研究管理部文档中心	文档编号	产品版本	密级
		1.0	机密
	文档名称：大规模逻辑设计指导书		共140页

大规模逻辑设计指导书

第一篇 方法论

(仅供内部使用)

文档作者：		日期：	2000/03/17
审 核 人：		日期：	2000/03/18
批 准 人：		日期：	yyyy/mm/dd

修订记录

日期	修订版本	描述	作者
2000/03/17	1.00	初稿完成	

目 录

第一章 VHDL语言编写规范

1 目的

编写该规范的目的是提高书写VHDL代码的可读性、可修改性、可重用性；优化代码综合和仿真的结果，指导设计工程师使用VHDL规范代码和优化电路，规范化公司的ASIC/FPGA设计输入，从而做到：① 逻辑功能正确，②可快速仿真，③ 综合结果最优，④ 可读性较好。

2 范围

该规范涉及VHDL编码风格、规定，编码中应注意的问题，VHDL代码书写范例等。

该规范适用于所有的采用VHDL代码进行设计的项目。

3 定义

VHDL：Very high speed IC Hardware Description Language, 甚高速集成电路的硬件描述语言

FSM：Finite Status Machine,有限状态机

simulate：仿真，通过输入激励在计算机上验证设计是否正确。包括RTL仿真和门级仿真。

模拟：是指对一个物理器件的结构、功能或其他特性如延时特性等用抽象的语言或高级语言（如用C语言进行算法描述）所进行的建模。

4 引用标准和参考资料

下列标准包含的条文，通过在本标准中引用而构成本标准的条文。在标准出版时，所示版本均为有效。所有标准都会被修订，使用本标准的 各方应探讨使用下列标准最新版本的可能性。

VHDL For Programmable Logic	Mr,Kevin Shahill USA

5 规范内容

以下内容中，有关的保留字用黑体标识。对不作为审核的内容用“建议”字眼标识。

5.1 VHDL编码风格

本章节中提到的VHDL编码规则和建议适应于 VHDL的任何一级（RTL， behavioral, gate_level)，也适用于出于仿真，综合或二者结合的目的而设计的模块

5.1.1 标识符（Identifiers)命名习惯

标识符用于定义实体名、结构体名、信号和变量名等，选择有意义的命名对设计是十分重要的。命名包含信号或变量诸如出处、有效状态等基本含义，下面给出一些命名的规则，包括VHDL语言的保留字。

1. 标识符定义命名规定

- 标识符第一个字符必须是字母，最后一个字符不能是下划线，不许出现连续两个下划线。
- 基本标识符只能由字母、数字和 下划线组成。
- 标识符两词之间须用下划线连接。

  如：*Packet_addr, Data_in, Mem_wr, Mem_ce*
- 标识符不得与保留字同名，VHDL保留字见附录6.1。

2. 标识符大小写规定

- 对常量、数据类型、实体名和结构体名采用全部大写；
- 对变量采用小写；
- 对信号采用第一个词首字符大写。
- 保留字一律小写。

3. 信号名连贯缩写的规定

长的名字对书写和记忆会带来不便，甚至带来错误。采用缩写时应注意同一信号在模块中的一致性。一致性的缩写习惯有利于文件的阅读理解和交流。

部分缩写的统一规定为：

Addr address； Clk clock； Clr clear；Cnt counter

En enable ； Inc increase；Lch latch；Mem memory

Pntr pointer； Pst preset ； Rst reset ；

Reg register； Rd reader ； ；Wr write

常用多个单词的缩写：

ROM RAM CPU FIFO ALU CS CE

自定义的缩写必须在文件头注释。

4. 信号名缩写的大小写规定

- 单词的缩写若是信号名的第一个单词则首字符大写，如：Addr_in中的Addr 。若该单词缩写不是第一 个单词则小写， 如：Addr_en 中的en 。
- 多个单词的首字符缩写都大写，不管该缩写在标识符的什么位置， 如：RAM_addr，Rd_CPU_en。

5. 信号名一致性规定

同一信号在不同层次应保持一致性。

6. 信号命名有关建议

- 建议用有意义而有效的名字，能简单包含该信号的全部或部分信息，如输入输出信息：Data_in （总线数据输入）、Din（单根数据线输入）、FIFO_out （FIFO数据总线输出）；如宽度信息：Cnt8_q （8位计数器输出信号的命名）。
- 建议添加有意义的后缀，使信号名更加明确，常用的后缀如下：

后缀	意义
_clk	时钟信号
_d	寄存器的数据输入信号
_q	寄存器的数据输出信号
_z	连到三态输出的信号
_L	_L后加数字，表示信号延迟时钟周期数，如_L1
_s	实体端口信号的反馈信号
_en	使能控制信号
_n	求反的信号
_xi	芯片原始输入信号
_xo	芯片原始输出信号
_xod	芯片的漏极开路输出
_xz	芯片的三态输出
_xbio	芯片的双向信号

说明：

1. 采用D触发器对信号进行延迟，延迟信号的命名在原信号名之后加后缀_L,若是在流水线设计中有级延迟，再分别加后缀_L1，_L2，..... 。 L 表示lock。

2. 模块内的反馈信号，在原信号名之后加后缀_s 。“s”表示 same。

5.1.2 数据对象和类型

1. 类型使用规定

- VHDL是很强的类型语言，可综合的数据类型为标量类型（包含可枚举类型、整型、浮点型、物理类型）和组合类型（包含记录、数组），模拟模型的数据类型为存取类型、文件型。可综合的VHDL代码的编写不采用模拟类型、浮点型、物理类型。
- 不同基本类型的数据不能由另一类型赋值，不同类型间的赋值需使用运算符的重载。如：Cnt8_q 为STD_LOGIC_VECTOR 类型，若不对‘+’运算符重载，则 Cnt8_q <= Cnt8_q + 1 语句在综合中将出错。可通过对 + 运算符进行重载 即使用**use IEEE**.std_logic_arith.**all** 语句，则上句赋值语句是正确的 。
- 常量名和数据类型必须用大写标识符表示。

2. 数据及数据类型使用建议

- 为改善代码的可读性，建议可把常用的常量和自定义的数据类型在程序包中定义。
- 建议使用别名来标识一组数据类型有利于代码的清晰。如：

signal Addr： STD_LOGIC_VECTOR(31 **downto** 0)**;**

alias Top_addr： STD_LOGIC_VECTOR(3 **downto** 0) **is** Addr(31 **downto** 28);

3. 数据使用注意内容

可枚举类型的值为标识符或单个字母的字面量，是区分大小写的，如 Z 与z 将是两个不同的量。

5.1.3 信号和变量

1. 信号不许赋初值。
2. 变量使用建议

变量主要用在高层次的模拟模型建模及用于运算的用途，但变量的综合较难定义，对于编写可综合的VHDL模块，在没有把握综合结果情况下建议不使用。

3. 信号、变量使用注意内容

- 在VHDL中，信号(signal)代表硬件连线，因此可以是逻辑门的输入输出，同时，信号也可表达存贮元件的状态。端口也是信号。
- 在进程（process）中，信号是在进程结束时被赋值。因此，在一个进程中，当一个信号被多个信号所赋值时，只有最后一个赋值语句起作用。如下例：

```
Sig_p： process(A，B，C）
    begin
        D <=  A；----- ignored!!
        X <= C or D；
        D <= B；---- overrids !!
        Y <= C xor D；
    end；
```

上面实际的结果是 B赋值给D， （C xor B）结果赋值给X、Y。

- 变量不能表达连线或存贮元件。变量的赋值是直接的、非预设的。变量将保持其值直到对它重新赋值。如下例：

```
Ver_p：process(A，B，C）
        variable d： STD_LOGIC；
    begin
        d： = A；
        X <= C or d；
        d： = B；
        Y <= C xor d；
    end process；
```

实际结果是 X <= C **or** A，Y<= C **xor** B。

5.1.4 实体

1. 实体、结构体使用规定

- library IEEE；use IEEE.std_logic_1164.all； 除IEEE大写外，其余小写。
- 实体名和结构体名必须用大写标识，实体名必须与文件名同名。自定义的其他标识符如信号名、变量名、标号等不得与实体名、结构体名同名。
- 实体端口数据模式不准使用buffer 模式

缓冲模式主要用在实体内部可读的端口，如计数器的输出 。为简化大型设计各模块间接口的配合，要求不要使用。需要反馈的信号可定义内部信号来解决。如计数器端口Count ，可内部定义信号 **signal** Cnt8_q ： STD_LOGIC_VECTOR(7 **downto** 0)；

Cnt8_q 该信号可在内部反馈，最后通过 赋值语句：

Count <= Cnt8_q；来实现端口的定义。

- 实体端口数据类型规定

实体端口的数据类型采用IEEE std_logic_1164 标准支持的和提供的最适合于综合的数据类型STD_ULOGIC、STD_LOGIC和这些类型的数组。不采用**IEEE** 1076 /93 标准支持和提供的BIT 、BIT数组、INTEGER及其派生类型。这是为保证模拟模型和综合模型的一致性及减少转换时间和错误。

- 一个文件只对应一个实体。

实体是设计文件的基本单元，其书写规范要求如下：

- 一条语句占用一行，每行应限制在80个字符以内。
- 如果较长（超出80个字符）则要换行。
- 代码书写要有层次即层层缩进格式、清晰、美观。
- 要有必要的注释（25%）
- 实体开始处应注明文件名、功能描述、引用模块、设计者、设计时间及版权信息等。

如：

```
-- Filename ：
-- Author ：
-- Description ：
-- Called by ：Top module
-- Revision History ：99-08-01
--                    Revision 1.0
-- Email ： M@huawei.com.cn
--Company ： Huawei Technologies .Inc
--Copyright(c) 1999, Huawei Technologies Inc, All right reserved

library IEEE;
use IEEE.std_logic_1164.all;
entity ENTITY_NAME
port(
    Port1 : in STD_LOGIC;
    Port2 : in STD_LOGIC;
   Port3 : out STD_LOGIC;
   ..
   Portn : out STD_LOGIC
```

```
);
end ENTITY_NAME ;
architecture BEHAVIOR of ENTITY_NAME is
begin
        Statements;
end BEHAVIOR ;
```

2. **实体使用建议**

- **实体名的命名建议能大致反映该实体的功能如**：COUNTER8（8**位宽的计数器模块**），DECODER38（3-8**线译码器模块**）。
- **行为级的结构体名命名为** BEHAVIOR，**结构级的结构体名命名为** STRUCTURE，**若有多个结构体，用后缀**A、B..... **命名。如：**

```
architecture BEHAVIOR of COUNTER8 is
begin
    .....
end BEHAVIOR;
```

- **一个实体可以有多个结构体，对单个结构体的实体，文件要包含结构体和实体说明，便于查阅。对多个结构体的实体，建议把常用的结构体放在文件中，其余结构体用单独文件表示，使用时用configuration 语句进行配置。**
- **结构体的描述分为行为级描述、数据流描述和结构化描述。若无特殊要求，建议采用行为级描述和数据流的描述，不采用结构化描述或BOOLEAN 数据流的描述。**
- VHDL**设计中，如果可以避免采用器件厂商的专用元件库（硬**Core），**则尽量不要使用，除非只有采用该库元件才能实现你设计的性能指标。这是因为要充分利用**VHDL**独立于工艺且易于维护的优点。**

3. **实体使用注意内容**

- VHDL**设计应是层级型的设计。**VHDL**设计实体由实体说明和结构体组合而成。实体是一个设计的基本单元模块。即顶层的设计模块由次一级的实体构成，每个次一级实体又可由再下一层次的实体构成。最低层模块可以是表达式或最基本的实体模块构成。这种设计方法就是** Top-To-down **的设计方法。**
- **实体端口模式为 in，out，buffer，inout。模式为in 的信号不能被驱动，模式out的信号不能用于反馈，同时必须仅被一个信号所驱动；缓冲模式的端口不能被多重驱动（除非用决断函数解决外）同时仅可以连接内部信号或另一个实体的缓冲模式的某个端口。**
- VHDL**设计各模块接口定义时要考虑模块间配合的方便，如实体端口的模式，端口的数据类型等。**

5.1.5 语句

1. VHDL**各语句使用规定**

- **with-select-when 语句书写规范规定**

with- select - when 语句提供选择信号赋值，是根据选定信号的值对信号赋值。代码的书写规范为：

```
with selection_signal select
        Select_name <= value_a when value_1_of_selection_signal，
                       value_b when value_2_of_selection_signal，
                       value_c when value_3_of_selection_signal，
                       .....
                       value_x when last_value_of_selection_signal;
```

- **with- select - when 语句的**selection_signal**的所有值必须具备完整性，若没写完整必须有一个 others 语句，如下三个写法，其综合的效果是一致的，因为**S**的元素不是已知的逻辑值，**X**将不被定义，但对**RTL**仿真而言，其结果是不一致的，这是因为**RTL**仿真支持多值元素。**

- ```
  with S select
       X <= A when 00 ,
            B when 01 ,
            C when 10 ,
            D when others;

  with S select
            X <= A when 00 ,
            B when 01 ,
            C when 10 ,
            D when 11 ,
            D when others;

  with S select
            X <= A when 00 ,
            B when 01 ,
            C when 10 ,
            D when 11 ,
              -- when others;
  ```

  建议不使用第三种写法。

- **with- select - when 语句中对有相同的支项可合并书写 如**X <= A **when** “00” | “10”。

- **when_else 语句书写规范规定**

  when_else 语句提供为条件信号赋值，即一个信号根据条件被赋一值，代码书写规范为：

  ```
  Signal_name <= value_a when condition1 else
                 value_b when condition2 else
  ```
  ```

```
                value_c  when  condition3 else
                 .....
                value_x;
```

- 当条件是表达式时，表达式须用（）括起来，使代码更为清晰。如：

  when （a = b and C= ‘1’ ）else

- if 必须有一个else 对应（除在如下面例子的情况下可不写else语句）。

  例：

```
process( Clk,Rst)
begin
        if ( Rst = '1') then
                Q <= '0';
        elsif ( Clk ‘ event and Clk = ‘ 1’ ) then
                Q <= D;
        end if;
end  process;
```

  当没有else语句时，将产生不希望的存储器。

- case- when 语句书写规范规定

  该语句用于规定一组根据给定选择信号的值而执行的语句，可用with-select-when 语句等效。

  代码的书写规范为：

```
case-  selection_signal is
    when value1_of _selection signal =>
            Statements1;
    when value2_of _selection signal =>
            Statements2;
    ....
    when last_value_of _selection signal =>
            Statements x ;
    when others =>
            Statements x;
    end case;
```

- case_when 语句必须有 when others 支项。
- 若信号在if -else 或case-when 语句作非完全赋值，必须给定一个缺省值。
- process 显示敏感列表必须完整

  对有Clk 的 process ，不同综合工具有不同的要求，有些只要写Clk和Rst就可，建议根据具体情况简化设计书写。

- 有clk的process 的敏感列表中，为方便修改，敏感列表书写规范如下：

```
Lab ： process（Clk, Rst，
```

```
                    list1，list2,....）
        begin
```

- **每个process 前须加个**lable。
- **不同逻辑功能采用不同的process 进程块，把相同功能的放在同一进程中，如触发器组进程块：**

```
        D_p  : process(Clk,Rst,
                    D1,D2,...,Dn)
            begin
               if  (Rst =  '1' ) then
                            Q1 <= '0' ;
                            Q2 <= '0' ;
                            ...
                            Qn <= Dn;
               elsif (Clk 'event and Clk = '1') then
                            Q1 <= D1;
                            Q2 <= D2;
                            ...
                            Qn <= Dn;
                end if;
            end process;
```

- **generate 语句书写规范规定**

在需要重复生成多个器件如多个器件的重复例化时，使用生成机构可简便代码书写。如下32位总线的三态缓冲器的例化：

```
        Gen_lab1: for  I  in 0 to 31 generate
                inst_lab: threestate  port  map(
                            Din => Value(i),
                            Rd  => Rd,
                            Dout => Value_out(i)
                            );
        end generate;
```

- **生成机构必须有一个标号，如上的**Gen_lab1
- **if -then 用在生成机构中，不能有else 或elsif 语句，如下复杂的生成机构语句：**

```
        G1：for I in 0 to 3 generate
                G2：  for  j in 0 to 7 generate
                        G3：if (I <1 )  then generate
                                Ua : thrst  port map( Val(j),Rd,Val_out(j));
                        end generate;
```

G4：**if** (i = 1) **then generate** .....

- **port map 语句书写规范规定**

```
Uxx ： Module_name
 port map （
      port1  => port 1,
      port2  => port2,
       .....
      portn  => port n
);
```

- **为便于阅读，port map 采用 名字对应**（=>）**映射方法**。
- **port map 中总线到总线映射时** （X **downto** Y）**要写全**。
- **向量采用降序方法即** X **downto** Y **格式，向量有效位顺序的定义为从大数到小数**。
- **port map 的**module （**设计者自编写的entity**）**名用**Uxx**标识**，cell（**如厂家提供的库元件**、RAM、Core**等**）**名用**Vxx **标识**。

2. VHDL **语句使用建议**

- **作为可综合的代码编写，** ‘-’**值建议不用。如下一个代码：**

```
with  tmp select
   X <=   A  when  “1---”,
          B   when “-1--”,
          C   when  “--1-”,
          D  when  “---1”,
           ‘0’  when others ;
```

该代码在RTL**级仿真中不会出错，但在综合过程中可能编译出错，视综合工具而定**。

- **由于不同综合工具支持能力问题，建议不采用wait 语句即不使用隐式敏感表**。

3. VHDL **语句使用注意内容**

- **when_else 语句具有优先级，第一个when 条件级别最高，最后一个最低，可用顺序语句的if-else 替代。书写时必须考虑敏感路径**。
- **当信号的值为不相关的值时，最好用选择信号赋值语句，如多路选择器；当信号的值为相关时，选用when-else 语句，如编写优先编码器**。
- **注意If --elsif ---elsif ---else的优先级。最后一个else 优先级最低。必须把关键路径放在优先级高的语句中**。

5.1.6 **运算符**(operator)

1. **表达式书写规定**

为便于理解，用（）**表示逻辑运算符执行的优先级。如：**

```
X <= ( A and B ) and ( C or  （not  D）);
```

建议运算操作符两边都加上空格。如：

2. 比较运算符规定

向量比较时，比较的向量的位宽要相等，否责会引起warning或error。除非重载等值比较运算符（调用numeric_std 库）。

5.1.7 function

1. function 使用规定

- function 代码书写规范规定

```
function  FUNCTION_NAME (参数1：参数类型，参数2：参数类型 .....）
return  返回类型 is
begin
        顺序语句;
end ;
```

实例见附录6.3。

2. function 使用建议

- 函数主要用于类型的转换或重载运算符的定义，对于使用IEEE1164标准的面向综合的VHDL设计，采用std_logic 类型，不必考虑与bit类型的转换，可调用 numeric_std 标准程序包实现类型转换和 + 运算符的重载。建议少用厂家提供的函数或自定义的类型转换函数。
- 对多次重复的表达式可用一个函数来定义。

3. function 使用注意内容

- 函数参数只能是输入类型，不能被赋值修改。
- 只能有一个返回值。
- 定义函数必须为顺序语句，且其中不能定义新的信号，但可在函数说明域中说明新的变量，并在定义域中对其赋值。
- 函数在结构体说明域中或程序包中定义。见附录6.3。

5.1.8 procedure

1. procedure 使用规定

- procedure 书写规范规定

```
procedure  PROCEDURE_NAME （signal 参数名： 模式 类型;
                            signal 参数名： 模式 类型;
                        ...） is
begin
        过程体;
end procedure ;
```

2. procedure 使用注意内容

- 过程用于数值运算、类型转换、运算符重载或设计元件的最高层设计结构。
- 过程参数缺省模式为 in。

5.1.9 类属(generics)

1. **generic 使用注意内容**

- **类属为传递给实体的具体元件的一些信息，如器件的上升下降沿的延时信息（对应类属为** rise、fall）、**用户定义的数据类型如负载信息（对应类属为**load）**及数据通道宽度、信号宽度等**。
- **对大型设计，建议使用类属来构造参数化的元件**。**其调用的方法为**：

Uxx : **参数化的实体名 generic map（实参）**

port map (

端口映射表;

）;

见附录6.5。

- **若元件的类属在定义时已经指定默认值，在调用时，若不改变该参数值可不用定义实参的映射即map（实参）可不写**。

5.1.10package

1. **package 使用建议**

- **对大型设计，建议把全局的常量（如数据宽度等）、指令状态编码、元件组、函数和子程序组分别用元素包、器件包、函数包来构造。如通过调用元件程序包，实体的结构体说明区域中就不必再对调用的器件进行component 说明**。
- **程序包通过use 语句使之可见。可通过保留字all使包中所有单元都可见。如**：

use work..yourpacketname.**all**;

其中yourpacketname**是你的**packet**名**。

2. **package 使用注意内容**

- **程序包包括程序包说明和可选包体。程序包说明用来声明包中的 类型、元件、函数和子程序；包体则用来存放说明中的函数和子程序**。
- **不含有子程序和函数的程序包不需要包体。（见附录**6.4）
- **程序包中的类型、常量、元件、函数和其他说明对其他设计单元是可见的**。

5.1.11 FSM（有限状态机）

1. FSM **使用规定**

- VHDL **的**FSM**为双进程的有限状态机，即组合逻辑进程定义次态的取值，时序逻辑进程定义时钟上升沿来时次态成为现态**。
- VHDL**的**FSM **不必为状态分配，而用行为级的枚举类型定义状态，根据需要在综合时选择状态的分配方式如** one-hot **或二进制编码**。
- **状态机在上电时必须明确进入一个初始状态**。
- **必须包括对所有状态都处理，不能出现无法处理的状态，不能使状态机进入死循环**。

5.1.12Comments

1. Comments **使用建议**

- **对更新的内容尽量要做注释**。
- **模块端口信号要做简要的功能描述**。

- 语法块做简要介绍。

5.1.13 TAB键间隔

对TAB键的间隔，我们建议采用4个字符，这与许多软件的缺省设置是一致的。并且，VHDL语言中大多数保留字也是4个字符。

6 代码模块划分

模块设计的好坏直接影响着系统的设计好坏，模块设计的不好，会给后面的设计流程带来许多麻烦。设计模块的基本原则是：

有利于模块的可重用性

模块设计得好，可节省大量的重复工作，并且为以后的设计带来方便。

1. 在组合电路设计中应当没有层次

可提高代码的可读性，另外一方面是综合的时候方便，并且时序较易满足。

2. 每个模块输出尽量采用寄存器输出形式

这样设计是有利于时序的满足。

3. 模块的按功能进行划分，划分要合理
4. 模块大小应适中，不能太大，也不能太小，一般为2000门左右。具体情况，应当依据综合工具的性能而定。
5. 模块的层次应当至少有三级

可将一个设计划分为三个层次：TOP、MID、功能CORE

- TOP

包括实例化的MID和输入输出定义（如果用综合工具插入管脚则可不要此层次）；

- MID

由两部分组成：1）时钟产生电路，如分频电路和倍频电路；2）功能CORE的实例化。

- 功能CORE

包括各种功能电路的设计。一个复杂的功能可以分成多个子功能来实现，即再划分子层。

7 代码编写中容易出现的问题

7.1 资源共享问题

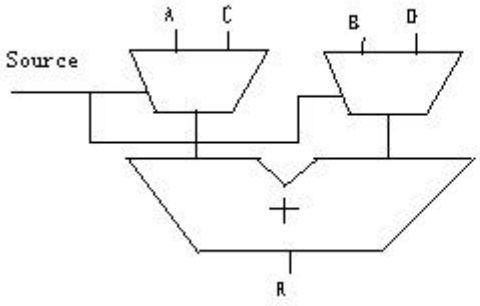
资源共享的主要思想是通过数据缓冲或多路选择的方法来共享数据通路的工作单元。在VHDL设计中资源共享必须与敏感路径问题进行综合考虑后采用适当的设计方法。

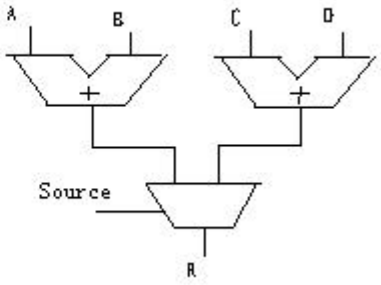
如：

```
R  <= ( A + B) when (Source = ‘ 1’ ) else
                ( C + D) ;
```

综合工具可综合成以下两种情况，当在综合工具选 中Resource Sharing 时，将资源共享一个加法器。这时，A、B、C或D到R是关键路径。如果关键路径是Source 到R，则不要要求资源共享。需要资源共享时可改成如下写法（建议采用该方式）：

```
R  <=  A  when (Source = ‘ 1’ ) else
       C;
S  <=  B when (Source = ‘ 1’ ) else
       D;
F  <=  R + S;
```





7.2 组合逻辑描述的多种方式

对组合逻辑的描述有多种方式，其综合结果是等效的。以4bit的与门为例：

C <= A and B;

等效于

C（3） <= A（3） and B（3）；

C（2） <= A（2） and B（2）;

C（1） <= A（1） and B（1）;

C（0） <= A（0） and B（0）;

等效于

```
for I in 3 downto 0 loop
  C（i）< = A（i） and B（i）;
end loop;
```

可以选择简洁的写法.

7.3 考虑综合的执行时间

通常会推荐将模块划分得越小越好，事实上要从实际的设计目标、面积和时序要求出发。好的时序规划和合适的约束条件要比电路的大小对综合时间的影响要大。要依照设计的目标来划分模块，对该模块综合约束的scripts也可以集中在该特性上。要选择合适的约束条件，过分的约束将导致漫长的综合时间。最好在设计阶段就做好时序规划，通过综合的约束scripts来满足时序规划。这样就能获得既满足性能的结果，又使得综合时间最省。

7.4 避免使用Latch

使用Latch必须有所记录，不希望使用Latch时，应该对将条件赋值语句写全，如在if语句最后加一个else，case 语句加others 。

不完整的if 和case 语句导致不必要的latch的产生，下面的语句中，DataOut会被综合成锁存器。如果不希望在电路中使用锁存器，它就是错误。

```
process（Cond)
begin
    if (Cond = ‘ 1’) then
        Data_out <= Data_in；
    end;
end process;
```

7.5 多赋值语句案例：三态总线

一根总线上挂多个三态电路时必须用多个进程来表示，如：

```
Tri_p1: process (Sel_a,A)
begin
        if (Sel_a = ‘ 1’ ) then
                T <= A;
        else
                T <= ‘ Z’ ;
        end if;
end process;

Tri_p2: process (Sel_b,B)
begin
        if (Sel_b = ‘ 1’ ) then
                T <= B;
        else
                T <= ‘ Z’ ;
        end if;
end process;
```

为什么不能在一个process中进行处理呢？如下所示：

```
Error : process (Sel_a,A,Sel_b,B)
begin
        if (Sel_a = ‘ 1’ ) then
```

```
            T <= A;
        else
            T <= ' Z' ;
        end if;
        if (Sel_b = ' 1' ) then
            T <= B;
        else
            T <= ' Z' ;
        end if;
    end process;
```

这是因为：上述两个if语句彼此没有优先级，又由于是对同一个信号（信号T）进行处理，则后一个处理会覆盖前一个处理。正确的做法是将这两个if 语句放在两个process中进行。

注意：只有三态电路才可以在多个process中出现，其它非三态电路若是在多个process中出现的话，有的综合工具会报告“短路”错误即多驱动问题，但在语法检查时不一定报错。

8 附录

8.1 VHDL保留字

VHDL**语言的保留字如下：**

absaccess	*after*	*alias*	*all*	*and*	*architecture*	*array*	*assert*	*attribute*			
begin	*block*	*buffer*	*bus*	*case*	*component*	*configuration*		*constant*			
disconnect		*downto*	*else*	*elsif*	*end*	*entity*	*exit*	*file*	*for*	*function*	
generate		*generic*	*group*	*guarded*		*if*	*impure*	*in*	*inertial*	*inout*	
is	*label*	*library*	*linkage*	*literal*	*loop*	*map*	*mod*	*nand*	*new*	*next*	*nor*
not	*null*	*of*	*on*	*open*	*or*	*others*	*out*	*package*		*port*	
postponed		*procedure*		*process*	*pure*	*range*	*record*	*register*	*reject*	*rem*	
report	*return*	*rol*	*ror*	*select*	*severity*	*signal*	*shared*	*sla*	*sll*	*sra*	*srl*
subtype	*then*	*to*	*transport*		*type*	*unaffected*		*units*	*until*	*use*	
variable	*wait*	*when*	*while*	*with*	*xnor*	*xor*					

另外，对不同的厂家，也有相应的保留字要求，可参见相应厂家提供的资料。

8.2 VHDL 编写范例

```
-- Filename ：Div5.vhd
-- Author ：zhouzhijian
```

```
-- Description ：Five division
-- Called by ：Top module
-- Revision History ：99-08-01

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

entity DIV5 is
    port(
            Rst             : in STD_LOGIC;
            Mclk            : in STD_LOGIC;
            Div5_clk        : out STD_LOGIC
    );
end DIV5;
architecture BEHAVIOR of DIV5 is
    signal Cnt3_q    : STD_LOGIC_VECTOR(2 downto 0);
    signal Cnt3_d    : STD_LOGIC_VECTOR(2 downto 0);
    signal Div0      : STD_LOGIC;
    signal Div1      : STD_LOGIC;
begin
    Cnt_pd : process(Cnt3_q)
    begin
            if (Cnt3_q = “100”) then
                    Cnt3_d <= “000”;
            else
                    Cnt3_d <= Cnt3_q + 1;
            end if;
    end process;

    Cnt_pq : process(Rst,Mclk)
    begin
            if (Rst = '1') then
                    Cnt3_q <= "000";
            elsif (Mclk = '1' and Mclk'event) then
                    Cnt3_q <= Cnt3_d;
```

```
        end if;
    end process  ;

    Div0_P : process(Rst,Mclk)
    begin
        if (Rst = '1') then
            Div0 <= '1';
        elsif( Mclk = '1' and Mclk'event ) then
            if (Cnt3_q = "100") then
                Div0 <= '1';
            elsif (Cnt3_q = "001") then
                Div0 <= '0';
            else
                Div0 <= Div0;
            end if;
         end if;
    end process ;

    Div1_P : process(Rst,Mclk)
    begin
        if (Rst = '1') then
            Div1 <= '0';
        elsif (Mclk = '0' and Mclk'event) then
            Div1 <= Div0;
        end if;
    end process  ;

    Div_clk <= Div0 or Div1;
end BEHAVIOR;
```

建议用双进程来实现带有复杂的同步复位、置位的寄存器信号设计，如上计数器的写法。对 Div0**也可类似地书写**。

8.3 函数书写实例

以下是一个使用多表决函数的全加器

-- Filename ：FullAdd.vhd

```
-- Author ：suwenbiao
-- Description ：A example of function
-- Called by ：Top module
-- Revision History ：99-08-01

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity FULLADD is
port(
    A                   : in STD_LOGIC;
    B                   : in STD_LOGIC;
    Carry_in            : in STD_LOGIC;
    Sum                 : out STD_LOGIC;
    Carry_out           : out STD_LOGIC
);
end FULLADD;

architecture BEHAVIOR of FULLADD is
    function Majority (A,B,C : STD_LOGIC)
    return STD_LOGIC is
    begin
        return (( A and b) or (A and C) or ( B and C ));
    end Majority;
begin
    Sum <= A xor B xor Carry_in;
    Carry_out <= Majority(A,B,Carry_in);
end BEHAVIOR;
```

8.4 程序包书写实例

```
-- Filename ：My_pkg.vhd
-- Author ：suwenbiao
-- Description ：A example of Package
-- Called by ：Top module
-- Revision History ：00-03-18
```

```
library IEEE;
use IEEE.std_logic_1164.all;
package MY_PKG is
    -- Defined constant

    constant NUM64K : integer := 65636;
    constant EXT_RAM_ADD_WIDTH : integer := 16;
    constant HW_NUM : integer := 4;
    constant CELL_OUTPUT_CHANNEL_WIDTH : integer := 5;
    constant INPUT_CHANNEL_WIDTH : integer := ADDRESS_BUS_WIDTH;
    constant QUEUE_WIDTH : integer := 5;
    constant WORD_LENGTH_WIDTH : integer := 6;

--Define subtype
constant QUEUE_OVERFLOAT_WIDTH : integer := 16;

subtype QUEUE_OVERFLOAT_VECTOR is
    STD_LOGIC_VECTOR(QUEUE_OVERFLOAT_WIDTH-1 downto 0);

--Defined Reg group
component Rddf1
port(
    Clk     : in STD_LOGIC;
    Rst     : in STD_LOGIC;
    D       : in STD_LOGIC;
    Q       :out STD_LOGIC
    );
end component;
component Rreg1
port(
    Clk             : in STD_LOGIC;
    Rst             : in STD_LOGIC;
    Load            : in STD_LOGIC;
    D               : in STD_LOGIC;
    Q               :out STD_LOGIC
    );
```

```
end component;
component Rreg
    generic(Size      : integer  := 2);
port(
    Clk               : in STD_LOGIC;
    Rst               : in STD_LOGIC;
    Load              : in STD_LOGIC;
    D                 : in STD_LOGIC_VECTOR( Size - 1 downto 0);
    Q                 :out STD_LOGIC _VECTOR( Size - 1 downto 0)
    );
end component;
end Reg_pkg;
```

8.5 参数化元件实例

```
-- Filename ：Reg_group.vhd
-- Author ：suwenbiao
-- Description ：A example of Reg Group with generic size
-- Called by ：Top module
-- Revision History ：00-03-18

library IEEE;
use IEEE.std_logic_1164.all
entity REG_GROUP is
    generic (Size : integer : = 2);
    port(
            Clk       : in STD_LOGIC;
            Rst       : in STD_LOGIC;
            Load      : in STD_LOGIC;
            D         :in STD_LOGIC_VECTOR(Size - 1 downto 0);
            Q         : out STD_LOGIC_VECTOR(Size - 1 downto 0)
    );
end REG_GROUP;
architecture  BEHAVIOR  of REG_GROUP is
begin
    R_p: process(Clk,Rst)
```

```
    begin
        if (Rst = ‘ 1’ ) then
            Q <= (others => ‘ 0’ ) ;
        elsif(Clk ‘ event  and Clk = ‘ 1’ ) then
            Q <= D;
        end if;
    end process;
end BEHAVIOR;
```

调用语句：

