航天器 FPGA 硬件描述语言编程指南

HDL Code Style Guide to FPGA Design of Spacecraft

（试行）

二〇〇八年四月八日

目　　次

1　范围.....3
2　规范性引用文件.....3
3　术语定义和缩略语.....3
4　概述.....3
5　编码风格.....4
5.1　结构化设计.....4
5.2　代码描述.....4
5.3　内部命名规则.....5
5.4　其他.....5
6　可综合代码描述规则.....5
6.1　概述.....5
6.2　复位.....7
6.3　时钟.....11
6.4　接口异步信号处理.....14
6.5　寄存器和锁存器.....17
6.6　有限状态机.....19
6.7　使用 IF-Then-Else 的有优先权的译码器.....24
6.8　使用 Case 语句的多路复用.....25
6.9　解码器.....26
6.10　计数器.....27
6.11　运算.....28
6.12　IO.....29
6.13　异步设计.....31
7　设计优化.....32
7.1　标准单元实现.....33
7.2　复杂逻辑运算单元共享.....33
7.3　中间信号.....34
7.4　针对目标 FPGA 的优化.....34
7.5　综合工具设置优化.....36
8　抗单粒子翻转效应设计.....38
附录 A.....42
附录 B.....45

航天器 FPGA 硬件描述语言编程指南

1 范围

本标准规定了航天器 FPGA 硬件描述语言编程指南。
本标准适用于航天器 FPGA 产品的研制。

2 规范性引用文件

本章无条文。

3 术语定义和缩略语

下列术语和缩略语适用于本标准。
Clock skew—时钟偏移；
Metastability—亚稳态；
EDA—Electronic Design automation，电子设计自动化；
FPGA—Field Programmable Gate Array，现场可编程逻辑阵列；
HDL—Hardware Description Language，硬件描述语言；
MTBF—Mean Time Between Failures，平均无故障工作时间；
RTL—Register Transfer Level，寄存器传输级；
SEU—Single Event Upset，单粒子翻转；
TMR—Triple Module Redundancy，三模冗余。

4 概述

硬件描述语言是用来描述数字系统行为和结构的语言。使用硬件描述语言作为 FPGA 设计输入方式适合大规模数字电路设计，可以用综合工具提高设计效率，适合采用自顶向下的设计方式，便于设计的移植和复用。

硬件描述语言从出现至今已出现了许多具体的规则需要设计人员遵循，否则设计将会出现问题。这些规则同样适用于航天器 FPGA 产品的设计，设计代码描述的随意性是航天器 FPGA 质量问题的根源之一。作为航天器 FPGA 硬件描述语言编程指南还必须包括可靠性设计和抗单粒子翻转效应设计方面的规则才能适应在空间环境应用的需求。

本标准从编码风格、可综合代码描述规则、设计优化、抗单粒子翻转效应设计等四个方面提出航天器 FPGA 硬件描述语言编程指南供设计者参考。可靠性设计要求主要有以下四条，结合在第 6 章可综合代码描述规则中提出：

a. 采用同步设计；

b. 保证复位的有效性；

c. 保证时钟信号的稳定和去除时钟偏移的影响；

d. 对异步接口信号进行处理。

采用硬件描述语言设计 FPGA，必须了解 EDA 工具(综合工具、仿真工具、实现工具)的使用要求和 FPGA 的内部结构，才能写出高效规范的代码。本标准不仅涉及 HDL 代码的编写，还涉及与之密切相关的部分 FPGA 硬件实现细节和 EDA 工具的要求。本标准中的例子仅提供 VHDL 代码，部分说明也仅针对 VHDL。

本标准最后的附录总结了使用硬件描述语言进行航天器 FPGA 设计时需要强制执行的要求(附录 A)和推荐执行的要求(附录 B)。强制类是指设计中必须要遵守的规则，推荐类是指设计中参照执行的规则。

5 编码风格

5.1 结构化设计

a. 顶层设计文件只例化底层模块和 I/O；

b. 模块例化时使用信号名和位置映射方式；

c. 每个文件内只包含一个模块(entity/module)，文件名与模块名一致；

d. 在专门的文件内打包所有的参数等定义；

e. 不同模块中的同一信号，使用同一信号加不同后缀连接，如*_i、*_o、*_c 分别表示某模块输入端口、某模块输出端口和模块之间的连接信号命名，连接信号关系复杂时可将相关模块输出定义为一个自定义类型作为连接信号的类型，连接信号命名为*(模块名)_c。

5.2 代码描述

a. 文件顶端加入注释，包括版权、项目名、模块名、文件名、作者、功能

和特点、版本号、日期、详细的更改记录等；

b. 端口申明时，同一类信号或相关信号放在一起；

c. 对代码进行详细注释，被注释过的代码和附带的文档将提供可信的设计的基础；

d. 代码要有层次结构，推荐采用制表符键(Tab)增加缩进量；

e. 组合逻辑部分和时序逻辑部分应分开描述(参见 6.5 节中带同步复位的 D 寄存器的示例)。

5.3 内部命名规则

a. 命名由字母、_、数字组成，第一个符号为字母，不超过 20 字符；

b. 除常量、用户定义类型用大写字母外，其他命名如信号、变量、端口等一般用小写字母；

c. 命名要有物理意义，如 ram_addr，clk，低电平有效信号以 n_开头；

d. 命名不能和 VHDL、Verilog、FPGA 内部资源等关键字同名。

5.4 其他

a. 仅使用 IEEE 标准类型 std_logic 及 std_logic_vector(VHDL)；

b. 使用常量代替数字，设计内部使能等信号的有效电平也可定义成常量；

c. 仅使用由高到低的总线方向；

d. 不在端口使用 buffer 类型使信号即可在内部使用又可为端口输出信号(VHDL)；

e. 禁止使用配置(VHDL)；

f. 考虑 IP 复用时，功能和接口应分开实现，使用固定的片上总线标准。

6 可综合代码描述规则

6.1 概述

采用硬件描述语言设计 FPGA 时，设计输入是行为级或 RTL 级代码描述，

从设计输入到最终的门级实现之间的转换、优化、映射等工作是由综合工具完成的，所以设计输入必须被综合工具识别，即可综合。可综合代码可以被转换成由 FPGA 内的基本库(如图 1)、连线资源等构成的硬件网表，所以必须要对硬件设计本身有很好的了解，而不是简单的软件编程；可综合代码可以被综合工具正确的转换，所以必须了解 HDL 语言描述与硬件实现的关系。

本章中(6.5 节至 6.11 节)列举了常用硬件逻辑及对应 VHDL 代码的例子。掌握常用的时序逻辑和组合逻辑的描述方法，采用 RTL 级描述可以使设计人员写出高效、标准的代码。

RTL 可综合代码应符合可综合的 VHDL 子集(IEEE1076.6)和 Verilog 子集(IEEE1364.1)。

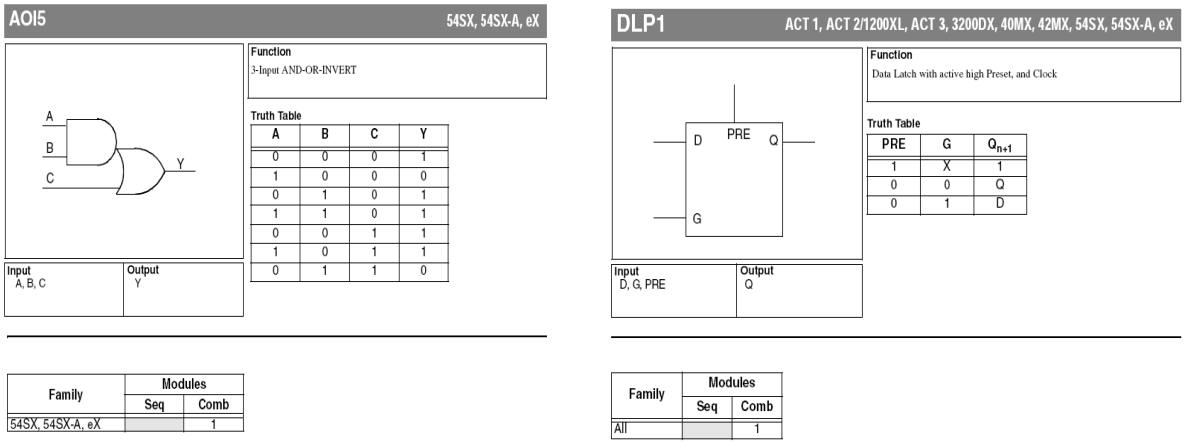


图 1 FPGA 内的基本库

同步设计具有航天应用所要求的稳定性，在变化的温度、电压之下能工作得更好。同步设计能轻易的使用不同工艺的 FPGA 实现。同步设计通过使用精确设计的时钟采样信号，可以简单的解决因信号通过不同的逻辑路径而产生不同延迟所带来的问题。逻辑之间的接口被标准的同步行为所简化，而且同步设计的时序特性能保证数据的正确接收，而异步接口需要详细精确的握手过程等方式来确保信息的完整性。Xilinx 公司的 FPGA 结构最适合采用同步设计。同步设计主要涉及以下几个方面：

a. 复位的使用(6.2 节)；
b. 时钟的使用(6.3 节)；
c. 接口异步信号处理(6.4 节)；
d. 寄存器和锁存器的选用(6.5 节)；
e. 有限状态机设计(6.6 节)；

f. 输入输出信号的寄存器同步。

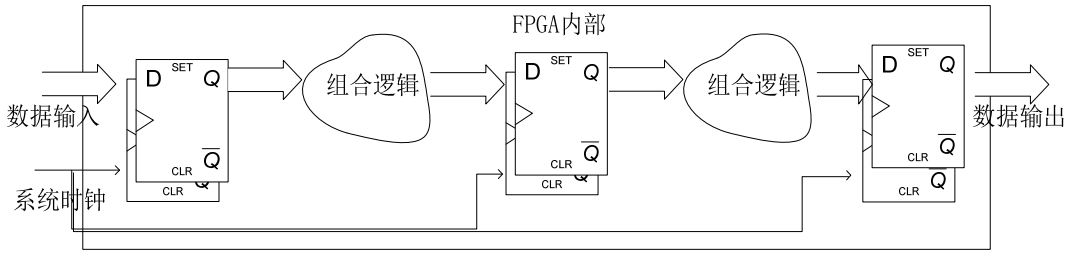


图 2 输入输出信号的寄存器同步

6.2 复位

6.2.1 上电复位时间要求

复位的目的是使 FPGA 内部寄存器和 IO 进入一个预先设计的状态。复位分上电复位和一般复位。

FPGA 器件初始加电时需要经过一个上电过程，上电复位才能够起作用，因为 FPGA 器件上电时间受上电速度影响较大，所以上电复位时间必须有设计余量。

表 1 FPGA 器件上电时间(参考值)

	慢上电	快上电	上电时间(慢速/快速)
Actel	0.2V/ms	0.5V/ μs	25ms/0.01ms
Xilinx 以 virtex 器件为例	0.1V/ms	2.5V/ms	50ms /2ms

对于 Xilinx 器件，在器件上电后还要经过启动配置和配置等过程上电复位才能够起作用，启动配置时间(T_{PI})最大 2ms。以 18V04 配置 PROM(4Mbits)为例，最大串行配置时钟频率是 60MHz，默认是 4MHz，那么配置时间默认是 1s，最少需 66ms。

