FPGA 设计的指导原则

第一章 FPGA 设计指导性原则

这一部分主要介绍 FPGA/CPLD 设计的指导性原则，如 FPGA 设计的基本原则、基本设计思想、基本操作技巧、常用模块等。FPGA/CPLD 设计的基本原则、思想、技巧和常用模块是一个非常大的问题，在此不可能面面俱到，只能我们公司项目中常用的一些设计原则与方法提纲携领地加以介绍，希望引起同事们的注意，如果大家能有意识的用这些原则方法指导日后的工作，不断积累和充实自己，将取得事半功倍的效果！

本章主要内容如下：

- 基本原则之一：面积和速度的平衡与互换；
- 基本原则之二：硬件原则；
- 基本原则之三：系统原则；
- 基本原则之四：同步设计原则；
- 基本设计思想与技巧之一:乒乓操作；
- 基本设计思想与技巧之二:串并转换；
- 基本设计思想与技巧之三:流水线操作；
- 基本设计思想与技巧之四：数据接口的同步方法；
- 常用模块之一：RAM；
- 常用模块之二：FIFO；

• 常用模块之三：全局时钟资源与时钟锁相环；

• 常用模块之四：全局复位、置位信号。

1.1 基本原则之一：面积和速度的平衡与互换

这里“面积”指一个设计消耗 FPGA/CPLD 的逻辑资源的数量，对于 FPGA 可以用所消耗的触发器（FF）和查找表（LUT）来衡量，更一般的衡量方式可以用设计所占用的等价逻辑门数。“速度”指设计在芯片上稳定运行，所能够达到的最高频率，这个频率由设计的时序状况决定，和设计满足的时钟周期，PAD to PAD Time，Clock Setup Time，Clock Hold Time，Clock-to-Output Delay 等众多时序特征量密切相关。面积（area）和速度（speed）这两个指标贯穿着 FPGA/CPLD 设计的始终，是设计质量的评价的终极标准。这里我们就讨论一下关于面积和速度的两个最基本的概念：面积与速度的平衡和面积与速度的互换。

面积和速度是一对对立统一的矛盾体。要求一个同时具备设计面积最小，运行频率最高是不现实的。更科学的设计目标应该是在满足设计时序要求（包含对设计频率的要求）的前提下，占用最小的芯片面积。或者在所规定的面积下，使设计的时序余量更大，频率跑得更高。这两种目标充分体现了面积和速度的平衡的思想。关于面积和速度的要求，我们不应该简单的理解为工程师水平的提高和设计完美性的追求，而应该认识到它们是和我们产品的质量和成本直接相关的。如果设计的时序余量比较大，跑的频率比较高，意味着设计的健壮性更强，整个系统的质量更有保证；另一方面，设计所消耗的面积更小，则意味着在单位芯片上实现的功能模块更多，需要的芯片数量更少，整个系统的成本也随之大幅度削减。

作为矛盾的两个组成部分，面积和速度的地位是不一样的。相比之下，满足时序、工作频率的要求更重要一些，当两者冲突时，采用速度优先的准则。

面积和速度的互换是 FPGA/CPLD 设计的一个重要思想。从理论上讲，一个设计如果时序余量较大，所能跑的频率远远高于设计要求，那么就能通过功能模块复用减少整个设计消耗的芯片面积，这就是用速度的优势换面积的节约；反之，如果一个设计的时序要求很高，普通方法达不到设计频率，那么一般可以通过将数据流串并转换，并行复制多个操作模块，对整个设计采取“乒乓操作”和“串并转换”的思想进行运作，在芯片输出模块再对数据进行“并串转换”，是从宏观上看整个芯片满足了处理速度的要求，这相当于用面积复制换速度提高。面积和速度的互换的具体操作有很多的技巧，比如模块复用，“乒乓操作”，“串并转换”等，需要大家在日后工作中积累掌握。下面举例说明如何使用“速度换面积”和“面积

换速度”。

例 1：如何使用“速度的优势换取面积的节约”？

在 WCDMA 预商用系统设计中，使用到了快速哈达码（FHT）运算，FHT 由四步相同的算法完成，如图 1 所示。FHT 的单步算法如下：

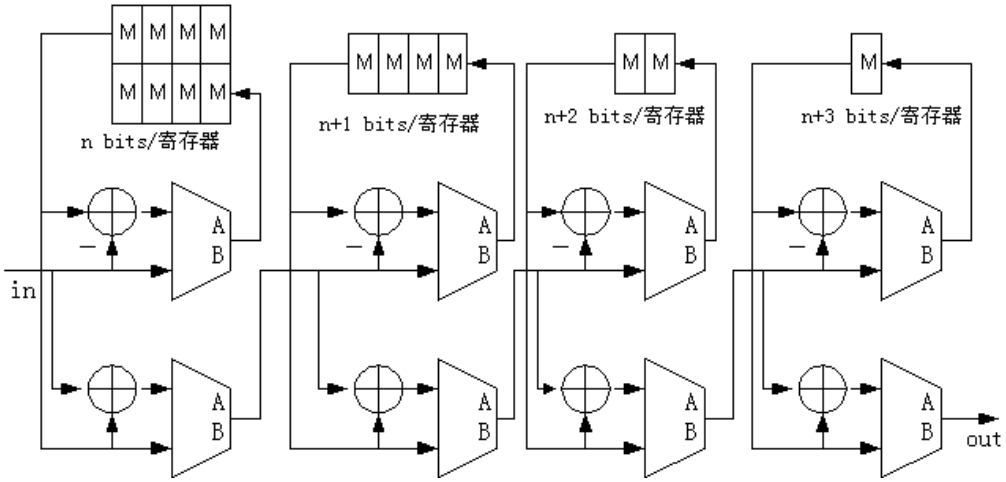

图 1-1　FHT 原理图

原设计由于考虑流水线式数据处理的要求，做了不同端口宽度的 4 个单步 FHT，并用将这 4 个步模块串联起来，以完成数据流的流水线处理。该 FHT 实现方式的代码如下：

```
//该模块是 FHT 的顶层，调用 4 个不同端口宽度的单步 FHT 模块，完成整个 FHT 算法
module fhtpart(Clk,Reset,FhtStarOne,FhtStarTwo,FhtStarThree,FhtStarFour,
               I0,I1,I2,I3,I4,I5,I6,I7,I8,
               I9,I10,I11,I12,I13,I14,I15,
               Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,
               Out9,Out10,Out11,Out12,Out13,Out14,Out15);
input Clk;     //设计的主时钟
input Reset;   //异步复位
input FhtStarOne,FhtStarTwo,FhtStarThree,FhtStarFour;  //4 个单步算法的时序控制信号
input [11:0] I0,I1,I2,I3,I4,I5,I6,I7,I8;
input [11:0] I9,I10,I11,I12,I13,I14,I15;                    //FHT 的 16 个输入
output [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7;
output [15:0] Out8,Out9,Out10,Out11,Out12,Out13,Out14,Out15;  //FHT 的 16 个输出
```

```
//第 1 次 FHT 单步运算的输出
wire [12:0] m0,m1,m2,m3,m4,m5,m6,m7,m8,m9;
wire [12:0] m10,m11,m12,m13,m14,m15;
//第 2 次 FHT 单步运算的输出
wire [13:0] mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7,mm8,mm9;
wire [13:0] mm10,mm11,mm12,mm13,mm14,mm15;
//第 3 次 FHT 单步运算的输出
wire [14:0] mmm0,mmm1,mmm2,mmm3,mmm4,mmm5,mmm6,mmm7,mmm8,mmm9;
wire [14:0] mmm10,mmm11,mmm12,mmm13,mmm14,mmm15;
//第 4 次 FHT 单步运算的输出
wire [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,Out9;
wire [15:0] Out10,Out11,Out12,Out13,Out14,Out15;
//第 1 次 FHT 单步运算
fht_unit1 fht_unit1(Clk,Reset,FhtStarOne,
          I0,I1,I2,I3,I4,I5,I6,I7,I8,
          I9,I10,I11,I12,I13,I14,I15,
          m0,m1,m2,m3,m4,m5,m6,m7,m8,
          m9,m10,m11,m12,m13,m14,m15
          );
//第 2 次 FHT 单步运算
fht_unit2 fht_unit2(Clk,Reset,FhtStarTwo,
          m0,m1,m2,m3,m4,m5,m6,m7,m8,
          m9,m10,m11,m12,m13,m14,m15,
          mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7,mm8,
          mm9,mm10,mm11,mm12,mm13,mm14,mm15
          );
//第 3 次 FHT 单步运算
fht_unit3 fht_unit3(Clk,Reset,FhtStarThree,
          mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7,mm8,
          mm9,mm10,mm11,mm12,mm13,mm14,mm15,
```

```
        mmm0,mmm1,mmm2,mmm3,mmm4,mmm5,mmm6,mmm7,mmm8,
        mmm9,mmm10,mmm11,mmm12,mmm13,mmm14,mmm15
        );
//第 4 次 FHT 单步运算
fht_unit4 fht_unit4(Clk,Reset,FhtStarFour,
        mmm0,mmm1,mmm2,mmm3,mmm4,mmm5,mmm6,mmm7,mmm8,
        mmm9,mmm10,mmm11,mmm12,mmm13,mmm14,mmm15,
        Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,
        Out9,Out10,Out11,Out12,Out13,Out14,Out15
        );
endmodule
```

