第1章 FPGA 设计的指导性原则

王诚 Westor Wang
westor@edacn.net

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：全局时钟资源与时钟锁相环;
- 常用模块之三：全局复位/置位信号;
- 常用模块之四： 高速串行收发器。

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，在芯片输出模块再在对数据进行“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：

$$Out[2i] = In[2i] + In[2i+8]; i = 0-7;$$
$$Out[2i+1] = In[2i+1] - In[2i+1+8]; i = 0-7$$

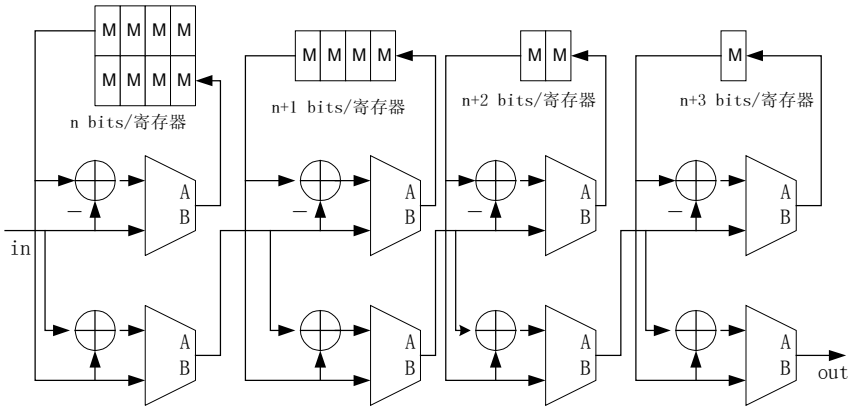


图1-1　FHT 原理图

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 个单步模块串联起来，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 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。

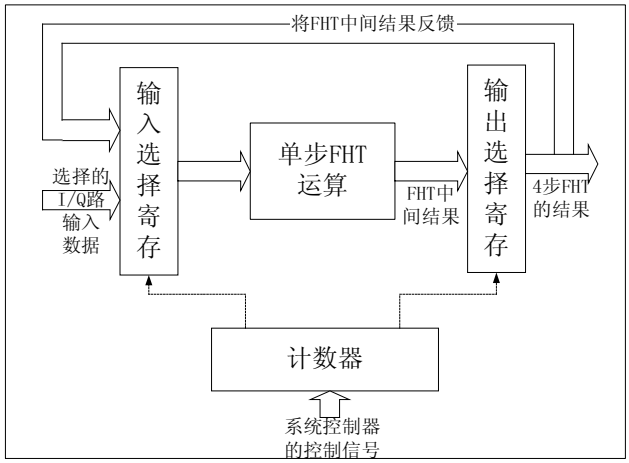


图1-2　FHT 运算复用结构图

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 3，图 4 所示。

Log Parameter	rev_2
fhtpart Part	xcv100ebg352-6
fhtpart I/O primitives	Not Available
fhtpart I/O Register bits	0
fhtpart Register bits (Non I	928 (38%)
fhtpart Total Luts	1328 (55%)

未复用的FHT实现方案占用的资源。取消了所有综合优化选项，并“Disable I/O Insertion”。

图1-3　未采样复用方案的“fhtpart”模块综合所消耗的资源

Log Parameter	rev_3
wch_fht Part	xcv100ebg352-6
wch_fht I/O primitives	Not Available
wch_fht I/O Register bits	0
wch_fht Register bits (Non I	263 (10%)
wch_fht Total Luts	392 (16%)

复用的FHT实现方案占用的资源。取消了所有综合优化选项，并“Disable I/O Insertion”。

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 450Mb/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 450Mb/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，也就是说利用了占用更多的芯片面积，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 5 所示。

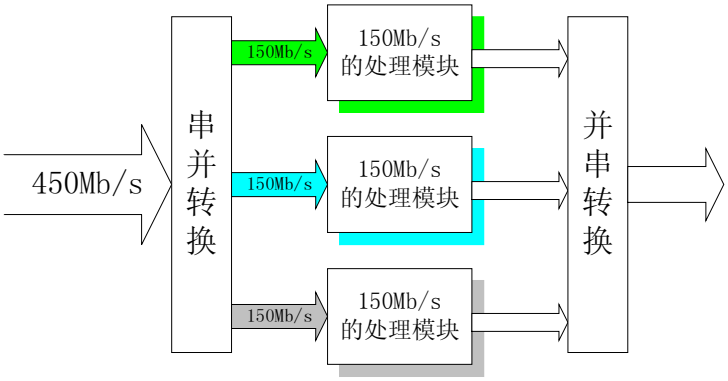


图1-5　“面积换速度”示意图

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 VerilogHDL 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"心有成竹"，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，在通信领域应用是否便捷。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、Excalibur 等系列芯片可以提供 Nios、DSP 和 Arm 等模块。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 3Gb/s 以上的数据处理能力，在 Xilinx、Altera、Lattice 都有相应的器件型号提供该功能。这些新功能是 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 6 所示。

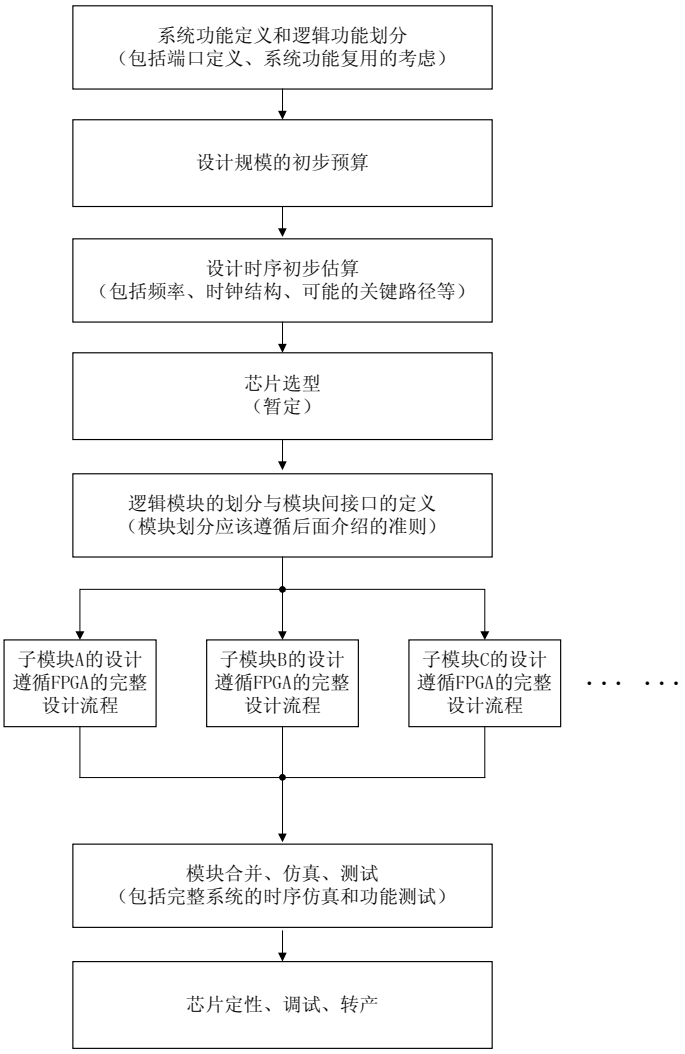


图1-6　系统规划的简化流程

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，它是是由顶向下、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。

例 4. 在系统层次复用模块。

某公司在一篇专利中提到可编程匹配滤波器实现 WCDMA 基站的方案，就重复利用了系统原则，提高单元模块的复用率，从而大大的降低硬件消耗。其简单功能框图 7 所示。

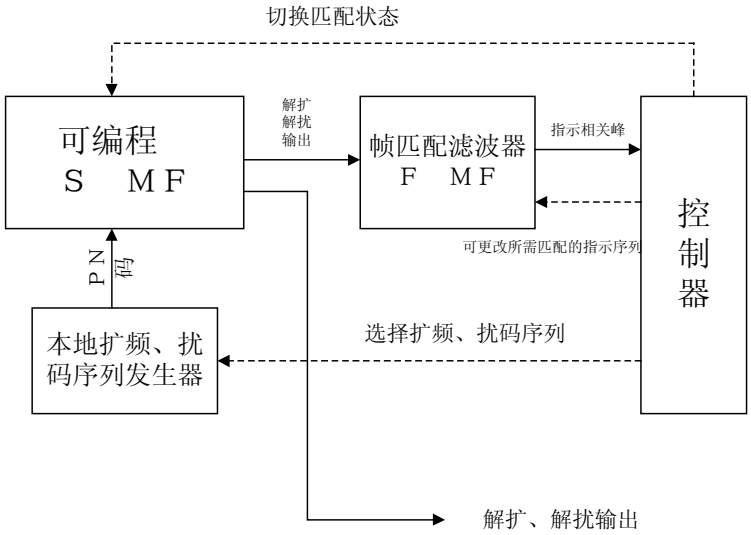


图1-7 可编程匹配滤波器原理框图

其设计思想是：利用信道固有特点（如信道 pilot 导频，信道结构等），应用现代可编程数字信号处理的技术（如 DSP、FPGA 等），采取反馈与控制匹配滤波方式，实现对某信道的已扩信息的自动解扩解扰。该可编程 MF 的主要组成部分为以下四部分：本地码发生器、可编程信号 MF（S MF）、帧匹配滤波器（FRAME MF）和控制器。本地码发生器可生成各种所需的扩频、加扰序列。可接收控制器的指示脉冲，产生规定的本地解扩、解扰序列，作为 S MF 的参考序列；S MF 是完成匹配滤波的主体，可接收控制器的指示脉冲，将自己的匹配状态切换倒下一匹配状态；F MF 完成对导频信号等特殊信号（信息比特待选集有限）的检测，生成指示相关峰，通知控制器将 S MF 切换倒下一匹配状态；控制器统一协调各部分工作。这种可编程滤波器可以在如越区切换、同步方面、CPCH 收发信机等多方面应用，如果适当安排时序流程，可以在较大的程度上节约硬件资源。

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。

- **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。

- **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”。

- **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，同步化的主要作用是使本级时钟的处理沿获得相对于数据的最长有效处理时间，从而获得了更长的时间余量。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。需要说明的是，两次寄存器采样的作用在于有效地防止了亚稳态（数据状态不定）的传播，是后级电路获得的电平为有效电平，但是这种处理并不能防止错误采样电平的产生。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”。

- **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 基本设计思想与技巧之一:乒乓操作

后面 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 8 所示。

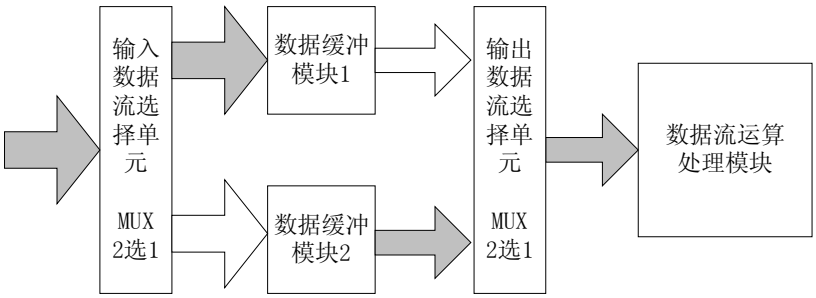


图1-8　乒乓操作示意图

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”，被运算与处理。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 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。