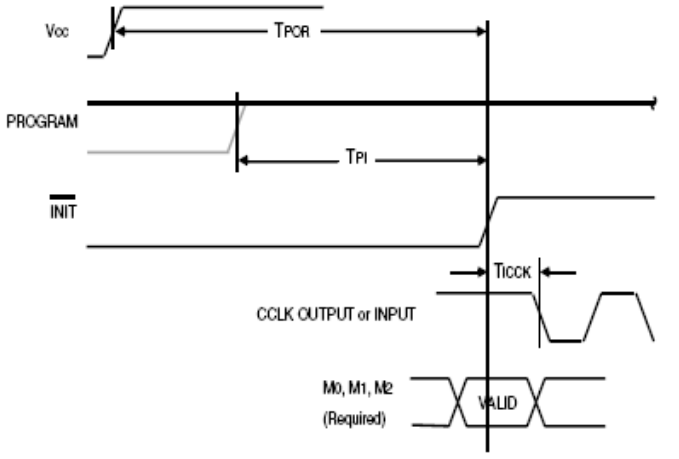


图 3 Xilinx 器件启动配置时序

大部分 FPGA 设计引入了时钟，那么上电复位时间还必须大于晶振上电稳定输出时间，一些晶振上电稳定输出时间的典型值为 10ms。建议设计的上电复位时间足够长，满足 FPGA 上电、配置时间要求及晶振上电时间要求，最好有 50%的设计余量。

6.2.2 FPGA 上电及配置时的 IO 特性

在 FPGA 的上电过程中，器件 IO 的表现也需要引起设计者注意。Actel 的 FPGA 在上电过程中大部分器件的 IO 表现为三态，但是部分器件 IO 表现为输出高或低。Xilinx 的 FPGA 由于器件本身的特性，内部的 GSR 信号在配置结束时自动把所有的寄存器和锁存器按设计要求复位或是置位(Xilinx 推荐保留外部输入的复位信号的复位作用，在 FPGA 配置结束后输入的复位信号依然有效)，并且在配置时，GTS 信号控制所有的 IO 都表现为三态。因为复位的作用在 FPGA 上电和配置期间并不能得到体现，即 FPGA 的 IO 表现为三态或输出而不是设计状态，如果对外有影响就需要根据具体情况处理，去除影响。

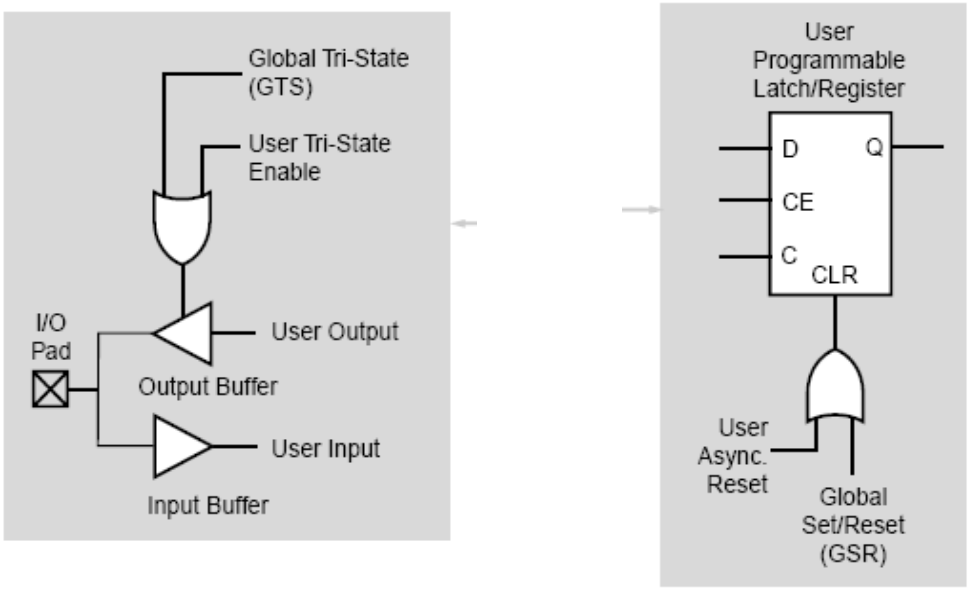


图 4 Xilinx 器件内部 GTS 和 GSR 结构图

目前 Actel 和 Xilinx 都不提供固定复位信号管脚，Actel 从 MX 系列开始提供了可以作为异步或同步复位信号网络使用的内部高速布线网络，Xilinx 的 Virtex 系列等也提供此种功能。

6.2.3 FPGA 输入复位信号及内部存储器宏单元上的异步/同步复位端口的使用

一般来说数字系统内的复位信号是与时钟信号无时序关系的异步复位，实际上就是一个特殊的接口异步信号。不管是在 FPGA 外对异步复位进行同步，还是在 FPGA 内对异步复位进行同步(如采用图 5 的方式)，都可以产生同步复位信号。需要注意的是在上电时，时钟源上电输出稳定和 FPGA 上电及配置完成都是同步复位有效的前提条件。同步复位结束后可以保证复位状态安全退出；在异步复位信号沿和时钟沿有竞争冒险的情况下，可能导致异步复位无效。建议对 FPGA 的输入复位信号进行同步，如果对输入复位信号进行同步，复位时间还必须长于若干个时钟周期的长度。

本节的同步复位和异步复位是以时钟信号和复位信号是否有时序关系来区分，一般代码中描述的同步复位和异步复位是以复位信号是否需要时钟信号的有效沿变化才能对被复位信号起到复位作用来区分，如果不需要时钟信号沿有效则是异步复位(代码实现后是类似图 5 的(a)寄存器宏单元)，否则是同步复位(代码实

现后是类似图 5 的(b)寄存器宏单元)。下面的例子在 FPGA 内部产生了同步复位信号和实现了带异步复位的寄存器和带同步复位的寄存器。

在对输入复位信号进行同步后，在写代码时采用类似图 5(a)寄存器宏单元还是图 5(b)寄存器宏单元，可根据 FPGA 的内部结构进行选择。因为 Xilinx 器件的结构决定了实现同步复位的灵活性，Xilinx 推荐采用带同步复位的存储器；Actel 的 XL、DX、MX、SX 和 ACT 3 系列器件内部的寄存器都有固定的异步复位输入端口，内部结构可参考图 29，对于 Actel 的这些器件推荐采用带异步复位的存储器。

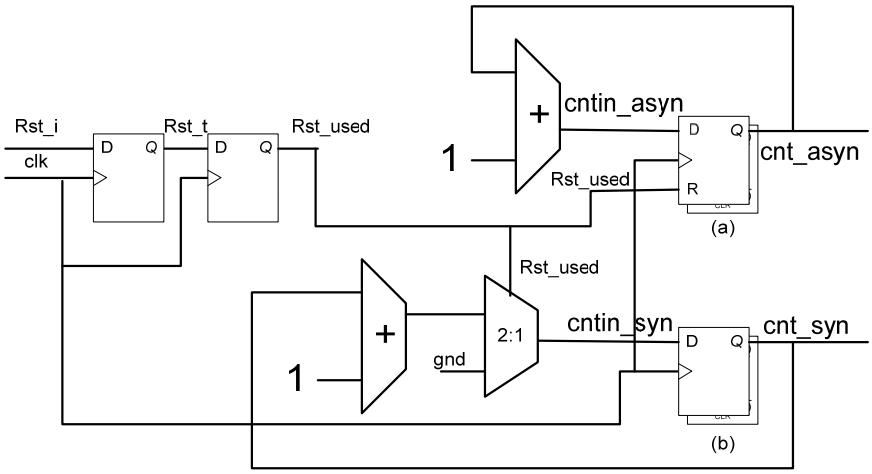


图 5 内部产生同步复位信号及带异步复位的寄存器和带同步复位的寄存器

--rst_i 是输入的异步复位信号

--rst_used 是产生的同步复位信号，在 FPGA 内部使用

```
process (clk)
 begin
    if (clk'event and clk = '1')  then
      rst_t <=   rst_i  ;
      rst_used <=   rst_t  ;
    end if;
  end process;
```

--内部采用带异步复位的寄存器，复位信号是经过同步的。

```
process (clk, rst_used)
begin
     if (rst_used = '0') then
            cnt_asyn <= (others => '0');
     elsif (clk'event and clk = '1') then
            cnt_asyn <= cntin_asyn;
```

```
        end if;
    end process;
process (cnt_asyn)
begin
        cntin_asyn <= cnt_asyn + 1 ;
end process;

--内部采用同步复位的寄存器，复位信号是经过同步的。
process (clk)
begin
      if (clk'event and clk = '1') then
             cnt_syn <= cntin_syn;
      end if;
    end process;

  -- RST_V 是有效复位电平的常量定义
process (cnt_syn, rst_used)
begin
        cntin_syn <= cnt_syn + 1 ;
      if (rst_used = RST_V) then
           cntin_syn   <= (others => '0');
      end if;
end process;
```

另外一个问题是存储器件是否需要复位需要具体分析，不能随意添加复位信号。例如有些信号的输入采样寄存器，添加复位信号可能没有坏处；但是有些逻辑不需要复位，添加后会画蛇添足，如 Xilinx 的一个 LUT 只能实现不带复位的移位寄存器，如需复位功能，需 2 个 LUT 实现降低了移位寄存器的时钟速度，增加了面积。

