第 5 章 CPLD/FPGA 应用实践

CHAPTER 5

5.1 常用组合逻辑电路的描述

常用的基本数字电路模块是数字系统中不可缺少的基本组成部分。数字逻辑电路可分为两类，一类逻辑电路的输出只与当时输入的逻辑值有关，而与输入的历史情况无关，这种逻辑电路称为组合逻辑电路(Combinational Logic Circuit)；另一类逻辑电路的输出不仅与电路当时输入的逻辑值有关，而且与电路以前输入过的逻辑值有关，这种逻辑电路称为时序逻辑电路(Sequential Logic Circuit)。

5.1.1 非门电路的设计

1. 模型

根据 VHDL 的特点，对非门电路进行直接描述，其 VHDI 模型是非门的逻辑符号，如图 5.1 所示。其布尔代数模型为 $b=\bar{a}$。用 VHDL 描述为 b<= NOT a。

图 5.1 非门电路

2. 程序设计

按照 VHDL 的结构特点，首先要为其确定一个实体，然后确定一个结构体。为了方便，取实体名为 not_gate，取结构体名为 behav。由于是门级的 VHDL 描述，直接给出 VHDL 文件。

【例 5.1】 非门的 VHDL 描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY not_gate IS
    PORT(a:IN STD_LOGIC;
         b:OUT STD_LOGIC);          --定义输入端口 a 和输出端口 b
END not_gate;
ARCHITECTURE behav OF not_gate IS
    BEGIN
      b<=NOT a;                     --逻辑"非"描述
END ARCHITECTURE behav;
```

3. 仿真验证

仿真验证的步骤如下：

(1) 编译源文件的语法的正确性。

(2) 根据设计功能,编写测试文件。

(3) 进行功能仿真。根据上面的源程序和测试文件,调用 Modelsim_Altera 平台运行,得到功能仿真的验证结果如图 5.2 所示。

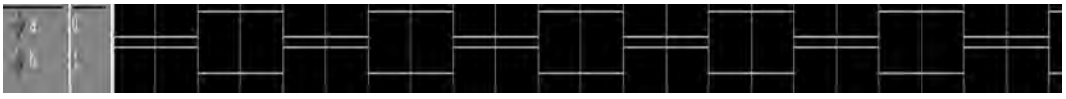

图 5.2 非门电路的功能仿真波形图

从图 5.2 可以看出非门的 VHDL 描述是正确的。

(4) 时间仿真。选择 CPLD 或 FPGA 作为承载器件,再次编译源文件时注意采用全编译方式,再次调用仿真结果如图 5.3 所示。从图中可以看出明显的延时信息,但逻辑关系是正确的。

图 5.3 非门电路的时间仿真结果

5.1.2 其他基本门电路的设计

按照非门电路设计的方法和步骤,我们很容易用 VHDL 描述其他门的电路。在此只给出各种门电路的 VHDL 文件,其他步骤读者可以自己进行。

【例 5.2】 与门的 VHDL 描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY and_gate IS
PORT(a,b:IN STD_LOGIC;
       c:OUT STD_LOGIC);
END and_gate;
ARCHITECTURE behav OF and_gate IS
BEGIN
    c <= a AND b;                    -- 逻辑"与"描述
END ARCHITECTURE behav;
```

【例 5.3】 与非门的 VHDL 描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nand_gate IS
  PORT(a,b:IN STD_LOGIC;
         c:OUT STD_LOGIC);
END nand_gate;
ARCHITECTURE behav OF nand_gate IS
  BEGIN
    c <= a NAND b;
END ARCHITECTURE behav;
```

【例 5.4】 或非门的 VHDL 描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nor_gate IS
  PORT(a,b:IN STD_LOGIC;
       c:OUT STD_LOGIC);
END nor_gate;
ARCHITECTURE behav OF nor_gate IS
  BEGIN
     c <= NOT(a OR b);
END ARCHITECTURE behav;
```

【例 5.5】 异或非门的 VHDL 描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nxor_gate IS
   PORT(a,b:IN STD_LOGIC;
        c:OUT STD_LOGIC);
END nxor_gate;
ARCHITECTURE behav OF nxor_gate IS
    BEGIN
        c <= NOT(a XOR b);
END ARCHITECTURE behav;
```

【例 5.6】 三态门的 VHDL 描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY tri_gate IS
    PORT(a,ena:IN STD_LOGIC;
         b:OUT STD_LOGIC);
END tri_gate;
ARCHITECTURE behav OF tri_gate IS
    BEGIN
        b <= a WHEN ena = '1' ELSE
            'Z';
END ARCHITECTURE behav;
```

三态门仿真结果如图 5.4 所示，当使能信号 ena 为低电平时，输出 b 为高阻状态；当使能信号 ena 为高电平时，输出 b 与输入 a 状态相同。

图 5.4 三态门仿真结果

5.2 基本时序逻辑电路的 VHDL 描述

5.2.1 D 触发器的设计

1. 电路模型

D 触发器的符号模型如图 5.5 所示。clr 为异步复位信号，当 clr 为高电平时，输出 Q 为低电平与时钟边沿无关；当复位信号为低电平且时钟的上升沿到来时，输出 Q 状态与输入 d 电平一致；当时钟的上升沿未到来时，则等待它的到来状态保持不变，直到时钟信号上升沿到来为止。

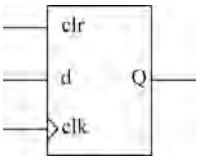

图 5.5 D 触发器的符号模型

2. 程序设计

【例 5.7】 异步复位的 D 触发器的 VHDL 描述。

```
library ieee;
use ieee.std_logic_1164.all;
entity dffa is
    port(D,clk,clr: in std_logic; Q: out std_logic);          -- 定义输入/输出端口
end entity dffa;
architecture behave of dffa is
    begin
        process(clk,D,clr)                                    -- 进程敏感信号
        begin
            if  clr = '1'  then
                Q <= '0';
            Elsif clk'event and clk = '1'then                 -- 检测时钟上升沿
                Q <= D;
        end if:
    end process;
end architecture behave;
```

3. 仿真结果

功能仿真波形图如图 5.6 所示。

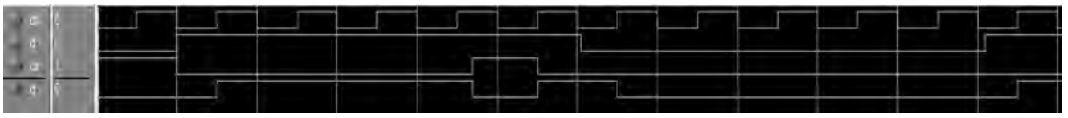

图 5.6 D 触发器的功能仿真波形图

5.2.2 T 触发器的设计

T 触发器逻辑符号如图 5.7 所示。图中 clr_n 为同步复位信号低电平有效，pr_n 为异步置位信号。当 pr_n 为低电平时输出 Q 为高电平不需 clk 的上升沿触发，当 pr_n 为高电平时正常按照 T 触发器功能执行。每一个 clk 上升沿到来时首先判断 clr_n 状态，当 clr_n 为低电平时输出 Q 为低电平执行同步复位功能；当 clr_n 为高电平时，输出 Q 为上一状态的 Q 与当前输入 t 信号电平做异或输出至当前状态。

【例 5.8】 异步置位同步复位的 T 触发器的 VHDL 描述。

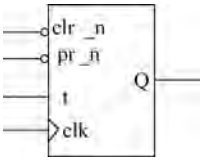

图 5.7 T 触发器逻辑符号

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY t_ff IS
PORT(clr_n,pr_n,t,clk:IN STD_LOGIC;
                Q:BUFFER STD_LOGIC);
                 -- 输出端口 Q 类型定义为 buffer,因为后续进程中需要读到该端口状态
END t_ff;
ARCHITECTURE behav OF t_ff IS
  BEGIN
      PROCESS(clk,t,clr_n,pr_n)                          -- 进程敏感信号
         BEGIN
             IF  pr_n = '0'  THEN
                 Q <= '1';
             ElSIF clk'EVENT AND clk = '1'THEN           -- 检测时钟上升沿
                 IF  clr_n = '0'THEN
                     Q <= '0';
                 ELSIF t = '1' THEN
                     Q <= t XOR Q;
                 END IF;
             END IF;
       END PROCESS;
END behav;
```

带有同步复位异步置位功能的 T 触发器的功能仿真波形图如图 5.8 所示。

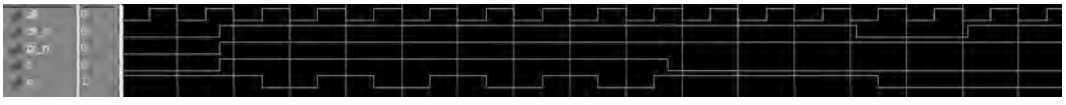

图 5.8 T 触发器的功能仿真波形图

5.2.3 JK 触发器的设计

【例 5.9】 基本 JK 触发器的 VHDL 描述(见图 5.9)。

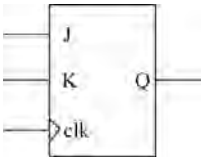

图 5.9 JK 触发器的 VHDL 描述

```
library ieee;
use ieee.std_logic_1164.all;
  entity jk is
  port(J,K,clk: in std_logic; Q: buffer std_logic);
       -- 定义输入/输出端口
  end entity jk;
architecture behave of jk is
  begin
    process(clk,J,K)                                     -- 进程敏感信号
     begin
      if clk'event and clk = '1'then                     -- 检测时钟上升沿
      Q <= ((J and (not Q ))or ((not K)and Q));
```

```
        end if;
    end process;
end architecture behave;
```

基本 JK 触发器的功能仿真波形图如图 5.10 所示。

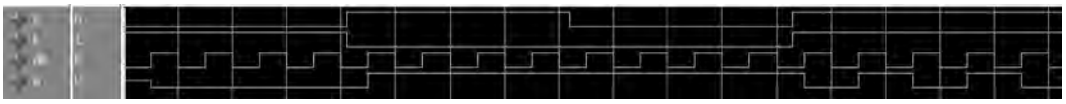

图 5.10 JK 触发器的功能仿真波形图

5.2.4 串行移位寄存器的设计

移位寄存器在微处理器的算术或逻辑运算中是一个常用的组件,移位方式有向左移位和向右移位。例 5.10 是一个双向串行移位寄存器,引入控制信号 dir 进行移位方向控制。

【例 5.10】 双向串行移位寄存器的 VHDL 描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift_reg IS
PORT(clk,din,dir:IN STD_LOGIC;
                 Q:OUT STD_LOGIC);
END shift_reg;

ARCHITECTURE behav OF shift_reg IS
    SIGNAL data:STD_LOGIC_VECTOR(7 DOWNTO 0):="00000000";--赋初值仿真使用
    BEGIN
    PROCESS(clk,dir,din)
        BEGIN
        IF clk'EVENT AND clk='1' THEN
            IF dir='0' THEN                                    --左移
                data(0)<=din;
                FOR i IN 1 TO 7 LOOP
                    data(i)<=data(i-1);
                END LOOP;
            ELSE                                               --右移
                data(7)<=din;
                FOR i IN 1 TO 7 LOOP
                      data(i-1)<=data(i);
                END LOOP;
            END IF;
        END IF;
    END PROCESS;
    Q<=data(7) WHEN dir='0' ELSE                               --移位输出
        data(0);
END behav;
```

串行移位寄存器的仿真结果如图 5.11 所示。

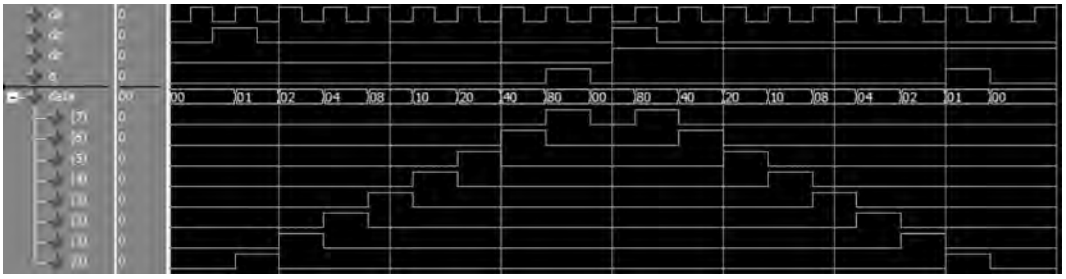

图 5.11　串行移位寄存器的仿真波形

5.2.5　分频电路的设计

计数器其实是一种分频电路，可以利用计数器实现偶数分频、奇数分频器、小数分频等，并且分频信号的占空比可以在一定范围内调整。

【例 5.11】　50%占空比的 10 分频器。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY clk_div10 IS
    PORT(   clk:IN STD_LOGIC;
        q:OUT STD_LOGIC);
END clk_div10;
ARCHITECTURE behave OF clk_div10 IS
    SIGNAL temp:INTEGER RANGE 0 TO 15:=0;
    BEGIN
    PROCESS(clk)
        BEGIN
          IF(clk'EVENT AND clk='1')THEN
              IF (temp=9) THEN          --根据分频系数设计计数器范围 N
                  temp<=0;
              Else
                  temp<=temp+1;
              END IF;
          END IF;
    END PROCESS;
    q<='1' WHEN temp>4 ELSE             --根据占空比调整高电平出现的时刻
         '0';
END behave;
```

仿真结果如图 5.12 所示。如果需要设计的是偶数分频 N 的分频器，须将计数器范围调整为 $N-1$。

图 5.12　50%占空比的 10 分频器的仿真波形

【例 5.12】 50%占空比的 9 分频器。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY clk_div9 IS
    PORT(  clk:IN STD_LOGIC;
        q:OUT STD_LOGIC);
END clk_div9;
ARCHITECTURE behav OF clk_div9 IS
    SIGNAL temp1,temp2:INTEGER RANGE 0 TO 15:=0;
    SIGNAL q1,q2:STD_LOGIC:='0';
    BEGIN
    PROCESS(clk)                          --上升沿计数器,根据分频系数设计计数器范围 N
        BEGIN
          IF(clk'EVENT AND clk='1')THEN
              IF (temp1=8) THEN
                  temp1<=0;
              Else
                  temp1<=temp1+1;
              END IF;
          END IF;
    END PROCESS;
    PROCESS(clk)                          --下升沿计数器,根据分频系数设计计数器范围 N
        BEGIN
          IF(clk'EVENT AND clk='0')THEN
              IF (temp2=8) THEN
                  temp2<=0;
              Else
                  temp2<=temp2+1;
              END IF;
          END IF;
    END PROCESS;
    q1<='1' WHEN temp1>4 ELSE '0';      --根据占空比调整低电平出现的时刻
    q2<='1' WHEN temp2>4 ELSE '0';      --根据占空比调整高电平出现的时刻
    q<=q1 OR q2;
END behav;
```

仿真结果如图 5.13 所示。如果需要设计的是奇数分频 N 的分频器,须将两个计数器范围调整为 $N-1$。

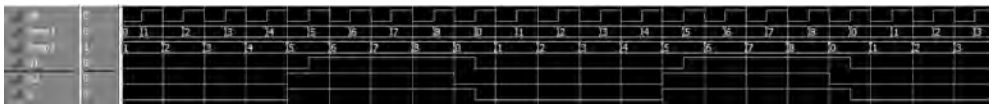

图 5.13 50%占空比的 9 分频器的仿真波形

5.3 常用算法 VHDL 实现

5.3.1 流水线加法器的设计

流水线规则可以应用在 FPGA 设计中,只需要极少或者根本不需要额外的成本,因为每一个逻辑元件都包括一个触发器。采用流水线有可能将一个算术操作分解成一些小规模

的基本操作，将进位和中间值存储在寄存器中，并在下一个时钟周期内继续计算。但是具体将加法器分成多少部分应视具体硬件结构而定。针对 Cyclone 器件，一个合理的选择就是采用一个带有 16 个 LE 的 LAB 组成一个流水线。例 5.13 给出了一个针对 Cyclone 器件的 15 位流水加法器的代码。

【例 5.13】 15 位流水加法器的 VHDL 设计。

```
LIBRARY LPM;
USE LPM.LPM_COMPONENTS.ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
ENTITY pipeline_adder IS
    GENERIC(WIDTH:INTEGER:=15;        --求位宽
            width_l:INTEGER:=7;       --低七位
            width_h:INTEGER:=8;       --高八位
            co:INTEGER:=1             --进位位
            );
    PORT(a,b:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        clk:IN STD_LOGIC;
        sum:OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0)
    );
END pipeline_adder;
ARCHITECTURE behav OF pipeline_adder IS
    SIGNAL a_l,b_l,resul_l,temp_l:STD_LOGIC_VECTOR(width_l-1 DOWNTO 0);
    SIGNAL a_h,b_h,resul_h,temp_h,temp_h1,temp_c:STD_LOGIC_VECTOR(width_h-1 DOWNTO 0);
    SIGNAL temp_sum:STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);          --输出寄存
    SIGNAL temp_co1,temp_co11:STD_LOGIC_VECTOR(co-1 DOWNTO 0);   --LSB进位寄存
BEGIN
    PROCESS(clk)                                                 --分解输入
        BEGIN
          IF clk'EVENT AND clk='1' THEN
              FOR k IN width_l-1 DOWNTO 0 LOOP
                  a_l(k)<=a(k);
                  b_l(k)<=b(k);
              END LOOP;
              FOR k IN width_h-1 DOWNTO 0 LOOP
                  a_h(k)<=a(k+width_l);
                  b_h(k)<=b(k+width_l);
              END LOOP;
          END IF;
    END PROCESS;
add_1:lpm_add_sub
    GENERIC MAP(lpm_width=>width_l,
            lpm_representation=>"UNSIGNED",
            lpm_direction=>"ADD")
    PORT MAP(dataa=>a_l,
            datab=>b_l,
            result=>resul_l,
            cout=>temp_co1(0));
add_2:lpm_add_sub                                                --MSB求和
```

```
    GENERIC MAP(lpm_width = > width_h,
            lpm_representation = >"UNSIGNED",
            lpm_direction = >"ADD")
    PORT MAP(dataa = > a_h,datab = > b_h,result = > resul_h);
reg_1:lpm_ff
    GENERIC MAP(lpm_width = > width_l)
    PORT MAP(data = > resul_l,q = > temp_l,clock = > clk);
reg_2:lpm_ff
    GENERIC MAP(lpm_width = > co)
    PORT MAP(data = > temp_co1,q = > temp_co11,clock = > clk);
reg_3:lpm_ff
    GENERIC MAP(lpm_width = > width_h)
    PORT MAP(data = > resul_h,q = > temp_h,clock = > clk);
    temp_c < = (others = >'0');
add_3:lpm_add_sub                                    -- MSB 求和结果与 LSB 进位相加
    GENERIC MAP(lpm_width = > width_h,
            lpm_representation = >"UNSIGNED",
            lpm_direction = >"ADD")
    PORT MAP(cin = > temp_co11(0),dataa = > temp_h,datab = > temp_c,result = > temp_h1);
    PROCESS                                                    -- 输出寄存
        BEGIN
            WAIT UNTIL clk = '1';
            FOR k IN width_l - 1 DOWNTO 0 LOOP
                temp_sum(k)< = temp_l(k);
            END LOOP;
            FOR k IN width_h - 1 DOWNTO 0 LOOP
                temp_sum(k + width_l)< = temp_h1(k);
            END LOOP;
    END PROCESS;
    sum < = temp_sum;
end behav;
```