单步 FHT 运算如下（仅仅举例第 4 步的模块）：

```
module fht_unit4(Clk,Reset,FhtStar,
        In0,In1,In2,In3,In4,In5,In6,In7,In8,
        In9,In10,In11,In12,In13,In14,In15,
        Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,
        Out9,Out10,Out11,Out12,Out13,Out14,Out15
        );
input Clk;          //设计的主时钟
input Reset;         //异步复位
input FhtStar;        //单步 FHT 运算控制信号
input [14:0] In0,In1,In2,In3,In4,In5,In6,In7,In8,In9;
input [14:0] In10,In11,In12,In13,In14,In15;          //单步 FHT 运算输入
output [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,Out9;
output [15:0] Out10,Out11,Out12,Out13,Out14,Out15;   //单步 FHT 运算输出
//Single FHT calculation
reg [15:0] Out0,Out1,Out2,Out3,Out4,Out5;
reg [15:0] Out6,Out7,Out8,Out9,Out10,Out11;
reg [15:0] Out12,Out13,Out14,Out15;
//补码运算
```

```
wire [14:0] In8Co =~In8+1;
wire [14:0] In9Co =~In9+1;
wire [14:0] In10Co=~In10+1;
wire [14:0] In11Co=~In11+1;
wire [14:0] In12Co=~In12+1;
wire [14:0] In13Co=~In13+1;
wire [14:0] In14Co=~In14+1;
wire [14:0] In15Co=~In15+1;
always @(posedge Clk or negedge Reset)
begin
 if(!Reset)
 begin
   Out0<=0;Out1<=0;Out2<=0;Out3<=0;
   Out4<=0;Out5<=0;Out6<=0;Out7<=0;
   Out8<=0;Out9<=0;Out10<=0;Out11<=0;
   Out12<=0;Out13<=0;Out14<=0;Out15<=0;
 end
 else
 begin
   if(FhtStar)
   begin
   Out0<={In0[14],In0 }+{In8[14],In8 };
   Out1<={In0[14],In0 }+{In8Co[14],In8Co };
   Out2<={In1[14],In1 }+{In9[14],In9 };
   Out3<={In1[14],In1 }+{In9Co[14],In9Co };
   Out4<={In2[14],In2 }+{In10[14],In10 };
   Out5<={In2[14],In2 }+{In10Co[14],In10Co };
   Out6<={In3[14],In3 }+{In11[14],In11 };
   Out7<={In3[14],In3 }+{In11Co[14],In11Co };
   Out8<={In4[14],In4 }+{In12[14],In12 };
```

```
    Out9<={In4[14],In4 }+{In12Co[14],In12Co };
    Out10<={In5[14],In5 }+{In13[14],In13 };
    Out11<={In5[14],In5 }+{In13Co[14],In13Co };
    Out12<={In6[14],In6 }+{In14[14],In14 };
    Out13<={In6[14],In6 }+{In14Co[14],In14Co };
    Out14<={In7[14],In7 }+{In15[14],In15 };
    Out15<={In7[14],In7 }+{In15Co[14],In15Co };
    end
 end
end
endmodule
```

当评估完系统的流水线时间余量后，发现整个流水线有 **16** 个时钟周期，而 **FHT** 模块的频率很高，加法本身仅仅消耗 **1** 个时钟周期，加上数据的选择和分配所消耗时间，也能完全满足频率要求，所以将单步 **FHT** 运算复用 **4** 次，就能大幅度节约所消耗的资源。这种复用单步算法的 **FHT** 实现框图如图 **1-2** 所示，由输入选择寄存、单步 **FHT** 模块、输出选择寄存、计数器构成。

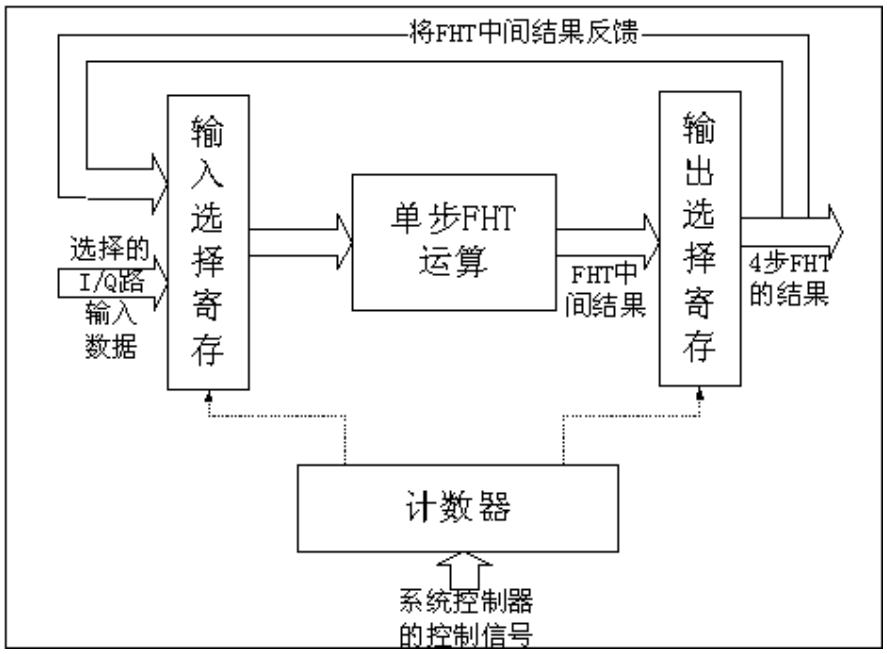

图 **1-2** **FHT** 运算复用结构图

代码如下：

```
//复用单步算法的 FHT 运算模块
module wch_fht(Clk,Reset,
          PreFhtStar,
          In0,In1,In2,In3,In4,In5,In6,In7,
          In8,In9,In10,In11,In12,In13,In14,In15,
          Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,
          Out9,Out10,Out11,Out12,Out13,Out14,Out15
          );
input Clk;          //设计的主时钟
input Reset;         //异步复位信号
input PreFhtStar;                    //FHT 运算指示信号，和上级模块运算关联
input [11:0] In0,In1,In2,In3,In4,In5,In6,In7;
input [11:0] In8,In9,In10,In11,In12,In13,In14,In15;          //FHT 的 16 个输入
output [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7;
output [15:0] Out8,Out9,Out10,Out11,Out12,Out13,Out14,Out15;    //FHT 的 16 个输出
//FHT 输出寄存信号
reg [15:0] Out0,Out1,Out2,Out3,Out4,Out5,Out6,Out7;
reg [15:0] Out8,Out9,Out10,Out11,Out12,Out13,Out14,Out15;
//FHT 的中间结果
wire [15:0] Temp0,Temp1,Temp2,Temp3,Temp4,Temp5,Temp6,Temp7;
wire [15:0] Temp8,Temp9,Temp10,Temp11,Temp12,Temp13,Temp14,Temp15;
//FHT 运算控制计数器，和前一级流水线模块配合
reg [2:0] Cnt3;//count from 0 to 4,when Reset Cnt3=7;
reg FhtEn;//Enable fht culculate
always @(posedge Clk or negedge Reset)
begin
  if (!Reset)
     Cnt3<= #1 3'b111;
  else
  begin
```

```
      if(PreFhtStar)
      Cnt3<= #1 3'b100;
      else
      Cnt3<= #1 Cnt3-1;
  end
end
always @(posedge Clk or negedge Reset)
if (!Reset)
  FhtEn<= #1 0;
else
begin
    if (PreFhtStar)
    FhtEn<= #1 1;
    if  (Cnt3==1)
    FhtEn<= #1 0;
end
//补码运算，复制符号位
       assign Temp0=(Cnt3==4)?{4{In0[11]},In0}:Out0;
       assign Temp1=(Cnt3==4)?{4{In1[11]},In1}:Out1;
       assign Temp2=(Cnt3==4)?{4{In2[11]},In2}:Out2;
       assign Temp3=(Cnt3==4)?{4{In3[11]},In3}:Out3;
       assign Temp4=(Cnt3==4)?{4{In4[11]},In4}:Out4;
       assign Temp5=(Cnt3==4)?{4{In5[11]},In5}:Out5;
       assign Temp6=(Cnt3==4)?{4{In6[11]},In6}:Out6;
       assign Temp7=(Cnt3==4)?{4{In7[11]},In7}:Out7;
       assign Temp8=(Cnt3==4)?{4{In8[11]},In8}:Out8;
       assign Temp9=(Cnt3==4)?{4{In9[11]},In9}:Out9;
       assign Temp10=(Cnt3==4)?{4{In10[11]},In10}:Out10;
       assign Temp11=(Cnt3==4)?{4{In11[11]},In11}:Out11;
       assign Temp12=(Cnt3==4)?{4{In12[11]},In12}:Out12;
```

```
        assign Temp13=(Cnt3==4)?{4{In13[11]},In13}:Out13;
        assign Temp14=(Cnt3==4)?{4{In14[11]},In14}:Out14;
        assign Temp15=(Cnt3==4)?{4{In15[11]},In15}:Out15;
always @(posedge Clk or negedge Reset)
begin
if (!Reset)
begin
   Out0<=0;Out1<=0;Out2<=0;Out3<=0;
   Out4<=0;Out5<=0;Out6<=0;Out7<=0;
   Out8<=0;Out9<=0;Out10<=0;Out11<=0;
   Out12<=0;Out13<=0;Out14<=0;Out15<=0;
end
else
begin
 if ((Cnt3<=4) && Cnt3>=0 && FhtEn)
   begin
   Out0[15:0]<= #1 Temp0[15:0]+Temp8[15:0];
   Out1[15:0]<= #1 Temp0[15:0]-Temp8[15:0];
   Out2[15:0]<= #1 Temp1[15:0]+Temp9[15:0];
   Out3[15:0]<= #1 Temp1[15:0]-Temp9[15:0];
   Out4[15:0]<= #1 Temp2[15:0]+Temp10[15:0];
   Out5[15:0]<= #1 Temp2[15:0]-Temp10[15:0];
   Out6[15:0]<= #1 Temp3[15:0]+Temp11[15:0];
   Out7[15:0]<= #1 Temp3[15:0]-Temp11[15:0];
   Out8[15:0]<= #1 Temp4[15:0]+Temp12[15:0];
   Out9[15:0]<= #1 Temp4[15:0]-Temp12[15:0];
   Out10[15:0]<= #1 Temp5[15:0]+Temp13[15:0];
   Out11[15:0]<= #1 Temp5[15:0]-Temp13[15:0];
   Out12[15:0]<= #1 Temp6[15:0]+Temp14[15:0];
   Out13[15:0]<= #1 Temp6[15:0]-Temp14[15:0];
```

```
    Out14[15:0]<= #1 Temp7[15:0]+Temp15[15:0];
    Out15[15:0]<= #1 Temp7[15:0]-Temp15[15:0];
    end
    end
end
endmodule
```