6.3 时钟

时钟到达 FPGA 内部不同寄存器相对的时间差别叫做时钟偏移。

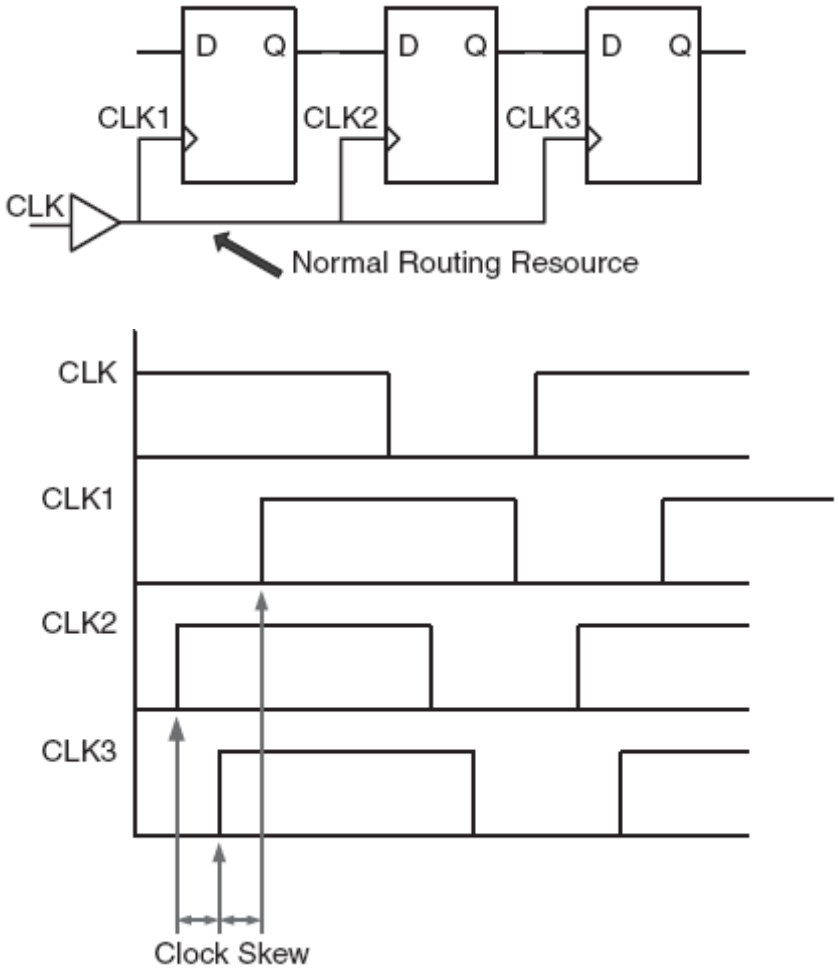


图 6 时钟偏移示意图

时钟到达 FPGA 内部两个相邻的寄存器时间差较大而数据路径较短时，有可能带来功能失效，这种问题一般发生在最好情况下(低环境温度、高输入电压)。

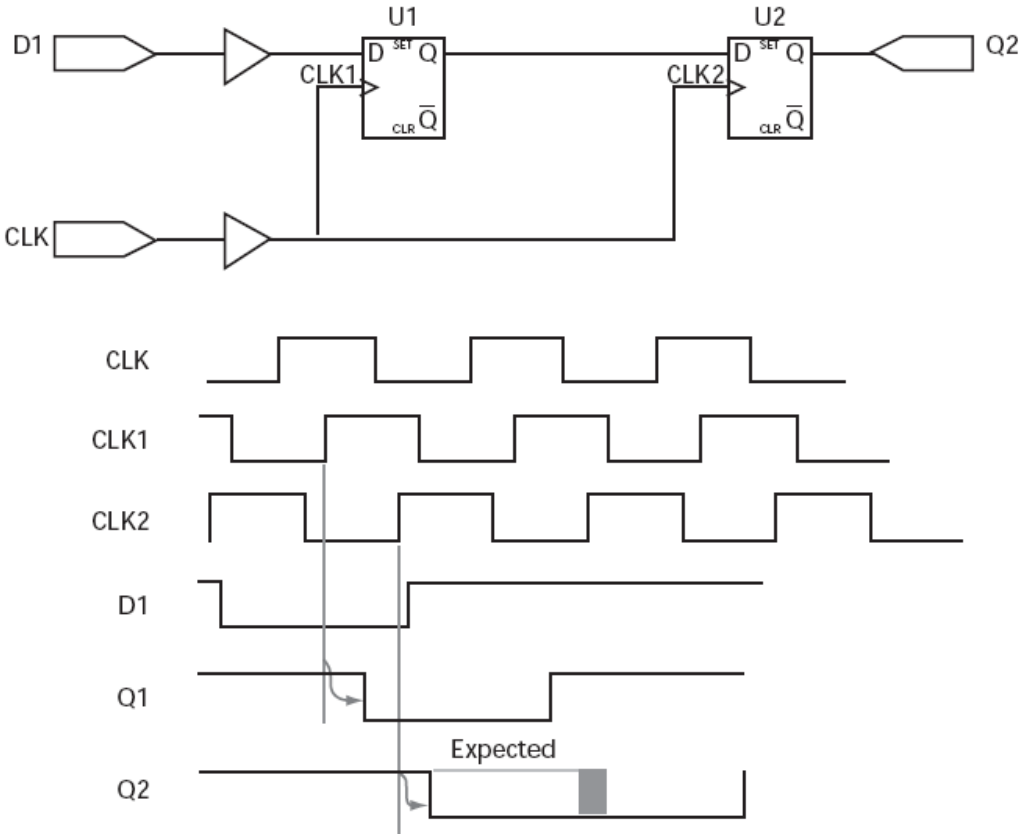


图 7 时钟偏移导致功能失效的示意图

一般使用 FPGA 内部的硬连时钟网络不会出现上述的问题，硬连时钟网络的 Skew 一般在 0.1ns~2ns 量级，越先进的器件提供的高速低 Skew 的网络越多。如果设计中时钟较多，而时钟网络资源不够，则必须考虑时钟偏移问题。如果综合工具不能自动把高扇出的网络例化到 FPGA 内部预布的网络上，就需要设计者手工例化。在确实有可能发生问题的最短路径上，可以通过加入 buffer 等方式处理。相关的时钟信号的 Skew 和最短路径可以通过 FPGA 布局布线工具里的静态时序分析工具进行检查。

如果需要对外部输入时钟进行倍频、降频或者相位变化等处理，在 FPGA 内部有相关时钟管理资源时，建议调用 FPGA 内部的时钟管理资源来完成。

建议只使用时钟上升沿，便于 FPGA 实现。

在多时钟时，要正确处理时钟域之间的接口，详见 6.3 节。

建议不使用 FPGA 内部组合逻辑产生的信号作为时钟，在有降低功耗需求必须对时钟控制时，采用时钟使能的方式而不是门控时钟，因为门控时钟可能带来时钟信号上的毛刺。

下面是门控时钟的例子：

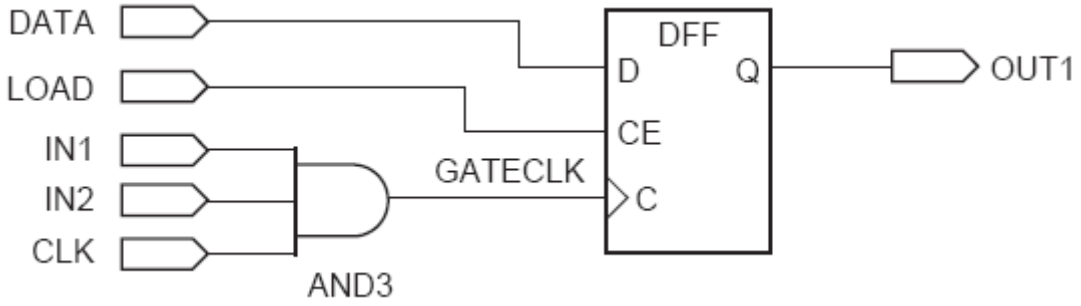


图8 门控时钟示意图

```
gateclk <= (in1 and in2 and clk);
process (gateclk)
begin
if (gateclk'event and gateclk = '1') then
   if (load = '1') then
      out1 <= data;
   end if;
end if;
end process;
```

下面是时钟使能的例子：

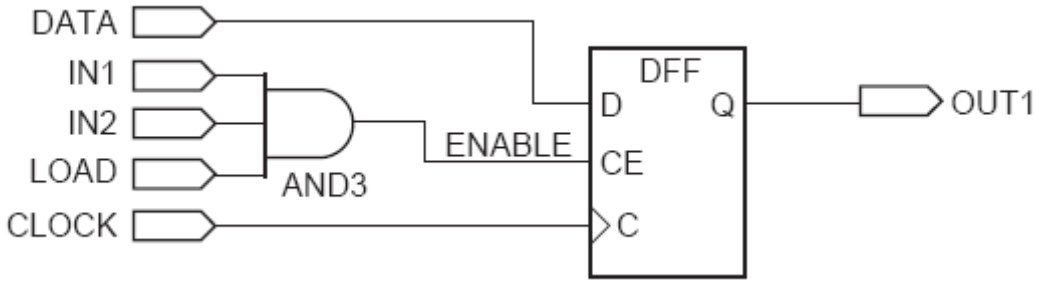


图 9 时钟使能示意图

```
enable <= in1 and in2 and load;
process
begin
if (clock'event and clock = '1') then
   if (enable = '1') then
      out1 <= data;
   end if;
end if;
end process;
```

6.4 接口异步信号处理

接口异步信号处理是为了减少亚稳态影响的传播。亚稳态现象就是如果 FPGA 中寄存器输入信号不满足建立时间的要求，正常的输出信号就会在一段不

可预知长度的时间内处于一个非“0”非“1”的中间状态(这段时间称为亚稳态恢复时间)，之后可能稳定在“1”状态，也可能稳定在“0”状态。

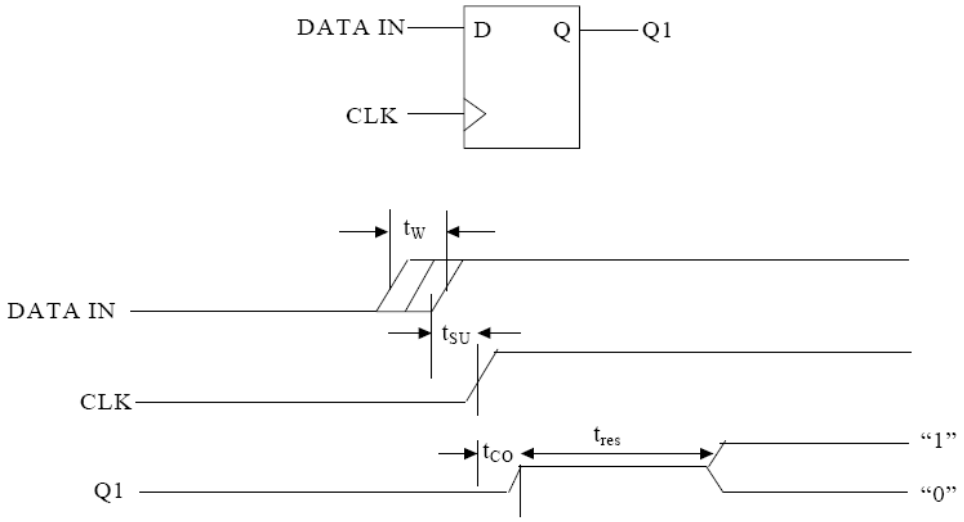


图 10 亚稳态时序示意图

注：T_w 寄存器输入数据变化可能形成亚稳态现象的时间窗口

T_{SU} 输入相对于时钟沿所需的建立时间

T_{CO} 输出相对于时钟沿之后的延迟时间

T_{res} 亚稳态恢复时间

亚稳态影响表现在如果发生亚稳态的信号直接参与到不同路径长度的逻辑中，该信号有可能在一个路径上表现为逻辑 1，而在另一个路径上表现为逻辑 0。如图 11 所示，图中的信号 A 和 B，逻辑上本应是两个逻辑相反的信号，而实际上由于亚稳态的原因有时会成为相同逻辑电平的信号。

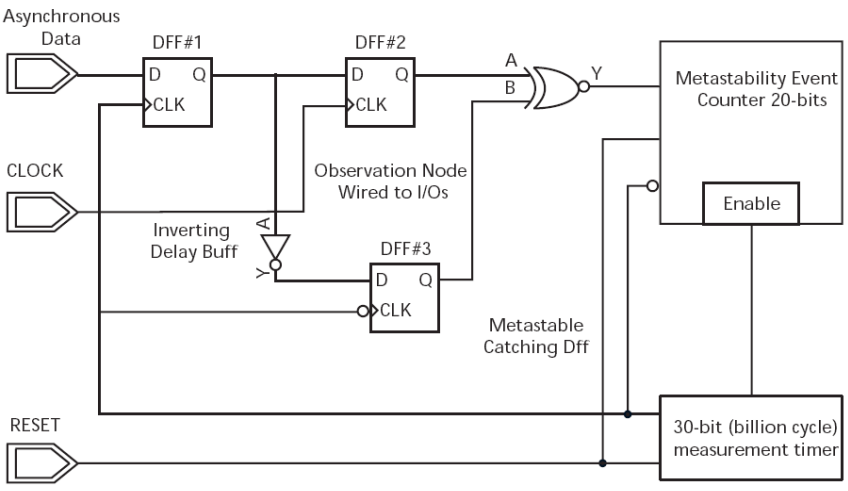


图 11 亚稳态试验示意图

亚稳态现象是FPGA的一个固有特点，并且由于亚稳态恢复时间的不确定性，不能根本的去除亚稳态的影响。通过在设计时预留一定的亚稳态恢复时间(预留的亚稳态恢复时间用 T_{met} 表示)，保证在绝大多数情况下，信号可以在 T_{met} 时间内从亚稳态中恢复之后才参与到后面的逻辑运算中，把亚稳态影响减少到一个可以接受的范围。在一定的 T_{met} 下，平均无故障时间和 FPGA 的工艺(C_1、C_2)、异步数据变化的频率(f_d)、时钟频率(f_c)有关系。FPGA 厂商通过实验的方法得到经验数据，提供了一个 MTBF 的计算公式和部分工艺因子(C_1、C_2)的数据，可以通过这个公式和实际情况计算看是否满足可靠性的要求。

$$\ln(MTBF)=C_2*T_{met}-\ln(C_1*f_d*f_c)$$

一个简单的同步接口异步信号和预留亚稳态恢复时间的方法如图 12 所示，T_{met} 约等于时钟周期，一般可以满足当前可靠性的要求，实际应用还要根据参考相关器件应用说明进行具体的估计。

另外建议采样时钟频率至少高于数据变化频率 2 倍；同时必须注意图 12 的方式仅仅是异步信号的处理方式，如果要保证异步数据交换的有效性，必须遵循一定的接口约定，如图 24 中的握手方式。

