深圳市华为技术有限公司 研究管理部文档中心	文档编号	版本	密级
		1.0	*内部公开*
	资源类别：HDL 语言		共 56 页

Verilog 基本电路设计指导书

（仅供内部使用）

拟制：	Verilog Group	日期：	2000/04/04
批准：		日期：	yyyy/mm/dd
批准：		日期：	yyyy/mm/dd

深圳市华为技术有限公司

修订记录

日期	修订版本	描述	作者
2000/04/04	1.00	初稿完成	Verilog Group
2001/02/28	1.01	修订，主要增加三态和一些电路图	苏文彪

目 录

Verilog 基本电路设计指导书

关键词：电路、

摘　要：本文列举了大量的基本电路的 Verilog HDL 代码，使初学者能够迅速熟悉基本的 HDL 建模；同时也列举了一些常用电路的代码，作为设计者的指导。

缩略语清单： ***对本文所用缩略语进行说明，要求提供每个缩略语的英文全名和中文解释。***

参考资料清单： ***请在表格中罗列本文档所引用的有关参考文献名称、作者、标题、编号、发布日期和出版单位等基本信息。***

参考资料清单					
名称	作者	编号	发布日期	查阅地点或渠道	出版单位*（若不为本公司发布的文献，请填写此列）*
Actel HDL coding Style Guide			November 1997	文档室	Actel 公司

1　前言

当前业界的硬件描述语言中主要有 VHDL 和 Verilog HDL。公司根据本身 ASIC 设计现有的特点、现状，主推 Verilog HDL 语言，逐渐淡化 VHDL 语言，从而统一公司的 ASIC/FPGA 设计平台，简化流程。

为使新员工在上岗培训中能迅速掌握 ASIC/FPGA 设计的基本技能，中研基础部 ASIC 设计中心开发了一系列的培训教材。该套 HDL 语言培训系列包括如下教程：

《Verilog HDL 入门教程》

《Verilog HDL 代码书写规范》

《Verilog 基本电路设计指导书》

《TestBench 编码技术》

系列教材完成得较匆忙，本身尚有许多不完善的地方，同时，可能还需要其他知识方面的培训但没有形成培训教材，希望大家在培训过程中，多提宝贵意见，以便我们对它进行修改和完善

2　典型电路的设计

在本章节中，主要讲述触发器、锁存器、多路选择器、解码器、编码器、饱和/非饱和计数器、FSM 等常用基本电路的设计。如果你是初学者，我们建议你从典型电路学起，如果你已经非常熟悉电路设计，我们建议你从第 3 章看起。

2.1 全加器的设计

```
/*************************************************************************\
    Filename        :       fulladd.v
    Author          :       Verilog_gruop
    Description     :       Example of a one-bit full add.
    Revision        :       2000/02/29
    Company         :       Verilog_group
\*************************************************************************/
module FULLADDR(Cout, Sum, Ain, Bin, Cin);
    input           Ain, Bin, Cin;
    output          Sum, Cout;

    wire            Sum;
    wire            Cout;
    assign          Sum = Ain ^ Bin ^ Cin;
    assign          Cout = (Ain & Bin) | (Bin & Cin) | (Ain & Cin);
 endmodule
```

2.2 数据通路

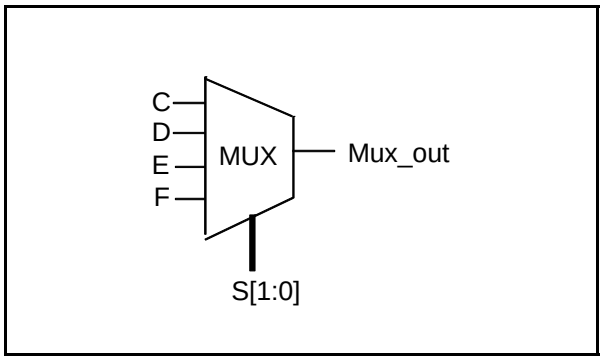
2.2.1四选一的多路选择器

用 case 语句实现的多路选择器，一般要求选择信号之间是相关的；case 的多路选择器一般是并行的操作，但有些工具也可能综合成优先级的译码器除非加一些控制参数。

```
/*************************************************************************\
    Filename        :       mux.v
    Author          :       Verilog_gruop
    Description     :       Example of a mux4-1.
    Revision        :       2000/02/29
    Company         :       Verilog_group
\*************************************************************************/
module  MUX( C,D,E,F,S,Mux_out);
```

```
    input           C,D,E,F ;       //input
    input   [1:0]   S ;             //select control
    output          Mux_out ;       //result

    reg Mux_out ;

    //mux
    always@(C or D or E or F or S)
    begin
        case (S)
            2'b00 : Mux_out = C ;
            2'b01 : Mux_out = D ;
            2'b10 : Mux_out = E ;
            default : Mux_out = F ;
        endcase
    end
endmodule
```

以上代码实现的功能如下所示：



1 Multiplexor using a case statement

2.2.2译码器

因为译码信号之间是相关的，因此，译码器要 case 语句实现。

```
/*************************************************************************\
    Filename        :       decode.v
    Author          :       Verilog_gruop
    Description     :       Example of a 3-8 decoder.
```

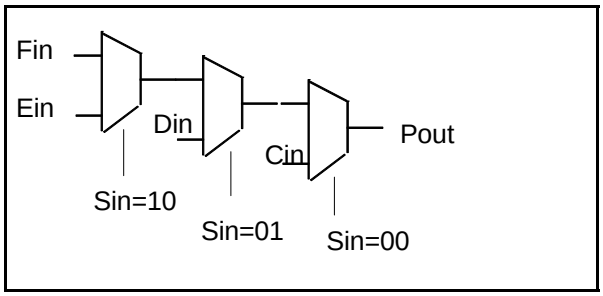
```
    Revision        :       2000/02/29
    Company         :       Verilog_group
\*************************************************************************/
module DECODE(Ain,En,Yout);
    input           En ;            //enable
    input   [2:0]   Ain ;           //input code
    output  [7:0]   Yout ;

    reg [7:0] Yout ;
    always@(En or Ain)
    begin
        if(!En)
            Yout = 8'b0 ;
        else
            case (Ain)
            3'b000 :  Yout = 8'b0000_0001 ;
            3'b001 : Yout = 8'b0000_0010 ;
            3'b010 : Yout = 8'b0000_0100 ;
            3'b011 : Yout = 8'b0000_1000 ;
            3'b100 : Yout = 8'b0001_0000 ;
            3'b101 : Yout = 8'b0010_0000 ;
            3'b110 : Yout = 8'b0100_0000 ;
            3'b111 : Yout = 8'b1000_0000 ;
            default : Yout = 8'b0000_0000 ;
            endcase
    end
endmodule
```

2.2.3优先编码器

```
/*************************************************************************\
    Filename        :       Prio-encoder.v
    Author          :       Verilog_gruop
    Description     :       Example of a Priority Encoder.
    Revision        :       2000/02/29
```

```
    Company         :       Verilog_group
\*********************************************************************/
module  PRIO_ENCODER (Cin,Din,Ein,Fin,Sin,Pout);
    input   Cin,Din,Ein,Fin;            // input signals
    input   [1:0]   Sin;                //input select control
    output  Pout;                       //output select  result
    reg             Pout;

    // Pout assignment
    always  @(Sin  or  Cin  or Din or Ein or Fin)
    begin
            if (Sin == 2'b00)
                    Pout    = Cin;
            else  if (Sin == 2'b01)
                    Pout =  Din;
            else if (Sin == 2'b10)
                    Pout =  Ein;
            else
                    Pout = Fin;
    end
endmodule
```

以上代码实现的功能如下图：



1　使用 if 的优先译码器

2.3 计数器

```
/*********************************************************************\
    Filename        :       count_en.v
    Author          :       Verilog_gruop
```

```
    Description     :       Example of a counter with enable.
    Revision        :       2000/02/29
    Company         :       Verilog_group
\*************************************************************************/
module COUNT_EN (En,Clock,Reset,Out);
    parameter               Width   =8 ;
    parameter               U_DLY =1;
    input                   Clock , Reset , En ;
    output  [Width-1:0]     Out ;

    reg     [Width-1:0]     Out ;
    always@(posedge Clock or negedge Reset)
        if (!Reset)
            Out     <= 8'b0 ;
        else if (En)
            Out     <= #U_DLY Out + 1 ;
endmodule
```

2.4算术操作

```
/*************************************************************************\
    Filename        :       arithmetic.v
    Author          :       Verilog_gruop
    Description     :       Example of a arithmetic include +, -, *, /.
    Revision        :       2000/02/29
    Company         :       Verilog_group
\*************************************************************************/
module ARITHMETIC (A , B, Q1, Q2 ,Q3, Q4 );
    input   [3:0]   A, B ;          //input operator
    output  [4:0]   Q1 ;            //output sum, with carry bit
    output  [3:0]   Q2;             //output sutract result
    output  [3:0]   Q3 ;            //output quotion
    output  [7:0]   Q4 ;            //product
```

```
    reg     [4:0]   Q1 ;
    reg     [3:0]   Q2 , Q3 ;
    reg     [7:0]   Q4 ;

    //arithmetic operate
    always@(A or B)
    begin
        Q1 = A+B ;
        Q2 = A-B ;
        Q3 = A/2 ;
        Q4 = A*B ;
    end
endmodule
```

