# 序言

在学习verilog HDL之前需要首先了解它的本质：硬件描述语言，verilog描述的是数字电路。数字电路分为组合逻辑电路（例如常用的与门、或门、非门以及将门电路连接起来的导线)以及时序逻辑电路（锁存器、触发器）。使用verilog专用语法可以分别实现组合电路和时序电路的设计，掌握正确描述组合电路和时序电路的能力个人认为是verilog语法入门的证明。

verilog语法分为3类：可综合的verilog语法、用于验证的verilog语法、编译指令（换个说法就是在综合前的源文件预处理过程中被替换或删除的语句，比如条件综合语句和宏定义）。

下文将重点分析可综合的verilog语法以及它对应生成的数字电路。

# Verilog关键字



常用关键字如下表



# Verilog可综合语法

1. 数值表示

0：逻辑 0 或 "假"

1：逻辑 1 或 "真"

x 或 X：未知

z 或 Z：高阻

示例：

4'b1011         // 4bit 数值

32'h3022\_c0de   // 32bit 的数值

敲黑板：在实际电路中只有0和1两种状态；不定态x只用于时序仿真；高阻态z用于描述逻辑器件GPIO的双向端口，在芯片内部不存在高阻态这一状态。

1. 数据类型

**线网（wire）**

示例

wire   interrupt ;

wire   flag1, flag2 ;

敲黑板：可以理解为给电路中的导线起了个名字，用于连接发送端口与接收端口。

**寄存器（reg）**

示例

reg    flag1, flag2 ;

敲黑板：一般来讲定义reg型信号用于生成触发器，即具有状态保持的功能，在某些条件下也会变为wire型信号。定义的寄存器型变量不一定会生成触发器！

**向量**

示例

wire [8:2]  addr ;      //声明7bit位宽的线型变量addr，位宽范围为8:2

reg  [3:0]  counter; //声明4bit位宽的寄存器counter

**数组**

示例

wire [7:0]  addr\_bus [3:0] ; //由4个8bit wire型变量组成的数组

reg  [3:0]  counter  [3:0] ; //由4个4bit计数器组成的数组

敲黑板：这种方式定义了多个位宽相同的变量，reg型数组也可以理解例化了一段RAM

**参数（parameter）**

示例

parameter      data\_width = 10'd32 ;

敲黑板：表示的是常量，数据不能被修改

**整数（integer）**

示例

integer j ;

敲黑板：integer与for语句配合使用，用于辅助生成电路，例如多个相同模块的例化或多个变量的赋值。verilog中的for语句和c语言中的for语句含义完全不同！

1. 表达式与操作符

示例

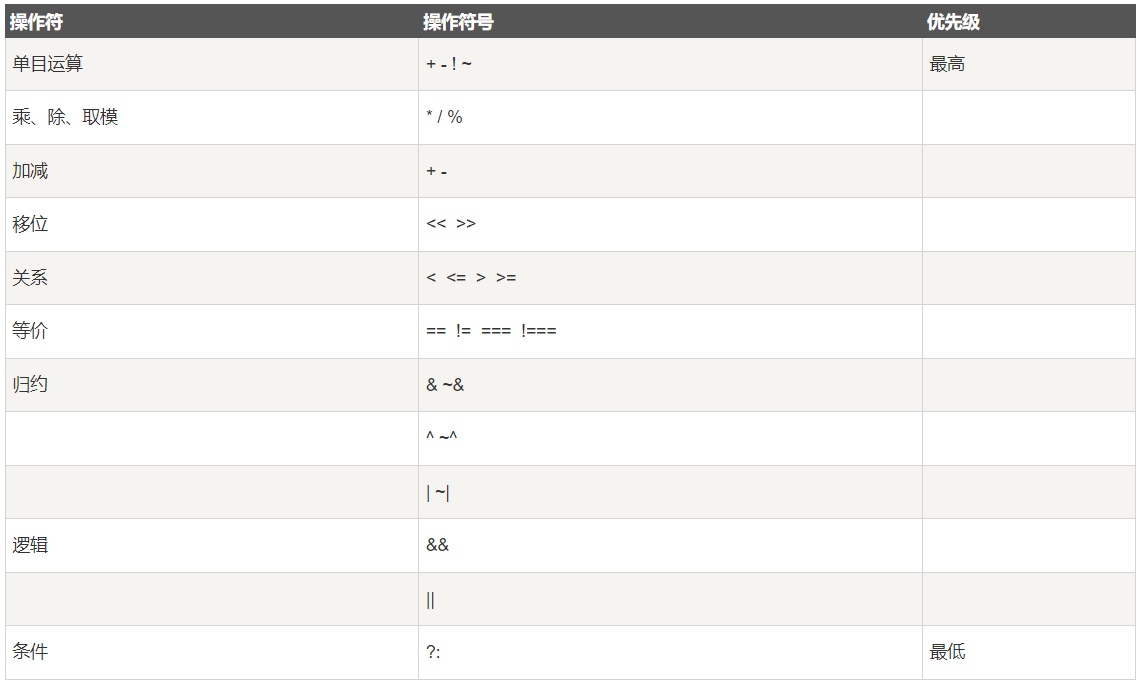
c = a + b ;

表达式由操作符和操作数构成，其目的是根据操作符的意义得到一个计算结果。表达式可以在出现数值的任何地方使用。

操作数可以是任意的数据类型，只是某些特定的语法结构要求使用特定类型的操作数。

Verilog 中提供了大约 9 种操作符，分别是算术、关系、等价、逻辑、按位、归约、移位、拼接、条件操作符。

veriolg操作符如下表



拼接操作符

示例

Y2 = {4{B}, 3'd4};

Y3 = {32{1'b0}};

敲黑板：注意运算符的优先级，建议使用（）区分优先级。注意位运算符与逻辑运算符的区别，&表示按位与；&&表示逻辑与；|表示按位或；||表示逻辑或；~表示按位取反；！表示逻辑非；===和！===在仿真中使用。

1. 赋值

**阻塞式赋值（assign）**

示例

wire      Cout, A, B ;

assign    Cout  = A & B ;     //实现计算A与B的功

敲黑板：此语句定义了一个2输入的与门，它的输出被命名为Cout，与门是A&B生成的。阻塞式赋值使用=

**非阻塞式赋值（always）**

示例

reg             byte\_ctrl\_start\_d2  ;

always @(posedge clk or negedge rst\_n) begin

    if (!rst\_n)

        byte\_ctrl\_start\_d2 <= #1 1'b0;

    else

        byte\_ctrl\_start\_d2 <= #1 byte\_ctrl\_start;

end

敲黑板：always语句中被赋值的变量必须为reg型，非阻塞式赋值使用<= 。这段代码生成了一个D触发器，它的输出信号名为byte\_ctrl\_start\_d2，在时钟上升沿对byte\_ctrl\_start进行采样。这是一个标准的D触发器电路描述，注意生成D触发器的3个要素：寄存器变量reg，时钟上升沿采样posedge clk，非阻塞试赋值<= 。在某些条件下，always语句也会生成组合逻辑或锁存器，这个问题在后面讨论。

**时延（#）**

示例

byte\_r\_done <= #1 1'b1;

敲黑板：#用于逻辑仿真，不可综合。实际的逻辑代码映射到器件后时延是真实存在的，由实际物理布线决定。

1. 条件控制语句

**if else条件语句**

示例

reg [1:0]     sout\_t ;

always @(\*) begin

    if (sel == 2'b00)

        sout\_t = p0 ;

    else if (sel == 2'b01)

        sout\_t = p1 ;

    else if (sel == 2'b10)

        sout\_t = p2 ;

    else

        sout\_t = p3 ;

end

敲黑板：这段代码生成了2个4输入1输出的数据选择器。条件（if）语句用于控制执行语句要根据条件判断来确定是否执行。条件语句用关键字 if 和 else 来声明，条件表达式必须在圆括号中。

**case多路控制语句**

示例

reg [1:0]     sout\_t ;

always @(\*) beign

    case(sel)

        2'b00:       sout\_t = p0 ;

        2'b01:       sout\_t = p1 ;