如果亚稳态恢复时间相对于时钟周期来说相当的小，那么图 12 中第二个寄存器可以去掉，如 Virtex-II 采用的工艺使得 Virtex-II 器件的亚稳态恢复时间非常短，时钟频率在 200MHz 以下时可以不预留亚稳态恢复时间，即此时就可以去掉图 12 中的第二个寄存器。

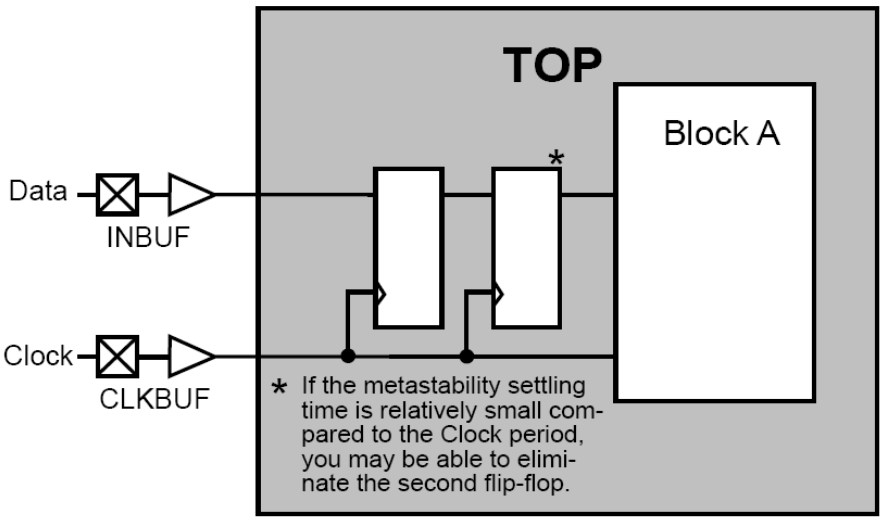


图 12 使用两个同步寄存器处理异步信号

6.5 寄存器和锁存器

锁存器是电平敏感存储单元，寄存器是边沿触发存储单元。在基于寄存器的设计里，竞争冒险情况较少甚至不存在，控制信号和寄存器输入信号上的毛刺不易造成影响，信号可在一个周期里准确的锁存，推荐使用寄存器而不是锁存器。

寄存器在 VHDL 源码中一般使用 if 语句实现，条件判断常见的有以下四种，一般使用上升沿有效判断表达式即可：

a. (clk'event and clk='1') –上升沿有效；

b. (clk'event and clk='0') –下降沿有效；

c. rising_edge(clk) –上升沿函数调用；

d. falling_edge(clk) –下降沿函数调用。

下面是一个带异步复位的 D 寄存器的 VHDL 源码的示例。

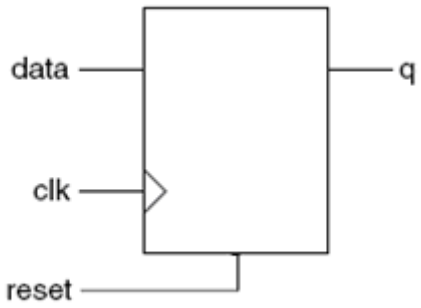


图 13 带异步复位的 D 寄存器

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff_async_rst is
port (
      data, clk, reset : in std_logic;
      q : out std_logic);
end dff_async_rst;
architecture behav of dff_async_rst is
begin
   process (clk, reset) begin
      if (reset = '1') then
         q <= '0';
      elsif (clk'event and clk = '1') then
         q <= data;
      end if;
   end process;
end behav;
```

下面是一个带同步复位的 D 寄存器的 VHDL 源码的示例。

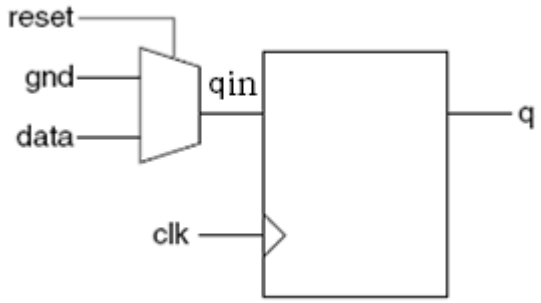


图 14 带同步复位的 D 寄存器

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff_sync_rst is
port (data, clk, reset : in std_logic;
q : out std_logic);
end dff_sync_rst;
architecture behav of dff_sync_rst is
   signal qin : std_logic;
begin
 process (reset, data)
 begin
      qin <= data;
      if (reset = '1') then
           qin <= '0';
      end if;
 end process;

 process (clk) begin
      if (clk'event and clk = '1') then
           q <= qin ;
      end if;
 end process;
end behav;
```

下面是一个带异步复位的锁存器的VHDL源码的示例。

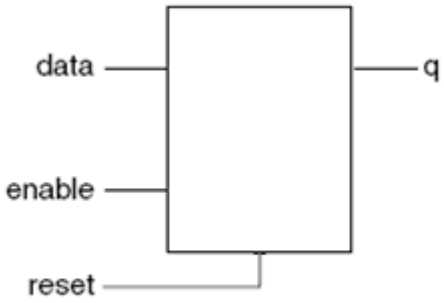


图15 带异步复位的锁存器

```
library IEEE;
use IEEE.std_logic_1164.all;
entity d_latch_rst is
        port (enable, data, reset: in std_logic;
        q :out std_logic);
end d_latch_rst;
architecture behav of d_latch_rst is
begin
process (enable, data, reset) begin
        if (reset = '1') then
                q <= '0';
        elsif (enable = '1') then
                q <= data;
        end if;
end process;
end behav;
```

6.6 有限状态机

不管是复杂的设计还是简单的设计，有限状态机都是最常采用的同步设计方法。对于 FPGA 实现来说，建议采用多个小的状态机实现大的状态机(比如 32 个状态以上)。有限状态机最基本的模型有两种：米兰机和摩尔机，摩尔有限状态机的输出仅与当前状态有关。米兰有限状态机的输出则与当前状态和输入有关。一个有限状态机由三部分组成：

a. 当前状态寄存器组：如果用一组 n 位的寄存器保存当前状态，状态机就可以有 2^n 个状态，保存的向量也就是状态编码。一个 m 个状态的有限状态机至少需要 $\log_2(m)$位状态寄存器。

b. 次态生成组合逻辑：一个有限状态机在任意时刻只能有一个状态，次态

生成组合逻辑将决定每个有效的时钟沿变化时将发生的状态转换。次态是由有限状态机的输入及其当前状态决定的。

c. 输出组合逻辑：输出通常是当前状态和有限状态机输入的函数。

摩尔有限状态机和米兰有限状态机如图 16、图 17 所示。

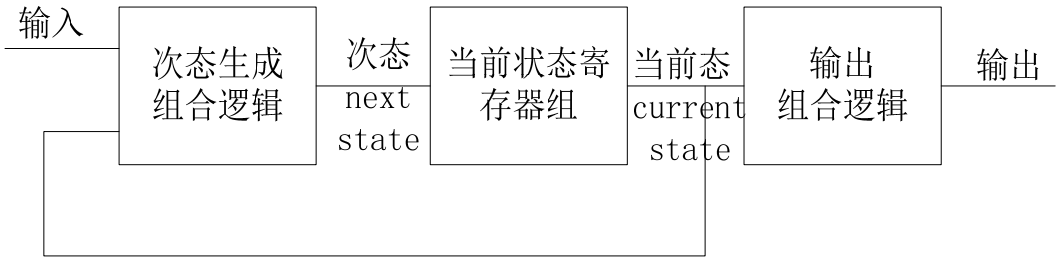


图 16 摩尔有限状态机示意图

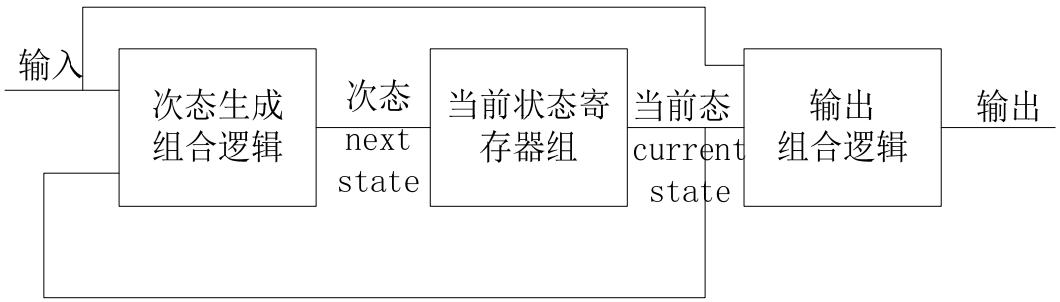


图 17 米兰有限状态机示意图

下面是一个米兰有限状态机的 VHDL 源码的示例，状态变化和输出见图 18。

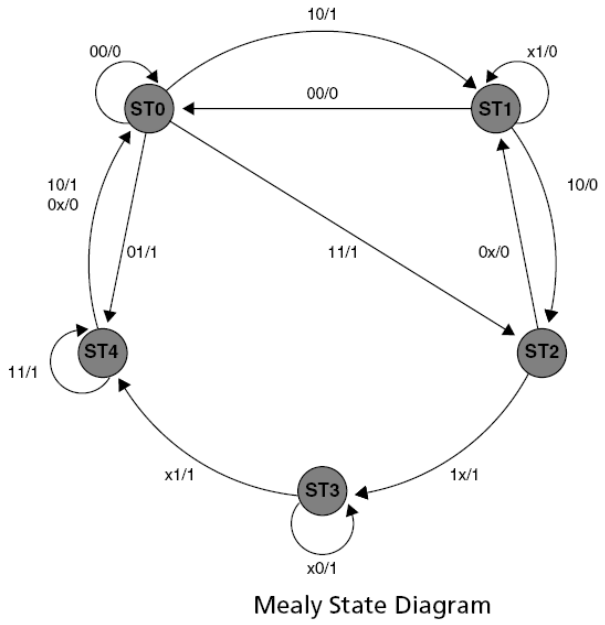


图 18 米兰有限状态机状态迁移图

```
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 curr_state, next_state: state_values;
begin
-- FSM register
statereg: process (clock, reset)
begin
        if (reset = '1') then
                curr_state <= st0;
        elsif (clock'event and clock ='1') then
                curr_state <= next_state;
        end if;
end process statereg;
-- FSM combinational block
fsm: process (curr_state, data_in)
begin
case curr_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 curr_state w/ data_in
outputs: process (curr_state, data_in)
begin
case curr_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;
```

下面是一个摩尔有限状态机的 VHDL 源码的示例。

```
library ieee;
use ieee.std_logic_1164.all;
entity moore is
port (clock, reset: in std_logic;
        data_out: out std_logic;
        data_in: in std_logic_vector (1 downto 0));
end moore;
architecture behave of moore is
        type state_values is (st0, st1, st2, st3, st4);
        signal curr_state, next_state: state_values;
begin    -- FSM register
statereg: process (clock, reset)
begin
if (reset = '1') then
        curr_state <= st0;
elsif (clock'event and clock ='1') then
        curr_state <= next_state;
end if;
end process statereg;
-- FSM combinational block
fsm: process (curr_state, data_in)
begin
case curr_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;
-- Moore output definition using curr_state only
outputs: process (curr_state)
begin
case curr_state is
          when st0 => data_out <= '1';
          when st1 => data_out <= '0';
          when st2 => data_out <= '1';
          when st3 => data_out <= '0';
          when st4 => data_out <= '1';
          when others => data_out <= '0';
end case;
end process outputs;
end behave;
```