流水线加法器的仿真结果如图 5.14 所示。

图 5.14 流水线加法器的仿真结果

5.3.2 8 位乘法器的设计

两个 N 位二进制数的乘积用 X 和 $A=\sum_{k=0}^{N-1}a_k 2^k$ 表示，按“手工计算”的方法给出就是：$P=A\times X=\sum_{k=0}^{N-1}a_k 2^k X$，从中可以看出，乘法的完成实际上是一个移位相加的过程，例 5.14 就采用该方法来实现两个 8 位整数相乘。乘法分为三个步骤：首先下载 8 位操作数并且重置移位寄存器；其次进行串并乘法运算；最后乘积结果输出到寄存器。

【例 5.14】 8 位乘法器的 VHDL 设计。

```
PACKAGE def_data_type IS                                          -- 自定义程序包
    SUBTYPE byte IS INTEGER RANGE 0 TO 127;
    SUBTYPE word IS INTEGER RANGE 0 TO 32767;
    TYPE state_type IS (s0,s1,s2);
END def_data_type;
LIBRARY WORK;                                                     -- 声明工作库
USE WORK.def_data_type.all;                                       -- 声明自定义程序包
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
ENTITY mul_8 IS
    PORT(clk:std_logic;
         x:in byte;
         y:in std_logic_vector(7 downto 0);
         z:out word);
END mul_8;
ARCHITECTURE behav OF mul_8 IS
    SIGNAL state:state_type;
    BEGIN
    PROCESS(clk)
        VARIABLE p,t:word:=0;
        VARIABLE c:INTEGER RANGE 0 TO 7;
        BEGIN
            IF clk'EVENT AND clk='1' THEN
                CASE state IS
                    WHEN s0=>state<=s1;c:=0;p:=0;t:=x;            -- 初始化
                    WHEN s1=>IF c=7 THEN state<=s2;               -- 移位相加
                                ELSE
                                    IF y(c)='1' THEN
                                        p:=p+t;
                                    END IF;
                                    t:=t*2;
                                    c:=c+1;
                                    state<=s1;
                                END IF;
                    WHEN s2=>z<=p;state<=s0;                      -- 结果寄存
                END CASE;
            END IF;
    END PROCESS;
END behav;
```

8 位乘法器的仿真结果如图 5.15 所示。

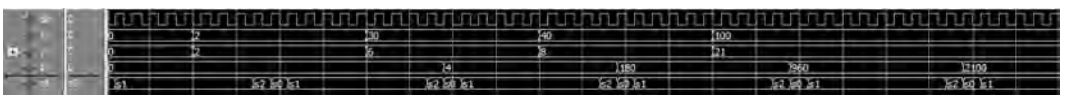

图 5.15　8 位乘法器的仿真结果

5.3.3 4抽头直接FIR滤波器的设计

FIR滤波器的结构主要是非递归结构，没有输出到输入的反馈，并且FIR滤波器很容易获得严格的线性相位特性，避免被处理信号产生相位失真。而线性相位体现在时域中仅仅是$h(n)$在时间上的延迟，这个特点在图像信号处理、数据传输等波形传递系统中是非常重要的。相位响应可以使线性系统绝对稳定，而且设计相对容易、高效。由于FIR滤波器没有反馈回路，因此它是无条件稳定系统，其单位冲激响应$h(n)$是一个有限长序列。由图5.16可见，FIR滤波器实际上是一种乘法累加运算，它不断地输入样本$x(n)$，经延时(Z^{-1})，做乘法累加，再输出滤波结果$y(n)$。其差分表达式为：$y(n)=\sum_{i=0}^{N-1}a_i x(n-i)$。图5.16给出$N$阶LTI型FIR滤波器的图解。

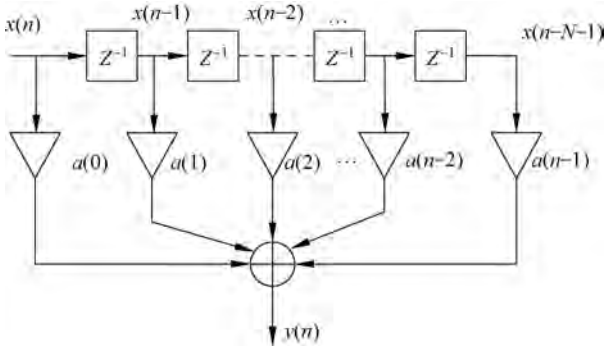

图5.16 FIR滤波器的结构图

例5.15是对图5.16中直接FIR滤波器结构的文字解释，这种设计对对称和非对称滤波器都是适用的。抽头延迟线每个抽头的输出分别乘以相应的加权二进制值，再将结果相加。以系数为{－1，3.75，3.75，－1}为例，设计相应滤波器如下。

【例5.15】 4抽头直接FIR滤波器的VHDL描述。

```
PACKAGE def_data_type IS
    SUBTYPE byte IS INTEGER RANGE -128 TO 127;
    TYPE array_byte IS ARRAY(0 TO 3) OF byte;
END def_data_type;
LIBRARY WORK;
USE WORK.def_data_type.all;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
ENTITY FIR_4 IS
    PORT(clk:std_logic;
         x:in byte;
         y:out byte);
END FIR_4;
ARCHITECTURE behav OF FIR_4 IS
    SIGNAL tap:array_byte:=(0,0,0,0);
    BEGIN
```

```
    PROCESS(clk)
        BEGIN
            IF clk'EVENT AND clk = '1' THEN
                y <= 2 * tap(1) + tap(1) + tap(1)/2 + tap(1)/4
                     + 2 * tap(2) + tap(2) + tap(2)/2 + tap(2)/4
                     - tap(3) - tap(0);
                FOR i IN 3 DOWNTO 1 LOOP
                    tap(i)<= tap(i - 1);
                END LOOP;
                tap(0)<= x;
            END IF;
    END PROCESS;
END behav;
```

FIR 滤波器的仿真结果如图 5.17 所示。

图 5.17　FIR 滤波器的仿真结果

5.3.4　IIR 数字滤波器的设计

IIR 滤波器比 FIR 滤波器获得更高的性能，它具有工作速度快、耗用存储空间少的特点。IIR 数字滤波器需要执行无限数量卷积，能得到较好的幅度特性，其相位特性是非线性，具有无限持续时间冲激响应。但是由于相位非线形的缺点，使其无法在图像处理以及数据传输中得到应用。

1. 有耗积分器 I

滤波器的一个基本功能就是使有干扰的信号平滑。假定信号 $x[n]$是以含有宽频带零平均值随机噪声的形式接收到的。从数学的角度讲，可以采用积分器来消除噪声的影响。如果输入信号的平均值能够保持的时间间隔是有限长，就可以采用有耗积分器处理含有额外噪声的信号。图 5.18 显示了一个简单的一阶有耗积分器，它满足离散时间差分方程：

$$y[n+1]=3/4y[n]+x[n]$$

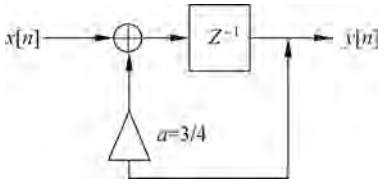

图 5.18　有耗积分器的一阶 IIR 滤波器

【例 5.16】 IIR 滤波器 I 的 VHDL 设计。

```
package n_bit_int is
subtype bits15 is integer range - 2 ** 14 to 2 ** 14 - 1;
end n_bit_int;
library work;
```

```
use work.n_bit_int.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity iir_i is
port( x_in:in bits15;
      clk:in std_logic;
      y_out:out bits15
      );
end iir_i;
architecture a of iir_i is
  signal x,y:bits15:=0;
begin
    process(clk)
        begin
            if clk'event and clk='1' then
                x<=x_in;
                y<=x+y/4+y/2;
            end if;
    end process;
    y_out<=y;
end a;
```

仿真结果如图 5.19 所示。仿真为滤波器对幅值为 1000[①] 的脉冲响应仿真结果。

图 5.19　有耗积分器Ⅰ脉冲的响应仿真结果

2. 有耗积分器Ⅱ

一阶 IIR 系统的差分方程为 $y[n+1]=ay[n]+bx[n]$，一阶系统的输出就是 $y[n+1]$。可以采用预先考虑的方法计算，将 $y[n+1]$代入 $y[n+2]$的差分方程就是：

$$y[n+2]=a\ y[n+1]+bx[n+1]=a^2y[n]+abx[n]+bx[n+1]$$

其等价系统如图 5.20 所示。

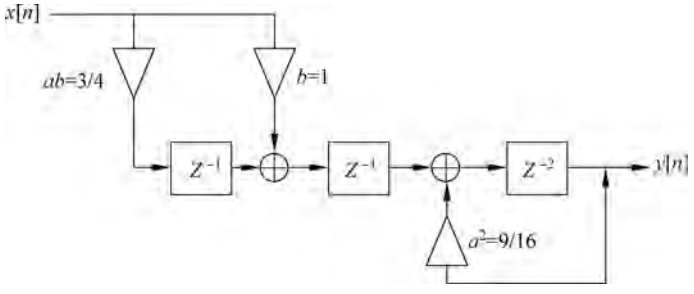

图 5.20　采用预先算法的有耗积分器

① “1000”为通过仿真软件数字化后的结果，无单位。全书同。

通过采用预先考虑(S－1)步骤的转换，就可以生成这一概念，结果如下：

$$y[n+S]=a^S y[n]+\underbrace{\sum_{k=0}^{S-1}a^k bx[n+S-1-k]}_{(\eta)}$$

从中可以看出：(η)项定义了 FIR 滤波器的系数$\{b,ab,a^2b,\cdots,a^{S-1}b\}$，后者可以采用流水线技术。而递归部分也可以利用系数为 a^2 的 S 阶流水线乘法器来实现。分析图 5.20 给出的有耗积分器，它由一个非递归部分(如 FIR 滤波器)和一个具有延迟为 2s 和系数为 9/16 的递归部分构成的。满足如下方程。

$$\begin{aligned}y[n+2]&=3/4y[n+1]+x[n+1]=3/4(3/4y[n]+x[n])+x[n+1]\\&=9/16y[n]+3/4x[n]+x[n+1]\end{aligned}$$

实现这种 IIR 滤波器的 VHDL 代码如例 5.17 所示。

【例 5.17】 有耗积分器Ⅱ的 VHDL 设计。

```
package n_bit_int is
subtype bits15 is integer range - 2 ** 14 to 2 ** 14 - 1;
end n_bit_int;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.n_bit_int.all;
entity iir2 is
port(     x_in:in bits15;
              y_out:out bits15;
          clk:in std_logic);
end iir2;
architecture a of iir2 is
    signal x,x3,sx,y,y9:bits15: = 0;
begin
    process
        begin
            wait until clk = '1';
            x < = x_in;
            x3 < = x/2 + x/4;            -- 计算 x * 3/4
            sx < = x + x3;               -- x 部分求和
            y9 < = y/2 + y/16;           -- 计算 y * 9/16
            y < = sx + y9;               -- 计算输出
    end process;
    y_out < = y;
end a;
```

该例中先计算了 9/16 * y[n]与 3/4 * x[n]；然后计算 3/4 * x[n]＋x[n＋1]＋9/16 * y[n]。仿真结果如图 5.21 所示。仿真为滤波器对幅值为 1000 的脉冲响应仿真结果。

图 5.21 有耗积分器Ⅱ脉冲的响应仿真结果

3. 有耗积分器Ⅲ

并行处理滤波器的实现是由 P 个并行 IIR 通路构成的，每个信道都以 $1/P$ 个输入采样速率运行，它们在输出位置靠多路复用器合成在一起，一般情况下，由于多路复用器要比乘法器和/或加法器速度快，所以并行方法速度也就更快。进一步讲，每个信道 P 都有一个 P 因子来计算其指定的输出。以 $P=2$ 的一阶系统为例，预先考虑与有耗积分器Ⅱ相同。现将其分成偶数 $n=2k$ 和奇数 $n=2k-1$ 输出序列，得到：

$$y[n+2]=\begin{cases}y[2k+2]=a^2y[2k]+ax[2k]+x[2k+1]\\ y[2k+1]=a^2y[2k-1]+ax[2k-1]+x[2k]\end{cases}$$

其中 $n,k\in\mathbf{Z}$。这两个方程是 IIR 滤波器 FPGA 实现的基础。

例 5.18 给出了 $a=3/4$ 的并行有耗积分器的 VHDL 实现。双信道并行有耗积分器是两个非递归部分(x 的 FIR 滤波器)和两个延迟为 2s、系数为 9/16 的递归部分的组合。其框图如图 5.22 所示。

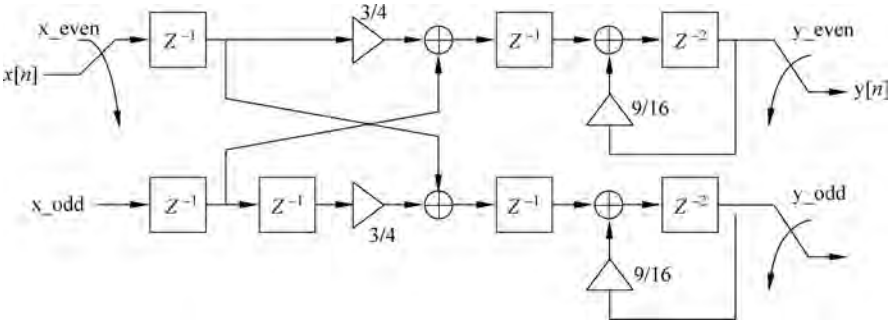

图 5.22　双通路并行 IIR 原理框图

【例 5.18】 有耗积分器Ⅲ的 VHDL 设计。

```
package n_bit_int is
subtype bits15 is integer range -2 ** 14 to 2 ** 14 - 1;
end n_bit_int;
library work;
use work.n_bit_int.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity iir3 is
port( x_in:in bits15;
      y_out:out bits15;
        clk:in std_logic;
      clk2:out std_logic);
end iir3;
architecture a of iir3 is
    type state_type is (even,odd);
    signal state:state_type:=even;
    signal x_even,x_odd,xd_odd,x_wait:bits15:=0;
    signal y_even,y_odd,y_wait,y:bits15:=0;
    signal x_e,x_o,y_e,y_o:bits15:=0;
```

```
    signal sum_x_even,sum_x_odd:bits15: = 0;
    signal clk_div2:std_logic;
    begin
    process                              -- 将输入 x 分为奇偶部分
    begin                                -- 在时钟驱动下重新组合 y
        wait until clk = '1';
            case state is
            when even = >
                x_even < = x_in;
                x_odd < = x_wait;
                clk_div2 < = '1';
                y < = y_wait;
                state < = odd;
            when odd = >
                x_wait < = x_in;
                y < = y_odd;
                y_wait < = y_even;
                clk_div2 < = '0';
                state < = even;
            end case;
        end process;
    y_out < = y;
    clk2 < = clk_div2;
        process                          -- 滤波器算法的实现
        begin
        wait until clk_div2 = '0';
            -- sum_x_even < = (x_even * 2 + x_even)/4 + x_odd;
            -- y_even < = (y_even * 8 + y_even)/16 + sum_x_even;
            sum_x_even < = x_even/2 + x_even/4 + x_odd;
            y_even < = y_even/2 + y_even/16 + sum_x_even;
            xd_odd < = x_odd;
            sum_x_odd < = xd_odd/2 + xd_odd/4 + x_even;
            y_odd < = y_odd/2 + y_odd/16 + sum_x_odd;
        end process;
end a;
```

并行 IIR 仿真结果如图 5.23 所示。

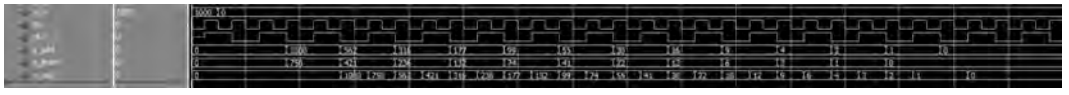

图 5.23　并行 IIR 滤波器对脉冲的响应仿真结果

5.4　TestBench 中随机数的设计

在 VHDL 写 TestBench 时有时会根据仿真需要产生一个随机数，如 Verilog 中的 random 函数。这个随机数的产生可以使用 math_real 函数包中的 uniform 函数得到一个 real 类型的归一随机数，然后可以对这个数进行其他处理来满足具体要求，例如扩大倍数、截掉小数等，此类操作需要利用一个类型转换函数实现。例 5.19 实现一个 4 位二进制数的

平方功能，为了验证该功能，在例 5.20 中的 TestBench 中利用随机数函数 uniform 和转换函数实现一个 4 位二进制随机数提供给乘法器输入端口。仿真结果如图 5.24 所示。