        2'b10:       sout\_t = p2 ;

        default:     sout\_t = p3 ;

    endcase

end

敲黑板：这段代码同样生成了2个4输入1输出的数据选择器。case 语句是一种多路条件分支的形式，可以解决 if 语句中有多个条件选项时使用不方便的问题。

上述两种语句在使用时需要将条件分支写全，否则容易生成锁存器，导致逻辑错误。

1. 循环语句

for循环

示例

integer      i ;

for (i=0; i<10; i=i+1) begin

    //模块例化代码

end

敲黑板：for循环一般用于同一个模块的多次例化，在源码综合前，综合工具会将for循环展开，展开后再进行综合。此外verilog还有while循环、repeat循环和forever循环，这三种循环一般用于时序仿真，在此不再介绍。

1. 语句块

数字电路是并行的，因此verilog描述的逻辑也是并行的。看如下三段语句块，假设这三段代码在一个module中：

reg             byte\_ctrl\_stop     ;

reg             byte\_ctrl\_stop\_d2  ;

always @(posedge clk or negedge rst\_n) begin

    if (!rst\_n)

        byte\_ctrl\_stop\_d2 <= #1 1'b0;

    else

        byte\_ctrl\_stop\_d2 <= #1 byte\_ctrl\_stop;

end

wire            negedge\_byte\_ctrl\_stop;

assign          negedge\_byte\_ctrl\_stop = byte\_ctrl\_stop\_d2 & (~byte\_ctrl\_stop);

always @(\*) begin

    if(byte\_count[15:0] <= 1'b1)

        trans\_count[15:0] <= #1 16'h0;

    else

        trans\_count[15:0] <= #1 byte\_count[15:0] - 1'b1;

end

上述代码包含1个assign语句块和2个always语句块，这三个语句块都是并行赋值的。其中第一个always语句块中的变量会在时钟上升沿更新数据，第二个always语句块的变量会在输入变化后立即更新，assign语句块的变量会在输入变化后立即更新。

同一个语句块中的变量也是同时更新的，例如如下代码中的四个变量。

always @(posedge clk or negedge nReset)

  if (~nReset) begin

        sSCL <= #1 1'b1;

        sSDA <= #1 1'b1;

        dSCL <= #1 1'b1;

        dSDA <= #1 1'b1;

    end

  else begin

        sSCL <= #1 scl\_i;

        sSDA <= #1 sda\_i;

        dSCL <= #1 sSCL;

        dSDA <= #1 sSDA;

    end

1. 模块端口

模块格式定义如下

module module\_name

#(parameter\_list)

(port\_list) ;

              Declarations\_and\_Statements ;

endmodule

其中parameter\_list为例化参数列表，比如配置信号位宽、工作模式。port\_list为端口列表。

示例

pca9548 #(

    parameter           CLK\_FREQ = 100          ,

    parameter           SCL\_RISE\_TIME = 100     ,

    parameter           SDA\_RISE\_TIME = 100

) (

    input               clk                     ,

    input               rst\_n                   ,

    //slave addr

    input       [7:0]   slave\_addr              ,

    //reg

    output      [7:0]   reg\_ctrl                ,

    //master port

    input               scl\_m\_i                 ,

    output              scl\_m\_oen               ,

    input               sda\_m\_i                 ,

    output              sda\_m\_oen               ,

    //slave port

    input       [7:0]   scl\_s\_i                 ,

    output      [7:0]   scl\_s\_oen               ,

    input       [7:0]   sda\_s\_i                 ,

    output      [7:0]   sda\_s\_oen

);

1. 模块例化

在一个模块中引用另一个模块，对其端口进行相关连接，叫做模块例化。

假设被例化的端口定义如下

module  ram

    #(  parameter       AW = 2 ,

        parameter       DW = 3 )

    (

        input                   CLK ,

        input [AW-1:0]          A ,

        input [DW-1:0]          D ,

        input                   EN ,

        input                   WR ,

        output reg [DW-1:0]    Q

     );

例化语句如下

pca9548 #(

    .CLK\_FREQ                   (CLK\_FREQ[7:0]          ),

    .SCL\_RISE\_TIME              (SCL\_RISE\_TIME[7:0]     ),

    .SDA\_RISE\_TIME              (SDA\_RISE\_TIME[7:0]     )

) u\_pca9548(

    .clk                        (clk                    ),

    .rst\_n                      (rst\_n                  ),

    .slave\_addr                 (ADDR\_9548[7:0]         ),

    .reg\_ctrl                   (reg\_sta\_9548[7:0]      ),

    .scl\_m\_i                    (scl\_m\_dmc\_i            ),

    .scl\_m\_oen                  (scl\_m\_dmc\_oen          ),

    .sda\_m\_i                    (sda\_m\_dmc\_i            ),

    .sda\_m\_oen                  (sda\_m\_dmc\_oen          ),

    .scl\_s\_i                    (scl\_s\_i[7:0]           ),

    .scl\_s\_oen                  (scl\_s\_oen[7:0]         ),

    .sda\_s\_i                    (sda\_s\_i[7:0]           ),

    .sda\_s\_oen                  (sda\_s\_oen[7:0]         )

);

pca9548表示模块名，u\_pca9548是被例化的模块名，被例化的模块，命名不能相同。parameter是模块参数，例化时不赋值的话，参数值就是模块中的默认值，例化时重新赋值的话，参数值就会被更新。

# 组合电路、触发器、锁存器的正确描述

1. 组合电路的描述
2. assign语句描述组合电路

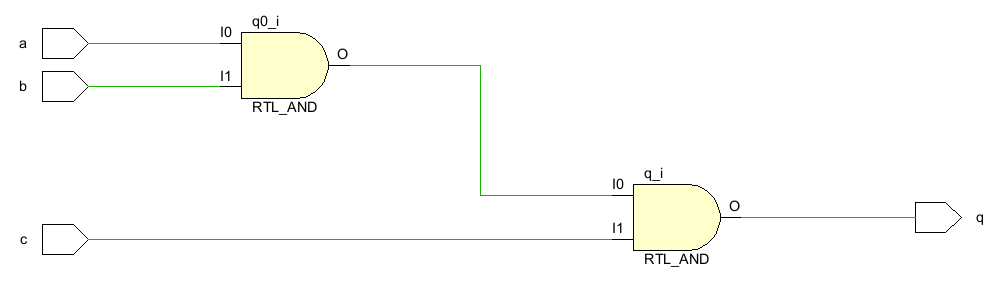
wire    a;

wire    b;

wire    c;

wire    q;

assign  q = a & b & c;



上述代码描述三输入与门. a, b, c为输入, q为输出. 综合出的电路如图.

1. always语句描述组合电路

wire    a;

wire    b;

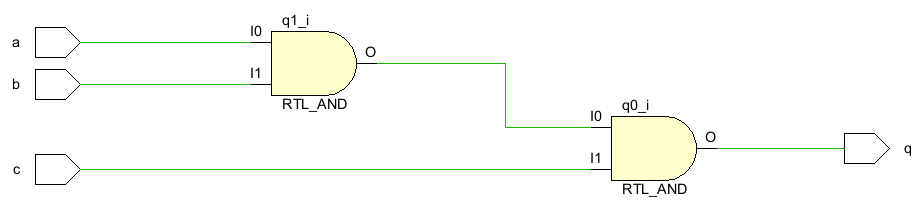
wire    c;

reg     q;

always@(\*) begin

    q <= a & b & c;

end



上述代码同样描述三输入与门. a, b, c为输入, q为输出. 综合出的电路如图.

由此可见, reg型变量不一定会生成时序电路, 生成电路的类型与赋值条件有关.

1. 触发器电路的描述

wire    clk;

wire    rst\_n;

wire    a, b, c;

reg     q;

always@(posedge clk or negedge rst\_n) begin

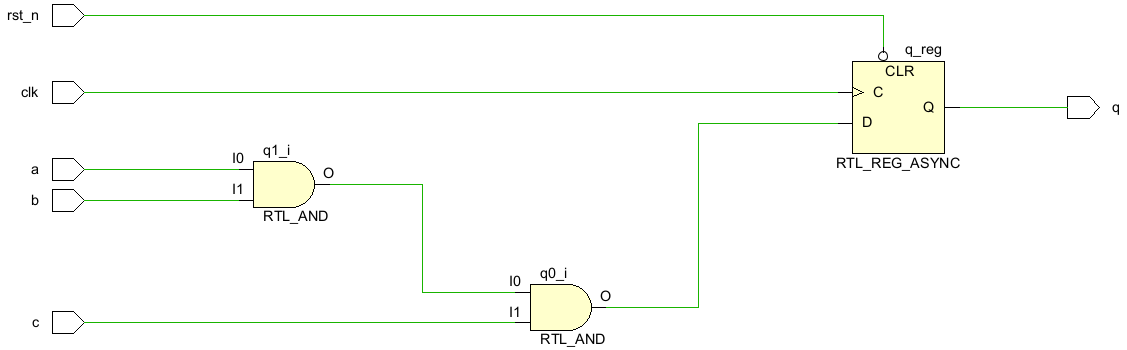
    if(!rst\_n)

        q <= 1'b0;

    else

        q <= a & b & c;

end



上述代码中, clk为触发器时钟, rst\_n为触发器异步复位信号, a, b, c为输入, q为输出. abc相与后通过D触发器将信号输出.