6.7 使用 IF-Then-Else 的有优先权的译码器

一般情况下不使用优先级译码，仅在对速度影响较大的关键路径使用。使用 If-then-else 语句可以给一个迟来的信号优先权，即把它放在输出级参与逻辑运算使得整个路径的速度达到最优。如图 19 中的信号 C 是最慢的信号，采用 If-then-else 语句的例子如下。

另外例子中需要说明的是，对所有在 If 语句里赋值的信号，为避免综合时产生无用的 latch，通常采用在前面赋值的方式。

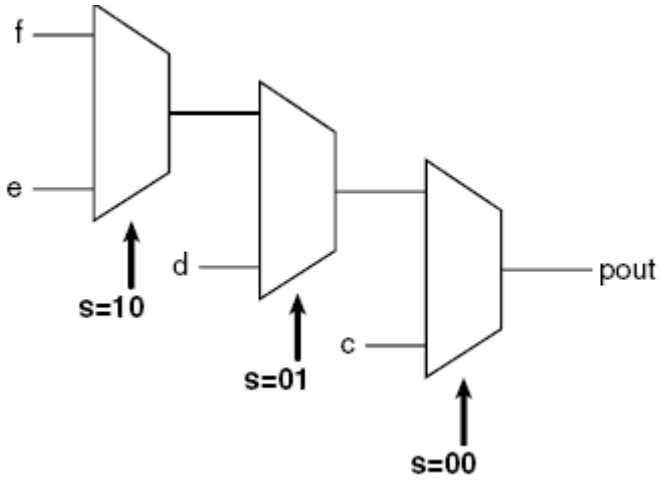


图 19 带优先权的译码器

```
library IEEE;
use IEEE.std_logic_1164.all;
entity my_if is
   port (
      c, d, e, f: in std_logic;
      s :in std_logic_vector(1 downto 0);
      pout : out std_logic
      );
end my_if;
architecture my_arc of my_if is
   begin
      myif_pro: process (s, c, d, e, f)
      begin
         pout <= f;
         if s = “00” then
            pout <= c;
         elsif s = “01” then
            pout <= d;
         elsif s = “10” then
            pout <= e;
         end if;
end process myif_pro;
end my_arc;
```

6.8 使用 Case 语句的多路复用

Case 语句必须包括所有可能的情况或者有一个缺省值用来在没有选中任何一个状态时执行，下面是 4 选 1 多路器的例子。

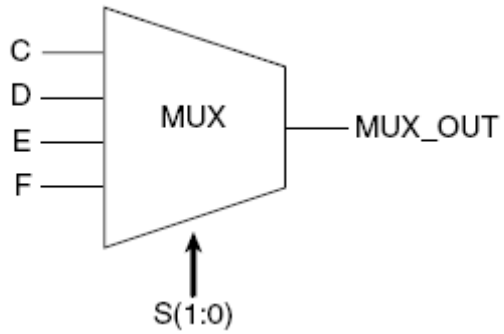


图 20 4 选 1 多路器

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux is
port (C, D, E, F : in std_logic;
            S :in std_logic_vector(1 downto 0);
            mux_out : out std_logic);
end mux;
architecture my_mux of mux is
begin
mux1: process (S, C, D, E, F) begin
            muxout <= F;
            case s is
                        when “00” => muxout <= C;
                        when “01” => muxout <= D;
                        when “10” => muxout <= E;
                        when others => muxout <= F;
          end case;
end process mux1;
end my_mux;
```

6.9 解码器

下面是一个带使能信号的 3-8 解码器的 VHDL 源码的示例。

```
library IEEE;
use IEEE.std_logic_1164.all;
entity decode is
port ( ain : in std_logic_vector (2 downto 0);
            en: in std_logic;
            yout : out std_logic_vector (7 downto 0));
end decode;
architecture decode_arch of decode is
begin
process (ain)
```

```
begin
        yout <= (others => '0');
        if (en='1') then
                case ain is
                        when "000" => yout <= "00000001";
                        when "001" => yout <= "00000010";
                        when "010" => yout <= "00000100";
                        when "011" => yout <= "00001000";
                        when "100" => yout <= "00010000";
                        when "101" => yout <= "00100000";
                        when "110" => yout <= "01000000";
                        when "111" => yout <= "10000000";
                        when others => yout <= "00000000";
                 end case;
      end if;
end process;
end decode_arch;
```

6.10 计数器

计数器用来计数一个随机发生或者有规律发生的事件。在设计中使用“+”就可以让综合工具生成一个计数器，但是综合工具往往对较大(超过 8 位)的计数器综合效果不好。如果在关键路径需要使用较大的计数器，推荐使用 FPGA 的计数器宏单元。下面是带有使能信号、置数、异步复位的 8 位加计数器的例子。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity counter8 is
port (clk, en, n_rst, load : in std_logic;
        data: in std_logic_vector (7 downto 0);
        count : out std_logic_vector (7 downto 0));
end counter8;
architecture behav of counter8 is
          signal cntin: std_logic_vector (7 downto 0);
          signal cnt: std_logic_vector (7 downto 0);
begin

process (en, cnt, load, data)
begin
          cntin <= cnt；
```

```
            if (load = '1') then
                cntin <= data ;
            elsif (en = '1') then
                cntin <= cnt + '1';
            end if;
end process;

process (clk, n_rst)
begin
        if (n_rst = '0') then
                cnt <= (others => '0');
        elsif (clk'event and clk = '1') then
                cnt <= cntin;
        end if;
end process;

count <= cnt;
end behav;
```

6.11 运算

操作	操作符
算术运算 加 减 不常用运算(乘、除、指数、求模、求余数、绝对值)	 + - * 、 / 、** 、mod 、rem 、abs
位连接	&
比较 等于 不等 不常用运算(大于、小于、大于等于、小于等于)	 = /= > 、< 、 >= 、<=
移位运算 逻辑左移 逻辑右移 逻辑循环左移位 逻辑循环右移位 算术左移位 算术右移位	 sll srl rol ror sla sra
逻辑运算 **与、或、非等**	 And 、or 、 not、 nand、 nor、 xor、 xnor

以下是比较运算的例子。

```
library IEEE;
use IEEE.std_logic_1164.all;
entity equality is
port (
a: in std_logic_vector (3 downto 0);
b: in std_logic_vector (3 downto 0);
q1, q2, q3, q4 : out std_logic
);
end equality;
architecture equality_arch of equality is
begin
   q1 <= '1' when a =b else '0';
   q2 <= '1' when a/=b else '0';
   q3 <= '1' when a <=b else '0';
   q3 <= '1' when a >b    else '0';
end equality_arch;
```

以下是左移运算和右移运算的例子：

```
library IEEE;
use IEEE.std_logic_1164.all;
entity shift is
   port (
   data : in std_logic_vector(3 downto 0);
   q1, q2 : out std_logic_vector(3 downto 0)
   );
end shift;
architecture rtl of shift is
   begin
   process (data)
      begin
         q1 <= data(1 downto 0) & “10”; --  逻辑左移
         q2 <= “10” & data(3 downto 2); --  逻辑右移
   end process;
end rtl;
```

6.12 IO

IO 应该在顶层文件例化或描述。

下面是一个三态输出 IO 的例子。

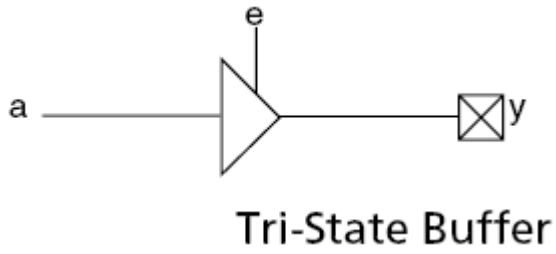


图 21 三态输出 IO

```
library IEEE;
use IEEE.std_logic_1164.all;
entity tristate is
port (e, a : in std_logic;
y : out std_logic);
end tristate;
architecture tri of tristate    is
begin
y <= a when (e = '1') else 'Z';
end tri;
```

下面是一个 OC 输出 IO 的例子。

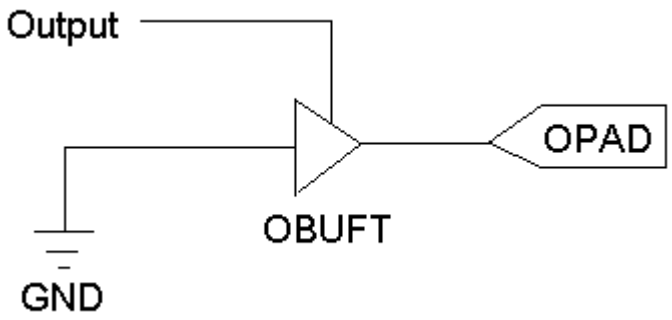


图 22 OC 输出 IO

```
library IEEE;
use IEEE.std_logic_1164.all;
entity oc_out is
port (output : in std_logic;
opad : out std_logic);
end oc_out;
architecture oc of oc_out    is
begin
opad <= 'Z' when output = '1' else '0';
end oc;
```