图 5.24 4 位乘法器的仿真结果

【例 5.19】 4 位二进制数平方。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity vhdl_random_test is
    port (
        idata:   in   std_logic_vector(3 downto 0);
        odata :  out  std_logic_vector(7 downto 0)
    );
end entity vhdl_random_test;
architecture one of vhdl_random_test is
begin
    odata <= idata * idata;
end architecture one;
```

【例 5.20】 4 位二进制数平方的 TestBench 设计。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;
use ieee.numeric_std.all;
entity vhdl_random_test_tb is
end entity vhdl_random_test_tb;
architecture one of vhdl_random_test_tb is
    component  vhdl_random_test is
        port (
            idata:   in   std_logic_vector(3 downto 0);
            odata :  out  std_logic_vector(7 downto 0)
            );
    end component;
    signal idata : std_logic_vector (3 downto 0);
    signal odata : std_logic_vector (7 downto 0);
    signal i     : integer := 0;
begin
    vhdl_random_test_inst : vhdl_random_test
        port map (
            idata => idata,
            odata => odata
                );
    process
```

```
        VARIABLE seed1, seed2: positive;
        VARIABLE rand: real;
        VARIABLE int_rand: integer;
    begin
        UNIFORM(seed1, seed2, rand);
        int_rand : = INTEGER(TRUNC(rand * 16.0));
        idata < = std_logic_vector(to_unigned(int_rand, 4));
        i < = i + 1;
        if i = 30 then
            wait;
        else
            wait for 20 ns;
        end if;
    end process;
end architecture one;
```

例子中 uniform 函数需要声明 ieee.math_real 程序包。uniform 在第一次调用之前需要统一种子值(SEED1,SEED2),该种子值必须初始化为范围在 12147483562～12147483398 的值,每次调用后统一修改种子值。产生的随机数在 0.0～1.0 之间。

TRUNC 函数主要防止数据溢出,当转换数据超出最大值后将保持最大值输出。本例中由于输入数据为 4bit,所以扩大 16 倍。示例中用了两个转换函数,一个是将随机数扩大 16 倍后转换为整数。后续将整数转换为 4bit 的 std_logic_vector 数据类型并赋值给乘法器输入。该转换函数为 std_logic_vector(to_unsigned(int_rand,4)),该函数使用前需声明 ieee.numeric_std.all;程序包。

5.5 二进制频移键控调制与解调的 VHDL 实现

频移键控(FSK)使用不同频率的载波来传送数字信号,并用数字基带信号控制载波信号的频率。二进制频移键控使用两个不同频率的载波来代表数字信号的两种电平。接收端接收到不同的载波信号再进行逆变换成数字信号,完成信息传输过程。

5.5.1 FSK 调制的 VHDL 实现

FSK 调制的建模方框图如图 5.25 所示。FSK 调制的核心部分包括分频器、二选一选通开关等。图 5.25 中的两个分频器分别产生两路数字载波信号。二选一选通开关的作用是:以基带信号作为控制信号,当基带信号为“0”,选通载波 f1;当基带信号为“1”,选通载波 f2。从选通开关输出的信号就是数字 FSK 信号。图中调制信号为数字信号。

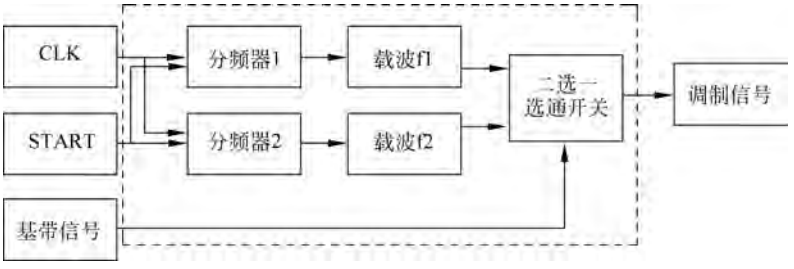

图 5.25 FSK 调制的建模方框图

【例 5.21】 FSK 调制的 VHDL 描述。

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

entity fsk is
    port(clk:in std_logic;                  -- 系统时钟
       start:in std_logic;                  -- 允许调制信号
           x:in std_logic;                  -- 基带信号
           y:out std_logic);                -- 调制信号
end fsk;
architecture a of fsk is
    signal q1:integer range 0 to 11;        -- 载波信号 f1 的分频计数器
    signal q2:integer range 0 to 3;         -- 载波信号 f2 的分频计数器
    signal f1,f2:std_logic;                 -- 载波信号 f1、f2
begin
    process(clk)                            -- 此进程完成时钟分频得到载波
    begin
      if clk'event and clk = '1' then
           if start = '0' then q1 <= 0;
           elsif q1 = 11 then q1 <= 0;
           elsif q1 <= 5 then f1 <= '1';q1 <= q1 + 1;
           else f1 <= '0';q1 <= q1 + 1;

           end if;
      end if;
    end process;
    process(clk)
    begin
          if clk'event and clk = '1' then
          if start = '0' then q2 <= 0;
          elsif q2 = 3 then f2 <= '0';q2 <= 0;
          elsif q2 <= 1 then f2 <= '1';q2 <= q2 + 1;
          else f2 <= '0';q2 <= q2 + 1;
          end if;
    end if;
    end process;
    process(clk,x)                          ---- 此进程完成对基带信号的 FSK 调制
    begin
          if clk'event and clk = '1' then
            if x = '0' then y <= f1;
            else y <= f2;
            end if;
          end if;
    end process;
end a;
```

FSK 调制的仿真结果如图 5.26 所示。

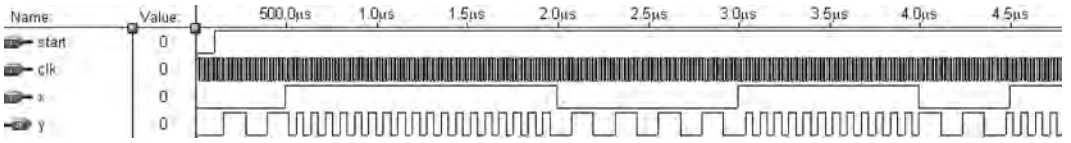

图 5.26　FSK 调制的仿真结果

5.5.2　FSK 信号解调的 VHDL 实现

FSK 解调方框图如图 5.27 所示。其核心部分由分频器、寄存器、计数器和判决器组成。其中分频器的分频系数取值对应图 5.25 中的分频器 1 和分频器 2 中较小的分频系数值，即 FSK 解调器的分频器输出为较高的那个载波信号。由于 f1 和 f2 的周期不同，若设 f1＝2f2，且基带信号电平“1”对应 f1；基带信号电平“0”对应载波 f2，则图 5.27 中计数器以 f1 为时钟信号，上升沿计数，基带信号“1”码元对应的计数器个数为 1/f1，基带信号“0”码元对应的计数器个数为 1/f2。计数器根据两种不同的计数情况，对应输出“0”和“1”两种电平。判决器以 f1 为时钟信号，对计数器输出信号进行抽样判决，并输出基带信号。图中没有包含模拟电路部分，调制信号为数字信号形式。

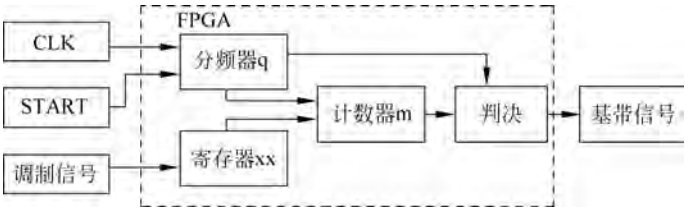

图 5.27　FSK 解调方框图

【例 5.22】 FSK 信号解调的 VHDL 描述。

```
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity fskm is
port(clk:in std_logic;                                  -- 系统时钟
   start:in std_logic;                                  -- 同步信号
       x:in std_logic;                                  -- 调制信号
       y:out std_logic);                                -- 基带信号
end fskm;
architecture a of fskm is
   signal q:integer range 0 to 11;                      -- 分频计数器
   signal m:integer range 0 to 5;                       -- 计数器
   signal xx:std_logic;                                 -- 寄存器
begin
   process(clk)                                         -- 对系统时钟进行分频
   begin
     if clk'event and clk = '1' then xx < = x;          -- 时钟上升沿存储调制信号值
       if start = '0' then q < = 0;
       elsif q = 11 then q < = 0;
```

```
        else q <= q + 1;
        end if;
      end if;
end process;
   process(xx,q)                                    -- 此进程完成 FSK 解调
   begin
     if q = 11 then m <= 0;
     elsif q = 10 then if m <= 3 then y <= '0';     -- 通过 m 大小判决输出电平
     else y <= '1';
     end if;
     if xx'event and xx = '1' then m <= m + 1;      -- 计 x 信号脉冲个数
     end if;
   end process;
end a;
```

5.6 基于 DDS 信号发生器的设计

本节介绍了一个频率和相位可调的正弦波信号发生器的设计。系统要求输出电压的最大值为 5V。①输出频率范围为 10～70Hz 的三相交流电，要求分别输出电流和电压三相电信号；②相位在 0～360°范围内，步进 0.1°可调；③频率步进 0.01Hz 可调；④输出电压波形应尽量接近正弦波，用示波器观察无明显失真。

此设计是以 FPGA、单片机为主要控制核心，控制键盘的输入与 LED 显示，完成单片机与 FPGA 之间的数据传输，FPGA 向存储器输出地址，使存储器将相应单元的数据传输到 D/A 转换器中，再经可调带通滤波器进行波形整形，最后输出完整的波形，如图 5.28 所示为设计的总体框图。

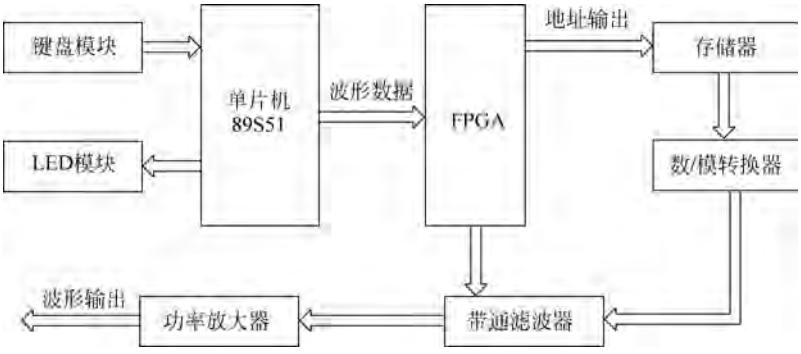

图 5.28 设计的总体框图

此系统利用 DDS 技术实现波形的发生与控制，单片机的控制电路主要完成将输入的数值转换成二进制以后将相应的频率控制字及相位数值输出到 DDS 模块。本节主要介绍 FPGA 部分的作用及实现。

5.6.1 DDS 设计及原理

直接数字频率合成(DDS)是从相位概念出发直接合成所需波形的一种新的频率合成技

术。它在相对带宽、频率转换时间、相位连续性、正交输出、高分辨率以及集成化等一系列性能指标方面已远远超过了传统频率合成技术。

DDS的基本原理框图如图5.29所示，它主要是以数控振荡器的方式，产生频率、相位可控制的正弦波。它主要由标准参考频率源、相位累加器、波形存储器、数/模转换器、低通平滑滤波器等构成。其中，参考频率源一般是一个高稳定度的晶体振荡器，其输出信号用于DDS中各部件同步工作。DDS的实质是对相位进行可控等间隔的采样。

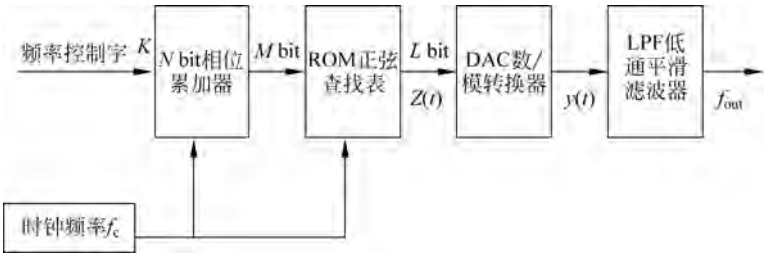

图5.29 DDS的基本原理框图

相位累加器的结构如图5.30所示。

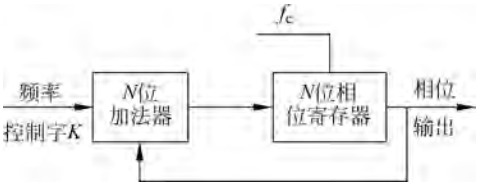

图5.30 相位累加器的结构示意图

它是实现DDS的核心，由一个 N 位字长的加法器和一个由固定时钟脉冲取样的 N 位相位寄存器组成。将相位寄存器的输出和外部输入的频率控制字 K 作为加法器的输入，在时钟脉冲到达时，相位寄存器对上一个时钟周期内相位加法器的值与频率控制字 K 之和进行采样，作为相位累加器在此刻时钟的输出。相位累加器输出的高 M 位作为波形存储器查询表的地址，从波形存储器中读出相应的幅度值送到数/模转换器。

当DDS正常工作时，在标准参考频率源的控制下，相位累加器不断进行相位线性累加（每次累加值为频率控制字 K），当相位累加器积满时，就会产生一次溢出，从而完成一个周期性的动作，这个周期就是DDS合成信号的频率周期。输出信号的频率为

$$f_{out}=\frac{w}{2\pi}=\frac{\frac{2\pi}{2^N}\times K\times f_c}{2\pi}=\frac{K\times f_c}{2^N}$$

显而易见，当 $K=1$ 时输出最小频率，即频率分辨率为 $f_{min}=\frac{f_c}{2^N}$。式中，f_{out} 为输出信号的频率；K 为频率控制字；N 为相位累加器字长；f_c 作为标准参考频率源工作频率。

5.6.2 FPGA内部的DDS模块的设计与实现

在此设计中，所要求的相位可调精度为0.1°，频率的可调精度要达到0.01Hz。由于一个周期为360°，所以在一个周期内进行3600点的采样。为了便于计算和提高精度，选用的

时钟频率为 3.6MHz。这样相位累加器的最大计数值为

$$\text{Pacc}=\frac{f_c}{f_{\min}}=\frac{3.6\times10^6\text{Hz}}{0.01\text{Hz}}=3.6\times10^8$$

式中，Pacc 为相位累加器的计数值，f_c 为标准参考频率源工作频率，$f_{\min}$ 为输出频率可调精度。

则相位累加器的最少位数 N 为

$$N=\frac{\lg(\text{Pacc})}{\lg2}=\frac{\lg(3.6\times10^8)}{\lg2}=29$$

因为频率为 3.6×10^6 Hz，相位累加器一周期的累加数为 3.6×10^8，取样 3600 次，则取样一次所需的计数值为

$$n=\frac{\text{Pacc}}{3600}=\frac{3.6\times10^8}{3600}=10^5$$

输入的频率精确到 0.01Hz，在计算频率控制字时，首先将输入的数值扩大 100 倍作为频率控制字 K，例如输入的频率为 10Hz，则频率控制字为 1000，当累加器累加到 10^5 时，取高 13 位地址作为外部存储器的读取地址。图 5.31 为 FPGA 内部的 DDS 模块。

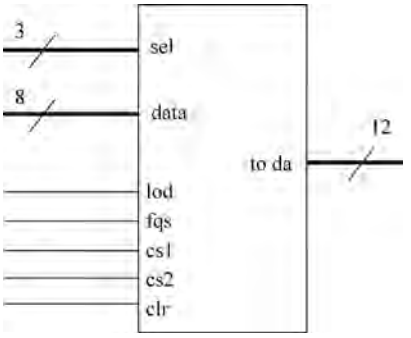

图 5.31 FPGA 内部的 DDS 模块

说明：

(1) cs1＝0 时，允许写数据到 DDS；cs1＝1 时，不允许写数据到 DDS。

(2) cs2＝1 时，DDS 停止工作；cs2＝0 时，DDS 正常工作。

(3) fqs＝1 时，由 SEL0、SEL1、SEL2 三位为低 8 位数据和高 8 位数据选择端。当 sel2、sel1、sel0 为“000”时选择频率的低 8 位，为“001”时选择频率的高 8 位；sel2、sel1、sel0 为“010”时选择电流 A 路相位的低 8 位；为“011”时选择电流 A 路相位的高 8 位；为“100”时选择电流 B 路相位的低 8 位；为“101”时选择电流 B 路相位的高 8 位；为“110”时选择电流 C 路相位的低 8 位；为“111”时选择电流 C 路相位的高 8 位。fqs＝0，选择电压的三路相位，其数据的控制命令字与 fqs＝1 相同。

(4) data 为数据线(8 位)。

(5) 例如选择输出 10.00Hz，(分辨率为 0.01Hz)需要输入频率字$(1000)_{10}(0000001111101000)_2$。

(6) 选择相位差为 10.1°(分辨率为 0.1°)，需输入相位字$(101)_{10}(0000000001100101)_2$。

(7) 相位和频率可任意时刻设置，频率默认值为 00000000001Hz，相位默认值为 0。

(8) to da 为 12 位地址线，直接接到外围 EPROM 的 12 位地址线。

【例 5.23】 产生频率、相位正弦信号的 VHDL 描述。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
…
entity dds18m1 is
port(clk:in std_logic;
```

```
      clkout:out std_logic;
      lod,fqs,cs1,cs2:in std_logic;
      sel: in std_logic_vector(2 downto 0);
      data: in std_logic_vector(7 downto 0);
     vaddera:out std_logic_vector(12 downto 0);
     vadderb:out std_logic_vector(12 downto 0);
     vadderc:out std_logic_vector(12 downto 0);
     iaddera:out std_logic_vector(11 downto 0);
     iadderb:out std_logic_vector(11 downto 0);
     iadderc:out std_logic_vector(11 downto 0)
   );
      -- vadder:out std_logic_vector(12 downto 0);     -- integer range 7200 downto 0;
      -- iadder:out std_logic_vector(12 downto 0));    -- integer range 3600 downto 0;
end  dds18m1;
--
architecture behave of dds18m1 is
    constant n  :  std_logic_vector(16 downto 0):="11000011010100000";
-- 100000=3.6M/3600  clk=3.6M  constant  n
std_logic_vector(18 downto 0):=x"7A120";              -- 50000=18M/3600 clk=18M
    signal    add :    std_logic_vector(11 downto 0); -- 7200 个地址单元
    signal    m   :    std_logic_vector(32 downto 0);
    signal    fset:    std_logic_vector(15 downto 0);
-- 频率分辨率位 0.01Hz ,最高频率 100Hz,100/0.01=10000,频率字要 16 位
    signal         pseta,psetb,psetc,psa,psb,psc:   std_logic_vector(11 downto 0);
-- 相位分辨率为 0.1 度 360*0.1=3600     2^11<3600<2^12
    signal    cp,ld,clkk:  std_logic;
    signal    acc: std_logic_vector(3 downto 0);
begin
    process(clk)
    begin
    if clk'event and clk='1'then
        if acc=4 then
            acc<="0000";
            clkk<='0';
        else
            acc<=acc+1;
            if acc<3 then
            clkk<='1';
            else
            clkk<='0';
            end if;
        end if;
    end if;
    end process;

    process(ld)
    begin
    if ld'event and ld='0'then
            if fqs='1'then                          -- 选择频率输入
                if sel="000" then
                    fset(7 downto 0)<=data;         -- SEL=00 选择低 8 位
                elsif sel="001"then
                    fset(15 downto 8)<=data;
                    -- fset(31 downto 16)<="0000000000000000";
```

```
                --elsif sel = "10"then
                    --fset(31 downto 16)<=data;
                   elsif sel = "100" then
                     psb(7 downto 0)<=data;
                elsif sel = "101" then
                      psb(11 downto 8)<=data(3 downto 0);
                      elsif sel = "110" then
                      psc(7 downto 0)<=data;
                elsif sel = "111" then
                      psc(11 downto 8)<=data(3 downto 0);
                      elsif sel = "010" then
                      psa(7 downto 0)<=data;
                elsif sel = "011" then
                      psa(11 downto 8)<=data(3 downto 0);
                 else
                NULL;
                end if;
            else                                          --选择相位输入
                if sel = "100" then
                    psetb(7 downto 0)<=data;
                elsif sel = "101" then
                    psetb(11 downto 8)<=data(3 downto 0);
                elsif sel = "110" then
                    psetc(7 downto 0)<=data;
                elsif sel = "111" then
                    psetc(11 downto 8)<=data(3 downto 0);
                elsif sel = "010" then
                    pseta(7 downto 0)<=data;
                elsif sel = "011" then
                    pseta(11 downto 8)<=data(3 downto 0);
                else
                    NULL;
                end if;
            end if;
         --end if;
    end if;
    end process;
    process(cp)
    begin
        if (cp'event and cp='1')  then
            if  (add="111000010000")  then
                add<="000000000000";
                elsif m<n then
                m<=m+fset;
            else
                m<=m-n+fset;
                add<=add+1;
            end if;
        end if;
    end process;
ld<=(not cs1) and lod;
cp<=(not cs2) and clkk;
clkout<=not clkk;
vaddera<=add+pseta;
```

```
vadderb <= add + psetb;
vadderc <= add + psetc;
iaddera(11 downto 0)<= add + psa;
iadderb(11 downto 0)<= add + psb;
iadderc(11 downto 0)<= add + psc;
end behave;
```