1. 锁存器的描述
2. 无时钟的锁存器

wire    en;

wire    a, b, c;

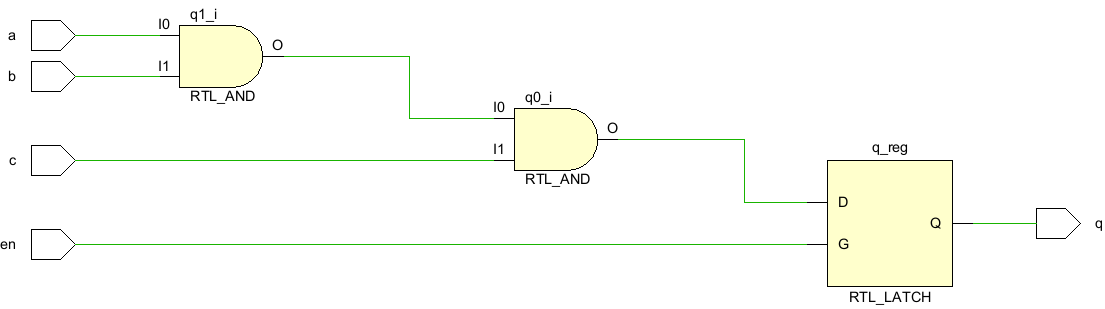
reg     q;

always@(\*) begin

    if(en == 1'b1)

       q <= a & b & c;

end



上述代码描述了一个锁存器, en为锁存使能信号, abc为输入, q为输出.

1. 有时钟的锁存器(带时钟使能信号的D触发器)

wire    clk;

wire    rst\_n;

wire    en;

wire    a, b, c;

reg     q;

always@(posedge clk or negedge rst\_n) begin

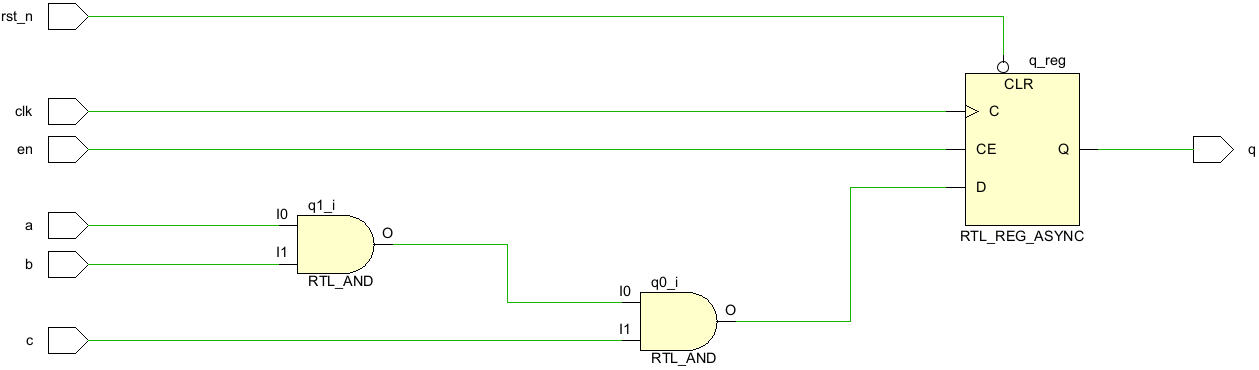
    if(!rst\_n)

        q <= 1'b0;

    else if(en == 1'b1)

        q <= a & b & c;

end



上述代码实际生成了一个带时钟使能信号的D触发器, 其中en为时钟使能信号, clk为时钟, abc为输入, q为输出.

1. 锁存器的消除

一般而言, 锁存器不是我们所期望的. 锁存器的产生通常是由条件分支覆盖不全导致的. 逻辑中的锁存器通常会导致逻辑状态异常, 比如输入发生变化但是输出状态不变. 除非你生成的电路中真的需要一个锁存器.我们继续研究第三节描述锁存器的代码.

1. 无时钟的情况

wire    en;

wire    a, b, c;

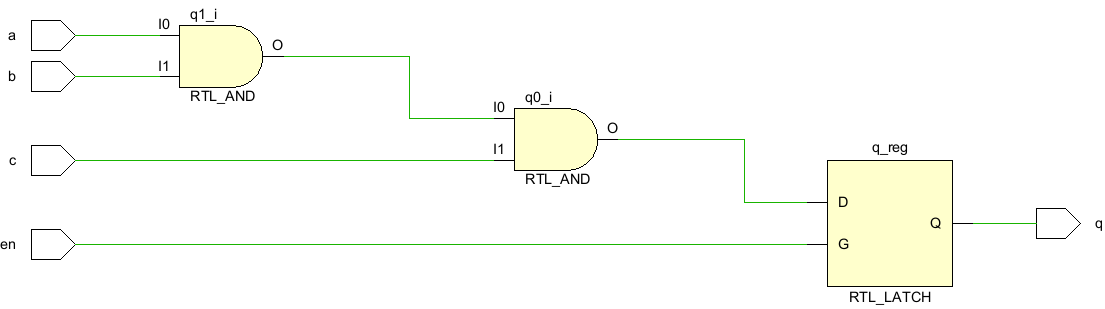
reg     q;

always@(\*) begin

    if(en == 1'b1)

       q <= a & b & c;

end



我们将条件分支补全

wire    en;

wire    a, b, c;

reg     q;

always@(\*) begin

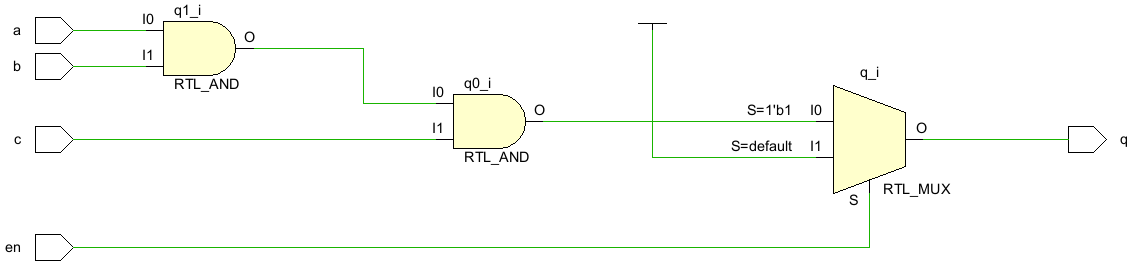
    if(en == 1'b1)

        q <= a & b & c;

    else

        q <= 1'b1;

end



红色背景的代码为新增代码, 将if分支补全后, 锁存器变成了数据选择器, 这符合我们对这段代码的预期.

当我们把条件分支赋值改为 q<=q呢?

wire    en;

wire    a, b, c;

reg     q;

always@(\*) begin

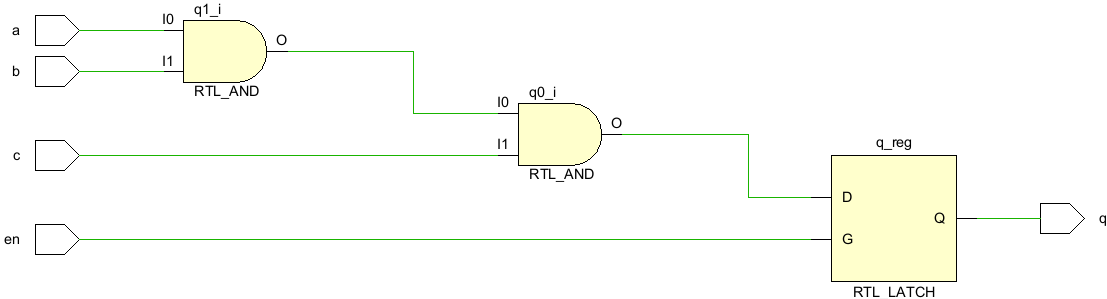
    if(en == 1'b1)

        q <= a & b & c;

    else

        q <= q;

end



生成的电路和第一种描述方式一致, en为0时, q保持, 只有当en为1时abc的输入才会传递到q, 这也符合我们对这段代码的预期.