2.5 逻辑操作

```
/*************************************************************************\
    Filename        :   relational.v
    Author          :   Verilog_gruop
    Description     :   Example of a relational operate
    Revision        :   2000/02/29
    Company         :   Verilog_group
\*************************************************************************/
module  RELATIONAL(A, B,Q1,Q2,Q3,Q4) ;
    input   [3:0]   A , B ;             //operator
    output          Q1 , Q2 , Q3 , Q4 ;     //result

    reg             Q1 , Q2 , Q3 , Q4 ;

    //compare
    always@(A or B)
    begin
        Q1      = A > B ;
        Q2      = A < B ;
        Q3      = A>= B ;
```

```
        if (A <= B)
                Q4      = 1 ;
        else
                Q4      = 0 ;
    end
endmodule
```

2.6 移位操作

```
/*************************************************************************\
    Filename        :       shifter.v
    Author          :       Verilog_gruop
    Description     :       Example of a shifter
    Revision        :       2000/02/29
    Company         :       Verilog_group
\*************************************************************************/
module SHIFT (Data ,Q1, Q2) ;
    input   [3:0]   Data ;
    output  [3:0]   Q1,Q2 ;

    parameter       B = 2 ;
    reg     [3:0]   Q1, Q2 ;
    always@(Data)
    begin
            Q1      = Data << B ;
            Q2      = Data >> B ;
    end
endmodule
```

2.7 时序器件

一个时序器件（指触发器或锁存器）就是一个一位存储器。锁存器是电平敏感存储器件，触发器是沿触发存储器件。

触发器也被称为寄存器，在程序中体现为对上升沿或下降沿的探测，VERILOG 中采用如下方法表示：

（posedge Clk) -------- 上升沿

(negedge Clk) --------下降沿

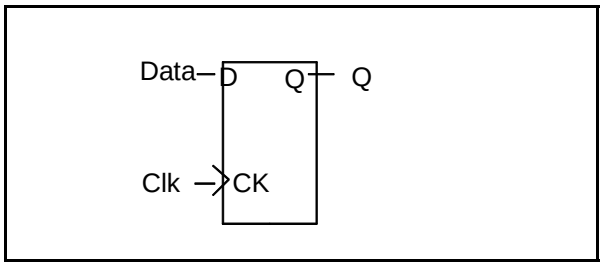
下面给出各种不同类型触发器的描述。

2.7.1上升沿触发的触发器

实现了一个 D 触发器。

```
/*************************************************************************\
    Filename      :     dff.v
    Author        :     Verilog_gruop
    Description   :     Example of a Rising Edge Flip-Flop.
    Revision      :     2000/03/30
    Company       :     Verilog_group
\*************************************************************************/
module DFF (Data, Clk, Q);
    input       Data, Clk;
    output      Q;

    reg         Q;
    always  @ (posedge Clk)
       Q     <= Data;
endmodule
```

功能如下图：



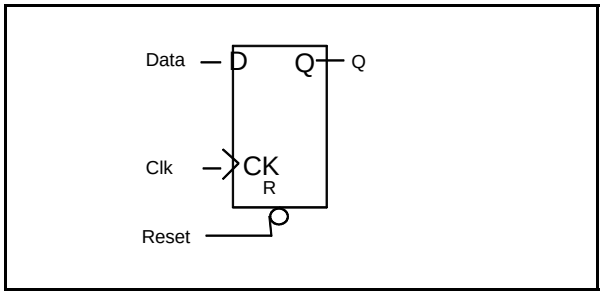
1 D 触发器

2.7.2 带异步复位、上升沿触发的触发器

```
/*************************************************************************\
    Filename      :     dff_async_rst.v
    Author        :     Verilog_gruop
    Description   :     Example of a Rising Edge Flip-Flop with  Asynchronous  Reset.
```

```
    Revision        :       2000/03/30
    Company         :       Verilog_group
\*********************************************************************/
module DFF_ASYNC_RST (Data, Clk, Reset, Q);
    input           Data, Clk, Reset;
    output          Q;
    parameter       U_DLY =1;

    reg Q;
    always @ (posedge Clk  or  negedge Reset)
            if ( ~Reset)
                    Q       <=  #U_DLY 1'b0 ;
            else
                    Q       <= #U_DLY Data ;
endmudule
```

功能如下图：



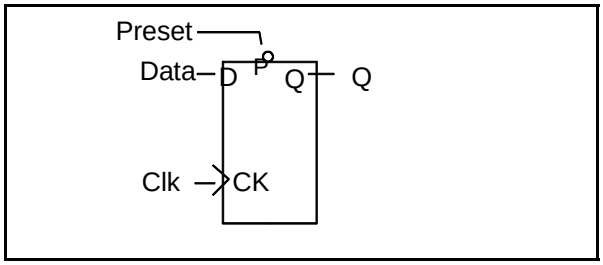
1　带异步复位 D 触发器

2.7.3带异步置位、上升沿触发的触发器

```
/*********************************************************************\
    Filename        :       dff_async_pre.v
    Author          :       Verilog_gruop
    Description     :       Example of a Rising Edge Flip-Flop with  Asynchronous  Preset.
    Revision        :       2000/03/30
    Company         :       Verilog_group
\*********************************************************************/
module DFF_ASYNC_PRE (Data, Clk, Preset, Q);
input           Data, Clk, Preset;
```

```
output          Q;
parameter   U_DLY =1;
reg Q;
always @ (posedge Clk  or  negedge Preset)
    if ( ~Preset)
        Q       <=  #U_DLY 1'b1 ;
    else
        Q       <= #U_DLY  Data ;
endmudule
```

功能如下图：



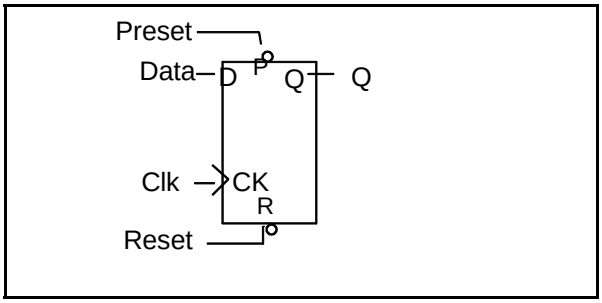
1　带异步置位 D 触发器

2.7.4带异步复位和置位、上升沿触发的触发器

```
/*************************************************************************\
    Filename        :       dff_async.v
    Author          :       Verilog_gruop
    Description     :       Example of a Rising Edge Flip-Flop
                                                    with  Asynchronous  Reset and  Preset.
    Revision        :       2000/03/30
    Company         :       Verilog_group
\*************************************************************************/
module  DFF_ASYNC (Data, Clk, Reset, Preset, Q);
    input           Data, Clk, Reset, Preset ;
    output          Q;
    parameter       U_DLY = 1;
    reg             Q;

    always @ (posedge Clk  or  negedge Reset  or  negedge  Preset)
```

```
        if ( ~Reset)
                Q       <= 1'b0 ;
        else if  ( ~ preset )
                Q       <= 1'b1;
        else
                Q       <= #U_DLY  Data ;
endmudule
```

功能如下图：

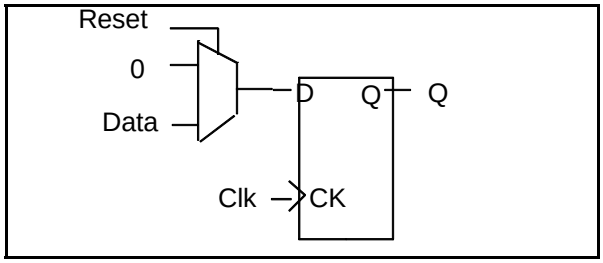


1　带异步置位、复位 D 触发器

2.7.5带同步复位、上升沿触发的触发器

```
/*************************************************************************\
    Filename        :       dff_sync_rst.v
    Author          :       Verilog_gruop
    Description     :       Example of a Rising Edge Flip-Flop with  Synchronous  Reset.
    Revision        :       2000/03/30
    Company         :       Verilog_group
\*************************************************************************/
module  DFF_SYNC_RST (Data, Clk, Reset, Q);
    input           Data, Clk, Reset;
    output          Q;
    parameter       U_DLY = 1;
    reg             Q;
    always @ (posedge Clk )
        if ( ~Reset)
                Q       <=  #U_DLY 1'b0 ;
        else
                Q       <= #U_DLY  Data ;
endmudule
```