```
U1：  REG_GROUP  map(4)
port map (
    Clk          =>     Clk,
    Rst          =>     Rst,
    Load         =>     Load,
    D            =>     D,
    Q            =>     Q
);
```

则该寄存器数据宽度为4。

第二章VERILOG语言编写规范

1 目的

本规范的目的是提高书写代码的可读性、可修改性、可重用性，优化代码综合和仿真的结果，指导设计工程师使用VerilogHDL规范代码和优化电路，规范化公司的ASIC设计输入，从而做到：① 逻辑功能正确，②可快速仿真，③ 综合结果最优（如果是hardware model)，④ 可读性较好。

2 范围

本规范涉及Verilog HDL编码风格，编码中应注意的问题，Testbench的编码等。

本规范适用于Verilog model的任何一级（RTL，behavioral, gate_level)，也适用于出于仿真、综合或二者结合的目的而设计的模块。

3 定义

Verilog HDL： Verilog 硬件描述语言

FSM ： 有限状态机

伪路径 ： 静态时序分析（STA）认为是时序失败，而设计者认为是正确的路径。

4 引用标准和参考资料

下列标准包含的条文，通过在本标准中引用而构成本标准的条文。在标准出版时，所示版本均为有效。所有标准都会被修订，使用本标准的各方应探讨，使用下列标准最新版本的可能性。

Verilog Style and Coding Guidelines	Sun Microsystems Revision 1.0
Actel HDL Coding Style Guider	

5 规范内容

5.1 Verilog 编码风格

本章节中提到的Verilog编码规则和建议适应于 Verilog model的任何一级（RTL，behavioral, gate_level)，也适用于出于仿真，综合或二者结合的目的而设计的模块。

5.1.1

选择有意义的信号和变量名，对设计是十分重要的。命名包含信号或变量诸如出处、有效状态等基本含义，下面给出一些命名的规则。

- **用有意义而有效的名字**

  有效的命名有时并不是要求将功能描述出来，如

  For (I = 0; I < 1024; I = I + 1)

  Mem[I] <= #1 32'b0;

  For **语句中的循环指针**I **就没必要用**loop_index**作为指针名**。

- **用连贯的缩写**

  长的名字对书写和记忆会带来不便，甚至带来错误。采用缩写时应注意同一信号在模块中的一致性。缩写的例子如下：

  Addr address

  Pntr pointer

  Clk clock

  Rst reset

- **用最右边的字符下划线表示低电平有效，高电平有效的信号不得以下划线表示，短暂的引擎信号建议采用高有效。**

  如：*Rst_ ，Trdy_，Irdy_， Idsel.*

- **大小写原则**

  名字一般首字符大写，其余小写（但parameter, integer **定义的数值名可全部用大写），两个词之间要用下划线连接。**

  如：*Packet_addr, Data_in, Mem_wr Mem_ce_*

- **全局信号名字中应包含信号来源的一些信息。**

  如：D_addr[7:2]，**这里的“**D**”指明了地址是解码模块**(Decoder module)**中的地址。**

- **同一信号在不同层次应保持一致性。**

- **自己定义的常数、类型等用大写标识**

  如：parameter CYCLE=100；

- **避免使用保留字**

  如：*in，out，x，z*等不能够做为变量、端口或模块名

- **添加有意义的后缀，使信号名更加明确，常用的后缀如下：**

后缀	**意义**
_Clk	**时钟信号**
_next	**寄存前的信号**
_z	**连到三态输出的信号**
_f	**下降沿有效的寄存器**
_xi	**芯片原始输入信号**
_xo	**芯片原始输出信号**
_xod	**芯片的漏极开路输出**
_xz	**芯片的三态输出**
-xbio	**芯片的双向信号**

5.1.2 Modules

- 顶层模块应只是内部模块间的互连。

  Verilog设计一般都是层次型的设计，也就是在设计中会出现一个或多个模块，模块间的调用在所难免。可把设计比喻成树，被调用的模块就是树叶，没被调用的模块就是树根，那么在这个树根模块中，除了内部的互连和模块的调用外，尽量避免再做逻辑，如不能再出现对reg变量赋值等。这样做的目的是为了更有效的综合，因为在顶层模块中出现中间逻辑，Synopsys 的design compiler 就不能把子模块中的逻辑综合到最优。

- 每一个模块应在开始处注明文件名、功能描述、引用模块、设计者、设计时间及版权信息等。

  如：

```
/* ==========================================*\
          Filename  : RX_MUX.v
          Author    :
          Description :
          Called by  : Top module
        Revision History  : 99-08-01
                            Revision 1.0
          Email  : M@sz.huawei.com.cn
          Company  : Huawei Technology .Inc
      Copyright(c) 1999, Huawei Technology Inc, All right reserved
    \* ==========================================*/
```

- 不要对Inpu t进行驱动, 在module 内不要存在没有驱动的信号，更不能在模块端口中出现没有驱动的输出信号，避免在仿真或综合时产生warning，干扰错误定位。
- 每行应限制在80个字符以内，以保持代码的清晰、美观和层次感。

  一条语句占用一行，如果较长（超出80个字符）则要换行。

- 电路中调用的 module名用Uxx标示。向量大小表示要清晰，采用基于名字（name_based）的调用而非基于顺序的（order_based）。

```
Instance      UInstance2(
              .DataOut        (DOUT        ),
              .DataIn         (DIN         ),
              .Cs_                    (Cs_         )
              );
```

- 用一个时钟的上沿或下沿采样信号，不能一会儿用上沿，一会儿用下沿。如果既要用上沿又要用下沿，则应分成两个模块设计。建议在顶层模块中对Clock做一非门，在层次模块中如果要用时钟下沿就可以用非门产生的Posedge Clk_， 这样的好处是在整个设计中采用同一种时钟沿触发，有利于综合。基于时钟的综合策略。
- 在模块中增加注释。

  对信号、参量、引脚、模块、函数及进程等加以说明，便于阅读与维护。

- Module 名要用大写标示，且应与文件名保持一致。

  如：Module *DFF_ASYNC_RST*(
  Reset,
  Clk,
  Data,
  Qout
  );

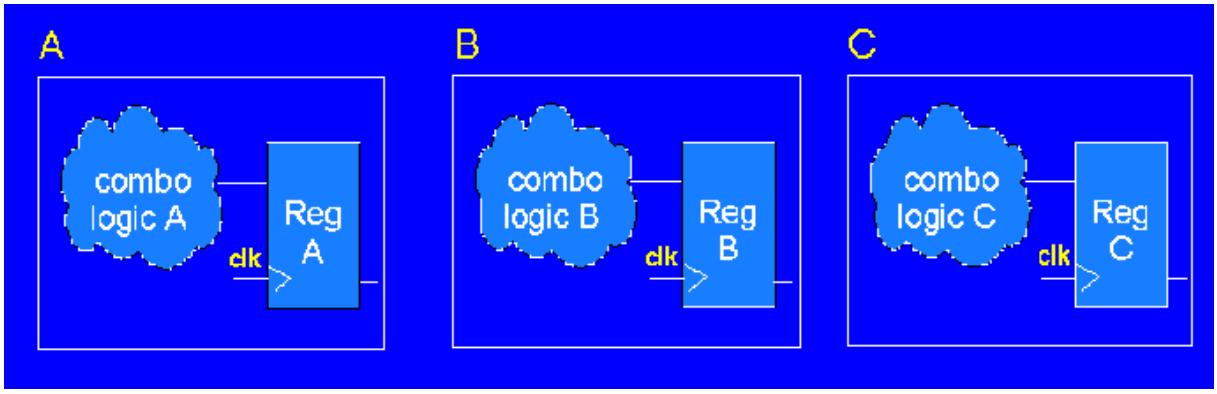
  严格芯片级模块的划分

  只有顶层包括IO引脚(pads)，中间层是时钟产生模块、JTAG、芯片的内核(CORE)，这样便于对每个模块加以约束仿真，对时钟也可以仔细仿真。

  模块输出寄存器化

  对所有模块的输出加以寄存（如图1），使得输出的驱动强度和输入的延迟可以预测，从而使得模块的综合过程更简单。

  - 输出驱动的强度都等于平均的触发器驱动强度



（图 1）

- 将关键路径逻辑和非关键路径逻辑放在不同模块

  保证DC可以对关键路径模块实现速度优化，而对非关键路径模块实施面积优化。在同一模块DC无法实现不同的综合策略。

  将相关的组合逻辑放在同一模块

  有助于DC对其进行优化，因为DC通常不能越过模块的边界来优化逻辑。

5.1.3 Net and Register

- 一个reg变量只能在一个always语句中赋值。
- 向量有效位顺序的定义一般是从大数到小数。

  尽管定义有效位的顺序很自由，但如果采用毫无规则的定义势必会给作者和读代码的人带来困惑，如Data[-4： 0]，则LSB[0][-1][-2][-3][-4]MSB，或Data[0： 4]，则

LSB[4][3][2][1][0]MSB，这两种情况的定义都不太好，推荐Data[4：0]这种格式的定义。

- 对net和register类型的输出要做声明（在PORT中）。
  如果一个信号名没做声明，Verilog将假定它为一位宽的wire变量。
- 线网的多种类型。寄存器的类型。

5.1.4 Expressions

- 用括号来表示执行的优先级
  尽管操作符本身有优先顺序，但用括号来表示优先级对读者更清晰，更有意义。
  If ((alpha < beta) && (gamma >= delta)).... 比下面的表达更合意
  If (alpha < beta && gamma >= delta)...
- 用一个函数(function)来代替表达式的多次重复
  如果代码中发现多次使用一个特殊的表达式，那么就用一个函数来代替，这样在以后的版本升级时更便利，这种概念在做行为级的代码设计时同样使用，经常使用的一组描述可以写到一个任务(task)中。

5.1.5 IF 语句

- 向量比较时，比较的向量要相等。
  当比较向量时，verilog将对位数小的向量做 0 扩展以使它们的长度相匹配，它的自动扩展为隐式的。建议采用显示扩展，这个规律同样适用于向量同常量的比较。

```
Reg     Abc  [7:0];
Reg     Bca  [3:0];
.....
If (Abc = = {4'b0, Bca})begin
.......
If (Abc = = 8'b0) begin
```

- 每一个If 都应有一个else 和它相对应
  在做硬件设计时，常要求条件为真时执行一种动作而条件为假时执行另一动作，即使认为条件为假不可能发生。没有else可能会使综合出的逻辑和RTL级的逻辑不同。
  如果条件为假时不进行任何操作，则用一条空语句。

```
always @(Cond)
   begin
   if (Cond)
     DataOut <= DataIn;
End
// Else ： ;
```

  以上语句DataOut会综合成锁存器.

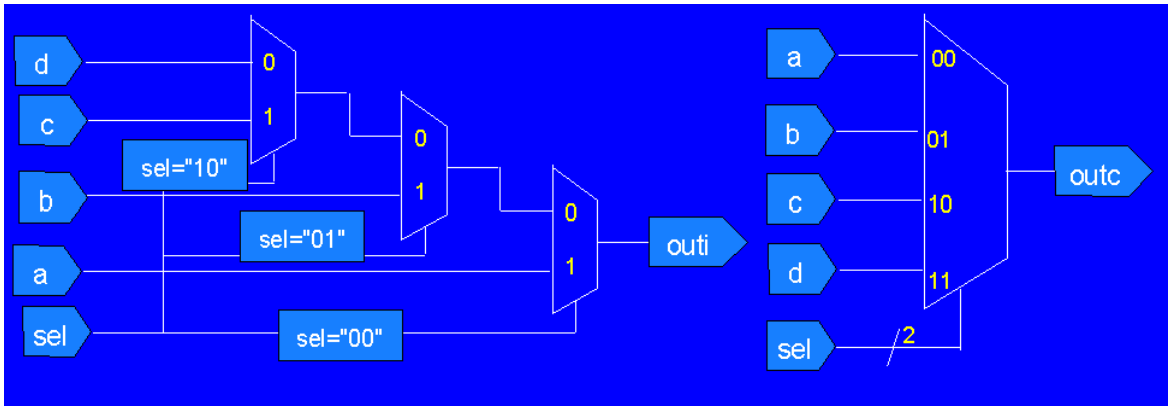
- 应注意If ..else if ...else if ...else 的优先级

- 如果变量在If-else 或case 语句中做非完全赋值，则应给变量一个缺省值，即：

```
V1 = 2'b00;
V2 = 2'b00;
V3 = 2'b00;
If (a = = b) begin
    V1 = 2'b01;          //V3 is not assigned
    V2 = 2'b10;
End
Else if (a = = c) begin
    V2 = 2'b10;          //V1 is not assigned
    V3 = 2'b11;
End
Else ： ;
```

5.1.6 case 语句

- case语句通常综合成一级多路复用器（图的右边部分），而if-then-else则综合成优先编码的串接的多个多路复用器，如图的左边部分。通常，使用case 语句要比if语句快，优先编码器的结构仅在信号的到达有先后时使用。条件赋值语句也能综合成多路复用器，而case 语句仿真要比条件赋值语句快。



- 所有的Case 应该有一个default case， 允许空语句

```
Default :  ;
```

5.1.7 Writing functions

- 在function的最后给function赋值

```
Function  CompareVectors;  // (Vector1, Vector2, Length)
Input     [199:0]     Vector1, Vector2;
Input     [31:0]      Length;
//local variables
Integer       i;
```

```
    Reg          Equal;
  Begin
      i = 0;
      Equal = 1;
      While ((i<Length) && Equal) begin
        If (Vector 2[i] !== 1'bx) begin
          If (Vector1[i] !== Vector2[i])
            Equal = 0;
          Else ;
        End
      i = i + 1;
    End
    CompareVectors =  Equal;
  End
  Endfunction      //compareVectors
```

- **函数中避免使用全局变量**

  否则容易引起HDL**行为级仿真和门级仿真的差异。如:**

```
function ByteCompare;
  input [15:0] Vector1;
  input [15:0] Vector2;
  input [7:0] Length;

  begin
    if (ByteSel)
      // compare the upper byte
    else
      // compare the lower byte
    。 。 。
  end
endfunction // ByteCompare
```

中使用了全局变量ByteSel，**可能无意在别处修改了，导致错误结果。最好直接在端口加以定义。** **注意，函数与任务的调用均为静态调用。**

5.1.8 Assignment

- Verilog **支持两种赋值：过程赋值**(procedural) **和连续赋值**(continuous)。**过程赋值用于过程代码**（initial, always, task or function)**中给**reg **和** integer**变量**、time\realtime、real**赋值，而连续赋值一般给**wire **变量赋值。**

- Always @(敏感表），敏感表要完整，如果不完整，将会引起仿真和综合结果不一致

```
always @(d or Clr)
    if (Clr)
      q = 1'b0;
    else if (e)
      q = d;
```

以上语句在行为级仿真时e的变化将不会使仿真器进入该进程,导致仿真结果错误

- Assign/deassign 仅用于仿真加速，仅对寄存器有用。
- Force/release 仅用于debug，对寄存器和线网均有用。
- 避免使用Disable
- 对任何reg赋值用非阻塞赋值代替阻塞赋值， reg 的非阻塞赋值要加单位延迟，但异步复位可加可不加。=与《=的区别。

```
Always @(posedge Clk or negedge Rst_)
Begin
 If (!Rst_)       // prioritize the "if conditions" in if statement
    Begin
     Rega <= 0;      //non_blocking assignment
     Regb <= 0;
    End
 Else if (Soft_rst_all)
   Begin
    Rega <= #u_dly      0;       //add unit delay
    Regb <= #u_dly      0;
    End
Else  if (Load_init)
    Begin
    Rega <= #u_dly      init_rega;
    Regb <= #u_dly      init_regb;
    End
Else
    Begin
    Rega <= #u_dly      Rega << 1;
    Regb <= #u_dly      St_1;
    End
End       // end Rega, Regb assignment.
```

5.1.9 Combinatorial Vs Sequential Logic

- 如果一个事件持续几个时钟周期，设计时就用时序逻辑代替组合逻辑

如： *Wire Ct_24_e4; //it ccarries info. Last over several clock cycles*

Assign Ct_24_e4 = (count8bit[7:0] >= 8'h24) & (count8bit[7:0] <= 8'he4);

那么这种设计将综合出两个8 比特的加法器，而且会产生毛刺，对于这样的电路，要采用时序设计，代码如下：

```
Reg      Ct_24_e4;
Always @(poseddge Clk or negedge Rst_)
Begin
If (!Rst_)
   Ct_24_e4  <= 1'b0;
Else if (count8bit[7:0] = = 8'he4)
    Ct_24_e4 <= #u_dly      1'b0;
Else if (count8bit[7:0] = = 8'h23)
     Ct_24_e4 <= #u_dly      1'b1;
Esle    ;
```

- 内部总线不要悬空。在default状态，要把它上拉或下拉。

```
Wire        OE_default;
Assign      OE_default = !(oe1 | oe2 | oe3);
Assign      bus[31:0] = oe1 ? Data1[31:0] :
                        oe2 ? Data2[31:0] :
                        oe3 ? Data3[31:0] :
                        oe_default ? 32'h0000_0000 :
                        32'hzzzz_zzzz;
```

5.1.10Macros

- 为了保持代码的可读性，常用 "`define" 做常数声明
- 把"`define"放在一个独立的文件中

  参数（parameter）必须在一个模块中定义，不要传替参数到模块（仿真测试向量例外），"`define"可以在任何地方定义，要把所有的"`define"定义在一个文件中，在编译原代码时首先要把这个文件读入。如果希望宏的作用域仅在一个模块中，就用参数来代替。

5.1.11Comments

- 对更新的内容更新要做注释
- 在语法块的结尾做标记

```
//style  1
  If (~OE_ && (state != PENDING)) begin
     ....
  End  // if enable = = ture and ready
```

```
//style  2  --- identical lables on begin and end
  If (~OE_ && (state != PENDING)) begin    //drive data
     ....
  End     //drive data
// Comment end<unit> with the name of the <unit>
  Function  Calcparity；//Data, ParityErr
    ....
  Endfunction     // Calcparity
```

- 每一个模块都应在模块开始处做模块级的注释（参考前面标准模块头）
- 在模块端口列表中出现的端口信号，都应做简要的功能描述

5.1.12FSM

- VerilogHDL状态机的状态分配
  VerilogHDL描述状态机时必须由parameter分配好状态,这与VHDL不同，VHDL状态机状态可以在综合时分配产生。
- 组合逻辑和时序逻辑分开用不同的进程
  组合逻辑包括状态译码和输出，时序逻辑则是状态寄存器的切换
- 必须包括对所有状态都处理，不能出现无法处理的状态，使状态机失控
- Mealy机的状态和输入有关,而Moore机的状态转换和输入无关。
  Mealy 状态机的例子如下:

```
    ...
 reg CurrentState, NextState, Out1;

Parameter S0=0,S1=1;

always @(posedge Clk or negedge Rst_)
// state vector flip-flops (sequential)
if (!Reset)
    CurrentState = S0;
else
    CurrentState <= #u_dly NextState;

always @(In1 or In2 or CurrentState)
// output and state vector decode (combinational)
case (CurrentState)
  S0: begin
    NextState <= #u_dly S1;
    Out1 <= #u_dly 1'b0;
```

```
        end
    S1: if (In1) begin
            NextState <= #u_dly S0;
            Out1 <= #u_dly In2;
        end
        else begin
            NextState <= #u_dly S1;
            Out1 <= #u_dly !In2;
        end
endcase
endmodule
```

5.2 代码编写中容易出现的问题

- 在for-loop中包括不变的表达式，浪费运算时间

```
for (i=0;i<4;i=i+1)
begin
    Sig1 = Sig2;
  DataOut[i] = DataIn[i];
end
```

for-loop中第一条语句始终不变,浪费运算时间.

- 资源共享问题

条件算子中不存在资源共享，如

```
z = (cond) ? (a + b) : (c + d);
```

必须使用两个加法器;

而等效的条件if-then-else语句则可以资源共享，如

```
if (Cond)
    z =  a + b;
else
    z =  c + d;
```

只要加法器的输入端复用,就可以实现加法器的共享,使用一个加法器实现

- 由于组合逻辑的位置不同而引起过多的触发器综合，如下面两个例子

```
module COUNT (AndBits, Clk, Rst);
Output      Andbits;
Input       Clk,
            Rst;
Reg         AndBits;
//internal reg
```

```
Reg          [2:0]        Count;

always @(posedge Clk) begin
begin
if (Rst)
   Count <= #u_dly 0;
else
   Count <=  #u_dly Count + 1;
End        //end if
   AndBits <=  #u_dly & Count;
End        //end always
endmodule
```

在进程里的变量都综合成触发器了,有4个;

```
module COUNT (AndBits, Clk, Rst);
Output        AndBits;
Input         Clk,
              Rst;
Reg           AndBits;
//internal reg
Reg           [2:0]      Count;

always @(posedge Clk) begin //synchronous
if (Rst)
   Count <= #u_dly 0;
else
   Count <= #u_dly Count + 1;
End      //end always

always @(Count) begin   //asynchronous
     AndBits = & Count;
End   //end always
Endmodule      //end  COUNT
```

组合逻辑单开,只有3个触发器.

- 谨慎使用异步逻辑

```
module COUNT (Z,  Enable, Clk, Rst);
Output        [2:0]        Z;
Input         Rst,
```

```
                    Enable,
                    Clk;

  reg [2:0] Z;

  always @(posedge Clk) begin
  if (Rst) begin
     Z <= #u_dly 1'b0;
  end
  else if (Enable == 1'b1) begin
        If (Z == 3'd7) begin
           Z <= #u_dly 1'b0;
        End
        else begin
           Z <= #u_dly Z + 1'b1;
        end
  End
  Else ;
  End      //end always
 Endmodule     //end COUNT
```

是同步逻辑,而下例则使用了组合逻辑作时钟,以及异步复位.实际的运用中要加以避免.

```
 module COUNT (Z,  Enable, Clk, Rst);
 Output          [2:0]          Z;
 Input           Rst,
                 Enable,
                   Clk;
   Reg           [2:0]          Z;
    //internal wire
   wire GATED_Clk = Clk & Enable;

   always @(posedge GATED_Clk or posedge Rst) begin
   if (Rst) begin
      Z <= #u_dly 1'b0;
   end
   else begin
       if (Z == 3'd7) begin
          Z <= #u_dly 1'b0;
```

```
        end
        else begin
            Z <= #u_dly Z + 1'b1;
        end
    End       //end if
    End       //end always

    Endmodule       //end module
```

- 对组合逻辑的描述有多种方式，其综合结果是等效的

  c = a &b;

  等效于

  c[3:0] = a[3:0] & b[3:0];

  等效于

  c[3] = a[3] & b[3];

  c[2] = a[2] & b[2];

  c[1] = a[1] & b[1];

  c[0] = a[0] & b[0];

  等效于

  for (i=0; i<=3; i = i + 1)

  c[i] = a[i] & b[i];

  可以选择简洁的写法.

- 考虑综合的执行时间

  通常会推荐将模块划分得越小越好，事实上要从实际的设计目标、面积和时序要求出发。好的时序规划和合适的约束条件要比电路的大小对综合时间的影响要大。要依照设计的目标来划分模块，对该模块综合约束的scripts也可以集中在该特性上。要选择合适的约束条件，过分的约束将导致漫长的综合时间。最好在设计阶段就做好时序规划，通过综合的约束scripts来满足时序规划。这样就能获得既满足性能的结果，又使得综合时间最省。从代码设计讲，500~5000行的长度是合适的。

- 避免点到点的例外

  所谓点到点例外（Point-to-point exception），就是从一个寄存器的输出到另一个寄存器的输入的路径不能在一个周期内完成。多周期路径就是其典型情况。多周期路径比较麻烦，在静态时序分析中要标注为例外，这样可能会因为人为因素将其他路径错误地标注为例外，从而对该路径没有分析，造成隐患。避免使用多周期路径，如果确实要用，应将它放在单独一个模块，并且在代码中加以注释。

- 避免伪路径(False path)

伪路径是那些静态时序分析（STA）认为是时序失败，而设计者认为是正确的路径。通常会人为忽略这些warning，但如果数量较多时，就可能将其他真正的问题错过了。

- 避免使用Latch

  使用Latch必须有所记录，可以用All_registers -level_sensitive来报告设计中用到的Latch。不希望使用Latch时，应该对所有输入情况都对输出赋值，或者将条件赋值语句写全，如在if语句最后加一个else，case语句加defaults。

  当你必须使用Latch时，为了提高可测性，需要加入测试逻辑。

  不完整的if和case语句导致不必要的latch的产生，下面的语句中，DataOut会被综合成锁存器。如果不希望在电路中使用锁存器，它就是错误。

  ```
  always @(Cond)
  begin
      if (Cond)
          DataOut <= DataIn；
  end
  ```

- 避免使用门控时钟

  使用门控时钟(Gated clock)不利于移植，可能引起毛刺，带来时序问题，同时对扫描链的形成带来问题。门控钟在低功耗设计中要用到，但通常不要在模块级代码中使用。可以借助于Power compiler来生成，或者在顶层产生。

- 避免使用内部产生的时钟

  在设计中最好使用同步设计。如果要使用内部时钟，可以考虑使用多个时钟。因为使用内部时钟的电路要加到扫描链中比较麻烦，降低了可测性，也不利于使用约束条件来综合。

- 避免使用内部复位信号

  模块中所有的寄存器最好同时复位。如果要使用内部复位，最好将其相关逻辑放在单独的模块中，这样可以提高可阅读性。

- 如果确实要使用内部时钟，门控时钟，或内部的复位信号，将它们放在顶层

  将这些信号的产生放在顶层的一个独立模块，这样所有的子模块分别使用单一的时钟和复位信号。一般情况下内部门控时钟可以用同步置数替代。例如:

  ```
  module COUNT (Reset， Enable， Clk， Qout)；
  input Reset， Enable， Clk；
  output [2:0] Qout；
  reg [2:0] Qout；
  ```

  ```
  module COUNT (Reset， Enable，
                Clk， Qout)；
  input Reset， Enable， Clk；
  output [2:0] Qout；
  reg [2:0] Qout；
  ```

```
wire GATED_Clk = Clk & Enable;

always @(posedge GATED_Clk or posedge Reset)
begin
  if (Reset) begin

    Qout = 1'b0;

  end
  else begin
    if (Qout == 3'd7) begin
      Qout= 1'b0;

    end
  else begin
     Qout = Qout + 1'b1;
  end
 end
end
endmodule
```

```
always @(posedge Clk) begin

 if (Reset) begin
      Qout = 1'b0;
 end
else if (Enable == 1'b1) begin

    if (Qout == 3'd7) begin

      Qout = 1'b0;
    end
  else begin
      Qout = Qout + 1'b1;

    end
 end
end
endmodule
```

6 附录

6.1 Module 编写示例

```
/* ==========================================*\
            Filename  :
            Author    :
           Description  :
            Called by  :
       Revision History   : mm/dd/yy
                          Revision 1.0
             Email  :  M@sz.huawei.com.cn
           Company  :  Huawei Technology .Inc
      Copyright(c) 1999, Huawei Technology Inc, All right reserved
```

```
\* ==========================================*/
  Module  module_name(
                          Output_ports,          //comment ; port description
                          Input_ports,           //comment ; port description
                          Io_ports,              //comment ; port descripttion
                          Clk_port,              //comment ; port description
                          Rst_port               //comment ; port description
                          );
 //port declarations
   Output        [31:;0]       Dataout;
   Input         [31:0]        Datain;
   Inout                       Bi_dir_signal;
  Input                        input1,
                               Input2;

 //interrnal wire/reg declarations
 Wire          [31:0]         internal_data;
Reg                           output_enable;

//module instantiations , Self-build module
    Module_name1     Uinstance_name1(...);
   Module_name2     Uinstance_name2(...);
 // TSC4000 cell
  DTC12 V1 (.Clk(Clk), .CLRZ(Clr), .D(Data), .Q(Qout));

 //continuous  assignment
  Assign        Data_out = out_enable ?  Internal_data :  32' hz;

//always block
 Always @(input2)
 Begin
 ...
 End

//function and task definitions
 Functiom   [function_type]  function_name;
 Declarations_of_inputs;
```

```
[declarations_of_local_variables];
Begin
 Behavirol_statement;
 Function_name = function_express;
End
 Endfunction      //end function_name
Endmodule       //end module_name
```

6.2 testbench编写示例

下面是一个格雷码的测试模块，

```
module TB_GRAY;
reg              Clock;
reg              Reset;

wire    [7:0]    Qout;
integer fout;              //输出文件指针
parameter CYC = 20;
GRAY DUT(.Clock(Clock),.Reset(Reset),.Qout(Qout));

initial
begin
  Clock = 1'b0;
  Reset =1'b1;
  #(5*CYC) Reset = 1'b0;
  #(5*CYC) Reset = 1'b1;
  #(5000*CYC)
      $fclose(fout);
      $finish;
end

  initial
  begin
      $shm_open("GRAY.shm");
      $shm_probe("AS");
      fout=$fopen("gray.dat");
  end
```

```
always #CYC Clock = ~ Clock;

