000010" => data <= conv_std_logic_vector(char_to_integer ('e') ,8);
        when "000011" => data <= conv_std_logic_vector(char_to_integer ('l') ,8);
        when "000100" => data <= conv_std_logic_vector(char_to_integer ('c') ,8);
        when "000101" => data <= conv_std_logic_vector(char_to_integer ('o') ,8);
        when "000110" => data <= conv_std_logic_vector(char_to_integer ('m') ,8);
        when "000111" => data <= conv_std_logic_vector(char_to_integer ('e') ,8);
        when "001000" => data <= conv_std_logic_vector(char_to_integer (' ') ,8);
        when "001001" => data <= conv_std_logic_vector(char_to_integer ('t') ,8);
        when "001010" => data <= conv_std_logic_vector(char_to_integer ('o') ,8);
        when "001011" => data <= conv_std_logic_vector(char_to_integer (' ') ,8);
        when "001100" => data <= conv_std_logic_vector(char_to_integer ('t') ,8);
        when "001101" => data <= conv_std_logic_vector(char_to_integer ('h') ,8);
        when "001110" => data <= conv_std_logic_vector(char_to_integer ('e') ,8);
        when "001111" => data <= conv_std_logic_vector(char_to_integer (' ') ,8);
        when "010000" => data <= conv_std_logic_vector(char_to_integer ('I') ,8);
        when "010001" => data <= conv_std_logic_vector(char_to_integer ('n') ,8);
        when "010010" => data <= conv_std_logic_vector(char_to_integer ('t') ,8);
        when "010011" => data <= conv_std_logic_vector(char_to_integer ('e') ,8);
        when "010100" => data <= conv_std_logic_vector(char_to_integer ('l') ,8);
        when "010101" => data <= conv_std_logic_vector(char_to_integer (' ') ,8);
        when "010110" => data <= conv_std_logic_vector(char_to_integer ('D') ,8);
        when "010111" => data <= conv_std_logic_vector(char_to_integer ('E') ,8);
        when "011000" => data <= conv_std_logic_vector(char_to_integer ('2') ,8);
        when "011001" => data <= conv_std_logic_vector(char_to_integer (' ') ,8);
        when "011010" => data <= conv_std_logic_vector(char_to_integer ('B') ,8);
        when "011011" => data <= conv_std_logic_vector(char_to_integer ('o') ,8);
        when "011100" => data <= conv_std_logic_vector(char_to_integer ('a') ,8);
        when "011101" => data <= conv_std_logic_vector(char_to_integer ('r') ,8);
        when "011110" => data <= conv_std_logic_vector(char_to_integer ('d') ,8);
        when others => data <= conv_std_logic_vector(char_to_integer (' ') ,8);
        end case;
end process;
end ZXY;
```

其仿真图如图 5-32 所示。

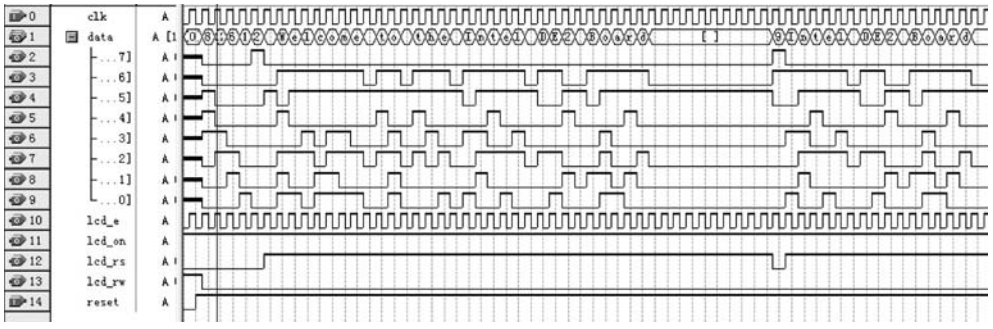

图 5-32　LCD 显示仿真图

显示效果图如图 5-33 所示。

图 5-33　显示效果图