为了便于对比两种实现方式的资源消耗，我在 **Synplify Pro** 对两种实现方法分别做了综合。两次综合选用的参数都完全一致，器件类型为：**Xilinx Virtex-E XCV100E －6 BG352**，出于仅仅考察设计所消耗的寄存器和逻辑资源，**Enable** “**Disable I/O Insertion**” 选项，不插入 **IO**，取消 **Synplify Pro** 中诸如“**FSM Compiler**”、“**FSM Explorer**”、“**Resource Sharing**”、“**Retiming**”、“**Pipelining**” 等综合优化选项。两次综合的结果如图 **1-3**，图 **1-4** 所示。

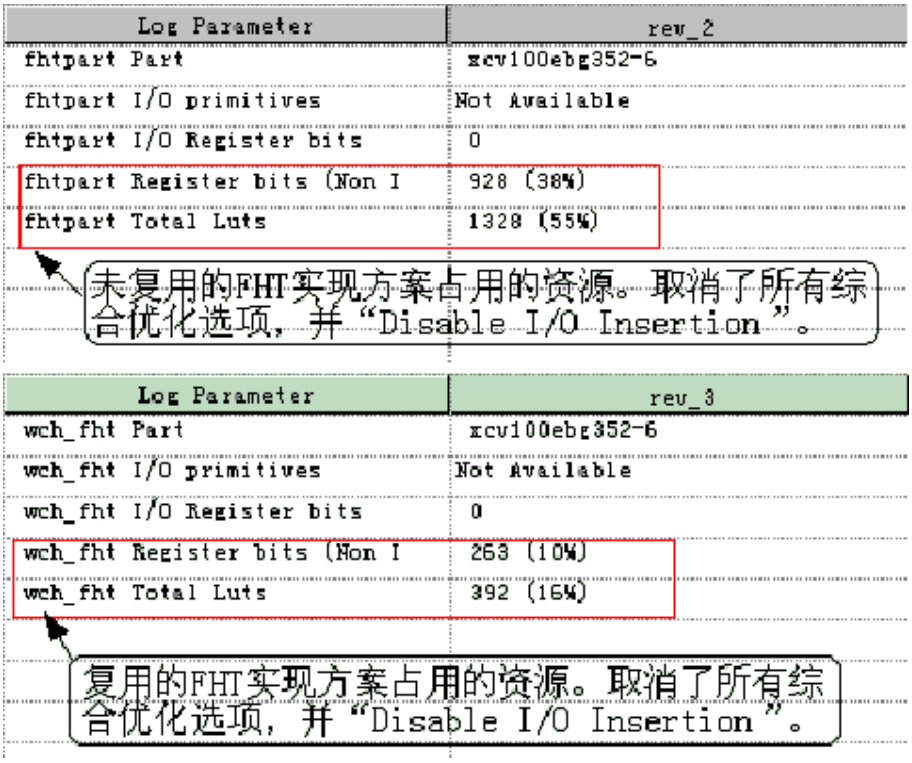

图 **1-3**　未采样复用方案的“**fhtpart**”模块综合所消耗的资源

图 **1-4**　采样复用方案的“**wch_fht**”模块综合所消耗的资源

通过对比可以清晰的观察到，采样复用实现方案所占面积约为原方案的 **1/4**，而得到这个好处的代价是：完成整个 **FHT** 运算的周期为原来的 **4** 倍。这个例子通过运算周期的加长，换取了消耗芯片面积的减少，是前面所述的用频率换面积的一种体现。本例所述“频率换面

积”的前提是：FHT 模块频率较高，运算周期的余量较大，采用 4 步复用后，仍然能够满足系统流水线设计的要求。其实，如果流水线时序允许，FHT 运算甚至可以采用 1bit 全串行方案实现，该方案所消耗的芯片面积资源更少！

例 2：如何使用“面积复制换速度提高”？

举一个路由器设计的一个例子。假设输入数据流的速率是 350Mb/s，在而 FPGA 上设计的数据处理模块的处理速度最大为 150Mb/s，由于处理模块的数据吞吐量满足不了要求，看起来直接在 FPGA 上实现是一个“impossible mission”。这种情况下，就应该利用“面积换速度”的思想，至少复制 3 个处理模块，首先将输入数据进行串并转换，然后利用这三个模块并行处理分配的数据，然后将处理结果“并串变换”，就完成数据速率的要求。我们在整个处理模块的两端看，数据速率是 350Mb/s，而在 FPGA 的内部看，每个子模块处理的数据速率是 150Mb/s，其实整个数据的吞吐量的保障是依赖于 3 个子模块并行处理完成的，也就是说占用了更多的芯片面积，实现了高速处理，通过“面积的复制换取处理速度的提高”的思想实现了设计。设计的示意框图如图 1-5 所示。

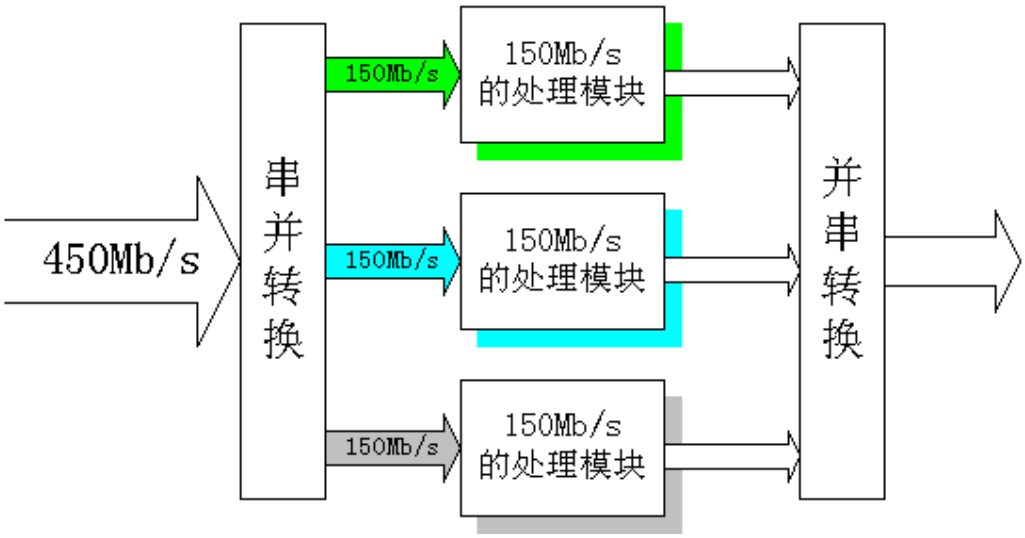

图 1-5　“面积换速度”示意图

上面仅仅是对“面积换速度”思想的一个简单的举例，其实具体操作过程中还涉及很多的方法和技巧，例如，对高速数据流进行串并转换，采用“乒乓操作”方法提高数据处理速率等。希望读者通过平时的应用进一步积累。

1.2 基本原则之二：硬件原则

硬件原则主要针对 HDL 代码编写而言的。

首先应该明确 FPGA/CPLD、ASIC 的逻辑设计所采用的硬件描述语言（HDL）与同软件语言（如 C，C++等）是有本质区别的！以 Verilog HDL 语言为例（我们公司多数逻辑工程师使用 Verilog），虽然 Verilog 很多语法规则和 C 语言相似，但是 Verilog 作为硬件描述语言，它的本质作用在于描述硬件！应该认识到 Verilog 是采用了 C 语言形式的硬件的抽象，它的最终实现结果是芯片内部的实际电路。所以评判一段 HDL 代码的优劣的最终标准是：其描述并实现的硬件电路的性能（包括面积和速度两个方面）。评价一个设计的代码水平较高，仅仅是说这个设计由硬件向 HDL 代码这种表现形式转换的更流畅、合理。而一个设计的最终性能，在更大程度上取决于设计工程师所构想的硬件实现方案的效率以及合理性。