功能如下图：



1　带同步复位 D 触发器

2.7.6带同步置位、上升沿触发的触发器

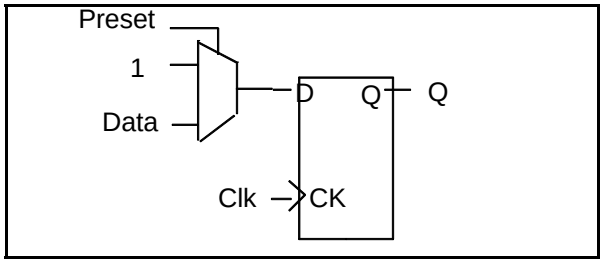
```
/*********************************************************************\
    Filename        :       dff_sync_pre.v
    Author          :       Verilog_gruop
    Description     :       Example of a Rising Edge Flip-Flop with  Synchronous  Preset.
    Revision        :       2000/03/30
    Company         :       Verilog_group
\*********************************************************************/
module  DFF_SYNC_PRE (Data, Clk, Preset, Q);
    input           Data, Clk, Preset;
    output          Q;
    parameter       U_DLY = 1;
    reg             Q;

    always @ (posedge Clk )
        if ( ~Preset)
            Q       <= #U_DLY 1'b1 ;
        else
            Q       <= #U_DLY Data ;
    endmudule
```

功能如下图：

1　带同步置位的D 触发器

2.7.7带异步复位和时钟使能、上升沿触发的触发器

```
/*****************************************************************************\
    Filename        :       dff_ck_en.v
```



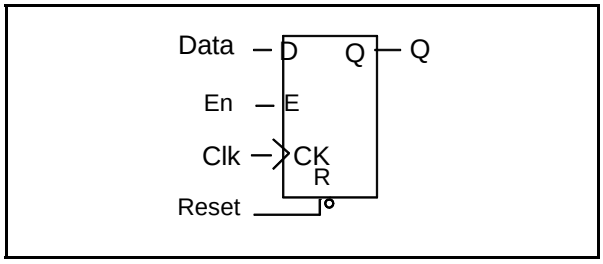

```
    Author          :       Verilog_gruop
    Description     :       Example of a Rising Edge Flip-Flop with  Asynchronous  Reset
                                                                    and  Clock Enable.
    Revision        :       2000/03/30
    Company         :       Verilog_group
\*****************************************************************************/
module DFF_CK_EN (Data, Clk, Reset, En, Q);
    input           Data, Clk, Reset, En;
    output          Q;
    parameter       U_DLY = 1;
    reg             Q;
    always @ (posedge Clk  or  negedge Reset)
            if ( ~Reset)
                    Q       <= 1'b0 ;
            else  if  (En)
                    Q       <= #U_DLY  Data ;
    endmudule
```

功能如下图：

1　带异步复位、使能端的D 触发器

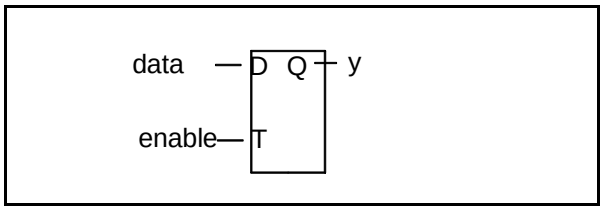
2.7.8D-Latch （锁存器）

锁存器是电平敏感器件，在 ASIC 设计中，锁存器会带来诸多问题，如额外时延、DFT 问题，因此，在实际设计中必须尽量避免锁存器的出现。




```
module  d_latch (enable,data,y);
    input enable ;
    input  data;
    output y;
    reg  y;

    always @(enable or data)
            if (enable )
                    y <= data;
endmodule
```

功能如下图：



1　D-Latch

2.8 ALU

```
/*****************************************************************************\
    Filename        :       alu.v
    Author          :       Verilog_gruop
    Description     :       Example of a  4-bit Carry Look Ahead ALU
```

```
    Revision        :       2000/02/29
    Company         :       Verilog_group
\*************************************************************************/
module ALU(A, B, Cin, Sum, Cout, Operate, Mode);
//input signals
    input   [3:0]   A, B;           // two operands of ALU
    input           Cin;            //carry in at the LSB
    input   [3:0]   Operate;        //determine f(.) of sum = f(a, b)
    input           Mode;           //arithmetic(mode = 1'b1) or logic operation(mode = 1'b0)
    output  [3:0]   Sum;            //result of ALU
    output          Cout;           //carry produced by ALU operation

    // carry generation bits and propogation bits.
    wire    [3:0]   G, P;

    // carry bits;
    reg     [2:0]   C;

    // function for carry generation:
    function gen
            input           A, B;
            input   [1:0]   Oper;

            begin
            case(Oper)
                    2'b00: gen = A;
                    2'b01: gen = A & B;
                    2'b10: gen = A & (~B);
                    2'b11: gen = 1'b0;
            endcase;
            end
endfunction
```

```
// function for carry propergation:
function prop
    input           A, B;
    input   [1:0]   Oper;

    begin
        case(Oper)
            2'b00: prop = 1;
            2'b01: prop = A | (~B);
            2'b10: prop = A | B;
            2'b11: prop = A;
        endcase;
    end
endfunction

// producing carry generation bits;
assign G[0] = gen(A[0], B[0], Oper[1:0]);
assign G[1] = gen(A[1], B[1], Oper[1:0]);
assign G[2] = gen(A[2], B[2], Oper[1:0]);
assign G[3] = gen(A[3], B[3], Oper[1:0]);

// producing carry propogation bits;
assign P[0] = por(A[0], B[0], Oper[3:2]);
assign P[1] = por(A[1], B[1], Oper[3:2]);
assign P[2] = por(A[2], B[2], Oper[3:2]);
assign P[3] = por(A[3], B[3], Oper[3:2]);

// producing carry bits with carry-look-ahead;
always @(G or P or Cin, Mode)
begin
    if (Mode) begin
        C[0] = G[0] | P[0] & Cin;
        C[1] = G[1] | P[1] & G[0] | P[1] & P[0] & Cin;
```

```
    C[2] = G[2] | P[2] & G[1] | P[2] & P[1] & G[0] | P[2] & P[1] & P[0] & Cin;
        Cout = G[3] | P[3] & G[2] | P[3] & P[2] & G[1] | P[3] & P[2] & P[1] & G[0] | P[3] &
                P[2] & P[1] & P[0] & Cin;
    end
    else begin
        C[0] = 1'b0;
        C[1] = 1'b0;
        C[2] = 1'b0;
        Cout = 1'b0;
    end
end

// calculate the operation results;
assign Sum[0] = (~G[0] & P[0]) ^ Cin;
assign Sum[1] = (~G[1] & P[1]) ^ C[0];
assign Sum[2] = (~G[2] & P[2]) ^ C[1];
assign Sum[3] = (~G[3] & P[3]) ^ C[2];

endmodule
```

2.9 有限状态机（FSM）的设计

2.9.1 概述

有限状态机（FSM）是一种常见的电路，由时序电路和组合电路组成。设计有限状态机的第一步是确定采用 Moore 状态机还是采用 Mealy 状态机。（Mealy 型：状态的转变不仅和当前状态有关，而且跟各输入信号有关；Moore 型：状态的转变只和当前状态有关）。从实现电路功能来讲，任何一种都可以实现同样的功能。但他们的输出时序不同，所以，在选择使用那种状态机时要根据具体情况而定，在此，把他们的主要区别介绍一下：

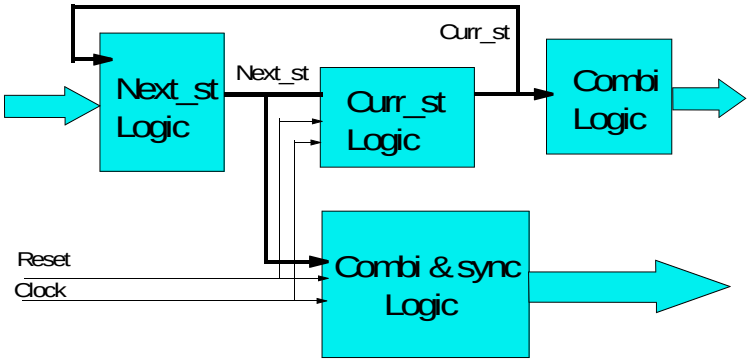
1. Moore 状态机：在时钟脉冲的有限个门延时之后，输出达到稳定。输出会在一个完整的时钟周期内保持稳定值，即使在该时钟内输入信号变化了，输出信号也不会变化。输入对输出的影响要到下一个时钟周期才能反映出来。把输入和输出分开，是 Moore 状态机的重要特征。

2. Mealy 状态机：由于输出直接受输入影响，而输入可以在时钟周期的任一时刻变化，这就使得输出状态比 Moore 状态机的输出状态提前一个周期到达。输入信号的噪声可能会出现在输出信号上。

3. 对同一电路，使用 Moore 状态机设计可能会比使用 Mealy 状态机多出一些状态。

根据他们的特征和要设计的电路的具体情况，就可以确定使用那种状态机来实现功能。一旦确定状态机，接下来就要构造状态转换图。现在还没有一个成熟的系统化状态图构造算法，所以，对于实现同一功能，可以构造出不同的状态转换图。但一定要遵循结构化设计。在构造电路的状态转换图时，使用互补原则可以帮助我们检查设计过程中是否出现了错误。互补原则是指离开状态图节点的所有支路的条件必须是互补的。同一节点的任何 2 个或多个支路的条件不能同时为真。同时为真是我们设计不允许的。