//输出数据到文件gray.dat
always @(posedge Clock)
begin
   $fwrite(fout,"%d %b\n",Qout,Qout);
end
endmodule
```

a）、在testbench中避免使用绝对的时间,如#20,#15或#(CYC+15)等,应该在文件前面使用parameter定义一些常量,使得时间的定义象#(CYC+OFF0)的形式,便于修改;

b）、观测结果可以输出到波形文件GRAY.shm，或数据文件gray.dat。生成波形文件可以用simwave观测结果，比较直观；而生成数据文件则既可以快速定位，也可以通过编写的小程序工具对它进行进一步的处理;

c）、对大的设计的顶层仿真，一般不要对所有信号跟踪，波形文件会很大，仿真时间延长，可以有选择的观测一些信号;

第三章 可编程ASIC设计方法简介

关键词：可编程逻辑电路，ASIC，FPGA。

摘　要：本文根据当今可编程逻辑电路发展趋势，论述了它对设计方法的影响，并着重介绍了一种不同以往的设计思想。

1 引言

随着集成电路技术的发展，集成电路的线宽越来越窄，设计规模越来越大，硬件设计正逐步走向SOC。以Xilinx 公司最新FPGA器件Virtex-E系列为例，其采用的工艺是当今非常先进的0.18微米工艺，6层金属布线，电路最大规模可达300万门，时钟频率超过300MHz。这就使可编程器件能够获得以前只能在ASIC芯片上实现的某些特性，如SOC等，具有ASIC特性。另外，随着设计规模的扩大，可编程器件的设计方法正逐步向ASIC靠拢。因此，从这种意义是说，大规模可编程逻辑电路是某种特殊的ASIC，即可编程ASIC。

当设计规模超过一万门时，原来采用原理图输入的设计方法就显得过于繁琐，一些设计手段已明显落后市场变化。因此，当我们面对大规模可编程ASIC设计时，应当从新认识过去我们建立起来的设计方法和设计手段。

从目前公司的发展来看，今后大规模可编程ASIC设计将占据一个非常重要的地位，可以说，华为硬件工程师对可编程ASIC设计方法的掌握程度直接关系到华为的生死存亡。

公司迅速发展依赖于ASIC开发设计水平，当然也依赖于可编程器件开发设计水平。从目前公司的发展及国外主要厂商的发展实践来看，ASIC部将逐渐成为硬件开发的最主要部门，因此今后的发展趋势将是系统设计人员与ASIC设计人员逐步融合，目前即将成立的大规模逻辑电路设计研究部是这一发展趋势的必然结果，同时也是公司研发紧跟世界潮流的可靠保证。

成立大规模逻辑电路设计研究部，就是为了让所有的硬件工程师都懂得基本的ASIC设计方法，掌握可编程ASIC设计方法是系统设计人员与ASIC设计人员走向融合的第一步。

2 芯片设计发展趋势

2.1 芯片设计类型

随着设计手段和方法的改进，芯片设计成为硬件设计中最关键的部分。一般而言，根据速度与面积的关系，芯片设计可以分为三类：

第一类，面积优先。

过去，由于受工艺限制，单位面积上可集成的电路规模较小，在设计中往往考虑的是如何有效利用芯片面积，降低设计规模，因此，传统的设计方法是以面积为考虑的第一要素。此外，由于当时的EDA辅助设计工具发展不够完善，并且价格又相对十分昂贵，因此多以原理图输入为设计的主要手段，比较复杂的软件仿真无法实现，并且在多数情况下也无此必要。

今天，由于设计要求越来越高，面积优先的设计正逐步减少，但是，在许多设计中，面积仍是考虑的重要因素，甚至是决定性的。

第二类，速度优先。

这里的速度有两方面含义：芯片工作速度与开发速度。

随着集成电路的发展，工艺尺寸越来越小，设计规模越来越大，设计复杂程度越来越高。与此同时，市场竞争越来越激烈，产品开发周期要求越来越短。谁能在最短的时间内开发出符合市场需要的相对稳定可靠的产品，谁就会在竞争中占据有利定位；反之，谁就有可能被淘汰出局。

残酷的市场竞争要求我们的设计：一方面，设计的芯片速度必须越来越快，以增加带宽，满足市场需要；另一方面，设计必须在最短的时间内完成，否则将面临死亡。因此，在今后的大多数设计中，速度将起决定性的作用，而面积则处于从属地位，为了速度，有原则的牺牲面积是可以考虑的。

第三类，面积＋速度。

事实上，任何一个设计都必须同时考虑面积和速度，只是各有侧重点而已。之所以这样做，是为了考虑综合成本：

（1）为了降低生产成本，希望我们的设计尽可能最优，即速度最快，面积最小。但这会延长设计周期，降低“机会窗”利润。

（2）为了尽快让产品上市，希望我们的设计能在最短的时间内开发成功，迅速抢占市场。但这有可能会牺牲产品的部分性能，为了迎合顾客需要而不断更新换代，从而增加生产和开发成本。

注意：

上述提到的三类芯片设计实际只是为了说明一个问题：谁是设计中考虑的第一要素，谁是第二要素，并根据他们的关系采取不同的开发策略。

2.2 芯片设计发展趋势

实际上，随着技术水平的发展，芯片设计正逐渐向速度优先靠拢，这是因为：

1. 设计容量越来越大，越来越复杂，正向SOC（System On a Chip）方向发展。

2. 由于工艺的改进，芯片规模（面积）已不是制约设计主要因素，芯片工作频率（速度）是关键。如何提高芯片速度是设计者首要考虑的因素。

3. 市场竞争日趋激烈，要求设计能在最短的时间内完成，“时间就是生命”，必须采用新的设计方法和手段，及相应规范化的措施来尽快缩短开发周期。

在今后的许多重要设计中，速度是关键，所有活动都是以“速度”为核心而开展，没有“速度”就没有一切，在速度的前提下考虑面积。并且，随着工艺的发展，可编程器件特性在向ASIC发展，与此相适应，可编程设计方法也在向ASIC靠拢，即可编程ASIC设计。

2.3 根据市场需求和产品发展策略确定芯片开发策略

事实上，芯片与产品（系统）的关系不外乎：

1. 系统决定芯片

这是由于：市场无法预测，产品主要是根据客户需要或其它因素才确定下来。

2. 芯片决定系统

当我们的市场把握能力很高时，能准确地预测市场走向，并依此决定我们的产品开发策略。此时，我们的芯片开发可以并且应当走在系统前面（类似芯片预研）。

目前，我司大多数的芯片开发属于第一种情况。针对这类芯片开发，功能复杂的可以分几个阶段进行，每个阶段采取不同的开发策略。

第一阶段，抢占市场，快速推出相对稳定可靠的产品（速度优先）

这种抢占市场的产品开发策略要求我们的芯片必须在最短的时间内完成开发。针对这种情况，芯片开发可采取如下对策：

1. 第一阶段的开发（开局版本）以满足最基本需求为主，适当降低开发难度与工作量，保证产品如期上市和相对稳定性，为将来的版本升级争取时间。当然，如果能一次开发成功则更好。

2. 芯片功能测试（软件测试与硬件测试相结合）尽可能完善。

3. 器件的选型必须留有足够的余量，保证将来版本升级时不用更改单板。

4. 芯片的基本功能必须正确，并采取一些通用的可靠性设计技术，保证芯片出现错误后系统能及时检测到并能自动对其进行恢复，将可能的错误危害降低到最小程度。

第二阶段，芯片性能改进和功能扩展（速度＋面积）

当产品上市之后，为了巩固阵地和进一步拓展市场：一方面，必须迅速提高芯片的可靠性，解决遗留问题；另一方面，由于市场变化很快，很有可能发现产品推出后，才发现要增加新的功能，或者需要在某些方面进行修改，以适应变化了的市场需要。因此：

1. 跟踪网上器件运行状况，有针对性地加大硬件测试力度，查出芯片深层问题。

2. 针对各种故障现象，专门编写仿真测试代码，加快故障解决进度。

3. 增加各种故障检测和保护电路，提高芯片的可靠性。

4. 根据需要，补充新功能，去掉不必要的电路，并进行适当的代码优化，确保芯片面积不超过“额定容量”。

在该阶段，应综合考虑速度和面积。

第三阶段，降低成本（面积优先）

当产品已经获得规模应用时，成本因素是考虑的重点：

1. 考虑转ASIC，同时进行电路优化，保证ASIC芯片面积最小。

2. 当需要再增加芯片功能时，可以考虑优化电路，确保芯片面积不超过“额定容量”。

注意：上面提到的芯片开发三个阶段，并不是绝对的。它主要是用来说明：在芯片开发的不同时期，不同的设计，针对不同的奋斗目标，我们的开发策略或重心应有所不同。另外，完整的开发阶段应当包含芯片预研阶段，这里不做重点讨论。

3 可编程ASIC设计

3.1 设计工程师基本素质要求

随着设计规模的增加，传统的设计方法已跟不上时代的发展需要。为了在最短的时间内将产品推向市场，要求我们的设计要快速可靠地完成。为此，我们的可编程ASIC设计应当是：

1. 能满足客户基本需要。

2. 是稳定可靠的。

3. 必须具备良好的可维护性和可继承性，容易改进和升级。

4. 具有很好的可移植性，便于技术经验共享，加快设计进度。

5. 设计规范，便于工程师进行技术交流和设计审查，保证设计质量。

这就要求我们的设计工程师：

1. 扎实的专业知识。

2. 同步电路设计方法及异步电路处理技巧。

3. 掌握硬件描述语言，如VHDL或Verilog语言。

4. 熟悉各种先进的EDA设计工具，如综合工具、仿真工具、静态时序分析工具及相关厂家的设计流程等。

3.2 基本设计流程

可编程ASIC设计最基本的流程是：

设计目标分析 → 功能模块划分 → 确定关键电路时序和模块间接口时序 → 具体电路设计 → 设计验证

3.2.1 设计目标分析

设计目标分析主要目的是确定芯片应当完成哪些功能。对于大规模的设计，一般设计比较复杂，因此应当根据产品的发展计划，将设计目标分若干步骤来实现：

1. 基本目标：对应开局版本，确定应完成的基本功能。
2. 阶段目标：确定分多少阶段来逐步实现这些目标。
3. 可能目标：分析未来可能要增加的功能。
4. ASIC计划：确定是否转ASIC，什么时候转ASIC。

在确定目标时，必须充分考虑各目标的实现难度和可行性。在基本目标确定之后，原则上不允许有改动，否则会对设计进度、士气造成恶劣影响。

在各个目标的实现过程中，要注意各版本的兼容性（包括对单板软件的影响），这就要求我们的器件在选型时必须留有足够的设计余量，保证升级时单板不会因此而修改。

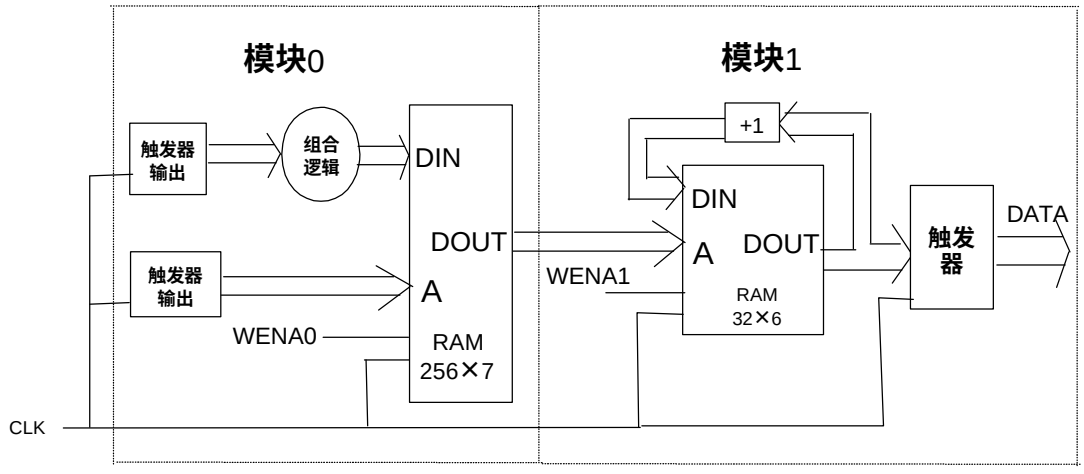
3.2.2 功能模块划分

主要目的是让设计层次分明，条理清晰。另外，在确定功能模块划分过程中，能使设计者在总体上考虑芯片的各个问题，发现一些比较深层次的问题，这对设计能否实现是非常重要的。

3.2.3 确定关键电路时序和模块间接口时序（总体方案）

事实上，在功能模块划分过程中，就必须考虑时序方面的问题。有时在确定设计目标时，就得考虑关键电路时序。

当前我们的设计过程中存在这样一种现象：当我们把一个设计进行功能分解之后，是直接交各项目组负责完成，然后再进行系统联调，看看会出哪些问题，再去更改设计。这是在凑电路，而不是在设计电路。设计电路，尤其是数字电路，最关键的一环是：设计各模块间的接口时序，确定关键电路的时序。这个工作必须在具体电路设计之前确定下来。凡是设计过电路的人都经历过：我们在系统联调时经常发现彼此之间的配合出了问题；当我们的设计不能满足时序要求时，都会在那挖空心思地想办法改进电路，甚至更改整个设计。



我们来看一个实际例子。上图是某个设计原来准备采用的电路，该设计采用Xilinx的FPGA器件4062xla来实现，工作频率是32.768MHz（即图中CLK频率）。设计原打算在每隔60ns输出一个数据，即DATA。然而，我们在设计之前，考虑到256x7的同步RAM延时可能比较大，如果在加上其后的同步RAM延时的话，估计在60ns之内很难完成。该部分电路是整个设计中的一个关键路径，因此，我们在进行具体设计之前，先对这种电路结构进行了验证，事实证明我们的担心是对的。正确的做法是：采用流水线方法，在256x7的RAM之后再加一个触发器，每个RAM都按60ns的速度读取数据，整个流程滞后60ns输出DATA。其它相关信号（在其它模块中）也随之滞后60ns输出。

因此，我们在做总体方案时，应该深入到模块间的时序划分，关键电路的时序确定，并以此指导各项目组、项目组各成员进行设计。这项工作的实质就是确定各模块设计需求。在设计之前，若选择的设计策略出现问题，则会极大增加设计实现的难度，影响产品开发的进程。

“时序是事先设计出来的，而不是事后测出来的”。

3.2.4 具体电路设计（详细设计文档）

当各个模块划分完毕，且模块间接口时序基本清楚之后，就可以交给项目组各成员进行具体电路开发。

在进行模块设计时，应先画出每个模块的原理结构，而后画出其工作原理时序图，在工作原理时序图的指导下，进行具体电路设计。这种“先时序后电路”设计方法的好出是：

1. 思路清晰，考虑周到，不容易出错；
2. 电路即使出错误，也很容易查出问题原因所在；
3. 在越复杂的电路，原理时序图越容易让人理解，便于项目组成员之间进行交流和互相查错。

事实上，“先时序后电路”的设计方法也体现了“时序是设计出来的”思想。

在进行大规模逻辑电路设计时，显然采用VHDL或Verilog语言进行设计是目前最好地选择，熟练掌握其中之一是未来硬件设计工程师的必备知识。为了能与将来ASIC设计接轨，最好能掌握Verilog语言。另外，熟练使用各种先进的EDA工具，如综合工具、仿真工具、静态时序分析工具等等，也是进行正常芯片设计所不可缺少的。

在进行具体电路设计过程中，应当注意哪些问题，可以参见“可编程器件设计规范”。

3.2.5 设计验证（仿真测试方案）

设计验证手段有软件和硬件之分。软件验证包括：RTL级功能仿真，静态时序分析和时序仿真。硬件验证主要是指单板测试或系统测试。

一、软件验证

1. RTL级功能仿真（功能验证）

在进行RTL级功能仿真时，在可能的情况下，最好是对每一个子模块进行仿真，确保90％以上的错误在设计前期就得到解决，否则会增加设计后期的困难，如查错。至少，需对设计的关键部分进行充分的RTL级验证。

当设计代码全部完成之后，可以进行整个设计的RTL级仿真。在进行芯片RTL级仿真之前，应与相关人员进行仔细讨论，制定完善的仿真测试方案，尽可能覆盖所有情况。不过，根据我们

的经验，要想100％的覆盖，完全靠RTL级仿真是很困难的，必须借助其它手段才有可能。但是，RTL级仿真是所有验证环节中最重要的一环，有时是决定设计成败的关键。在进行RTL级功能仿真时，我们的目标是解决所有的功能方面的问题。因此，RTL级功能仿真可称之为功能验证。

2. 静态时序分析

静态时序分析有两个作用：分析设计是否满足时序要求；分析设计容限。

传统的动态时序分析（指带延时参数的时序仿真）受条件限制，无法完成100％的时序电路分析。而静态时序分析所采用的方法与动态时序分析不同，它根据电路的拓扑结构和工艺参数，来分析电路的时序关系。在进行静态时序分析时，分析工具需要设计者给出相应的“合法路径”，由分析工具自动地完成时序分析，如路径延时等信息。

在进行静态时序分析时，要求设计者针对所有路径提出具体时序要求，如I/O管脚之间时序要求，内部“合法路径”时序要求等，这样就能保证时序分析的全面性，避免给设计者以错误信息。

在Xilinx FPGA设计过程中，ucf文件中的时序约束就是设计本身的时序要求。在进行布局布线时，工具本身针对该时序要求不断进行静态时序分析，并进行有关调整，使设计尽量满足要求。

在设计中要切记：通过静态时序分析进行的时序调整只是一种微调，对于大的时序调整，必须从设计本身来保证。

3. 时序仿真（在ASIC设计领域，有前仿真和后仿真之分）

时序仿真既有动态时序分析功能，又有功能验证之功能。由于时序仿真带有延时信息，因此软件在仿真，其运算量比RTL级仿真时要多得多；而且，若设计改动较多，每次功能验证都通过时序仿真来完成的话，极为费时，严重影响设计进度。因此，设计的功能验证应主要由RTL级仿真来保证，时序验证主要由静态时序分析来保证。

然而，在设计中我们无法保证：

（1）功能验证通过的设计代码无法保证按照设计原样真实地体现在最终的设计版图上；

（2）静态时序分析时，设计者有可能忽略某种情况，但事实上这是错误的。

（3）某些功能无法或很难通过RTL级仿真和静态时序分析来保证其功能的正确性，如一些异步处理电路的容限问题等。

因此，一定程度的时序仿真是有必要的。但是，时序仿真的次数越少越好，甚至可以为零（目前不推荐）。

二、硬件验证（单板测试方案）

完善的硬件验证是保证设计高可靠性的重要手段。为了进行完善的硬件验证，需要所有的单板人员，包括单板软件人员，一起来制定测试方案。同时，为了方便硬件测试，有必要在原设计中增加必要的辅助测试电路：针对可能出现问题的部分，根据错误类型设置不同的错误判断电路，有利于在硬件测试时方便地定位故障。

无论是从硬件测试角度，还是从可靠性设计角度出发，都要求我们的设计尽可能的“透明化”：希望在单板硬件出现问题时，能很迅速地定位是否是芯片自身问题；若是芯片自身问题，能很迅速地知道是哪部分出现问题，从而采取有针对的处理措施。

有关内容可参见“ASIC可靠性设计”。

3.3 设计的可靠性

一般而言，同步电路比异步电路可靠，并且容易实现，具体内容可参见“可编程器件设计规范”。

另外，有关如何从整个设计角度考虑可靠性问题的内容可参见“ASIC可靠性设计”和“ASIC设计如何考虑可靠性”。

3.4 设计的规范性

设计应当尽可能规范，目的是：

1. 增加代码可读性，方便相关人员进行设计交流和代码检查。

2. 增加代码的可移植性（类似软IP），方便经验技术交流，加快设计进度。

有关规范性的内容参见“VHDL语言设计规范”和“Verilog语言设计规范”。

4 附录

4.1 ASIC可靠性设计

产品可靠性是产品质量的重要指标，是产品维护成本最重要的驱动因素，控制了产品的可靠性指标也就在源头上基本控制了维护成本。 根据日本电子行业的统计，产品不可靠的原因中，设计占80%，元器件占15%，制造工艺占5%，持续不断地对产品进行优化、测试，制定和推行产品的可靠性增长计划，对降低产品维护成本非常显著。

在我司发展的历程中，由于在产品的开发设计中没有有意识地强化可靠性设计，对产品的使用环境条件论证不充分，导致产品投放市场后故障率较高，不仅维护成本大增，也影响了公司的声誉。从我们基础部ASIC设计角度来看，由于我们设计的芯片在某些方面的不稳定性和不可靠性，一方面影响了公司整个产品的开发进度，导致不能按时交货，影响公司声誉；另一方面影响了我们基础部的形象，导致其它部门对我们设计出的芯片持怀疑态度，不敢用我们设计出的芯片。如果我们任这种状况发展下去，那么，在不久的将来，可以说我们ASIC部门没必要继续存在。

开发产品犹如建一座大厦，芯片犹如大厦的基石，基石没打好，即便大厦建成，过不了多久也会轰然倒塌。“基石”稍微有一点换动，住在“大厦”里的人（用户）感觉就是一场地震。在128模块开发过程中，尤其是在系统联调过程中，我们项目组提供的“基石”（SD530）曾给128这座大厦带来若干次“地震”，而且震感一次比一次强。造成这些“地震”的原因有许多，但最重要的是我们当初在设计时对芯片外部工作环境了解不够，对芯片在系统中的重要性认识不足，在芯片可靠性方面考虑很少。根据这几个月的调试，我觉得应在以下几个方面认真考虑：

一、时钟

1、主时钟“抖动”处理。在芯片实际应用中，主时钟不可能完全一致，应当有一个正常波动范围。当芯片的主时钟和系统提供的同步信号（例如8K同步信号）是来自不同的源时，也就是说

这两者之间没有固定地相位关系，则“抖动”处理尤为重要。这种情况下，时钟同步调整点附近不应该安排任何操作。

2、时钟“丢失”处理。正常情况下，这种现象不应当发生。然而，当系统发生意外时，有可能导致在一段时间内时钟没了（例如8K同步信号），不久又恢复正常，此时我们的芯片应当随之恢复正常。

二、输入控制信号

1、去毛刺处理。这一点大家基本都知道，也比较有经验，这里就不在罗嗦。

2、误码处理。所谓误码是指输入信号不是毛刺，但出现了正常情况下不可能出现的信号。在这种情况下，要注意芯片不能陷入“死循环”，或者陷入不可恢复的错误状态。

三、芯片“透明性”

所谓“透明性”，是指芯片的一些“活动”应尽可能让系统知道，例如芯片收到多少帧，发现多少错误帧，芯片由于本身队列溢出丢弃多少帧，发出多少帧等。

在系统联调时，什么问题都有可能出现，如果芯片的“透明度”不高，对系统来讲是一个“黑箱”，则很难进行故障定位，会严重推迟开发进度；在系统实际应用过程中，芯片越“透明”，系统越容易监控芯片，一旦发生故障，可在故障起始阶段就进行处理，不致故障被放大，提高系统的可靠性。

因此，增加芯片透明度，一可方便故障定位，加快调试进度；二可起到一定的防患作用，提高系统可靠性。

1、接口处理透明。在芯片接口处，芯片应提供各种各样的统计表，供系统分析判断。 例如，在时隙交换网络里，输入UHW上收到多少有效帧数据、错误帧数据，输出DHW发送多少帧数据、溢出丢弃多少帧数据等。

2、存储管理处理透明，或者说芯片核心处理透明。一般而言，芯片的存储管理部分是芯片的核心，也是最容易出问题的环节。正常情况下，芯片的存储管理基本不会出错。然而，当系统出现一些意想不到的情况时，有可能导致存储管理出错，严重时会导致整个芯片工作不正常，进而导致整个系统崩溃。因此，芯片的核心处理部分，应尽可能向系统开放，很多问题是通过它才得以定位。

四、异常复位，即芯片可恢复性

“百密难免一疏”，有很多情况是我们设计者在设计之初所无法想象的。但是，当发生异常现象时，芯片应保证能极时检查到，并将异常信息反映给系统，由系统决定如何处理。

如果发生重大异常情况，导致芯片工作极不正常，并有可能引发整个系统崩溃，此时，芯片应提供异常复位功能，由系统在“错误放大”之前，对芯片强行进行软复位。

异常复位是一种破坏性恢复手段，是为了处理突发性异常现象。出现问题并不十分可怕，可怕的是出现问题后无法恢复正常。

“可恢复性”是芯片可靠性最重要的一个指标。

五、芯片可靠性测试

可靠性测试分两方面，一是通过软件进行，也就是写测试代码，如Verilog格式测试代码；二是通过FPGA进行。

1、测试代码

在我们以往的测试代码中，过多地考虑的是基本功能测试，而在系统不稳定因素方面考虑较少。为了使测试代码更加完善，尽可能模拟实际运行情况，我觉得应把测试当作一个设计来对待，在总体方案形成的同时，测试方案也随之产生，并经大家评审，看是否有遗漏之处。在测试方案产生之前，开发人员须主动向系统人员了解与芯片接口有关的信号质量，包括正常情况和异常情况，以及异常时对芯片有哪些具体要求。最终，总体方案讨论产生两个需求：芯片基本功能需求和可靠性需求。其中可靠性需求包括上述提到的四个方面。当然，实际情况可能不止这些。

2、FPGA调试

FPGA调试应尽可能模拟实际应用环境，如果这一点不能保证的话，芯片的可靠性会大打折扣，同时芯片的基本功能是否完全正确也不能完全确定。当然，这需要通过部门之间的协调才能完成。同时，我们在设计之初，尽可能考虑完善些。

另外，在进行FPGA调试时，应对系统测试人员进行必要的指导，与他们一起制定一个详细的测试方案，保证测试的全面性。

总之，在进行可靠性设计时，要注意以下几个原则：

1、芯片抗干扰、抗误码。

2、芯片能自我检查故障。

3、芯片处理尽可能“透明”。

4、芯片出故障可恢复。

5、测试手段必须全面，哪怕是一个很简单的功能。越简单越容易出错。

芯片的可靠性是一项长期而艰巨的任务，须经过不断地优化、测试，甚至“几代人”的努力才能完成。只要我们从现在做起，不断积累，不断完善，就一定能把我们的芯片做得更稳定、更可靠。

4.2 ASIC设计如何考虑可靠性

随着我司产品的发展，ASIC芯片在产品中的份量越来越重，如128中的SD530等芯片，其表现的好坏直接影响到整个产品的性能；同时，ASIC芯片自身也在向高速、大容量发展，复杂程度越来越高。这就造成：一方面，产品对ASIC芯片的可靠性要求越来越高；另一方面，芯片复杂程度的提高势必影响芯片的可靠性程度，若处理不当，将造成非常严重的后果。而目前我部门在可靠性设计方面的历史积累还很少，在可靠性设计方面还有一段很长路要走。这里，我们以SD530为例，阐述如何进行可靠性设计，希望能起到抛砖引玉的作用，共同为我部门ASIC可靠性设计水平的提高做出贡献。

在ASIC芯片可靠性设计中，应考虑如下因素对芯片工作的影响：

一、外部因素

考虑的外部因素有：

1、外界环境的变化，如温度、湿度等；

2、单板信号质量引起的毛刺；

3、外界干扰造成的误码、毛刺等；

4、上电之后一段时间内，芯片各输入信号不稳，尤其是控制信号，可能不稳定。

二、内部因素

考虑的内部因素有：

1、实际工作频率与芯片极限工作频率之间的裕量；

2、异步电路；

3、复杂的电路，如状态处理电路等；

4、输出信号有毛刺。

另外，为了提高芯片的可靠性，还需增加一些必要的辅助电路：

1、DFT电路；

2、故障检测定位电路；

3、故障纠错保护电路；

4、异常复位电路。

在SD530中，采用了“三级防火墙”体系，最大限度地保证SD530的可靠性。

第一级：针对接口输入信号，尤其是各种控制信号，进行必要的去毛刺处理，同时针对可能出现的乱码（尤其是上电一瞬间），进行抗乱码处理；输出信号保证无毛刺，并尽量采用触发器的输出作为端口输出信号。

SD530与其它硬件的接口信号包括：CPU接口信号，UHW输入信号，总线控制信号。在进行SD530设计时，对CPU接口读写信号、UHW输入信号、总线控制信号均进行了去毛刺处理。然而，在设计开始阶段，并没有进行乱码处理，尤其是总线控制信号。由于总线控制信号会影响SD530的数据存储管理，结果发现在上电开始阶段SD530经常工作不正常。引入乱码处理之后，问题得以解决。

引入乱码处理的实质是：保证设计能正确处理可能出现的每一个状态，尤其是偶而出现的错误状态，保证芯片在“偶尔”过后，能正常运行而不至于“死掉”。

因此，在进行设计时，对一些重要的控制信号，应按“随机信号”来对待。

第二级：对一些复杂电路和关键电路，增加自我纠错保护电路，防止可能出现的意外或者芯片自身的不完善造成对芯片的严重影响。

在SD530的接收芯片，其UHW处理电路中的状态处理部分十分复杂，在绝大部分情况下，无任何问题。但是，当UHW出现误码时，在某一“特定情况”下其状态处理结果出错，导致芯片的核心部分存储管理电路出问题。然而，虽经多次艰苦努力，仍未找到是何种“特定情况”。最后，我们决定增加状态纠错保护电路，来检查状态处理是否正确，当发现处理不对时，采用特殊手段，保证芯片的存储管理部分不出乱子。事实证明，该方法非常有效。

第三级：提供芯片软件复位电路，芯片自身检查可能出现的暂时无法预防的最严重故障，由软件完成芯片的复位。

在SD530设计过程中，经常发现芯片的存储管理部分出现问题，经过多次努力之后问题似乎解决。但是，我们担心可能还有深层次的问题未能发现，为此决定：在SD530中增加存储管理错误检查电路和提供异常复位电路，当发现存储管理出现问题时，有软件启动复位电路，使芯片迅速恢复正常。

事实证明，这是一个非常明智的决定，在随后的测试中，我们发现已经出去开局的版本仍然存在问题，但由于异常复位电路的存在，该问题的影响程度大大降低。

有了复位电路，我们的芯片还有什么好怕的？

SD530除了富有特色的防火墙体系外，另一大特点是其完善的故障检查定位电路，极大地提高了系统快速定位故障的能力，能使系统在电路出现故障之初就能进行相应的保护性处理，大大降低产品的风险性，使产品的可靠性又上一个台阶。

这些故障检查定位电路主要是针对单板和系统的，包括：

1、在UHW接口处，设置协议错误检查电路，包括：HDLC协议错误，128自身定义的协议错误。

2、外部RAM存储错误检查电路和外部RAM自检电路。

3、总线错误检查电路。

4、收发帧数统计电路。

5、存储管理错误显示电路。

另外，为了保证可靠性设计能顺利实现，必须进行可靠性测试。

可靠性测试分两方面，一是通过软件进行，也就是写测试代码，如Verilog格式测试代码；二是进行硬件测试。

1、测试代码

在我们以往的测试代码中，过多地考虑的是基本功能测试，而在系统不稳定因素方面考虑较少。为了使测试代码更加完善，尽可能模拟实际运行情况，开发人员须主动向系统人员了解与芯片接口有关的信号质量，包括正常情况和异常情况，以及异常时对芯片有哪些具体要求。

2、硬件测试

硬件测试应尽可能模拟实际应用环境，如果这一点不能保证的话，芯片的可靠性会大打折扣，同时芯片的基本功能是否完全正确也不能完全确定。当然，这需要通过部门之间的协调才能完成。

另外，在进行硬件测试时，应对系统测试人员进行必要的指导，与他们一起制定一个详细的测试方案，保证测试的全面性。

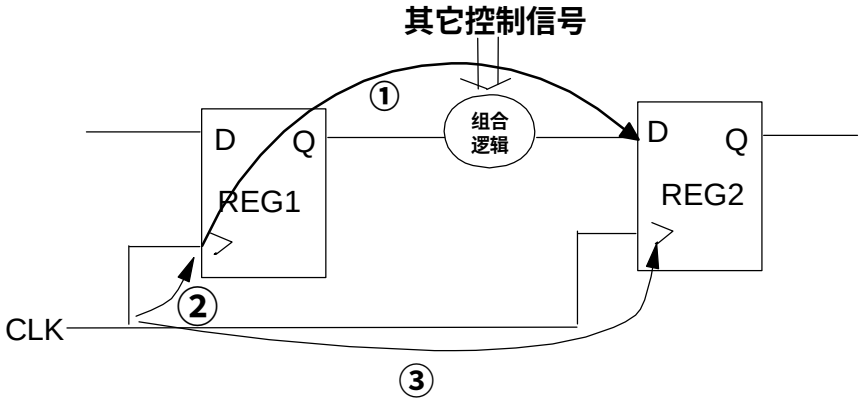
第四章同步电路设计技术及规则

1 设计可靠性

为了增加可编程逻辑器件电路工作的稳定性，一定要加强可编程逻辑器件设计的规范要求，要尽量采用同步电路设计。对于设计中的异步电路，要给出不能转换为同步设计的原因，并对该部分异步电路的工作可靠性(如时钟等信号上是否有毛刺，建立-保持时间是否满足要求等）作出分析判断，提供分析报告。

2 时序分析基础

电路设计的难点在时序设计，而时序设计的实质就是满足每一个触发器的建立/保持时间的要求。



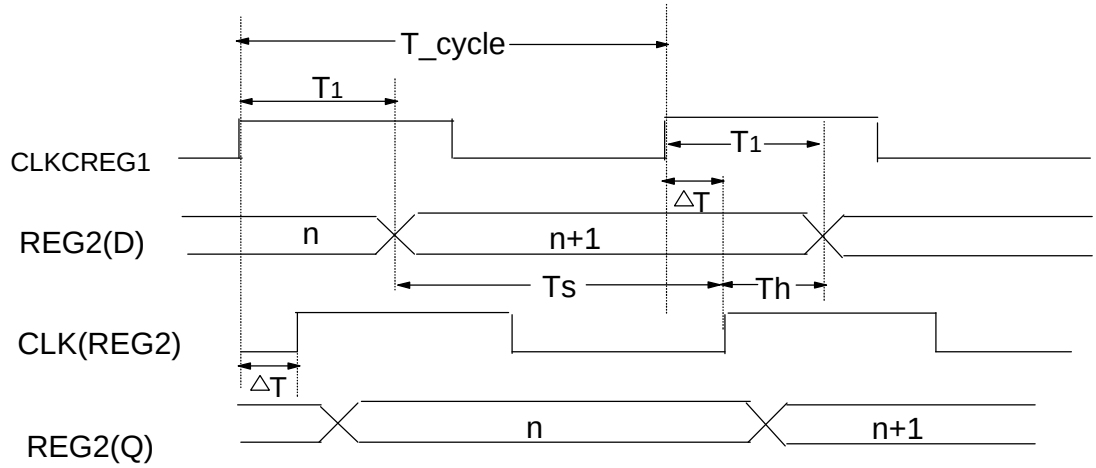


图1.1

如上图所示，以REG2为例，假定

触发器的建立时间要求为：T_setup，保持时间要求为：T_hold，

路径①延时为：T1，路径②延时为：T2，路径③延时为：T3，

时钟周期为：T_cycle，

Ts ＝（T_cycle ＋ △T）－T1，Th ＝T1－△T，

令 △T ＝T3－T2，**则**

条件1.如果T_setup < Ts ，**即** T_setup <（T_cycle ＋ △T）－T1，**这说明信号比时钟有效沿超过**T_setup **时间到达**REG2**的**D**端，满足建立时间要求。反之则不满足**；

条件2.如果T_hold < Th ，**即**T_hold < T1－△T ，**这说明在时钟有效沿到达之后，信号能维持足够长的时间，满足保持时间要求。反之则不满足。**

从条件1**和**2**我们可以看出，当**△T > 0 **时，**T_hold**受影响；当**△T < 0 **时，**T_setup **受影响。**

如果我们采用的是严格的同步设计电路，即一个设计只有一个CLK**，并且来自时钟**PAD**或时钟**BUFF（**全局时钟**），则**△T**对电路的影响很小，几乎为**0**；如果采用的是异步电路，设计中时钟满天飞，无法保证每一个时钟都来自强大的驱动**BUFF（**非全局时钟**），如下图所示，则**△T**影响较大，有时甚至超过人们想象。这就是为什么我们建议采用同步电路进行设计的重要原因之一。**

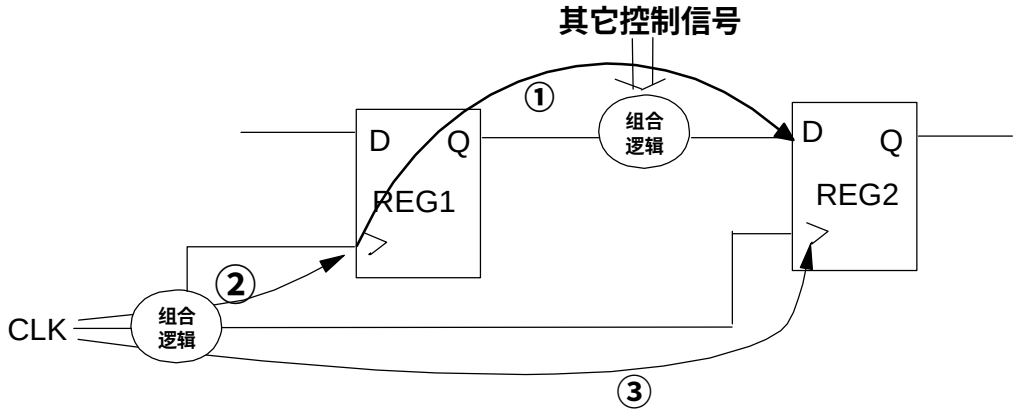


图1.2

3 同步电路设计

3.1 同步电路的优越性

1.**同步电路比较容易使用寄存器的异步复位/置位端，以使整个电路有一个确定的初始状态**；

2.**在可编程逻辑器件中，使用同步电路可以避免器件受温度，电压，工艺的影响，易于消除电路的毛刺，使设计更可靠，单板更稳定**；

3.**同步电路可以很容易地组织流水线，提高芯片的运行速度，设计容易实现**；

下图是一个设计中所要准备采用的电路，该设计采用 Xilinx**的**FPGA**器件**4062xla**来实现，工作频率是**32.768MHz（**即图中**CLK**频率）。设计原打算在每隔**60ns**输出一个数据，即**DATA。然而，我们在设计之前，考虑到**256x7**的同步**RAM**延时可能比较大，如果在加上其后的同步**RAM**延时的话，估计在**60ns**之内很难完成。该部分电路是整个设计中的一个关键路径，因此，我们在进行具体设计之前，先对这种电路结构进行了验证，事实证明我们的担心是对的。正确的做法是，采用流水线方法，在**256x7**的**RAM**之后再加一个触发器，每个**RAM**都按**60ns**的速度读取数据，整个流程滞后**60ns**输出**DATA。其它相关信号（在其它模块中）也随之滞后**60ns**输出。**

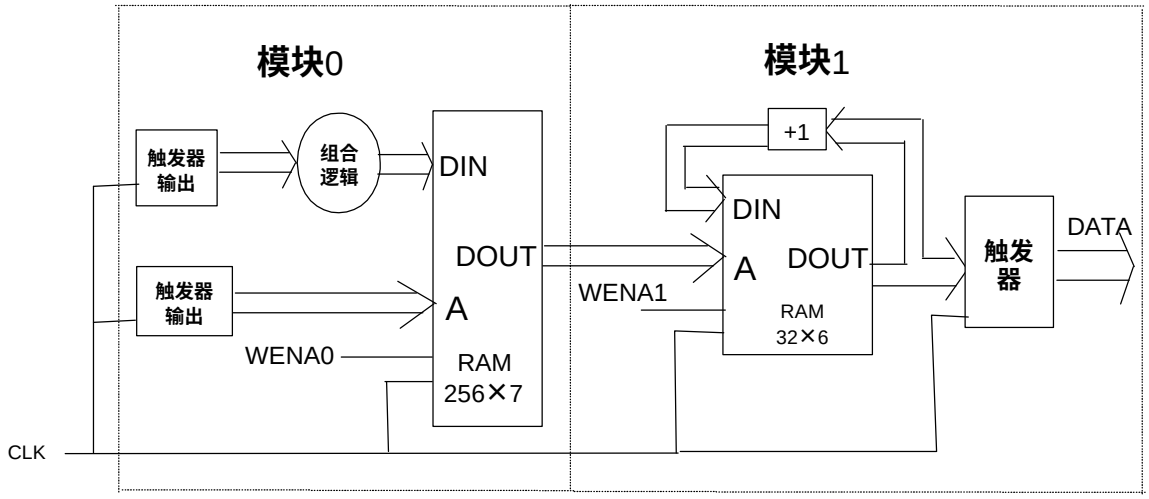


图1.3

4.同步电路可以很好地利用先进的设计工具，如静态时序分析工具等，为设计者提供最大便利条件，便于电路错误分析，加快设计进度。

3.2 同步电路的设计规则

1.尽可能在整个设计中只使用一个主时钟，同时只使用同一个时钟沿，主时钟走全局时钟网络。

2.在FPGA设计中，推荐所有输入、输出信号均应通过寄存器寄存，寄存器接口当作异步接口考虑。

3.当全部电路不能用同步电路思想设计时，即需要多个时钟来实现，则可以将全部电路分成若干局部同步电路（尽量以同一个时钟为一个模块），局部同步电路之间接口当作异步接口考虑。

4.当必须采用多个时钟设计时，每个时钟信号的时钟偏差（△T）要严格控制。

5.电路的实际最高工作频率不应大于理论最高工作频率，留有设计余量，保证芯片可靠工作。

6.电路中所有寄存器、状态机在单板上电复位时应处在一个已知的状态。

3.3 异步设计中常见问题及其解决方法

异步电路设计主要体现在时钟的使用上，如使用组合逻辑时钟、级连时钟和多时钟网络；另外还有采用异步置位、复位、自清零、自复位等。这些异步电路的大量存在，一是增加设计难度，二是在出现错误时，电路分析比较困难，有时会严重影响设计进度。

很多异步设计都可以转化为同步设计，对于可以转化的逻辑必须转化，不能转化的逻辑，应将异步的部分减到最小，而其前后级仍然应该采用同步设计。下面给出一些异步逻辑转化为同步逻辑的方法：

1.**组合逻辑产生的时钟**

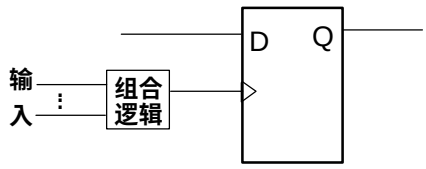


图1.4 **组合逻辑产生的时钟**

组合逻辑的时钟如果产生毛刺，易使触发器误翻转。

2.**行波计数器/行波时钟**

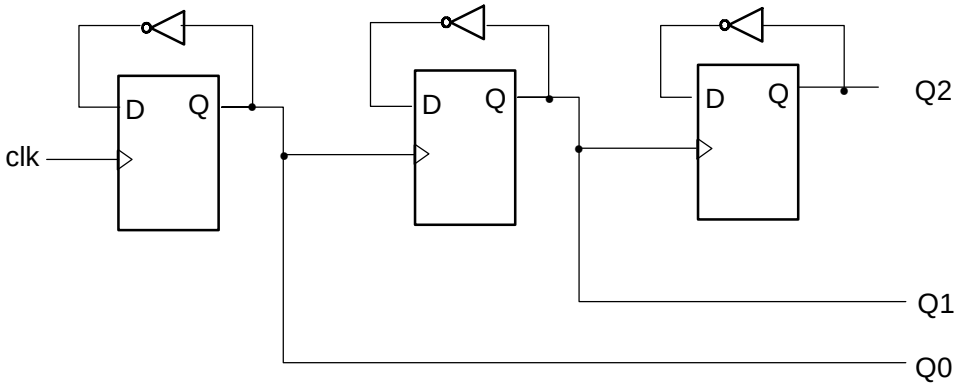


图1.5 **行波计数器**

行波记数器虽然原理简单，设计方便，但级连时钟（行波时钟）最容易造成时钟偏差（△T**），级数多了，很可能会影响其控制的触发器的建立/保持时间，使设计难度加大。转换的方法是采用同步记数器，同步计数器用原理图描述可能较难，但用**VHDL**很简单就可以描述一个**4**位计数器：**