5.6.3 仿真结果及说明

这部分的仿真是对核心部分 DDS 的测试。如图 5.32 所示是信号频率为 10Hz 时输入频率控制字及各路相位的时序仿真图。其中电流 a 路相位为 0°，电流 b 路相位为 120°，电流 c 路相位为 240°，电压 a 路相位为 0°，电压 b 路相位为 100°，电压 c 路相位为 200°；图 5.33 所示为 DDS 内部的实现过程，由相位仿真时序图可以看出在 DDS 内部累加 3600 个数以后，重新开始循环，且循环一个周期的时间为 100ms，正好与设定值相对应，因此，此 DDS 程序设计是正确的。

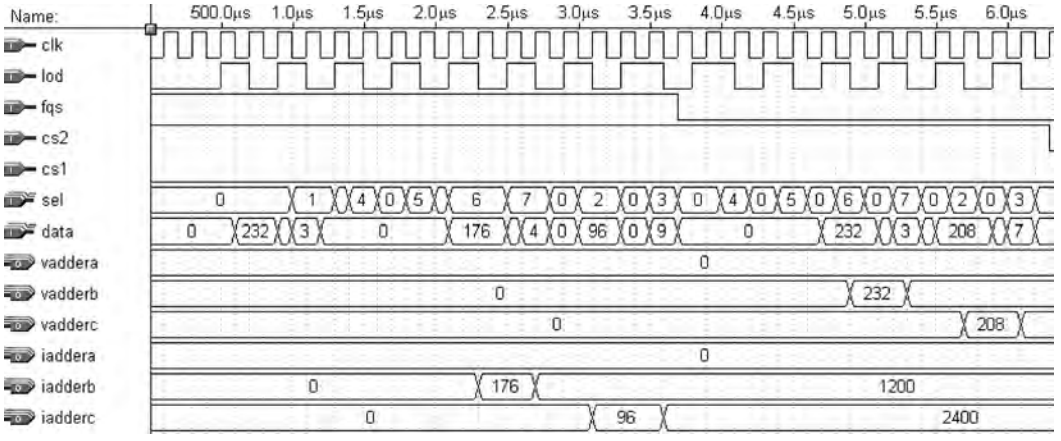

图 5.32 频率控制字及各相位的时序仿真图

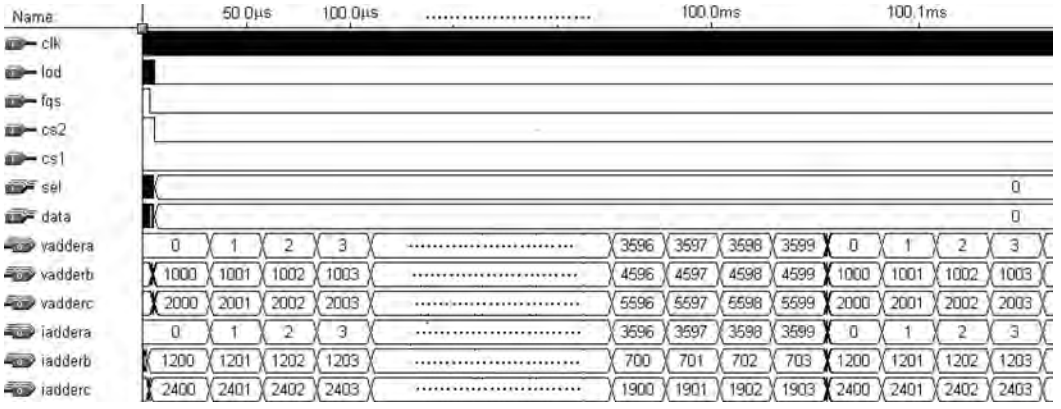

图 5.33 DDS 时序仿真图

5.7 SD卡驱动器设计

SD卡在现在的日常生活与工作中使用非常广泛，时下已经成为最为通用的数据存储卡。在诸如MP3、数码相机等设备上大多采用SD卡作为其存储设备。SD卡之所以得到如此广泛的使用，是因为它价格低廉、存储容量大、使用方便、通用性与安全性强。

SD卡按容量(Capacity)分类，可以分为：

(1) 标准容量卡：Standard Capacity SD Memory Card(SDSC)：容量小于等于2GB。

(2) 高容量卡：High Capacity SD Memory Card(SDHC)：容量大于2GB，小于等于32GB。

(3) 扩展容量卡：Extended Capacity SD Memory Card(SDXC)：容量大于32GB，小于等于2TB。

5.7.1 SD卡电路结构

SD卡内部结构如图5.34所示。由图可知，SD卡上所有单元由内部时钟发生器提供时钟。接口驱动单元同步外部时钟的DAT和CMD信号到内部所用时钟。另外SD卡有6个

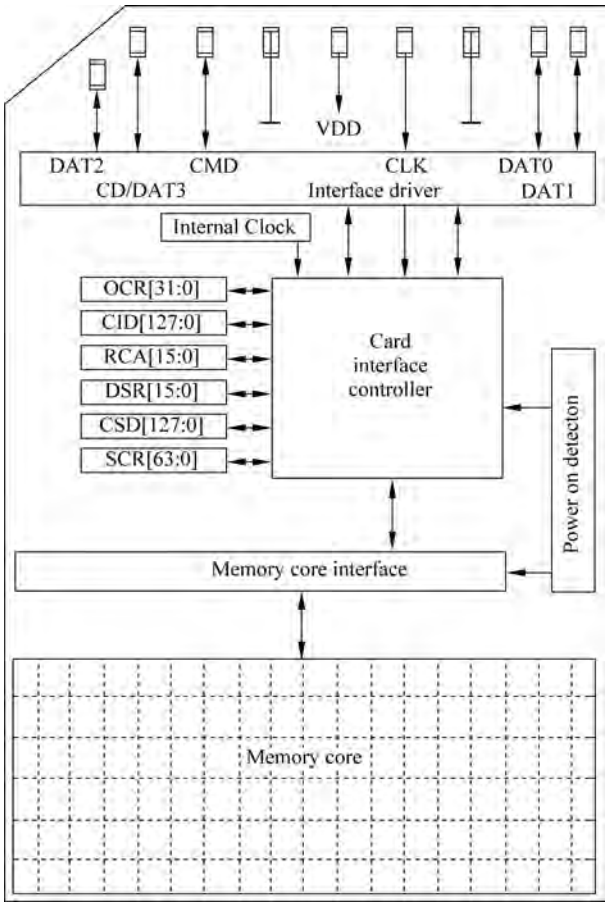

图5.34 SD卡内部结构图

寄存器 OCR、CID、CSD、RCA、DSR、SCR。表 5.1 为各个寄存器功能含义。其中前 4 个保存卡的特定信息，后两个用来对卡进行配置。在 DE2-115 平台中 SD 卡与 FPGA_N 电路图如图 5.35 所示。

表 5.1　SD 卡寄存器说明

寄存器名称	位宽	功 能 描 述
OCR	32	支持的电压
CID	128	卡信息：生产商、OEM、产品名称、版本、出产日期、CRC 校验
RCA	16	卡地址：在初始化时发布，用于与 host 通信，0x0000 表示与所有卡通信
DSR	16	驱动相关、总线电流大小、上升沿时间、最大开启时间、最小开启时间
CSD	128	数据传输要求：包括读写时间、读写电压最大最低值、写保护、块读写错误
SCR	64	特性支持，如 CMD 支持、总线数量支持

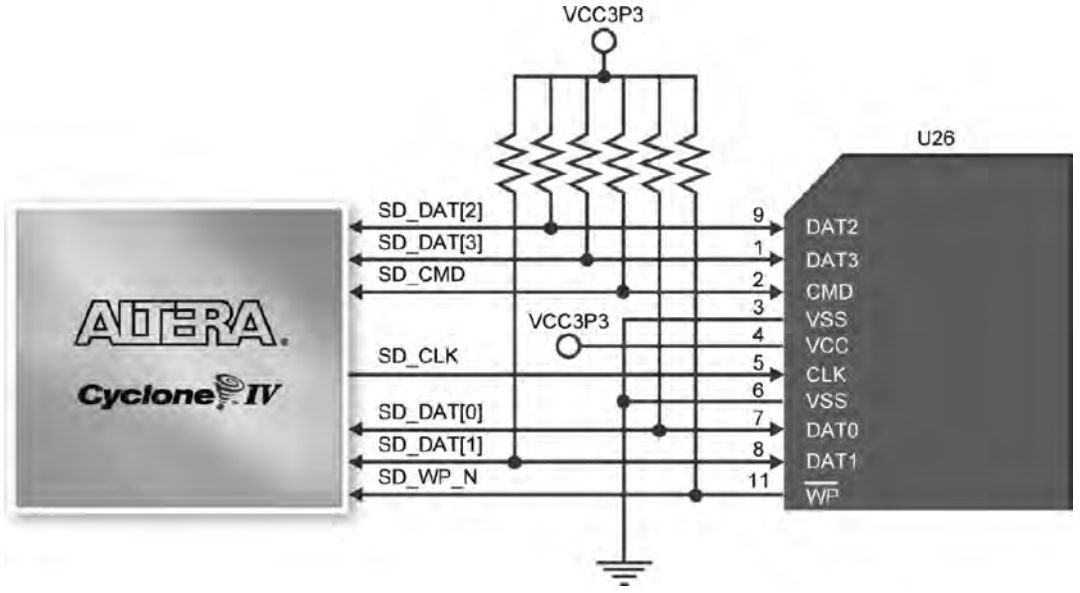

图 5.35　SD 卡与 FPGA 连接电路原理图

SD 卡支持两种总线方式：SD 方式与 SPI 方式。各个模式下引脚功能如表 5.2 所示。其中 SD 方式采用 6 线制，使用 CLK、CMD、DAT0～DAT3 进行数据通信。而 SPI 方式采用 4 线制，使用 CS、CLK、DataIn、DataOut 进行数据通信。SD 方式时的数据传输速度比 SPI 方式要快。采用不同的初始化方式可以使 SD 卡工作于 SD 方式或 SPI 方式。SD 卡模式允许 4 线的高速数据传输，SPI 模式允许简单通用的 SPI 通道接口，这种模式相对于 SD 模式的不足之处是丧失了速度。如果接到复位命令(CMD0)时，CS 信号有效(低电平)，SPI 模式启用。本书只对其 SPI 方式进行介绍。CLK：每个时钟周期传输一个命令或数据位。频率可在 0～25MHz 之间变化。在 SPI 模式下，CRC 校验是被忽略的，但依然要求主从机发送 CRC 码，只是数值可以是任意值，一般主机的 CRC 码通常设为 0x00 或 0xFF。

表 5.2 SD卡引脚功能

引脚序号	SD模式			SPI模式		
	名称	类型	描述	名称	类型	描述
1	CD/DAT3	IO/PP	卡检测/数据线3	#CS	I	片选
2	CMD	PP	命令/回应	DI	I	数据输入
3	VSS1	S	电源地	VSS	S	电源地
4	VDD	S	电源	VDD	S	电源
5	CLK	I	时钟	SCLK	I	时钟
6	VSS2	S	电源地	VSS2	S	电源地
7	DAT0	IO/PP	数据线0	DO	O/PP	数据输出
8	DAT1	IO/PP	数据线1	RSV		
9	DAT2	IO/PP	数据线2	RSV		

5.7.2 SD卡命令

1. 命令类型

广播指令bc：不需要响应。

广播指令bcr，每个卡都会独立接收指令并发送响应。

点对点指令ac：发送完此类命令后，只有指定地址的SD卡会给予反馈（地址通过命令请求SD卡发布，是唯一的）。此时DAT线上无数据传输。

点对点数据传输指令adtc：发送完此类命令后，只有指定地址的SD卡会给予反馈。此时DAT线上有数据传输。

2. 命令格式

所有命令均按照表5.3所示格式，总共48bit。首先是1bit起始位'0'，然后是1bit方向位（主机发送'1'），6bit命令位（0～63），32bit参数（部分命令需要），7bit CRC7校验，1bit停止位'1'。

表 5.3 SD卡命令格式

位	47	46	[45:40]	[39:8]	[7:1]	0
宽度	1	1	6	32	7	1
值	'0'	'1'	X	X	X	'1'
功能	起始位	传输位	命令	命令参数	CRC7	停止位

3. 命令分类

SD卡操作命令根据不同的类型分成不同的class。其中class0、2、4、5、8是每个卡都必须支持的命令，不同的卡所支持的命令保存在CSD中。

4. 应答格式

所有应答都是通过CMD发送，不同的应答长度可能不同，共有4种类型的应答。

(1) R1：长度为48bit，格式如表5.4所示。

表 5.4 R1 应答响应格式

位	47	46	[45:40]	[39:8]	[7:1]	0
宽度	1	1	6	32	7	1
值	'0'	'0'	X	X	X	'1'
功能	起始位	传输位	命令	命令参数	CRC7	停止位

(2) R2(CID CSD 寄存器)：长度为 136bit，CID 为 CMD2 和 CMD10 的应答，CSD 为 CMD9 的应答。应答格式如表 5.5 所示。

表 5.5 R2 应答响应格式

位	135	134	[133:128]	[127:1]	0
宽度	1	1	6	127	1
值	'0'	'0'	"111111"	X	'1'
功能	起始位	传输位	保留	CID 或 CSD 寄存器包含了 CRC7	停止位

(3) R3(OCR 寄存器)：长度为 48bit，作为 ACMD41 的应答，格式如表 5.6 所示。

表 5.6 R3 应答响应格式

位	47	46	[45:40]	[39:8]	[7:1]	0
宽度	1	1	6	32	7	1
值	'0'	'0'	"111111"	X	"111111"	'1'
功能	起始位	传输位	保留	OCR 寄存器	保留	停止位

(4) R6(RCA 地址应答)：长度为 48bit，格式如表 5.7 所示。

表 5.7 R6 应答响应格式

位	47	46	[45:40]	[39:8]		[7:1]	0
宽度	1	1	6	16	16	7	1
值	'0'	'0'	X	X	X	X	'1'
功能	起始位	传输位	命令序号 ("000011")	新版 RCA[31:16]	[15:0] 标准卡位	CRC7	停止位

5.7.3 SD 卡数据读取流程

本节主要讲解如何读取 SD 中数据流程。数据为 bin 格式的图像数据，图像大小为 640×480 RGB656 格式数据，该数据存储在 SD 卡起始地址为 40092 的位置，偏移地址为 1200。注意例中为 SD HC 卡，该卡只能以块进行读取。偏移地址计算办法为图像数据字节数除以 512(块字节)＝640×480×2/512＝1200。

读取 SD 卡数据步骤如下：

(1) 延时至少 74clock，等待 SD 卡内部操作完成，在 MMC 协议中有明确说明。

(2) CS 低电平选中 SD 卡。

(3) 发送 CMD0，需要返回 0x01，进入 Idle 状态。

(4) 为了区别 SD 卡是 2.0、1.0,还是 MMC 卡,根据协议向上兼容的原理,首先发送只有 SD2.0 才有的命令 CMD8,如果 CMD8 返回无错误,则初步判断为 2.0 卡,进一步发送命令循环发送 CMD55+ACMD41,直到返回 0x00,确定 SD2.0 卡初始化成功,进入 Ready 状态,再发送 CMD58 命令来判断是 HCSD 还是 SCSD,到此 SD2.0 卡初始化成功。如果 CMD8 返回错误则进一步判断为 1.0 卡还是 MMC 卡,循环发送 CMD55+ACMD41,返回无错误,则为 SD1.0 卡,至此 SD1.0 卡初始成功,如果在一定的循环次数下,返回为错误,则进一步发送 CMD1 进行初始化,如果返回无错误,则确定为 MMC 卡,如果在一定的次数下,返回为错误,则不能识别该卡,初始化结束。

(5) 发送 CMD17(单块)或 CMD18(多块)读命令,返回 0x00。

(6) 接收数据开始令牌 0xfe(或 0xfc)+正式数据 512Byte+CRC 校验 2Byte,默认正式传输的数据长度是 512Byte,可用 CMD16 设置块长度。