在检查无冗余状态和错误条件后，就可以开始用 verilog HDL 来设计电路了。



1　状态机电路逻辑图

在设计的过程中要注意以下方面：

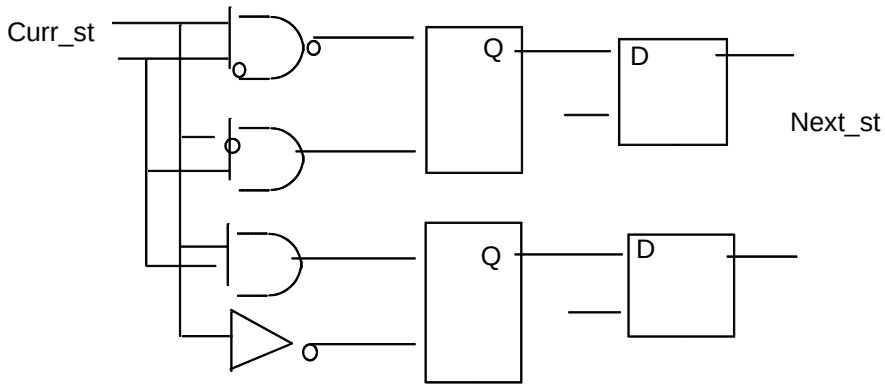
1. full_case spec

定义完全状态，即使有的状态可能在电路中不会出现。目的是避免综合出不希望的 Latch，因为 Latch 可能会带来：a. 额外的延时；b. 异步 Timing 问题

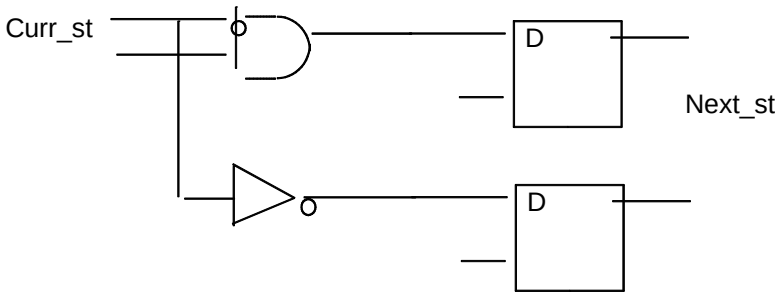
```
always @(Curr_st)
begin
      case(Curr_st)
              ST0 : Next_st = ST1;
              ST1 : Next_st = ST2;
              ST2 : Next_st = ST0;
             endcase
    end
```

1　没有采用 full-case

```
always @(Curr_st)
begin
      case(Curr_st)                    //synthesis full_case
              ST0 : Next_st = ST1;
              ST1 : Next_st = ST2;
              ST2 : Next_st = ST0;
         default : Next_st = ST0;
```




```
          endcase
end
```



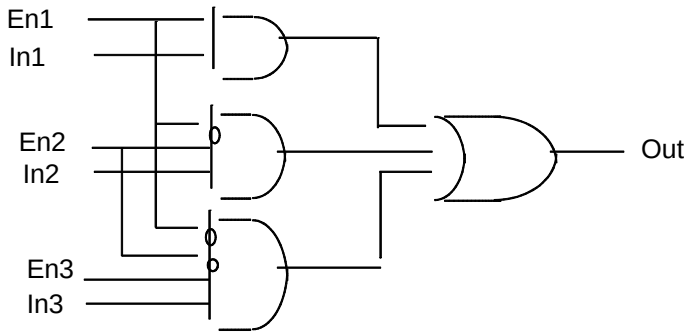
1　采用 full-case

2. parallel_case spec

确保不同时出现多种状态

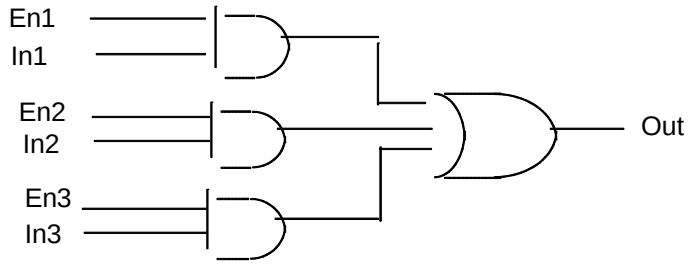
```
case({En3, En2, En1})
      3'b??1 : Out = In1;
```

```
        3'b?1? : Out = In2;
        3'b1?? : Out = In3;
endcase
```



1　没采用 parallel-case

```
case({En3, En2, En1})                    //synthesis parallel_case
        3'b??1 : Out = In1;
        3'b?1? : Out = In2;
        3'b1?? : Out = In3;
endcase
```



1　采用 parallel-case

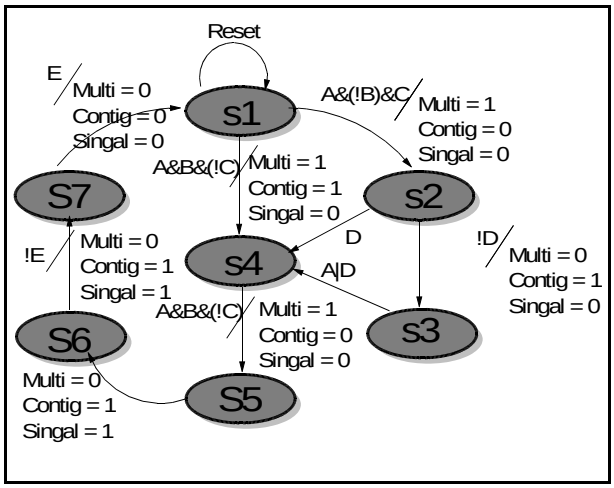
3. 禁止使用 casex

casex 在综合时，认为 Z，X 为 Dont cares，会导致前仿真和后仿真不一致。如果电路中出现 X，一定要分析是否会传递。

4. 推荐在模块划分时，把状态机设计分离出来，便于使用综合根据对状态机优化。

5. 在条件表达式或附值语句中，要注意向量的宽度适配。否则，前仿真和后仿真不一致，RTL 级的功能验证很难找出问题所在。

下图是一个状态机的状态转换图，在 Verilog HDL 中我们可以用如下方法设计该状态机。



1　状态转换图

2.9.2One-hot 编码

```
/*************************************************************************\
    Filename        :       one_hot_fsm.v
    Author          :       Verilog_gruop
    Description     :       Example of a one-hot encoded state machine.
    Revision        :       2000/02/29
    Company         :       Verilog_group
\*************************************************************************/
module ONE_HOT_FSM (Clock, Reset, A, B, C, D, E,
                    Single, Multi, Contig);

    input       Clock;              //system Clock
    input       Reset;              //async Reset, active high
    input       A, B, C, D, E;      //FSM input signals
    output      Single, Multi, Contig;  //FSM output signals

    //define output signals type
    reg         Single;
    reg         Multi;
    reg         Contig;
```

```
// Declare the symbolic names for states
parameter       [6:0]           // enum STATE_TYPE one-hot
                S1      = 7'b0000001,
                S2      = 7'b0000010,
                S3      = 7'b0000100,
                S4      = 7'b0001000,
                S5      = 7'b0010000,
                S6      = 7'b0100000,
                S7      = 7'b1000000;
parameter       U_DLY           = 1;

// Declare current state and next state variables
reg     [2:0]   Curr_st;
reg     [2:0]   Next_st;

//Curr_st assignment, sequential logic
always @ (posedge Clock or posedge Reset)
begin
        if (Reset)
          Curr_st       <= S1;
        else
          Curr_st       <= #U_DLY Next_st;
 end

//combinational logic
 always @ (Curr_st or A or B or C or D or D or E)
begin
        case (Curr_st)          //full_case
                S1 :
                begin
                        Multi           = 1'b0;
                        Contig          = 1'b0;
                        Single          = 1'b0;
```

```
            if (A & ~B & C)
                Next_st     = S2;
            else if (A & B & ~C)
                Next_st     = S4;
            else
                Next_st     = S1;
        end
        S2 :
        begin
            Multi       = 1'b1;
                Contig      = 1'b0;
                Single      = 1'b0;
            if (!D)
                                Next_st             = S3;
            else
                                Next_st             = S4;
        end
       S3 :
        begin
            Multi       = 1'b0;
                Contig      = 1'b1;
            Single      = 1'b0;
            if (A | D)
                Next_st     = S4;
            else
                Next_st     = S3;
        end
        S4 :
        begin
            Multi       = 1'b1;
            Contig      = 1'b1;
            Single      = 1'b0;
            if (A & B & ~C)
```

```
                    Next_st         = S5;
            else
                    Next_st         = S4;
        end
        S5 :
        begin
            Multi       = 1'b1;
            Contig      = 1'b0;
            Single      = 1'b0;
            Next_st             = S6;
        end
        S6 :
        begin
            Multi       = 1'b0;
            Contig      = 1'b1;
            Single       = 1'b1;
            if (!E)
                    Next_st     = S7;
            else
                    Next_st     = S6;
        end
        S7 :
        begin
            Multi       = 1'b0;
            Contig      = 1'b1;
            Single      = 1'b0;
            if (E)
                    Next_st     = S1;
            else
                    Next_st     = S7;
            end
    endcase
end
```

```
endmodule
```