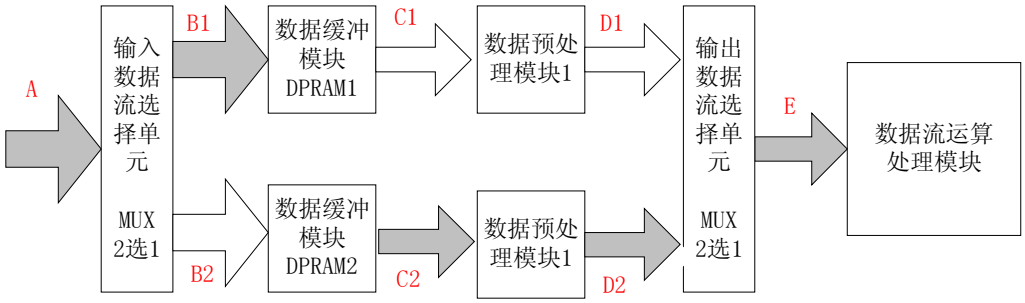


图1-9　利用乒乓操作降低数据速率

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！

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 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

首先需要声明的是这里所讲述的流水线是指一种处理流程和顺序操作的设计思想，并非 FPGA、ASIC 设计中优化时序所用的“Pipelining”，关于 Pipelining 优化时序的方法在第二章有详细介绍。

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：

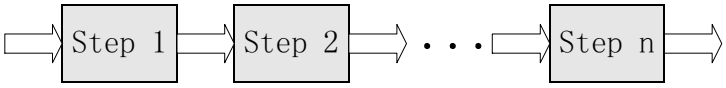


图1-10　流水线设计的结构示意图

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。流水线操作的最多特点和要求是，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：

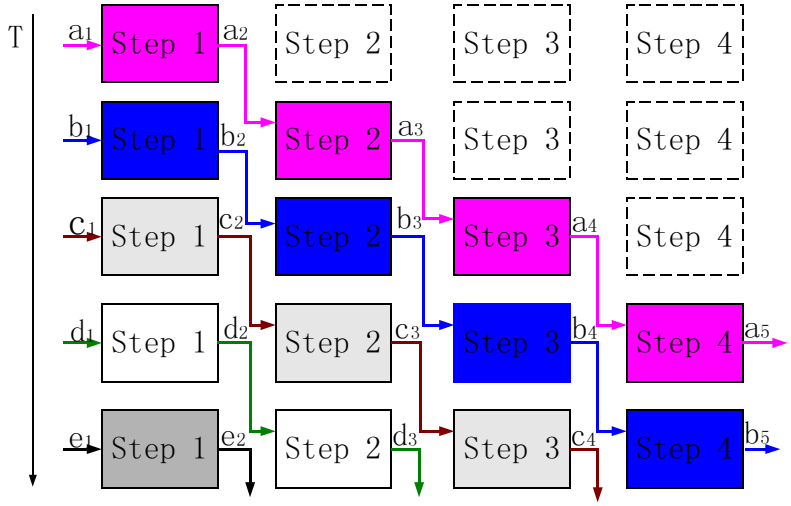


图1-11　流水线设计时序示意图

流水线设计的一个关键在于：整个设计时序的合理安排、前后级接口间数据流速的匹配。这就要求每个操作步骤的划分必须合理，要统筹考虑各个操作步骤间的数据流量。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。如果前级操作时间小于后级的操作时间，则需要对前级的输出数据适当缓存，才能汇入后级，还必须注意数据速率的匹配，防止后级数据的溢出。如果前级操作时间大于后级的操作时间，则必须通过逻辑复制、串并转换等手段将数据流分流，或者在前级对数据采用存储、后处理方式，否则会造成与后级的除了节拍不匹配。

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。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。

- **设计数据接口同步是否需要添加约束？**

建议最好添加适当的约束，特别是对于高速设计，一定要对周期、建立、保持时间等添加相应的约束。

这里附加约束的作用有 2:

(1)提高设计的工作频率，满足接口数据同步的要求。

通过附加周期、建立时间、保持时间等约束可以控制逻辑的综合、映射、布局和布线，以减小逻辑和布线延时，从而提高工作频率，满足接口数据同步的要求。

(2)获得正确的时序分析报告

几乎所有的 FPGA 设计平台都包含静态时序分析工具，利用这类工具可以获得映射或布局布线后的时序分析报告，从而对设计的性能做出评估。静态时序分析工具以约束作为判断时序是否满足设计要求的标准，因此要求设计者正确输入约束，以便静态时序分析工具输出正确的时序分析报告。

- **与数据接口同步相关的常用约束都有那些？**

下面分别介绍 Xilinx 和 Altera 关于数据接口同步的常用约束。

1. Xilinx 可以附加的约束有：Period、OFFSET_IN_BEFORE、OFFSET_IN_AFTER、OFFSET_OUT_BEFORE 和 OFFSET_OUT_AFTER 等约束。

(1) 周期约束 Period

Xilinx 的周期定义如图 12 所示。

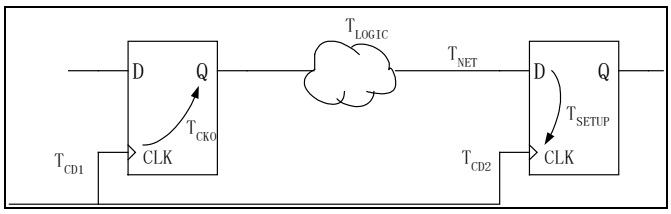


图1-12　时钟周期的计算

时钟的最小周期定义为为:

$T_{CLK} = T_{CKO} + T_{LOGIC} + T_{NET} + T_{SETUP} - T_{CLK\_SKEW}$

$T_{CLK\_SKEW} = T_{CD2} - T_{CD1}$

其中 T_{CKO} 为时钟输出时间，T_{LOGIC} 为同步元件之间的组合逻辑延迟，T_{NET} 为

网线延迟，T_{SETUP} 为同步元件的建立时间，T_{CLK_SKEW} 为时钟信号延迟的差别。

(2) OFFSET_IN_BEFORE 约束和 OFFSET_IN_AFTER 约束

OFFSET_IN_BEFORE 和 OFFSET_IN_AFTER 都是输入偏移约束，OFFSET_IN_BEFORE 说明了输入数据比有效时钟沿提前多长时间准备好，于是芯片内部与输入引脚相连的组合逻辑的延迟就不能大于该时间，否则有效时钟沿到来时，数据还没有稳定，采样将发生错误；OFFSET_IN_AFTER 指出输入数据在有效时钟沿之后多长时间到达芯片的输入引脚，这样也可以得到芯片内部延迟的上限，从而对与输入引脚相连的组合逻辑进行约束。

输入数据到达芯片引脚延迟的计算参见图 13 所示。

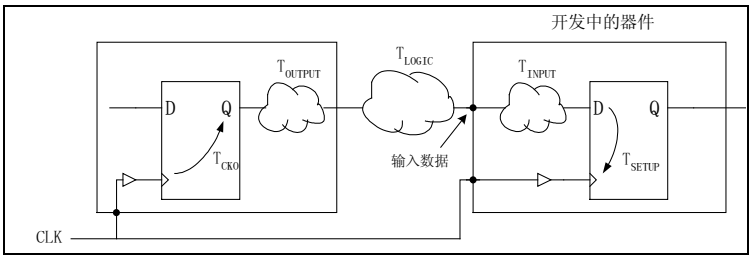


图1-13　输入到达时间的计算

计算公式如下:

$T_{ARRIVAL}=T_{CKO}+T_{OUTPUT}+T_{LOGIC}$

即输入数据在有效时钟沿之后的 $T_{ARRIVAL}$ 时刻到达。有了这个数据之后，可以对设计输入端附加 OFFET_IN_AFTER 约束。例如，NET DATA_IN OFFET=IN $T_{ARRIVAL}$ AFTER CLK，这样在对同步元件输入端的逻辑进行约束后，综合实现工具会努力使输入端的延迟 T_{INPUT} 满足以下关系:

$T_{ARRIVAL}+T_{INPUT}+T_{SETUP}<T_{CLK}$

其中 T_{INPUT} 为输入端的组合逻辑、网线和 PAD 的延迟之和，T_{SETUP} 为输入同步元件的建立时间。

(3) OFFSET_OUT_AFTER 约束和 OFFSET_OUT_BEFORE 约束

两者都是输出偏移约束，OFFSET_OUT_AFTER 规定了输出数据在有效时钟沿之后多长时间稳定下来，那么芯片内部的输出延迟就必须小于这个值。OFFSET_OUT_BEFORE 指出下一级芯片的输入数据应在有效时钟沿之前多长时间准备好。

从下一级输入端的延迟可以计算出当前设计输出的数据必须在何时稳定下来，根据这个数据对设计输出端的逻辑布线进行约束，以满足下一级的建立时间要求，保证下一级采样的数据是稳定的。计算要求的输出稳定时间如图 14 所示。

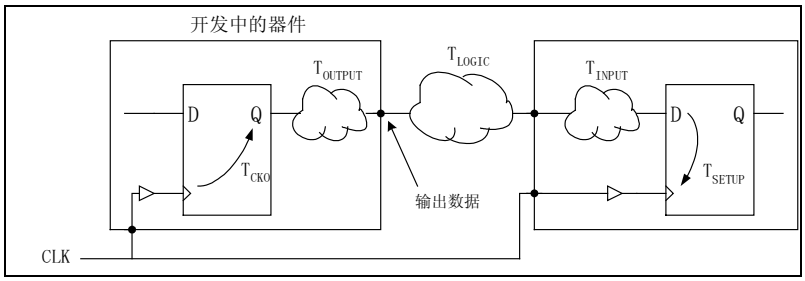


图1-14　计算要求的输出稳定时间

公式如下：

$T_{STABLE} = T_{LOGIC} + T_{INPUT} + T_{SETUP}$

那么只要当前设计输出端的数据比时钟上升沿提早 T_{STABLE} 时间稳定下来，下一级就可以正确地采样数据。

根据这个数据可以对设计输出端附加 OFFET_OUT_BEFORE 约束，例如，NET DATA_OUT OFFET=OUT T_{STABLE} BEFORE CLK，这样对同步元件输出端的逻辑进行约束后，综合实现工具会努力调整该逻辑的实现，使输出端的延迟满足以下关系：

$T_{CKO} + T_{OUTPUT} + T_{STABLE} < T_{CLK}$

其中 T_{OUTPUT} 为设计中连接同步元件输出端的组合逻辑、网线和 PAD 的延迟之和，T_{CKO} 为同步元件时钟输出时间。

2. Altera 可以附加的约束有：Period、tsu、tH tco 等约束。具体意义如下：

(1) Altera 的周期定义如图 15 所示，公式描述如下：

Clock Period = Clk-to-out + Data Delay + Setup Time - Clk Skew

即：

Tclk = Tco + B + Tsu -(E-C)

Fmax = 1/Tclk

注：对比一下前面的介绍，只要理解了 B 包含了两级寄存器之间的所有 logic 和 net 的延时就会发现与 Xilinx 周期定义公式完全一致。

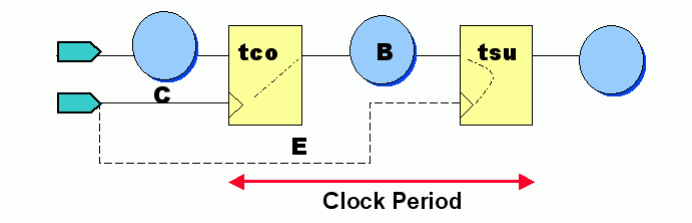


图1-15　Altera 的周期概念示意图

(2) Clock Setup Time (tsu)