初学者，特别是由软件转行的初学者，片面追求代码的整洁，简短，这是错误的！是与评价 HDL 的标准背道而驰的！正确的编码方法是，首先要做到对所需实现的硬件电路“胸有成竹”，对该部分硬件的结构与连接十分清晰，然后再用适当的 HDL 语句表达出来即可。

另外，Verilog 作为一种 HDL 语言，是分层次的。比较重要的层次有：系统级(System)、算法级(Algorithm)、寄存器传输级(RTL)、逻辑级(Logic)、门级(Gate)、电路开关级(Switch)设计等。系统级和算法级与 C 语言更相似，可用的语法和表现形式也更丰富。自 RTL 级以后，HDL 语言的功能就越来越侧重于硬件电路的描述，可用的语法和表现形式的局限性也越大。相比之下 C 语言与系统级和算法级 Verilog 描述更相近一些，而与 RTL 级，Gate 级、Switch 级描述从描述目标和表现形式上都有较大的差异。

例 3　举例 RTL 级 Verilog 描述语法和 C 语言描述语法的一些区别。

简单的举例 RTL 级 Verilog 描述语法和 C 语言描述语法的一些区别。在 C 语言的描述中，为了代码执行效率高，与表述简洁，经常用到如下所示的 for 循环语句：

```
for (i=0; i<16; i++)
   DoSomething();
```

但是在我们工作中，除了描述仿真测试激励（testbench）时，使用 for 循环语句外，极少在 RTL 级编码中使用 for 循环。其原因是 for 循环会被综合器展开为所有变量情况的执行语句，每个变量独立占用寄存器资源，每条执行语句并不能有效的复用硬件逻辑资源，造成巨大的资源浪费。在 RTL 硬件描述中，遇到类似算法，推荐的方式是先搞清楚设计的时序要求，做一个 reg 型计数器，在每个时钟沿累加，并在每个时钟沿判断计数器情况，做相应的处理。如果能复用的处理模块，尽量复用，即使所有操作都不能复用，也采用 case 语句展开处理。如下所示：

```
reg [3:0] counter;
```

```
always @ (posedge clk)
if (syn_rst)
   counter <= 4'b0;
else
   counter <= counter+1;
always @ (posedge clk)
   begin
        case (counter)
              4'b0000:
              4'b0001:
              ...    ...
              default:
        endcase
   end
```

另外在 C 语句描述中有 **if…else** 和 **switch** 条件判断语句，其语法如下所示：

```
if (flag)// 表示 flag 为真
…
else
…
```

switch 语句的基本格式是：

```
switch (variable)
{
case value1 : …
break;
case value2 : …
break;
…
default :  …
break;
}
```

两者之间的区别主要在于**switch**是多分支选择语句,而**if**语句只有两个分支可供选择。虽然可以用嵌套的 **if** 语句来实现多分支选择，但那样的程序冗长难读。

对应 **Verilog** 也有 **if…else** 语句和 **case** 语句，**if** 语句的语法相似，**case** 语句的语法如下：

```
case (var)
        var_value1:
        var_value1:
        ...    ...
        default:
endcase
```

姑且不论 **casex** 和 **casez** 的作用（这两个语句的应用一定要小心，要注意是否可综合），**case** 语句和 **if…else** 嵌套描述结构就有很大的区别。在 **Verilog** 语法中，**if…else if…else** 语句是有优先级的，一般来说第一个 **if** 的优先级最高，最后一个 **else** 的优先级最低。如果描述一个编码器，在 **Xilinx** 的 **XST** 综合参数就有一个关于优先级编码器硬件原语的选项 **Priority Encoder Extraction**。而 **case** 语句是“平行”的结构，所有的 **case** 的条件和执行都没有“优先级”。而建立优先级结构（优先级树）会消耗大量的组合逻辑，所以如果能够使用 **case** 语句的地方，尽量用 **case** 替换 **if...else** 结构。

关于这点简单的引申两点：第一，也可以用 **if...; if…;**的结构描述出不带优先级的“平行”条件判断语。第二，随着现在综合工具的优化能力越来越强，大多数情况下可以将不必要的优先级树优化掉。关于 **if** 和 **case** 语句的更详细的阐释，见后面关于 **Coding style** 的讨论。

1.3 基本原则之三：系统原则

系统原则包含两个层次的含义：更高层面上看，是一个硬件系统，一块单板如何进行模块花费与任务分配，什么样的算法和功能适合放在 **FPGA** 里面实现，什么样的算法和功能适合放在 **DSP**、**CPU** 里面实现，以及 **FPGA** 的规模估算数据接口设计等；具体到 **FPGA** 设计就要求对设计的全局有个宏观上的合理安排，比如时钟域，模块复用，约束，面积，速度等问题。要知道在系统上复用模块节省的面积远比在代码上小打小闹来的实惠得多。

一般来说实时性要求高、频率快的功能模块适合使用 **FPGA/CPLD** 实现。而 **FPGA** 和 **CPLD**

相比，更适合实现规模较大、频率较高、寄存器资源使用较多的设计。使用 FPGA/CPLD 设计时，应该对芯片内部的各种底层硬件资源，和可用的设计资源有一个较深刻的认识。比如 FPGA 一般触发器资源比较丰富，而 CPLD 组合逻辑资源更丰富一些，这点直接影响着两者使用的编码风格。FPGA/CPLD 一般是由底层可编程硬件单元、Block RAM 资源、布线资源、可配置 IO 单元、时钟资源等构成。底层可编程硬件单元一般由触发器（FF）和查找表（LUT）组成，Xilinx 的底层可编程硬件资源叫 SLICE，由 2 个 FF 和 2 个 LUT 组成，Altera 的底层可编程硬件资源叫 LE，由 1 个 FF 和 1 个 LUT 组成，评估两者芯片的可编程资源时需要注意这个区别。可配置 IO 单元，是 FPGA/CPLD 内部的一个重要单元，通过在实现中配置相应选项，可用使 IO 单元适配不同的 IO 接口标准。不同的器件可支持的 IO 标准不同，一些高端器件可以支持：LVTTL、LVCMOS、PCI、GTL、GTLP、HSTL、SSTL、LDT、LVDS、LVDSEXT、BLVDS、ULVDS、LVPECL、LVDCI 等多种 IO 标准，在通信领域应用非常便捷。

布线资源用以连接不同硬件单元，根据用途不同，布线资源的工艺、速度、驱动能力都不同。有全铜层的全局时钟布线资源，也有速度较快，抖动时延很小的长线资源，普通的布线资源也分很多种。时钟资源主要指片内集成的一些 DLL 或者 PLL，用于完成时钟的高精度、低抖动的倍频、分频、移相等操作。目前，Xilinx 芯片主要集成的是 DLL，而 Altera 芯片集成的是 PLL，他们各有优缺点，时钟控制的功能复杂，而精度却非常高，一般在 ps 的数量级。Block RAM 是 FPGA 的一个重要资源，在片内集成 RAM 是 FPGA 的优势之一，高端 FPGA 的片内 RAM 规模越来越大，应用也越来越广泛，是 SOPC（可编程片上系统）的有力硬件支持。使用片内 RAM 可以实现单口 RAM、双口 RAM、同步/异步 FIFO、ROM、CAM 等常用单元模块。目前 FPGA 的两个重要发展与突破是，大多数厂商在其高端器件上都提供了片上的处理器（如 CPU、DSP）等硬核（Hard Core）或固化核（Fixed Core）。比如 Xilinx 的 Virtex II Pro 芯片可以提供 Power PC，而 Altera 的 Stratix 系列芯片可以提供 Nios 和 Arm 模块。在 FPGA 上集成微处理器，使 SOPC 设计更加便利与强大。另一个发展是在不同器件商推出的高端芯片上大都集成了高速串行收发器，一般能够达到 3.125Gb/s 的数据处理能力，在 Xilinx、Altera、Lattice 都有相应的器件型号提供该功能。这个新功能是 FPGA 的数据吞吐能力大幅度增强。