```
Counter4:
Process(nreset,clk)
Begin
 If  nreset = ‘ 0’  then
    Cnt <= ( others => “0” );
```

```
 Elsif clk = ‘ 1’ and clk’ event then
    Cnt <= cnt + 1;
 End if;
End process counter4;
```

通常逻辑综合工具都会对上述描述按不同器件的特点进行不同的优化，我们并不需要关心它是逐位进位计数器还是超前进位计数器。

4.**不规则的计数器**

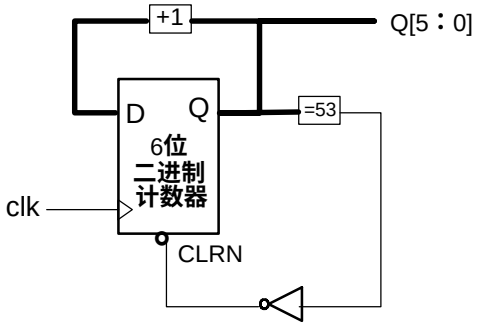


图1.6　**不规则的计数器**

这是一个53**计数器，采用计到**53**后产生异步复位的办法实现清**0**，产生毛刺是必然的。然而最严重的是，当计数器所有**bit**或相关**bit**均在翻转时，电路有可能出错，例如：计数器从“**110011**”—>“**110100**”，由于电路延时的原因，中间会出现“**110101**”状态，导致计数器误清**0**。**

采用同步清0**的办法，不仅可以有效地消除毛刺，而且能避免计数器误清**0**。电路如下图所示。**

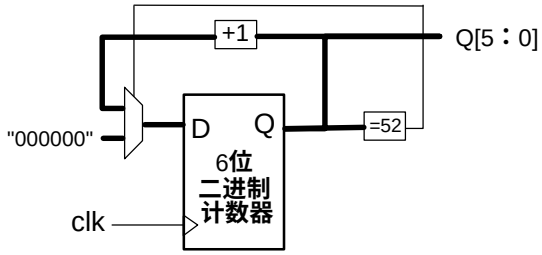


图1.7　**规则的计数器**

5.**分频器**

这是3**和**4**的特例，我们推荐使用同步计数器最高位的方法，如果需要保证占空比，可以使用图**1.8**所示电路进行最后一次二分频。下图是**19.44MHz**分频到**8kMHz**(分频数为**2430**)的电路：**

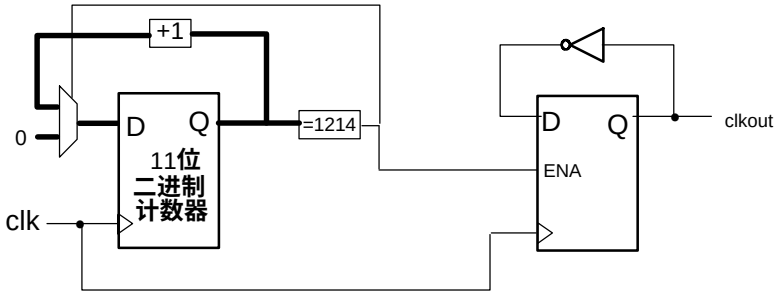
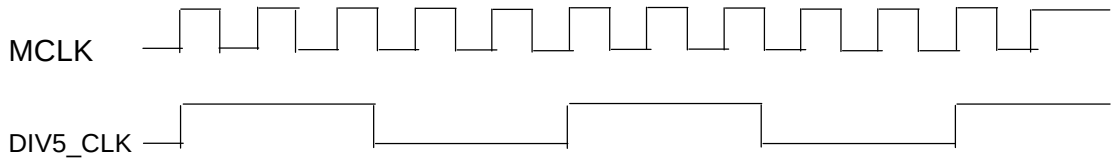


图1.8　**分频数为**2430**的电路**

若是奇数分频，则处理比较特殊，以5**分频器为例，其要求产生的时序关系如下图所示，**



很显然，该电路要用上MCLK**的上沿和下沿，对上图时序进行分解，得下图**

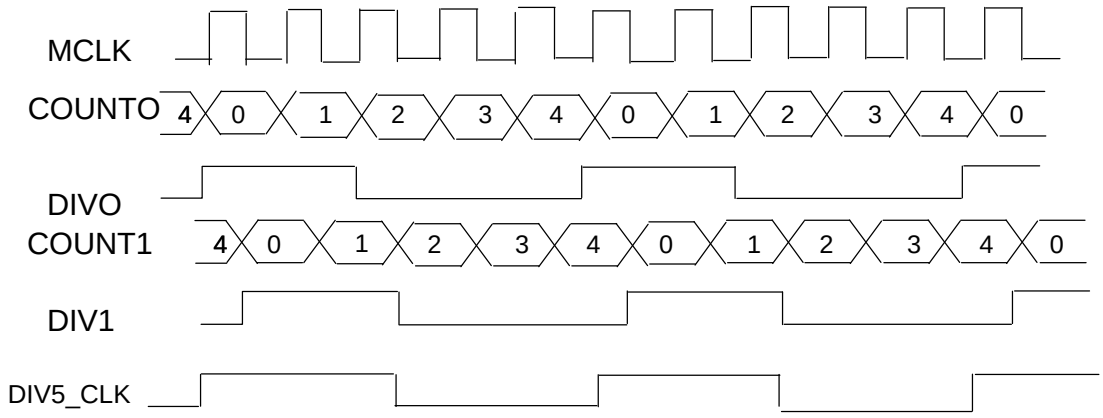


图1.9 5**分频信号时序分解**

图中，COUNT0**采用上沿计数，**COUNT1**采用下沿计数，**DIV0**和**DIV1**是分别是上沿触发器和下沿触发器的输出，**DIV5_CLK**是**DIV0**和**DIV1**的或门输出。读者可根据该时序图，画出相应的原理图，或者用**HDL**语言进行描述。**

在使用该电路时，需要注意：

（1）DIV0**和**DIV1**到**DIV5_CLK**的约束要严，越快越好。不然，无法保证**1:1**的占空比。**

（2）MCLK**频率要求较高，尽量不要出现窄脉冲，尤其是在高频电路里。**

（3）COUNT1**可有可无，视时钟频率高低而定。频率越高，**COUNT1**越需要。**

6.多时钟的同步化

我们在设计中，经常预见这种情况：一个控制信号来自其它芯片（或者芯片其它模块），该信号相对本电路来讲是异步的，即来自不同的时钟源。其模型可用图1.10表示。

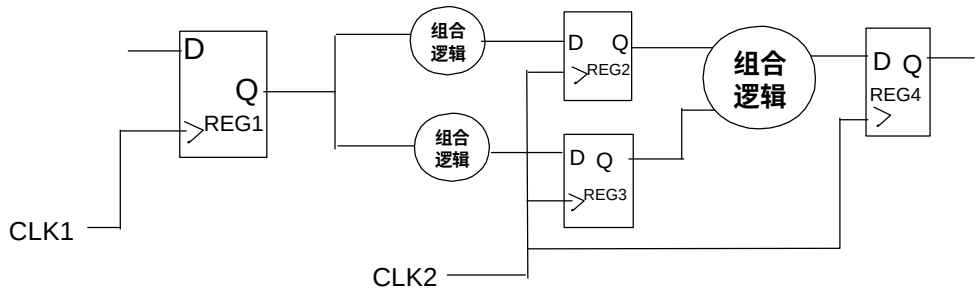


图1.10

在图1.10中，CLK1与CLK2来自不同的时钟源，该电路即可能出现在同一芯片里，又可能出现在不同芯片里。但效果是一样的，即存在危险性：由于时钟源不同，对REG2和REG3来讲，在同一时刻，一个“认为”REG1的输出是“1”，另一个认为是“0”。这必定造成电路判断出现混乱，导致出错。这种错误的实质是内部其它电路对输入控制信号（也可认为是状态信号）认识不一致，导致不同的电路进入不同的状态。正确的做法是在REG1之后再加一个触发器，用CLK2的时钟沿去采样，然后用该触发器的输出参与其后同步电路“活动”。如图1.11所示。

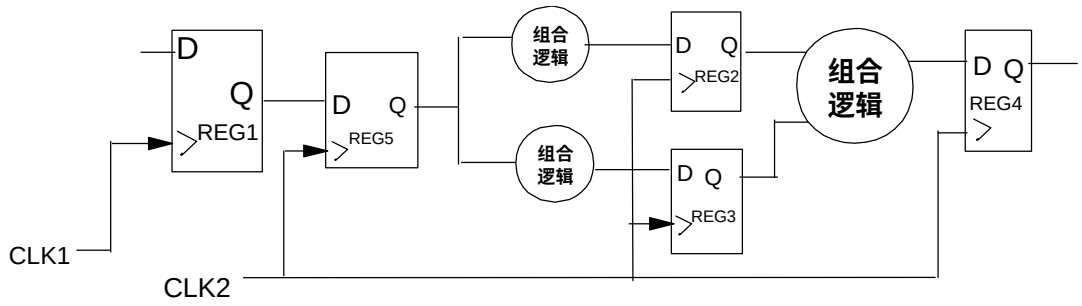


图1.11

如果输入信号是两根以上信号线，如下图所示，则该处理方法不准确。应引入专门的同步调整电路或其它特殊处理电路。我们在设计时，会对总线数据进行同步调整，却往往忽略了对一组控制信号进行同步调整。

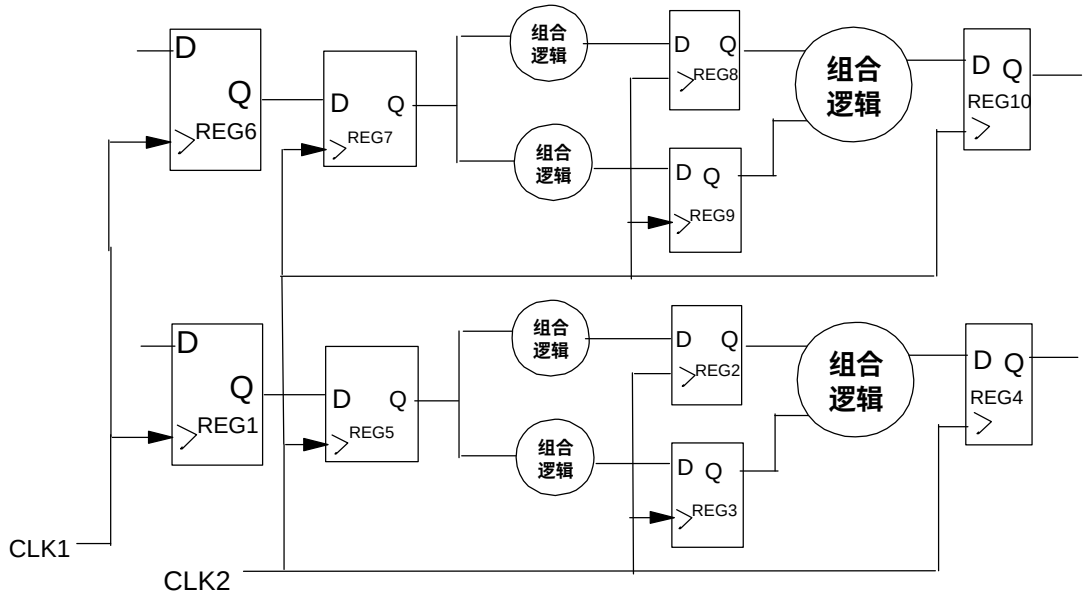


图1.12 **问题电路**

7.RS**触发器**

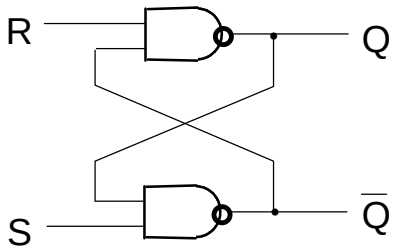


图1.13

RS**触发器是一种危险的触发器，**R=S=1**会导致不稳定态，初始状态也不确定。在设计时尽量避免采用这种电路，或用如图**1.14**电路改进**

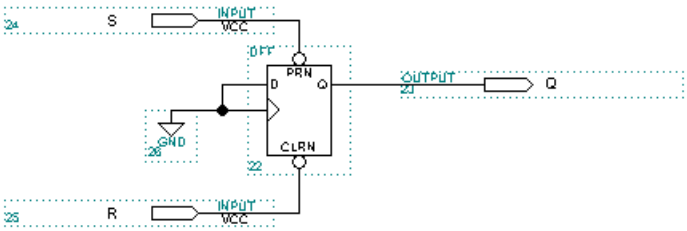


图1.14

可编程逻辑器件一般选用结构中的D**触发器来完成设计。**

我们认为，设计时最好从系统的角度来考虑，实现电路的功能，建议使用用VHDL**直接描述所需要的设计。这样作，既安全，又具有极大的灵活性：**

```
Process(nreset,clk)
Begin
 If  nreset = ‘ 0’  then
    Rs <= ‘ 0’ ;
 Elsif clk = ‘ 1’  and clk’ event then
  Case r&s is
        When “00” =>
         Rs <= rs;
        When “01” =>
         Rs <= ‘ 1’ ;
         When “10” =>
          Rs <=  ‘ 0’ ;
        When “11” =>
        --这里可以自定义R=S=1的行为
        When others =>
        Rs <= ‘ 0’ ;
  End case;
 End if;
End precess;
```

8.**上升沿检测**

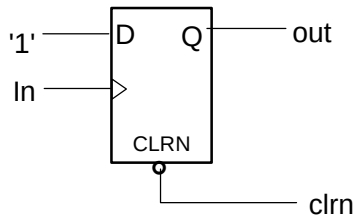


图1.15　**上升沿检测**

不建议采用这种电路检测信号的上升沿，因为IN**为数据信号，上边容易有毛刺，使触发器误动作。**

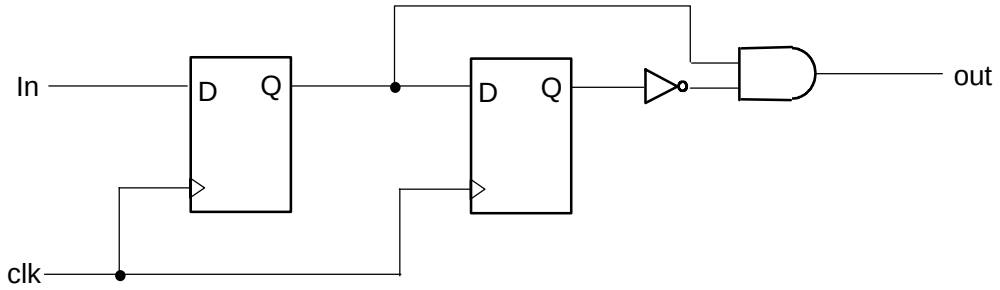


图1.16

采用时钟检测信号，出现0->1**的变化即为上升沿。当被测信号与时钟相关时，可以不用第一个触发器。**

9.**下降沿检测**

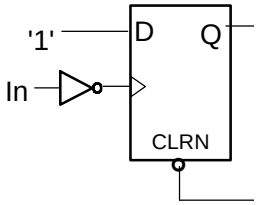


图1.17

同上升沿检测电路，这种电路对毛刺、干扰极为敏感。

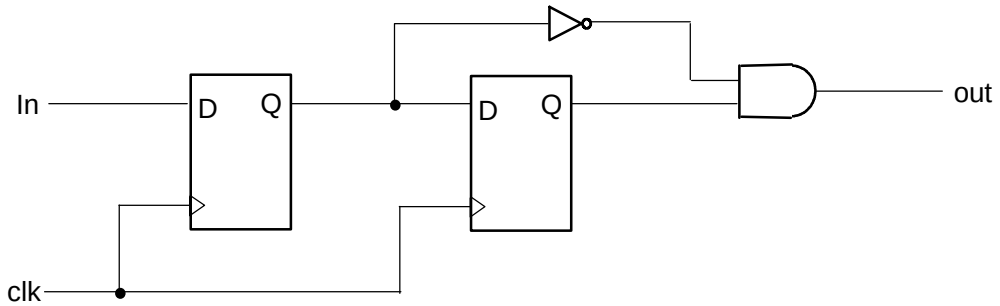


图1.18

采用时钟检测信号，出现1->0**的变化即为下降沿。 当被测信号与时钟相关时，可以不用第一个触发器。**

10.**上升/下降沿检测**

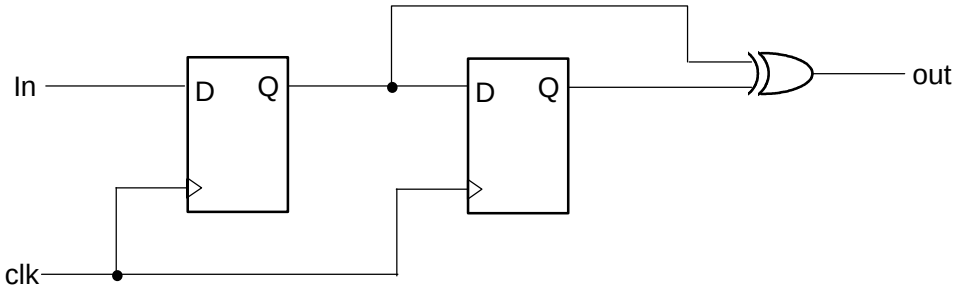


图1.19

采用时钟检测信号，出现变化即为上升/下降沿。 当被测信号与时钟相关时，可以不用第一个触发器。

11.对计数器的译码

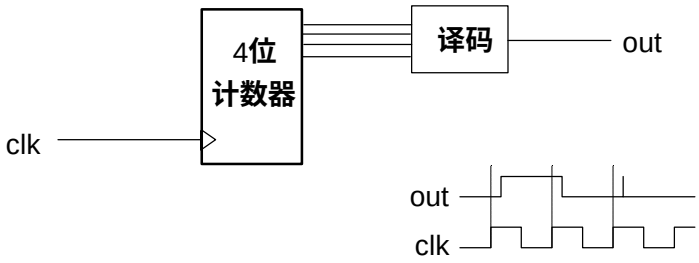


图1.20

对计数器译码，可能由于竞争冒险产生毛刺。如果后级采用了同步电路，我们完全可以对此不予理会。如果对毛刺要求较高，推荐采用Gray编码(PLD)或One-hot编码(FPGA)的计数器，一般不要采用二进制码.具体描述中，我们可以用状态机来描述，而利用逻辑综合工具来编码，有经验的选手可以自己强制定义状态机的编码。

12.门控时钟

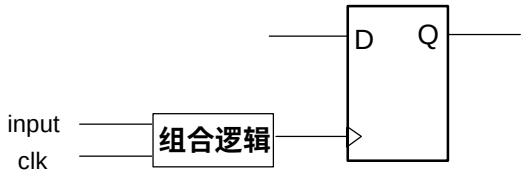


图1.21 门控时钟

门控时钟是非常危险的，极易产生毛刺，使逻辑误动作。 在可编程逻辑器件中，一般使用触发器的时钟使能端，而这样，并不增加资源，只要保证建立时间，可使毛刺不起作用。

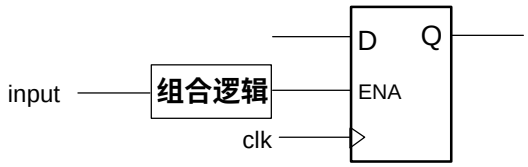


图1.22

13.锁存器

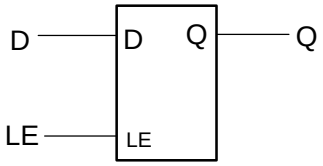


图1.23

锁存器是较危险的电路，没有确定的初始状态，输出随输入变化，这意味着毛刺可以通过锁存器。若该电路与其它D触发器电路相连，则会影响这些触发器的建立保持时间。除非有专用电路特别需要（其实总线锁存之类的功能用373之类的小规模IC更好），在设计内部，不要使用锁存器。

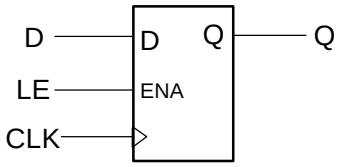


图1.24

引入时钟，可以实现锁存器的功能。

14 多级时钟或多时钟网络

由于时钟建立-保持时间的限制，FPGA设计中应尽量避免采用多时钟网络，或尽量减少时钟的个数。

图1.25是有问题的电路，这是一个含有险象的多级时钟的例子，多路选择器的输入是clk和clk的2分频，时钟是有SEL引脚控制的多路选择器输出的，在这两个时钟均为逻辑1时，当SEL线的状态改变时，存在静态冒险竞争现象，冒险竞争险象的程度取决于工作的条件。 图1.26是 改 进后的 电 路。

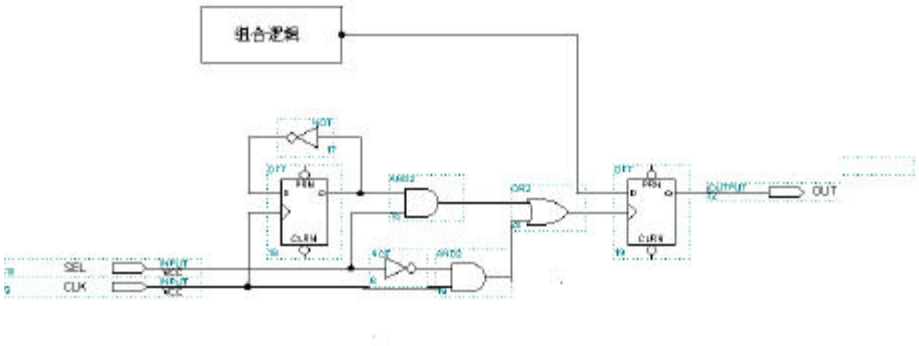


图1.25

改进：

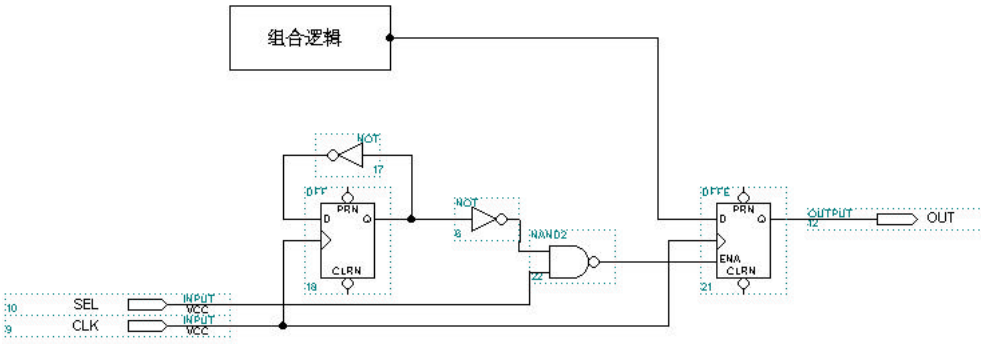


图1.26

总之，任何数字设计，为了成功地操作，时钟是关键。设计不良的时钟在极限的温度、电压或制造工艺的偏差下将导致错误的行为，而设计良好的时钟应用，则是整个数字系统长期稳定工作的基础。

对于一个设计项目来说，全局时钟（或同步时钟）是最简单和最可预测的时钟，在真正的同步系统情况下，由输入引脚驱动的单个主时钟去钟控设计项目中的每一个触发器。几乎所有的可编程逻辑器件中都有专用的全局信号引脚，它的特殊布线方式可以直接连到器件中的每一个寄存器。只要保证输入数据的建立和保持时间满足要求（相应可编程器件的数据手册中可查到），整个同步系统就可以在全局时钟的驱动下可靠工作，所以在可能的情况下，一定要使用全局时钟。

3.4 不建议使用电路

1 不建议使用组合逻辑时钟或门控时钟

组合逻辑很容易产生毛刺，用组合逻辑的输出作为时钟很容易使系统产生误动作。

2 不建议使用行波时钟

3 尽量避免采用多个时钟，多使用触发器的使能端来解决。

在可编程逻辑器件设计时，由于时钟建立应尽量避免采用多时钟网络，或者采用适当的措施减少时钟的个数，使用频率低的时钟尽量简化消除。

4 触发器的置/复位端尽量避免出现毛刺，及自我复位电路等，最好只用一个全局复位信号。

5 电路中尽量避免“死循环”电路，如RS触发器等。

6 禁止时钟在不同可编程器件中级连，尽量降低时钟到各个器件时钟偏差值。

同一个时钟在不同可编程器件中使用时不允许级连，而且应该保证该时钟到达不同可编程器件输入引脚的时钟偏差足够小。因为经过多个可编程器件的延时后，难以估计该时钟在链上各个触发器时钟之间产生的大量时间偏移，如果这种时间偏移引起建立时间、保持时间难以满足的话，那么设计的逻辑就极有可能要失败了。

4 SET和RESET信号处理

在设计时应尽量保证有一全局复位信号，或保证触发器、计数器在使用前已经正确清零和状态机处于确知的状态。

寄存器的清除和置位信号，对竞争条件和冒险也非常敏感。在设计时，应尽量直接从器件的专用引脚驱动。另外，要考虑到有些器件上电时，触发器处于一种不确定的状态，系统设计时应加入全局复位/Reset。这样主复位引脚就可以给设计中的每一个触发器馈送清除或置位信号，保证系统处于一个确定的初始状态。需要注意的一点是：不要对寄存器的置位和清除端同时施加不同信号产生的控制，因为如果出现两个信号同时有效的意外情况，会使寄存器进入不定状态。

对于状态机设计，由于有可能存在一些状态对于系统而言是“非法的”（或称“无关的”），所以除了在状态机设计时充分考虑各种可能出现的状态以及一旦进入“非法”状态后可以强迫状态机在下一个时钟周期内进入“合法”状态（一般时初始状态）外，一定要保证系统初始化时状态机就处于“合法”的初始状态，这里最好的办法仍然是使用全局主复位信号强迫状态机进入已知的“合法”状态。下面给出一段包含状态机的全局时钟和复位的AHDL描述：

```
SUBDESIGN example
        (
                clock           : INPUT ;
                /reset          : INPUT ;
        )
        VARIABLE
         -- State Machine declaration
                poll                    : MACHINE WITH STATES
                                        (Idle,s1,s2);
         -- Varible & Counter declaration
                var1                    : DFF;
                var2                    : DFFE;
                counter[5..0]           : DFF;
```

```
BEGIN
 -- global reset & clock, initial set to 1
        var1.clk = GLOBAL(clock);
        var1.prn = GLOBAL(/reset);

 -- global reset & clock, initial set to 0
        var2.clk = GLOBAL(clock);
        var2.clrn = GLOBAL(/reset);

-- global reset & clock for counter,  initial value set to 0
        counter[].clk = GLOBAL (clock);
        counter[].clrn = GLOBAL (/reset);
        IF counter[] == 53 THEN
                counter[] = 0;
        ELSE
                counter[] = counter[] + 1;
        END IF;