5.7.4 SD 卡数据读取代码说明

本节给出 SD 卡数据读取代码如例 5.24 所示。下面对代码中主要设计思想进行说明。SD 卡中大小为 640×480bin 格式图像文件由软件 image2LCD 制作生成。

类属声明中 ADDR 地址为 SD 卡中 BIN 文件所在的初始地址 40992,读者可以利用 winhex 软件查看 SD 卡文件所在地址,偏移地址 OFF_MAX 为 1200。

CMD0 为 X"400000000095",该指令为 SD 卡复位指令,其中第一个字节 X"40"参照表 5.3 得到,X"95"为 CRC7 校验+停止位'1'得出。

CMD8 为 X"48000001aa87",该指令为 SD 卡类型检测(SD V2.0),其中第一个字节 X"48"参照表 5.3 得到,X"01"为支持电压,DE2-115 支持 2.7～3.6V 电压故该字节为 X"01",字节 X"aa"为校验字节,主机发送后从机会原样返回,此处可任意修改。字节 X"87"为 CRC7 校验+停止位'1'得出。

SD 卡初始化成功后,CMD55 与 ACMD41 命令中 CRC 校验位可以忽略,以 X"ff"填入即可。

SD 初始化过程设计采用序列机方式进行设计。序列存储在信号 state 中,各个序列以十六进制表示。

序列 X"00"～X"02"为上电延时等待 SD 内部初始化;

序列 X"03"片选拉低,发送 CMD0;

序列 X"04"接收响应,如果完成第一个响应字节接收则跳转到 X"05",没有响应字节跳回至 X"01"重新初始化;

序列 X"05"判断响应字节是否为 X"01",是则跳转到下一序列 X"06",否则跳回至 X"01"重新初始化;

序列 X"06"～X"07"发送命令 CMD8 跳转到 X"08";

序列 X"08"判断响应是否接到 8 个响应字节,如果接到跳转到序列 X"09",否则跳回至序列 X"06";

序列 X"09"判断接收字节中第[19:16]位是否为 X"01",是则跳转到序列 X"0A",否则跳回至序列 X"06";

序列 X"0A"～X"0B"发送命令 CMD55 跳转至 X"0C";

序列 X“0C”接收响应，如果完成第 1 个响应字节接收则跳转到 X“0D”，没有响应字节则跳回至 X“0A”重新发送 CMD55；

序列 X“0D” 判断响应字节是否为 X“01”，是则跳转到下一序列 X“0E”，否则跳回至 X“0A”重新发送 CMD55；

序列 X“0E”～X“0F”发送命令 ACMD41 跳转至序列 X“10”；

序列 X“10”判断响应是否接到第 1 个响应字节，如果接到则跳转到序列 X“11”，否则跳回至序列 X“0E”；

序列 X“11” 判断响应字节是否为 X“00”，是则跳转到下一序列 X“12”，否则跳回至 X“0A”重新发送 CMD55；

序列 X“12”～X“14”发送命令 CMD17 跳转至序列 X“15”；其中 CMD17 命令由字节 X“51”＋4 字节地址(初始地址＋偏移地址)＋X“FF”构成，代码中地址按照块读取速度在增加；

序列 X“15”接收响应，如果完成第 1 个响应字节接收则跳转到 X“16”，没有响应字节跳回至 X“12”重新发送 CMD17；

序列 X“16” 判断响应字节是否为 X“00”，是则跳转到下一序列 X“17”，否则跳回至 X“12”重新发送 CMD17；

序列 X“17” 判断回传的 512B 数据的第一个起始位是否开始到达即响应字节是否为 X“FE”，是则跳转到下一序列 X“18”并置读使能，否则等待数据回传；

序列 X“18”判断是否接收完 512B 数据，如果完成则跳转序列 X“19”；

序列 X“19”～X“1A”块偏移地址＋1，如果偏移地址小于 1200 则跳转到序列序列 X“12”继续接收下一块数据，否则跳转至序列 X“1B”；

序列 X“1B”数据接收完毕。

【例 5.24】 SD 卡读取数据 VHDL 代码。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity sd_init_read is
    generic(
        T_10ms  :integer  := 100000;
        CNT_FREE_CLK:integer  := 100;
        ADDR  :integer:= 40992;                          -- X"A020"
        OFF_MAX  :integer:= 1200;
        CMD0:  std_logic_vector(47 downto 0):= X"400000000095";
        CMD8:  std_logic_vector(47 downto 0):= X"48000001aa87";
        CMD55: std_logic_vector(47 downto 0):= X"7700000000ff";
        ACMD41:std_logic_vector(47 downto 0):= X"6940000000ff"
    );
    port (
        clk  :  in  std_logic;                           -- 25MHz
        rst_n:  in  std_logic;
        read_done  :  out  std_logic;
```

```
        rdata:out std_logic_vector(7 downto 0);
        rflag:out std_logic;
        spi_cs_n:out std_logic;
        spi_mosi:out std_logic;
        spi_miso:in    std_logic
    );
end entity sd_init_read;
architecture one of sd_init_read is
    signal CMD17:    std_logic_vector(47 downto 0);
    signal cnt:      integer;                          -- 接收块计数器、延时计数器
    signal rec_en:   std_logic;                        -- 接收字节使能
    signal rec_clr:  std_logic;                        -- 接收字节复位全 1
    signal rec_data: std_logic_vector(47 downto 0);    -- 响应接收缓存
    signal send_data:std_logic_vector(47 downto 0);    -- 发送命令缓存
    signal read_en:  std_logic;                        -- 读字节使能
    signal read_en_r:std_logic;                        -- 读字节使能寄存
    signal rec_cnt:  integer;                          -- 接收字节计数器
    signal sd_addr:  integer;
    signal offset_addr:integer;
    signal state:    std_logic_vector(7 downto 0);
begin
    process (clk) begin                                -- 读使能寄存
        if clk'event and clk = '1' then
            read_en_r <= read_en;
        end if;
    end process;
    process (clk) begin                                -- 接收缓冲区处理
        if clk'event and clk = '1' then
            if rst_n = '0' then
                rec_data <= (others => '1');
            else
                if rec_clr = '1' then
                    rec_data <= (others => '1');
                else
                    if rec_en = '1' then
                        rec_data <= rec_data(46 downto 0) & spi_miso;
                    else
                        rec_data <= rec_data;
                    end if;
                end if;
            end if;
        end if;
    end process;
    process (clk) begin            -- 接收有效数据满 1B 则输出至外部端口
        if clk'event and clk = '1' then
            if rst_n = '0' then
                rdata <= X"00";
            else
                if rec_cnt = 0 and read_en_r = '1' then   -- 接收字节计数器为 0 与使能寄存
                    rdata <= rec_data(7 downto 0);
                end if;
```

```
            end  if;
        end if;
    end process;
    process(clk) begin
        if clk'event and clk = '1' then
            if rst_n = '0'then
                rflag < = '0';
            else
                if rec_cnt = 0 and read_en_r = '1' then
                    rflag < = '1';
                else
                    rflag < = '0';
                end if;
            end if;
        end if;
    end process;
    process(clk) begin
        if clk'event and clk = '1' then
            if rst_n = '0'then
                rec_cnt < = 0;
            else
                if read_en = '1' and rec_cnt < 7 then
                    rec_cnt < = rec_cnt + 1;
                else
                    rec_cnt < = 0;
                end if;
            end if;
        end if;
    end process;
    process(clk) begin
        if clk'event and clk = '0' then
            if rst_n = '0' then
                read_done < = '0';
                spi_cs_n < = '1';
                spi_mosi < = '1';
                state < = (others = > '0');
                cnt < = 0;
                rec_clr < = '1';
                rec_en < = '0';
                send_data < = (others = > '0');
                read_en < = '0';
                offset_addr < = 0;
            else
                case state is
                    when X"00"      = >
                        read_done < = '0';
                        spi_cs_n < = '1';
                        spi_mosi < = '1';
                        state < = X"01";
                        cnt < = 0;
                        rec_clr < = '1';
```

```
        rec_en <= '0';
        send_data <= (others => '0');
        read_en <= '0';
        offset_addr <= 0;
    when X"01" =>
        rec_clr <= '0';
        if cnt < T_10ms - 1 then
            cnt <= cnt + 1;
        else
            cnt <= 0;
            state <= X"02";
        end if;
    when X"02" =>
        spi_cs_n <= '0';
        if cnt < CNT_FREE_CLK - 1 then
            cnt <= cnt + 1;
        else
            cnt <= 0;
            state <= X"03";
            send_data <= CMD0;
        end if;
    when X"03" =>
        spi_cs_n <= '0';
        spi_mosi <= send_data(47);
        send_data <= send_data(46 downto 0) & '0';
        if cnt < 47 then
            cnt <= cnt + 1;
        else
            cnt <= 0;
            state <= X"04";
        end if;
    when X"04" =>
        spi_mosi <= '1';
        rec_en <= '1';
        if cnt < 100 then
            if rec_data(7) = '0' then
                rec_en <= '0';
                state <= X"05";
                spi_cs_n <= '1';
                cnt <= 0;
            else
                cnt <= cnt + 1;
            end if;
        else
            cnt <= 0;
            state <= X"01";
            spi_cs_n <= '1';
        end if;
    when X"05" =>
        rec_en <= '0';
        spi_cs_n <= '1';
```

```
        rec_clr <= '1';
        if rec_data(7 downto 0) = X"01" then
            state <= X"06";
        else
            state <= X"01";
        end if;
    when X"06" =>
        rec_clr <= '0';
        if cnt < 10 then
            cnt <= cnt + 1;
        else
            cnt <= 0;
            state <= X"07";
            send_data <= CMD8;
        end if;
    when X"07" =>
        spi_cs_n <= '0';
        spi_mosi <= send_data(47);
        send_data <= send_data(46 downto 0) & '0';
        if cnt < 47 then
            cnt <= cnt + 1;
        else
            cnt <= 0;
            state <= X"08";
        end if;
    when X"08" =>
        spi_mosi <= '1';
        if cnt < 100 then
            if rec_data(47) = '0' then
                rec_en <= '0';
                state <= X"09";
                spi_cs_n <= '1';
                cnt <= 0;
            else
                rec_en <= '1';
                cnt <= cnt + 1;
            end if;
        else
            cnt <= 0;
            state <= X"06";
            rec_en <= '0';
            spi_cs_n <= '1';
        end if;
    when X"09" =>
        rec_en <= '0';
        spi_cs_n <= '1';
        rec_clr <= '1';
        if rec_data(19 downto 16) = X"1" then
            state <= X"0A";
        else
            state <= X"06";
```

```
        end if;
    when X"0A" = >
        rec_clr < = '0';
        if cnt < 20 then
            cnt < = cnt + 1;
        else
            cnt < = 0;
            state < = X"0B";
            send_data < = CMD55;
        end if;
    when X"0B" = >
        spi_cs_n < = '0';
        spi_mosi < = send_data(47);
        send_data < = send_data(46 downto 0) & '0';
        if cnt < 47 then
            cnt < = cnt + 1;
        else
            cnt < = 0;
            state < = X"0C";
        end if;
    when X"0C" = >
        spi_mosi < = '1';
        if cnt < 100 then
            if rec_data(7) = '0'then
                rec_en < = '0';
                state < = X"0D";
                spi_cs_n < = '1';
                cnt < = 0;
            else
                rec_en < = '1';
                cnt < = cnt + 1;
            end if;
        else
            cnt < = 0;
            state < = X"0A";
            rec_en < = '0';
            spi_cs_n < = '1';
        end if;
    when X"0D" = >
        rec_en < = '0';
        spi_cs_n < = '1';
        rec_clr < = '1';
        if rec_data(7 downto 0) = X"01" then
            state < = X"0E";
        else
            state < = X"0A";
        end if;
    when X"0E" = >
        rec_clr < = '0';
        if cnt < 20 then
            cnt < = cnt + 1;
```

```
        else
            cnt <= 0;
            state <= X"0F";
            send_data <= ACMD41;
        end if;
    when X"0F" =>
        spi_cs_n <= '0';
        spi_mosi <= send_data(47);
        send_data <= send_data(46 downto 0) & '0';
        if cnt < 47 then
            cnt <= cnt + 1;
        else
            cnt <= 0;
            state <= X"10";
        end if;
    when X"10" =>
        spi_mosi <= '1';
        if cnt < 100 then
            if rec_data(7) = '0' then
                rec_en <= '0';
                state <= X"11";
                spi_cs_n <= '1';
                cnt <= 0;
            else
                rec_en <= '1';
                cnt <= cnt + 1;
            end if;
        else
            cnt <= 0;
            state <= X"0E";
            rec_en <= '0';
            spi_cs_n <= '1';
        end if;
    when X"11" =>
        rec_en <= '0';
        spi_cs_n <= '1';
        rec_clr <= '1';
        if rec_data(7 downto 0) = X"00" then
            state <= X"12";
        else
            state <= X"0A";
        end if;
    when X"12" =>
        rec_clr <= '0';
        if cnt < 20 then
            cnt <= cnt + 1;
        else
            cnt <= 0;
            state <= X"13";
        end if;
    when X"13" =>
```

```
            state <= X"14";
            send_data <= CMD17;
        when X"14" =>
            spi_cs_n <= '0';
            spi_mosi <= send_data(47);
            send_data <= send_data(46 downto 0) & '0';
            if cnt < 47 then
                cnt <= cnt + 1;
            else
                cnt <= 0;
                state <= X"15";
            end if;
        when X"15" =>
            spi_mosi <= '1';
            if cnt < 100 then
                if rec_data(7) = '0' then
                    rec_en <= '0';
                    state <= X"16";
                    spi_cs_n <= '0';
                    cnt <= 0;
                else
                    rec_en <= '1';
                    cnt <= cnt + 1;
                end if;
            else
                cnt <= 0;
                state <= X"12";
                rec_en <= '0';
                spi_cs_n <= '1';
            end if;
        when X"16" =>
            rec_en <= '0';
            spi_cs_n <= '0';
            rec_clr <= '1';
            if rec_data(7 downto 0) = X"00" then
                state <= X"17";
            else
                state <= X"12";
            end if;
        when X"17" =>
            rec_clr <= '0';
            rec_en <= '1';
            if rec_data(7 downto 0) = X"FE" then
                state <= X"18";
                read_en <= '1';
            else
                state <= X"17";
            end if;
        when X"18" =>
            if cnt < 4095 then
                cnt <= cnt + 1;
```