下面是一个双向 IO 的例子。

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;
entity bidir is
port (e, a : in std_logic;
b : out std_logic;
y : inout std_logic);
end bidir;
architecture bi of bidir is
begin
y <= a when (e = '1') else 'Z';
b <= y;
end bi;
```

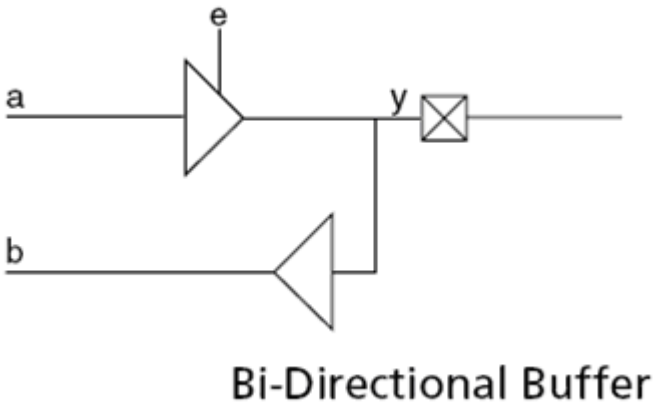


图 23　双向 IO

6.13　异步设计

一般采用异步设计的原因有以下几种：

a. 系统要求，如采用异步总线协议 VME；

b. 满足速度要求；

c. 满足功耗要求。

异步设计的一般要求如下：

a. 异步接口一般采用握手、中断、状态查询等方式保证数据交换的有效性；

b. 设计中需要保证时序要求，保证建立时间、保持时间和数据稳定，保证设计余量。

FPGA 内部异步(不同时钟域)的两个模块或 FPGA 和外部的异步系统接口，在主动数据交换时，可采用握手的方式保证数据有效性，图 24 中 Strobe A-B 信号和 Strobe B-A 信号有效的含义可以是数据准备好或者请求对方发送数据。

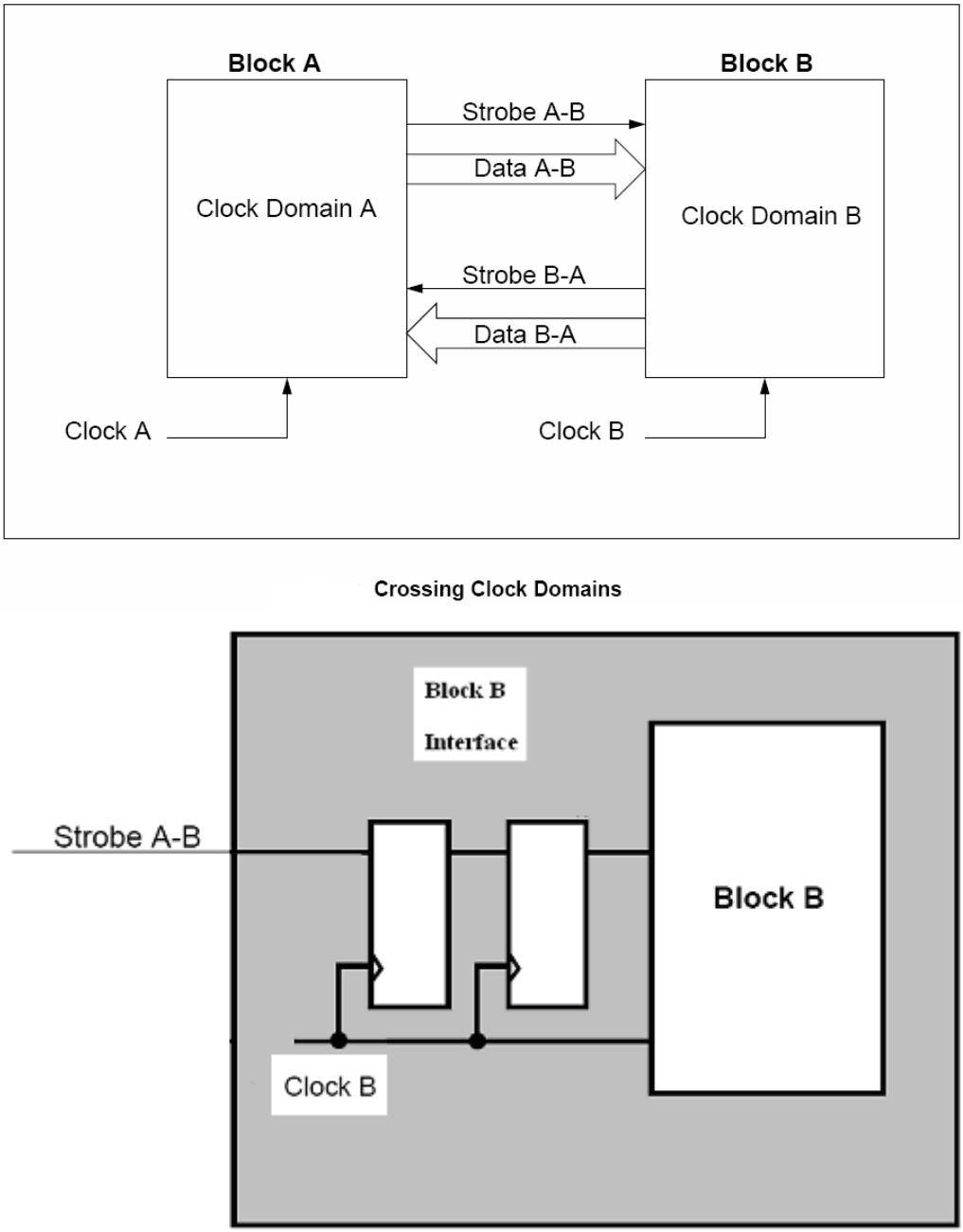


图 24 异步接口示意图

7 设计优化

由于 FPGA 工艺、综合工具的性能在变化，对 HDL 描述的要求、设计优化的要求也在变化，关键是实现后的结果是否能满足要求。例如 16 位计数器的实现，使用早期工艺的 FPGA(A1280XL)，连线资源、连线驱动能力和速度都可能对综合后的结果造成很大影响，可以考虑用两个 8 位同步计数器(FPGA 厂商提供的固核)级联实现；而较先进工艺的 FPGA (XCV600)实现时不需要特殊的处理

优化，因为实现后的速度足够满足要求。这点类似于早期为节省内存而采用汇编语言编程，当技术发展到内存容量已不成为障碍时，便可使用高级语言编程了。

当设计的性能与任务要求相差较大的情况下，选择速度等级更快的器件或更换性能更好的器件更能快速地解决性能不满足的问题。

针对 FPGA 的设计优化可简单的分为用户设计优化、工具设置优化。在 FPGA 设计中，设计优化往往针对的是关键路径的速度优化。

用户设计优化：

a. 标准单元实现；

b. 复杂逻辑单元共享；

c. 信号参与运算位置的选择，最慢的信号接近输出；

d. 针对目标 FPGA 的优化。

工具设置优化：

a. 由 HDL 转换成逻辑时，综合工具设置优化；

b. 由逻辑转换成门级网表时，综合工具设置优化；

c. 布局布线工具设置优化。

7.1 标准单元实现

一些基本的标准单元如加法、计数器等，一般是综合工具实现的，也可以使用 FPGA 器件厂商提供的库。复杂的标准单元可以自行设计，也可以采用商业 IP。

7.2 复杂逻辑运算单元共享

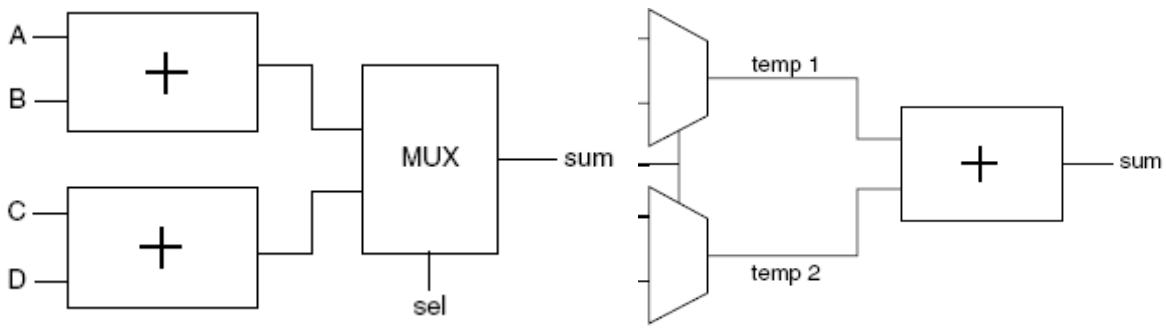


图 25 复杂逻辑运算单元共享示意图

如图 25 所示，复杂逻辑运算单元(加法)共享可以节省大量的 FPGA 内部资源。

7.3 中间信号

如图 26 所示中间信号是一个较慢的信号，需要经过 Tpa 的延时才能参与产生 pout 信号，那么如果把中间信号设计到 c 位置，那么产生 pout 信号的延时简单的计算就是 $Tpa+T_{c\text{-}pout}$，但是如果设计成 f 位置，那么产生 pout 信号的延时就是 $Tpa+T_{f\text{-}pout}$。一般进程间或模块间未经过寄存器的组合逻辑信号，都可以看成一个中间信号，再次参与到逻辑运算时都可能是一个慢速信号，需要设计时放在接近输出的位置。

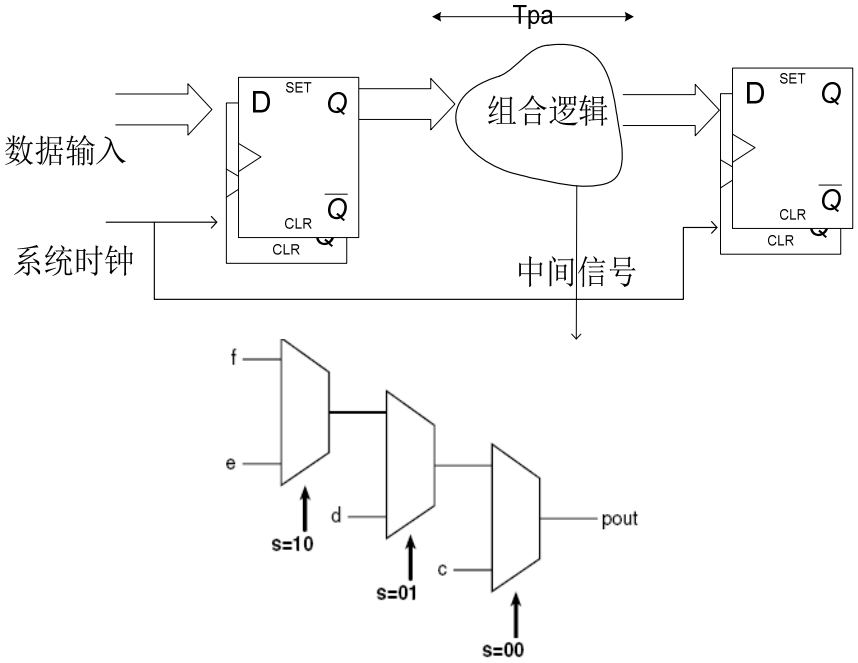


图 26 信号参与运算位置的选择示意图

7.4 针对目标 FPGA 的优化

在当前综合工具对行为级代码的综合结果还不是很令人满意的情况下，只有了解真实的硬件实现才能写出好的 HDL 编码。

FPGA 的内部资源主要是以下几种：

a. 逻辑单元结构；

b. 连线资源；

c. 特殊器件资源，如锁相环、内部 RAM/ROM、进位链等。

Actel 和 Xilinx 的 FPGA 的主要特点分别是：

a. Actel 组合逻辑功能受限制，但编程后延迟时间较固定；Xilinx 基于查找表(LUT)的结构，可实现任意 4 输入逻辑功能输出，但延迟时间可预测性差；

b. 对于 Actel 和 Xilinx 的 FPGA，在布局布线后查看的时序信息具有参考意义，而综合后查看的时序信息意义较小，因为布局布线对时序影响巨大；

c. 须注意 Actel 内部驱动能力固定有限，而 Xilinx 驱动能力较大；

d. Actel 和 xilinx 结构决定了编码时应尽量使用 case 语句综合生成多路器而不是 if-then-else 语句生成优先编码；

e. Actel 的器件不支持内部三态；Xilinx 的 Virtex 系列器件支持内部三态。

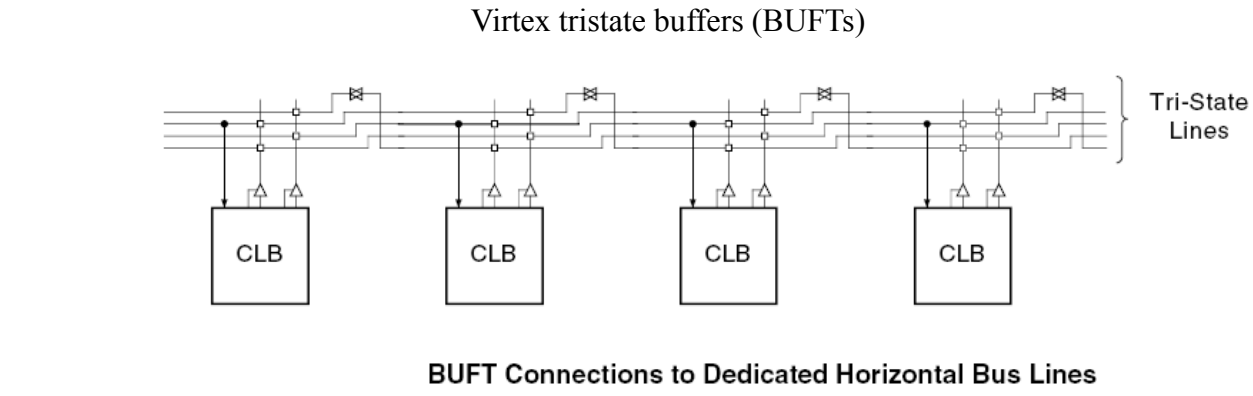


图 27 Virtex tristate buffers (BUFTs)结构示意图

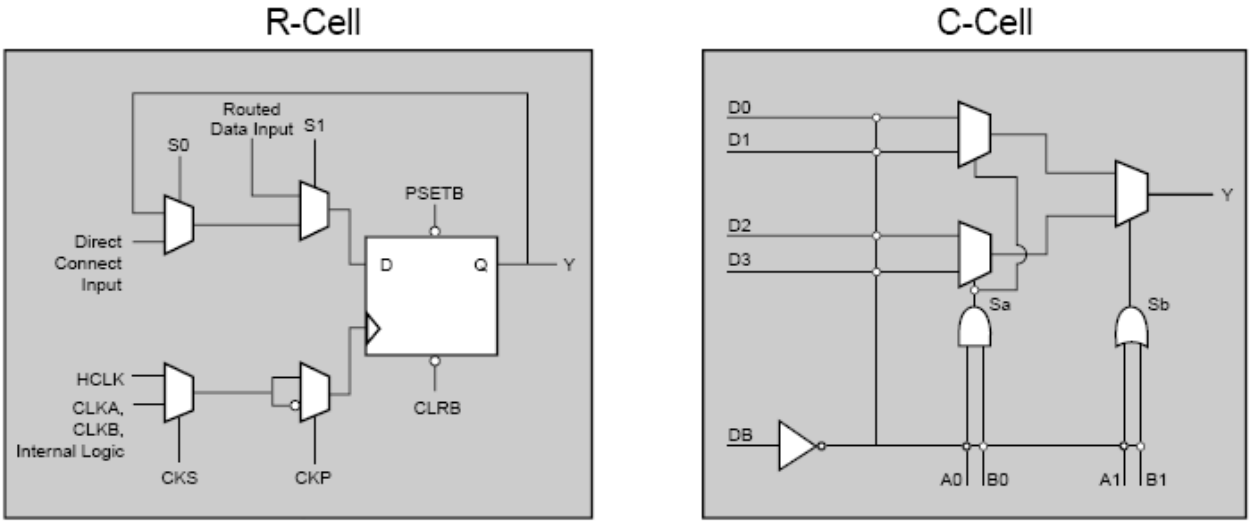


图 28 Actel 的 SX 系列逻辑单元结构示意图

图 28 是 Actel 的 SX 系列逻辑单元结构，内部寄存器单元和组合逻辑单元的比例大约为 1：2，特点如下：

a. 建议使用 Case 而不是 If-then-else，因为组合逻辑结构如图 28 所示更容易实现 Case 语句，面积和速度实现效率高；
b. 上升沿时钟、低电平复位、置位；
c. 寄存器资源丰富，可以采用 one-hot 状态机编码，可以采用移位寄存器替代计数器计数。

7.5 综合工具设置优化

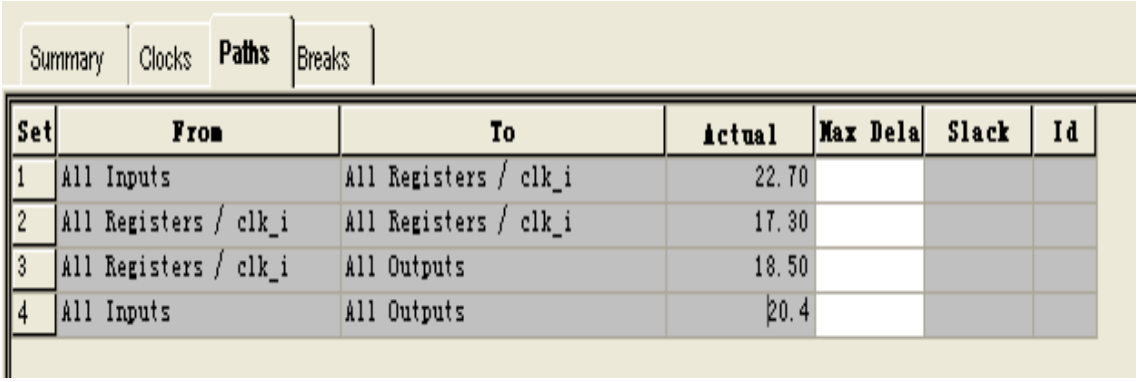
常见的综合工具选项设置如下：

1. 结构化(Hierarchy)和展平(Flatten)
   a. 可分别设置，根据综合结果选择。展平设置从逻辑上说好，但是目前综合工具的优化能力还有待提高。
   b. 原则是模块划分在寄存器处，综合优化一般只针对组合逻辑。
2. 有限状态机编码优化
   a. 一般情况下状态机编译设置使用 Fast 选项，但是在航天应用必须选择 Safe 编译模式，而此时设计的系统速度可能会降低；
   b. 一旦选择 Safe 编译模式，状态机编码可能就不是 One-hot 编码速度快 而是 Binary 编码速度快。但是 One-hot 编码海明距离是 2，Binary 编码海明距离是 1，如发生 SEU 可能更危险。如果选择海明距离为 3 的编码，可以检错纠错；