一般 FPGA 系统规划的简化流程如 1-6 所示。

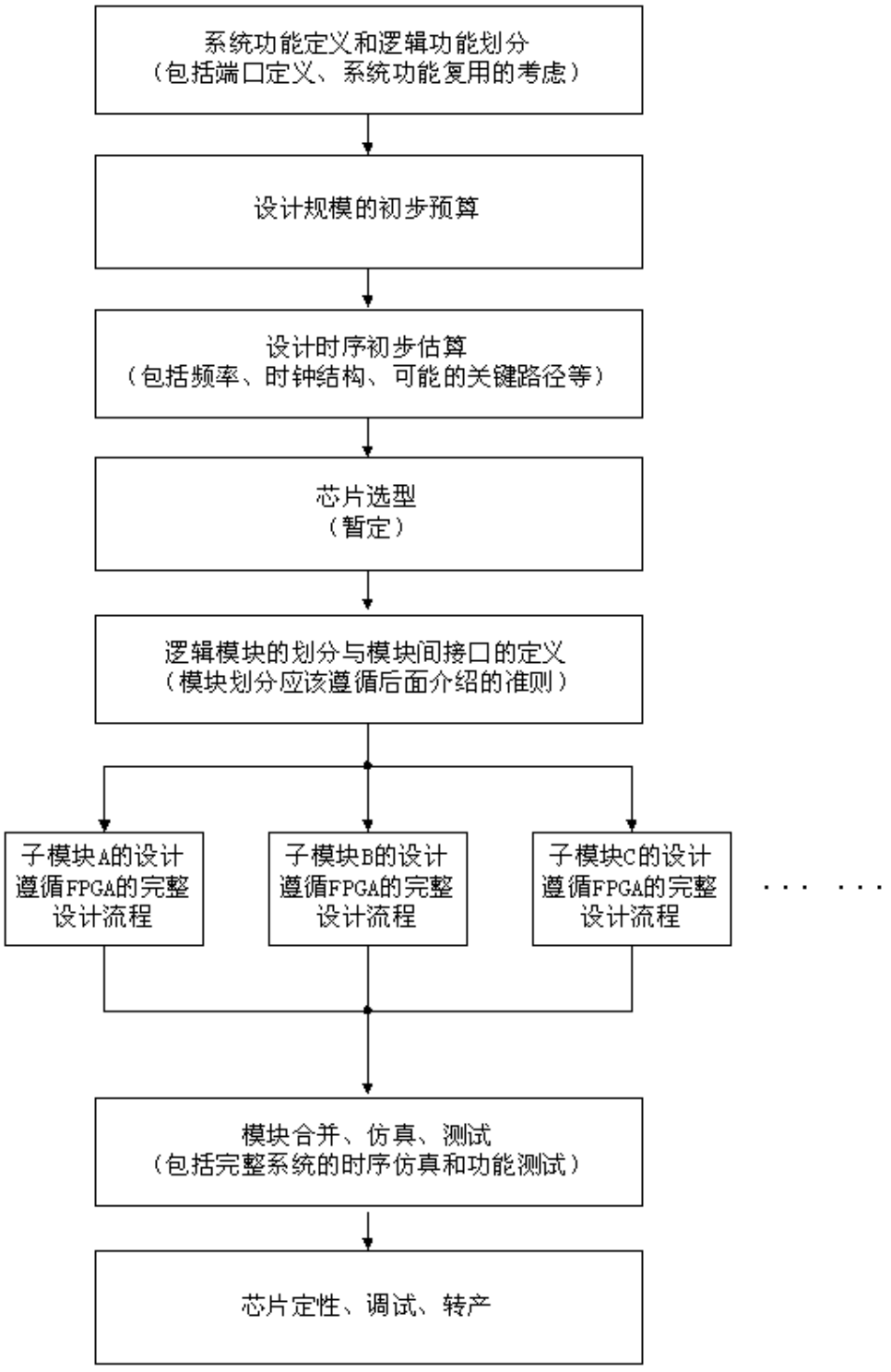

图 **1-6**　系统规划的简化流程

其中整设计的模块复用应该在系统功能定义后就初步考虑，并对模块的划分起指导性作用。模块划分非常重要，除了关系到是否最大程度上发挥项目成员的协同设计能力，而且直接决定着设计的综合、实现效果和相关的操作时间，模块划分的具体方法请参考第二章的

Coding style 中关于模块划分技巧的论述。

对于系统原则做一点引申，简单谈谈模块化设计方法。模块化设计是系统原则的一个很好的体现，它不仅仅是一种设计工具，它更是一种设计思路、设计方法，它是自顶向下、模块划分、分工协作设计思路的集中体现，是当代大型复杂系统的推荐设计方法。目前很多的 EDA 厂商都提高了模块化设计工具，如 Xilinx ISE 5.x 系列中的“Modular Design”工具包。在“Modular Design”设计流程中，实现步骤的第一步，也是整个设计流程的最重要的一步就是 Initial Budgeting。Initial Budgeting 就重复的体现了系统原则的设计理念，在该步骤，设计管理者对设计的整体进行位置布局，并完成约束每个子模块的规模和区域，定位每个模块的输入/输出，对设计进行全局时序约束等任务。

1.4 基本原则之四：同步设计原则

采用同步时序设计是 FPGA/CPLD 设计的一个重要原则。简单比较一下异步电路和同步电路的特点。

- **异步电路**

电路的核心逻辑用组合逻辑电路实现。比如异步的 FIFO/RAM 读写信号，地址译码等电路。电路的主要信号、输出信号等并不依赖于任何一个时钟性信号，不是由时钟信号驱动 FF 产生的。

异步时序电路的最大缺点是容易产生毛刺。在布局布线后仿真和用逻辑分析仪观测实际信号时，这种毛刺尤其明显。

- **同步时序电路**

电路的核心逻辑用各种各样的触发器实现。

电路的主要信号、输出信号等都是由某个时钟沿驱动触发器产生出来的。

同步时序电路可以很好的避免毛刺。布局布线后仿真，和用逻辑分析仪采样实际工作信号都没有毛刺。

由于大家对同步时序设计的原则都有一定的了解，在此不累述同步时序设计的重要性，仅仅对同步时序设计中一些常见疑问做以分析和解答。

- **是否同步时序电路一定比异步电路使用更多的资源呢？**

如果单纯的从 ASIC 设计来看，大约需要 7 个门来实现一个 D 触发器，而一个门即可实现一个 2 输入与非门，所以一般来说 ASIC 设计中，同步时序电路比异步电路占用更大的面

积。但是由于 FPGA/CPLD 是定制好的底层单元，对于 Xilinx 器件一个底层可编程单元 Slice 包含 2 个触发器（FF）和 2 个查找表（LUT），对于 Altera 器件，一个底层可编程单元 LE 包含 1 个触发器（FF）和 1 个查找表（LUT）。其中 FF 用以实现同步实现电路，LUT 用以实现组合电路。FPGA/CPLD 的最终使用率用 Slice 或者 LE 的利用率来衡量。所以对于某个选定器件，其可实现为同步实现电路和异步电路的资源的数量和比例是固定的。这点造成了过度使用 LUT，会浪费 FF 资源；过度使用 FF，会浪费 LUT 资源的情况，因而对于 FPGA/CPLD，同步时序设计不一定比异步设计多消耗资源。单纯的从节约资源的角度考虑，应该按照芯片配置的资源比例实现设计，但是设计者还要时刻权衡到同步时序设计带来的没有毛刺、信号稳定的优点，所以从资源使用的角度上看，FPGA/CPLD 设计，也是推荐采用同步时序设计的。

Ø 如何实现同步时序电路的延时？

异步电路产生延时的一般方法是插入一个 Buffer、两级非门等，这种延时调整手段是不适用于同步时序设计思想的。首先要明确一点 HDL 语言中的延时控制语法，例如：**#5 a<=4'b0101**，其中的延时 5 个时间单位，是行为级代码描述，常用于仿真测试激励，但是在电路综合是会被忽略，并不能启动延时作用。

同步时序电路的延时一般是通过时序控制完成的。换句话说，同步时序电路的延时被当做一个电路逻辑来设计。对于比较大的和特殊定时要求的延时，一般用高速时钟产生一个计数器，根据计数器的计数，控制延时；对于比较小的延时，可以用 D 触发器打一下，这种做法不仅仅使信号延时了一个时钟周期，而且完成了信号与时钟的初次同步，在输入信号采样和增加时序约束余量中使用。

Ø 同步时序电路的时钟如何产生？

同步时序电路的核心就是时钟，时钟沿驱动 FF 控制数据的产生，是同步时序电路的主要表现形式。所以时钟的质量和稳定性直接决定着同步时序电路的性能。

为了获得高驱动能、低抖动时延、稳定的占空比的时钟信号，一般使用 FPGA/CPLD 内部的专用时钟资源产生同步时序电路的主工作时钟。专用时钟资源主要指两部分，一部分是布线资源，包括全局时钟布线资源，和长线资源等。另一部分是 FPGA 内部的 PLL 或者 DLL。关于专用时钟资源和 PLL/DLL 模块的使用方法，详见“常用模块之三：全局时钟资源与时钟锁相环”。

Ø 输入信号的同步

同步时序电路要求对输入信号进行同步化，如果输入数据的节拍和本级芯片的处理时钟同频，并且建立、保持时间匹配，可以直接用本级芯片的主时钟对输入数据寄存器采样，完

成输入数据的同步化。如果输入数据和本级芯片的处理时钟是异步的，特别是频率不匹配的时候，则要用处理时钟对输入数据做两次寄存器采样，才能完成输入数据的同步化。

关于输入数据的同步化的详细论述，参见“基本设计思想与技巧之四：数据接口的同步方法”。

Ø 是不是定义为 reg 型，就一定综合成寄存器，并且是同步时序电路呢？

答案是否定的。在 **Verilog** 代码中最常用的两种数据类型是 **wire** 和 **reg**，一般来说，**wire** 型指定的数据和网线通过组合逻辑实现，而 **reg** 型指定的数据不一定就是用寄存器实现。下面的例子就是一个纯组合逻辑的译码器。请大家注意，代码中将输出信号 **Dout** 定义为 **reg** 型，但是综合与实现结果却没有使用 **FF**，这个电路是一个纯组合逻辑设计。

```
module reg_cmb(Reset,
               CS,
               Din,
               Addr,
               Dout);
input Reset;           //Asynchronous reset
input CS;              //Chip select, low effect
input [7:0] Din;       //Data in
input [1:0] Addr;      // Address
output [1:0] Dout;     //Data out
reg [1:0] Dout;
always @(Reset or CS or Addr or Din )
if (Reset)
   Dout = 0;
else if (!CS)
    begin
         case (Addr)
               2'b00: Dout = Din[1:0];
               2'b01: Dout = Din[3:2];
               2'b10: Dout = Din[5:4];
             default: Dout = Din[7:6];
```

```
        endcase
    end
else
    Dout = 2'bzz;
endmodule
```