```
                    else
                        cnt <= 0;
                        read_en <= '0';
                        rec_en <= '0';
                        state <= X"19";
                    end if;
                when X"19" =>
                    rec_clr <= '1';
                    if cnt < 100 then
                        cnt <= cnt + 1;
                    else
                        cnt <= 0;
                        state <= X"1A";
                    end if;
                when X"1A" =>
                    rec_clr <= '0';
                    spi_cs_n <= '1';
                    if offset_addr < OFF_MAX - 1 then
                        offset_addr <= offset_addr + 1;
                        state <= X"12";
                    else
                        state <= X"1B";
                    end if;
                when X"1B" =>
                    read_done <= '1';
                    state <= X"1B";
                when others => state <= X"00";
            end case;
        end if;
    end if;
end process;
process(offset_addr) begin
    sd_addr <= ADDR + offset_addr;
end process;
CMD17(47 downto 40) <= X"51";
CMD17(39 downto 8) <= std_logic_vector(to_unsigned(sd_addr, 32));
CMD17(7 downto 0) <= X"FF";
end architecture one;
```

5.8 SDRAM 控制器设计

SDRAM(Synchronous Dynamic Random Access Memory),中文名为同步动态随机存储器。同步是指 Memory 工作需要同步时钟,内部命令的发送与数据的传输都以它为基准;动态是指存储阵列需要不断地刷新来保证数据不丢失;随机是指数据不是线性依次存储,而是自由指定地址进行数据读写。SDRAM 在图像处理、大数据处理等领域有着广泛的应用。

DE2-115 平台所使用的 SDRAM 芯片型号为 IS42S16320B-7TL(8M×16bit×4Bank),其内部结构如图 5.36 所示。由图 5.36 可以知道该 SDRAM 有 16bit 双向数据总线接口,即读写都是通过这 16bit 的数据总线。还可以看出其中包含 4 个 M-Bank(Memory Cell

Array Bank，存储阵列块），每个 M-Bank 包含 8M 个存储单元，即整块 SDRAM 的存储大小为 4×8M×16bit＝512Mbit 的容量。SDRAM 的内部结构还包括许多其他模块，如自刷新定时器、内部行地址计数、行地址和列地址解码器、列地址累加器、模式寄存器、突发计数器、状态机和地址缓存器。

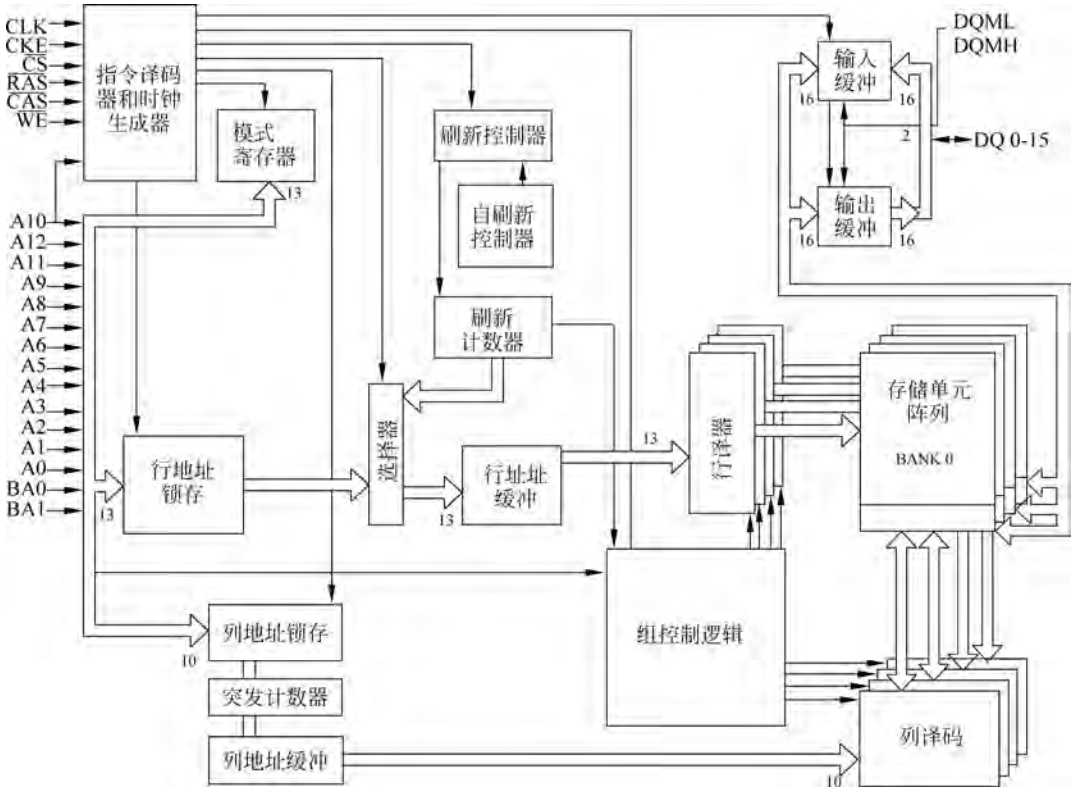

图 5.36 SDRAM(8M×16bit×4Bank)功能框图

5.8.1 SDRAM 引脚、命令和模式寄存器介绍

IS42S16320B 引脚功能如表 5.8 所示。

表 5.8 SDRAM(IS42S16320B)引脚功能描述

引脚名称	功能描述	引脚名称	功能描述
A0～A12	行地址输入	$\overline{WE}$	写使能
A0～A9	列地址输入	DQML	×16 低字节掩码(高阻)
BA0、BA1	Bank 选择地址	DQMH	×16 高字节掩码(高阻)
DQ0～DQ15	数据 I/O	VDD	供电
CLK	系统时钟输入	VSS	地
CKE	时钟使能	VDDQ	I/O 供电
$\overline{CS}$	片选	VSSQ	I/O 供电地
$\overline{RAS}$	行地址选通命令	NC	不连接
$\overline{CAS}$	列地址选通命令		

SDRAM 的控制通常是使用厂家给定的一系列命令来实现的。因此 SDRAM 内部有一个命令解码器，SDRAM 的初始化、读写等操作实际上就是将命令发送给解码器，通过解码器解码后告知 SDRAM，表 5.7 中给出 SDRAM 的基本操作命令。

表 5.9 SDRAM 命令集合表

命令	缩写	$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	$\overline{WE}$	A10
器件未选	DESL	H	X	X	X	X
空操作	NOP	L	H	H	H	X
激活操作	ACT	L	L	H	H	X
读操作	RD	L	H	L	H	L
读操作带预充电	RDPr	L	H	L	H	H
写操作	WR	L	H	L	L	L
写操作带预充电	WRPr	L	H	L	L	H
突发终止	BST	L	H	H	L	X
被选 Bank 预充电	PRE	L	L	H	L	L
所有 Bank 预充电	PALL	L	L	H	L	H
CBR 自动刷新 CKE=H	REF	L	L	L	H	X
自刷新 CKE=L	SELF	L	L	L	H	X
配置模式寄存器	LMR	L	L	L	L	L

SDRAM 内部有一个非常重要的模式寄存器(Mode Register，MR)。模式寄存器的地址总线定义如表 5.10 所示。

表 5.10 模式寄存器的地址总线定义

地　址	说　明	描　述
BA0/1、A12～A10	保留	写入 0
A9	写突发模式(M9)	0：编程突发模式；1：单一入口地址
A8、A7	操作模式(M8M7)	00：标准模式；其他：保留
A6、A5、A4	潜伏模式(M6M5M4)	010：潜伏期 2；011：潜伏期 3；其他：保留
A3	突发模式(M3)	0：顺序；1：间隔
A2、A1、A0	突发长度 BL	000：1；001：2；010：4；011：8；111：全页

突发长度(BL)的值，即连续读几列地址。所谓的全页操作是指对同一个 Bank 中同一个行地址进行操作，对于 x16 模式来说相当于连续操作 512 个字。

由于 SDRAM 的寻址具有独占性，所以在进行完读写操作后，如果要对同一 L-Bank 的另一行进行寻址，就要将原来有效(工作)的行关闭，重新发送行/列地址。L-Bank 关闭现有工作行，准备打开新行的操作就是预充电(Precharge)。预充电可以通过命令控制，也可以通过辅助设定让芯片在每次读写操作之后自动进行预充电。实际上，预充电是对工作行中所有存储体进行数据重写，并对行地址进行复位，同时释放 S-AMP(重新加入比较电压，一般是电容电压的 1/2，以帮助判断读取数据的逻辑电平，因为 S-AMP 是通过一个参考电压与存储体的位线电压的比较来判断逻辑值的)，以准备新行的工作。具体而言，就是将 S-AMP 中的数据回写，即使是没有工作过的存储体也会因行选通而使存储电容受到干扰，所以也需要 S-AMP 进行读后重写。此时，电容的电量(或者说其产生的电压)将是判断逻

辑状态的依据(读取时也需要),为此要设定一个临界值,一般为电容电量的 1/2,超过它的为逻辑 1,进行重写,否则为逻辑 0,不进行重写(等于放电)。为此,现在基本都将电容的另一端接入一个指定的电压(即 1/2 电容电压),而不是接地,以帮助重写时的比较与判断。从表 5.9 可以发现地址线 A10 控制着读写之后当前 Bank 是否自动进行预充电。而在单独的预充电命令中,A10 则控制着是对指定的 Bank 还是所有的 Bank(当有多个 Bank 处于有效/活动状态时)进行预充电,前者需要提供 Bank 的地址,后者只需将 A10 信号置于高电平。

5.8.2 SDRAM 初始化

SDRAM 要想正确地工作需要正确配置其模式寄存器以使其按照预期的方式进行工作。SDRAM 的初始化就是在上电后对模式寄存器进行配置的过程。SDRAM 的初始化有着严格的流程规定,如图 5.37 所示。

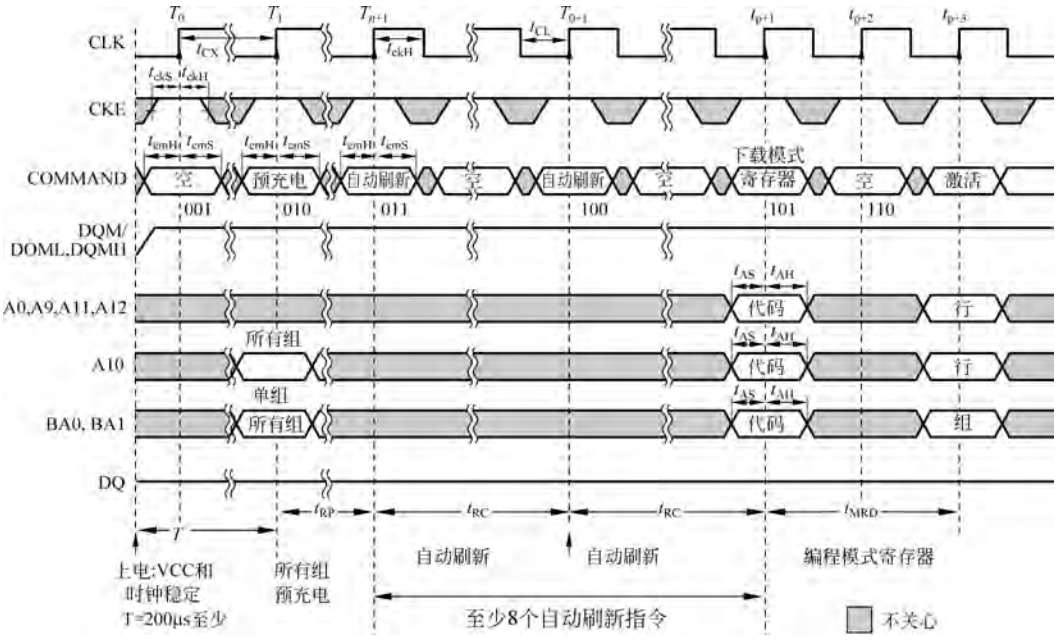

图 5.37 SDRAM 初始化时序

从图 5.37 得出 SDRAM 上电序列,初始化代码如例 5.25 所示。代码中命令 $\overline{CS}$、$\overline{RAS}$、$\overline{CAS}$、$\overline{WE}$ 组合为 4 位逻辑位矢量,代码以状态机方式设计,下面对状态加以说明。

(1) 状态 X"0":命令为禁用("1111");

(2) 状态 X"1":根据图 5.37 上电后需等待至少 200μs,此时命令采用空操作命令 NOP ("0111");

(3) 状态 X"2":给出所有 Bank 预充电命令,此时命令采用 PALL 命令("0010")给所有 Bank 预充电时要求 A10 为高电平,代码中采用 sdr_addr(10) <='1'语句完成 A10 赋值;

(4) 状态 X"3":给出两个 NOP 后设置为 CBR 自动刷新(CKE 端口始终设置为 H),REF 命令为("0001");

(5) 状态 X"4":给出两个 NOP 后设置为 CBR 自动刷新(CKE 端口始终设置为 H),

REF 命令为(“0001”)；

(6) 状态 X“5”：给出 8 个 NOP 后配置模式寄存器 MR，命令 LMR 为(“0000”)；Bank 设置为“00”，MR 设置为"0000000100111"，含义为潜伏期 2，全页顺序突发模式。

(7) 状态 X“6”：给出 3 个 NOP 后给出初始化完成标志 init_done <= '1'；初始化任务完成。

【例 5.25】 SDRAM 初始化。

```
library ieee;
use ieee.std_logic_1164.all;
entity sdr_init is
    port (
        clk       :  in   std_logic;
        rst_n     :  in       std_logic;
        init_done :  out  std_logic;
        init_bus  :  out       std_logic_vector(18 downto 0)
    );
end entity sdr_init;
architecture one of sdr_init is
    signal sdr_cmd   :  std_logic_vector(3 downto 0);
    signal sdr_bank  :  std_logic_vector(1 downto 0);
    signal sdr_addr  :  std_logic_vector(12 downto 0);
    signal state     :  std_logic_vector(2 downto 0);
    signal cnt       :  integer;
begin
    init_bus(18 downto 15) <= sdr_cmd;
    init_bus(14 downto 13) <= sdr_bank;
    init_bus(12 downto 0) <= sdr_addr;
    process(clk) begin
        if clk'event and clk = '1' then
            if rst_n = '0' then
                sdr_cmd <= X"F";
                state <= "000";
                cnt <= 0;
                sdr_addr <= (others => '0');
                sdr_bank <= "00";
                init_done <= '0';
            else
                case state is
                    when "000" =>
                        sdr_cmd <= X"F";                    -- inh
                        state <= "001";
                        cnt <= 0;
                        sdr_addr <= (others => '0');
                        sdr_bank <= "00";
                        init_done <= '0';
                    when "001" =>
                        if cnt < 10000 then
                            cnt <= cnt + 1;
                            sdr_cmd <= X"7";                -- nop
```

```
                    else
                        cnt <= 0;
                        state <= "010";
                    end if;
                when "010" =>
                    sdr_cmd <= X"2";                -- PREC
                    sdr_addr(10) <= '1';
                    state <= "011";
                when "011" =>
                    if cnt < 2 then
                        cnt <= cnt + 1;
                        sdr_cmd <= X"7";            -- nop
                    else
                        cnt <= 0;
                        sdr_cmd <= X"1";            -- ref
                        state <= "100";
                    end if;
                when "100" =>
                    if cnt < 7 then
                        cnt <= cnt + 1;
                        sdr_cmd <= X"7";            -- nop
                    else
                        cnt <= 0;
                        sdr_cmd <= X"1";            -- ref
                        state <= "101";
                    end if;
                when "101" =>
                    if cnt < 7 then
                        cnt <= cnt + 1;
                        sdr_cmd <= X"7";            -- nop
                    else
                        cnt <= 0;
                        sdr_cmd <= X"0";            -- LMR
                        sdr_addr <= "0000000100111";
                        sdr_bank <= "00";
                        state <= "110";
                    end if;
                when "110" =>
                    if cnt < 3 then
                        cnt <= cnt + 1;
                        sdr_cmd <= X"7";            -- nop
                    else
                        init_done <= '1';
                    end if;
                when others => state <= "000";
            end case;
        end if;
    end if;
  end process;
end architecture one;
```

5.8.3 SDRAM 读写操作

1. SDRAM 写操作

SDRAM 可以通过地址的检索实现单个数据的读写，也可以通过突发读写来实现数据的连续操作。为了实现更高的速率，一般采用突发读写的方式来实现海量数据的高速读写。这也是 SDRAM 控制器的读写实现方式。以突发长度 BL＝全页，读潜伏期 CL＝2，响应延时 t_{RCD}＝2Clock 为例的 SDRAM 的突发写时序图如图 5.38 所示。突发(Burst)是指在同一行中相邻的存储单元连续进行数据传输的方式，连续传输的周期数就是突发长度，寻址与数据的读写将自动进行连续操作，由图可知，全页模式可连续操作 512 个地址(x16 模式)。用户只需要控制好两段突发读写命令的间隔周期即可做到连续的突发传输。

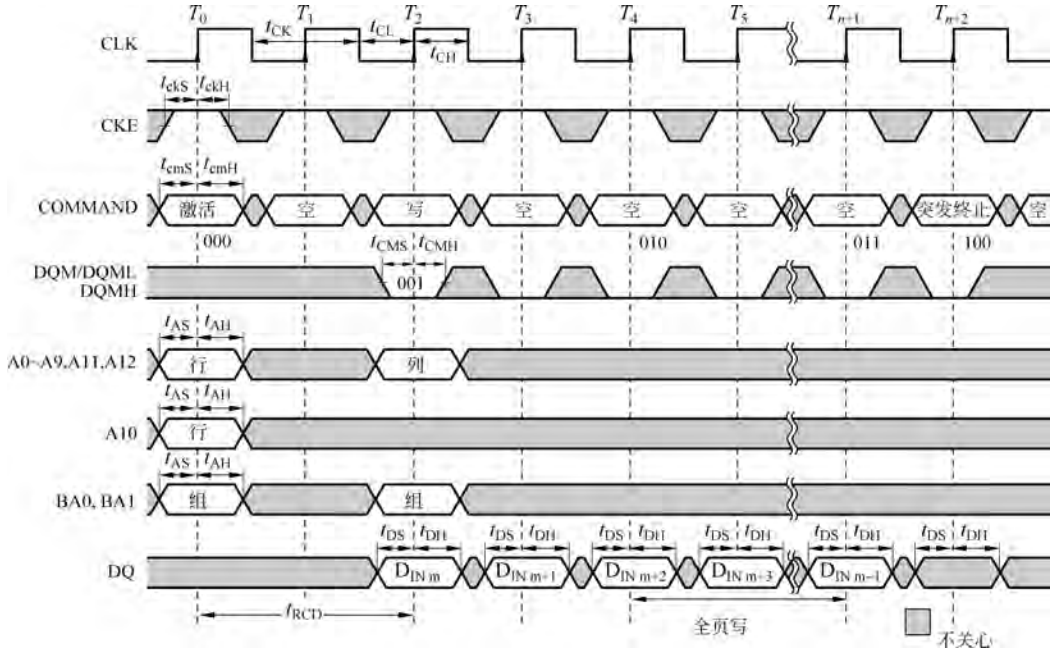

图 5.38 SDRAM 突发写时序图

在高速的图像数据缓存过程中，设置为顺序突发读写方式，同时以全页读写(数据长度最长为 512×16bit，本例为 320×16bit)的方式进行突发读写将能够得到更大的带宽，更高的效率。由于 SDRAM 的时钟为 100MHz，进行突发读写的前提是每次进行读写时都必须要准备好 320 个数据的缓存器，以保证这些数据的读写能够快速而准确地连续进行，这样就需要为 SDRAM 读写配备异步 FIFO。

从图 5.38 得出 SDRAM 写操作序列，写操作代码如例 5.26 所示。此例数据来源于存储在 SD 卡中 640×480×16bit 图像数据。每次从 SD 卡中读取 320 个字(半行图像数据)存储在 SDRAM 中，共读取 960 次完成全部图像数据的读取(320×2×480×16bit)。该模块主要结构也采用状态机结构完成写操作，状态“000”～“100”与图 5.39 序列一致，状态“101”～“111”为下一次突发写操作做准备，下面对各个操作状态加以详细说明。

(1) 状态 X“0”：命令为激活操作 ACT(“0011”)；给出 Bank 为“00”，行地址为外部端口 wr_addr 赋值。

(2) 状态 X"1": 根据图 5.38 此时先给出空操作命令 NOP("0111")后,给预充电写操作 WRPr("0100"),此时 A10 由 sdr_addr(10) <='1'赋值,列地址从 0 开始。

(3) 状态 X"2": 突发模式连续读取 319 个数据。

(4) 状态 X"3": 存储第 320 个数据。

(5) 状态 X"4": 给出突发终止命令 BST("0110")。

(6) 状态 X"5": 给出空操作命令 NOP("0111")。

(7) 状态 X"6": 给出所有 Bank 预充电命令,此时命令采用 PALL 命令("0010"),给所有 Bank 预充电时要求 A10 为高电平,代码中采用 sdr_addr(10) <='1'语句完成 A10 赋值。

(8) 状态 X"7": 给出两个 NOP 后给出完成写标志 wr_done <='1'; 第一块写任务完成。

【例 5.26】 SDRAM 突发写操作。

```
library ieee;
use ieee.std_logic_1164.all;
entity sdr_write is
    port(
        clk           :        in       std_logic;
        rst_n         :        in          std_logic;
        write_bus     :        out      std_logic_vector(18 downto 0);
        sdr_dq        :        inout      std_logic_vector(15 downto 0);
        wr_fifo_rdreq :        out      std_logic;
        wr_fifo_rdata :        in          std_logic_vector(15 downto 0);
        wr_addr       :        in          std_logic_vector(12 downto 0);
        wr_done       :        out      std_logic
    );
end entity sdr_write;
architecture one of sdr_write is
    signal     flag       :        std_logic;
    signal     dq_buf     :        std_logic_vector(15 downto 0);
    signal     cmd        :        std_logic_vector(3 downto 0);
    signal     sdr_bank   :        std_logic_vector(1 downto 0);
    signal     sdr_addr   :        std_logic_vector(12 downto 0);
    signal     state      :        std_logic_vector(3 downto 0) : = X"0";
    signal     cnt        :        integer : = 0;
begin
    process(flag, dq_buf) begin
        if flag = '1' then
            sdr_dq < = dq_buf;
        else
            sdr_dq < = (others = > 'Z');
        end if;
    end process;
    write_bus(18 downto 15) < = cmd;
    write_bus(14 downto 13) < = sdr_bank;
    write_bus(12 downto 0) < = sdr_addr;
    process(clk) begin
        if clk'event and clk = '1' then
            if rst_n = '0' then
                state < = X"0";
```