   c. 具体状态机编码的选择需要实现后根据可靠性和性能要求综合考虑。
3. 复杂逻辑共享
4. 寄存器复制设置

   为了满足扇出(Fanout)设置，有两种方法，一是采用插入 Buffer 的方式，另一种是寄存器复制(Register duplicate/register replication)的方式。为了尽可能的减小 SEU 带来的影响禁止选择寄存器复制的方式；
5. 路径

a. 静态时序分析是在路径的基础上进行，常见的路径如图 29 输入-输出、寄存器-寄存器、寄存器-输出、输入-寄存器；

b. 路径的延时计算的基础是器件的使用环境和器件等级。



Set	From	To	Actual	Max Dela	Slack	Id
1	All Inputs	All Registers / clk_i	22.70			
2	All Registers / clk_i	All Registers / clk_i	17.30			
3	All Registers / clk_i	All Outputs	18.50			
4	All Inputs	All Outputs	20.4			

图 29 静态时序分析路径示意图

6. Retiming、Pipeline

a. Synplify 综合工具可以在 Actel、Xilinx 的特定器件上作 Retiming 和 Pipeline；

b. Synopsys 综合工具只能在 Xilinx 的特定器件上作 Pipeline。

图 30 和图 31 中上半部分是优化前的示意图，下半部分是优化后的示意图。

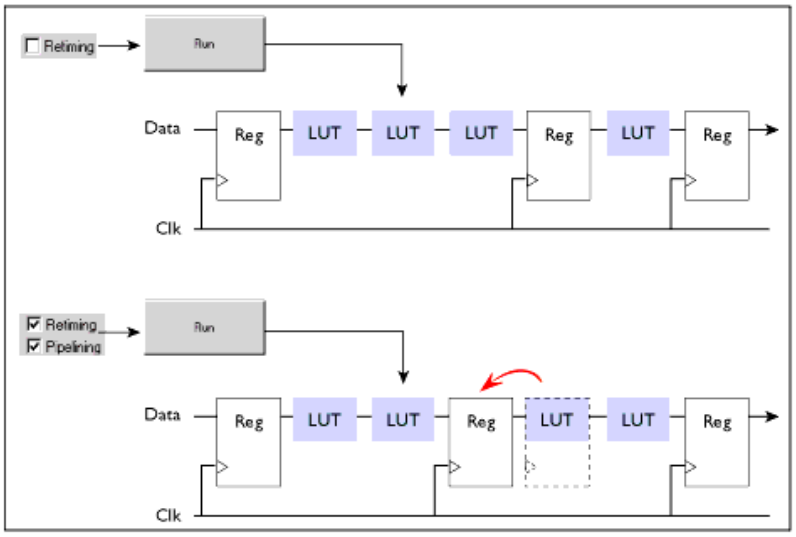


图 30 Retiming 示意图

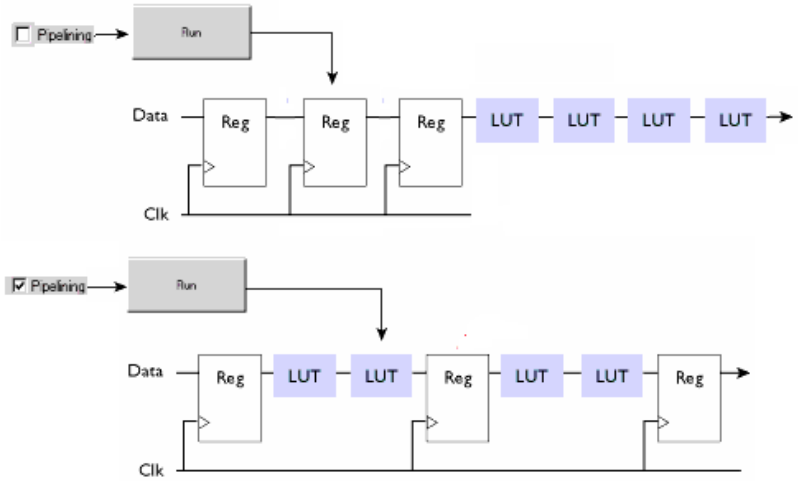


图 31 Pipeline 示意图

8 抗单粒子翻转效应设计

FPGA 航天应用必须预防 SEU 的影响，和商用 FPGA 设计不同之处主要有两点(在 7.5 节中也有相关的描述)：

a. 有限状态机的设计必须考虑无效状态的正确退出和状态错误跳转；

b. 禁止采用寄存器复制方式减少信号扇出。

由于 SEU 不是永久性故障，所以关键是发生 SEU 后系统是否可以容忍。如果系统可以容忍瞬时故障，那么只要发生 SEU 的寄存器可以恢复即可。简单的把 FPGA 内部寄存器分为控制寄存器和数据寄存器。数据寄存器的内容往往是经常刷新的，那么发生了 SEU 可能仅仅影响一次数据。控制寄存器设置完了往往很少改变，那么发生 SEU 后可能会一直影响系统，所以需要在设计时考虑保证控制寄存器的设置正确，如软件每次操作前都回读相关的控制寄存器内容进行检查或者每次操作前都设置一下控制寄存器。如果系统不能容忍关键的寄存器发生 SEU，就必须进行冗余设计。

对于 Actel 的 FPGA 目前常见的两种片内抗 SEU 加固策略有：

a. 对单独的寄存器进行三模冗余处理并周期性对寄存器刷新，从设计上作 TMR 的方法主要有两种，一是整个设计三模，所有的输出作表决；另一种是对所有的寄存器或重要的寄存器作 TMR。

b. 对一组数据(寄存器)进行冗余编码，例如常见的对 Memory 采用 EDAC 保护、对串行总线数据进行奇偶校验保护。

在此仅介绍利用工具对 Actel 的 FPGA 进行 TMR 设计的方法。如图 32 所示对于 Actel 的大部分 FPGA，Synplify 工具提供了 TMR 综合选项，设定的寄存器在综合后是 TMR；FPGA compiler II 工具则是提供相应的后处理方法，对正常综合的网表中的寄存器进行替换，达到 TMR 的目的，由于是网表替换所以所有 Actel 的 FPGA 都可以采用这种方法。

采用 TMR 方式进行抗单粒子翻转效应设计的缺点也是很明显的，主要是性能下降较大，输出信号可能有毛刺。

Actel Register Implementation Techniques

Device Families	C-C	TMR	TMR_CC
ACT1, 40MX	No	No	No
ACT2, ACT3, 42MX, 1200XL, 3200DX	Yes	Yes	Yes
54SX, 54SXA	No	Yes	No

图 32 Synplify 工具 C-C、TMR、TMR_CC 综合选项适用的器件

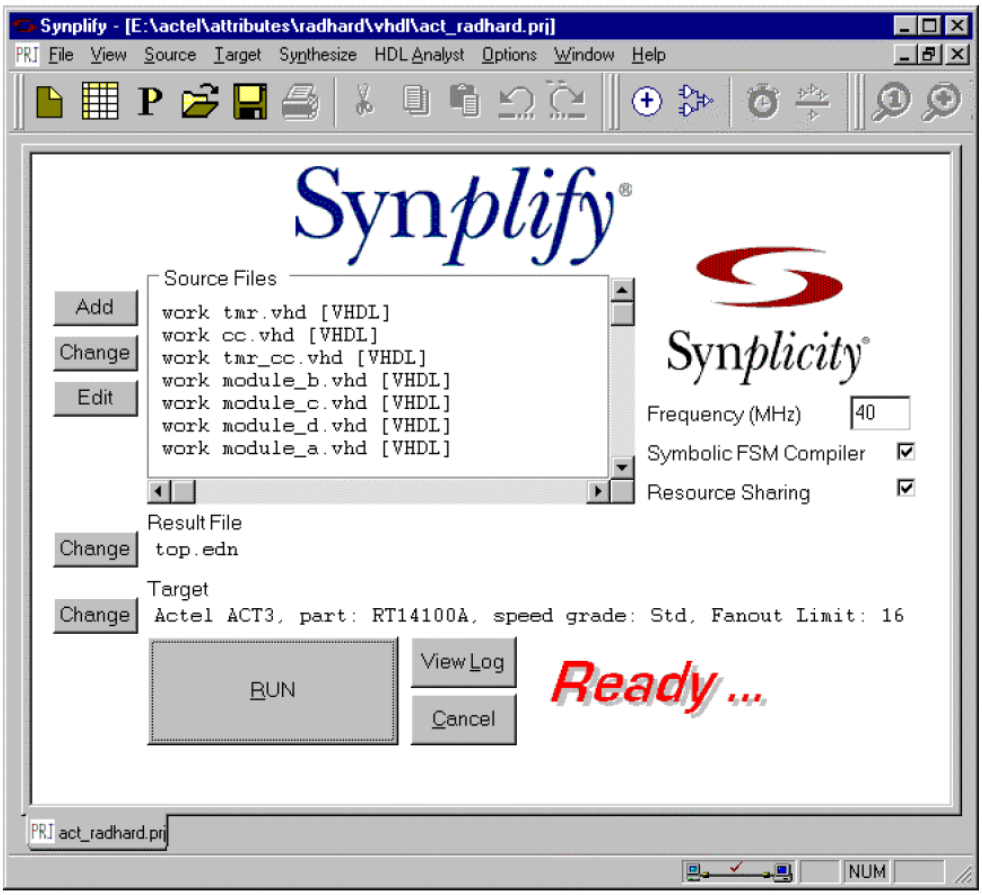


图 33 Synplify 工具 TMR 综合界面

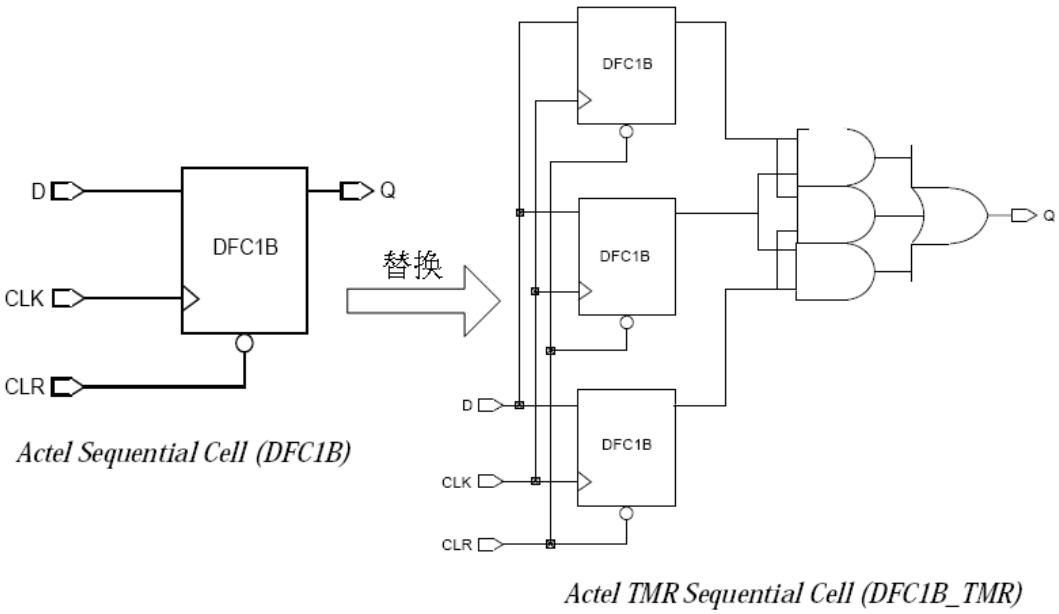


图 34 FPGA compiler II 工具寄存器替换示意图

对于 Xilinx 的 FPGA 主要的单粒子效应的软故障模式有单粒子翻转(Single Event Upset，SEU)、单粒子瞬时干扰(Single Event Transient，SET)、软错误功能中断(Single Event Functional Interrupts，SEFI)，其中 SEFI 只能通过重新配置的方式恢复。一般情况下，FPGA 中未用到的配置位翻转不会引起功能错误。Xilinx 提供了 TMRtools 进行 TMR 设计，提供了 JBits 工具进行重配置相关的设计。根据实际可靠性要求，对于 Xilinx 的 FPGA 可以采用表 2 所列的方式灵活组合来消除单粒子效应造成的软故障的影响，一般方式 1 和方式 2 或方式 1 和方式 3 应同时采用应对不同的故障模式，建议至少实现方式 4。

表 2　消除单粒子效应造成的软故障的方式及优缺点

方式	优点	缺点
方式 1： 使用 TMRtools 进行 TMR 设计	可以容忍 SEU 和 SET 引起的功能错误	不能容忍使用到的配置位翻转错误，逻辑增加约 3.5 倍，速度降低
方式 2： 周期性开环配置刷新	刷新期间，FPGA 功能不受影响，通过刷新可以纠正配置位翻转错误	只能刷新 FPGA 内部的 CLB 配置 (Configuration Memory)，BlockRAM 等资源配置不能被刷新
方式 3： 周期性闭环配置刷新(读回、比较和部分重配置)	读回、比较和部分重配置期间，FPGA 功能不受影响，通过刷新可以纠正配置位翻转错误，还可以统计错误发生概率	只能刷新 FPGA 内部的 CLB 配置，BlockRAM 等资源配置不能被刷新，比周期性开环配置刷新方式实现复杂
方式 4： 重新配置(分为上电重配置和不断电情况下重新配置两种方式)	实现简单，通过重新配置可以使发生了 SEU、SET、SEFI 错误的 FPGA 恢复正常	配置期间，功能中断

另外，在进行冗余设计的时候，在综合及布局布线阶段必须保证冗余逻辑不被工具优化。

附录 A

（规范性附录）

强制执行要求

A.1 顶层设计文件只例化底层模块和 I/O。

A.2 模块例化时使用信号名和位置映射方式。

例：