1. 有时钟的情况

wire    clk;

wire    rst\_n;

wire    en;

wire    a, b, c;

reg     q;

always@(posedge clk or negedge rst\_n) begin

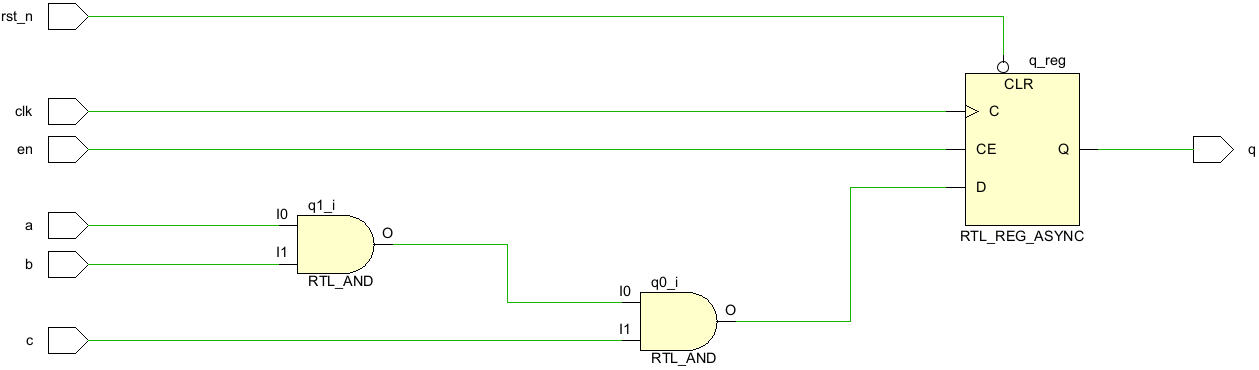
    if(!rst\_n)

        q <= 1'b0;

    else if(en == 1'b1)

        q <= a & b & c;

end



我们同样补全条件分支

wire    clk;

wire    rst\_n;

wire    en;

wire    a, b, c;

reg     q;

always@(posedge clk or negedge rst\_n) begin

    if(!rst\_n)

        q <= 1'b0;

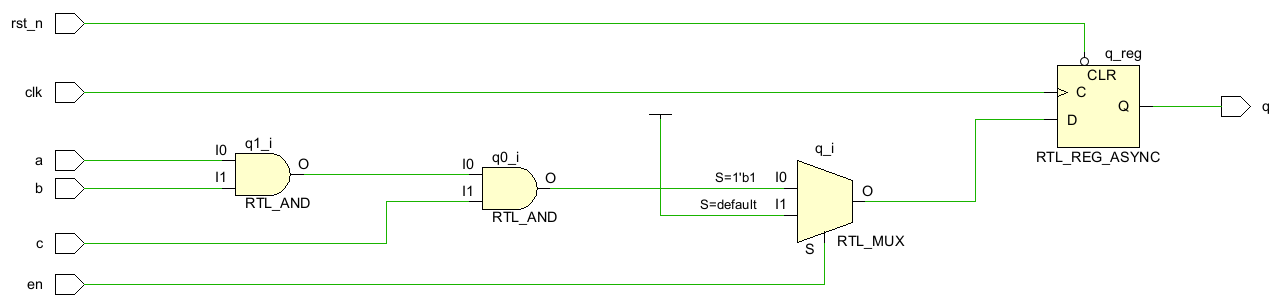
    else if(en == 1'b1)

        q <= a & b & c;

    else

        q <= 1'b1;

end



条件分支补全后, 触发器的时钟使能信号消失, 在触发器的数据输入端增加了一级数据选择器. 与无时钟情况相比, 此描述方式增加了一级触发器.

我们同样把条件分支赋值改为 q<=q呢?

always@(posedge clk or negedge rst\_n) begin

    if(!rst\_n)

        q <= 1'b0;

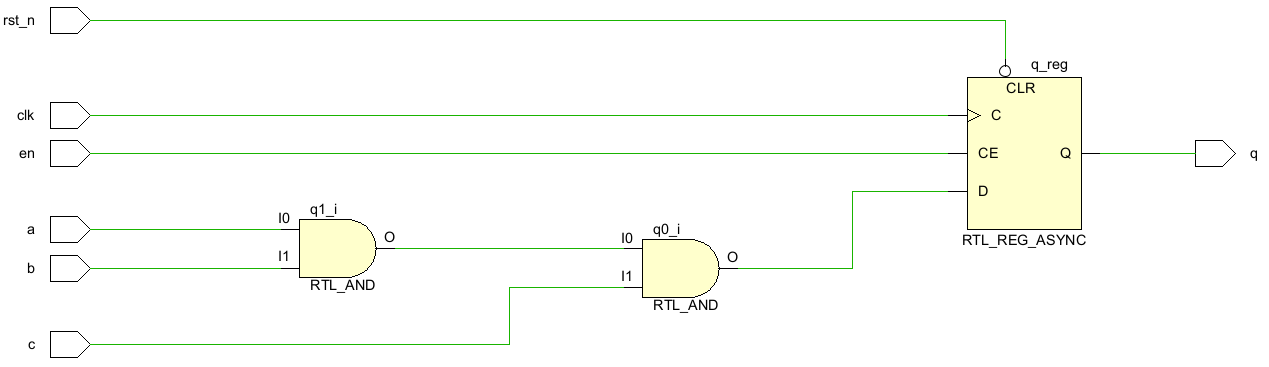
    else if(en == 1'b1)

        q <= a & b & c;

    else

        q <= q;

end



同样的, 与第一种描述方式一致.

总结一下, 生成锁存器的描述有两种(我将带时钟使能信号的触发器归纳到锁存器), 第一种,条件分支覆盖不全; 第二种, 条件分支覆盖全的条件下, 输出信号给自身赋值. 第一种情况是大多数工程师会出现的错误, 特别是在条件变量比较多的情况下.

因此我们再来讨论一下由if – else构成的条件控制语句, 假设判断条件有a, b, c三种, q为输出. 假设a, b, c为单bit信号且三个信号之间无耦合, 那么a, b, c的组合有8种, 如下:

    a   b   c

1   0   0   0

2   0   0   1

3   0   1   0

4   0   1   1

5   1   0   0

6   1   0   1

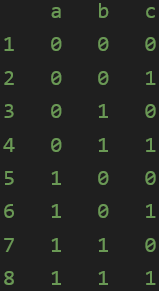
7   1   1   0

8   1   1   1

a, b, c组成的判断语句有如下几种情况, 我们来判断每条条件分支与上述组合是如何对应的

第一种

always@(posedge clk or negedge rst\_n) begin

    if(!rst\_n)

        q <= xxx;

    else if(a == 1'b0)

        q <= xxx;

    else if(b == 1'b0)

        q <= xxx;

    else if(c == 1'b0)

        q <= xxx;

    else

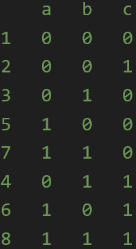
        q <= xxx;

end

第二种

always@(posedge clk or negedge rst\_n) begin

    if(!rst\_n)

        q <= xxx;

    else if((a == 1'b0) && (b == 1'b0))

        q <= xxx;

    else if(c == 1'b0)

        q <= xxx;

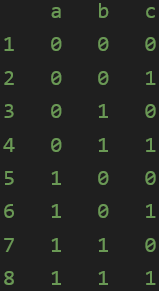
    else

        q <= xxx;

end

第三种

always@(posedge clk or negedge rst\_n) begin

    if(!rst\_n)

        q <= xxx;

    else if(a == 1'b0)

        if(b == 1'b0)

            q <= xxx;

        else

            q <= xxx;

    else

        q <= xxx;

end

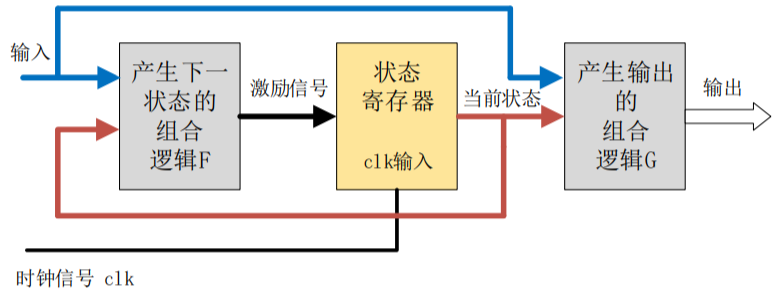
上述三种情况覆盖了所有的条件分支描述方式, 在使用不同的方式描述时需注意判断结果的覆盖状态, 避免出现状态遗漏, 产生逻辑异常. 建议在条件控制语句if的结尾增加else, 在条件控制语句case的结尾增加default, 保证覆盖所有状态.