```
        cnt <= 0;
        wr_done <= '0';
        wr_fifo_rdreq <= '0';
        cmd <= x"7";                                -- nop
        sdr_bank <= "00";
        sdr_addr <= (others => '0');
        flag <= '0';
        dq_buf <= X"0000";
    else
        case state is
            when X"0" =>
                cmd <= X"3";                        -- act
                sdr_addr <= wr_addr;
                sdr_bank <= "00";
                wr_fifo_rdreq <= '1';
                state <= X"1";
            when X"1" =>
                if cnt < 1 then
                    cmd <= X"7";                    -- NOP
                    cnt <= cnt + 1;
                else
                    cnt <= 0;
                    cmd <= X"4";                    -- wr
                    sdr_addr(10) <= '1';
                    sdr_addr(12 downto 11) <= "00";
                    sdr_addr(9 downto 0) <= (others => '0');
                    flag <= '1';
                    dq_buf <= wr_fifo_rdata;
                    state <= X"2";
                end if;
            when X"2" =>
                cmd <= X"7";                        -- nop
                dq_buf <= wr_fifo_rdata;
                if cnt < 317 then
                    cnt <= cnt + 1;
                else
                    cnt <= 0;
                    wr_fifo_rdreq <= '0';
                    state <= X"3";
                end if;
            when X"3" =>
                dq_buf <= wr_fifo_rdata;
                state <= X"4";
            when X"4" =>
                cmd <= X"6";                        -- bt
                state <= X"5";
            when X"5" =>
                cmd <= X"7";                        -- nop
                flag <= '0';
                state <= X"6";
            when X"6" =>
```

```
                cmd <= X"2";                    -- prec
                sdr_addr(10) <= '1';
                state <= X"7";
            when X"7" =>
                if cnt < 2 then
                    cmd <= X"7";                -- nop
                    cnt <= cnt + 1;
                else
                    cmd <= X"7";                -- nop
                    wr_done <= '1';
                    state <= X"7";
                end if;
            when others => state <= X"0";
        end case;
    end if;
  end if;
 end process;
end architecture one;
```

2. SDRAM 读操作

SDRAM 突发读时序图如图 5.39 所示。从图 5.39 得出 SDRAM 读操作序列，读操作代码如例 5.27 所示。读操作与写操作代码结构相似。该模块也采用状态机结构完成写操作，状态“000”～“100”与图 5.39 序列一致，状态“101”～“111”为下一次突发写操作做准备。下面对各个操作状态加以详细说明。

(1) 状态 X“0”：命令为激活操作 ACT(“0011”)；给出 Bank 为“00”，行地址为外部端口 rd_addr 赋值。

(2) 状态 X“1”：根据图 5.39 此时先给出一个空操作命令 NOP(“0111”)后，给带预充

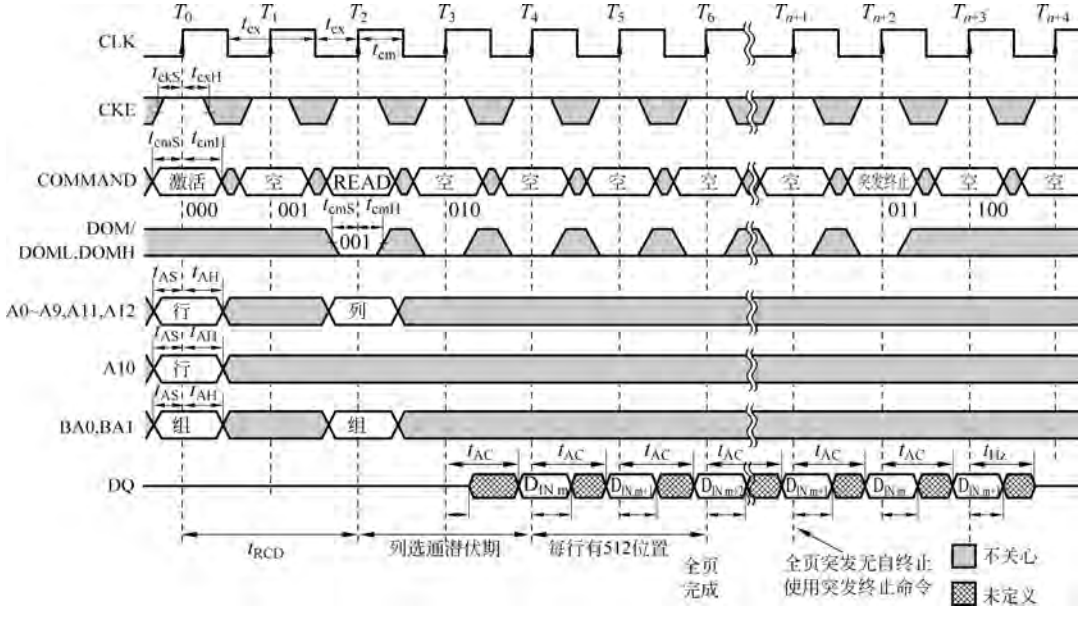

图 5.39　SDRAM 突发读时序图

电写操作 RDPr(“0101”),此时 A10 由 sdr_addr(10) <='1'赋值,列地址从 0 开始。

(3) 状态 X“2”:给出两个空操作命令 NOP(“0111”)后,置读 FIFO 写请求标志。

(4) 状态 X“3”:突发模式连续读取 319 个数据后给出突发终止命令 BST(“0110”)。

(5) 状态 X“4”:存储第 320 个数据后给出空操作命令 NOP(“0111”)。

(6) 状态 X“5”:清读 FIFO 写请求标志。

(7) 状态 X“6”:给出所有 Bank 预充电命令,此时命令采用 PALL 命令(“0010”),给所有 Bank 预充电时要求 A10 为高电平,代码中采用 sdr_addr(10) <='1'语句完成 A10 赋值。

(8) 状态“111”给出两个空操作命令 NOP(“0111”)后,给出完成读标志 rd_done <='1';第一行读任务完成。

【例 5.27】 SDRAM 突发读操作。

```
library ieee;
use ieee.std_logic_1164.all;
entity sdr_read is
    port (
        clk           :    in            std_logic;
        rst_n         :    in            std_logic;
        rd_bus        :    out           std_logic_vector(18 downto 0);
        sdr_dq        :    inout         std_logic_vector(15 downto 0);
        rd_addr       :    in            std_logic_vector(12 downto 0);
        rd_fifo_wdata :    out           std_logic_vector(15 downto 0);
        rd_fifo_wrreq :    out           std_logic;
        rd_done       :    out           std_logic
    );
end entity sdr_read;
architecture one of sdr_read is
    signal     flag             :        std_logic;
    signal     dq_buf           :        std_logic_vector(15 downto 0);
    signal     cmd              :        std_logic_vector(3 downto 0);
    signal     sdr_bank         :        std_logic_vector(1 downto 0);
    signal     sdr_addr         :        std_logic_vector(12 downto 0);
    signal     state            :        std_logic_vector(3 downto 0) : = X"0";
    signal     cnt              :        integer : = 0;
begin
    process(flag, dq_buf) begin
        if flag = '1' then
            sdr_dq < = dq_buf;
        else
            sdr_dq < = (others = > 'Z');
        end if;
    end process;
    rd_bus(18 downto 15) < = cmd;
    rd_bus(14 downto 13) < = sdr_bank;
    rd_bus(12 downto 0) < = sdr_addr;
    process(clk) begin
        if clk'event and clk = '1' then
            if rst_n = '0' then
                cmd < = X"7";                                    -- nop
```

```
        sdr_bank <= "00";
        sdr_addr <= (others => '0');
        flag <= '0';
        dq_buf <= (others => '0');
        rd_fifo_wdata <= X"0000";
        rd_fifo_wrreq <= '0';
        rd_done <= '0';
        state <= X"0";
        cnt <= 0;
    else
        case state is
            when X"0" =>
                cmd <= X"3";                          -- act
                sdr_bank <= "00";
                sdr_addr <= rd_addr;
                state <= X"1";
                flag <= '0';
            when X"1" =>
                if cnt < 1 then
                    cmd <= X"7";                      -- nop
                    cnt <= cnt + 1;
                else
                    cmd <= X"5";                      -- rd
                    sdr_addr(10) <= '1';
                    sdr_addr(12 downto 11) <= "00";
                    sdr_addr(9 downto 0) <= (others => '0');
                    cnt <= 0;
                    state <= X"2";
                end if;
            when X"2" =>
                if cnt < 2 then
                    cmd <= X"7";                      -- nop
                    cnt <= cnt + 1;
                else
                    rd_fifo_wdata <= sdr_dq;
                    rd_fifo_wrreq <= '1';
                    cnt <= 0;
                    state <= X"3";
                end if;
            when X"3" =>
                rd_fifo_wdata <= sdr_dq;
                rd_fifo_wrreq <= '1';
                if cnt < 317 then
                    cnt <= cnt + 1;
                else
                    cnt <= 0;
                    cmd <= X"6";
                    state <= X"4";
                end if;
            when X"4" =>
                rd_fifo_wdata <= sdr_dq;
                rd_fifo_wrreq <= '1';
                cmd <= X"7";                          -- nop
                state <= X"5";
```

```
                when X"5" = >
                    rd_fifo_wrreq < = '0';
                    state < = X"6";
                when X"6" = >
                    cmd < = X"2";                          -- prec
                    sdr_addr(10) < = '1';
                    state < = X"7";
                when X"7" = >
                    if cnt < 2 then
                        cnt < = cnt + 1;
                        cmd < = X"7";                      -- nop
                    else
                        cmd < = X"7";
                        rd_done < = '1';
                        state < = X"7";
                    end if;
                when others = > state < = X"0";
            end case;
        end if;
    end if;
end process;
end architecture one;
```

5.8.4 SDRAM 自动刷新时序

SDRAM 的 Bank 逻辑单元是电容型结构，电容易掉电，因此要及时充电。若一段时间不充电，电会放完，数据也就丢失了。因此 SDRAM 的这一物理特性决定了必须不停地刷新或者预刷新，手册要求必须每 64ms 对所有的行、列进行一次刷新以确保数据的完整。也就是说每行刷新的循环周期是 64ms，这样刷新速度就是：行数量/64ms。手册中 4096 Refresh Cycles/64ms 或 8192RefreshCycles/64ms 的标识，4096 行刷新间隔为 15.625μs，8192 行时就为 7.8125μs。

刷新操作分为两种：自动刷新(Auto Refresh，REF)和自刷新(self Refresh，SELF R)。不论是何种刷新方式，都不需要外部提供行地址信息，对于 REF，SDRAM 内部有一个行地址生成器(也称刷新计数器)，用来自动地依次生成行地址，由于刷新是针对一行中的所有存储体进行，所以无须列寻址，或者说 CAS 在 RAS 之前有效。所以 REF 又称 CBR(CAS Before RAS，列提前于行定位)式刷新。由于刷新涉及所有 Bank，因此在刷新过程中，所有 Bank 都停止工作，所有工作指令只能等待而无法执行，刷新之后就可进入正常的工作状态。64ms 之后需再次对同一行进行刷新，因此循环刷新操作会对 SDRAM 的性能造成影响，这也是 DRAM 相对于 SRAM 牺牲性能获取了成本优势。刷新时序如图 5.40 所示。刷新范例如例 5.28 所示，例中采用自动刷新模式设计。

例 5.28 SDRAM 自动刷新模式也是根据图 5.40 中各个命令顺序关系利用状态机完成设计。下面对各个操作状态加以详细说明。

(1) 状态 X“0”：给出所有 Bank 预充电命令，此时命令采用 PALL 命令(“0010”)，给所有 Bank 预充电时要求 A10 为高电平，代码中采用 sdr_addr(10) <='1'语句完成 A10 赋值；

(2) 状态 X“1”：根据图 5.40，此时先给出两个空操作命令 NOP(“0111”)后，设置为 CBR 自动刷新(CKE 端口始终设置为 H)，REF 命令为(“0001”)；

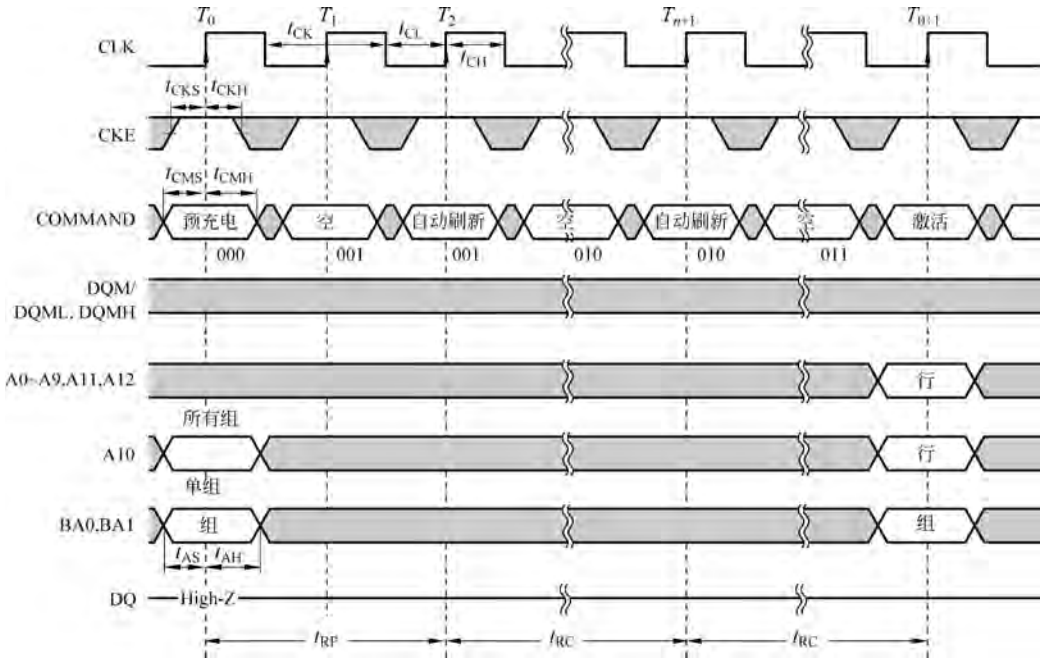

图 5.40　SDRAM 自动刷新时序图

(3) 状态 X"2"：先给出 8 个空操作命令 NOP("0111")后，设置为 CBR 自动刷新(CKE 端口始终设置为 H)，REF 命令为("0001")；此处设置 8 个空操作命令是根据初始化时序要求设置 t_{RC}；

(4) 状态 X"3"：先给出 8 个空操作命令 NOP("0111")后，设置刷新结束标志位 refresh_done <= '1'。

【例 5.28】 SDRAM 自动刷新。

```
library ieee;
use ieee.std_logic_1164.all;
entity sdr_refresh is
    port(
        clk   :in  std_logic;
        rst_n:in  std_logic;
        refresh_bus:  out  std_logic_vector(18 downto 0);
        refresh_done:  out  std_logic
    );
end entity sdr_refresh;
architecture one of sdr_refresh is
    signal sdr_cmd :    std_logic_vector(3 downto 0);
    signal sdr_bank:    std_logic_vector(1 downto 0);
    signal sdr_addr:    std_logic_vector(12 downto 0);
    signal state   :    std_logic_vector(3 downto 0);
    signal cnt     :    integer;
begin
    refresh_bus(18 downto 15) <= sdr_cmd;
    refresh_bus(14 downto 13) <= sdr_bank;
    refresh_bus(12 downto 0) <= sdr_addr;
    process(clk) begin
```