要想正确采样数据，就必须使数据和使能信号在有效时钟沿到达前就准备好，所谓时钟建立时间就是指时钟到达前，数据和使能已经准备好的最小时间间隔。Altera 的 Clock Setup Time 概念如图 16 所示。

tsu = Data Delay – Clock Delay + Micro tsu

注：这里定义 Setup 时间是对应于整个路径的，需要区别的是另一个概念 Micro tsu。Micro tsu 指的是一个触发器内部的建立时间，它是触发器的固有属性，一般典型值小于 1~2ns。在 Xilinx 等的时序概念中，称 Altera 的 Micro tsu 为 setup 时间，用 Tsetup 表示，请大家区分一下。

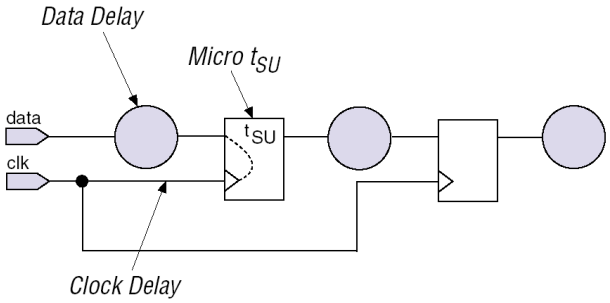


图1-16　Altera 的 Clock Setup Time 概念示意图

(3) Clock Hold Time （tH）

时钟保持时间是只能保证有效时钟沿正确采用的数据和使能信号的最小稳定时间。其定义如图 17 所示。定义的公式为：

tH= Clock Delay – Data Delay + Micro tH

注：其中 Micro tH 是指寄存器内部的固有保持时间，同样是寄存器的一个固有参数，典型值小于 1 ~ 2ns。

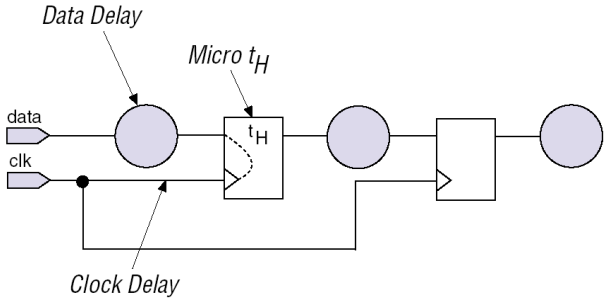


图1-17　Altera 的 Clock Hold Time 概念示意图

(4) Clock-to-Output Delay（tco）

这个时间指的是当时钟有效沿变化后，将数据推倒同步时序路径的输出端的最小时间间隔。如图 18 所示。

tco　=　Clock Delay + Micro tco + Data Delay

注：其中 Micor tco 也是一个寄存器的固有属性，指的是寄存器相应时钟有效沿，将数据送到输出端口的内部时间参数。它与 Xilinx 的时序定义中，有一个概念叫 Tcko 是同一个概念。

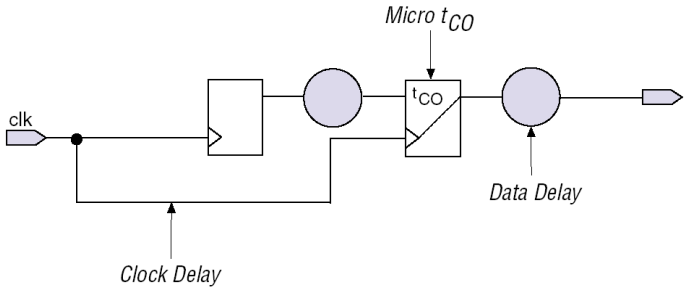


图1-18　Altera 的 Clock-to-Output Delay 概念示意图

1.9 常用模块之一：RAM/ROM/CAM 等存储单元

FPGA 中 RAM、ROM、CAM 的存储单元的设计有其自身规律和特点，与 ASIC 的设计方法有较大差异，对于 FPGA 初学者而言，RAM 等存储器件的设计与使用是比较头疼的问题。在此希望通过本节的讨论，解决初学者的一些疑惑，并引起对该问题的足够重视。

RAM 的一般概念和功能读者应该非常熟悉了，在此不再冗述。在 FPGA 中其实并没有专用的 ROM 硬件资源，实现 ROM 的思路是对 RAM 赋予初值值，并保持该初值。所谓 CAM，是 Content Addresssable Memory 的缩写，即内容地址储存器。CAM 这种存储器在其每个存储单元都包含了一个内嵌的比较逻辑，写入 CAM 的数据会和其内部存储的每一个数据进行比较，并返回与端口数据相同的所有内部数据的地址。概括的讲，RAM 是一种根据地址读、写数据的存储单元；而 CAM 和 RAM 恰恰相反，它返回的是与端口数据相匹配的内部地址。CAM 的应用也非常广泛，比如在路由器中的地址交换表等。

FPGA 中良好的 RAM、ROM、CAM 等存储单元的设计使用方法概括为一句话是：根据所选器件的架构特点，最有效使用 FPGA 内部资源，以合理的面积使用率，获得高的工作频率，以及稳定可靠的工作性能。

1.9.1 FPGA 中可以综合实现为 RAM/ROM/CAM 的 3 种物理资源

当代 FPGA 的发展突飞猛进，满足专用需求的底层硬件资源越来越丰富。与 RAM 等存储单元相关的资源有三类：Block RAM、LUT、Register，这三种资源都可以被综合实现为 RAM 等存贮单元。Block RAM 即通常所称的块 RAM，有时也被成为 Embedded Block RAM，指当代 FPGA 中内嵌的存贮单元，这种 RAM 单元可以通过配置和 glue logic(粘合逻辑)实现为单口/双口 RAM、ROM、CAM、FIFO 等。为了满足 SOPC 和复杂应用，高端 FPGA 中的内嵌 Block RAM 的容量与可配置的端口数量越来越丰富。目前某些高端器件的内嵌 Block RAM 的容量超过 10Mbit。无论那家生产商的 FPGA，其底层可编程逻辑单元无一例外的都是基于触发器（FF）和查找表（LUT，常用 4 输入查找表 4－LUT）结构的。这两种基本可编程资源都可以综合并实现为 RAM 等存贮单元。比较小，速度高的存储结构可以用寄存器资源实现；相对大一些，时序余量宽松的存贮单元可以用 LUT 实现。目前很多综合工具支持类推 RAM、ROM、CAM 等存贮单元使用 Block RAM、Register，或者 LUT 资源。下面强调几点使用不同资源综合为 RAM 等存贮器的注意事项。

1. 生产 RAM 等存储单元时，应该首选 Block RAM 资源

特别需要强调的是 Block RAM 资源的使用。由于 Block RAM 资源是 FPGA 内嵌的一种独立资源，所以推荐在生产 RAM、ROM、CAM 等存贮单元时，尽量首先使用 Block RAM，其原因有二：第一，使用 Block RAM 资源，可以节约更多的 FF 和 4－LUT 等底层可编程单元。使用 Block RAM 可以说是“不用白不用”，是最大程度发挥所选器件效能，节约成本的一种体现；第二，Block RAM 是一种可以配置的硬件结构，其可靠性和速度与用 LUT 和 Register 构建的存贮器更有优势。

2. 弄清 FPGA 的硬件结构，合理使用 Block RAM 资源

使用 Block RAM 时，必须要配合所选器件的结构。反之，器件选型时也必须要考虑到设计中使用到的 RAM、ROM、CAM 等存贮单元的容量、端口数量、结构等。一般来说用 Blcok RAM 可以构建出带有同步复位/置位、异步复位/置位，可选输入寄存器采样，可选输出寄存器采样等不同应用的单口/双口/多口 RAM、ROM、CAM、同步/异步 FIFO 等典型存贮单元。在设计 RAM 的端口时，必须要根据 Block RAM 的特点，不可随意设计。

比如某个设计为了能够在一个节拍一次读出 256 个数据，于是就写了一个单端口写入，256 端口读出的 RAM。事实上，目前没有任何一款 FPGA 的 Block RAM 的端口数量 256 左右！该 RAM 如果深度和宽度较大，也不宜于使用 FF 或者 4－LUT 实现，那样将会消化巨大的可编程逻辑资源；如果用 Blcok RAM 实现，需要多个 Block RAM 拼接起来才能完成 256 个读端口，且不论拼接多个 Block RAM 需要的复杂的 glue logic 将会消耗巨大的逻辑资源，把很多 Block RAM 拼接起来，如果每个 Block RAM 的容量都没有使用完，将会造成非常大的 Block RAM 资源的浪费。

目前大多数 Block RAM 硬件上真正的端口配置是 2 port（true dual-read/write port）或者 4 port（true quad-read/write port）的，设置者在使用时应该查证该器件的器件手册，尽量使用端口数量匹配的 RAM，此时消耗的 glue logic 最少，对 Block RAM 的利用率最高。在 FPGA 设计中，对于上面的那个例子应该摒弃使用 256 端口读的 RAM，改用其它设计方法。例如在设计频率允许时，可以高速从 1 个或者一组 DPRAM（双口 RAM）、QPRAM（4 端口 RAM）读数据，然后把读出的数据缓存等齐。具体选择几块 RAM 和深度应该根据其器件速度和读写速度确定。通过提高频率、缓冲等齐、状态机控制等方法完全可以将那个 256 端口读的设计改为等效的 DPRAM 或 QPRAM 的设计。从而避免了对逻辑资源的浪费，和物理实现上的问题。EDACN 论坛上一位工程师使用 Xilinx VirtexII1000 器件实现上述设计，改进前资源利用率大于 40%的 slice，最高时钟频率 Max clock frequency 为 32MHz；改进后资源利用率为 6% slice＋2blockRAM，最高时钟频率 Max clock frequency 为 140MHz。

3. 分析 Block RAM 容量，高效使用 Block RAM 资源

上面举是没有考虑到 Block RAM 的端口数量，导致无法有效使用 Block

RAM 的例子。其实 Block RAM 还有很多特性非常重要，比如硬件上 Block RAM 的分块数量、单块的容量、单块支持的存储深度和数据宽度等。每种 FPGA 的 Block RAM 都是以“块”的形式组织的，每块是一个一个相对独立的可配置存储器，支持不同存储深度和数据宽度的组合。有的 FPGA 里面只有一种 Block RAM 的块大小，相对比较好掌握，但是 Altera FPGA 为了加强 Blcok RAM 的利用率，为用户提供更灵活的存贮方案，在其高端 FPGA 中，提供了多种不同容量的 Block RAM。比如其 Stratix/Stratix GX 等系列 FPGA 中有一种叫 TriMatrix Memory 的 Block RAM 架构，其特点是在芯片中内嵌了三种不同大小的 RAM 块，分别是：512Bits 的 Block RAM，简称为 M512 Block；4Kbits 大小的 Block RAM，简称为 M4K Block，和 512Kbits 的 Block RAM，简称为 M－RAM Block。这三种 Block 的架构不同，等效到每 Kbit 的端口数量也不同，在不同应用范围的芯片中三种 Block 的数量按照一定比例配置。这种不同大小的 Block RAM 结构能够较好的满足不同项目的需求，例如 M512 可以被实现为容量较小的 FIFO、RAM、ROM，或者应用于在前面 1.8 小节介绍的“数据接口的同步方法”中完成异步时钟域之间数据交换的 DPRAM 等；M4K 应用最广，可以实现为较大容量的 RAM、ROM、CAM、FIFO 等，可以存储数据的索引表，处理器代码，中间结果等；容量的最大的 M-RAM 是专为大容量存贮而设计的，可以完成通信系统，网络交换器等设备中的大容量数据存贮。使用这种分层次，不同大小的 Block RAM 的最大好处就是，给用户提供了多样的选择，用户根据自己存储单元大小选择需要使用的 Block RAM，可以较大程度的提高 Block RAM 的利用率。设置使用何种大小的 Block RAM 实现存储器的最简单方法是，在 Altera Quartus 的 IP 生成器 MegaWizard 中直接设置所选 IP core 实现在那种规格的 Block RAM 中，如图 1－19 所示。