# Verilog状态机

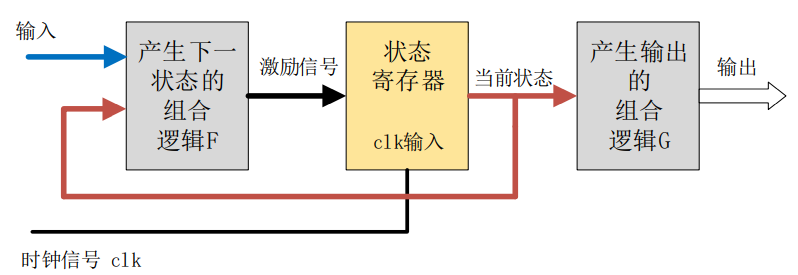
1. 状态机模型

根据状态机的输出是否与输入条件相关, 状态机可以分为2类, Moore状态机和Mealy状态机. 状态机模型中第一个方框是产生下一状态的组合逻辑; 第二个方框为状态寄存器, 用来保存当前的状态; 第三个方框为组合逻辑, 是当前状态的输出结果.

Mealy状态机 : 输出不仅取决于当前状态，还取决于输入状态.



Moore状态机 : 组合逻辑的输出只取决于当前状态，而与输入状态无关.



1. 三段式状态机模板

三段式状态机使用三个always 模块, 一个always 模块采用同步时序描述状态转移, 一个 always 采用组合逻辑判断状态转移条件, 描述状态转移规律, 另一个 always 模块描述状态输出.

Mealy状态机

//--状态机第一段：同步时序描述状态转移

always@(posedge sys\_clk or negedge sys\_rst\_n)begin

    if(!sys\_rst\_n)

        cur\_state <= IDLE;              //复位初始状态

    else

        cur\_state <= next\_state;        //次态转移到现态

end

//--状态机第二段：组合逻辑判断状态转移条件，描述状态转移规律以及输出

always@(\*)begin

    case(cur\_state)                     //组合逻辑

        IDLE:begin

            if(condition)

                next\_state = STA1;

            else

                next\_state = IDLE;

        end

        STA1:begin

            if(condition)

                next\_state = IDLE;

            else

                next\_state = STA1;

        end

        default:begin

                next\_state = IDLE;

        end

    endcase

end

//--状态机第三段：时序逻辑描述输出

always@(posedge sys\_clk or negedge sys\_rst\_n)begin

    if(!sys\_rst\_n)

        cola <= 1'b0;                   //复位、初始状态

    else

        case(cur\_state)

            IDLE:   cola <= 1'b0;

            STA1:begin

                if(condition)

                    cola <= 1'b1;

                else

                    cola <= 1'b0;

            end

            default:cola <= 1'b0;

        endcase

end

Moore状态机

//--状态机第一段：同步时序描述状态转移

always@(posedge sys\_clk or negedge sys\_rst\_n)begin

    if(!sys\_rst\_n)

        cur\_state <= IDLE;              //复位初始状态

    else

        cur\_state <= next\_state;        //次态转移到现态

end

//--状态机第二段：组合逻辑判断状态转移条件，描述状态转移规律以及输出

always@(\*)begin

    case(cur\_state)                     //组合逻辑

        IDLE:begin

            if(condition)

                next\_state = STA1;

            else

                next\_state = IDLE;

        end

        STA1:begin

            if(condition)

                next\_state = IDLE;

            else

                next\_state = STA1;

        end

        default:begin

                next\_state = IDLE;

        end

    endcase

end

//--状态机第三段：时序逻辑描述输出

always@(posedge sys\_clk or negedge sys\_rst\_n)begin

    if(!sys\_rst\_n)

        cola <= 1'b0;                   //复位、初始状态

    else

        case(cur\_state)

            IDLE:   cola <= 1'b0;

            STA1:begin

                    cola <= 1'b1;

            end

            default:cola <= 1'b0;

        endcase

end

在代码设计中, 建议状态机只负责进行状态转换与其它模块的工作状态控制, 这样可以简化状态机的逻辑. 例如通过如下方式使用计数器, 状态机只控制计数器的使能和清零, 通过检测eq信号判断计数器是否溢出, 而计数器则在独立的always语句块中运行.

reg     [7:0]   counter\_dev;

reg             counter\_dev\_en;

reg             counter\_dev\_clr;

always @(posedge clk or negedge rst\_n) begin

    if (!rst\_n)

        counter\_dev[7:0] <= #1 8'h0;

    else if( counter\_dev\_clr == 1'b1 )

        counter\_dev[7:0] <= #1 8'h0;

    else if( counter\_dev\_en == 1'b1 )

        counter\_dev[7:0] <= #1 counter\_dev[7:0] + 1'b1;

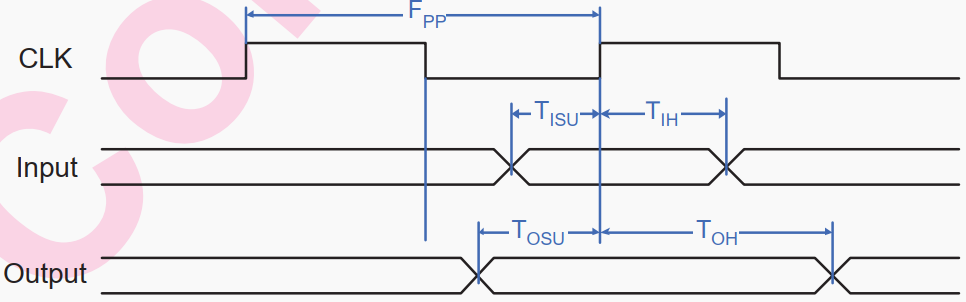
end

wire            counter\_dev\_eq;

assign          counter\_dev\_eq = (counter\_dev[7:0] == dev\_number[7:0]) ? 1'b1 : 1'b0;

# CPLD逻辑开发

1. 建立, 保持时间与亚稳态



D触发器的正常工作需要满足建立时间和保持时间上的要求. 在时钟上升沿出现之前, D值必须在一段指定的时间保持稳定. 时钟上升沿之前D保持稳定的最短时间称为建立时间, 时钟上升沿之后保持稳定的时间称为保持时间. 当输入D在建立保持时间窗口内发生变化时, 在此后的一个时钟周期内将无法确定输出电平的状态, 这种不稳定的状态被称为亚稳态. 如果亚稳态输出对用于其它逻辑门的输入, 系统的工作将发生异常. 在接下来学习时钟, 复位, 同步设计时, 我们将讨论亚稳态问题的处理方法.

1. 时钟设计

ClkDivTree u\_clkdivtree (

    .clk                        (clk\_sys\_100m               ),

    .rst\_n                      (rst\_out\_n                  ),

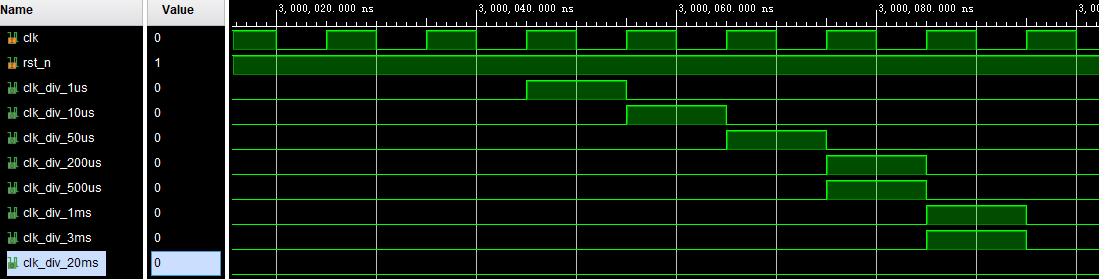
    .clk\_div\_1us                (),

    .clk\_div\_10us               (clk\_div\_10us               ),

    .clk\_div\_250ms              (clk\_div\_250ms              ),

    .clk\_dic\_500ms              (clk\_div\_500ms              )

);



项目中一般使用25M或者50M时钟作为CPLD系统时钟. 服务器CPLD开发一般逻辑相对简单, 没有使用多个系统时钟的场景, CPLD中的触发器时钟应全部使用同一时钟. 如果两级触发器之间使用不同时钟, 会出现跨时钟域问题, 造成后级触发器建立保持时间不满足, 导致出现亚稳态问题. 针对跨时钟域问题有几种处理措施, 因CPLD开发不涉及, 在此不再展开.