1．5 设计思想与技巧之一：乒乓操作

这个版本的语言是 Verilog 的，因为该资料是给公司准备的一篇文档。由于要求和时间，我只列了 guideline，没有详细展开。一旦有时间，我打算把里面的内容展开，添加例子，每例都用 VHDL 和 Verilog 都写一遍，然后考虑出一个册子。不知能否得到大家的支持。前面 3 个小节，简单介绍一些 FPGA 的设计思想与操作技巧。FPGA 的设计思想和技巧多种多样，篇幅所限，我们不可能将日常工作中涉及的所有设计思想和技巧都一一讨论，在此仅仅挑选了 3 个有代表性的方法加以简要介绍，希望读者能够通过日常工作实践，总结出更多的设计思想与技巧。

“乒乓操作”是一个常常应用于数据流控制的处理技巧。典型的乒乓操作方法如图 1-8 所示。

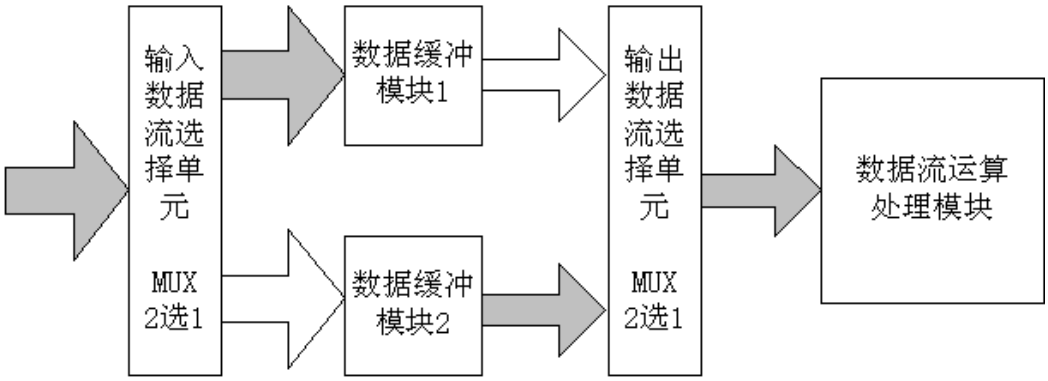

图 1-8　乒乓操作示意图

乒乓操作的处理流程描述如下：输入数据流通过“输入数据选择单元”，等时的将数据流等时分配到两个数据缓冲区。数据缓冲模块可以为任何存储模块，比较常用的存储单元为双口 RAM（DPRAM）、单口 RAM(SPRAM)、FIFO 等。在第一个缓冲周期，将输入的数据流缓存到“数据缓冲模块 1”。在第 2 个缓冲周期，通过“输入数据选择单元”的切换，将输入的数据流缓存到“数据缓冲模块 2”，与此同时，将“数据缓冲模块 1”缓存的第 1 个周期的数据通过“输入数据选择单元”的选择，送到“数据流运算处理模块”被运算处理。在第 3

个缓冲周期，通过“输入数据选择单元”的再次切换，将输入的数据流缓存到“数据缓冲模块 1”，与此同时，将“数据缓冲模块 2”缓存的第 2 个周期的数据通过“输入数据选择单元”的切换，送到“数据流运算处理模块”被运算处理。如此循环，周而复始。

乒乓操作的最大特点是，通过“输入数据选择单元”和“输出数据选择单元”按节拍、相互配合的切换，将经过缓冲的数据流没有时间停顿的送到“数据流运算处理模块”，进行运算与处理。把乒乓操作模块当做一个整体，站在这个模块的两端看数据，输入数据流和输出数据流都是连续不断的，没有任何停顿，因此非常适合对数据流进行流水线式处理。所以乒乓操作常常并应用于流水线式算法，完成数据的无缝缓冲与处理。

乒乓操作的第二个优点是可以节约缓冲区空间。比如在 WCDMA 基带应用中，1 帧（Frame）是由 15 个时隙（Slot）组成的，有时需要将 1 整帧的数据延时一个时隙后处理，比较直接的办法是将这帧数据缓存起来，然后延时 1 个时隙，进行处理。这时缓冲区的长度是 1 整帧数据长，假设数据速率是 3.84Mb/s，1 帧长 10ms，则此时需要缓冲区长度是 38400bit。如果采用乒乓操作，只需定义两个能缓冲 1 个 slot 数据的 RAM（单口 RAM 即可），当向一块 RAM 写数据的时候，从另一块 RAM 读数据，然后送到处理单元处理，此时，每块 RAM 的容量仅需 2560bit 即可。2 块 RAM 加起来也只有 5120bit 的容量。

另外巧妙的运用乒乓操作，还可以达到用低速模块处理高速数据流的效果。如图 1-9 所示，数据缓冲模块采用了双口 RAM，并在 DPRAM 后引入了一级数据预处理模块，这个数据预处理可以根据需要是各种数据运算，比如在 WCDMA 设计中，对输入数据流的解扩、解扰、去旋转等。假设端口 A 的输入数据流的速率为 100Mb/s，乒乓操作的缓冲周期是 10ms。我们下面一起分析一下各个节点端口的数据速率。

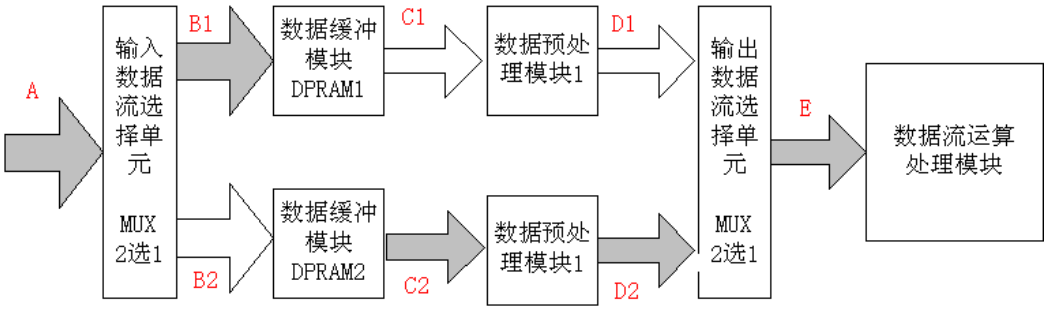

图 1-9　利用乒乓操作降低数据速率

输入数据流 A 端口处数据速率为 100Mb/s，在第 1 个缓冲周期 10ms 内，通过“输入数据选择单元”，从 B1 到达 DPRAM1。B1 的数据速率也是 100Mb/s，在 10ms 内，DPRAM1 要写入 1Mb 数据。同理在第 2 个 10ms，数据流被切换到 DPRAM2，端口 B2 的数据速率也是 100Mb/s，

DPRAM2 在第 2 个 10ms 被写入 1Mb 数据。周而复始，在第 3 个 10ms，数据流又切换到 DPRAM1，DPRAM1 被写入 1Mb 数据。

仔细分析一下，就会发现到第 3 个缓冲周期时，留给 DPRAM1 读取数据并送到“数据预处理模块 1”的时间一共是 20Ms。有的同事比较困惑于 DPRAM1 的读数时间为什么是 20ms，其实这一点完全可以实现。首先在在第 2 个缓冲周期，向 DPRAM2 写数据的 10ms 内，DPRAM1 可以进行读操作；另外在第 1 个缓冲周期的第 5ms 起（绝对时间为 5ms 时刻），DPRAM1 就可以边向 500K 以后的地址写数，边从地址 0 读数，到达 10ms 时，DPRAM1 刚好写完了 1Mb 数据，并且读了 500K 数据，这个缓冲时间内 DPRAM1 读了 5ms 的时间；另外在第 3 个缓冲周期的第 5ms 起（绝对时间为 35ms 时刻），同理可以边向 500K 以后的地址写数，边从地址 0 读数，又读取了 5 个 ms，所以截止 DPRAM1 第一个周期存入的数据被完全覆盖以前，DPRAM1 最多可以读取了 20ms 时间，而所需读取的数据为 1Mb，所以端口 C1 的数据速率为：1Mb/20ms=50Mb/s。因此“数据预处理模块 1”的最低数据吞吐能力也仅仅要求为 50Mb/s。同理“数据预处理模块 2”的最低数据吞吐能力也仅仅要求为 50Mb/s。换言之，通过乒乓操作，“数据预处理模块”的时序压力减轻了，所要求的数据处理速率仅仅为输入数据速率的 1/2。

通过乒乓操作实现低速模块处理高速数据的实质是：通过 DPRAM 这种缓存单元，实现了数据流的串并转换，并行用“数据预处理模块 1”和“数据预处理模块 2”处理分流的数据，是面积与速度互换原则的有一个体现！

第 1 章 FPGA 设计的指导性原则

- 基本原则之一：面积和速度的平衡与互换；
- 基本原则之二：硬件原则；
- 基本原则之三：系统原则；
- 基本原则之四：同步设计原则；
- 基本设计思想与技巧之一:乒乓操作；
- 基本设计思想与技巧之二:串并转换；
- 基本设计思想与技巧之三:流水线操作；
- 基本设计思想与技巧之四：数据接口的同步方法；
- 常用模块之一：RAM；
- 常用模块之二：FIFO；
- 常用模块之三：全局时钟资源与时钟锁相环；