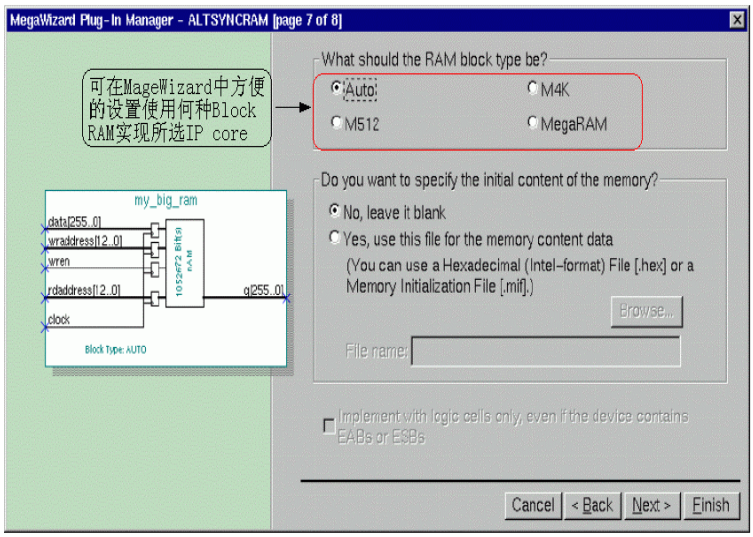


图1-19　在 MegaWizard 中设置使用何种规格的 Block RAM 实现所选 IP core

在使用 Block RAM 时，一定要注意端口宽度和容量的匹配问题。又如在 ALTERA APEX-II 系列器件中实现深度为 128，宽度为 64bits 的 DPRAM，其容量为 128 × 64bit = 8192Bits，似乎恰好能在 2 块 M4K 中实现。事实上实现这个 128 × 64bits 的 DPRAM 需要 4 块 M4K，这是因为一块 4Kbits Block RAM 的最大数据宽度是 16bit，实现数据宽度是 64bits 的 DPRAM 必须使用 4 快 M4K 才能满足端口宽度，这样的实现方式会造成 8Kbits RAM 资源的浪费。在设计时和容量估计时必须要考虑。在 Xilinx 和 LatticeFPGA 中也存在同样的问题，所以说要合理高效的使用 Block RAM 资源，必须要了解器件的硬件结构和特性。

4. 分布式 RAM 资源（distributed RAM）

目前 FPGA 的底层可编程单元基本都是由 4－LUT 和 FF 构成的，其工艺基于 SRAM 结构。4－LUT 的本质是 16 逻辑真值表，覆盖了 4 × 4 的所有逻辑。Xilinx 基于 SRAM 结构的 4－LUT 加以特殊的地址运算和时钟结构，实现了对 1bit 数据的 16 个可选地址的存取，相当于深度为 16 宽度为 1bit RAM。由于这种小 RAM 分布于每个 FPGA 的底层可编程单元，所以被成为分布式 RAM。Lattice 的 ispXPGA 也可以将底层可配置单元综合成 distributed RAM。分布式 RAM 的存于 FPGA 可编程单元之内，到底可编程逻辑的路径最短，非常易于实现灵活、高速、小容量的数据缓冲、FIFO 等。

Xilinx FPGA 支持 distributed RAM 的器件族有：Spartan-3、Spartan-II/Spartan-IIE、Virtex/Virtex-E、Virtex-II/Virtex-II Pro 等器件族。Lattice FPGA 支持 distributed RAM 的器件族为基于 ispXPGA 的所有器件族。下面以 Xilinx FPGA 为例简单介绍一下 distributed RAM 的特点，其特点如下：

- Distributed RAM 的机制是同步写、异步读的，但是只需将 Slice 内部与 LUT 连接也 FF 利用起来就可以实现同步写、同步读的 RAM。
- 将同一个 SLICEM 单元里面的 2 个 LUT 连接起来，还可以实现深度为 16，宽度为 1bit 的双口 RAM，这种实现方式也比较好理解，相当于其中一个 LUT 实现了带有读/写端口的 16 × 1bit 的 RAM，另一个 LUT 实现为仅有度端口的 16 × 1bit 的 RAM。按照同步时序同时在两个 LUT 实现的 16 × 1bit RAM 写入数据，然后在第二个 LUT 实现的 RAM 中独立的读出数据即可。所以 distributed RAM 支持的存储类型有：单口、双口；同步写/异步读、同步写/同步读等不同类型。图 1－20 为单口、双口 distributed RAM 的基本硬件原语（primitive）示意图。

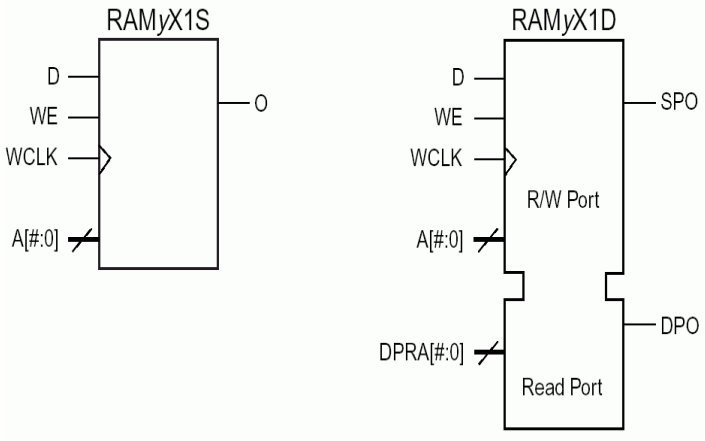


图1-20　单口、双口 distributed RAM 的硬件原语（primitive）示意图

- 前面介绍了 1 个 4－LUT 可以实现深度为 16，宽度为 1bit 的 distributed RAM，如果需要实现更大的 RAM 时，就需要将 Slice 或者 CLB 内部的 distributed RAM 级连起来。一个 CLB 内部可以实现的 distributed RAM 的大小取决于该 CLB 的 4－LUT 的数量。表 1－1 为 Xilinx 不同器件族一个 CLB 可实现的 distributed RAM 资源数量表。表 1－2 为 Xilinx 不同器件族一个 CLB 可实现的 distributed RAM 的类型表。

表1－1.　Xilinx 不同器件族一个 CLB 可实现的 distributed RAM 资源数量表

Feature	Spartan-3 Family	Virtex/Virtex-E, Spartan-II/Spartan-IIE Families	Virtex-II, Virtex-II Pro Families
LUTs per CLB	8	4	8
ROM bits per CLB	128	64	128
Single-port RAM bits per CLB	64	64	128
Dual-port RAM bits per CLB	32	32	64

表1－2.　Xilinx 不同器件族一个 CLB 可实现的 distributed RAM 的类型表

Family	Single-Port RAM				Dual-Port RAM		
	16x1	32x1	64x1	128x1	16x1	32x1	64x1
Spartan-3	4	2	1		2		
Spartan-II/IIE Virtex/E	4	2	1		2		
Virtex-II/Pro	8	4	2	1	4	2	1

- Distributed RAM 资源除了支持单口、双口 RAM 以外，如果被赋予固顶的值还可以实现为 ROM，同样也可以实现为 FIFO。

- Distributed RAM 的生成方式也支持代码直接描述，附加综合实现属性或使用器件商的 IP core 生成器产生等方式。如 Xilinx FPGA 可在 Verilog 或 VHDL 代码中附加约束属性，或者用 CoreGenerator 生成。对 RAM/ROM 等指定初值可使用 CoreGenerator 产生扩展名为“ceo”的初值文件或者用“INIT”指示原语实现。关于如何生成 Distributed RAM 和为其设置初值，在后面章节会有详细介绍。

除了 Xilinx 的 FPGA 有 Distributed RAM 外，Lattice FPGA 也有这种结构，Lattice FPGA 的 Distributed RAM 支持 64bit 单口 RAM、32bit 双口 RAM 和 32bit 移位寄存器，其特性与 Xilinx FPGA 一致，生成也非常方便。

1.9.2 RAM/ROM/CAM 等存储单元的三种基本生成方式

RAM、ROM、CAM 等存储单元有三种基本的实现方法：使用 HDL 代码描述、使用综合约束属性例化或类推、使用器件商的 IP core 生成器。其实这 3 中描述方法很具代表性，典型功能的 IP core 都可以通过这 3 种途径实现。其中前两种方法需要学习综合 RAM、ROM、CAM 等存储单元的 Coding Style 或约束属性，后一种方法非常方便直接，建议初学者首先要掌握使用器件商 IP core 生成器设计 RAM、ROM、CAM 等存储单元的方法。下面依次分别讲解。

1. 直接在 HDL 代码中描述的方法

使用代码描述 RAM、ROM、CAM 等存储单元包含两种含义，其一是用直接用 HDL 代码描述这些存储单元的模型，然后让综合器综合类推。这种方法在 FPGA 设计中不推荐，因为自由的让综合器类推，很难有效的控制所使用的 RAM 等存储器的所使用的资源类型和面积速度等关键指标；第二种方法是直接调用器件商提供的与这些存储单元相关的硬件原语。这种方法虽然实现目标明确，但是对于相对复杂的设计，用户需要手动完成如 glue logic 等辅助逻辑，设计工作量巨大，而且要求用户非常属性器件商的硬件原语和器件的内部结构，所以也不推荐。

例 5. 直接使用 HDL 代码描述 RAM 等存储单元

这是一个反例，用以说明直接使用 HDL 代码描述 RAM 等存储单元的缺陷。很多初学者根据 Verilog 或 VHDL 的语法书用下面的语句描述一个宽度是 8bit，深度是 16 的带有异步复位、使能的 RAM:

```
module ram_WR_EN(A,CLK,D,WR,EN,RST,Q);
input [7:0] D;
input WR,EN;
input [3:0] A;
input CLK,RST;
output [7:0] Q;
```

```
reg [7:0] mem[15:0];
reg [7:0] Q;

integer i;
always @(posedge CLK or posedge RST)
begin
  if(RST)
  begin
    Q=0;
    for(i=0;i<15;i=i+1)
    mem[i]=0;
  end
  else
  begin
    if(EN)
    begin
      if(WR)
        mem[A]=D;
      else
        Q=mem[A];
    end
  end
end

endmodule
```

我们简单的用 Synplify Pro 对上面那段代码综合一下，选择器件为 Xilinx Virtex2 XC2V40－6 CS144，综合优化参数选取默认值。通过综合，发现实际结果并没有用 Xilinx 的 Block RAM 实现，而是用 FF 和 LUT 完成了上述功能，其 RTL View 如图 1－21 所示。