2.9.3Binary 编码

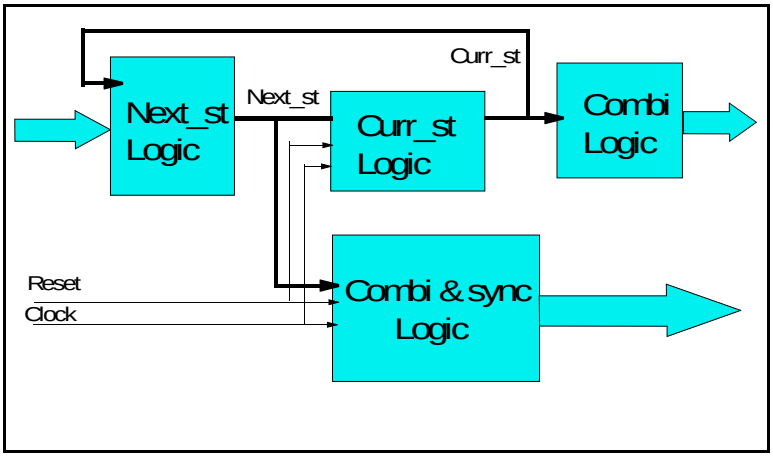
```
/*****************************************************************************\
    Filename        :       binary_fsm.v
    Description     :       Example of a binary encoded state machine.
    Revision        :       2000/02/29
    Company         :       Huawei Ltd.
\*****************************************************************************/
`timescale 1ns / 10ps
module binary (Clock, Reset, A, B, C, D, E,
        Single, Multi, Contig);
    input           Clock;              //system Clock
    input           Reset;              //async Reset, active high
    input           A, B, C, D, E;      //FSM input signals
    output          Single, Multi, Contig;  //FSM output signals

    //define output signals type
    reg             Single;
    reg             Multi;
    reg             Contig;
// Declare the symbolic names for states
parameter   [2:0]           //enum STATE_TYPE  binary
                    S1      = 3'b001,
                    S2      = 3'b010,
                    S3      = 3'b011,
                    S4      = 3'b100,
                    S5      = 3'b101,
                    S6      = 3'b110,
                    S7      = 3'b111;
parameter   U_DLY = 1;

// Declare current state and next state variables
    reg     [2:0]   Curr_st;
```

```
reg     [2:0]    Next_st;

//Curr_st assignment, sequential logic
always @ (posedge Clock or posedge Reset)
begin
        if (Reset)
          Curr_st       <= S1;
        else
          Curr_st       <= #U_DLY Next_st;
end

//combinational logic
always @ (Curr_st or A or B or C or D or D or E)
begin
                case (Curr_st)          //full_case
               S1 :
                begin
                        Multi           = 1'b0;
                        Contig          = 1'b0;
                        Single          = 1'b0;
                                if (A & ~B & C)
                                 Next_st        = S2;
                        else if (A & B & ~C)
                                Next_st         = S4;
                        else
                                Next_st         = S1;
                end
                S2 :
                begin
                                        Multi           = 1'b1;
                                Contig          = 1'b0;
                                        Single          = 1'b0;
                                if (!D)
```

```
                    Next_st         = S3;
            else
                    Next_st         = S4;
        end
    S3 :
      begin
        Multi       = 1'b0;
            Contig      = 1'b1;
        Single      = 1'b0;
        if (A | D)
            Next_st     = S4;
        else
            Next_st     = S3;
      end
      S4 :
      begin
        Multi       = 1'b1;
        Contig      = 1'b1;
        Single      = 1'b0;
        if (A & B & ~C)
            Next_st         = S5;
        else
            Next_st     = S4;
      end
      S5 :
      begin
        Multi       = 1'b1;
        Contig      = 1'b0;
        Single      = 1'b0;
        Next_st         = S6;
      end
      S6 :
      begin
```

```
                Multi           = 1'b0;
                Contig          = 1'b1;
                Single          = 1'b1;
                if (!E)
                        Next_st         = S7;
                else
                        Next_st         = S6;
        end
        S7 :
        begin
                Multi           = 1'b0;
                Contig          = 1'b1;
                Single          = 1'b0;
                if (E)
                        Next_st         = S1;
                else
                        Next_st         = S7;
        end
    endcase
  end
endmodule
```

以上介绍的用 Verilog HDL 设计来实现的 FSM 电路，可用下面的逻辑图来表现：



1　FSM 逻辑框图

2.10三态总线

2.10.1三态 buffer

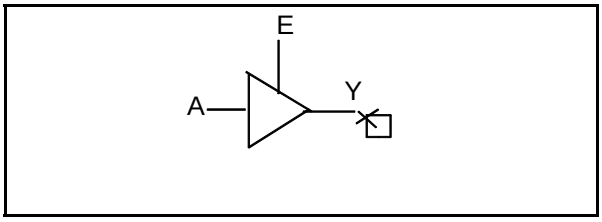
三态 buffer 是带有高阻输出能力的输出 buf 。在总线结构中，为解决总线竞争问题，必须采用三态的输出 buf 。

```
module TRISTATE (E, A,Y);
    input E,A;
    output  Y;
    reg  Y;

    always @(E or  A)
    begin
            if ( E)
                    Y = A;
            else
                    Y = 1'b Z;
    end
endmodule
```

或者：

```
module TRISTATE(E,A,Y);
     input E,A;
    output  Y;
    assign  Y = E? A:1'bZ;
```

功能图如下：



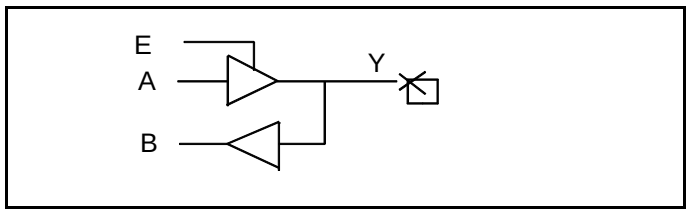
1　三态 buffer

2.10.2双向 I/O buffer

双向总线可输入、输出，输出带高阻。

```
module  BIDIR ( E,A,Y,B);
    input E,A;
    output B;
    inout Y;            // in and out bus

    tri Y;              // net type  is tri

    assign B = Y;
    assign  Y =  E?  A:1'bz;
endmodule
```

功能图如下：



1　双向总线 buffer

3　常用电路设计

3.1CRC 校验码产生器的设计

3.1.1概述

冗余编码是在二进制通信系统中常用的差错检测方法，它是通过在原始数据后加冗余校验码来检测差错，冗余位越多，检测出传输错误的机率越大。循环冗余编码（Cyclic Redundancy Codes，简称 CRC）是一种常用的冗余编码，CRC 校验的基本原理是：CRC 可由一称为生成多项式的常数去除该数据流的二进制数值而得，商数被放弃，余数作为冗余编码追加到数据流尾，产生新的数据流进行发送。在接收端，新的数据流被同一常数去除，检查余数是否为零。 如果余数为零，就认为传输正确，否则就认为传输中已发生差错，该数据流重发。

3.1.2CRC 校验码产生器的分析与硬件实现

在产生 CRC 校验码时，需要用到除法运算。一般说来，非常大的数字进行除法时，用数字逻辑实现时是比较麻烦的。因此，把二进制信息预先转换成一定的格式，这就是 CRC 的多项式表示。 二进制数表示为生成多项式的系数，如下例所示：

$$1,0001,0000,0010,0001 = x^{16} + x^{12} + x^{5} + 1$$

在多项式表示中，所有的二进制数均被表示成一个多项式，多项式的系数就是二进制中的对应值。D 为数据流多项式，G 为生成多项式，Q 为商数多项式，R 为余数多项式。在生成 CRC 校验码时，数据流多项式 D 被乘以 X^n，这里 n 为生成多项式 G 的最高次数，也就是 CRC 的长度。这个操作是通过将左移 n 位得到的，我们可以用 CRC 来代替多项式最后的 n 个 0，组成新的数据流多项式。由于二进制的加法和减法是等价的，所以产生新的数据流多项式应能被生成多项式 G 除尽。用以下公式表示为：

$$（X^nD）＋R ＝ （QG）＋ 0$$

在接收端，传输信息的前一部分为原始数据流 D；后一部分（最后 n 位数）为余数 R。整个数据流多项式被同一生成多项式 G 去除，商数被丢弃，余数应为 0。如果余数不为 0，说明传输数据时发生错误，数据需要重传。

不同的生成多项式有不同的检错能力，为了得到优化的结果，我们必须根据需要选择合适的生成多项式，CRC-16 的生成多项式为：

$$G(x) = x^{16} + x^{12} + x^5 + 1$$

CRC 校验码产生器分两种：串行 CRC 校验码产生器和并行 CRC 校验码产生器。本文用到的是并行 CRC 校验码产生器。由于计算并行 CRC 时用到了串行 CRC 的一些思想，所以在此先讲一下串行 CRC 的产生。