CPLD逻辑开发中经常会使用低频时钟进行计数, 低频时钟由系统时钟分频产生, 分频模块代码如下

ClkDiv #

(

    .MAX\_DIV\_BITS ( 7 ),

    .MAX\_DIV\_CNT  ( 99 )

)m1uSCE

(

    .iClk               ( clk ),

    .iRst               ( ~rst\_n ),

    .iCE                ( 1'b1 ),

    .oDivClk            ( w1uSCE )

);

分频模块输出信号的高电平时间为系统时钟的一个时钟周期, 如仿真波形所示. 如果分频时钟用于CPLD管脚输出, 例如SGPIO时钟, 输出时钟的占空比应调整为50%. 分频时钟应作为触发器输入的判断条件而不是直接将分频时钟作为触发器时钟源. 使用示例如下:

reg     [3:0]   counter\_ret\_por;

always@(posedge clk or negedge rst\_n) begin

    if(!rst\_n)

        counter\_ret\_por[3:0] <= #1 4'h0;

    else if(pg\_retimer == 1'b0)

        counter\_ret\_por[3:0] <= #1 4'h0;

    else if((pg\_retimer == 1'b1) && (clk\_div\_10us == 1'b1) && (counter\_ret\_por[3:0] < 4'd14))

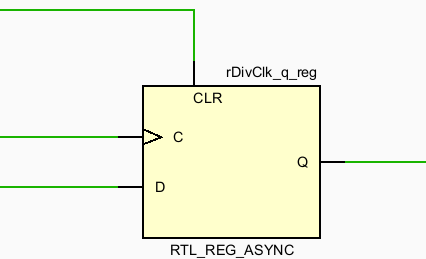
        counter\_ret\_por[3:0] <= #1 counter\_ret\_por[3:0] + 4'd1;

end

如果将分频时钟作为触发器时钟源则会产生跨时钟域问题.

1. 复位设计
2. 异步复位

采用异步复位时, 触发器存在一个复位管脚(reset\_n). reset\_n可以在任意时刻被拉低, 它与时钟边沿之间没有关系, 因此这种复位方式被称为异步复位.



当reset\_n被拉高时必须与时钟的上升沿同步, 不然会产生与违反建立保持时间类似的输出不稳定问题. cpld的逻辑复位管脚与时钟信号是异步的, 因此需要对全局复位信号做时钟同步处理.

reg     rst\_in\_sync;

reg     rst\_out\_n;

always@(posedge clk or negedge rst\_in\_n) begin

    if(!rst\_in\_n) begin

        rst\_in\_sync <= 1'b0;

        rst\_out\_n   <= 1'b0;

    end

    else begin

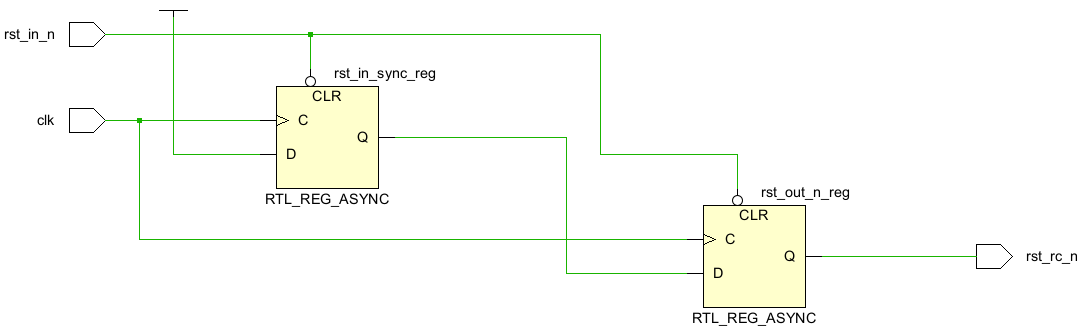
        rst\_in\_sync <= 1'b1;

        rst\_out\_n   <= rst\_in\_sync;

    end

end

assign  rst\_rc\_n = rst\_out\_n;



当rst\_in\_n有效时, 第二个触发器的输出rst\_rc\_n被立即置0, 当rst\_in\_n拉高时, 两个触发器都可能出现复位异常, 但是这种异常是可接受的, 因为rst\_rc\_n会有一个周期时间以上的时间保持低电平, 并且在时钟上升沿后从0变1.

其余逻辑使用经过同步之后的信号作为复位信号.

always@(posedge clk or negedge rst\_rc\_n) begin

    if(!rst\_rc\_n) begin

        parameter\_1 <= 1'b0;

        parameter\_2 <= 1'b0;

    end

    else begin

        parameter\_1 <= xxx;

        parameter\_2 <= xxx;

    end

end

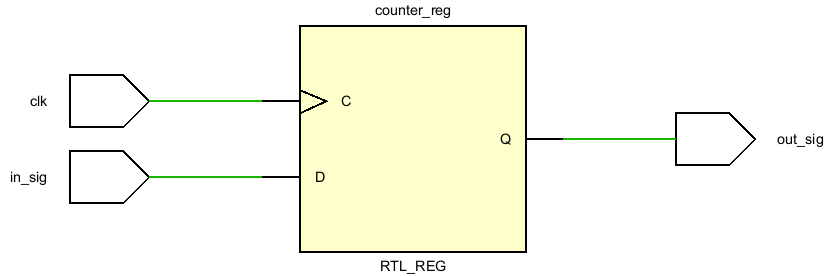
1. 同步复位

采用同步复位时, D触发器没有专用的复位管脚, 复位信号是决定触发器输入信号值的变量之一. 在时钟信号的上升沿复位信号被采样. 由于复位信号被当成输入信号的一部分, 因此它必须满足和一般数据输入一样的建立时间和保持时间要求.

reg counter;

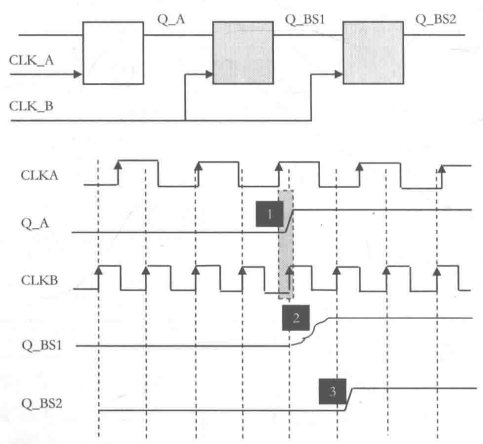
always@(posedge clk)

    counter <= in\_sig;



1. 外部信号同步

CPLD逻辑的输入来自CPLD的输入管脚, 信号经过组合电路输入到触发器, 而输入信号与CPLD系统时钟并不是同步的, 有些输入信号还可能存在毛刺, 如果不对输入信号进行同步或滤波处理, 触发器同样会存在亚稳态问题. 下图是外部信号经过2级寄存器同步的原理



输入同步代码如下:

wire            pex\_rst0\_n\_d2;

wire    [7:0]   oam\_thermtrip\_d2;

InputsSyncWithDefault  #(

    .SIZE                       (9                         ),

    .DEFAULT\_OUT                (9'h0                       )

)

u\_InputsSyncWithDefault (

    .iClk                       (clk\_sys\_100m               ),//i

    .iRst\_n                     (rst\_out\_n                  ),//i

    .ivSync                     (

                                {pex\_rst0\_n,

                                 ~oam\_thermtrip\_n[7:0]}

                                                            ),//i

    .ovSync                     (

                                {pex\_rst0\_n\_d2,

                                 oam\_thermtrip\_d2[7:0]}

                                                            ) //o

);

对于存在毛刺或上升时间过长的信号, 则使用滤波模块对信号进行滤波和同步

wire            pg\_54v\_sys\_d2;

GlitchFilter2 #(

    .NUMBER\_OF\_SIGNALS          (1                          ),

    .RST\_VALUE                  (1'h0                       )

)

GlitchFilter\_pg\_54v\_sys (

    .iClk                       (clk\_sys\_100m               ),

    .iARst\_n                    (rst\_out\_n                  ),

    .iSRst\_n                    (1'b1                       ),

    .iEna                       (clk\_div\_10us               ),

    .iSignal                    (pg\_54v\_sys                 ),

    .oFilteredSignals           (pg\_54v\_sys\_d2              )

   );

1. 双向端口设计

如下代码是i2c sda信号的例化代码, bmc\_i2c\_7\_sda是双向信号, 被绑定到CPLD管脚上, i2c\_dev\_ubb\_sda\_1\_i读取管脚电平用于内部逻辑使用, i2c\_dev\_ubb\_sda\_1\_o信号用于控制管脚的输入输出方向. 双向端口本质上是三态门, 当输出时, 三态门使能, 管脚输出由三态门输入决定, 此时CPLD管脚阻抗很低. 当输入时, 三态门关闭, 管脚电平由外部驱动器决定.