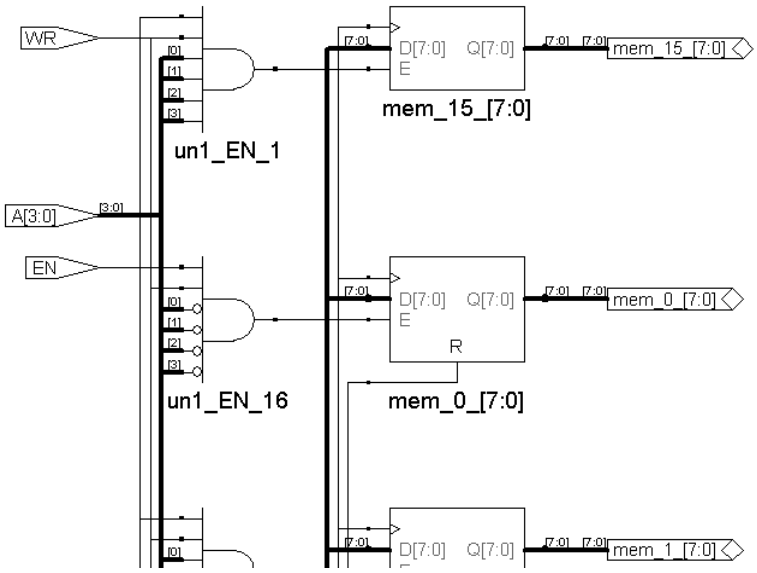


图1-21　该异步复位 RAM 在 Synplify Pro 的综合结果的 RTL 视图

该异步复位 RAM 在 Synplify Pro 的综合结果的 Technical View 的部分图形如图 1－22 所示。

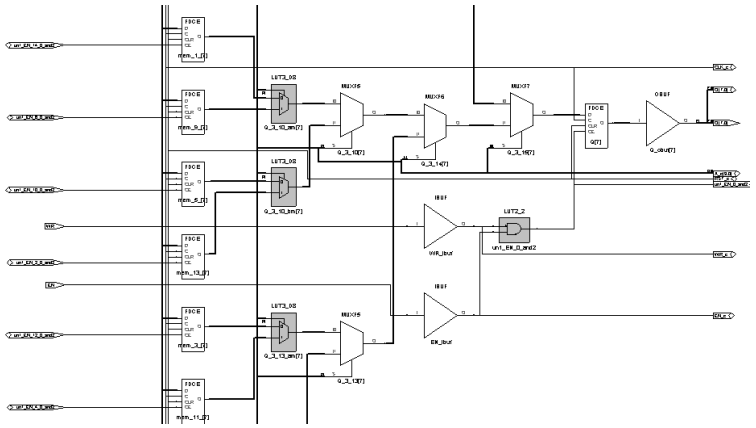


图1-22　该异步复位 RAM 在 Synplify Pro 的综合结果的 Technical View（部分）

可见综合结果并未如设计者所愿，综合为 RAM。

为了能够综合成 RAM 可以去除上述代码中关于异步复位的部分，修改为:

```
module ram_WR_EN(A,CLK,D,WR,EN,RST,Q);
input [7:0] D;
input WR,EN;
input [3:0] A;
input CLK,RST;
output [7:0] Q;
```

```
reg [7:0] mem[15:0];
reg [7:0] Q;

always @(posedge CLK or posedge RST)
begin
  if(RST)
  begin
    Q <= 0;
  end
  else
  begin
    if(EN)
    begin
      if(WR)
        mem[A] <= D;
      else
        Q <= mem[A];
    end
  end
end


endmodule
```

此时在 Synplify Pro 中完全相同的综合条件下，综合结果为 RAM。代码修改后的综合结果的 RTL 视图如图 1－23 所示。

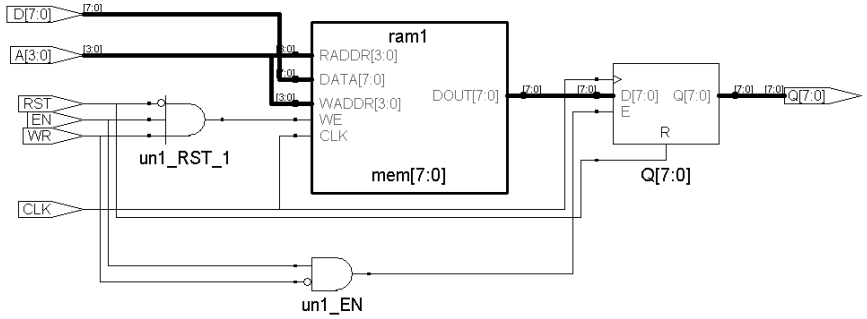


图1-23　代码修改后的综合结果的 RTL 视图

代码修改后的综合结果的 Technical 视图（部分）如图 1－24 所示。从图中可以清醒的看到 Synplify Pro 自动将该代码类推使用 RAM16X8S 这一 Select RAM 的硬件原语。

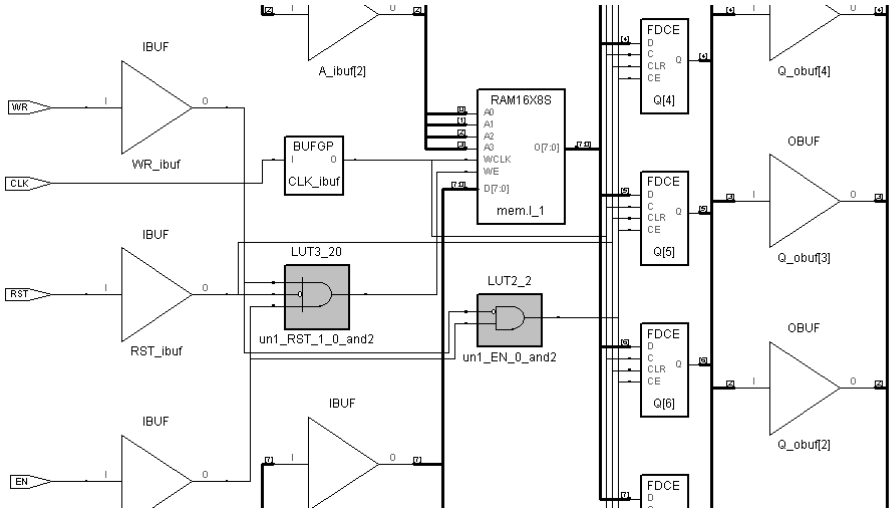


图1-24　代码修改后的综合结果的 Technical 视图（部分）

特别需要强调的 3 点是：

- 举这个例子的目的并不是告诉读者如何修改代码以达到时综合工具能够自动类推 RAM。这个例子是要告诉大家，直接用 HDL 代码描述 RAM 等存储单元，而不通过约束属性等方式控制综合器，常常会综合出与设计意图不相符的硬件结构。
- 更有甚者，当上述代码中描述的数据宽度和数据深度变化，或者采用不同综合工具，甚至是同一综合工具的不同版本对上述代码进行综合时都会获得不同的映射结构，有时会采用 FF 加 LUT 实现，有时会采用 Block RAM 实现，控制困难。
- 那么上述代码描述方式在何时使用呢？这种代码一般是在系统级或者行为级仿真过程中使用。并不直接用来综合实现，在综合时要配合约束属性，指导综合结构。

下面以 Xilinx FPGA 为例，直接调用和 RAM 相关的硬件原语，例化 RAM 等存储单元。由于和 RAM 相关的 primitive 非常多，所以仅选取几个典型的 Verilog 和 VHDL 实例化描述。

例 6. 双口 RAM 的 VHDL 实例化代码

下面是端口 A:8192 x 2 bits，B: 8192 x 2 bits 的双口 RAM 的 VHDL 实例化代码，使用“RAMB16_S2_S2” Block SelectRAM 硬件原语，适用于 Spartan-3 等器件族。

```
-- Components Declarations:
--
component RAMB16_S2_S2
-- pragma translate_off
```

```
      generic (
    -- "Read during Write" attribute for functional simulation
       WRITE_MODE_A : string := "READ_FIRST" ; -- WRITE_FIRST(default)/
READ_FIRST/ NO_CHANGE
       WRITE_MODE_B : string := "WRITE_FIRST" ; -- WRITE_FIRST(default)/
READ_FIRST/ NO_CHANGE
    -- RAM initialization ("0" by default) for functional simulation: see
example
       );
    -- pragma translate_on
      port (
       DIA     : in std_logic_vector (1 downto 0);
       ADDRA   : in std_logic_vector (12 downto 0);
       ENA     : in std_logic;
       WEA     : in std_logic;
       SSRA    : in std_logic;
       CLKA    : in std_logic;
       DOA     : out std_logic_vector (1 downto 0);
    --
       DIB     : in std_logic_vector (1 downto 0);
       ADDRB   : in std_logic_vector (12 downto 0);
       ENB     : in std_logic;
       WEB     : in std_logic;
       SSRB    : in std_logic;
       CLKB    : in std_logic;
       DOB     : out std_logic_vector (1 downto 0)
       );
    end component;
    --
    ------------------------------------------------------------------------
    --
    -- Architecture  section:
    --
    --  Attribute  "Read  during  Write  mode"  =  WRITE_FIRST(default)/
READ_FIRST/ NO_CHANGE
    attribute WRITE_MODE_A : string;
    attribute WRITE_MODE_A of U_RAMB16_S2_S2: label is "READ_FIRST";
    attribute WRITE_MODE_B : string;
    attribute WRITE_MODE_B of U_RAMB16_S2_S2: label is "WRITE_FIRST";
```

```
-- Attributes for RAM initialization ("0" by default): see example
--
-- Block SelectRAM Instantiation
U_RAMB16_S2_S2: RAMB16_S2_S2
  port map (
    DIA    => , -- insert 2 bits data in bus (<1 downto 0>)
    ADDRA  => , -- insert 13 bits address bus
    ENA    => , -- insert enable signal
    WEA    => , -- insert write enable signal
    SSRA   => , -- insert set/reset signal
    CLKA   => , -- insert clock signal
    DOA    => , -- insert 2 bits data out bus (<1 downto 0>)
--
    DIB    => , -- insert 2 bits data in bus (<1 downto 0>)
    ADDRB  => , -- insert 13 bits address bus
    ENB    => , -- insert enable signal
    WEB    => , -- insert write enable signal
    SSRB   => , -- insert set/reset signal
    CLKB   => , -- insert clock signal
    DOB    =>  -- insert 2 bits data out bus (<1 downto 0>)
    );
```

例 7. 双口 RAM 的 Verilog 实例化代码

和上面例子相同，该双口 RAM 的端口 A:8192 x 2 bits，B: 8192 x 2 bits，使用“RAMB16_S2_S2”Block SelectRAM 硬件原语，适用于 Spartan-3 等器件族。

其中注释内容为 Synopsys 等综合工具的综合约束属性。

```
// Syntax for Synopsys FPGA Express
// synopsys translate_off
  defparam
//"Read during Write" attribute for functional simulation
   U_RAMB16_S2_S2.WRITE_MODE_A    =    "READ_FIRST"    -type    string;
//WRITE_FIRST(default)/ READ_FIRST/ NO_CHANGE
   U_RAMB16_S2_S2.WRITE_MODE_B    =    "WRITE_FIRST"    -type    string;
//WRITE_FIRST(default)/ READ_FIRST/ NO_CHANGE
//RAM initialization ("0" by default) for functional simulation
// synopsys translate_on

//Block SelectRAM Instantiation
```

```
    RAMB16_S2_S2 U_RAMB16_S2_S2   (   .DIA(),    //insert 2-bit data_in bus
([1:0])
                            .ADDRA(),  //insert 13-bit address bus ([12:0])
                            .ENA(),    //insert enable signal
                            .WEA(),    //insert write enable signal
                            .SSRA(),   //insert set/reset signal
                            .CLKA(),   //insert clock signal
                            .DOA(),    //insert 2-bit data_out bus ([1:0])
    //
                           .DIB(),    //insert 2-bit data_in bus ([1:0])
                            .ADDRB(),  //insert 13-bit address bus ([12:0])
                            .ENB(),    //insert enable signal
                            .WEB(),    //insert write enable signal
                            .SSRB(),   //insert set/reset signal
                            .CLKB(),   //insert clock signal
                            .DOB()     //insert 2-bit data_out bus ([1:0])
                           );
    // Attribute Decalrations:
      /* synopsys attribute
    Attribute "Read during Write mode" = WRITE_FIRST(default)/ READ_FIRST/
NO_CHANGE
       WRITE_MODE_A "READ_FIRST"
       WRTIE_MODE_B "WRITE_FIRST"
       */
```