```
wb_uut: wb port map
       clk   => clk_c,
       rst_i => rst_c,
       ack_o => ack_c,
       adr_i => adr_c,
       cyc_i => cyc_c,
       rty_o => rty_c,
       stb_i => stb_c,
       we_i  => we_c;
```

A.3 每个文件内只包含一个模块(entity/module)，文件名与模块名一致。

A.4 在专门的文件内打包所有的参数等定义。

A.5 不同模块中的同一信号，使用同一信号加不同后缀连接，如*_i、*_o、*_c 分别表示某模块输入端口、某模块输出端口和模块之间的连接信号命名，连接信号复杂时可把将相关模块输出定义为一个自定义类型作为连接信号的类型，连接信号命名为*(模块名)_c。

例：

```
type INSERT_OUT is
  record
    insertbit   : std_logic;
    error       : std_logic;
  end record;
type SYNC_OUT is
  record
    rbit      : std_logic;
     enable    : std_logic;
    syncbit   : std_logic;
  end record;
……(省略的代码)
 signal insert_c : INSERT_OUT;
signal sync_c   : SYNC_OUT;
……(省略的代码)
insert_component : insert
   port map (
   -- 输入
   clock_i       => clock_c,
   bit_i         => sync_c.rbit,
   enable_i      => sync_c.enable,
   -- 输出
    insertbit_o   => insert_c.insertbit,
    error_o       => insert_c.error);


sync_component : sync
   port map (
   -- 输入
   clock_i     => clock_c,
   reset_i     => reset_c,
   tbit_i      => tbit_c,
   insertbit_i => insert_c.insertbit,
   -- 输出
   rbit_o     => sync_c.rbit,
   enable_o   => sync_c.enable,
   syncbit_o  => sync_c.syncbit);
```

A.6 文件顶端加入注释，包括版权、项目名、模块名、文件名、作者、功能和特点、版本号、日期、详细的更改记录等。

例：

```
--------------------------------------------------------------
-- 版权(copyright)：中国空间技术研究院(CAST)
-- 项目名：XXX
-- 模块名：XX
-- 文件名：XX.vhd
-- 作者：  XXX
-- 功能和特点概述：XXXXXXX
-- 初始版本和发布时间：1.0，2006-8-27
--------------------------------------------------------------
-- 更改历史：
------------------------------------------------
-- 更改版本和更改时间：1.2，2006-12-27
-- 更改人员：XXX
-- 更改描述：
-- 更改版本和更改时间：1.1，2006-10-27
-- 更改人员：XXX
-- 更改描述：
```

A.7 代码应具有详细注释，注释比例不少于 20%，无特殊要求时应采用中文释。

A.8 代码应具有层次结构。

A.9 代码中组合逻辑部分和时序逻辑部分应分开描述。

A.10 命名由字母、_、数字组成，第一个符号为字母，不超过 20 字符。

A.11 除常量、用户定义类型用大写字母外，其他命名如信号、变量、端口等用小写字母。

A.12 命名有意义，如 ram_addr，clock，低电平有效信号以 n_开头。

A.13 命名不能和 VHDL、Verilog、FPGA 内部资源等关键字同名。

A.14 使用常量代替数字。

A.15 仅使用由高到低的总线方向。

A.16 不在端口使用 buffer 类型使信号即可在内部使用又可为端口输出信号

(VHDL)。

A.17　RTL 可综合代码应符合可综合的 VHDL 子集(IEEE1076.6)和 Verilog 子集(IEEE1364.1)。

A.18　采用时钟使能的方式，不采用门控时钟。

A.19　状态机编译设置必须选择 Safe 编译模式。

A.20　禁止选择寄存器复制(register duplicate/register replication)的方式。

附录 B

（规范性附录）

推荐执行要求

B.1　仅使用 IEEE 标准类型 std_logic 及 std_logic_vector(VHDL)。

B.2　禁止使用配置(VHDL)。

B.3　采用同步设计。(详见 6.1 节)

B.4　用寄存器同步输入输出信号。(详见 6.1 节)

B.5　保证复位的有效性。(详见 6.2 节)

B.3.1　设计的上电复位时间满足 FPGA 上电时间要求、配置时间要求(如 Xilinx 器件)及晶振上电时间要求，有 50%的设计余量。

B.3.2　对 FPGA 的 IO 在上电及配置过程中是否对外有影响进行分析，根据具体情况处理去除影响。

B.3.3　把高扇出的信号映射到 FPGA 内部的高速布线网络上，如复位信号。

B.3.4　对 FPGA 的输入复位信号进行同步。

B.3.5　对于 Xilinx 公司的 FPGA 器件，采用带同步复位的存储器实现。

B.3.6　对于 Actel 公司的 FPGA 器件，采用带异步复位的存储器实现。

B.6　保证时钟信号的稳定和去除时钟偏移的影响。(详见 6.3 节)

B.4.1　如果设计中时钟较多，而时钟网络资源不够，则必须分析时钟偏移的影响，根据具体情况处理去除影响。

B.4.2　只使用时钟上升沿，便于 FPGA 实现。

B.4.3　不使用 FPGA 内部组合逻辑产生的信号作为时钟。

B.4.4　在 FPGA 器件内部有相关时钟管理资源时，如果需要对外部输入时钟进

行倍频、降频或者相位变化等处理，调用 FPGA 内部的时钟管理资源来完成。

B.7　对异步接口信号进行处理，使用两个同步寄存器处理异步信号。(详见 6.4 节)

B.8　采用本标准的示例描述常用硬件逻辑(VHDL)。(详见 6.5 节-6.12 节)

B.9　对于 Xilinx 的 FPGA 器件应实现在轨重新配置，使发生了 SEU、SET、SEFI 错误的 FPGA 可以恢复正常。

B.10　在进行冗余设计的时候，在综合及布局布线阶段必须保证冗余逻辑不被工具优化。