```
        if clk'event and clk = '1' then
            if rst_n = '0'then
                state <= X"0";
                sdr_cmd <= X"7";                    -- nop
                cnt <= 0;
                sdr_addr <= (others => '0');
                sdr_bank <= "00";
                refresh_done <= '0';
            else
                case state is
                    when X"0" =>
                        sdr_cmd <= X"2";            -- prec
                        sdr_addr(10) <= '1';
                        state <= X"1";
                    when X"1" =>
                        if cnt < 2 then
                            sdr_cmd <= X"7";        -- nop
                            cnt <= cnt + 1;
                        else
                            sdr_cmd <= X"1";        -- ref
                            cnt <= 0;
                            state <= X"2";
                        end if;
                    when X"2" =>
                        if cnt < 7 then
                            cnt <= cnt + 1;
                            sdr_cmd <= X"7";        -- nop
                        else
                            cnt <= 0;
                            sdr_cmd <= X"1";        --ref
                            state <= X"3";
                        end if;
                    when X"3" =>
                        if cnt < 7 then
                            cnt <= cnt + 1;
                            sdr_cmd <= X"7";        -- nop
                        else
                            state <= X"3";
                            refresh_done <= '1';
                        end if;
                    when others => state <= X"0";
                end case;
            end if;
        end if;
    end process;
end architecture one;
```

5.8.5 SDRAM 控制器

前面章节介绍了 SDRAM 主要的操作设计方法和代码案例，本节将结合前面章节内容继续利用状态机完成 SDRAM 控制器的整体设计。控制器 RTL 视图如图 5.41 所示。图中 timer 模块主要是产生自动刷新触发信号，每 4μs 产生一次刷新触发信号。实际上就是

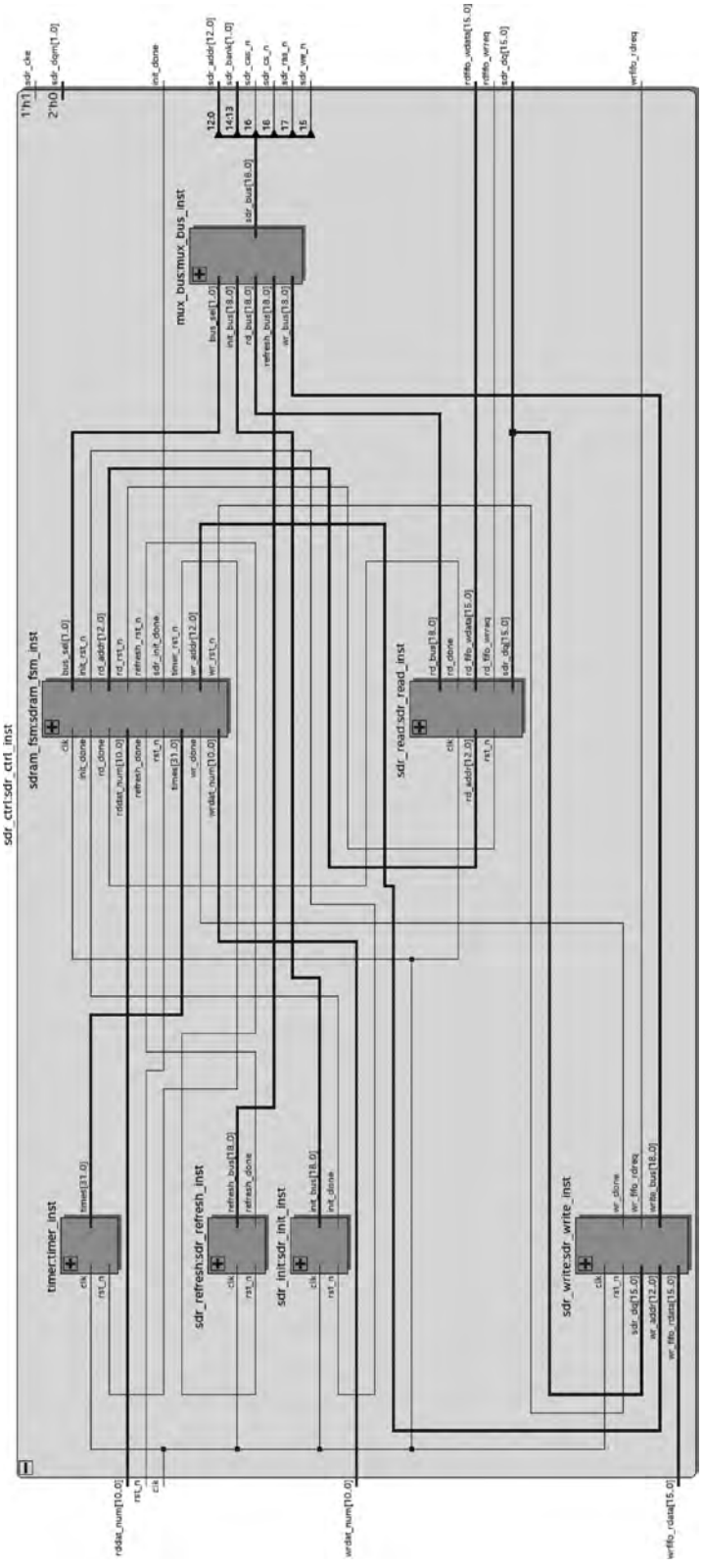

图 5.41　SDRAM 自动刷新时序图

一个计数器计数时钟由锁相环产生的100MHz作为计数时钟源。Mux_bus为总线选择器，实际上就是一个4选1数据选择器，通过Sdram_fsm控制器状态机产生的总线选择端选择不同的总线命令传输到SDRAM端口。总线命令来自于前面章节介绍的SDRAM初始化操作总线、写操作总线、读操作总线和刷新操作总线。例5.29为总线状态机模块，总线状态机主要用来调度各个模块协同完成SDRAM读写、初始化和刷新操作。下面对例5.29中各个状态设计进行说明。

(1) 状态X"0"：选择初始化模块总线等待初始化完成标志init_done，初始化完成后复位初始化模块同时跳转到状态X"1"并使能刷新计数器；

(2) 状态X"1"：选择刷新总线等待刷新结束，结束后跳转到状态X"2"；

(3) 状态X"2"：判断是否到刷新时间，刷新时间到跳转到状态X"1"，否则判断写FIFO数据是否少于150个，如果少则跳转到读操作状态X"3"、X"4"进行读取SDRAM到写FIFO中，如果读FIFO中多于340个数据，则跳转到状态X"5"、X"6"将FIFO模块数据读取到SDRAM中。状态机的状态转移图如图5.42所示。

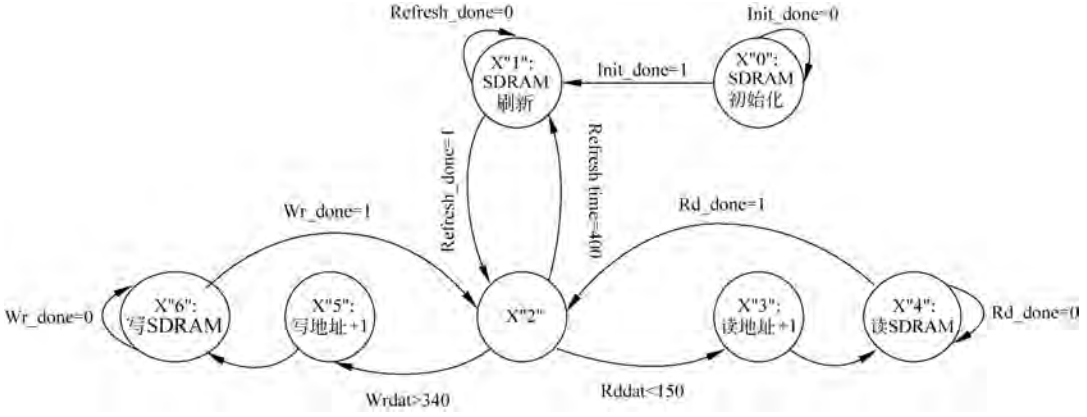

图5.42 SDRAM控制器状态转移图

【例5.29】 SDRAM控制器。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity sdram_fsm is
    port (
        clk          :    in    std_logic;
        rst_n        :    in    std_logic;
        init_rst_n   :    out   std_logic;
        timer_rst_n  :    out   std_logic;
        refresh_rst_n:    out   std_logic;
        wr_rst_n     :    out   std_logic;
        rd_rst_n     :    out   std_logic;
        bus_sel      :    out   std_logic_vector(1 downto 0);
        wrdat_num    :    in    std_logic_vector(10 downto 0);
```

```
        rddat_num     :    in    std_logic_vector(10 downto 0);
        init_done     :    in    std_logic;
        refresh_done  :    in    std_logic;
        wr_done       :    in    std_logic;
        rd_done       :    in    std_logic;
        sdr_init_done :    out   std_logic;
        wr_addr       :    out   std_logic_vector(12 downto 0);
        rd_addr       :    out   std_logic_vector(12 downto 0);
        times         :    in    integer
    );
end entity sdram_fsm;
architecture one of sdram_fsm is
    signal     state  :          std_logic_vector(3 downto 0);
    signal     cnt    :          integer;
    signal     rdaddr :          integer : = 0;
    signal     wraddr :          integer : = 0;
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if rst_n = '0' then
                state < = X"0";
                init_rst_n < = '0';
                timer_rst_n < = '0';
                refresh_rst_n < = '0';
                wr_rst_n < = '0';
                rd_rst_n < = '0';
                bus_sel < = "00";
                wraddr < = 0;
                rdaddr < = 0;
                sdr_init_done < = '0';
            else
                case state is
                    when X"0" = >
                        if init_done = '0' then
                            init_rst_n < = '1';
                            bus_sel < = "00";              -- init_bus
                        else
                            init_rst_n < = '0';
                            bus_sel < = "01";
                            timer_rst_n < = '1';
                            state < = X"1";
                        end if;
                    when X"1" = >
                        if refresh_done = '0' then
                            timer_rst_n < = '1';
                            refresh_rst_n < = '1';
                        else
                            refresh_rst_n < = '0';
```

```
                state <= X"2";
                sdr_init_done <= '1';
            end if;
        when X"2" =>
            if times = 400 then
                timer_rst_n <= '0';
                bus_sel <= "01";                -- refresh_bus
                state <= X"1";
            else
                if conv_integer(rddat_num) < 150 then
                    bus_sel <= "11";            -- rd_bus
                    state <= X"3";
                else
                    if conv_integer(wrdat_num) > 340 then
                        bus_sel <= "10";        -- wr_bus
                        state <= X"5";
                    else
                        state <= X"2";
                    end if;
                end if;
            end if;
        when X"3" =>
            state <= X"4";
            if rdaddr < 960 then
                rdaddr <= rdaddr + 1;
            else
                rdaddr <= 1;
            end if;
        when X"4" =>
            if rd_done = '0' then
                rd_rst_n <= '1';
            else
                rd_rst_n <= '0';
                state <= X"2";
            end if;
        when X"5" =>
            state <= X"6";
            if wraddr < 960 then
                wraddr <= wraddr + 1;
            else
                wraddr <= 1;
            end if;
        when X"6" =>
            if wr_done = '0' then
                wr_rst_n <= '1';
            else
                wr_rst_n <= '0';
                state <= X"2";
            end if;
        when others => state <= X"0";
    end case;
```

```
            end if;
        end if;
    end process;
    wr_addr <= conv_std_logic_vector(wraddr, 13);
    rd_addr <= conv_std_logic_vector(rdaddr, 13);
end architecture one;
```

5.9 利用 VGA 接口显示 SD 卡图像数据

该工程主要原理框图如图 5.43 所示。FPGA 通过 SPI 接口读取存储在 SD 卡内的图像并先存入内部的 SDR_WrFIFO 里，通过 SDRAM 控制器将 SDR_WrFIFO 数据读取写入外部 SDRAM 中，直到完整图像数据存入 SDRAM 中。VGA 显示是利用 SDRAM 控制器将 SDRAM 的图像数据读到内部的 SDR_RdFIFO 中，再驱动 VGA 显示器把图像传输给 VGA 液晶屏上显示。由于无法将 SD 卡读取到的内容直接放到 VGA 驱动器中显示，需要经过 SDRAM 及 FIFO 做显示缓存。SD 卡中图像数据是特指 bin 格式的图像数据，图像格式为 640×480，每个像素宽度为 16bit，即 RGB565。用户将 bin 格式文件存储在 SD 卡中时需要利用 Winhex 软件查看文件所在扇区地址。用户在操作 SD 卡时需要用到该地址。

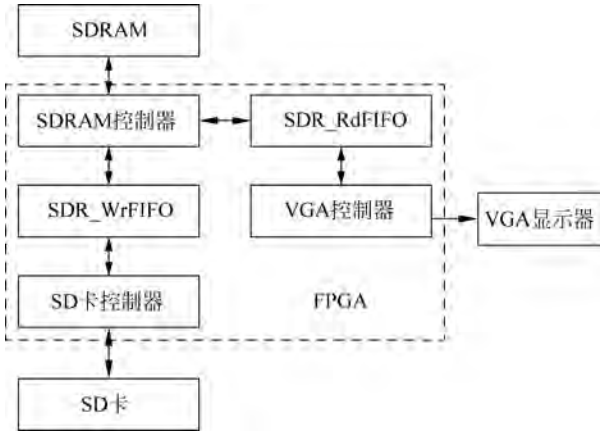

图 5.43 SD 卡图像数据 VGA 显示框图

整体工程 RTL 视图如图 5.44 所示，图中包含 4 个主要功能模块。

(1) Clock_reset_processor：专门用来处理系统时钟和复位信号。该模块利用锁相环 IP 完成，锁相环产生 3 个时钟输出。c0 输出时钟 100MHz，用于 sys_clk_100m 输出提供给 FPGA 内部 SDRAM 控制器时钟使用；c1 输出时钟 100MHz 相移 270°，用于片外 SDRAM 时钟使用；c2 输出时钟 25MHz，用于 SD 卡操作、VGA 时钟、FIFO 读写时钟。另外利用锁相环锁定信号产生了用于 100MHz 和 25MHz 工作模块的复位信号。

(2) Vga_ctrl：是采用 640×480 视频标准的 VGA 驱动器。数字信号转换为模拟信号(R、G、B)，采用高速视频 DAC 芯片 ADV7123 实现。

(3) Sd_init_read：用来初始化 SD 卡，并且从 SD 卡中读取数据。本例程的设计采用

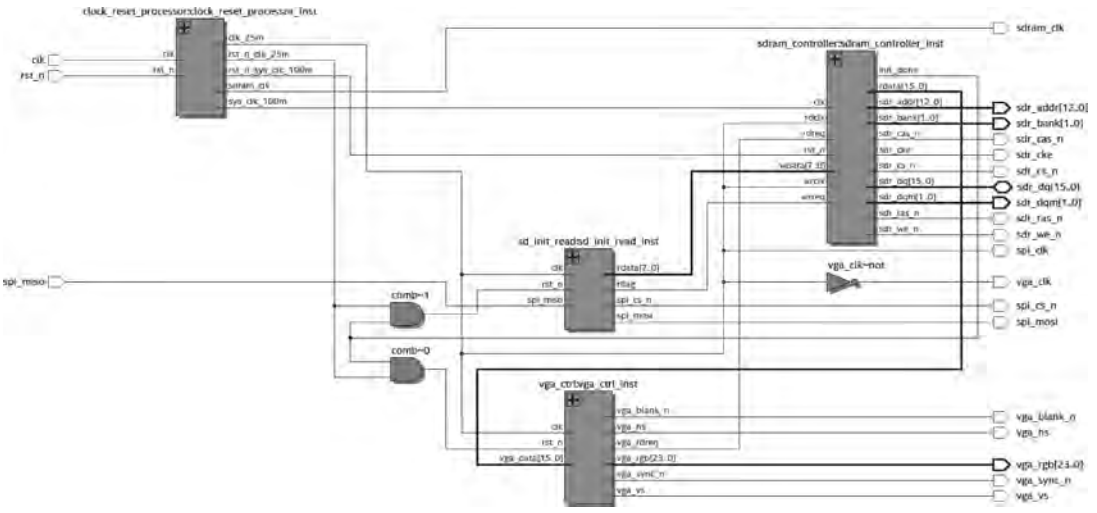

图 5.44　整体工程 RTL 视图

SDHC 的 SD 卡(8GB),SD 卡存储是按照扇区(512B)存储的。本模块利用 SPI 协议将数据读出,按照字节的方式进行输出。

(4) Sdram_controller：用来初始化 SDRAM,将 SD 卡的数据存入 SDRAM 中,并将 SDRAM 中的数据输出给 VGA。该模块内部集成了两个用于数据缓存的 FIFO,内部 RTL 视图如图 5.45 所示。由图 5.45 可知,sdr_wrfifo 用于缓存从 SD 卡读出的数据,通过 SDRAM 控制器将其数据缓存至 SDRAM 中,该 FIFO 配置如图 5.46 所示。sdr_rdfifo 用于缓存从 SDRAM 读出的数据,用于 VGA 显示使用,该 FIFO 配置如图 5.47 所示。

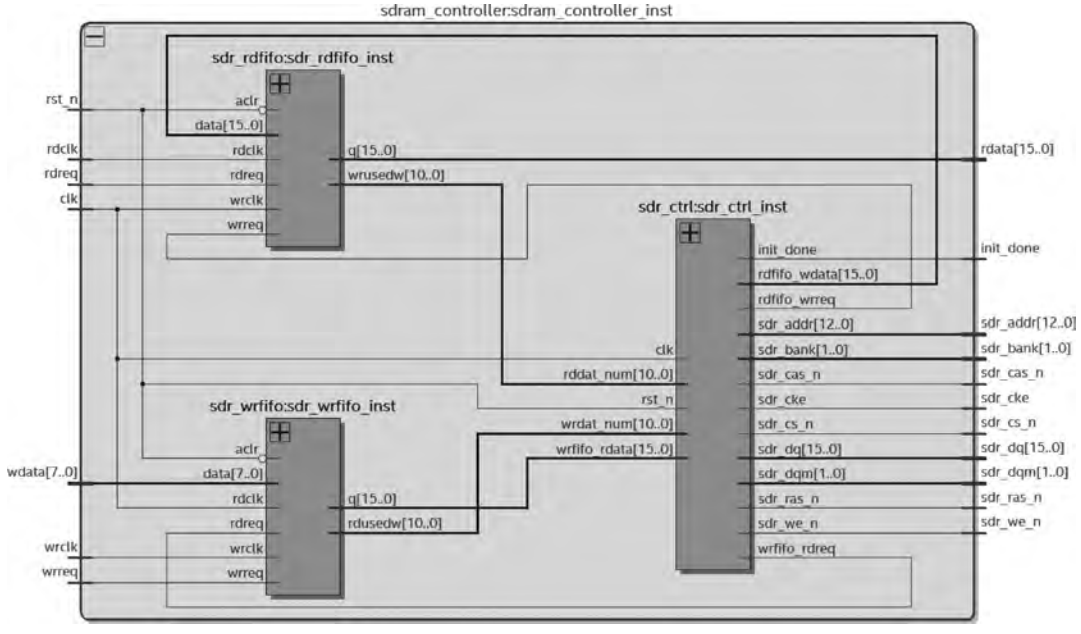

图 5.45　Sdram_controller 模块 RTL 视图

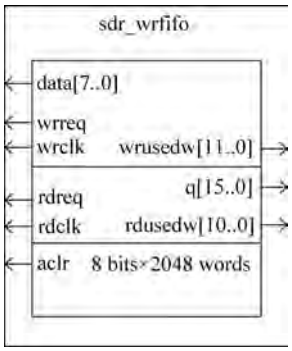

图 5.46 sdr_wrfifo IP 配置

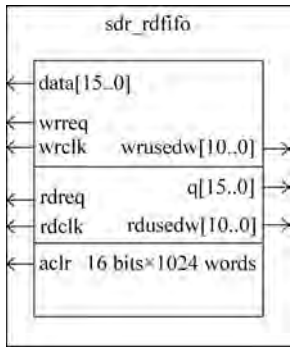

图 5.47 sdr_rdfifo IP 配置

完整工程软件代码扫码可见。

代码

5.10 本章小结

本章主要介绍了 VHDL 在各个数字模块或系统中的应用实例。组合逻辑主要介绍了基本门电路的建模，本时序电路建模讲解了从基本触发器到移位寄存器和分频器的设计。介绍了常用算法的 VHDL 设计，包括流水线加法器、乘法器及数字滤波器的设计。介绍了在通信领域中最常用的 FSK 调制与解调设计方法。本章最后给出了综合应用实例，包括 DDS、SD 卡驱动及 SDRAM 控制器的具体设计思想及设计代码，可以让读者了解复杂设计问题的设计思路及方法。