例 8. 单口分布式 RAM 的 VHDL 实例化代码

该分布式 RAM 的结构为 32×1bit，代码如下：

```
    -- Components Declarations:
    --
    component RAM32X1S
    -- pragma translate_off
      generic (
    -- RAM initialization ("0" by default) for functional simulation:
          INIT : bit_vector := X"00000000"
       );
    -- pragma translate_on
      port (
       D    : in std_logic;
          WE   : in std_logic;
```

```
      WCLK : in std_logic;
      A0   : in std_logic;
      A1   : in std_logic;
      A2   : in std_logic;
      A3   : in std_logic;
      A4   : in std_logic;
      O    : out std_logic
  );
end component;
-----------------------------------------------------------------------
-- Architecture  section:
--
-- Attributes for RAM initialization ("0" by default):
attribute INIT: string;
--
attribute INIT of U_RAM32X1S: label is "00000000";
--
-- Distributed SelectRAM Instantiation
U_RAM32X1S: RAM32X1S
  port map (
   D     => , -- insert input signal
   WE    => , -- insert Write Enable signal
   WCLK  => , -- insert Write Clock signal
   A0    => , -- insert Address 0 signal
   A1    => , -- insert Address 1 signal
   A2    => , -- insert Address 2 signal
   A3    => , -- insert Address 3 signal
   A4    => , -- insert Address 4 signal
   O     =>  -- insert output signal
   );
```

例 9. 单口分布式 RAM 的 Verilog 实例化代码

下面是该分布式 RAM 的 Verilog 实例化代码，其中注释内容为 Synopsys 等综合工具的综合约束属性。

```
// Syntax for Synopsys FPGA Express
// synopsys translate_off

  defparam
```

```
  //RAM initialization ("0" by default) for functional simulation:
  U_RAM32X1S.INIT = 32'h00000000;
// synopsys translate_on


//Distributed SelectRAM Instantiation
RAM32X1S U_RAM32X1S  (  .D(),    //insert input signal
              .WE(),   //insert Write Enable signal
              .WCLK(), //insert Write Clock signal
              .A0(),   //insert Address 0 signal
              .A1(),   //insert Address 1 signal
              .A2(),   //insert Address 2 signal
              .A3(),   //insert Address 3 signal
              .A4(),   //insert Address 4 signal
              .O()     //insert output signal
             );

// synthesis attribute declarations
  /* synopsys attribute

   INIT "00000000"
  */
```

可以看到这种调用和 RAM 等存储单元相关的硬件原语，直接例化的方法的方法虽然保证了综合和实现时使用用户指定的 RAM 资源（如 Block RAM 和 Distributed RAM 等），但是需要用户对所用器件的硬件原语十分熟悉才可以正确使用。另外当用户设计的 RAM 等存贮单元的宽度和深度与需要使用的 Block RAM 和 Distributed RAM 资源实际硬件容量不十分相符时，必须手工设计粘合几块 Block RAM 或者几个 Slice 之间关系的 glue logic，设计工作量很大。

2. 使用综合约束属性指导综合器类推 RAM 等存储单元

这种方式的优点是使用灵活、高效；缺点是要求设计者对目标器件的结构和综合工具的约束属性熟悉程度高。前面介绍了，单纯依靠综合工具的自动类推综合 RAM、ROM、CAM 等存贮单元是非常不可靠的，可能因为综合工具的不同或者同一个综合工具的版本不同，或者代码描述的风格以及 RAM 等存贮单元的深度、宽度等变化得到不一致的映射结构，有时用 FF + LUT 实现，有时可能使用器件的 Blcok RAM 资源，非常难以控制，是一种不推荐的设计方法。为了克服这种方法的不确定性，需要在综合时显化地指定需要综合的

RAM 等存储单元的结构，以及在目标器件的映射类型等。

一般性的讲，综合时附加约束属性的方法有 2 种：第一种是直接在代码中使用某种综合工具的约束属性关键字加约束指示原语；第二种方法是在综合工具的图形界面下输入约束属性。前一种方法使用灵活，可以覆盖几乎所有的约束属性；后一种方法使用简便，将用户从烦琐的约束属性记忆中解脱出来。

约束属性因综合工具不同而异，不同的综合工具的约束属性关键字不同，这里这能介绍 Synplicity、Synopsys 等典型综合工具综合 RAM 等存储单元的常用命令。

Synplicity 公司的 FPGA 综合工具主要有 Synplify、Synplify Pro、Amplify 等产品。Synplicity 综合工具与 RAM 相关的主要综合属性是“syn_ramstyle”。这个综合属性用于显化地指定所综合的 RAM 使用的资源类型和存储结构。根据不同的器件类型，“syn_ramstyle”可选的约束属性最多有以下 4 种：

- registers
  此属性值用以指定所综合的 RAM 类型为“寄存器型”。即用 FF+LUT 实现该存贮结构，如果使用此综合约束属性，则综合器会摒弃该器件的专有 RAM 硬件资源（如 Block RAM 等），而用 FF+LUT 实现 RAM 结构。
- block_ram
  此属性值用以指定所综合的 RAM 类型为“块 RAM”。即用该 FPGA 器件内部的 Block RAM 实现 RAM 结构，如果实现的 RAM 结构较复杂，综合器会自动生成 Block RAM 之间的 glue logic（粘合逻辑，用 FF+LUT 实现）。
- no_rw_check
  此属性值用以指定所综合的 RAM 类型为“无读写校验的 RAM”。该参数仅仅对 Xilinx Virtex/VirtexE/Virtex II/Virtex II Pro 和 Altera Stratix/Stratix GX 等器件族的 FPGA 有效。对于 Xilinx Virtex/VirtexE/Virtex II/Virtex II Pro 等器件族，使用该约束属性将会综合出无任何附加 glue logic 而仅仅用 Select RAM 实现的存储结构；对于 Altera Stratix/Stratix GX 等器件族，使用该约束属性将会综合出无任何附加 glue logic 而仅仅用 EAB 实现的存储结构。对同一地址同时进行读写操作将会导致输出为不定态，这种情况会造成 RTL 功能仿真和 PAR 后仿真的，所谓“读写校验”，就是用以检查这种不安全操作的附加逻辑，指定本参数后，Synplicity 综合工具将不生成这种附加检验逻辑，从而减少了逻辑负荷，但是取消读写校验逻辑，必须十分小心，否则会造成 RAM 存储器的可靠性下降。
- select_ram
  这个参数仅对 Xilinx 器件有效。当对 Xilinx FPGA 使用此属性值时，Synplicity 综合器会将所描述的 RAM 综合为 Distributed RAM（分布式 RAM），关于 Distributed RAM 的详细概念，请参考 1.9.1 节第 4 小点的论述。

不同厂商的 FPGA 可用的“syn_ramstyle”约束原语的属性值与综合类型如表 1-3 所示。

表1－3.　不同厂商的 FPGA 可用的“syn_ramstyle”属性值与综合类型对照表

器件商	器件族	Syn_ramstyle 可用的约束属性值	综合结构	默认情况
Altera	Flex 10K ACEX APEX	registers	FF+LUT	如果 altera_auto_use_eab=1 或 ture,则默认情况下，小容量 RAM 综合为 Registers 型，大容量为 EAB 或者 ESB 型。
		block_ram	EAB	
	Stratix Cyclone	registers	FF+LUT	
		block_ram	ESB	
		no_rw_check	无读写校验 glue logic 的 EAB	
Xilinx	XC4000	registers	FF+LUT	select_ram
		select_ram	Distributed RAM	
	Virtex VirtexE VirtexII VirtexII Pro	registers	FF+LUT	默认值为 block_ram（即尽量综合成 Block SelectRAM），如果无法综合为 Block SelectRAM 则综合为 Distributed RAM
		block_ram	Block SelectRAM	
		no_rw_check	无读写校验 glue logic 的 Block SelectRAM	
		select_ram	Distributed RAM	
Lattice	ORCA、ispXPGA 系列器件族	registers	FF+LUT	尽量使用 EBR（Embedded Block RAM）资源，如果无法使用 Block RAM，则尽量使用 Distributed RAM
		block_ram	EBR	

Synplicity 综合工具添加综合约束属性的方法比较灵活，有在 HDL 代码中直接附加综合约束属性，在 SCOPE 图形界面下设置约束属性，SDC 综合约束文件中添加约束属性等三种方法实现。

例 10.　在 Verilog 代码中附加 syn_ramstyle 综合约束属性，指定综合存贮单元的类型

在 Verilog 代码中指定 syn_ramstyle 的语法格式为：

```
object /* synthesis syn_ramstyle = "string" */ ;
```

其中“*object*”为 reg 类型的存储器信号；用黑体表示的“synthesis syn_ramstyle”中注释符号后面“synthesis”是 Synplicity 综合工具的 Verilog 约束属性通用关键字；“syn_ramstyle”是综合 RAM 类型的约束属性关键字；“string”根据所选器件类型，可以选

择 registers、block_ram、no_rw_check、select_ram 中的一种属性值。在 Verilog 代码中一般直接在 mem 型定义时附加该综合约束属性，如下面示例，将存储器“mem”定义为“registers”，即使用 FF＋LUT 实现“men”。代码如下：

```
reg [7:0] mem[31:0] /* synthesis syn_ramstyle="registers" */;
```

又如在上面举例代码中，定义 mem 使用 Block RAM 资源：

```
module ram_WR_EN(A,CLK,D,WR,EN,RST,Q);
input [7:0] D;
input WR,EN;
input [3:0] A;
input CLK,RST;
output [7:0] Q;

reg [7:0] mem[15:0] /* synthesis syn_ramstyle="block_ram" */;
reg [7:0] Q;
```

例 11.　在 VHDL 代码中附加 syn_ramstyle 综合约束属性，指定综合存贮单元的类型

在 VHDL 代码中指定 syn_ramstyle 的语法格式为：

```
attribute syn_ramstyle of object : object_type is "string" ;
```

其中 *object* 可以是 signal 也可以是器件的实例化；黑体表示的“**attribute syn_ramstyle of**”是 synpilicity 综合工具的 VHDL 综合约束属性关键字；"string"同样根据所选器件类型，选择 registers、block_ram、no_rw_check、select_ram 中的一种属性值。

在 VHDL 代码中指定“mem”信号使用 Block RAM 资源的示例如下：

```
library ieee;
use ieee.std_logic_1164.all;
entity ram4 is
 port (d : in std_logic_vector(7 downto 0);
           addr : in  std_logic_vector(2 downto 0);
           we : in  std_logic;
           clk : in  std_logic;
           ram_out : out std_logic_vector(7 downto 0));
end ram4;
    library synplify;
architecture rtl of ram4 is
type mem_type is array (127 downto 0) of std_logic_vector (7 downto 0);
signal mem : mem_type;
-- mem is the signal that defines the RAM
attribute syn_ramstyle : string;
```

```
attribute syn_ramstyle of mem : signal is "block_ram";
```