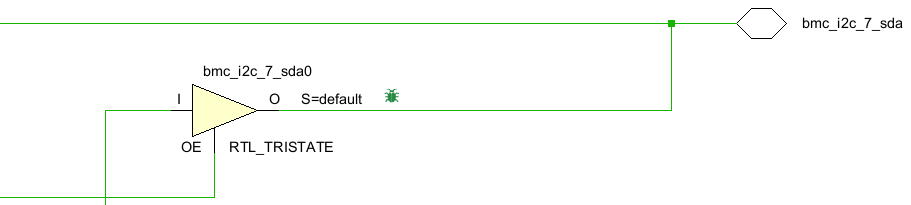
inout   wire    bmc\_i2c\_7\_sda               ,

wire            i2c\_dev\_ubb\_sda\_1\_i;

wire            i2c\_dev\_ubb\_sda\_1\_o;

assign          i2c\_dev\_ubb\_sda\_1\_i = bmc\_i2c\_7\_sda;

assign          bmc\_i2c\_7\_sda       = ( i2c\_dev\_ubb\_sda\_1\_o == 1'b1 ) ? 1'bz : 1'b0;



在CPLD模块端口设计时, 建议将双向端口分为输入, 输出, 方向控制三个信号, 在将双向端口绑定到CPLD管脚时再将这三个信号连接至三态门, 以此来提高代码设计的灵活性. i2c控制器的GPIO定义如下:

    .scl\_i      ( scl\_pad\_i             ),

    .scl\_o      ( ),

    .scl\_oen    ( scl\_padoen\_o          ),

    .sda\_i      ( sda\_pad\_i             ),

    .sda\_o      ( ),

    .sda\_oen    ( sda\_padoen\_o          )

);

如果i2c模块的端口被设计成inout模式, 在顶层例化时, CPLD的一对i2c管脚则无法同时绑定2个及以上i2c模块. 模块定义中将双向端口拆分后则不会出现这种限制, 如下代码将三个i2c模块绑定到了FPGA的同一对i2c管脚:

wire            i2c\_dev\_fpga\_scl\_i;

wire            i2c\_dev\_fpga\_scl\_o;

wire            i2c\_dev\_fpga\_sda\_i;

wire            i2c\_dev\_fpga\_sda\_o;

assign          i2c\_dev\_fpga\_scl\_o  = 1'b1;

assign          i2c\_dev\_fpga\_scl\_i  = I2C2\_BMC\_FPGA\_SCL;

assign          i2c\_dev\_fpga\_sda\_i  = I2C2\_BMC\_FPGA\_SDA;

wire            i2c\_dev\_sens\_scl\_i;

wire            i2c\_dev\_sens\_scl\_o;

wire            i2c\_dev\_sens\_sda\_i;

wire            i2c\_dev\_sens\_sda\_o;

assign          i2c\_dev\_sens\_scl\_o  = 1'b1;

assign          i2c\_dev\_sens\_scl\_i  = I2C2\_BMC\_FPGA\_SCL;

assign          i2c\_dev\_sens\_sda\_i  = I2C2\_BMC\_FPGA\_SDA;

wire            i2c\_hst\_e2p\_scl\_i;

wire            i2c\_hst\_e2p\_scl\_o;

wire            i2c\_hst\_e2p\_sda\_i;

wire            i2c\_hst\_e2p\_sda\_o;

assign          i2c\_hst\_e2p\_scl\_i   = I2C2\_BMC\_FPGA\_SCL;

assign          i2c\_hst\_e2p\_sda\_i   = I2C2\_BMC\_FPGA\_SDA;

wire            i2c\_dev\_e2p\_scl\_i;

wire            i2c\_dev\_e2p\_scl\_o;

wire            i2c\_dev\_e2p\_sda\_i;

wire            i2c\_dev\_e2p\_sda\_o;

assign          i2c\_dev\_e2p\_scl\_i   = e2p\_i2c\_scl;

assign          i2c\_dev\_e2p\_sda\_i   = e2p\_i2c\_sda;

assign          e2p\_i2c\_scl         = ( i2c\_dev\_e2p\_scl\_o == 1'b1 ) ? 1'bz : 1'b0;

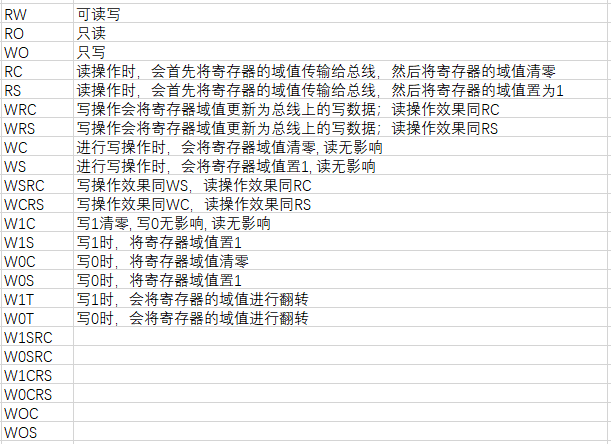
assign          e2p\_i2c\_sda         = ( i2c\_dev\_e2p\_sda\_o == 1'b1 ) ? 1'bz : 1'b0;

assign          I2C2\_BMC\_FPGA\_SCL   = ( (i2c\_dev\_fpga\_scl\_o & i2c\_dev\_sens\_scl\_o & i2c\_hst\_e2p\_scl\_o) == 1'b1 ) ? 1'bz : 1'b0;

assign          I2C2\_BMC\_FPGA\_SDA   = ( (i2c\_dev\_fpga\_sda\_o & i2c\_dev\_sens\_sda\_o & i2c\_hst\_e2p\_sda\_o) == 1'b1 ) ? 1'bz : 1'b0;

1. 寄存器电路设计

寄存器读写模型如下



逻辑器件通常作为从设备与bmc通信, bmc与逻辑器件之间的常用接口为i2c或local bus.以i2c接口为例, 介绍常用的寄存器模型RO RW W0C定义方式

i2c slave模块定义如下:

i2c\_slave #(

    .CLK\_FREQ       (CLK\_FREQ           )

)u\_i2c\_slave(

    .clk            (clk                ),

    .rst\_n          (rst\_n              ),

    .slave\_addr     (8'h60              ),   //器件地址

    .reg\_addr       (regaddr[7:0]       ),   //寄存器地址

    .wr\_en          (wr\_en              ),   //master写

    .data\_out       (data\_out[7:0]      ),   //master发送的数据

    .data\_in        (data\_in[7:0]       ),   //master接收的数据

    .scl\_pad\_i      (scl\_slave\_i        ),   //SCL-line input

    .sda\_pad\_i      (sda\_slave\_i        ),   //SDA-line input

    .sda\_pad\_oen    (sda\_slave\_oen      )    //SDA-line output enable

);

寄存器定义如下:

reg             event\_heartbeat\_trigger;

reg             event\_heartbeat\_trigger\_clr;

always @(posedge clk or negedge rst\_n) begin

    if (!rst\_n)

        event\_heartbeat\_trigger <= #1 1'h0;

    else if(event\_heartbeat\_trigger\_clr == 1'h0 )

        event\_heartbeat\_trigger <= #1 1'h0;

    else if( wire\_heartbeat\_trigger == 1'b1 )

        event\_heartbeat\_trigger <= #1 1'h1;

end

always @(\*) begin

    case (regaddr[7:0])

        8'h00:      data\_in[7:0]    =   reg\_test[7:0];