 -- state machine global clock & reset, initial state set to idle
        poll.clk = GLOBAL (Clk25m);
        poll.reset = not GLOBAL (/Reset);
        CASE poll IS
                WHEN idle =>
                        。
                        poll = s1;
                        。
                WHEN s1 =>
                        。
                        poll = s2;
                        。
                WHEN s2 =>
                        。
                        poll = s1;
                        。
                WHEN OTHERS =>
                        poll = idle;
        END CASE;
```

END

5 时延电路处理

时延电路是指在可编程器件的设计中，为了能够满足电路之间时序配合的要求，利用可编程器件的内部资源而进行时序调整，如：利用线延时或者若干串联 Buffer 电路，使时钟或数据滞后一段时间。典型的，该时间即不能太长，又不能太短，以便满足“特定”需要。

然而，这种时延电路严重依赖器件工艺，也依赖每次的布线结果，给设计带来许多麻烦，有时会引起严重后果。因此，必须正确处理时延电路：

1

在设计中应尽量避免时延电路，绝大多数时延电路是由设计者在设计之初考虑不完善造成的。例如，在设计中没必要地存在多个时钟电路，模块划分不合理，关键电路关键时序考虑不周等。

我们的设计经常是在功能模块划分完成之后，就急急忙忙去做具体电路，许多关键时序并没有考虑清楚，这不可避免的造成设计多次反复。正确的做法是：

设计目标分析 ⟶ 功能模块划分 ⟶ 确定关键电路时序和模块间接口时序 ⟶ 具体电路设计

设计电路，尤其是数字电路，最关键的一环是：设计各模块间的接口时序，确定关键电路的时序。这个工作必须在具体电路设计之前确定下来。

“时序是事先设计出来的，而不是事后测出来的，更不是凑出来的”。

2 若实在无法，则尽量采用高频电路，对所需信号加触发器进行延时。该延时只跟时钟频率和触发器个数有关，而与工艺基本无关。

6 全局信号的处理方法

全局信号处理的原则是：时钟信号、异步清零、置位信号上不允许存在毛刺；不允许异步清零、置位信号同时有效。

在下述几种情况下，时钟信号、异步清零、置位信号上可能会有毛刺：

（1） 时钟信号、异步清零、置位信号为组合逻辑输出

由于组合逻辑是电平敏感的，比较容易产生毛刺，而组和逻辑的细小毛刺一旦经过时序电路则其对电路的影响则会放大。因此在设计中对时钟信号、异步清零、置位信号这些对时序电路来讲非常重要的信号应尽量采用同步电路，而对于非用组合逻辑不行的地方则必须用卡诺图严格的分析时序电路，确定彻底消除竞争与冒险后才可引入到时序电路中使用。下面举例说明：

- 对于必须用组合逻辑的输出作为时钟、异步清零、置位信号的电路，对其组合电路的输出必须采用卡诺图进行严格的分析，保证彻底消除了竞争与冒险后才可引入到时序电路中使用。对于时钟信号、异步清零、置位信号是多个信号中选择的情况可按下述方法同步化

对组合电路产生的时钟信号的处理：

情况1：同一个时钟源，通过组合逻辑控制它的通断，如下图：

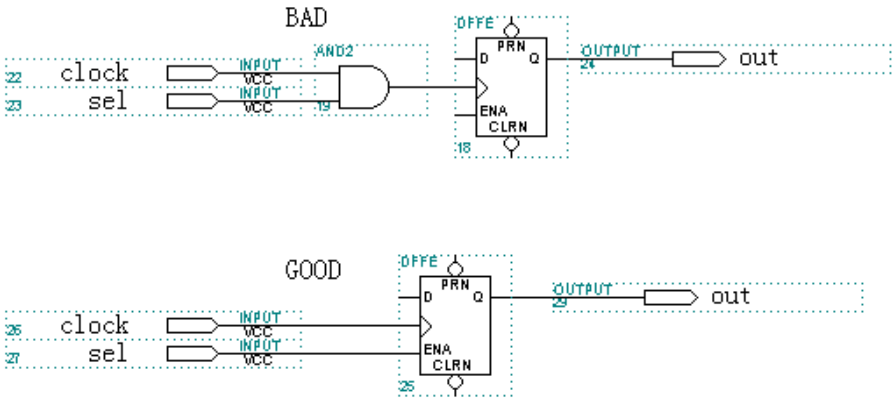


图 1.27　通过组合逻辑控制它的通断

上面的图中，触发器的时钟是由CLOCK与SEL相与后的输出，目的是通过控制触发器的时钟的通断达到控制触发器的输出的目的，这种情况下由于CLOCK与SEL不是严格同步的，则有可能会在触发器的时钟脚产生毛刺，而在FPGA中，触发器对时钟端的毛刺是敏感的，上述毛刺可能导致触发器的误动作。

在下面的图中，触发器的时钟是CLOCK，而将SEL信号与触发器的使能端连接，这样同样达到了通过SEL信号控制触发器的输出的目的，但由于CLOCK不会产生毛刺，可以保证触发器的可靠触发。

情况2：通过组合逻辑对触发器的时钟在多个时钟中选择一个，如下图所示：

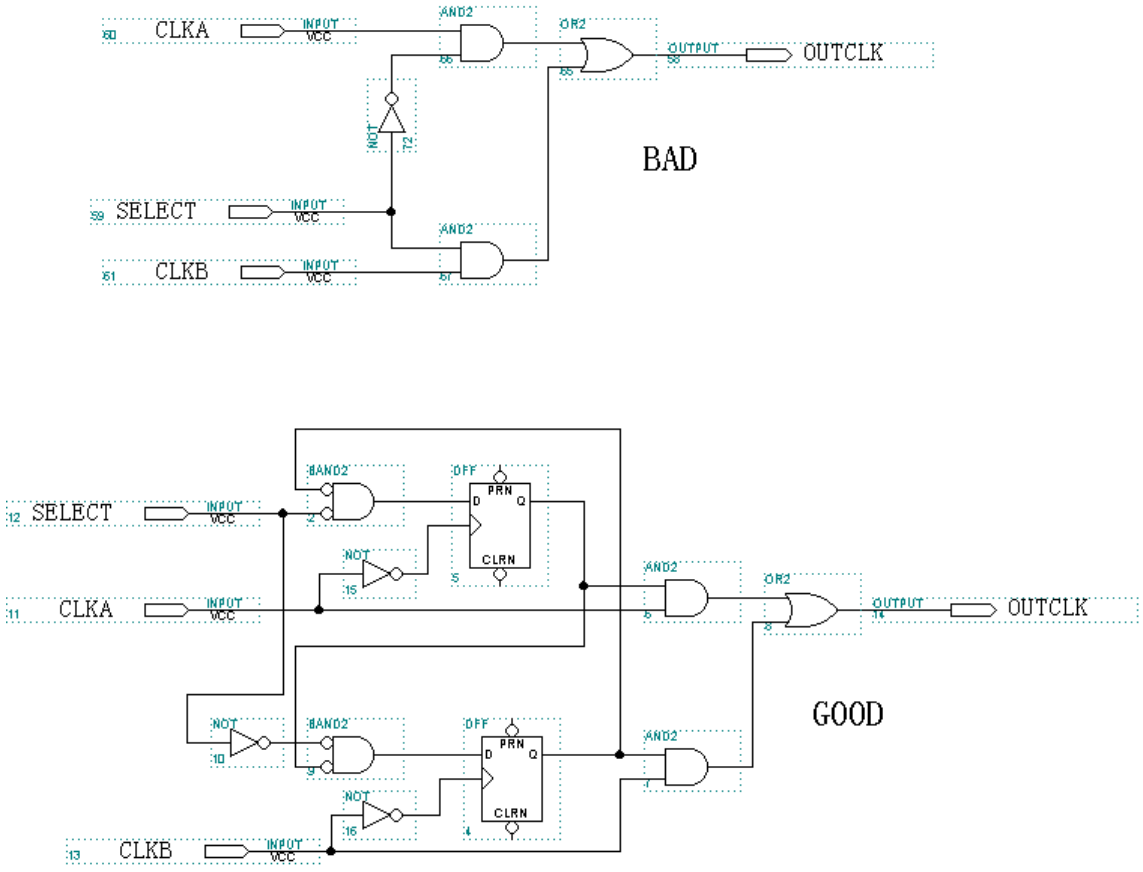


图1.28 **组合逻辑在多个时钟中选择一个**

上图中，标为BAD**的一个由于直接用组合逻辑实现在**CLKA、CLKB**两个时钟中二选一的功能，而组合电路由于不同路径的延迟不同，所以在电路的时钟输出很容易产生毛刺；而标为“**GOOD**”的一个由于对选择信号**SELECT**分别用两个触发器进行了同步化，所以在时钟的输出端不会产生毛刺。**

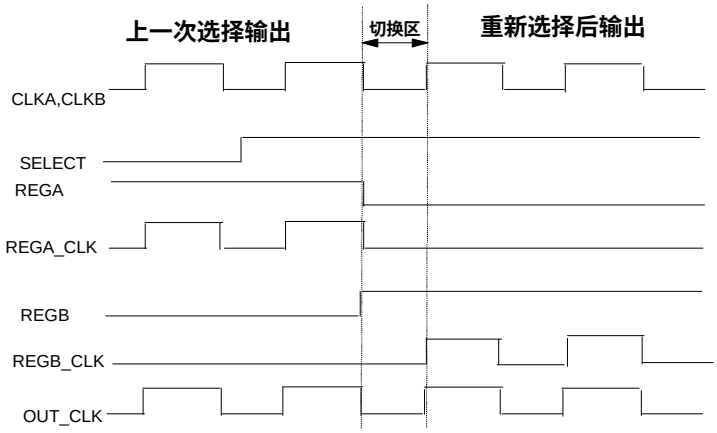


图 1.29

图1.29是图1.28中GOOD电路的原理时序图。其中REGA表示接CLKA的触发器，其对应的与门输出为REGA_CLK；同样定义REGB和REGB_CLKB。很显然，当REGA或REGB前后有限“漂移”时，OUT_CLK是不会出现毛刺得。

对GOOD电路，有两个问题：

1. SELECT如果在CLK下降沿左右发生变化，该电路能否正常？
2. 该电路的触发器能采用上升沿触发吗？

（2）使用自我清除、自我置位和自我钟控的寄存器；

• 建议尽量使用触发器的使能端来达到钟控的目的。

对于自我清除、自我置位，最好通过组合逻辑将其转变成对输入端数据的置位与清除，从而提高电路的可靠性，具体说明如下二图：

图1.30是一个自我清除的电路图，当输出端均为“1”时，电路复位，这样消除了产生毛刺的隐患，但使输出端体现出复位延迟了一个CLOCK。

图1.31是一个自我置位的电路图，当输出端均为“0”时，电路置位，这样做输出端体现出置位延迟了一个CLOCK，但消除了产生毛刺的隐患。

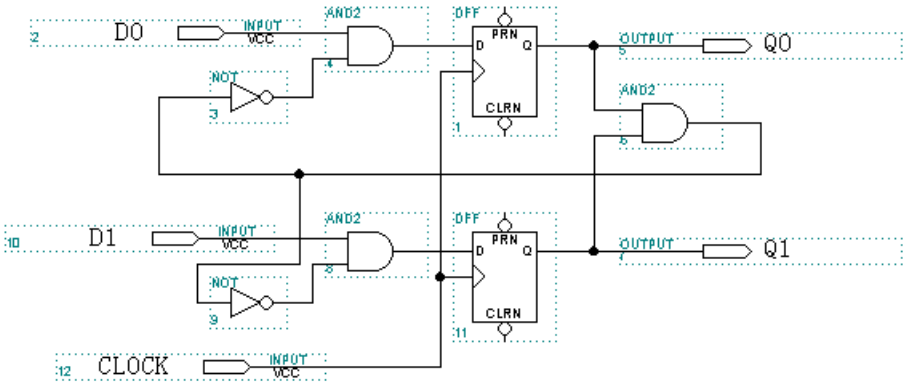


图 1.30

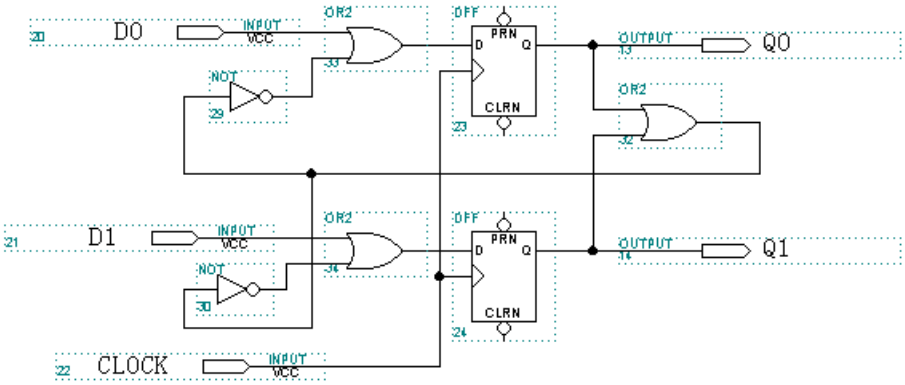


图 1.31

（3）时钟信号、异步清零、置位信号由外部输入，输入时有毛刺；

对于由外部输入的有毛刺的时钟、异步清零、置位信号通常的处理方法是使用内部触发器锁存，使之同步化。

7 时序设计的可靠性保障措施

（1）时钟偏差要加以控制

在同步电路里，时钟信号要连接到所有的寄存器，触发器以及锁存器等器件上。这些巨大的负载就象一个大电容加在时钟线上，再加上时钟线本身的分布电容和电阻，这样时钟线就象分布的RC线。由于RC线的延时是线长的函数，这样就使得连到同一根时钟线上的时钟由于距离时钟源的远近不一而产生不同的延时。因而造成了同一时钟到达各个器件的时间不一致，使得各个以时钟为基准器件的动作也不一致，而造成时序上的混乱。这就是同步电路时钟偏差。

要让同步电路可靠地运行，就要对时钟偏差进行控制，以使时钟偏差减小到可用的范围。影响时钟偏差的主要有以下几个因素：

1、用于连接时钟树的连线

2、时钟树的拓扑结构

3、时钟的驱动

4、时钟线的负载

5、时钟的上升及下降时间

在通常的FPGA设计中对时钟偏差的控制主要有以下几种方法：

1、控制时钟信号尽量走可编程器件的全局时钟网络。在可编程器件中一般都有专门的时钟驱动器及全局时钟网络，不同种类、型号的可编程器件，它们中的全局时钟网络数量不同，因此要根据不同的设计需要选择含有合适数量全局时钟网络的可编程器件。一般来说，走全局时钟网络的时钟信号到各使用端的延时小，时钟偏差很小，基本可以忽略不计。

2、若设计中时钟信号数量很多，无法让所有的信号都走全局时钟网络，那么可以通过在设计中加约束的方法，控制不能走全局时钟网络的时钟信号的时钟偏差。如在UCF文件中加上：NET *** MAXSKEW=3；

一般而言，时钟偏差的控制值应按保持时间要求来计算： $\triangle T < T1 - T\_hold$ 。

注意：Xilinx器件应用中，MAXSKEW约束必须直接加在触发器的时钟端上，MAXSKEW约束不能通过任何门电路，尤其是对芯片PAD进行约束时，应对IBUF之后的信号加约束，许多人在初学时经常忘记这一点。

3、异步接口时序裕度要足够大

局部同步电路之间接口都可以看成是异步接口，比较典型的是设计中的高低频电路接口、I/O接口，那么接口电路中后一级触发器的建立-保持时间要满足要求，时序裕度要足够大。

4、在系统时钟大于30MHz时，设计难度有所加大，建议采用流水线等设计方法。

采用流水线处理方式可以达到提高时序电路的速度，但使用的器件资源也成倍增加。

5、**要保证电路设计的理论最高工作频率大于电路的实际工作频率**；

最高频率是指设计软件经综合，布局，布线后，软件计算出的可工作的最高频率。比如经过综合及布局布线后，最长的延时为20ns,**则其理论最高工作频率为**50MHz。**需要指出的是：此频率更设计有很大关系，良好的设计可使器件工作在更高的频率范围。**

所谓电路的实际工作频率是指所设计的逻辑电路实际要达到的工作频率，既实际电路中所使用的时钟频率。

要使时序电路可靠地工作就必须使理论最高工作频率大于电路的实际的工作频率，但理论要高于实际的值是多少，这需要根据实际情况而定。主要要考虑的是时钟的最大可能抖动是多少。

8 ALTERA参考设计准则

Ensure Clock, Preset, and Clear configurations are free of glitches.

Never use Clocks consisting of more than one level of combinatorial logic.

Carefully calculate setup times and hold times for multi-Clock systems.

Synchronize signals between flipflops in multi-Clock systems when the setup and hold time requirements cannot be met.

Ensure that Preset and Clear signals do not contain race conditions.

Ensure that no other internal race conditions exist.

Register all glitch-sensitive outputs.

Synchronize all asynchronous inputs.

Never rely on delay chains for pin-to-pin or internal delays.

Do not rely on Power-On Reset. Use a master Reset pin to clear all flipflops.

Remove any stuck states from state machines or synchronous logic.

第五章 VHDL数字电路设计指导

关键词：VHDL，设计重用

摘要：

缩略语清单：

VHDL：Very High speed IC Hardware Description Language

FSM：Finite Status Machine

DFT：Design For Test

参考资料清单				
名称	作者	编号	发布日期	查阅地点或渠道
VHDL代码书写规范	苏文彪，喻志清，周志坚			硬件开发园地，中研基础部
同步电路设计方法及规则	周志坚等			中研基础部

1 前言

目前，关于VHDL语言方面的书籍多如牛毛，但这些书籍绝大对数是为了讲语言而讲语言，缺乏针对性，尤其缺乏如何正确使用VHDL语言或VHDL使用注意事项等有关设计技巧方面的文章。

本文就是出于这个目的而作。因此，我们要求读者再看这篇文章之前，必须有一定VHDL语言基础。

2 VHDL代码风格

一个好的设计，不仅仅能够实现所需要的所有功能，还应当具有设计可重用、可移植、可维护等特点，便于设计经验技术的交流与积累，形成“货架”，加快设计进度。

2.1 代码编写风格

编写代码时，应当尽量考虑可重用因素：

1. 代码信号、变量等的命名通俗易懂，便于交流和维护。这要求我们的代码尽可能规范，尽可能多加注释。

2. 尽量采用package和generic，方便代码修改和移植。

3. 项目组内设计风格一致，减少不必要的误会。

4. 在同等和可能的条件下，尽量采用与工艺无关的代码，只要不影响大局，必要时可以牺牲一定的面积。但是，若对芯片的面积要求比较苛刻，则应当尽量采用厂家提供的各种基本单元。

5. 原则上，一个process处理一个信号，思路清晰，便于理解，有利于今后设计更改与查错。

6. 按照“VHDL代码书写规范”进行编码。

2.2 代码模块划分

模块设计的好坏直接影响着系统的设计好坏，模块设计的不好，会给后面的设计流程带来许多麻烦。模块设计的基本原则是：

1. 模块划分有利于模块的可重用性

模块设计得好，可节省大量的重复工作，并且为以后的设计带来方便。

2.在组合电路设计中应当没有层次

一方面可提高代码的可读性，另一方面方便综合，时序较易满足。

3.每个模块输出尽量采用寄存器输出形式

设计划分（Design Partition）必须以寄存器为边界。划分设计的层次使得每个模块的输入和输出都是寄存器信号。这样有利于时序设计（加约束），时序容易满足。

4. 模块的按功能进行划分，划分要合理

5. 异步逻辑与同步逻辑除了必要的衔接部分，尽量不要在同一个模块中

6. 尽量区分面积敏感和速度敏感电路部分，并放在不同的模块中

7. 模块间不要有glue 关系，如下：

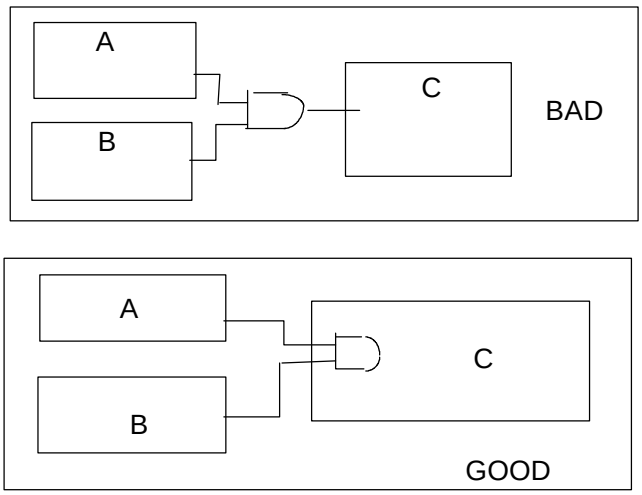


图1 glue关系

8. 模块大小应适中，不能太大，也不能太小，一般为2000门左右。具体情况，应当依据综合工具的性能而定。

9.模块的层次应当至少有三级

可将一个设计划分为三个层次：TOP、MID、功能CORE

•TOP

包括实例化的MID和输入输出定义（如果用综合工具插入管脚则可不要此层次）；

•MID

由两部分组成：1）时钟产生电路，如分频电路和倍频电路；2）功能CORE的实例化。

•功能CORE

包括各种功能电路的设计。一个复杂的功能可以分成多个子功能来实现，即再划分子层。

3 常见问题

3.1 不可综合的代码

3.1.1 信号或变量赋初值

对大多数综合工具而言，信号或变量所赋的初值在综合时会被忽略掉，可能造成设计错误。请不要使用类似如下案例的语句：（这是对VHDL语言而言）

```
variable SUM:interger = 0 ;
```

3.1.2 采用时间相关语句（仿真语句）

对综合工具而言，所有带延时的语句在综合时均报错。因为从功能上讲，只存在正常的与或非等逻辑，不存在“延时逻辑”。在ASIC设计中，可以采用厂家提供的专门延时模块（Delay Buffer）来实现特定延时，而FPGA设计中则无类似的Delay Buffer。

在FPGA设计时，实现特定延时非常麻烦，且很容易受每次的布局布线结果所干扰。建议采用其它方法避免采用延时电路，如采用高频时钟处理方法等。

时间相关语句主要目的是用来编写测试代码，仿真验证电路功能是否正确，可以认为是仿真语句。常见的仿真语句有：

```
wait for 10 ns ;
Data_in <= 55 after  MCLK_PERIOD ;
```

其中，MCLK_PERIOD是定义的时间常数。

3.2 采用std_logic以外的信号类型

采用VHDL进行设计时，我们建议信号、变量、端口尽量使用STD_LOGIC或其派生类型，这样做的目的是为了统一信号格式，信号连接方便，不容易出错误，尤其是模块与模块之间连接时（port map）。因为一旦采用其它类型，例如BIT，则与此相连接的所有线都得定义成BIT，人为增加麻烦。

Synopsys公司也建议在使用VHDL编写代码时，尽量只采用STD_LOGIC类型。

3.3 错误使用inout

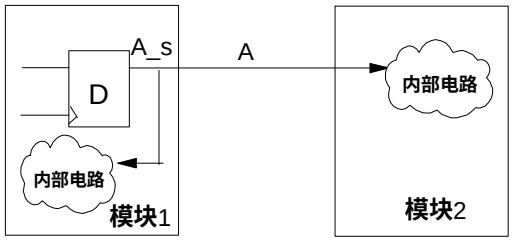


图2 内部反馈电路

如上图所示电路，对模块1而言，信号A既是输出，又是“输入”（内部反馈），因此在entity定义里面，想当然地采用了inout类型：

.....

A : inout STD_LOGIC ;

.....

然而，inout**一般是用来表示双向口电路，如下图所示：**

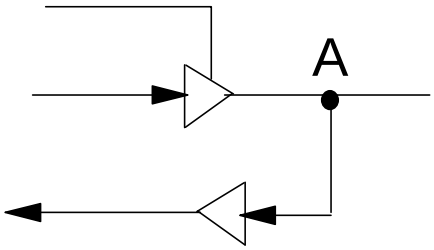


图3 **双向口**

为了避免异意，我们建议采用另起一个信号A_s**来处理**（“s”**表示**“same”），**方法如下：**

1. **在**architecture**中定义信号**A_s

signal A_s : std_logic ;

2. **在本**entity**内，采用**A_s**信号处理一切逻辑，包括**A_s**信号的产生**。

3. **对**A**信号赋值，如下：**

A <= A_s ;

3.4 产生不必要的Latch

在处理组合逻辑电路时，初学者经常犯如下错误：

```
process （A，B，C，Cntr）
begin
    if  Cntr  = '1' then
            A <= '1' ;
            B <= '1' ;
    else
            C <= '1' ;
    end if;
end process ;
```

结果，A、B、C**都成为了**LATCH。**为了避免产生**LATCH，**正确处理方法是**

```
process （A，B，C，Cntr）
begin
    if  Cntr  = '1' then
            A <= '1' ;
            B <= '1' ;
            C <= '0' ;  -- 缺省处理
    else
```

```
            A <= '0' ;  -- 缺省处理
            B <= '0' ;  -- 缺省处理
            C <= '1' ;
    end if;
end process ;
```

同样，在使用case语句时，每种情况下所有的信号都必须处理到。或者，在case语句之前，对所有的信号赋一个初值，例如：

```
process （A，B，C，D，S）
begin
    A <= '0' ;
    B <= '0' ;
    C <= '0' ;
    D <= '0' ;
    case S is
            when  "00" =>
                    A <= '1' ;
            when "01" =>
                    B <= '1' ;
            when "10" =>
                    C <= '1' ;
            when "11" =>
                    D <= '1' ;
            when others =>
                    null ;
    end case ;
end process ;
```

3.5 同一个信号在两个或两个以上的process中赋值

除了三态输出信号，其它信号是不允许在不同的process中分别赋值。

3.6 错误地使用变量或信号（VHDL）

变量只能在进程语句、函数语句和子程序结构中使用，它是一个局部量。变量的赋值是立即生效的。信号是电路内部硬件连接的抽象，信号的赋值在进程结束后才进行。 信号的赋值与顺序无关，变量赋值与顺序有关。

下面的VHDL实例显示了一个设计分别使用信号和变量的综合结果，请仔细比较异同。

注意，如果在一个进程中对信号赋多次值，只有最后的值有效。变量的赋值立即生效，变量在赋新值前保持原来的值。

例1、组合逻辑中使用信号

```
-- XOR_SIG.VHD
-- May 1997
Library IEEE;
use IEEE.std_logic_1164.all;

entity xor_sig is
    port (A, B, C: in STD_LOGIC;
          X, Y: out STD_LOGIC);
end xor_sig;
architecture SIG_ARCH of xor_sig is
    signal D: STD_LOGIC;
begin
    SIG:process (A,B,C)
    begin
        D <= A; -- ignored !!
        X <= C xor D;
        D <= B; -- overrides !!
        Y <= C xor D;
    end process;
end SIG_ARCH;
```

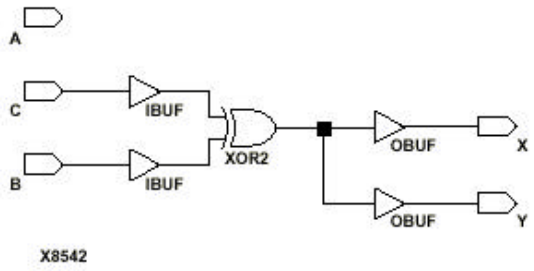


图4 组合逻辑中使用信号

例2、组合逻辑中使用变量

```
-- XOR_VAR.VHD
-- May 1997
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity xor_var is
      port (A, B, C: in STD_LOGIC;
            X, Y: out STD_LOGIC);
end xor_var;
architecture VAR_ARCH of xor_var is
begin
      VAR:process (A,B,C)
```

```
        variable D: STD_LOGIC;
    begin
        D := A;
        X <= C xor D;
        D := B;
        Y <= C xor D;
    end process;
end VAR_ARCH;
```

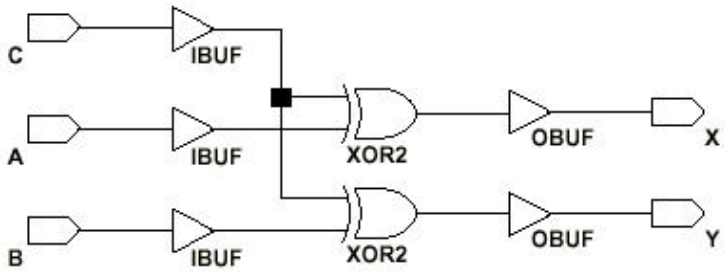


图5 组合逻辑中使用变量

例3、 触发器中使用信号

```
P1:process（CLK)
begin
    if (CLK = ‘ 1’ and CLK ‘ event ) then
        X <= A and B;
        Y <= X;
        Z <= Y;
    end if;
end process;

P2:process（CLK)
begin
    if (CLK = ‘ 1’ and CLK ‘ event ) then
        Z <= Y;
        Y <= X;
        X <= A and B;
    end if;
end process;
```

p1, p2 **综合的结果是一样的：**

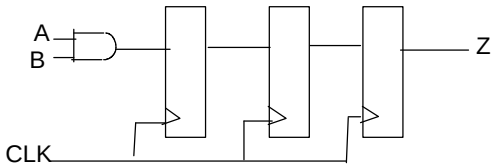


图6 触发器中使用信号

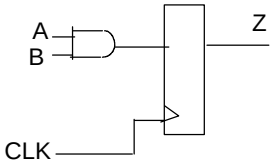
例4、 触发器中使用变量

```
P1:process (CLK)
        variable  X , Y : std_logic;
begin
        if (CLK ‘ event and CLK = ‘ 1’ ) then
                X  := A and B;
                Y  := X;
                Z  <= Y;
        end if;
end process;

P2:process (CLK)
        variable  X , Y : std_logic;
begin
        if (CLK ‘ event and CLK = ‘ 1’ ) then
                Z  <= Y;
                Y  := X;
                X  := A and B;
        end if;
end process;
```

则综合结果为：

P1：



P2：

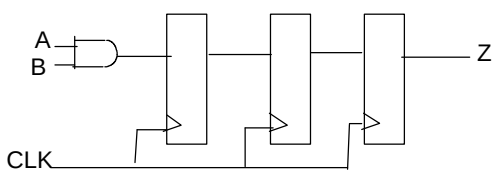


图7 触发器中使用变量

合理使用变量，可方便地实现较为复杂的逻辑电路，如并行CRC电路（所谓并行是指一次处理多个bit）。但变量使用很容易出错，且有时不方便交流、理解和查错。

建议：在对变量不是很把握的情况下，尽量少使用变量；若无必要，则不使用变量。

3.7 合理使用内部RAM

当内部RAM大到一定程度时，例如64X16的同步RAM，此时每个地址线驱动的基本单元都比较多，电路扇出很大，我们建议：

（1）地址线直接来自专门地址寄存器

（2）RAM的输出直接接寄存器

这样，保证地址线的扇出较小，且按照流水线设计，可获得较高频率。

因此，我们要求设计者在设计之前，必须了解所采用的RAM最快速率是多少，以决定是否加和如何加寄存器。

3.8 三态电路设计（参考VERILOG）

三态电路一般用于：

（1）双向口；

（2）多路数据竞争总线或者多路选择电路。

注意：*在ASIC设计里，除了芯片的双向口或输出口可用三态外，芯片内部其它部分不允许出现三态，这是因为ASIC设计里，考虑到DFT（Design For Test）等因素，一般ASIC厂家在芯片内部不提供三态电路。因此，在ASIC设计中，三态电路一般用组合逻辑替代*。

双向口的处理比较简单，我们主要考虑第（2）种情况。

考虑一根总线问题，如下例所示，A、B竞争总线T：

```
Tri_p1: process (Sel_a,A)
begin
        if (Sel_a = ‘ 1’ ) then
                T <= A;
        else
                T <= ‘ Z’ ;
        end if;
end process;

Tri_p2: process (Sel_b,B)
begin
        if (Sel_b = ‘ 1’ ) then
                T <= B;
        else
                T <= ‘ Z’ ;
        end if;
```

```
    end process;
```

为什么不能在一个process中进行处理呢？如下所示：

```
    Error : process (Sel_a,A,Sel_b,B)
    begin
            if (Sel_a = ‘ 1’ ) then
                    T <= A;
            else
                    T <= ‘ Z’ ;
            end if;
            if (Sel_b = ‘ 1’ ) then
                    T <= B;
            else
                    T <= ‘ Z’ ;
            end if;
    end process;
```

这是因为：上述两个if语句彼此没有优先级，又由于是对同一个信号（信号T）进行处理，则后一个处理会覆盖前一个处理。正确的做法是将这两个if 语句放在两个process中进行。

3.9 异步复位电路设计

异步复位信号的产生原则上应当遵守：

（1）异步复位信号无毛刺。

（2）一个信号不能既作为异步复位信号，又作为内部电路控制信号。

有关设计原理分析，可参见“同步电路设计技术及规则”。

当设计中存在两个或两个以上的异步复位信号时，例如软复位（一般由CPU下达）和硬复位，则在写代码时，经常会采用如下形式：

```
process （Mclk，CPU_reset，Hard_reset）
begin
    if ( CPU_reset = '1' or Hard_reset = '1') then
            D <= '0' ;
    elsif Mclk = '1' and Mclk'event then
            ......
    end if;
end process ;
```

上述形式在多个entity的多个process中出现，将导致以下几个问题：

（1）每个触发器都带一个“或”门复位电路，浪费资源；

（2）有些综合工具会出错。

正确地写法是：（可以采用）

```
Rst <= CPU_reset = '1' or Hard_reset = '1' ;
process （Mclk，CPU_reset，Hard_reset）
begin
    if ( Rst ='1' ) then
            D <= '0' ;
    elsif Mclk = '1' and Mclk'event then
            ......
    end if;
end process ;
```

其中所有类似的触发器都共用一个Rst。

在FPGA**设计里，可将**Rst**接在专用复位器件**startup**上，详情参见“**5.4.4**专用全局**Set/Reset**资源”**。

3.10时钟电路设计

许多人在设计电路时，经常采用“分频信号”作为时钟控制某一部分电路。在实际电路中，这种做法大多数是不可取的，这是因为：许多分频时钟不是全局时钟，也未经过时钟buffer**进行驱动，导致该分频时钟到各个触发器**clk**端的延时不一致，从而给许多触发器的建立时间或保持时间带来恶劣影响（原理参见“同步电路设计技术及规则”第**2**章“时序分析基础”），使设计不容易实现。**

因此，在同步电路设计里，一个原则是：凡是分频信号，一般均不作为时钟，而是作为控制信号。例如：

```
process （Mclk，Reset）
begin
    if Reset = '1' then
            Mclk_d2 <= '0' ;
    elsif Mclk = '1' and Mclk'event then
            Mclk_d2 <=  not Mclk_d2 ;
    end if;
end process ;
process（Mclk_d2，Reset）
begin
    if Reset = '1' then
            D_out <= '0' ;
    elsif Mclk_d2 = '1' and Mclk_d2'event then
            D_out <= D_in ;
    end if;
end process ;
```

上述电路可改为：

```
process（Mclk，Reset）
begin
    if Reset = '1' then
            D_out <= '0' ;
    elsif Mclk = '1' and Mclk'event then
            if Mclk_d2 = '0' then
                    D_out <= D_in ;
            end if ;
    end if;
end process ;
```

但是，对于如下电路，我们可以采用分频时钟。请注意：图中两个模块是无任何联系的。

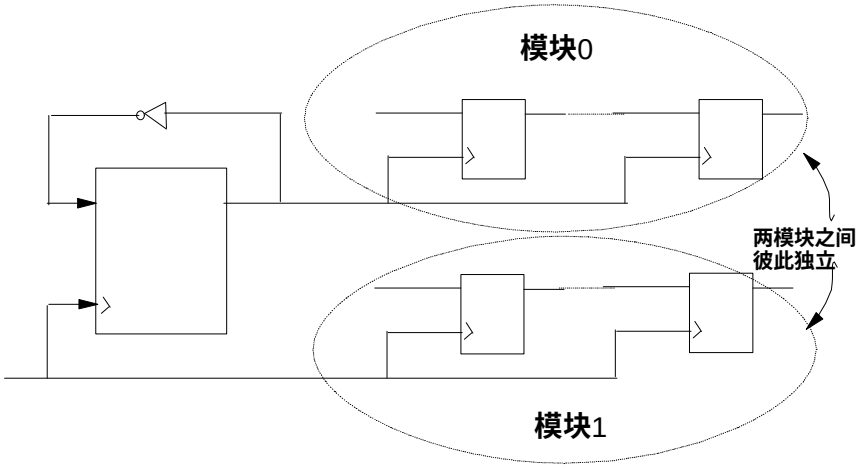


图4 分频时钟在独立模块中的应用

关于时钟电路的设计，我们总结如下（对可编程器件而言）：

（1）芯片外部时钟输入信号尽量从时钟管脚（PAD**内带全局时钟驱动**buffer**）引入，由于时钟管脚资源有限，这就要求我们在做设计时，应当了解某一器件最多能带多少个时钟，依次规范整个单板的设计（时钟考虑）。**

（2）一般而言，尽量少采用分频时钟，以减少时钟个数，简化设计。若要采用分频时钟，则尽量接在片内时钟buffer**上，而后再去驱动各触发器。**

（3）若是时钟太多，时钟buffer**资源不够，则应当对未经时钟**buffer**驱动的时钟加严格的约束，例如对**xilinx**器件而言，加**MAXDELAY**和**MAXSKEW。

4 设计技巧

在设计过程中，经常遇到速度或面积问题：在功能基本正确之后，设计要么速度不满足要求，要么面积太大，或者两者都不满足设计要求。经常在速度和面积上花费大量的时间。

本章着重从速度和面积角度出发，考虑如何编写代码或设计电路，以获得最佳的效果。但是，有些方法是以牺牲面积来换取速度，而有些方法是以牺牲速度来换取面积，也有些方法可同时获得速度和面积的好处。具体如何操作，应当依据实际情况而定。

速度和面积的处理，实际是对电路结构的处理，即如何获得最优的电路结构。可以在高抽象层次对电路的结构进行科学构造，如算法的优化，这样可以提升电路综合后的性能。综合工具并不总能导出电路的详细结构。如果设计者能提供附加的结构信息，能帮助综合工具生成更高效的电路

在处理速度与面积问题的一个原则是：向关键路径（部分）要时间，向非关键路径（部分）要面积。为了获得更高的速度，应当尽量减少关键路径上的逻辑级数；为了获得更小的面积，应当尽量共享已有的逻辑电路。

4.1 合理选择加法电路

4.1.1 串行进位与超前进位

改变赋值语句的顺序和使用信号或变量可以控制设计的结构。每一个VHDL信号赋值、进程或元件的引用对应着特定的逻辑。每个信号代表一条信号线。使用这些结构，能将不同的实体连接起来，实现不同的结构。下面的VHDL实例为加法器的进位链电路的两种可能的描述。

例、串行进位链