例 12.　在 SDC 文件中附加 syn_ramstyle 综合约束属性，指定综合存贮单元的类型

SDC 文件是 Synplicity 综合工具通用的综合约束属性文件，其扩展名为”sdc”。在 SDC 指定 syn_ramstyle 的语法格式为：

```
define_attribute {signal_name [bit_range]} syn_ramstyle {string}
```

其中，黑体字表示的“define_attribute”是 SDC 文件的约束属性关键字；“*signal_name* [*bit_range*]”是代码中需要综合为 RAM 等资源的信号名；黑体字表示的“syn_ramstyle”是综合 RAM 类型约束属性关键字；"string"同样根据所选器件类型，选择 registers、block_ram、no_rw_check、select_ram 中的一种属性值。

例如指定信号“mem[7:0]”综合为“register”类型的存储单元，需要在 sdc 文件中添加如下约束属性：

```
define_attribute {mem[7:0]} syn_ramstyle {registers};
```

另外除了手工在 SDC 文件中添加综合约束属性外，还可以直接在 Synplify Pro 等综合工具的 SCOPE 图形界面中直接设置所需的综合约束属性。SCOPE（Synthesis Constraints Optimization Environment）采用图形化集成界面，可以让用户可以方便、全面、有效地对设计进行综合约束。

使用方法非常简单，单击按钮，启动 SCOPE，在约束属性（Attributes）选项卡中选择信号名称，并选择选约束属性为 syn_ramstyle，设置属性值即可。

例 13.　对例 5 代码所描述的 RAM 指定不同的 syn_ramstyle 属性值，观察 Synplify Pro 综合结果所使用的 RAM 资源类型

本例，我们采用在 SCOPE 中设置综合约束属性（Attributes）的方法，为 syn_ramstyle 约束属性赋不同的属性值，并用 Technolgy View 比较综合结果。

创建工程后，设置器件为 Xilinx Virtex2 XC2V40 –6 CS144，首先预跑一遍综合流程，然后单击按钮，启动 SCOPE，在约束属性（Attributes）选项卡中选择信号名称为“mem”，选择约束属性为“syn_ramstyle”，属性值为“block ram”，如图 1－25 所示。

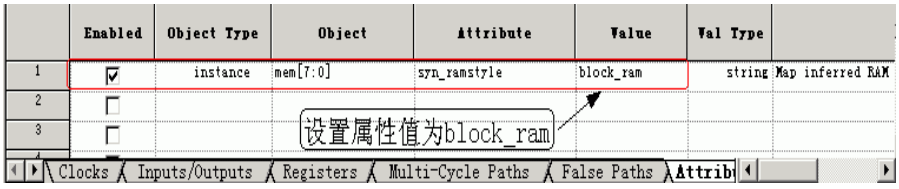


图1-25　在 SCOPE 中设置 syn_ramstyle 属性值为 block ram

保存约束文件，在 Synplify Pro 中重新综合后，使用 Technolgy View 观察综合结果，如图 1－26 所示。可见 Synplify Pro 将该 RAM 使用“RAM16X8S”SelectBlock RAM 硬件原语实现，对应硬件资源是 Block RAM 资源。

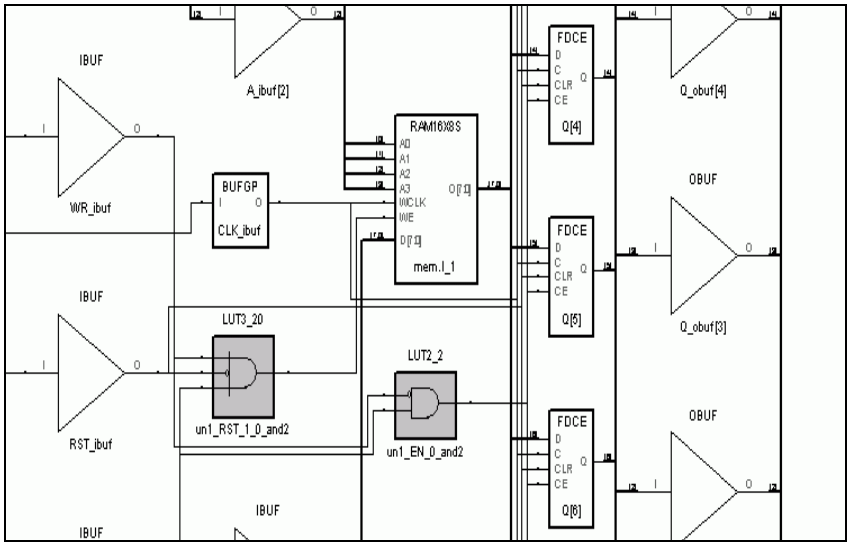


图1-26　syn_ramstyle 属性值为 block ram 时的 Technology View

同样的方法设置约束属性为“syn_ramstyle”，属性值为“register”，保存约束文件，在 Synplify Pro 中重新综合。使用 Technolgy View 观察综合结果，如图 1－27 所示。可见可见 Synplify Pro 用 FF＋LUT 实现该 RAM 单元。

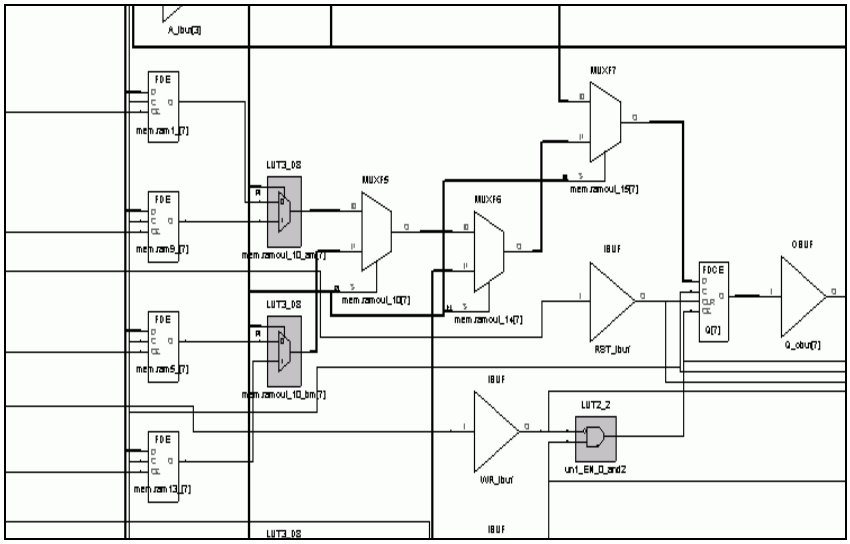


图1-27　syn_ramstyle 属性值为 registers 时的 Technology View

从上面的例子我们可以清晰的看到，在 Synplicity 中使用 syn_ramstyle 综合约束属性可以有效、方便地改变 RAM 映射的资源类型。

在 Synlicity 综合工具中，和 ROM 资源相关的综合约束属性是“syn_romstyle”，该综合约束属性的含义和用法与“syn_ramstyle”有很强的类比性，在此仅作简单的介绍，希望读者通过类比 syn_ramstyle 学习。

syn_ramstyle 的可选属性值有三个：logic、block_rom、select_rom。第一个属性值对应

于用组合逻辑（使用 LUT）实现 ROM；第二个属性值对应使用 Block RAM 资源赋予固定的初值实现 ROM；第三个属性值对应于使用分布式 RAM 资源实现 RAM，如在 Xilinx Virtex/Virtex E/Virtex 2/Virtex 2 Pro 和 Lattice 器件中的分布式 RAM 资源实现 ROM。默认情况下比较小的 ROM 会使用组合逻辑实现，大块的 ROM 尽量采用 Block RAM 资源实现。对于小 ROM，如果器件有分别式 RAM 资源（如 Xilinx、Lattice 某些器件族的 FPGA），Synplify Pro 还会自动类推采用分布式 RAM 资源实现。Altera 器件中实现大块 ROM，APEX 等器件族常用 ESB（Extended System Block）实现，FLEX 等器件族常用 EAB（Extended Array Block）实现。

syn_romstyle 约束属性的附加方法也有三种：在 HDL 代码中直接附加综合约束属性，在 SCOPE 图形界面下设置约束属性，SDC 综合约束文件中添加约束属性。其实这三种方法也是 Synplicity 综合工具附加综合约束属性的一般性方法，请读者熟练掌握这三种方法。

下面通过举例，讲解 syn_romstyle 的使用方法。

例 14.　在 Verilog 代码中附加 syn_romstyle 综合约束属性，指定 ROM 实现类型

在 Verilog 代码中指定 syn_romstyle 的语法格式为：

```
object /* synthesis syn_ramstyle = "string" */ ;
```

其中“*object*”为 reg 型信号或者 output 型输出；用黑体表示的“synthesis syn_romstyle”是综合 ROM 类型的约束属性关键字；“string”根据所选器件类型，可以选择 logic、block_rom、select_rom 中的一种属性值。使用时一般直接附加在信号声明后。如指定信号“rom_table”综合为“logic”类型的 ROM，即使用 LUT 实现。代码如下：

```
reg [7:0] rom_table /* synthesis syn_romstyle="logic" */;
```

例 15.　在 VHDL 代码中附加 syn_romstyle 综合约束属性，指定 ROM 实现类型

在 VHDL 代码中指定 syn_romstyle 的语法格式为：

```
attribute syn_romstyle of object : object_type is "string";
```

其中 *object* 可以是 ROM 类型的输出；黑体表示的“**attribute syn_romstyle of**”是综合工具的 VHDL 综合约束属性关键字；"string"同样根据所选器件类型，选择 logic、block_rom、select_rom 中的一种。

在 VHDL 代码中指定信号“rom_table”使用 Block RAM 资源实现 ROM 的示例如下：

```
signal rom_table : std_logic_vector(63 downto 0);
attribute syn_romstyle : string;
attribute syn_romstyle of rom_table : signal is "block_rom";
```

例 16.　在 SDC 综合约束文件中附加 syn_romstyle 综合约束属性，指定 ROM 实现类型

在 SDC 文件中指定 syn_romstyle 的语法格式为：

```
define_attribute {rom_primitive_name [bit_range]} syn_ramstyle {string}
```

其中，黑体字表示的“define_attribute”是 SDC 文件的约束属性关键字；“rom_primitive_*name* [*bit_range*]”是代码中需要综合为 ROM 资源信号名称；黑体字表示

的“syn_romstyle”是综合 ROM 类型约束属性关键字；"string"同样根据所选器件类型，选择 logic、block_rom、select_rom 中的一种属性值。

例如指定信号“rom_table[3:0]”综合为“logic”类型的 ROM 单元，需要在 sdc 文件中添加如下约束属性：

```
define_attribute {rom_table[3:0]} syn_romstyle {logic}
```

同样，除了手工在 SDC 文件中添加综合约束属性外，也可以直接在 Synplify Pro 等综合工具的 SCOPE 图形界面中直接设置所需的综合约束属性。

1.9.3 RAM/ROM/CAM 等存贮单元的综合、仿真、实现

上节介绍过生成 RAM 等存储单元的推荐方法是使用器件商的 IP core。目前 EDA 软件的用户界面都比较友好，Xilinx、Altera、Lattice 等主流 FPGA 厂商都有相应的 IP Core 生成器，使用非常简便，只需在根据向导的指示，设置所需参数，然后点击按钮，生成即刻。如果使用厂商的 IP Core 实现 RAM，那么 RAM 等存储器件的综合、仿真、实现和一般的 IP Core 完全一致。如果在厂商提供的软件中直接综合实现，则非常简单，一般不需特殊操作，厂商的软件会自动顺利完成所有步骤；如果在第三方软件中完成综合、仿真、实现，则略有不同。