通常，CRC 校验码的值可以通过线性移位寄存器和异或门求得，线性移位寄存器一次移一位，完成除法功能，异或门完成不带进位的减法功能。 如果商数为’1’，则从被除数的高阶位减去除数，同时移位寄存器右移一位 ，准备为被除数的较低位进行运算。 如果商数为’0’，则移位寄存器直接右移一位 。串行 CRC-16 校验码产生器的原理图如图 2 所示。

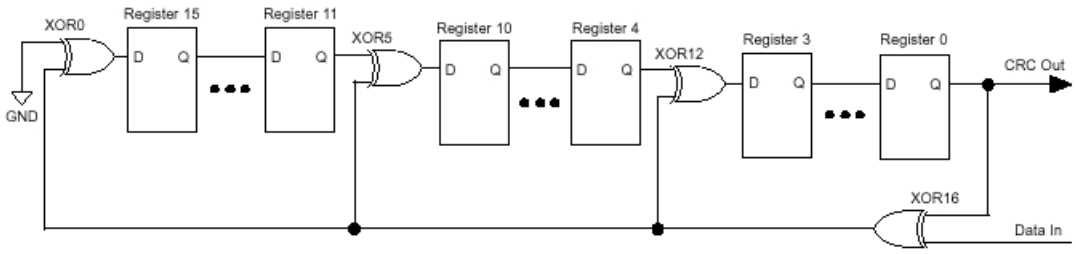


图 2　串行 CRC-16 校验码产生器原理图

在设计并行 CRC 校验码产生器的时候，我们可以采用串行 CRC 校验码的思想，用线性移位寄存器的方法产生并行 CRC 校验码。 与串行 CRC 校验码产生器不同的是，并行 CRC 校验码产生器 16 位 CRC 同时输出，所以要求在一个时钟周期内，移位寄存器一次需要移 16 位。实际上，移位寄存器不可能在一个时钟周期内移 16 位，所以这部分电路是用组合逻辑来完成。整个 CRC 校验码产生器由组合逻辑和 16 个输出寄存器组成，通过仿真和综合，满足设计要求。

3.1.3并行 CRC-16 校验码产生器的 Verilog HDL 编码

```
/*********************************************************************
*          Filename : crc16_para.v
*           Auther : Verilog group
*       Description : This module is used to check CRC_16 of 8-bits cell
*                     data, the generator polynomial is x^16+x^12+x^5+1.
*         Called by :
*    Revision History : 2000-5-5 Revision 1.0
*            Email : zhangnb@sz.huawei.com.cn
*          Company : Huawei Technology Inc.
*    Copyright(c) 1999,Huawei Technology Inc.,All right reserved.
*********************************************************************/

//----------------------------
// TOP MODULE
//----------------------------
module CRC16_PARA(
                        Reset   ,    //Reset signal
                        Gclk    ,    //Clock signal
                        Soc     ,    //Start of cell
                        Data_in ,    //input data of cell
                        Crc_out      //output CRC signal
                      ) ;

//----------------------------
// SIGNAL DECLARATIONS
//----------------------------
input        Reset   ;
input        Gclk    ;
input        Soc     ;
input  [7:0]   Data_in ;
output [15:0]  Crc_out ;
```

```
//----------------------------
// SIGNAL DECLARATIONS
//----------------------------
wire          Reset   ;
wire          Gclk    ;
wire          Soc     ;
wire   [7:0]  Data_in ;
reg    [15:0] Crc_out ;

reg    [15:0] Crc_tmp ;
reg           Temp    ;
integer       i,j,k,l ;

//----------------------------
// PARAMETERS
//----------------------------
parameter  U_DLY=1    ;

//----------------------------
// Crc_out signal
//----------------------------
always @(posedge Reset or posedge Gclk)
begin
        if (Reset)
                Crc_out <= #U_DLY 16'b0 ;
        else if (Soc == 1'b1)
                Crc_out <= #U_DLY 16'b0 ;
        else
                Crc_out <= #U_DLY Crc_tmp ;
end

//----------------------------
// Crc_tmp signal
```

```
//----------------------------
always @(Crc_out or Data_in)
begin
        Crc_tmp = Crc_out ;
        for (i=7;i>=0;i=i-1)
        begin
                Temp = Data_in[i] ^ Crc_tmp[15]  ;

                for (j=15;j>12;j=j-1)
                        Crc_tmp[j] = Crc_tmp[j-1] ;
                Crc_tmp[12] = Temp ^ Crc_tmp[11] ;

                for (k=11;k>5;k=k-1)
                        Crc_tmp[k] = Crc_tmp[k-1] ;
                Crc_tmp[5] = Temp ^ Crc_tmp[4]   ;

                for (l=4;l>0;l=l-1)
                        Crc_tmp[l] = Crc_tmp[l-1] ;
                Crc_tmp[0] = Temp               ;
        end
end

endmodule
```

3.1.4串行 CRC-16 校验码产生器的 Verilog HDL 编码

```
/*****************************************************************************
*          Filename : crc16_ser.v
*            Auther : Verilog group
*       Description : This module is used to check CRC_16 of serial data,
*                     the generator polynomial is x^16+x^12+x^5+1.
*         Called by :
*   Revision History : 2000-5-5 Revision 1.0
*             Email : zhangnb@sz.huawei.com.cn
```

```
*           Company : Huawei Technology Inc.
*    Copyright(c) 1999,Huawei Technology Inc.,All right reserved.
********************************************************************************/

//-----------------------------
// TOP MODULE
//-----------------------------
module CRC16_SER(
                        Reset   ,    //Reset signal
                        Gclk    ,    //Clock signal
                        Soc     ,    //Start of cell
                        Data_in ,    //input data of cell
                        Crc_out      //output CRC signal
                      ) ;

//-----------------------------
// SIGNAL DECLARATIONS
//-----------------------------
input       Reset   ;
input       Gclk    ;
input       Soc     ;
input       Data_in ;
output [15:0]  Crc_out ;

//-----------------------------
// SIGNAL DECLARATIONS
//-----------------------------
wire        Reset   ;
wire        Gclk    ;
wire        Soc     ;
wire        Data_in ;
reg    [15:0]  Crc_out ;
```

```
reg          Temp    ;
integer      i,j,k,l ;

//-----------------------------
// PARAMETERS
//-----------------------------
parameter  U_DLY=1    ;

//-----------------------------
// Crc_out signal
//-----------------------------
always @(posedge Reset or posedge Gclk)
begin
        if (Reset)
                Crc_out <= #U_DLY 16'b0 ;
        else if (Soc == 1'b1)
                Crc_out <= #U_DLY 16'b0 ;
        else
        begin
                Temp = Data_in ^ Crc_out[15]  ;

                for (j=15;j>12;j=j-1)
                        Crc_out[j] <= #U_DLY Crc_out[j-1] ;
                Crc_out[12] <= #U_DLY Temp ^ Crc_out[11] ;

                for (k=11;k>5;k=k-1)
                        Crc_out[k] <= #U_DLY Crc_out[k-1] ;
                Crc_out[5] <= #U_DLY Temp ^ Crc_out[4]   ;

                for (l=4;l>0;l=l-1)
                        Crc_out[l] <= #U_DLY Crc_out[l-1] ;
                Crc_out[0] <= #U_DLY Temp               ;
        end
```

end

endmodule

3.2 随机数产生电路设计

3.2.1概述

伪随机序列又称为伪随机码，是一组人工生成的周期序列。它不仅具有随机序列的一些统计特性和高斯噪声所有的良好的自相关特征，而且具有某种确定的编码规则，同时又便于重复产生和处理，因而在通信领域应用广泛。

伪随机序列的产生方式很多，通常产生的伪随机序列的电路为一反馈移位寄存器。它又可分为线性反馈移位寄存器和非线性反馈移位寄存器两类。由线性反馈移位寄存器产生出的周期最长的二进制数字序列称为最大长度线性反馈移位寄存器序列，简称 m 序列，移位寄存器的长度为 n，则 m 序列的周期为 2n-1，没有全 0 状态。

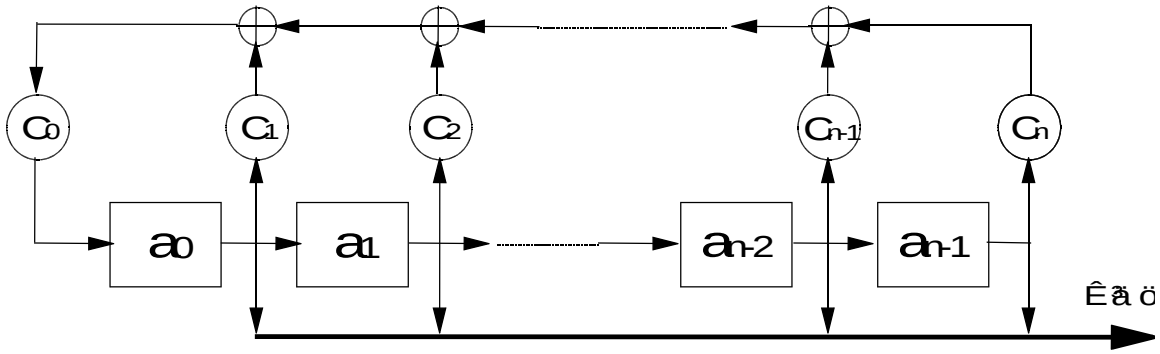
其中，伪随机数发生器的初始状态由微处理器通过 SEED 寄存器给出。

3.2.2伪随机序列发生器的硬件实现

伪随机序列发生器的初始状态是由微处理器中 SEED 寄存器提供的，而 SEED 寄存器的位数为 8 位，所以需要设计一种 8 位的伪随机序列发生器，它的本原多项式为：