```
-- A is the addend
-- B is the augend
-- C is the carry
-- Cin is the carry in
C0 <= (A0 and B0) or
     ((A0 or B0) and Cin);
C1 <= (A1 and B1) or
        ((A1 or B1) and C0);
```

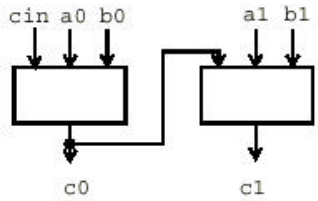


图8 串行进位

例、超前进位链（并行结构）

```
-- Ps are propagate
-- Gs are generate
p0 <= a0 or b0;
g0 <= a0 and b0;
p1 <= a1 or b1;
```

```
g1 <= a1 and b1;
c0 <= g0 or (p0 and cin);
c1 <= g1 or (p1 and g0) or
    (p1 and p0 and cin);
```

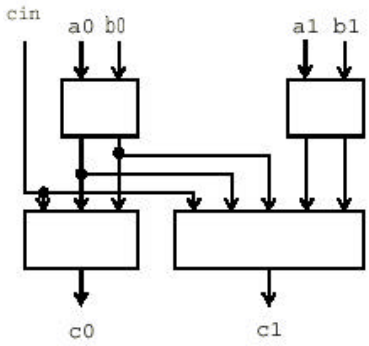


图9 超前进位

显然，第二种方法速度快，但面积大；第一种方法速度慢，但面积小。

4.1.2 使用圆括号处理多个加法器

控制设计结构的另一种方法是使用圆括号来定义逻辑分组。下面的例子描述了一个4输入的加法器分组及其实现结果。

例、 Z <= A + B + C + D;

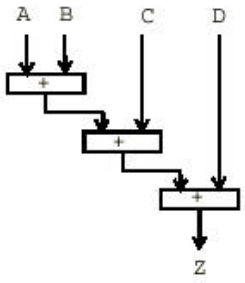


图10 串行加法电路

用圆括号重新构造的加法器分组如下所示。

例、 Z <= (A + B) + (C + D);

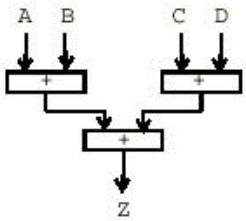


图11 并行加法电路

上述两种方法的在速度和面积上的区别是：

第一种方法（不带括号）：面积小，但整体速度慢。但是，如果信号D是关键路径，其它信号是非关键路径；或者，设计中关键路径与A、B、C、D无关，则应当采用这种方法。

第二种方法（带括号）：面积大，但整体速度快。如果对A、B、C、D的时序要求都比较苛刻，应当采用这种方法。

4.1.3 改变操作数的位宽

下面的例子，加法器求A和TEMP的和，A和TEMP都是8位宽。TEMP的值为B（当B<16时）或C（C为4位宽）。因此，TEMP的高4位总是0。但是综合器并不能利用此点，因为TEMP为BYTE类型。修改TEMP为NIBBLE类型，如第2个例子所示。这样，原来要用8个全加器完成的加法运算，现在低4位运算用4个全加器，高4位运算用4个半加器实现，大大节省了电路所占面积。

例1、原设计

```
function ADD_IT_16 (A, B: BYTE; C: NIBBLE) return BYTE is
 variable TEMP: BYTE;
begin
   if B < 16 then
         TEMP := B;
   else
      TEMP := C;
   end if;
   return A + TEMP;
end;
```

例2、修改操作数位宽

```
function ADD_IT_16 (A, B: BYTE; C: NIBBLE) return BYTE is
   variable TEMP: NIBBLE; -- Now only 4 bits
begin
   if B < 16 then
      TEMP := NIBBLE(B); -- Cast BYTE to NIBBLE
   else
      TEMP := C;
   end if;
   return A + TEMP; -- Single adder
end;
```

其中，BYTE是指8bits宽数据类型，BIBBLE是指4bits宽数据类型。

4.2 巧妙处理比较器

在我们的设计中，经常遇见比较两个值的大小关系。在许多情况下，参与比较的值不会超过某一特定值，例如：A和B都是5bit的计数器，但最大值是16，假如要做以下事情：

```
if  A >= B then
        做某一件事
else
        做另一件事
end if ;
```

上述电路用到了一个5bits的比较电路，比较复杂。事实上，该电路可以改写为：

```
if  ( A(4) = '1' )  or ( B(4) = '0' and (A(3 downto 0) >= B(3 downto 0) ) ) then
        做某一件事
else
        做另一件事
end if ;
```

显然，后一个电路只用到一个4bit的比较器，面积比前一个电路要简化很多，且速度也提高不少。

对比较器而言，若参与比较的两个信号的值是变化的，则比较器逻辑较大；反之，则会简化很多。我们应当尽量根据电路特点，抽取出“固定”部分（特征值），减少“变化”部分。

4.3 IF语句和Case语句：速度与面积的关系

IF语句指定了一个优先级编码逻辑，而Case语句生成的逻辑是并行的，不具有优先级。IF语句可以包含一套不同的表达式，而Case语句比较的是一个公共的控制表达式。通常，Case结构速度较快，但占用面积较大，所以用Case语句实现对速度要求较高的编解码电路。IF-Else结构速度较慢，但占用的面积小。如果对速度没有特殊要求，而对面积有较高要求，则可用IF-Else语句完成编解码。不正确的使用嵌套的IF语句会导致设计需要更大的延时。为了避免较大的路径延时，不要使用特别长的嵌套IF结构。用IF语句实现对延时要求苛刻的路径（speed-critical paths）时，应将最高优先级给最迟到达的关键信号（Critical Signal）。有时，为了兼顾面积和速度，可以将IF和Case语句合用。

例、七级嵌套IF结构

```
NESTED_IF: process (CLK)
begin
   if (CLK'event and CLK ='1') then
      if (RESET = '0') then
         if (ADDR_A = "00") then
            DEC_Q(5 downto 4) <=ADDR_D;
            DEC_Q(3 downto 2) <="01";
            DEC_Q(1 downto 0) <="00";
            if (ADDR_B = "01") then
               DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
               if (ADDR_C = "01") then
                  DEC_Q(5 downto 4) <=unsigned(ADDR_D) + '1';
                  if (ADDR_D = "11") then
                     DEC_Q(5 downto 4) <= "00";
                  end if;
               else
                  DEC_Q(5 downto 4) <= ADDR_D;
               end if;
            end if;
         else
            DEC_Q(5 downto 4) <= ADDR_D;
            DEC_Q(3 downto 2) <= ADDR_A;
            DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
         end if;
      else
         DEC_Q <= "000000";
      end if;
   end if;
end process;
```

7 Levels of Indentation

例、采用IF-Case的嵌套IF结构

```
IF_CASE: process (CLK)
begin
   if (CLK'event and CLK = '1') then
      if (RESET = '0') then
         case ADDR_ALL is
            when "00011011" =>
               DEC_Q(5 downto 4) <= "00";
               DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
            when "000110--" =>
               DEC_Q(5 downto 4) <= unsigned(ADDR_D) + '1';
               DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
            when "0001----" =>
               DEC_Q(5 downto 4) <= ADDR_D;
               DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
            when "00------" =>
               DEC_Q(5 downto 4) <= ADDR_D;
               DEC_Q(3 downto 2) <= "01";
               DEC_Q(1 downto 0) <= "00";
            when other =>
               DEC_Q(5 downto 4) <= ADDR_D;
               DEC_Q(3 downto 2) <= ADDR_A;
               DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
            end case;
         else
            DEC_Q <= "000000";
         end if;
      end if;
   end process;
```

5 Levels of Indentation

由上面两个例子可以看出，采用IF-Case结构比嵌套IF结构更有效，减少了大约3ns的延时（使用XC4005E-2）。

IF-Then-Else**适于完成优先级编码，此时应将最高优先级给关键信号**（Critical Signal），**在下面的例中关键信号为**in[0]。

例、用IF-Then-Else**完成**8**选**1**多路选择器**

```
MUX6to1:process(sel,in)
begin
    if(sel= "000") then
          out <= in(0);
    elseif(sel = "001") then
          out <= in(1);
    elseif(sel = "010") then
       out <= in(2);
    elseif(sel = "011") then
       out <= in(3);
    elseif(sel = "100") then
       out <= in(4);
    else
          out <= in(5);
    end if;
end process;
```

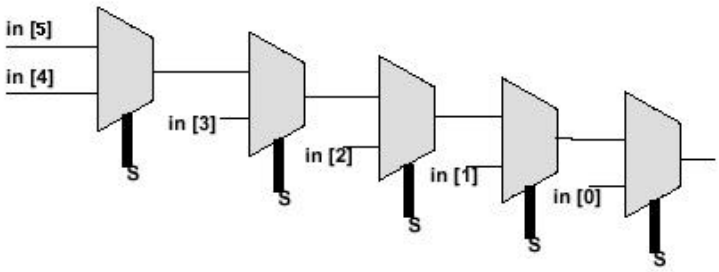


图12 if-else**完成多路选择**

下面的例子是用Case**语句完成**8**选**1**多路选择器的**VHDL**实例。在大多数**FPGA**结构中能够在单个**CLB**中完成一个**4**选**1**的多路选择器，**Virtex**可以在单个**CLB**中完成一个**8**选**1**的多路选择器。而用**IF-Else**语句需要多个**CLB**才能完成相同功能。因此，**Case**语句生成的设计速度更快延时更小。**

例、用 Case**实现**8**选**1**多路选择器**

```
MUX8to1：process( C, D, E, F, G, H, I, J, S )
begin
    case S is
        when “000” => Z <= C;
        when “001” => Z <= D;
        when “010” => Z <= E;
        when “011” => Z <= F;
        when “100” => Z <= G;
        when “101” => Z <= H;
        when “110” => Z <= I;
```

```
            when others => Z <= J;
        end case;
    end process;
```

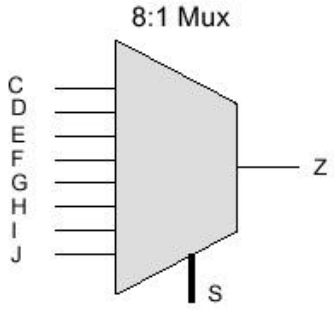


图13 case**语句完成电路选择**

4.4 减少关键路径的逻辑级数

在FPGA**中，关键路径**（critical path）**上的每一级逻辑都会增加延时。为了保证能满足定时约束，就必须在对设计的行为进行描述时考虑逻辑的级数。减少关键路径延时的常用方法是给最迟到达的信号最高的优先级，这样能减少关键路径的逻辑级数。下面的实例描述了如何减少关键路径上的逻辑级数。**

例、此例中critical**信号经过了**2**级逻辑**

```
if (clk'event and clk ='1') then
    if (non_critical='1' and critical='1') then
              out1 <= in1;
    else
              out1 <= in2;
    end if;
end if;
```

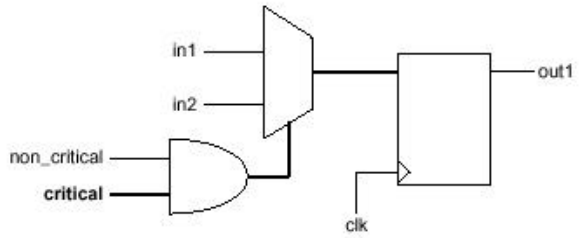


图14 critical**信号经过**2**级逻辑**

为了减少critical**路径的逻辑级数，将电路修改如下，**critical**信号只经过了一级逻辑。**

```
signal  out_temp : std_logic;
process (non_critical, in1, in2)
      if (non_critical='1') then
            out_temp <= in1;
   else
        out_temp <= in2;
  end if;
end process;
```

```
process(clk)
  if (clk'event and clk ='1') then
       if (critical='1') then
            out1 <= out_temp;
    else
            out1 <= in2;
       end if;
  end if;
end process;
```

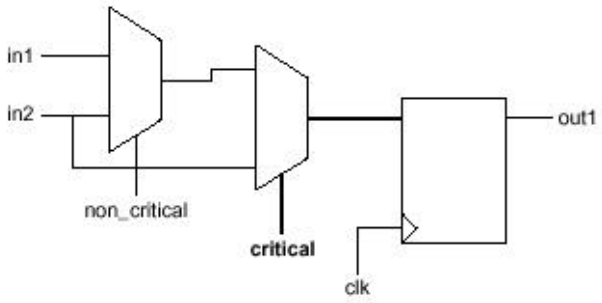


图15 critical**信号只经过一级逻辑**

4.5 资源共享

4.5.1 if**语句**

资源共享能够减少HDL**设计所用逻辑模块的数量。否则，每个**HDL**描述都要建立一套独立的电路。下面的**VHDL**实例说明如何使用资源共享来减少逻辑模块的数量。**

例、没有资源共享时用了4**个加法器完成**

```
if (...(siz = "0001")...) then
    count <= count + "0001";
else if (...((siz = "0010")...) then
    count <= count + "0010";
else if (...(siz = "0011")...) then
    count <= count + "0011";
else if (...(siz == "0000")...)then
    count <= count + "0100";
end if;
```

利用资源共享，采用下面的代码，可以节省2**个加法器**

```
if (...(siz = "0000")...) then
    count <= count + "0100";
else if (...) then
    count <= count + siz;
end if;
```

例、没有利用资源共享时用了2**个加法器实现**

```
if (select = '1') then
  sum<=A +B;
else
  sum<=C +D;
end if;
```

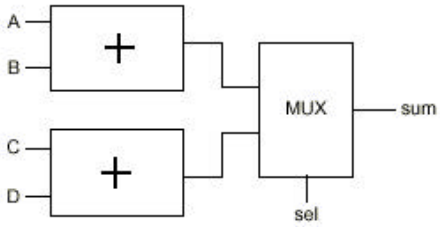


图16 资源共享前，2个加法器

加法器要占用宝贵的资源。利用资源共享，修改代码如下，只用2个选择器和1个加法器实现，减少了资源占用。

```
if (sel ='1') then
   temp1 <=A;
   temp2 <=B;
else
   temp1 <=C;
   temp2 <=D;
end if;
sum <= temp1 + temp2;
```

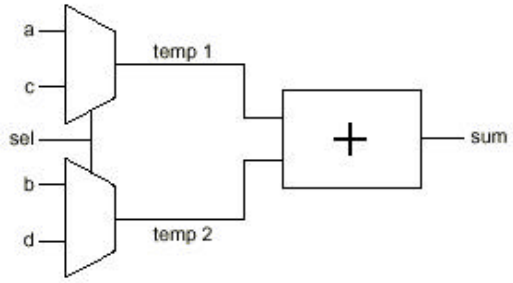


图17 资源共享后，1个加法器

4.5.2 loop语句

与选择器相比，运算符占用更多的资源。如果在循环语句中有一个运算符，综合工具必须对所有的条件求值。下面的VHDL实例，综合工具用4个加法器和一个选择器实现。 只有当“req”信号为关键信号时，才建议采用这种方法。

```
for i in 0 to 3 loop
  if (req(i)='1') then
     sum <= vsum + offset(i);
  end if;
end loop;
```

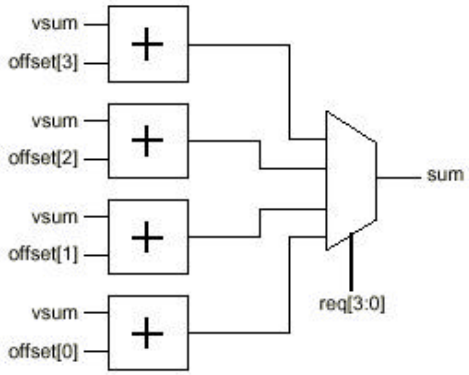


图18 资源共享前4个加法器

如果“req”信号不是关键信号，运算符应当移到循环语句的外部。这样在执行加法运算前，综合工具可以对数据信号进行选择。修改代码如下，用一个多路选择器和一个加法器即可实现。

```
for i in 0 to 3 loop
   if (req(i)='1') then
       offset_1 <= offset(i);
   end if;
end loop;
sum <= vsum + offset_1;
```

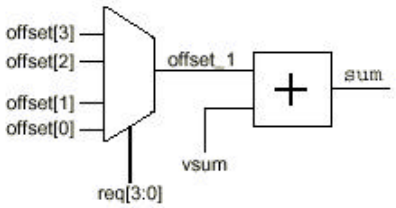


图19 资源共享后一个加法器

4.5.3 子表达式共享

一个表达式中子表达式包含2个或更多的变量。如果相同的子表达式在多个等式中出现，应共享这些运算，减少电路的面积。通过声明一个临时变量存储子表达式，然后在任何需要重复使用这个子表达式的地方用临时变量代替。

下面的VHDL实例描述了用相同的子表达式（a+b）完成一组简单的加法运算。

```
temp <= a + b;
x <= temp;
y <= temp + c;
```

但是，在某些特定情况下，资源共享不仅不能降低面积（以占用基本资源的数量来划分，如占用CLB的数量），而且还会增加延时。详情参见5.4.1相关部分。

4.5.4 综合工具与资源共享

通过设置FPGA CompilerII/FPGA Express的相应选项（缺省为共享），可以让综合工具自动决定是否要共享相同的子表达式，而不需声明一个临时变量存储子表达式。

但是综合工具对共享的代码编写规则有限制，如下：

1. 可共享的操作符为 * + - > < >= <=

2. 子表达式具有相同position

例如：

```
sum1 <= A + B + C;
sum2 <= D + A +B;
sum3 <= E + (A +B);
```

则sum1和sum3 可共享(A +B),但与sum2不共享，原因是表达式的计算是从左到右。

3. 必须在同一block （ process）中，并且是在同一条件下控制，如下：

```
if (cond1 = ..) then
        S1 <=  A +B ;
else
        if (cond2 ....) then
                S2  <=  E + F ;
        else
                S3 <= G+ H ;
        end if ;
end if ;
```

则 S2 和S3可共享一个加法器，但与S1不可共享。

为了使代码更加具备通用性，最好尽量自行编写共享资源代码。

4.6 流水线（Pipelining）

流水线能动态的提升器件性能，它的基本思想是：对经过多级逻辑的长数据通路进行重新构造，把原来必须在一个时钟周期内完成的操作分成多个周期完成。这种方法允许更高的工作频率，因此提高了数据吞吐量。因为FPGA的寄存器资源非常丰富，所以对FPGA设计而言，流水线通常是一种先进的结构，而又不耗费过多的器件资源。然而，采用流水线后，数据通道变成多时钟周期，必须特别考虑设计的其余部分，解决增加通路带来的延迟。在定义这些路径的延时约束时必须特别小心。

当一个设计的寄存器之间存在多级逻辑时，其延时为源触发器的clock-to-out时间、多级逻辑的延时、多级逻辑的走线延时和目的寄存器的建立时间之和。工作时钟频率的提高受到这个延时的限制。采用流水线，减少了寄存器间的逻辑的级数。最终的结果是系统的工作频率提高了。

例、采用流水线前的电路

```
process(clk, a, b, c) begin
   if(clk'event and clk = '1') then
        a_temp <= a;
        b_temp <= b;
        c_temp <= c;
   end if;
```

```
end process;

Process(clk, a_temp, b_temp, c_temp)
begin
   if(clk'event and clk = '1') then
            out <= (a_temp * b_temp) + c_temp;
   end if;
end process;
```

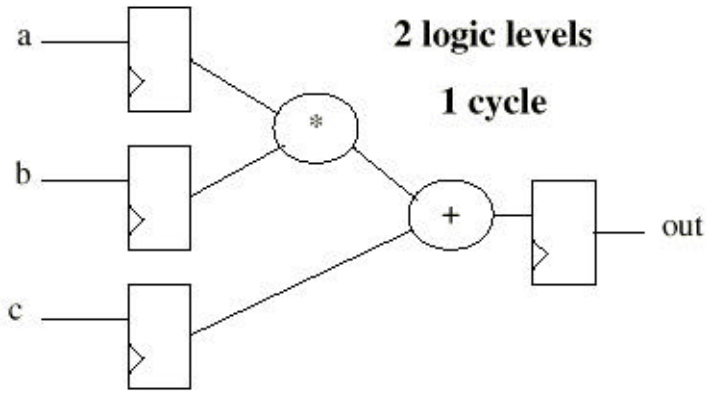


图5 采用流水线之前电路结构

例、采用流水线后的电路

```
process(clk, a, b, c) begin
   if(clk'event and clk = '1') then
            a_temp  <= a;
            b_temp  <= b;
            c_temp1 <= c;
   end if;
end process;

process(clk, a_temp, b_temp, c_temp1)
begin
   if(clk'event and clk = '1') then
            mult_temp <= a_temp * b_temp
            c_temp2 <= c_temp1;
   end if;
end process;

process(clk, mult_temp, c_temp2)
begin
```

```
    if(clk'event and clk = '1') then
            out <= mult_temp + c_temp2;
    end if;
end process;
```

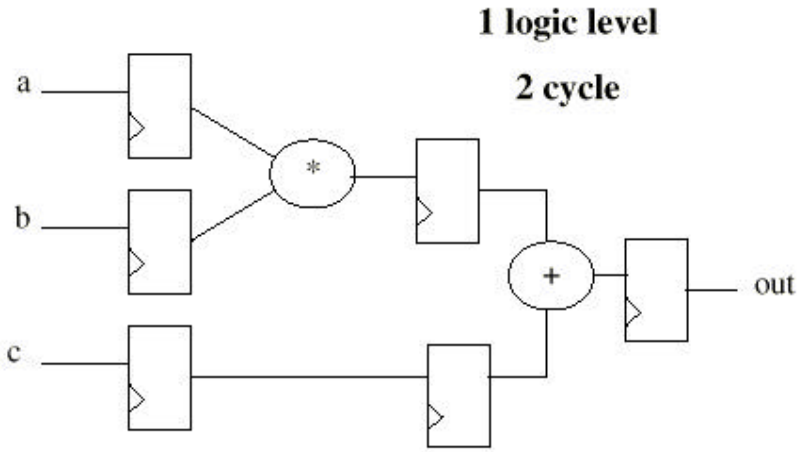


图20 采用流水线之后的电路结构

4.7 组合逻辑和时序逻辑分离

包含寄存器的同步存储电路和异步组合逻辑应分别在独立的进程中完成，组合逻辑中关联性强的信号应放在一个进程中，这样在综合后面积和速度指标较高。这种方法常用在设计Mealy状态机中。Mealy状态机的基本结构如下图所示。

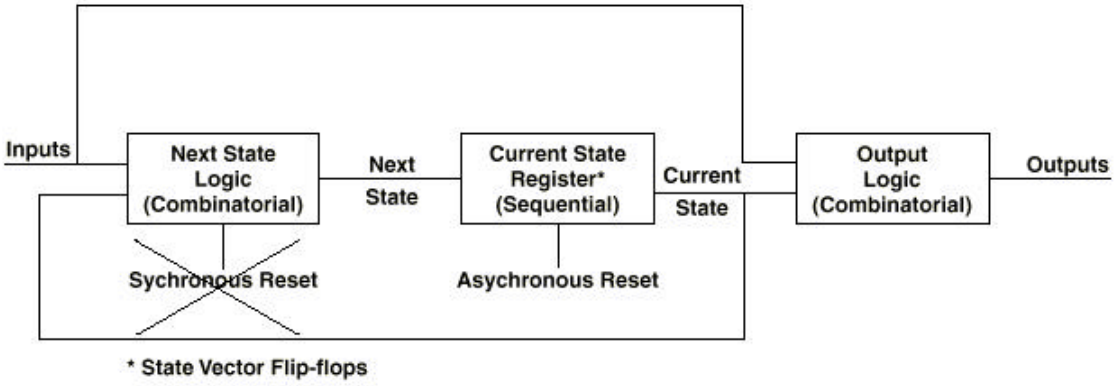


图7 Mealy状态机的基本结构

由图可看出，Mealy状态机由次状态逻辑、当前状态寄存器和输出逻辑三部分组成，其中次状态逻辑和输出逻辑为组合逻辑，当前状态寄存器为时序逻辑。因此，Mealy机可由三个进程实现，如下面VHDL实例。

例、5个状态的Mealy状态机

```
-- Example of a 5-state Mealy FSM
library ieee;
use ieee.std_logic_1164.all;
```

```
entity mealy is
  port (clock, reset: in std_logic;
    data_out: out std_logic;
    data_in: in std_logic_vector (1 downto 0));
end mealy;

architecture behave of mealy is
  type state_values is (st0, st1, st2, st3, st4);
  signal pres_state, next_state: state_values;
begin
  -- FSM register
  statereg: process (clock, reset)
  begin
    if (reset = '0') then
      pres_state <= st0;
    elsif (clock'event and clock ='1') then
      pres_state <= next_state;
    end if;
  end process statereg;

  -- FSM combinational block
  fsm: process (pres_state, data_in)
  begin
      case pres_state is
        when st0 =>
              case data_in is
                when "00" => next_state <= st0;
                when "01" => next_state <= st4;
                when "10" => next_state <= st1;
                when "11" => next_state <= st2;
                when others => next_state <= (others <= 'x');
              end case;
            when st1 =>
              case data_in is
                when "00" => next_state <= st0;
                when "10" => next_state <= st2;
                when others => next_state <= st1;
              end case;
            when st2 =>
              case data_in is
                when "00" => next_state <= st1;
                when "01" => next_state <= st1;
                when "10" => next_state <= st3;
                when "11" => next_state <= st3;
                when others => next_state <= (others <= 'x');
              end case;
            when st3 =>
              case data_in is
```

```
                when "01" => next_state <= st4;
                when "11" => next_state <= st4;
                when others => next_state <= st3;
              end case;
            when st4 =>
              case data_in is
                when "11" => next_state <= st4;
                when others => next_state <= st0;
              end case;
            when others => next_state <= st0;
        end case;
    end process fsm;

    -- Mealy output definition using pres_state w/ data_in
    outputs: process (pres_state, data_in)
    begin
        case pres_state is
          when st0 =>
            case data_in is
              when "00" => data_out <= '0';
              when others => data_out <= '1';
            end case;
          when st1 => data_out <= '0';
          when st2 =>
            case data_in is
              when "00" => data_out <= '0';
              when "01" => data_out <= '0';
              when others => data_out <= '1';
            end case;
          when st3 => data_out <= '1';
          when st4 =>
            case data_in is
              when "10" => data_out <= '1';
              when "11" => data_out <= '1';
              when others => data_out <= '0';
            end case;
          when others => data_out <= '0';
        end case;
    end process outputs;
end behave;
```

4.8 利用电路的等价性，巧妙地“分配”延时

在功能等价的情况下，我们可以根据时序需要，安排组合逻辑电路在寄存器前后的位置，合理分配延时。

例、组合逻辑在寄存器之后

如下图所示，假定a、b**信号的延时非常大，则**

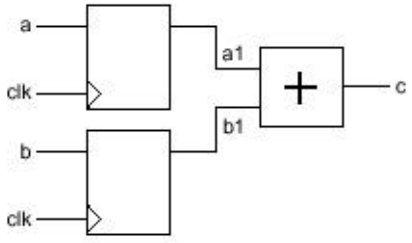


图8 组合逻辑（加法器）在后

```
process (clk, a, b) begin
  if (clk'event and clk = '1') then
      a1 <= a;
      b1 <=b;
  end if;
end process;

process (a1, b1) begin
      c <=a1 +b1;
end process;
```

例、组合逻辑放在寄存器之前

如果a、b信号的延时并不大，而寄存器c信号经过的逻辑比较多，延时大，则

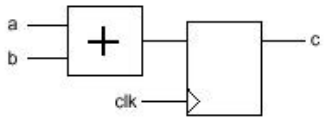


图22 组合逻辑（加法器）在前

```
process (clk, a, b) begin
    if (clk'event and clk = '1') then
      c <=a +b;
    end if;
end process;
```

这种处理方法的实质是：将关键路径中的部分延时“挪到”其它非关键路径上。

4.9 复制电路，减少扇出（fanout），提高设计速度

提高关键路径速度的一个常用方法是复制寄存器，减少关键路径的扇出。因为FPGA的寄存器资源丰富，所以使用寄存器不会占用过多的资源。当一个信号网络所带负载增加时，其路径延时也相应增加。这对复位信号网络可能影响不大，但对象三态使能信号是不能容忍的。扇出对关键路径延时的影响甚至超过了逻辑的级延时，确保一个网络的扇出少于一定值（例如16，表示某个信号所驱动的基本器件不超过16个）是很重要的。下面的VHDL实例中，信号“Tri_en”的扇出为24。

例、寄存器扇出为24

```
architecture load of four_load is
```

```
  signal Tri_en std_logic;
  begin
   loadpro: process (Clk)
     begin
       if (clk'event and clk ='1') then
            Tri_end <= Tri_en;
       end if;
  end process loadpro;

   endpro : process (Tri_end, Data_in)
     begin
       if (Tri_end = '1') then
            out <= Data_in;
       else
            out <= (others => 'Z');
       end if;
  end process endpro;
end load;
```

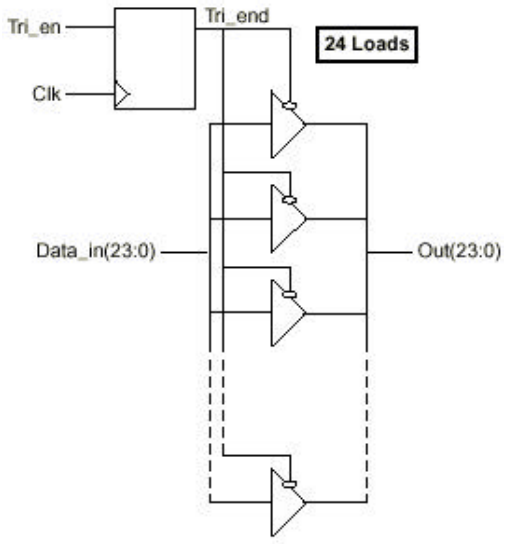


图23 扇出较大

为了将扇出降低一半，增加一个寄存器使负载分成两部分，每个寄存器扇出为12。

注意：有些综合工具自动复制寄存器解决定时和扇出问题，不需要在编码中复制寄存器。

例、复制寄存器后寄存器扇出为12

```
architecture loada of two_load is
  signal Tri_en1, Tri_en2 : std_logic;
begin
  loadpro: process (Clk)
    begin
     if (clk'event and clk ='1') then
          Tri_en1 <= Tri_en;
          Tri_en2 <= Tri_en;
```

```
        end if;
    end process loadpro;

    process (Tri_en1, Data_in)
        begin
         if (Tri_en1 = '1') then
             out(23:12) <= Data_in(23:12);
         else
             out(23:12) <= (others => 'Z');
         end if;
    end process;

    process (Tri_en2, Data_in)
        begin
         if (Tri_en2 = '1') then
             out(11:0) <= Data_in(11:0);
         else
             out(11:0) <= (others => 'Z');
         end if;
    end process;
end loada;
```

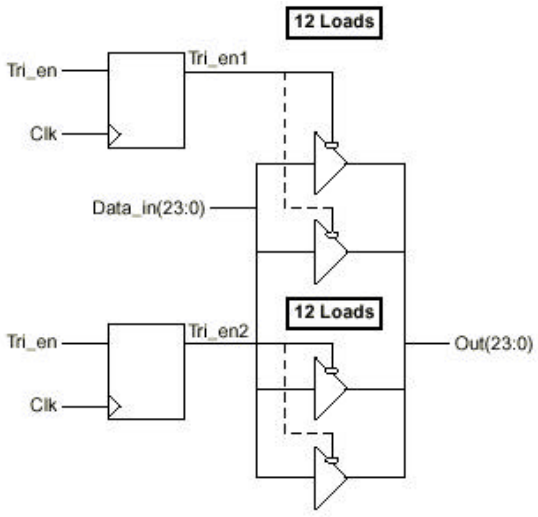


图24 扇出较小

类似地，我们还可以复制组合逻辑电路、网络上插入buffer等手段，减少扇出，提高速度。

5 与工艺结构相关的设计技巧（以Xilinx为例）

5.1 Xilinx器件结构特点

Xilinx的FPGA具有这样的结构特点：器件内部为可编程逻辑块（CLB），CLB实现FPGA的大部分逻辑，器件内部强大而灵活的多层布线资源将CLB互相连接起来，CLB外面由输入/输出块

（IOB）包围，IOB提供外部引脚和内部逻辑的接口，IOB和CLB之间有丰富的布线资源，可以实现最复杂的互连模式。

4000系列，每个CLB主要由两个寄存器和两个LUT组成，另外再加上CLB内连线资源和其它逻辑组成。其中，LUT即可做组合逻辑，又可做分布式RAM。做组合逻辑时，每个LUT可以形成任意4输入的组合逻辑；做RAM时，可以构成16X1的单端口RAM。 Virtex和Virtex-E系列，CLB是由原来的两个CLB组成，原CLB成为现CLB中的一个slice。

4000系列和Virtex系列，每个IOB主要由一个输入寄存器、一个输出寄存器、一个三态门、一个三态控制寄存器构成，另外还有内部连线资源。因此，在设计电路时，应当充分利用IOB资源，以减少CLB的占用量。

采用三态控制寄存器的好处是：可以大大降低三态控制信号的线延时，使得双向口的输出更快，增加单板设计裕量，例如访问外部RAM等。

有关CLB和IOB的详细结构，可以参见Xilinx相关参考资料。

5.2 状态机编码及设计技巧

5.2.1 状态机编码

状态机是系统设计中最常用到的功能之一。状态机传统的设计方法要经过状态分配、绘制状态表、化简次态方程等繁琐的步骤。这样设计出来的电路通常需要少量的触发器和宽输入的组合逻辑函数，因此较适合PAL和门阵列结构。FPGA拥有丰富的寄存器资源结构和包含在CLB中的窄输入函数产生器。对FPGA来说，因为传统的编码方法需要许多组合逻辑对状态进行译码，所以会在状态寄存器之间生成多级逻辑，将导致实现后的电路的速度和面积指标较差。

在FPGA中设计状态机常用的三种编码方法是：二进制、枚举类型和one-hot。

二进制编码和枚举类型编码需用大量的反馈逻辑进行状态译码，完成当前状态向下一个状态的跳变。与one-hot相比，使用较少的寄存器。因此对CPLD如XC9500较适合。

一位有效（one-hot）编码技术使用n位状态寄存器表示n个状态的状态机，每个状态都有它独立的寄存器位，并且在任何时刻其中只有一位有效（为1），该位也称为“热点（hot）”。对现态的译码只需要简单找出值为1的位即可，而状态的转移则是将其中的1位由0变1，另1位由1变0。在FPGA中，该方式明显缩减实现设计所需的逻辑资源，并且其简单的次态方程减少了状态寄存器之间的逻辑级数，因此提高了运行速度。FPGA的寄存器资源丰富，而连接到每个寄存器的组合电路较少，使用one-hot方式比较合适。利用HDL属性、命令行参数或图形界面的选项等多种方式指定综合参数，许多综合工具能够对状态机设计选用one-hot编码。

5.2.2 设计技巧

实现基于FPGA的大型状态机通常用one-hot编码。对规模较小的状态机（少于8个状态），二进制编码效率更高。由于大部分FPGA的CLB中有4输入LUT。所以，对one-hot编码方式，限制输入状态在4以内，能获得最快的速度。

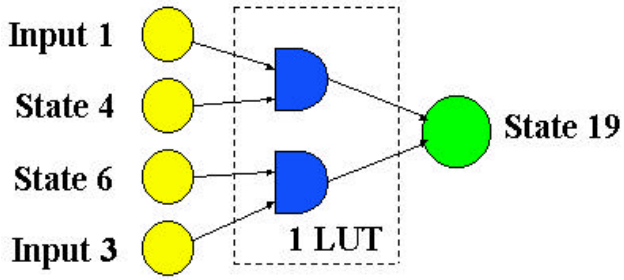


图28 4输入

为了提高设计的性能，可以将大状态机（大于32个状态）拆分成几个小状态机，每个状态机使用不同的编码方式。此外，对复杂状态应进行划分，将复杂状态划分成简单的状态传统能提高电路的性能。

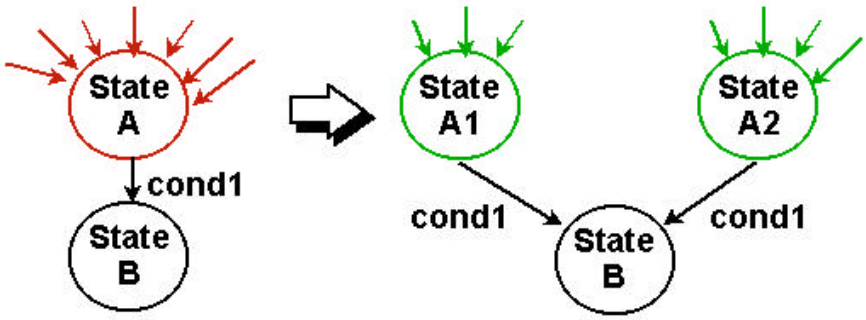


图26 分解大状态机

为了缩短状态机输出的延时，保证输出无毛刺，应增加输出寄存器，如下图所示，状态机输出在每个时钟沿打出，保证了状态机的输出无毛刺，并满足时序要求。

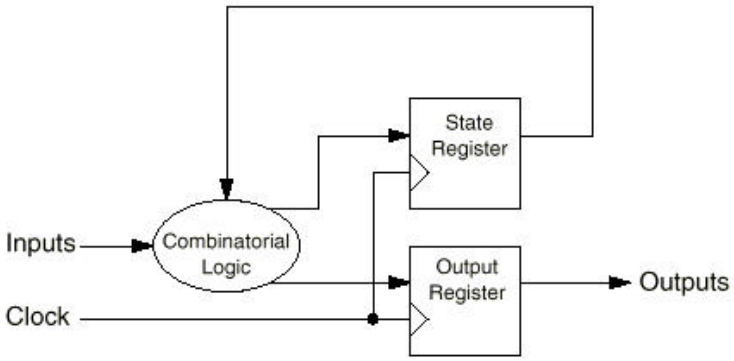


图27 输出加寄存器

5.2.3 状态机的容错性

状态机的容错（Fault Tolerance）性是指当状态机进入非法状态时是否能自动转移到正常状态。系统的容错性是以逻辑资源为代价的。对非法状态的处理可以采用下面的语句强迫转移到正常状态。