• 常用模块之四：全局复位/置位信号；

• 常用模块之五：高速串行收发器。

是绝对原汁原味的原创！除了会在 RAM、FIFO、时序约束和 startup 中根据 Xilinx/Altera/Lattice 的 Datasheet，写一些性能和参数(这个 westor 不能自己凭空造啊)，其余的观点和文字，朋友们应该在任何地方都看不到，因为它们是完全原创，如果出现文章或者帖子和我相似，请转告我，我告他盗版侵权！

在第二章，Coding style 由于先辈已经说了很多，我只能根据自己的理解，博采众家之长，针砭漏缺汇集成篇了。

第三章，就是根据 westor 在网上回答的这些帖子，整理一下写出来。

在这 3 章本来是有很多例子的，但是由于这份材料原是为公司编写，而我们公司通用的是 Verilog 语言，所以没有附带 VHDL 的例子，另外有很多例子设计专利和机密，所以在发贴的时候都略去了。

看到大家反应这么强烈，westor 很欣慰，打算将这些材料组成成一本册子，奉献给大家。我最近正在联系一些顶尖高手，请他们将对些内容斧证、补充、完善。打算在新书中根据我们做的项目，举大量经典、实用的例子，每个例子都由 Verilog 和 VHDL 两种语言描述，并作充分的验证。现在新书的大纲和框架已经有了，并已经向出版社表示了意向，出版社很支持，如果 westor 能联合到那几个顶尖高手，相信在明年初新书就能面试，到时会令大家感到物有所值的！

westor，你在前面的“1.4 基本原则之四：同步设计原则”的关于“输入信号的同步”论述中，写道：“如果输入数据和本级芯片的处理时钟是异步的，特别是频率不匹配的时候，则只是要用处理时钟对输入数据做两次寄存器采样，才能完成输入数据的同步化。”我感觉有些问题。

记得 Deve 在“亚稳态与设计可靠性”的讨论中，也有类似的说法。我觉得只是简单的加两级触发器，只能有效减小亚稳态，避免亚稳态的传播，同步后的信号，并不一定是我们想要的正确信号，而是根输入无关的有效电平。因此，是否有必要在时钟同步的同时，加握手一类信号呢？

你前面提到 Modular Design，我有些问题不知在这里提出是否合适？感觉这种方法不适合一个人完成的项目。因为每个模块（包括顶层和全部子模块），都必须在不同的 project 中完成。想随时对某一模块修改很不方便。而我的设计有些复杂，布局布线要花很多时间。我现在没用 Modula Design 的方法，因此任意小的改动，都必须全部重新布局布线。也很麻

烦，就我的情况，有好办法么？谢谢。

客气了，想您这样的帖子我非常欢迎，大家讨论讨论，互相学习，才能进步。您的指正让我受益非浅。您说的 Modular Design，它的本意是大规模集团作战，但是我在另外一个帖子也指出，Modular Design 的成败，在很大承担上依靠“team leader”的能力（项目领导能力，和技术水平），详见我在本站发的帖子“team leader 的压力!”。

您说的情况，其实已经基本完成了设计，在进行后期调试过程中，需要不断的改进。这时应用 Modular Design 也是非常适合的，因为它是增量化设计（增量综合，增量实现）的一个重要体现。您可以反向应用 Modular Design，根据你设计的大概 PAR 结果，用 Floor planner 做位置约束，完成 initial budgeting。接着做后面两步：active module implementation 和 final assembly。相当于利用 Modular Design 方法完成了设计的分割。以后修改的时候，就可以仅仅修改一个子模块，直接对该模块综合，实现，然后再次 final assembly 即可，这是 modular design 的一个重要引申!

1.6 基本设计思想与技巧之二:串并转换

串并转换是 FPGA 设计的一个重要技巧，从小的着眼点讲，它是数据流处理的常用手段，从大的着眼点将它是面积与速度互换思想的直接体现。串并转换的实现方法多种多样，根据数据的排序和数量的要求，可以选用寄存器、RAM 等实现。前面在乒乓操作图 1-9 的举例，就是通过 DPRAM 实现了数据流的串并转换，而且由于使用了 DPRAM，数据的缓冲区可以开的很大。对于数量比较小的设计可以采用寄存器完成串并转换。如无特殊需求，应该用同步时序设计完成串并之间的转换。比如数据从串行到并行，数据排列顺序是高位在前，可以用下面的编码实现：prl_temp <= {prl_temp,srl_in}。其中，prl_temp 是并行输出缓存寄存器，srl_in 是串行数据输入。对于排列顺序有规定的串并转换，可以用 case 语句判断实现。对于复杂的串并转换，还可以用状态机实现。串并转换的方法总的来说比较简单，在此不做更多的解释。

1.7 基本设计思想与技巧之三:流水线操作

流水线处理是高速设计中的一个常用设计手段。如果某个设计的处理流程分为若干步骤，而且整个数据处理是“单流向”的，即没有反馈或者迭代运算，前一个步骤的输出是下一个步骤的输入则可以考虑采用流水线设计方法提高系统的工作频率。

流水线设计的结构示意图如图 10 所示：

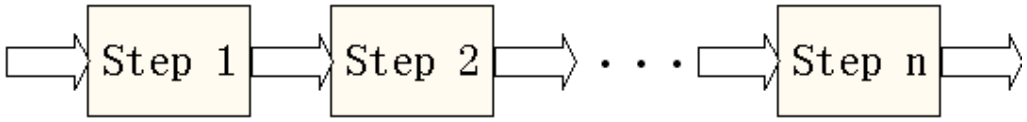

图 1-10　流水线设计的结构示意图

其基本结构为：将适当划分的 n 个操作步骤单流向串联起来。流水线操作的最大特点和要求是，数据流在各个步骤的处理，从时间上看是连续的，如果将每个操作步骤简化假设为通过一个 D 触发器（就是用寄存器打一个节拍），那么流水线操作就类似一个移位寄存器组，数据流依次流经 D 触发器，完成每个步骤的操作。流水线设计时序示意图如图 11 所示：

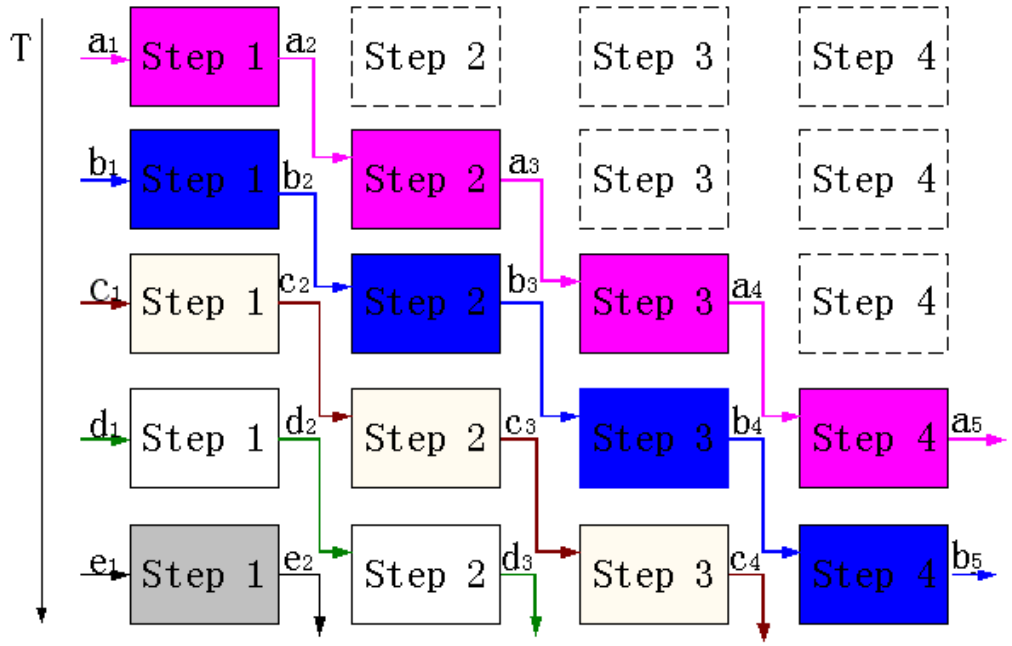

图 1-11　流水线设计时序示意图

流水线设计的一个关键在于，整个设计时序的合理安排。要求每个操作步骤的划分合理。如果前级操作时间恰好等于后级的操作时间，设计最为简单，前级的输出直接汇入后级的输入即可。如果前级操作时间大于后级的操作时间，则需要对前级的输出数据适当缓存，才能汇入后级的输入端。如果前级操作时间恰好小于后级的操作时间，则必须通过复制逻辑，将数据流分流，或者在前级对数据采用存储、后处理方式，否则会造成后级数据溢出。

在 WCDMA 设计中经常使用到流水线处理的方法，如 RAKE 接收机、搜索器、前导捕获等。

流水线处理方式之所以频率较高，是因为复制了处理模块，它是面积换取速度思想的又一种具体体现。

1.8 基本设计思想与技巧之四：数据接口的同步方法

数据接口的同步在是 FPGA/CPLD 设计的一个常见问题，也是一个重点和难点。很多设计工作不稳定都是源于数据接口的同步有问题。

在电路图设计阶段，有一些设计者养成了手工加入 BUFT 或者非门调整数据延迟，从而保证本级模块的时钟对上级模块数据的建立、保持时间的要求。还有一些设计者为了有稳定的采样，生成了很多相差 90 度的时钟信号，时而用正沿打一下数据，时而用负沿打一下数据，用以调整数据的采样位置。这两种做法都是万万取不得的。这些做法，一旦芯片更新换代，或者移植到其它器件族的芯片上，采样实现必须从新设计。而且这两种做法造成电路实现的余量不够，一旦外界条件变换（比如温度升高），采样时序就有可能完全紊乱，造成电路瘫痪。