F(x) = x8 + x4 + x3 + x2 + 1

伪随机序列发生器结构如图 1 所示。



1　　伪随机序列发生器结构框图

图中 Ci 代表本原多项式 F(x)中各项的系数。

3.2.38 位伪随机序列发生器的 Verilog HDL 编码

```
/*****************************************************************************
*       Filename : rangen.v
```

```
*           Auther : Verilog group
*       Description : This module is used to generate 8-bits random number,
*                     the polynomial is x^8+x^4+x^3+x^2+1.
*         Called by :
*   Revision History : 2000-5-5
*                     Revision 1.0
*            Email : zhangnb@sz.huawei.com.cn
*          Company : Huawei Technology Inc.
*     Copyright(c) 1999,Huawei Technology Inc.,All right reserved.
********************************************************************************/

//-----------------------------
// TOP MODULE
//-----------------------------
module RANGEN (
                        Reset,   //Reset signal
                        Gclk ,   //Clock signal
                        Load ,   //Load seed to Ran_num
                        Seed ,   //initialize Ran_num
                        Ran_num  //output random number
                      ) ;

//-----------------------------
// SIGNAL DECLARATIONS
//-----------------------------
input       Reset   ;
input       Gclk    ;
input       Load    ;
input  [7:0] Seed    ;
output [7:0] Ran_num ;

//-----------------------------
// SIGNAL DECLARATIONS
```

```
//----------------------------
wire        Reset   ;
wire        Gclk    ;
wire        Load    ;
wire   [7:0] Seed    ;
reg    [7:0] Ran_num ;
integer     i       ;

//----------------------------
// PARAMETERS
//----------------------------
parameter  U_DLY=1   ;



//----------------------------
// Ran_num signal
//----------------------------
always @(posedge Reset or posedge Gclk)
begin
        if (Reset)
                Ran_num <= 8'b0 ;
        else if ( Load )
                Ran_num <= #U_DLY Seed;
        else
        begin
                for (i=1;i<8;i=i+1)
                        Ran_num[i] <= #U_DLY Ran_num[i-1] ;
                Ran_num[0] <= #U_DLY Ran_num[1] ^ (Ran_num[2] ^ (Ran_num[3] ^ Ran_num[7])) ;
        end
end

endmodule
```

3.3 双端口 RAM 仿真模型

用一个 512X8 的双端口 RAM 来实现同步 FIFO，该 RAM 的仿真模型如下所述：

```
/************************************************************\
    MODULE:       Dual Port RAM
    FILE NAME:    dualram.v
    VERSION:      2000-4-20
    AUTHOR:
    CODE TYPE:    Behavioral and RTL
    DESCRIPTION:  This module defines a Synchronous Dual Port
                  Random Access Memory.
\************************************************************/
module DUALRAM(
            Read_clock,
            Write_clock,
            Read_allow,
            Write_allow,
            Read_addr,
            Write_addr,
            Write_data,
            Read_data
            );
parameter   DLY             1;      // Clock-to-output delay. Zero
                                    // time delays can be confusing
                                    // and sometimes cause problems.

parameter   RAM_WIDTH   8;      // Width of RAM (number of bits)
parameter   RAM_DEPTH   512;    // Depth of RAM (number of bytes)
parameter   ADDR_WIDTH  9;      // Number of bits required to
                                // represent the RAM address

input                           Read_clock;     // RAM read clock
input                           Write_clock;     // RAM write clock
input       [RAM_WIDTH-1:0]     Write_data;     // RAM data input
input       [ADDR_WIDTH-1:0]    Read_addr;      // RAM read address
```

```
input       [ADDR_WIDTH-1:0]    Write_addr;     // RAM write address
input                           Read_allow;     // Read control
input                           Write_allow;    // Write control
output      [RAM_WIDTH-1:0]     Read_data;      // RAM data Output
reg [RAM_WIDTH-1:0]     Read_data;

reg [RAM_WIDTH-1:0]             Mem [RAM_DEPTH-1:0];

// Look at the rising edge of the clock
always @(posedge Write_clock) begin
    if (Write_allow)
            Mem[Write_addr] <= #DLY Write_data;
end
always @(posedge Read_clock) begin
    if (Read_allow)
            Read_data <= #DLY Mem[Read_addr];
end
endmodule
```

3.4 同步 FIFO 的设计

3.4.1 功能描述

下面的同步 FIFO 是上述的双端口 RAM 来实现的。由于读写是用同一个时钟，可以直接用 FIFO 长度计数器产生 Empty 和 Full 标志。执行一次写操作，长度计数器（Facntr）加 1，执行一次写操作，Facntr 减 1。当下一次读地址等于写地址，并且只执行读操作时，将产生 Empty 标志；当下一次写地址等于读地址，并且只执行写操作时，将产生 Full 标志。

3.4.2 设计代码

```
/*****************************************************************************\
    Filename        :       syncfifo.v
    Description     :       FIFO controller top level
                            Implements a 512x8 FIFO with common read/write clocks.
    Author          :       Verilog Group
    Revision        :       2000-04-20
```

```
    Company         :       Huawei Ltd.
\*********************************************************************/
`timescale 1ns / 10ps
module SYNCFIFO(
    Fifo_rst,               //async reset
    Clock,          //write and read clock
    Read_enable,
    Write_enable,
    Write_data,
    Read_data,
    Full,           //full flag
    Empty,          //empty flag
    Fcounter        //count the number of data in FIFO
    );

parameter               DATA_WIDTH = 8;
parameter               ADDR_WIDTH = 9;

input                           Fifo_rst;
input                           Clock;
input                           Read_enable;
input                           Write_enable;
input   [DATA_WIDTH-1:0]        Write_data;
output  [DATA_WIDTH-1:0]        Read_data;
output                          Full;
output                          Empty;
output  [ADDR_WIDTH-1:0]        Fcounter;

reg [DATA_WIDTH-1:0]    Read_data;
reg                     Full;
reg                     Empty;
reg [ADDR_WIDTH-1:0]    Fcounter;
```

```
reg [ADDR_WIDTH-1:0]    Read_addr;      //read address
reg [ADDR_WIDTH-1:0]    Write_addr;     //write address

wire        Read_allow = (Read_enable && !Empty);
wire        Write_allow = (Write_enable && ! Full);
/*************************************************************************\
    BLOCK RAM instantiation for FIFO. Module is 512x8, of which one
    address location is sacrificed for the overall speed of the design
\*************************************************************************/
DUALRAM U_RAM(
            Read_clock(Clock),
            Write_clock(Clock),
            Read_allow(Read_allow),
            Write_allow(Write_allow),
            Read_addr(Read_addr),
            Write_addr(Write_addr),
            Write_data(Write_data),
            Read_data(Read_data)
            );
/*************************************************************\
    Empty flag is set on Fifo_rst (initial), or when on the
    next clock cycle, Write Enable is low, and either the
    FIFOcount is equal to 0, or it is equal to 1 and Read
    Enable is high (about to go Empty).
\*************************************************************/
always @(posedge Clock or posedge Fifo_rst)
    if (Fifo_rst)
            Empty <= 'b1;
    else
            Empty <= (! Write_enable && (Fcounter[8:1] == 8'h0) &&
                        ((Fcounter[0] == 0) || Read_enable));
/*************************************************************\
    Full flag is set on Fifo_rst (but it is cleared on the
```

```
    first valid clock edge after Fifo_rst is removed), or
    when on the next clock cycle, Read Enable is low, and
    either the FIFOcount is equal to 1FF (hex), or it is
    equal to 1FE and the Write Enable is high (about to go Full).
\*****************************************************************/
always @(posedge clock or posedge Fifo_rst)
    if (Fifo_rst)
            Full <= 'b1;
    else
            Full <= (! Read_enable && (Fcounter[8:1] == 8'hFF) &&
                    ((Fcounter[0] == 1) || Write_enable));
/*****************************************************************\
    Generation of Read and Write address pointers.
\*****************************************************************/
always @(posedge clock or posedge Fifo_rst)
    if (Fifo_rst)
            Read_addr <= 'h0;
    else if (Read_allow)
            Read_addr <= Read_addr + 'b1;
always @(posedge clock or posedge Fifo_rst)
    if (Fifo_rst)
            Write_addr <= 'h0;
    else if (Write_allow)
            Write_addr <= Write_addr + 'b1;
/*****************************************************************\
    Generation of FIFOcount outputs. Used to determine how
    Full FIFO is, based on a counter that keeps track of how
    many words are in the FIFO. Also used to generate Full
    and Empty flags. Only the upper four bits of the counter
    are sent outside the module
\*****************************************************************/
always @(posedge clock or posedge Fifo_rst)
    if (Fifo_rst)
```

```
            Fcounter <= 'h0;
        else if ((! Read_allow && Write_allow) || (Read_allow && ! Write_allow))
         begin
            if (Write_allow) Fcounter <= Fcounter + 'b1;
            else Fcounter <= Fcounter - 'b1;
         end
    endmodule
```