```
case present_state is
    。 。 。
```

when others =>next_state <= **指定某一固定值**;

end case;

但是，对非法状态的判断将占用额外的逻辑资源。对于采用one-hot**编码的状态机，由于存在大量未定义状态，进入非法状态的可能性显著增加。例如，在**one-hot**方式下，对**n**位状态向量，考虑出错的可能性，可能会存在**2**的**n**次方种状态值组合，但实际状态只有**n**个。于是会产生这样的矛盾：采用**one-hot**方式是为了有效实现设计，但如果定义所有由非法状态向合法状态或复位转移将导致低效的实现结果。对一个具有**18**个状态的例子，完整定义其数目庞大的**262126**个转移是不可能的。因此对非法状态可以忽略不做处理。方法如下：**

when others => next_state <= "-----";

但是，虽然无法实现脱离非法状态的转移，但可以增加一个对有效位是否大于1**的判断逻辑，生成一个冲突信号提供给系统。**

对于状态机的容错性，必须慎重考虑，特别是对一位有效方式编码的设计，容错性和资源占用、系统性能互相矛盾，因此必须综合考虑，权衡利弊。

5.2.4 三种编码方式的VHDL实例

下图描述了一个具有7**个状态的状态机，后面分别给出了三种编码方式的**VHDL**实例。**

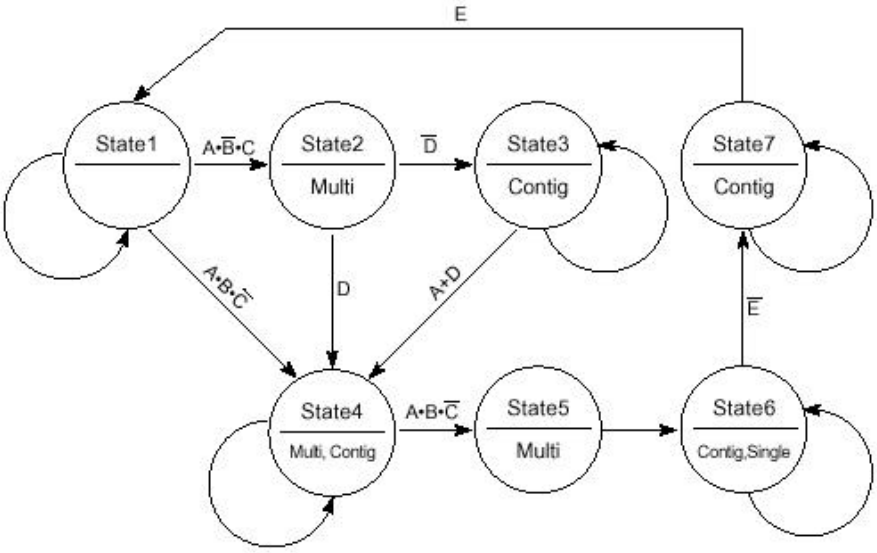


图28 **状态机实例**

例、二进制编码方式

```
-------------------------------------------------
-- BINARY.VHD Version 1.0 --
-- Example of a binary encoded state machine --
-- May 1997 --
-------------------------------------------------
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
   entity binary is
       port (CLOCK, RESET : in STD_LOGIC;
         A, B, C, D, E: in BOOLEAN;
         SINGLE, MULTI, CONTIG: out STD_LOGIC);
   end binary;

   architecture BEHV of binary is
       type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
       attribute ENUM_ENCODING: STRING;
        attribute ENUM_ENCODING of STATE_TYPE:type is "001 010 011 100
101 110 111";
       signal CS, NS: STATE_TYPE;
       begin
     SYNC_PROC: process (CLOCK, RESET)
     begin
          if (RESET='1')then
                CS <= S1;
          elsif (CLOCK'event and CLOCK = '1')then
                CS <= NS;
                end if;
          end process; --End REG_PROC

          COMB_PROC: process (CS, A, B, C, D, E)
          begin
                case CS is
                      when S1 =>
                        MULTI  <= '0';
                        CONTIG <= '0';
                        SINGLE <= '0';
                        if (A and not B and C) then
                             NS <= S2;
                        elsif (A and B and not C) then
                             NS <= S4;
                        else
                             NS <= S1;
                        end if;
                      when S2 =>
                        MULTI  <= '1';
                        CONTIG <= '0';
                        SINGLE <= '0';
                        if (not D) then
                             NS <= S3;
                        else
```

```
                    NS <= S4;
                  end if;
                when S3 =>
                  MULTI  <= '0';
                  CONTIG <= '1';
                  SINGLE <= '0';
                  if (A or D) then
                       NS <= S4;
                  else
                       NS <= S3;
                  end if;
                when S4 =>
                  MULTI  <= '1';
                  CONTIG <= '1';
                  SINGLE <= '0';
                  if (A and B and not C) then
                       NS <= S5;
                  else
                       NS <= S4;
                  end if;
                when S5 =>
                  MULTI  <= '1';
                  CONTIG <= '0';
                  SINGLE <= '0';
                  NS <= S6;
                when S6 =>
                  MULTI  <= '0';
                  CONTIG <= '1';
                  SINGLE <= '1';
                  if (not E) then
                       NS <= S7;
                  else
                       NS <= S6;
                  end if;
                when S7 =>
                  MULTI  <= '0';
                  CONTIG <= '1';
                  SINGLE <= '0';
                  if (E) then
                       NS <= S1;
                  else
                       NS <= S7;
                  end if;
            end case;
        end process; -- End COMB_PROC
    end BEHV;
```

例、枚举类型编码方式

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity enum is
    port (CLOCK, RESET : in STD_LOGIC;
         A, B, C, D, E: in BOOLEAN;
         SINGLE, MULTI, CONTIG: out STD_LOGIC);
end enum;

architecture BEHV of enum is
    type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
    signal CS, NS: STATE_TYPE;
    begin
     SYNC_PROC: process (CLOCK, RESET)
     begin
           if (RESET= '1') then
                 CS <= S1;
           elsif (CLOCK'vent and CLOCK = '1') then
                 CS <= NS;
           end if;
     end process; --End SYNC_PROC

     COMB_PROC: process (CS, A, B, C, D, E)
     begin
           case CS is
               when S1 =>
                  MULTI  <= '0';
                  CONTIG <= '0';
                  SINGLE <= '0';
     .
     .
     .
```

例、one-hot**编码方式**

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

entity one_hot is
    port (CLOCK, RESET : in STD_LOGIC;
         A, B, C, D, E: in BOOLEAN;
         SINGLE, MULTI, CONTIG: out STD_LOGIC);
end one_hot;

architecture BEHV of one_hot is
    type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
    attribute ENUM_ENCODING: STRING;
    attribute ENUM_ENCODING of STATE_TYPE: type is "0000001 0000010 0000100
    0001000 0010000 0100000 1000000 ";
```

```
signal CS, NS: STATE_TYPE;

begin
 SYNC_PROC: process (CLOCK, RESET)
 begin
        if (RESET='1') then
               CS <= S1;
        elsif (CLOCK'event and CLOCK ='1') then
               CS <= NS;
        end if;
 end process; --End SYNC_PROC

 COMB_PROC: process (CS, A, B, C, D, E)
 begin
        case CS is
            when S1 =>
               MULTI  <= '0';
               CONTIG <= '0';
               SINGLE <= '0';
               if (A and not B and C) then
                      NS <= S2;
               elsif (A and B and not C) then
                      NS <= S4;
               else
                      NS <= S1;
               end if;
 .
 .
 .
```

5.3 桶形移位器（Barrel Shifter）的两种实现方式

在这个例子中，采用两级多路选择器提高16**位**Barrel shifter**的速度**。

下面的VHDL**实例描述了用**16**个**16**选**1**的多路选择器完成**16**位**Barrel shifter。**每一个**16**选**1**的多路选择器是一个包含**16**位数据信号和**4**为选择信号的共**20**个输入的函数。当设计映射进基于**4**输入查找表结构的**FPGA**器件（如**XC4000**系列）时，一个**20**输入函数最少需要**5**个**CLB。**因此，设计的最小尺寸为**80**个**CLB（16×5）。

例、用16**个**16**选**1**的多路选择器实现**16**位**Barrel shifter

```
-------------------------------------------------
-- VHDL Model for a 16-bit Barrel Shifter --
-- barrel_org.vhd --
-- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! --
-- THIS EXAMPLE IS FOR COMPARISON ONLY --
-- May 1997 --
-- USE barrel.vhd --
-------------------------------------------------
```

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

entity barrel_org is
   port (S:in STD_LOGIC_VECTOR (3 downto 0);
       A_P:in STD_LOGIC_VECTOR (15 downto 0);
       B_P:out STD_LOGIC_VECTOR (15 downto 0));
end barrel_org;

architecture RTL of barrel_org is
begin
  SHIFT: process (S, A_P)
  begin
       case S is
         when "0000" =>
            B_P <= A_P;
         when "0001" =>
            B_P(14 downto 0) <= A_P(15 downto 1);
            B_P(15) <= A_P(0);
         when "0010" =>
            B_P(13 downto 0) <= A_P(15 downto 2);
            B_P(15 downto 14) <= A_P(1 downto 0);
         when "0011" =>
            B_P(12 downto 0) <= A_P(15 downto 3);
            B_P(15 downto 13) <= A_P(2 downto 0);
         when "0100" =>
            B_P(11 downto 0) <= A_P(15 downto 4);
            B_P(15 downto 12) <= A_P(3 downto 0);
         when "0101" =>
            B_P(10 downto 0) <= A_P(15 downto 5);
            B_P(15 downto 11) <= A_P(4 downto 0);
         when "0110" =>
            B_P(9 downto 0) <= A_P(15 downto 6);
            B_P(15 downto 10) <= A_P(5 downto 0);
         when "0111" =>
            B_P(8 downto 0) <= A_P(15 downto 7);
            B_P(15 downto 9) <= A_P(6 downto 0);
         when "1000" =>
            B_P(7 downto 0) <= A_P(15 downto 8);
            B_P(15 downto 8) <= A_P(7 downto 0);
         when "1001" =>
            B_P(6 downto 0) <= A_P(15 downto 9);
            B_P(15 downto 7) <= A_P(8 downto 0);
         when "1010" =>
            B_P(5 downto 0) <= A_P(15 downto 10);
            B_P(15 downto 6) <= A_P(9 downto 0);
         when "1011" =>
```

```
            B_P(4 downto 0) <= A_P(15 downto 11);
            B_P(15 downto 5) <= A_P(10 downto 0);
          when "1100" =>
            B_P(3 downto 0) <= A_P(15 downto 12);
            B_P(15 downto 4) <= A_P(11 downto 0);
       when "1101" =>
            B_P(2 downto 0) <= A_P(15 downto 13);
            B_P(15 downto 3) <= A_P(12 downto 0);
          when "1110" =>
            B_P(1 downto 0) <= A_P(15 downto 14);
            B_P(15 downto 2) <= A_P(13 downto 0);
          when "1111" =>
            B_P(0) <= A_P(15);
            B_P(15 downto 1) <= A_P(14 downto 0);
          when others =>
            B_P <= A_P;
        end case;
  end process; -- End SHIFT
end RTL;
```

下例的VHDL**设计使用两级多路选择器实现**16**位**Barrel Shifter**。它采用**32**个**4**选**1**多路选择器分两级实现，每一级**16**个。第一级将数据旋转**0、1、2**或**3**位，第二级将数据旋转**0、4、8**或**12**位。因为可以在单个**CLB**中实现**4**选**1**多路选择器，所以设计的最小尺寸为**32**个**CLB（32×1）。

例、用32**个**4**选**1**多路选择器分两级实现**16**位**Barrel shifter

```
-- BARREL.VHD
-- Based on XAPP 26 (see http://www.xilinx.com)
-- 16-bit barrel shifter (shift right)
-- May 1997

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

entity barrel is
    port (S: in STD_LOGIC_VECTOR(3 downto 0);
     A_P: in STD_LOGIC_VECTOR(15 downto 0);
     B_P: out STD_LOGIC_VECTOR(15 downto 0));
end barrel;

architecture RTL of barrel is

signal SEL1,SEL2: STD_LOGIC_VECTOR(1 downto 0);
signal C: STD_LOGIC_VECTOR(15 downto 0);
begin
  FIRST_LVL: process (A_P, SEL1)
  begin
       case SEL1 is
```

```
            when "00"=> -- Shift by 0
                  C <= A_P;
            when "01"=> -- Shift by 1
                  C(15) <= A_P(0);
                  C(14 downto 0) <= A_P(15 downto 1);
            when "10"=> -- Shift by 2
                  C(15 downto 14) <= A_P(1 downto 0);
                  C(13 downto 0) <= A_P(15 downto 2);
            when "11"=> -- Shift by 3
                  C(15 downto 13) <= A_P(2 downto 0);
                  C(12 downto 0) <= A_P(15 downto 3);
            when others =>
                  C <= A_P;
      end case;
   end process; --End FIRST_LVL

   SECND_LVL: process (C, SEL2)
   begin
      case SEL2 is
            when "00"=> --Shift by 0
                  B_P <= C;
            when "01"=> --Shift by 4
                  B_P(15 downto 12) <= C(3 downto 0);
                  B_P(11 downto 0) <= C(15 downto 4);
            when "10"=> --Shift by 8
                  B_P(7 downto 0) <= C(15 downto 8);
                  B_P(15 downto 8) <= C(7 downto 0);
            when "11"=> --Shift by 12
                  B_P(3 downto 0) <= C(15 downto 12);
                  B_P(15 downto 4) <= C(11 downto 0);
            when others =>
                  B_P <= C;
      end case;
   end process; -- End SECOND_LVL
   SEL1 <= S(1 downto 0);
   SEL2 <= S(3 downto 2);
end rtl;
```

当上面两个设计在XC4005E-2**器件中实现时，后者使用的总门数减少了**64%（**从占**88**个**CLB**减少到**32**个**CLB）。**另外，速度上也有**19%**的提高，从**35.58ns（5**级逻辑）减少到**28.85ns（4**级逻辑）。如果采用流水线方式，在两级多路选择器之间插入寄存器，其速度还能进一步提高。**

5.4 高效利用CLB或IOB资源

5.4.1 组合逻辑合理划分（四输入特点）

在FPGA**中，由**CLB**实现大部分逻辑。设计高速**FPGA**的目标是能将大多数逻辑映射到一个可编程逻辑块（**CLB**）中。** spartan**系列的**CLB**结构如下图所示。共有**3**个查找表（**LUT**）实现逻辑函数产生器，**2**个触发器和**2**组信号控制选择器。**2**个**16X1**存储器查找表（**F-LUT**和**G-LUT**）用于实现**

4输入的函数产生器，每个4输入查找表可以实现任意4个独立输入信号的布尔函数。3输入查找表（H-LUT）可以实现可以实现任意3输入的布尔函数。

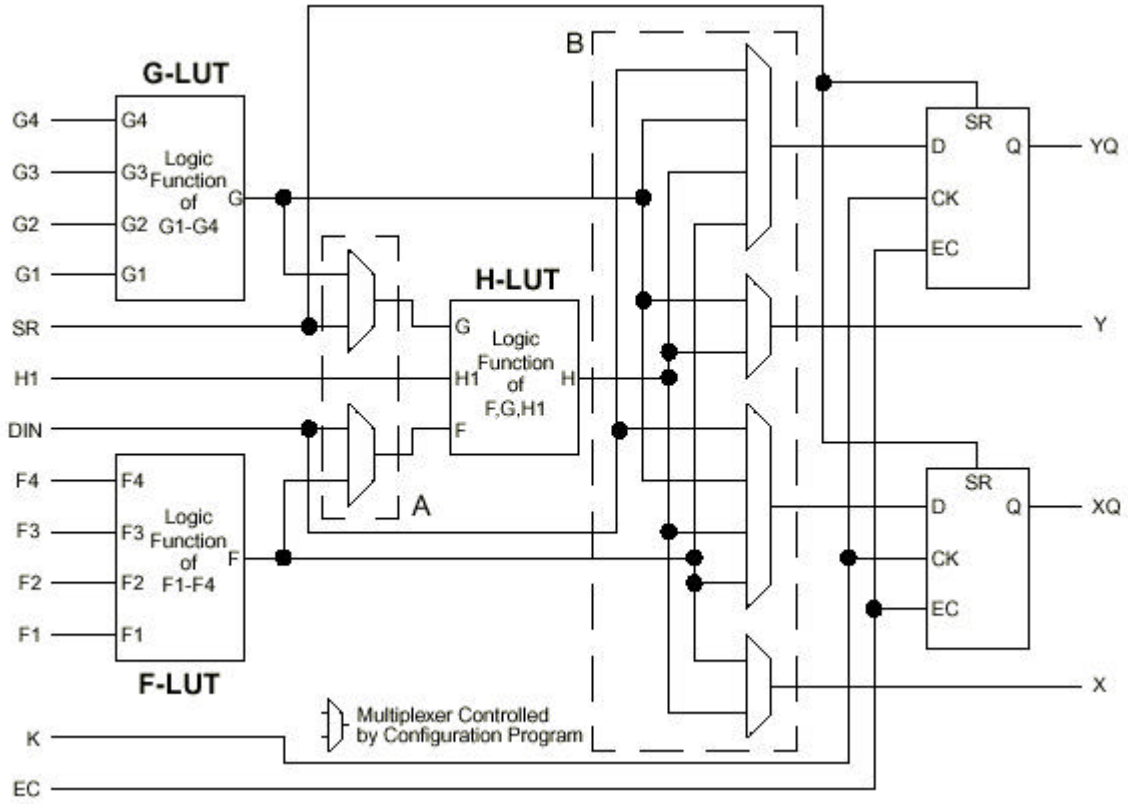


图29 CLB结构（spartan）

由4输入查找表（F或G函数产生器）和3输入查找表（H函数产生器）组合，在spartan器件的CLB中可以实现4种组合逻辑块。从上图可以推出，在一个CLB里，可以做到：(a)可实现任意4变量布尔函数；(b)可实现某些6变量布尔函数；(c)可实现任意5输入布尔函数，为F-G-H组合的特殊情况；(d)可实现某些9输入函数，如奇偶校验等。

在单个CLB中实现宽输入函数可以减少关键路径上的CLB级数和延时，因此提高了面积和速度指标。以Xilinx的Virtex器件为例，可在一个CLB中完成8选1的多路选择器，其延时只有一级CLB延时加一级局部互连延时，采用-6的器件时为2.5ns的延时。

设计实现的第一个阶段是对设计进行划分，必须将组合逻辑映射进各个独立的查找表中。低效的划分将降低设计的性能，并且使用更多的CLB。下图列出两种划分方式，第一种需要3个级联的函数产生器，第二种只需要2个。

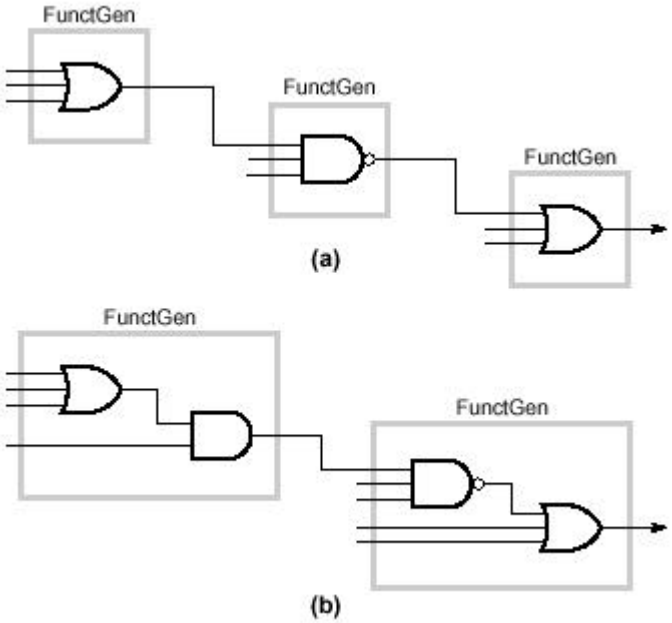


图30 合理划分组合逻辑

说明：图中的虚心框表示模块划分，表示各组合逻辑在不同的子模块中实现。

如果组合逻辑的扇出大于1，则综合工具总是将其输出映射为一个函数产生器的输出，而不会自动复制逻辑。下图中(b)需要更少的CLB数量，并且只有一级逻辑。对设计中的关键部分（对延时要求较高）进行仔细的划分，能提高其性能。

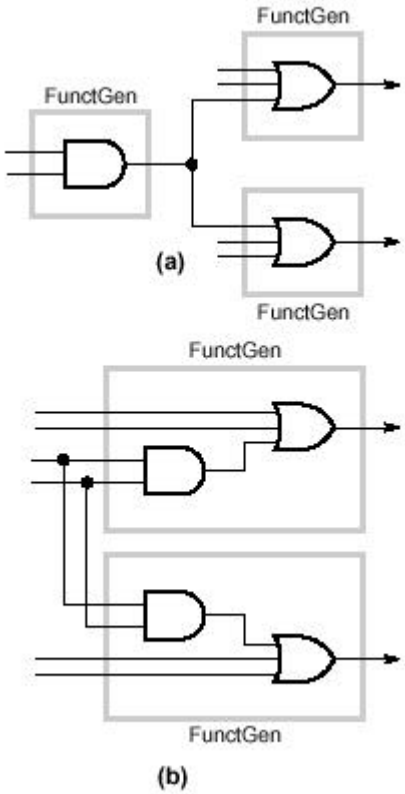


图31 不能盲目共享资源

下面的VHDL**实例中**，(a)**需要**2**个**CLB，**有**2**级逻辑延时**，(b)**只需一个**CLB，**为一级延时**。

（a）2**个**CLB

temp <= a and b

```
one : process (clk, temp, d, en) begin
     if (clk'event and clk ='1') then
          if (en = '1') then
               q2 <= temp or d;
          end if;
         end if;
end process one;
part_two: process (clk, temp, c) begin
    if (clk'event and clk ='1') then
           q1 <= temp or c;
    end if;
end process part_two;
```

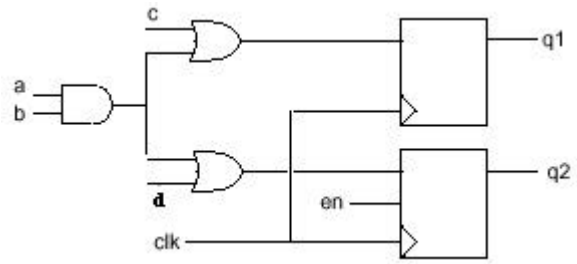


图32 **门数虽少，但占用较多资源**

```
part_one: process (clk, a, b, c, d, en) begin
    if (clk'event and clk ='1') then
          if (en = '1') then
            q2 <= a and b or d;
          end if;
    end if;
end process part_one;
part_two: process (clk, a, b, c) begin
    if (clk'event and clk ='1') then
           q1 <= a and b or c;
    end if;
end process part_two;
```

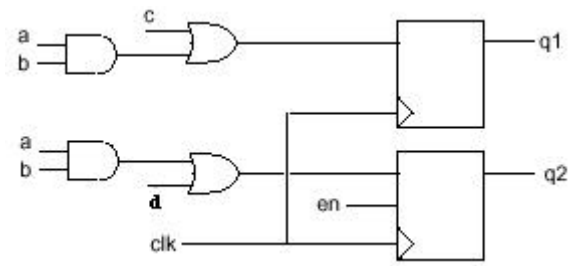


图33 门数虽多，但占用资源较少

特别注意：一些综合工具自动在关键路径上复制逻辑。另一些综合工具检测到两个进程中有相同的表达式“a&b”，会用同一个与门实现此表达式。如果综合工具自动实现子表达式的共享，则必须用引用（instantiation）元件的方式完成逻辑的复制。

5.4.2 充分利用IOB资源

将逻辑从CLB中移到IOB中，可以增加走线资源，因而能提升了设计性能。下面以Virtex系列为例，介绍IOB的特点。

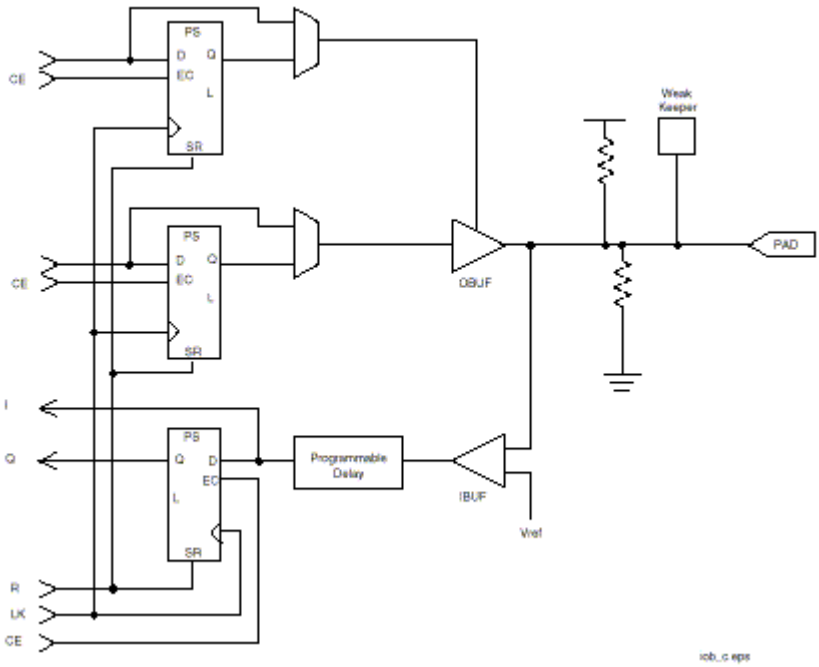


图34 IOB结构图（Virtex）

为了满足外部保持时间要求，在外部引脚和IOB输入触发器（或锁存器）的D输入端 有一个延时块。如果不需要，可以通过NODELAY设置（可通过UCF）去掉输入寄存器的延时块。

在IOB中，输出信号驱动可编程的三态输出buffer。IOB中的寄存器为上升沿触发，时钟的极性在IOB中可以翻转。Xilinx软件自动将反相器优化进IOB中。大多数FPGA综合工具能将与输出管脚相连的触发器优化进IOB。但是，有些工具不能将与双向管脚相连的触发器优化进IOB。

IOB中包含有一个输入寄存器（或锁存器）和一个输出寄存器。在设计中，应当尽量利用IOB中的寄存器，以减少CLB的使用量。另外，将输入寄存器移入IOB减少了外部的建立时间，如下图所示。尽管将输出寄存器移入IOB可能会增加内部建立时间，但可以减少clock-to-output延

时。

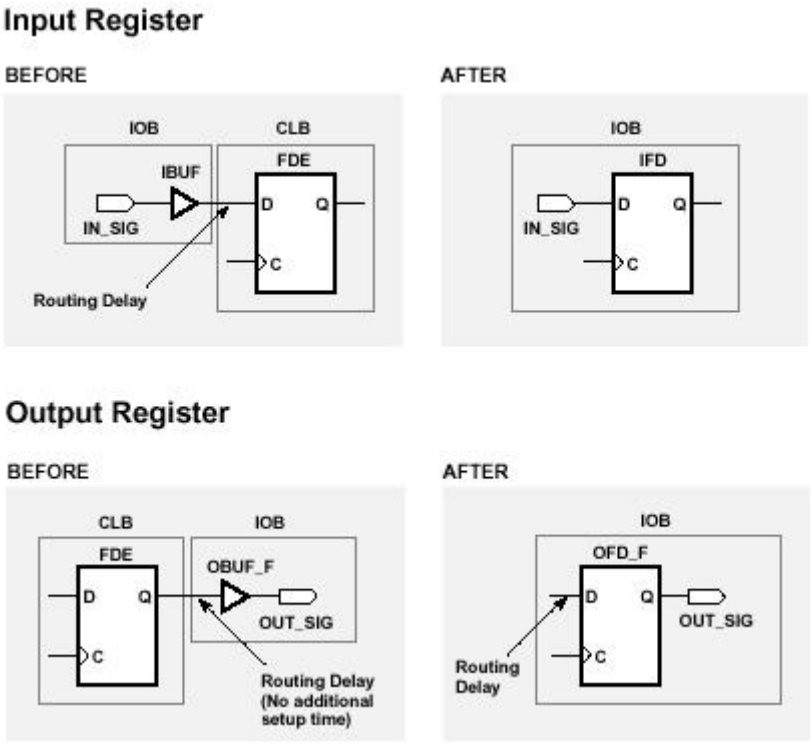


图35 输入输出寄存器移入IOB中

通过综合工具可以控制I/O管脚的边沿速率（Slew rate）。**不管I/O配置成输入、输出还是双向，都能在IOB中获得上拉和下拉电阻。缺省时，所有未用的IOB都配置为接上拉电阻的输入信号。**

Xilinx FPGA**的IOB包含了通过输入**buffer（IBUF）**进FPGA的直接输入路径和通过三态输出**buffer（OBUFT）**出FPGA的输出路径。输入路径可以寄存，输出路径可以寄存。描述或引用双向I/O，必须分两步，先描述输入路径，再描述输出路径。**

特别注意：应当在设计的顶层描述I/O路径。如果在低层模块描述I/O路径，综合工具可能生成不正确的I/O结构。可以采用下面方法产生双向I/O。

1、**描述I/O路径的行为。**

2、**引用合适的IOB库元件**

3、**用**LogiBLOX**生成。**

下面给出1、2**两种方法的**VHDL**实例，例中假定使用**LATCH**不会产生任何问题**

例1、**双向IO口设计**

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity bidir_infer is
    port (DATA : inout STD_LOGIC_VECTOR(1 downto 0);
        READ_WRITE : in STD_LOGIC);
end bidir_infer;
```

```
architecture XILINX of bidir_infer is
    signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);
    begin
     process(READ_WRITE, DATA)
     begin
         if (READ_WRITE = '1') then
            LATCH_OUT <= DATA;
         end if;
     end process;

     process(READ_WRITE, LATCH_OUT)
     begin
         if (READ_WRITE = '0') then
            DATA(0) <= LATCH_OUT(0) and LATCH_OUT(1);
            DATA(1) <= LATCH_OUT(0) or LATCH_OUT(1);
         else
            DATA(0) <= 'Z';
            DATA(1) <= 'Z';
         end if;
     end process;
end XILINX;
```

例、

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity bidir_instantiate is
    port (DATA : inout STD_LOGIC_VECTOR(1 downto 0);
        READ_WRITE : in STD_LOGIC);
end bidir_instantiate;

architecture XILINX of bidir_instantiate is
    signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);
    signal DATA_OUT : STD_LOGIC_VECTOR(1 downto 0);
    signal GATE : STD_LOGIC;


    component ILD_1
     port (D, G : in STD_LOGIC;
           Q : out STD_LOGIC);
    end component;

    component OBUFT_S
     port (I, T : in STD_LOGIC;
           O : out STD_LOGIC);
    end component;
```

```
begin
 DATA_OUT(0) <= LATCH_OUT(0) and LATCH_OUT(1);
 DATA_OUT(1) <= LATCH_OUT(0) or LATCH_OUT(1);
 GATE <= not READ_WRITE;

 INPUT_PATH_0 : ILD_1
        port map (D => DATA(0), G => GATE,
                  Q => LATCH_OUT(0));

 INPUT_PATH_1 : ILD_1
        port map (D => DATA(1), G => GATE,
                  Q => LATCH_OUT(1));

 OUPUT_PATH_0 : OBUFT_S
        port map (I => DATA_OUT(0), T => READ_WRITE,
                  O => DATA(0));

 OUPUT_PATH_1 : OBUFT_S
        port map (I => DATA_OUT(1), T => READ_WRITE,
                  O => DATA(1));
end XILINX;
```