下面简单介绍几种不同情况下的数据接口的同步方法。

- 输入、输出的延时（芯片间、PCB 布线、一些驱动接口元件的延时等）不可测，或者有可能变动，如何完成数据的同步？

对于数据的延迟不可测，或者变动，就需要建立同步机制。可以用一个同步使能，或者同步指示信号。另外使数据通过 RAM 或者 FIFO 的存取，也可以达到数据同步的目的。

把数据存放在 RAM 或 FIFO 的方法如下，将上级芯片提供的数据随路时钟作为写信号，将数据写入 RAM 或者 FIFO，然后使用本级的采样时钟（一般是数据处理的主时钟），将数据读出来即可。这种做法的关键是数据写入 RAM 或者 FIFO 要可靠，如果使用同步 RAM 或者 FIFO，就要求有应该有一个与数据相对延迟关系固定的随路指示信号，这个信号可以是数据的有效指示，也可以是上级模块将数据打出来的时钟。对于慢速数据，也可以采样异步 RAM 或者 FIFO，但是这种做法不推荐。

- 数据是有固定格式安排的，很多重要信息在数据的起始位置。

这种情况在通信系统非常普遍，通讯系统中，很多数据是按照“帧”组织的。而由于整个系统要求对时钟要求很高，常常专门设计一块时钟板完成高精度时钟的产生与驱动。而数据又是有起始位置的，如何完成数据的同步，并发现数据的“头”？

数据的同步方法完全可以采用上一点的方法，采用同步指示信号，或者使用 RAM、FIFO 缓存一下。找到数据头的方法有两种，第一种很简单，随路传输一个数据起始位置的指示信号即可。对于有些系统，特别是异步系统，则常常在数据中插入一段同步码（比如训练序列），接收端通过状态机检测到同步码后，就能发现数据的“头”了，这种做法叫做“盲检测”。

- 上级数据和本级时钟是异步的，也就是说上级芯片或模块和本级芯片或模块的时钟是异步时钟域的。

前面在输入数据同步化中已经简单介绍了一个原则：如果输入数据的节拍和本级芯片的处理时钟同频，可以直接用本级芯片的主时钟对输入数据寄存器采样，完成输入数据的同步化；如果输入数据和本级芯片的处理时钟是异步的，特别是频率不匹配的时候，则要用处理时钟对输入数据做两次寄存器采样，才能完成输入数据的同步化。需要说明的是用寄存器对异步时钟域的数据进行两次采样，其作用是有效的防止了亚稳态（数据状态不稳定）的传播，使后级电路处理的数据都是有效电平。但是这种做法并不能保证两级寄存器采样后的数据是正确的电平，这种方式处理，一般都会产生一定数量的错误电平数据。所以仅仅适用于对少量错误不敏感的功能单元。

为了避免异步时钟域产生错误的采样电平，一般使用 RAM、FIFO 缓存的方法完成异步时钟域的数据转换。最常用的缓存单元是 DPRAM，在输入端口使用上级时钟写数据，在输出端口使用本级时钟读数据，就非常方便的完成了异步时钟域之间的数据交换。

我是一个业余的初学者，看了您的帖子，有疑问向您请教。

您在乒乓操作降低数据流速率里面讲到：首先在第 2 个缓冲周期，向 DPRAM2 写数据的 10ms 内，DPRAM1 可以进行读操作……疑问：按照上面所说的，在第三个缓冲周期的第 5ms 起读的数据应该是在该缓冲周期写入的数据，而不是第一个缓冲周期写入的数据，是吗？我觉得是不是应该这样：在第三个缓冲周期开始的 5ms 内继续读第一个缓冲周期写进去的数，同时向地址 0 以后的地址写数，5ms 以后再从地址 0 读数。

“在第 3 个缓冲周期的第 5ms 起（绝对时间为 25ms 的时刻）。”

用绝对时刻描述应该比较容易理解，可以读的时间是 5ms～10ms（第一个周期）；10ms～20ms（第二个周期）；25ms～30ms（第 3 个周期），一共 20ms 的时间。从 25ms 到 30ms 读的是另一块 ram 在 20ms～25ms 写的数。这个时间利用时看起来比较复杂，如果是为了获得更多的处理时间，可以在接口上放更多的 DPRAM，作为乒乓操作的推广，其实是一种轮转查询一样的处理思路。

讨论一些 if 和 case。

不知道 westor 是用什么做的综合。如果用 fpga compiler（和 dc 是一个引擎），根据 synopsys 的文档，if...else 和 case 的结果是有优先级的，在前的优先级高，做法是使用 mux_op 器件，将优先级在 mux 的选择端处理，所有 mux 输入到输出的路径都是一样长的。要想没有优先级，需要在 case 后加//synopsys parallel_case。

而连续的 **if....if....**语句也是有优先级的，在后面的优先级高（这一点是显然的，比如在组合逻辑 **always** 模块中对一个信号连续赋值，最后的赋值才是最终的赋值），做法是在数据通路上加一层层的 **mux**，所以不同的输入口到输出的路径是不一样长的，优先级高的短，优先级低的长。

不过这可能和综合器有关，但 **westor** 的有些说法似乎有误。

首先对于多路选择器来讲，用语言来写的话，既可以用 **if..else..**来描述，也可以用 **case** 来进行描述，但是这两者的不同点在于（不加任何综合条件下）：**if..else..**语句综合出的电路是有优先级的。而 **case** 语句综合出的电路是没有优先级的。这一点对于 **synopsys** 的工程师也是承认的。而你所说的“而连续的 **if....if....**语句也是有优先级的，在后面的优先级高”这句话中应当注意一点，最后的赋值并不代表优先级最高，对于这种描述方式来讲在可综合的电路描述中是应该要避免的，可能会出现你想描述的电路和实际综合出的电路并不相同。再对于“这一点是显然的，比如在组合逻辑 **always** 模块中对一个信号连续赋值，最后的赋值才是最终的赋值”这句话来讲，它只代表信号的赋值，当然是最后的值被赋到信号上，这和电路中优先级又有什么关系，语言中最后的赋值和电路中的优先级不一样。

有些地方是有点含糊，**case** 语句不需要讨论优先级的问题，条件是互斥的，不会同时满足，但是 **casex** 就不一样，是有优先级的。至于 **if...if....**，在 **synopsys** 的 **Verilog coding style guide** 里做了明确的说明。

下面的三段代码，都可以描述优先级编码，都是 **synopsys** 文档中提及的，但是综合出的电路会有不同。当然我也说了，可能是和综合器有关的。

1）

```
begin
out=1'bx
 casex(sel[2:0])
   3'bxx1  : out = a;
   3'bx1x  : out = b;
   3'b1xx  : out = c;
 endcase
end
```

2）

```
begin
```

```
out = 1'bx
 if(sel[0])
    out = a;
 else if(sel[1])
    out = b;
 else if(sel[2])
    out = c;
end
```

3）

```
begin
out = 1'bx
 if(sel[2])
    out = c;
 if(sel[1])
    out = b;
 if(sel[0])
    out = a;
end
```

本来关于 **if** 和 **case** 的优先级在第二章讨论的，然而大家感兴趣，简单说两句我的想法。

一般来说 **case** 语句是"平行"（**balance**，**parallel**）的结构，所有的 **case** 的条件和执行都没有"优先级"。而"**if...else**"大多数情况是有优先级（**Prior**）。而建立优先级结构（优先级树）会消耗大量的组合逻辑，所以如果能够使用 **case** 语句的地方，尽量用 **case** 替换 **if...else** 结构。这点有三点补充：

首先 **if...else** 也可以写出不带优先级的"平行"结构的条件判断语句。（其实理论上 **if...else** 可以描述所有的条件判断逻辑，完全可以取代 **case** 语句）

其次，随着现在综合工具的优化能力越来越强，大多数情况下可以将不必要的优先级树优化掉。比如业界比较流行的 **Synplicity** 综合工具（**Synplify Pro**、**Amplify**，**etc.**）和 **Examplar** 的 **Leonardo Spectrum** 等。

另外，大多数综合工具在综合过程可以通过综合属性显化控制是否综合出带有优先级的结构。

关于综合器，比较流行的我都在用 DC、Fpga Compiler2、Synplify Pro、Leonardo 等。Synopsys 的比较忠于作者，Synplicity 和 Examplar 的优化力度比较大，各有千秋。

第二章 FPGA 设计的具体准则

这章论述独立于综合器和布局布线器之外的，一般意义上的 Coding style 和设计准则。

本章的主要内容如下：

- Coding style 的含义；
- HDL 语言的层次含义；
- 结构层次化编码；
- 模块的划分的技巧；
- 比较判断语句 case 和 if...else 的优先级；
- 慎用 Latch；
- FSM 设计的一般型原则；
- 用 Verilog 语言设计 FSM 的技巧；
- 使用 Pipelining 方法优化时序；
- 如何有效地进行的 Resource Sharing；

第三章 FPGA 设计的常见疑问

本章讨论 FPGA 设计中的常见问题，和解决方法与思路。

本章的主要内容如下：

- 时序优化的基本思想与技巧；
- 综合后仿真的含义；
- 如何观察综合报告；
- 赋值延时语句的作用；