3.5异步 FIFO 设计

3.5.1概述

异步 FIFO 使用完全独立的读写时钟，Empty 由读时钟产生，Full 由写时钟产生，两者关系完全异步，所以不能采用同步 FIFO 中的计数器来产生 Empty 和 Full 信号。为解决这一问题，采用了将二进制地址转换为格雷码（Gray-code）地址的方法。

3.5.2设计代码

```
/////////////////////////////////////////////////////////////////////\
    Filename        :    asyncfifo.v
    Description     :    Async FIFO controller top level
                         Implements a 512x8 FIFO with common read/write clocks.
    Author          :    Verilog Group
    Revision        :    2000-04-20
    Company         :    Huawei Ltd.
\/////////////////////////////////////////////////////////////////////*/
`timescale 1ns / 10ps
module ASYNCFIFO(
    Fifo_rst,                //async reset
    Read_clock,
    Write_clock,
    Read_enable,
    Write_enable,
    Write_data,
    Read_data,
    Full,            //Full flag
    Empty            //Empty flag
```

```
    );

    parameter                   DATA_WIDTH = 8;
    parameter                   ADDR_WIDTH = 9;
    input                               Fifo_rst;
    input                               Read_clock;
    input                               Write_clock;
    input                               Read_enable;
    input                               Write_enable;
    input       [DATA_WIDTH-1:0]        Write_data;
    output      [DATA_WIDTH-1:0]        Read_data;
    output                              Full;
    output                              Empty;
    reg                         Full;
    reg                         Empty;

    reg [ADDR_WIDTH-1:0]        Write_addrgray;
    reg [ADDR_WIDTH-1:0]        Write_nextgray;
    reg [ADDR_WIDTH-1:0]        Read_addrgray;
    reg [ADDR_WIDTH-1:0]        Read_nextgray;
    reg [ADDR_WIDTH-1:0]        Read_lastgray;

    wire                                Read_allow;
    wire                                Write_allow;
    /*************************************************************************\
        BLOCK RAM instantiation for FIFO. Module is 512x8, of which one
        address location is sacrificed for the overall speed of the design.
    \*************************************************************************/
    DUALRAM U_RAM(
                .Read_clock(Read_clock),
                .Write_clock(Write_clock),
                .Read_allow(Read_allow),
                .Write_allow(Write_allow),
```

```
        Read_addr(Read_addr),
        Write_addr(Write_addr),
        Write_data(Write_data),
        Read_data(Read_data)
        );
/*****************************************************************\
    Empty flag is set on Fifo_rst (initial), or when gray
    code counters are equal, or when there is one word in
    the FIFO, and a Read operation is about to be performed
\*****************************************************************/
always @(posedge Read_clock or posedge Fifo_rst)
    if (Fifo_rst)
        Empty <= 1'b1;
    else
        Empty <= (Emptyg || (Almostemptyg && Read_enable && ! Empty));
/*****************************************************************\
    Full flag is set on Fifo_rst (initial, but it is cleared
    on the first valid Write_clock edge after Fifo_rst is
    de-asserted), or when Gray-code counters are one away
    from being equal (the Write Gray-code address is equal
    to the Last Read Gray-code address), or when the Next
    Write Gray-code address is equal to the Last Read Gray-code
    address, and a Write operation is about to be performed.
\*****************************************************************/
always @(posedge Write_clock or posedge Fifo_rst)
    if (Fifo_rst)
        Full<= 1'b1;
    else
        Full <= (Fullg || (Almostfullg && Write_enable && ! Full));
/*****************************************************************\
    Generation of Read address pointers. The primary one is
    binary (read_addr), and the Gray-code derivatives are
    generated via pipelining the binary-to-Gray-code result.
```

```
    The initial values are important, so they're in sequence.
    Grey-code addresses are used so that the registered
    Full and Empty flags are always clean, and never in an
    unknown state due to the asynchronous relationship of the
    Read and Write clocks. In the worst case scenario, Full
    and Empty would simply stay active one cycle longer, but
    it would not generate an error or give false values.
\************************************************************/
always @(posedge Read_clock or posedge Fifo_rst)
    if (Fifo_rst)
            read_addr <= 'b0;
    else if (read_allow)
            read_addr <= read_addr + 1;
always @(posedge Read_clock or posedge Fifo_rst)
    if (Fifo_rst)
            Read_nextgray <= 9'b100000000;
    else if (read_allow)
            Read_nextgray <= { read_addr[8], (read_addr[8] ^ read_addr[7]),
                    (read_addr[7] ^ read_addr[6]), (read_addr[6] ^ read_addr[5]),
                    (read_addr[5] ^ read_addr[4]), (read_addr[4] ^ read_addr[3]),
                    (read_addr[3] ^ read_addr[2]), (read_addr[2] ^ read_addr[1]),
                    (read_addr[1] ^ read_addr[0]) };
always @(posedge Read_clock or posedge Fifo_rst)
    if (Fifo_rst)
            Read_addrgray <= 9'b100000001;
    else if (read_allow)
            Read_addrgray <= Read_nextgray;
always @(posedge Read_clock or posedge Fifo_rst)
    if (Fifo_rst)
            Read_lastgray <= 9'b100000011;
    else if (read_allow)
            Read_lastgray <= Read_addrgray;
/************************************************************\
```

```
    Generation of Write address pointers. Identical copy of *
    read pointer generation above, except for names. *
\*****************************************************************/
always @(posedge Write_clock or posedge Fifo_rst)
    if (Fifo_rst)
            write_addr <= 'b0;
    else if (write_allow)
            write_addr <= write_addr + 1;
always @(posedge Write_clock or posedge Fifo_rst)
    if (Fifo_rst)
            Write_nextgray <= 9'b100000000;
    else if (write_allow)
            Write_nextgray <= { write_addr[8], (write_addr[8] ^ write_addr[7]),
                    (write_addr[7] ^ write_addr[6]), (write_addr[6] ^ write_addr[5]),
                    (write_addr[5] ^ write_addr[4]), (write_addr[4] ^ write_addr[3]),
                    (write_addr[3] ^ write_addr[2]), (write_addr[2] ^ write_addr[1]),
                    (write_addr[1] ^ write_addr[0]) };
always @(posedge Write_clock or posedge Fifo_rst)
    if (Fifo_rst)
            Write_addrgray <= 9'b100000001;
    else if (write_allow)
            Write_addrgray <= Write_nextgray;
/*****************************************************************\
    Allow flags determine whether FIFO control logic can *
    operate. If Read_enable is driven high, and the FIFO is *
    not Empty, then Reads are allowed. Similarly, if the *
    Write_enable signal is high, and the FIFO is not Full, *
    then Writes are allowed. *
\*****************************************************************/
assign read_allow = (Read_enable && ! Empty);
assign write_allow = (Write_enable && ! Full);
/*****************************************************************\
    When the Write/Read Gray-code addresses are equal, the
```

```
    FIFO is Empty, and Emptyg (combinatorial) is asserted.
    When the Write Gray-code address is equal to the Next
    Read Gray-code address (1 word in the FIFO), then the
    FIFO potentially could be going Empty (if Read_enable is
    asserted, which is used in the logic that generates the
    registered version of Empty).

     Similarly, when the Write Gray-code address is equal to
    the Last Read Gray-code address, the FIFO is Full. To
    have utilized the Full address space (512 addresses)
    would have required extra logic to determine Full/Empty
    on equal addresses, and this would have slowed down the
    overall performance. Lastly, when the Next Write Gray-
    code address is equal to the Last Read Gray-code address
    the FIFO is Almost Full, with only one word left, and
            it is conditional on Write_enable being asserted.
\*************************************************************/
always @Write_addrgray or Read_addrgray)
    if( Write_addrgray == Read_addrgray )
            Emptyg = 'b1;
    else
            Emptyg = 'b0;
always @Write_addrgray or Read_nextgray)
    if( Write_addrgray == Read_nextgray )
            Almostemptyg = 'b1;
    else
            Almostemptyg = 'b0;
always @Write_addrgray or Read_lastgray)
    if( Write_addrgray == Read_lastgray )
            Fullg = 'b1;
    else
            Fullg = 'b0;
always @Write_nextgray or Read_lastgray)
```

```
    if( Write_nextgray == Read_lastgray )
            Almostfullg = 'b1;
    else
            Almostfullg = 'b0;
endmodule
```