为了充分合理利用IOB**资源，我们建议：**

1. **所有的输入输出信号都经过寄存器处理。这么做能够放宽对外部电路或其它芯片的时序要求，提高单板设计速度。并且，**IOB**中的寄存器不用白不用。**

2. **在处理双向口时，为了将三态控制寄存器、输入输出寄存器、三态电路都移入**IOB**中，减少**CLB**的使用量，应当：**

（1）**采用同样的时钟触发；**

（2）**同样的异步复位电路；**

（3）**每个管脚都有独立的三态控制信号，并且三态控制信号直接来自寄存器。三态控制信号不许共用；**

（4）**三态控制低电平有效，否则三态寄存器无法引进**IOB**，增加信号输出延时。**

关于三态控制信号任何设计，可参见“5.4.5 **用三态**Buffer**实现多路选择器”。**

另外，xlinx**器件有专门的时钟**IOB**，内设时钟驱动**Buffer（BUFGP），**详情参见下一节。**

5.4.3 全局时钟Buffer

对于设计中的全局信号，一般要使用全局时钟buffer**，这样可以充分利用目标器件的专用全局时钟网络的低时钟偏差和高驱动能力的优点。当器件的输入脚信号驱动的是一个时钟信号时，综合工具自动插入一个全局时钟**buffer（BUFGP）。

例如，每个XC4000E/L**和**Spartan**器件包含**4**个一级（**primary**）和**4**个二级（**secondary**）全局**buffer，**它们共享相同的走线资源，如下图所示。只有这**8**个全局**buffer**能获得全局走线资源。全局走线资源与正常走线通道是完全独立的，所以不占用资源。**

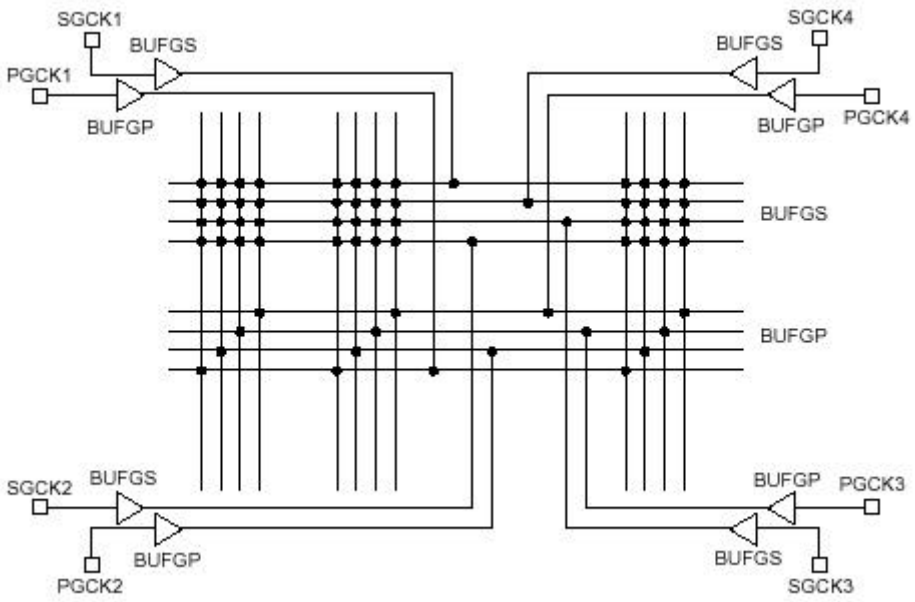


图36 BUFGP与BUFGS

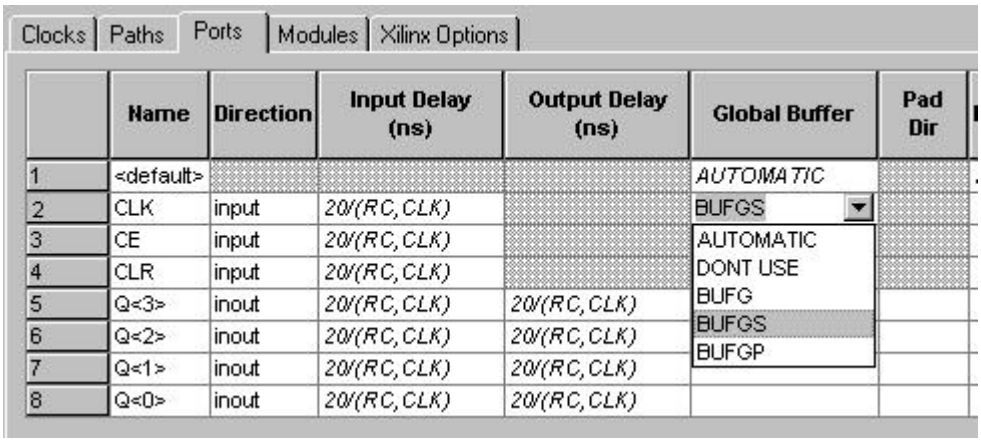
对于来自FPGA内部需要高扇出和低时钟偏差的信号，可以使用二级全局buffer（BUFGS）。使用BUFGS，只需要引用BUFGS元件。对于来自FPGA外部的信号，可以使用一级全局buffer（BUFGP）驱动分配信号，BUFGP必须由半专用管脚（semi-dedicated）驱动。使用BUFGP能对内部信号进行全局分配。BUFGS具有更灵活的走线能力，所以应尽量考虑使用。

对于需要高扇出和低时钟偏差的时钟信号必须使用全局buffer。但是，也可以用全局buffer驱动高扇出的时钟使能、清零和CLB和IOB的时钟脚（K）。如果设计中的高扇出信号少于4个，可以用余下的全局buffer驱动其它次高扇出的信号，这样可以减少走线的阻塞。Xilinx推荐BUFGS用于驱动设计中的高扇出信号（如时钟、时钟使能和复位信号等）。一般由内部产生的时钟信号必须用寄存器打一下，避免毛刺。

Xilinx的实现软件可以针对具体的设计结构自动选择合适的时钟buffer。如果想用指定的全局buffer，就必须引用它（instantiate）。可以在VHDL代码中引用或利用综合工具插入全局时钟buffer。 如果你熟悉器件的结构，并想指定使用指定的资源，必须加LOC属性，下面给出了VHDL实例。

注意：有的综合有insert pads功能，只要说明某管脚是时钟pad，则会自动插入BUFG，因而在VHDL代码中不需要对时钟pad引用BUFG。

```
attribute LOC: string;
attribute LOC of CLOCKBUF: label is "BR";
...
CLOCKBUF:BUFG port map(I=>oscout,O=>clkint);
```



每个器件的BUFGP**或**BUFGS**资源是有限的。下表列出了各种器件的全局**bufferz**资源。

Device Family	Global Buffer Symbol Name	Number Available	Foundation Express Infers Maximum of
XC3000	GCLK	1	1
	ACLK	1	1
XC4000/XC4000E/ XC4000L	BUFGS	4	0
	BUFGP	4	0
	BUFG	(8)	4
XC4000EX/ XC4000XL/ XC4000XV	BUFGLS	8	0
	BUFGE	4	0
	BUFFCLK	4	0
	BUFG	(8)	8
XC5200	BUFG	4	4
Spartan	BUFGS	4	0
	BUFGP	4	0
	BUFG	(8)	4
SpartanXL	BUFGLS	8	0
	BUFG	(8)	8
Virtex	BUFG	4	4
XC9500	BUFG	3	0

5.4.4 专用全局Set/Reset资源

X4000**和**Spartan**器件具有专用的全局置位/复位（**GSR**）网络，可以用来初始化所有的**CLB**和**IOB**中的存储元件，如对**flip-flops, RAM, and ROM**进行异步置位或复位。使用**GSR**网络，必须通过引用元件**STARTUP**的**GSR**或**STARTBUF**的**GSRIN**脚，如下图所示。由于**GSR**网络有专用的走线资源与每个触发器的置位或清零端相连，使用**GSR**网络可以提高设计性能和减少走线阻塞。

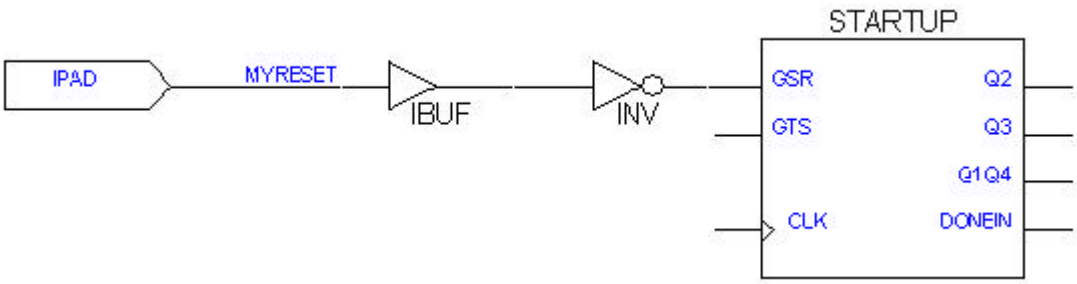


图37 低电平复位电路

CLB中的每个触发器既可以配置成全局置位或复位（GSR），也可以由局部置位/复位（SR）控制（有些器件无SR，如4062xla）。下图为spartan器件的CLB中触发器的功能框图。由图所示，局部置/复位SR和全局置/复位GSR是或的关系。一个触发器可以由SR置位，也可以由GSR置位。类似的，如果在复位模式，SR和GSR都可以进行复位。

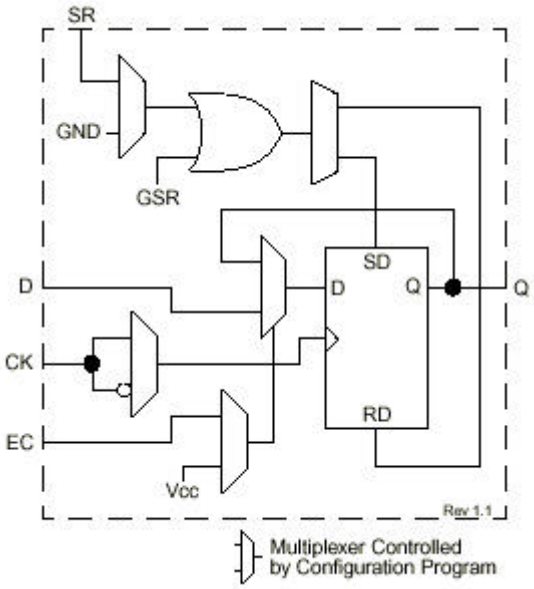


图38 SR与GSR

对X4000和Spartan器件，GSR信号为高电平有效，如果要求低电平有效，必须在代码中引用一个反相器对此全局复位信号求反。

使用CLB中触发器的局部置位/清零端（SR）可以对每个独立的触发器进行置位或清零操作。如果触发器的置位或清零端连到全局复位信号，则全局复位信号将控制触发器的初始状态。如果触发器的置位或清零端不连信号，则缺省的初始状态为清零状态。可以给触发器增加INIT=S属性改变初始状态为置位状态。为了增加代码的通用性，我们建议触发器不要采用隐式复位（缺省值），而采用显式复位。

许多设计中都有一个全局复位信号用于初始化设计中的触发器为已知状态。 在VHDL中使用专用全局复位资源，可以调用STARTUP元件，将用户的复位信号与它的GSR脚相连， 复位信号不必连到每个触发器上，如下例所示。

U1: STARTUP port map (GSR => RESET);

下表列出了各种器件的全局Set/Reset资源。

Family	Global Buffer Symbol Name
XC3000	none
XC4000	STARTUP
XC5200	STARTUP
Spartan	STARTUP
Virtex	STARTUP_VIRTEX
XC9500	BUFGSR

5.4.5 用三态Buffer实现多路选择器

多路选择器一般用CLB中的函数发生器实现。一个4选1的多路选择器可以在XC4000或Spartan的单个CLB中实现，因此效率较高。6个输入信号（4个输入，2个选择）使用函数发生器的F、G和H。但是，如果多路复用器规模大于4选1，则将超过一个CLB的容量。例如，16选1的多路复用器需要有2级逻辑，占用了5个CLB，结果增加了面积和延时。因此，Xilinx推荐使用内部的三态buffer（BUFT）实现规模较大的多路复用器。用BUFT实现规模较大的多路复用器具有下面一些优点：

1、在输入宽度变化时，面积和延时特性几乎不变。

2、多路选择器的最大输入宽度与目标器件的每条水平长线的三态buffer数目相等。

3、BUFT在CLB之外，是专门用于三态电路的，不占CLB资源，不用白不用。

4、选择信号采用one-hot编码。

下面给出两种方法的VHDL实例。

例、用函数发生器实现

```
SEL_PROCESS: process (SEL,A,B,C,D,E)
begin
   case SEL is
           when "000"  => SIG <= A;
           when "001"  => SIG <= B;
           when "010"  => SIG <= C;
           when "011"  => SIG <= D;
           when others => SIG <= E;
   end case;
end process SEL_PROCESS;
```

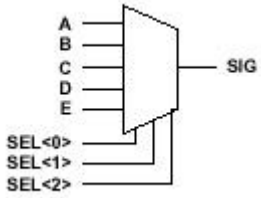


图39 多路选择

例、用BUFT实现

```
SIG <= A when (SEL(0)= '0') else 'Z';
SIG <= B when (SEL(1)= '0') else 'Z';
SIG <= C when (SEL(2)= '0') else 'Z';
SIG <= D when (SEL(3)= '0') else 'Z';
SIG <= E when (SEL(4)= '0') else 'Z';
```

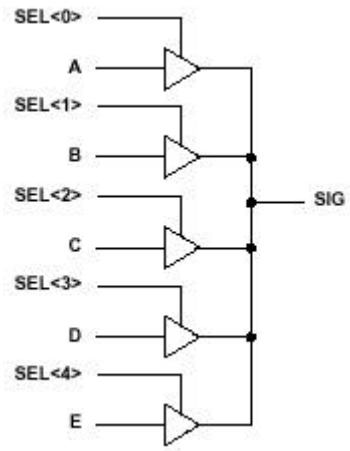


图40 采用三态电路实现电路选择

注意：xilinx的三态电路是低电平导通，高电平三态。若设计中采用高电平导通，低电平三态，则相应的三态控制信号会增加一个反向器，有可能会增加一级CLB（例如三态控制信号直接来自寄存器输出），导致不必要的增加电路延时。因此，设计中尽量采用低电平导通、高电平三态方式。

5.4.6 存储器的实现

XC4000E和Spartan系列FPGA提供片内分布式RAM和ROM。利用CLB的函数发生器可以配置成ROM（ROM16X1，ROM32X1），电平触发RAM（RAM16X1，RAM32X1），边沿触发的单口RAM（RAM16X1S，RAM32X1S）或边沿触发的双口RAM（RAM16X1D）。Spartan中没有电平触发RAM。分布式RAM可用于状态寄存器、索引寄存器、计数存储、常系数乘法器、分布式移位寄存器，LIFO堆栈或任何数据存储操作。双口RAM简化了FIFO的设计。

实现ROM常用下面三种方法：

1、引用系统库中的16X1或32X1的ROM元件，为了定义ROM的值，必须用Set属性设置ROM元件的INIT特性。。

2、用LogiBLOX实现任意尺寸的ROM。

不要用RTL描述实现RAM，因为这样编译效率低且会引起带反馈的组合逻辑。实现RAM常用下面三种方法：

1、引用系统库中的16X1或32X1的RAM元件

2、用LogiBLOX实现任意尺寸的RAM。

3、综合工具根据代码导出RAM。

下例给出了第1种方法的VHDL实例。

例、

```
RAM0 : RAM16X1S port map  (O => DATA_OUT(0), D => DATA_IN(0),
                           A3 => ADDR(3), A2 => ADDR(2),
                           A1 => ADDR(1), A0 => ADDR(0),
                           WE => WE, WCLK => CLOCK);
```

推荐使用同步写、边沿触发的RAM（RAM16X1S，RAM32X1S，RAM16X1D），这样可以简化写的时序和提高RAM性能。

对Virtex系列，RAM有分布式RAM（用LUT实现）和专门的Block RAM。若是分布式RAM，并且RAM比较大，则其占用的LUT将比较分散，为了提高设计速度，我们建议对RAM的地址和输出采用寄存器处理方式。

5.4.7 专用I/O译码器

XC4000系列在器件的每边分布有4个宽输入的译码器电路，每个译码器对输入信号进行线与，输入信号来自对应边的IOB或内部的CLB，如下图所示。

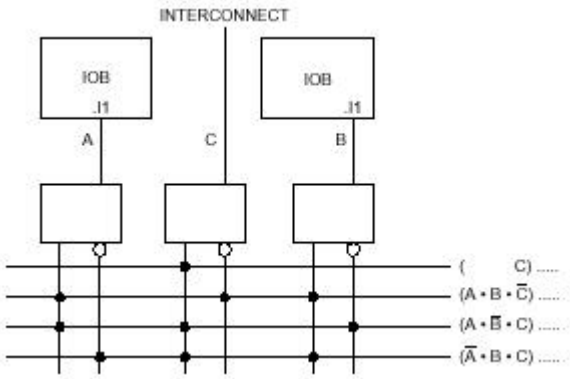


图44 IO译码电路

在HDL中使用XC4000系列的宽输入译码器，可以引用DECODE1_IO，DECODE1_INT，DECODE4，DECODE8，DECCODE16等元件。宽译码器的输出为开路输出，需要上拉电阻，所以必须引用PULLUP元件。下面为使用宽输入译码器的VHDL实例。

例、

```
----- Instantiation of Edge Decoder: Output "DECODE(0)" -----
A0: DECODE4 port map (ADR(3), ADR(2), ADR(1), ADR_INV(0), DECODE(0));
A1: DECODE1_IO port map (ADR(4), DECODE(0));
A2: DECODE1_INT port map (CLB_INV(0), DECODE(0));
A3: DECODE1_INT port map (CLB_INT(1), DECODE(0));
A4: DECODE1_INT port map (CLB_INT(2), DECODE(0));
A5: DECODE1_INT port map (CLB_INT(3), DECODE(0));
```

```
A6: PULLUP port map (DECODE(0));
```

5.4.8 实现边界扫描（JTAG 1149.1）

XC4000、Spartan等FPGA包含了与IEEE Standard 1149.1兼容的边界扫描电路。Xilinx器件支持外部测试和部分内部自测试。在上电和开始配置之间的一段时间可以访问内建边界扫描逻辑。为了能在配置完成后访问边界扫描逻辑，必须在HDL设计中引用边界扫描符号BSCAN和边界扫描I/O管脚TDI、TMS、TCK和TDO，如下例所示。

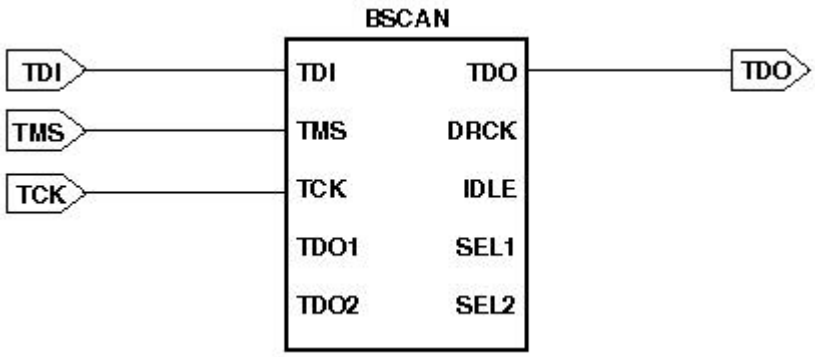


图41 边界扫描电路

```
U1: BSCAN port map (TDI=>TDI_P, TMS=>TMS_P, TCK=>TCK_P, TDO=>TDO_P);
U2: TDI port map (I=>TDI_P);
U3: TMS port map (I=>TMS_P);
U4: TCK port map (I=>TCK_P);
U5: TDO port map (O=>TDO_P);
```

5.4.9 元件的引用（CoreGen/LogiBLOX）

如果你想使用的模块不在系统库中，可以使用LogiBLOX或CoreGen生成元件，并在你的代码中引用。这对于大规模的存储器阵列很有用。生成元件的同时，也生成了仿真模型，可以在编译前进行RTL级仿真。 LogiBLOX是一个图形化的工具，能够生成算术运算、逻辑、I/O、时序和数据存储模块，并在设计中引用。下面的例子是引用LogiBLOX生成的RAM。

例、

```
instance_name : MY_RAM port map
(A => addr_sig,
 DO => dataout_sig,
 DI => datain_sig,
 WR_EN => write_enable_sig,
 WR_CLK => write_clock_sig);
```

第六章 代码可重用性设计

1 目的

提高HDL代码的可读性、可修改性、可重用性。使设计有好的综合与仿真效果，指导设计工程师进行可重用性代码设计，规范HDL语言的输入。

2 范围

本规范适用于可重用模块(IP)设计,使用IP的系统设计以及一般的设计。

3 基本原则

使代码简单而规整。简单规范的结构，从本质上是易于设计编码、验证和综合，可重用设计编码总的目标应该是：在满足功能和性能目标的前提下尽可能使设计简单。下面是一些基本的规则：

- 简单的时钟方案；
- 使用一致的编码风格，一致命名公约，一致的Process和状态机结构；
- 使用规整的模块划分方案，锁存所有输出，模块规模大致相等；
- 使用注释、有意义的命名、常量或参数、使代码易于理解。

4 原则描述

4.1 命名定义

①为设计开发一个命名约定，以文档的形式记录下来，并在整个设计中采用一致的命名约定；

②对于所有的信号名，变量名和端口名应小写；

③对于常量名和用户定义的类型用大写；

④使用有意义的信号名、端口名、函数名和参数名；

如：对于RAU的地址总线用ram_addr， 而不用ra；

⑤如果设计中用了几个参数，使用短的，但足以清楚的命名；

⑥来自同一驱动器源的所有时钟信号使用相同名字；

⑦对于低电平有效的信号，应该以一个下划线跟一个小写字母结尾如_b或_n，使整个设计中使用同一个小写字母结尾来表明低电平有效；

⑧对复位信号使用rst，如果复位信号是低电平有效，使用rst_n作为名字；

⑨对端口和连接端口上的信号尽可能使用相同的和相似的名字；

⑩尽可能的使用下表中的名字约定：

约定	用途
*—r	寄存器输出
*—a	异步信号
*—pn	在第n步｛phase｝使用的信号
*—nxt	锁存前的信号
*—z	三态内部信号

4.2 VHDL中Architecture的命名约定

对于VHDL中的Architecture建议使用下表中的约定

Architeture	命名约定
综合模型	ARCHITECTURErtl OF my-syn-model IS or ARCHITECTUREstr OF my-structurou-design IS
仿真模型	ARCHITECTUREsim OF my-behave-rrvodcl IS or ARCHITECTUREtb OF my-test-bench IS

4.3 源文件中要有文件头

在源文件、Script文件的开始应包含一个文件头，文件头至少包含下列信息：

①文件名；

②作者；

③模块功能描述；

④关键特征的列表；

⑤文件产生的日期、更改、记录（日期、更改者、更改的内容）。]

请见下面模板：

```
//
// (C) COPYRIGHT 1999, HUAWEI Technology Inc.
// ALL RIGHTS RESERVED
//
// FILE name: A_program .ihd
// Author : Zhang Zhanlong
// E_mail : Zhangzhanlong @ huawei.com.cn
// Date : /2000/6/4
// version : 1.0
// Abstract : This file is the top_level of ……
//                 ……
// Called by : None
// Modification: History:
// Date BY Revision charge decsription
```

```
//  ------------------------------------------------------------
//  99/11/10   ZZL    1.0        original
//  99/11/20   ZZL               Vse   RAM to replace vegistor
//    ……④
```

4.4 使用注释

①使用适当的注释来解释所有的process、函数和类型以及子类型的声明。

②使用注释来解释端口、信号、变量、信号组和变量组。

③注释应该放在它所描述的代码的附近，注释应该简明扼要，并足够说明问题，避免注释杂乱，显而易见的功能不用加注释，注释关键是说明设计意图。

④当代码需要拷贝时,请注意要修改相应的注释,以免产生错误的指引,导致理解上的错误。

4.5 独立成行

对于HDL语句使用独立一行，尽管VHDL和Verilog允许一行写多个语句，但每个语句独占一行或增加可读性和可维护性。

4.6 行长度

保持每行小于或等于72个字符，因为有的终端式打印机每行不能超过80个字符，规定72个字符是为了留出边空，提高可读性。还有个原因是VI编辑器有显示行号的地方。

4.7 缩进

①用缩进来提高续行和嵌套循环的可读性。

②缩进采用2个空格，如果空格太多深层嵌套时会限制行长。

③设置Set tabstop=z

4.8 不要使用HDL的保留字

在HDL源代码中任何元素、端口、信号、变量、函数、任务、模块、ENTITY等的名字，不能取VHDL和Vevilog的保留字，因为设计要在VHDL和Veviley之间转换，所以Verilog的保留字不能用于VHDL中，VHDL的保留字也不能用在Verilog中。

4.9 端口顺序

①用一逻辑的顺序来声明端口，并且在整个设计中保持一致的顺序。

②每行声明一个端口并有注释，最好在同一行。

③建议用下述顺序声明端口。

Inputs:

- CLocks;
- Resets;
- Enables;
- Other control Signals;

- Data and address Lines。

Outputs:

- Clocks；
- Resets；
- Enables；
- Other control signats；
- Data。

4.10 端口映射和Generic映射

①对端口和Generic采用显示的映射，使用名字相关的映射而不要用位置相关的映射；

②在输入和输出每类端口之间，一个空行来提高可读性。

4.11 VHDL Entity_Architecture和Configuration段。

对VHDL设计，将Entity，Architecture和Configuration段放在同一文件中，将一个特定设计的所有信息放于一个文件，可使得易于理解和维护。

如果在有entity和architecture声明的源文件中包含了子设计的configuration，必须对综合做出注释，在VHDL中可以使用program translate_off和progam translate_on两个虚拟注释来做到这一点。例：

```
－－program translate_off
configuratin cfg_example_struc of example is
    for struc
        use example_gate;
end efg_example_struc;
－－program translate-on
```

4.12 使用函数

尽可能使用函数，而不要将同一段代码重用多次，如果有可能可以将函数通用化，以使得它可能重用，要对函数加注释。

4.13 使用循环、 LOOP和数组

①使用循环数组来提高源代码的可读性，这样可以有效减少行数。

②在仿真时，数组比for loop快得多，为了提高仿真性能，尽可能对数组用向量操作而不要用for loop。

4.14 使用有意义的标号

①对每一个process块加一个有意义的标号，这有助于调试，可以通过标号来设断点；

U1： process

②每个process块的标号取名〈name〉－PROL；

③每个instance的标号取名 V－〈name〉

④对于一个多嵌套的设计，保持标号有意义，同时应使得标号尽量短，长的标号可能导致在设计嵌套中过多的路径名；

⑤标号名不要和信号名，变量名或entity名重名。

5 可移植性的编码准则

5.1 只使用IEEE的标准类

①只使用IEEE的标准类；

可以定义子类型，但所有子类型必须基于IEEE的标准类型；

②使用std-logic而不要用std-ulogic，使用std-logic-vector而不要std-wlogic-vector；

③尽可能少的定义子类型，使用太多的子类型使得代码难于理解；

④不要使用bit或bit-vector类型，许多仿真器没有为这种类型提供内部数学函数。

5.2 不要直接使用数字

在设计中不要直接使用数字，Hard-Coded Numeric Value，作为例外，可以使用0和1，但不要组合使用，如1001。

例如：

差的编码风格

```
wire     [7:0]    my-in-bus;
reg      [7:0]    my-out-hus;
```

好的编码风格

```
define MY-BVS-SIIE 8
wire   [MY-BUS-SIIE-1:0]  my-in-bus;
reg    [MY-BOS-SRIE-1:0] my-out-bus;
```

5.3 使用package

将一个设计的所有参数值和函数集中到一个单独的文件package中，并改名字，DesignName-Package.vhd。

5.4 VHDL到Verilog的变换（针对VHDL）

①不要使用generate语句，在verilog中没有相应的结构；

②不要使用 block结构，在verilog中没有相应的结构；

③不要使用代码去改变constant声明，在verilog中没有相应功能。

6 Clock和Reset编码准则

基本原则：简单的时钟结构易于维护理解和分析，它也能一贯地产生好的综合结果，最好的是有单一的全局时钟，所有Reg都在上升沿触发。

6.1 避免使用混合时钟沿

①避免在设计中既使用上升沿又使用下降沿来触发寄存器。混合时钟沿的设计在延时要求苛刻的设计中可能是必要的，但经常会产生一些问题，使用须特别小心；在时序分拆时，除了时钟频率之外，时钟的占空比成为一个关键问题；多数基于扫描链的测试方法要求对一上升沿和下降沿触发的寄存器分开处理。

②如果不得有使用混合的时钟沿，在综合和时序分析时确保能满足时钟精度最差情况下的占空比，在物理设计中，占空比可能不会正好是50%，可供信号传输的实际时间会比时钟周期的一半要不。

③如果不得有用混合时钟沿要确保把假定的占空比写入用户文档。core的用户是芯片的设计者或系统集成者，在多数设计中占空比是时树的函数，而时钟树的插入通常又依赖于具体的工艺，使用Core的芯片设计者必须检查实际的占空比能够满足Core的要求，也应该了解怎样改变综合和时序分析策略以使得Core能够满足实际的条件。

④如果必须使用大量的上升沿和下降沿触发的寄存器将上升沿和下降沿触发的寄存器发到不同的模块中是很有用的，这样容易确定下降沿触发的寄存器，并可将它们放到不同的扫描链中。

6.2 避免使用时钟Buffer

避免在RTL级手工实例化时钟Buffer。

时钟Buffer通常是在综合以后在物理设计时插入的，在可综合的RTL代码中，时钟网络通常被认为是理想网络，没有延时，在布局线时，时钟树插入工具插入适当的结构，尽可能的接近理想的平衡的时钟配件网络。

6.3 避免门控时时钟

避免在RTL级使用门控时钟。

门控时钟电路依赖于具体的工艺和时序，门控时钟不正确的时序可能导致假的时钟和误操作，不同局域的时钟SKEW还会导致保持时间冲突violation门控时钟，但它们不应该出现在RTL级的编码中，象power Compiler这类工具可以自动去做。

6.4 避免内部产生时钟信号

避免使用内部产生的时钟信号

内部时钟会降低电路的可测性，也使得综合的约束变得困难。

6.5 门控时钟和低功耗设计

①如果设计中必须使用门控时钟或复位信号，应该让产生这些信号的电路位于设计顶层的一个独立的模块中，根据电路所使用的时钟和复位信号划分子模块。

特别地 Core中不应出现门控时钟，如果需要，时钟的电路应该出现在设计的顶层，将时钟和复位信号隔离成单独的模块可以解决一系列问题。

- 对于其他模块将可采用标准的时序分拆和扫描链插入技术；
- 它将违反编码规范的地方限制在一个小的范围内；
- 它有利于对这些产生电路开发特殊的测试策略

②如果设计中需要使用门控时钟，使用下例所示的同文打入。

```
--Poor coding style:
clk-p1<=  CLK  and  pl-gate;
EXA-PROC: Process (clk-pl)
        kegin
          if(clk-p1' eveut and clk-p1='1') then
          ……
          end if;
end process EXA-PROC;
--Good  coding style
EXB-PROC:  Process(clk)
        begin
          if(clk' tvent and clk='1') then
            if(p1-gate='1') then
            ……
             endif;
          endif;
 end process EXB-PROC
```

6.6 避免内部产生的复位信号

①确保所有寄存器只被简单的复位信号控制

②尽可能避免内部产生的条件复位信号，通常模块内所有寄存器应在同一时间内被复位，这种方式可使代码更易读，并易于综合出好的结果。

7 针对综合的编码准则

7.1 寄存器推断

使用下述模板来描述独立于工艺的寄存器，用多位信号来初始化寄存器，在VHDL中不要在声明的时候给信号赋初值。

例：时序逻辑的VHDL模板

```
--Process with synchronous reset
EXA-PROC: process(clk)
begin
  if(clk' event and clk='1')then
    if rst='1' then
    …
    else
     eud if;
  end if;
```

```
    end protess EXA-PROC;
－－Process with asynchronous reset
EXB-PROC：process（rst_a，clk）
begin
        if rst_a='1' then
          …
        elsif (clk'event snd clk='1') then

          …
       end  if;
      end process  EXB-PROC;
```

7.2 避免使用LATCH

①**在设计中避免产生任何**LATCH

作为例外可以实例化独立于工艺的GTECH D LATCH。**但必须显示地实例化每一个**LATCH，**并且必须在文档中列出每一个**LATCH，**描述出由于**LATCH**导致的任何特殊的时序要求，大的寄存器、内存**、FIFO**和其他的存储单元中允许使用**。

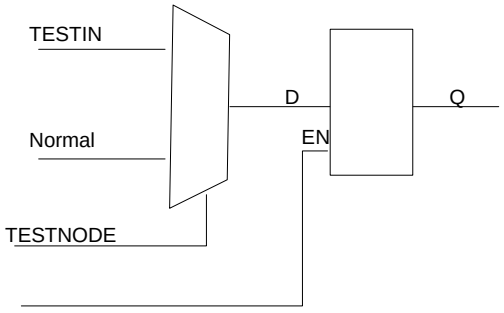
②**为检查设计中是否有**LATCH，**编译设计并用命令**all-registers-level-sensitive,**可能检查出如**LATCH**这样电平敏感单元**。

③if**语句同缺乏**else**子句**，case**语句中各个条件所处理的变量不同，在综合时推出**LATCH，**使用下述方法可避免**LATCH**在每个**prcess**的开始给信号赋初始值**。

对所有的输入条件都给出输出。

在最终优先级的出发上使用else**子句，而不用**elsif。

④**如果必须使用**LATCH,**建议使用下述电路使得**LATCH**要得可测**。



7.3 避免组合反馈

在设计中避免组合反馈电路。

7.4 定义完整的敏感表

①在每一个process（VHDL中）要求定义完整的敏感表，对于组合模块，敏感表中必须包含被process所利用的所有信号，这通常意味着所有出现在赋值语句右边和条件表述式中的信号，对于时序模块，敏感表中必须包含时钟，如果有异步复位还包含复位信号。

②确保敏感表中没有包含不必要的信号，敏感表中不必要的信号会降低仿真的性能。