        8'h01:      data\_in[7:0]    =   firmware\_ver[7:0];

        8'h2B:      data\_in[7:0]    =   {1'h0, event\_heartbeat\_trigger, 6'h0};

        default:    data\_in[7:0]    =   8'hff;

    endcase

end

wire            wr\_en;

wire    [7:0]   data\_out;

always @(posedge clk or negedge rst\_n) begin

    if (!rst\_n) begin

        reg\_test[7:0]                       <= #1 8'h0;

        event\_heartbeat\_trigger\_clr         <= #1 1'h1;

    end

    else if(wr\_en == 1'b1) begin

        case(regaddr[7:0])

            8'h00:      reg\_test[7:0]                       <= #1 data\_out[7:0];

            8'h2B:  begin

                        event\_heartbeat\_trigger\_clr         <= #1 data\_out[6];

                    end

        endcase

    end

    else begin

        event\_heartbeat\_trigger\_clr         <= #1 1'h1;

    end

end

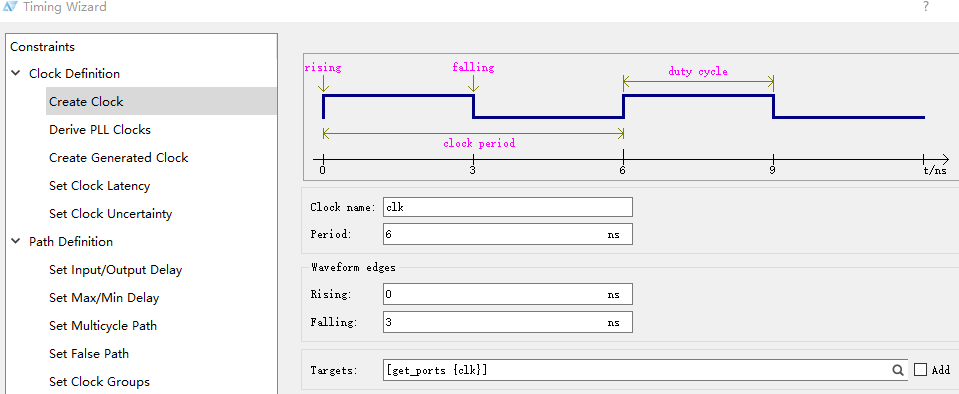
其中reg\_test为RW寄存器, 地址为0x00; firmware\_ver为RO寄存器, 地址为0x01; event\_heartbeat\_trigger与event\_heartbeat\_trigger\_clr共同组成W0C寄存器, 地址为0x2B.

1. 器件工作的最高频率

逻辑器件的系统时钟频率选择比较灵活, 时钟频率越高, 逻辑开发会更容易一些. 项目中逻辑能够运行的最高频率受2方面因素影响: 第一个方面受使用的逻辑器件自身性能影响(例如工艺, 制程, 芯片架构), 公司常用CPLD速率上限在100M左右. XILINX 7系列FPGA速率上限在250M左右. 第二个方面受逻辑布线影响, 触发器之间的信号路径越长, 组合电路级联的层级越多, 信号传播延时越大, 触发器建立保持时间的裕量就越小.

针对实际项目, 我们可以使用逻辑器件的开发工具进行时序分析. 通过设置时序约束来判断工程的时序裕量是否充足(CPLD开发中如果时钟频率在50M以下且只用一个系统时钟则基本不需要考虑时序裕量问题).

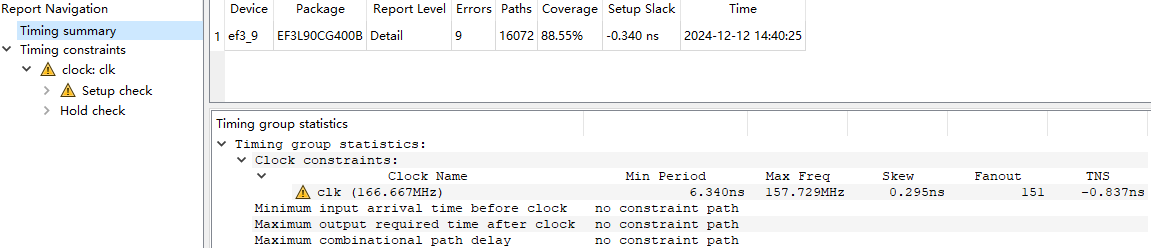
以安路CPLD为例, 通过创建系统时钟, 告诉综合工具当前逻辑实际的工作频率

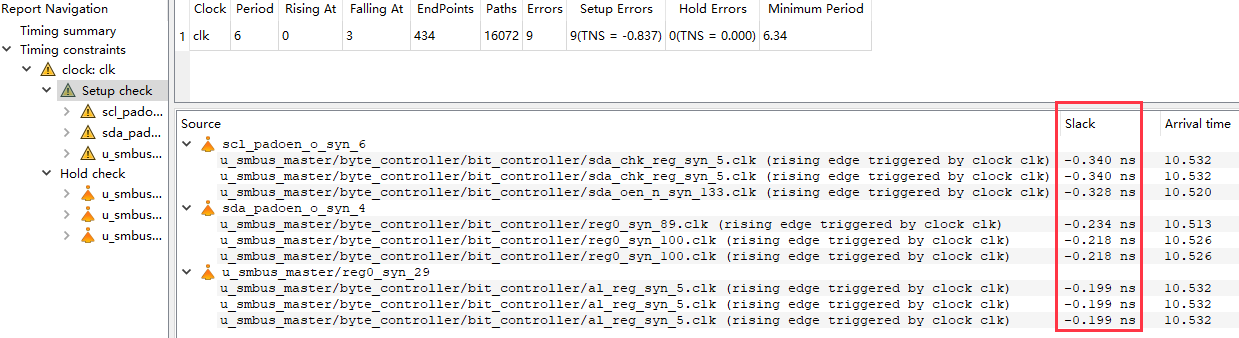


如图定义了一个周期为6ns的系统时钟(167M), 生成的约束文件如下:

create\_clock -name clk -period 6 -waveform {0 3} [get\_ports {clk}]

综合后的产生的报告如下:

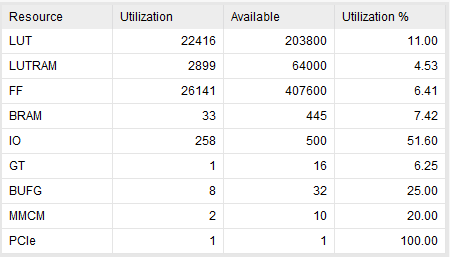




报告说明当前工程能够工作的最高频率为157.729M, 有9个时钟路径建立时间Slack为负值, 表示时序违规. 此时我们应降低系统时钟频率或更换速度等级更高的器件.

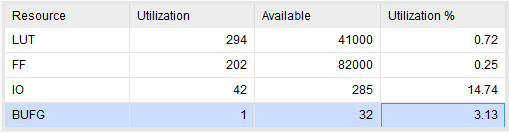
1. 资源使用评估

在进行资源评估之前, 我们先看一下逻辑器件都有哪些资源, 下图是某个FPGA项目的逻辑资源使用报告, 其中LUT是查找表, 它是搭建组合电路的基本单元; FF是寄存器, 用于搭建时序电路中的各种寄存器. LUT和FF数量是评估逻辑器件规模最重要的指标. 其余资源是逻辑器件的IP核, 不同型号的逻辑器件IP资源也有较大差异.

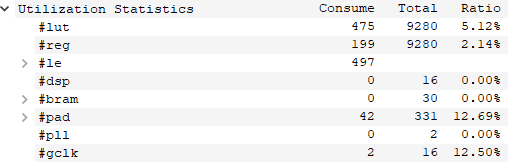


不同厂家综合工具的资源利用率会存在比较大的差异, 下图是XILINX, ANLOGIC, lattice综合相同代码的资源使用情况. 在进行资源评估时应根据具体逻辑器件型号下每种模块的资源消耗情况进行评估. 总资源消耗可以按照单模块消耗的资源 X 数量再加20%的裕量进行评估.

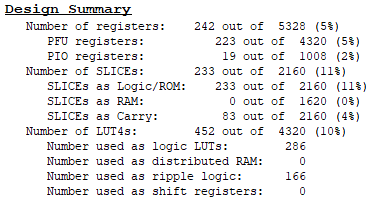
XILINX



ANLOGIC



Lattice



郝英杰

2024/12/12