- **综合**

第三方软件处理厂商的 IP Core 的一般方法是综合为黑匣子（Block Box）。例如 Synplify Pro 中声明黑盒子的方法有两种：一是对该模块仅仅进行模块名称和端口声明，其功能实体为空，则 Synplify Pro 自动将之综合成黑盒子；第二种方法是使用综合约束条件，在 SCOPE 中属性（Attribute）设置页面或者 HDL 源代码中添加综合成黑盒子的属性，如表 1－4 所示。

表1－4.　Black Box 综合属性声明

表1－5. 语法 / 表1－6. 语言	1－7. 综合为黑盒子	1－8. 指定黑盒子引脚
Verilog	/* synthesis syn_black_box */;	/* synthesis syn_black_box black_box_pad_pin="引脚名"*/
VHDL	attribute syn_black_box of bbox: component is true;	attribute black_box_pad_pin of 模块名: component is "引脚名";

由于综合工具不了解黑匣子内部的结构，所有不能有效的约束黑匣子的时序行为。对于黑匣子进行时序约束比较麻烦，需要使用 STAMP，即 Synopsys 模块描述语言，该语言为大多数 EDA 软件制造商和器件制造商所支持。使用 STAMP 可以描述黑盒子的时序，如延时、建立保持时间等。在 Synplify Pro 中对黑盒子使用 STAMP 描述时序关系可以使 Synplify Pro 综合结果估算出的频率关系与实际布局布线后的频率关系更加接近。

随着 EDA 厂商之间的密切交流，综合软件和厂方合作日益广泛，目前 Synplify Pro 等主流综合软件已经可以自动对很多器件族的 RAM 等存储单元进行时序约束，更有效的使用

Timing Driven 方式进行综合。

- **仿真**

仿真方法大致有两种：第一种是采用上节介绍的编码方法，自行编写 Behaivior 级的仿真代码，进行仿真；第二种方法是使用器件商提供的仿真模型。

前一种方法适用于简单的，和以往继承的设计，它的优点是使用非常灵活，缺点只能用于功能仿真，不适用於时序后仿真。因为时序后仿真的仿真模型非常难以编写。另外编写仿真模型必须要于 IP Core 的功能完全一致，所有这种方法很少使用。

一般的仿真方法是采用器件商提供的仿真模型。一般在生成 IP Core 的过程中，IP Core 生成软件会自动根据用户定义的参数同时生成 Verilog 和 VHDL 的仿真模型，仿真时只需直接将仿真模型调入即可。但是由于仿真模型是一个参数化的模型，其中调用了大量器件商的硬件原语和底层宏单元，所以一般使用 IP Core 的仿真模型仿真时，都必须在仿真工具中加载相应的仿真库。仿真库的编译和加载方法详见仿真工具说明。时序后仿真没有什么特殊事项，因为在实现后，RAM 等 IP Core 已经被厂方的实现工具扩充为功能实体，从最后布局布线数据库提炼出的实现后仿真模型与一般设计没有任何区别，所以其时序后仿真过程与一般后仿真无差别。

- **实现**

实现过程最为简单。在代码中根据 IP Core 生成器中设置的参数实例化 RAM 等 IP Core，很多 IP Core 生成器在生成 IP Core 的同时自动生成一个专为实例化而设计的模板文件，只需剪切该部分代码即可，非常方便。然后将综合生成的网表导入厂方的布局布线工具，直接实现即可。厂方的布局布线器会自动调用相关的硬件原语和宏单元，扩展 RAM 的功能实体，完成整个实现过程。

1.10 常用模块之二：全局时钟资源与时钟锁相环

全局时钟布线资源一般使用特殊工艺实现（比如全铜层），并设计了专用时钟缓冲与驱动结构，从而使全局时钟到达芯片内部的所有可配置单元、I/O 单元和选择性块 RAM 的时延和抖动都为最小。长线资源，有时也称为第二全局时钟资源。它是分布在芯片的行、列的栅栏（Bank）上，一般采用铜、铝工艺，其长度和驱动能力仅次于全局时钟资源。与全局时钟相似，第二全局时钟资源直接同可配置单元、I/O 单元和选择性块 RAM 等逻辑单元连接，第二全局时钟信号的驱动能力和时钟抖动延迟等指标仅次于全局时钟信号。长线资源一般比全局时钟资源的数量更丰富一些。

目前大多数 FPGA 厂商都在 FPGA 内部集成了硬的 DLL（Delay-Locked Loop）或者 PLL（Phase-Locked Loop），用以完成时钟的高精度、低抖动的倍频、分频、占空比调整移相等。目前高端 FPGA 产品集成的 DLL 和 PLL 资源越来越丰富，功能越来越复杂，精度越来越高（一般在 ps 的数量级）。Xilinx 芯片主要集成的是 DLL，而 Altera 芯片集成的是 PLL。Xilinx 芯片 DLL 的模块名称为 CLKDLL，在高端 FPGA 中，CLKDLL 的增强型模块为 DCM（Digital Clock Manager，数字时钟管理模块）。Altera 芯片的 PLL 模块也分为增强型 PLL（Enhanced PLL）和高速（Fast PLL）等。这些时钟模块的生成和配置方法一般分为

两种，一种是在 HDL 代码和原理图中直接实例化，另一种方法是在 IP 核生成器中配置相关参数，自动生成 IP。Xilinx 的 IP 核生成器叫 Core Generator，另外在 Xilinx ISE 5.x 版本中通过 Archetecture Wizard 生成 DCM 模块。Altera 的 IP 核生成器叫做 MegaWizard。另外可以通过在综合、实现步骤的约束文件中编写约束属性完成时钟模块的约束。

下面分别介绍 Xilinx 和 Altera 关于全局时钟资源和 DLL、PLL 相关的硬件原语和功能模块。

- Xilinx 器件与全局时钟资源和 DLL 相关的硬件原语和功能模块

  常用的与全局时钟资源相关的 Xilinx 器件原语包括：IBUFG、IBUFGDS、BUFG、BUFGP、BUFGCE、BUFGMUX、BUFGDLL、DCM 等，如图 19 所示。

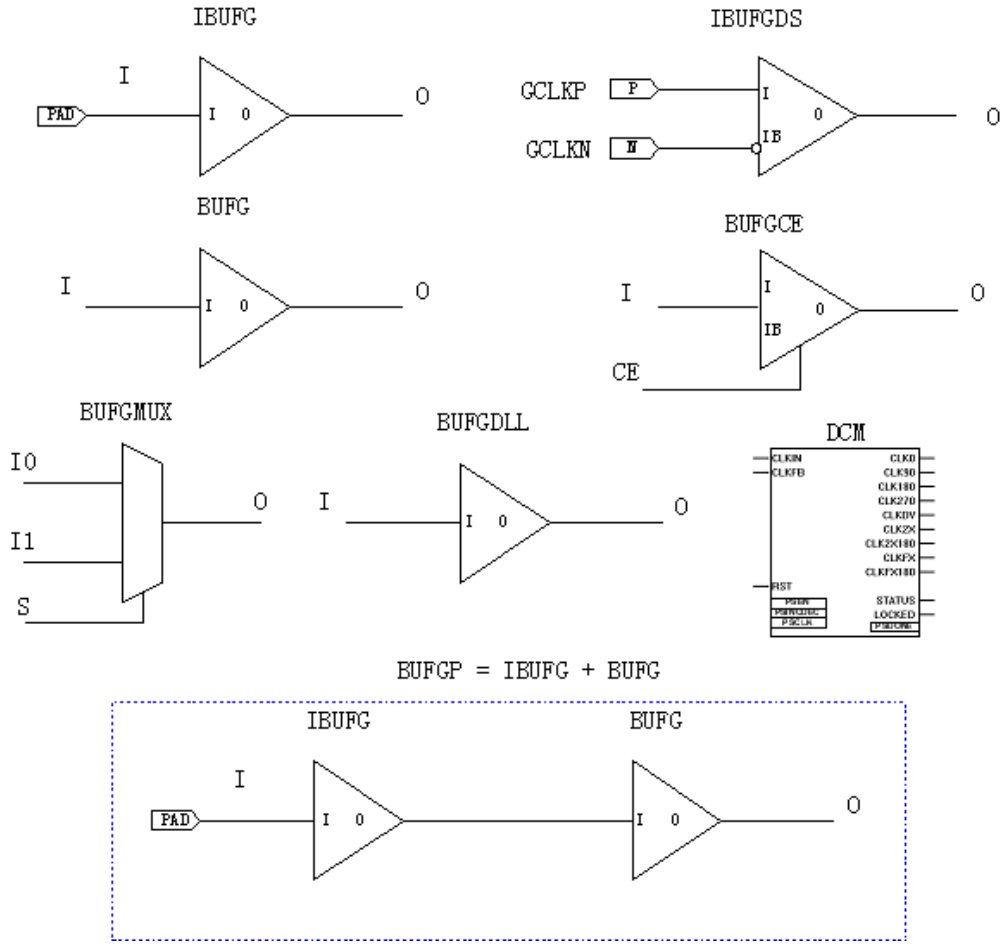


图1-28　图 1 IBUFG、IBUFGDS、BUFG、BUFGP、BUFGCE、BUFGMUX、BUFGDLL 示意图

(1) IBUFG 即输入全局缓冲，是与专用全局时钟输入管脚相连接的首级全局缓冲。所有从全局时钟管脚输入的信号必须经过 IBUFG 单元，否则在布局布线时会报错。IBUFG 支持 AGP、CTT、GTL、GTLP、HSTL、LVCMOS、LVDCI、LVDS、LVPECL、LVTTL、PCI、PCIX、SSTL 等多种格式的 IO 标

准。

(2) IBUFGDS 是 IBUFG 的差分形式，当信号从一对差分全局时钟管脚输入时，必须使用 IBUFGDS 作为全局时钟输入缓冲。IBUFGDS 支持 BLVDS、LDT、LVDSEXT、LVDS、LVPECL、ULVDS 等多种格式的 IO 标准。

(3) BUFG 即全局缓冲，它的输入是 IBUFG 的输出，BUFG 的输出到达 FPGA 内部的 IOB、CLB、Block Select RAM 的时钟延迟和抖动最小。

(4) BUFGCE 是带有时钟使能端的全局缓冲。它有一个输入 I、一个时能端 CE、一个输出端 O。仅当 BUFGCE 的时能端 CE 有效（高电平）时，BUFGCE 才有输出。

(5) BUFGMUX 是全局时钟选择缓冲，它有两个输入 I0 和 I1，一个控制端 S，一个输出端 O。当 S 为低电平时输出时钟为 I0，反之为 I1。需要指出的是 BUFGMUX 的应用十分灵活，I0 和 I1 两个输入时钟甚至可以为异步关系。

(6) BUFGP 相当于 IBUG 加上 BUFG。

(7) BUFGDLL 是全局缓冲延迟锁相环，相当于 BUFG 与 DLL 的结合。在早期设计中经常使用，用以完成全局时钟的同步、驱动等功能。随着数字时钟管理单元（DCM）的日益完善，目前 BUFGDLL 的应用已经逐渐被 DCM 所取代。

(8) DCM 即数字时钟管理单元，主要完成时钟的同步、移相、分频、倍频、去抖动（skew）等。DCM 与全局时钟有着密不可分的联系，为了达到最小的延迟和抖动，几乎所有的 DCM 应用都要使用全局缓冲资源。DCM 可以用 Xilinx ISE 软件中的 Architecture Wizard 直接生成。

- **Xilinx 全局时钟资源的使用方法**

  Xilinx 全局时钟资源的使用方法有以下 5 种，如图 20 所示。

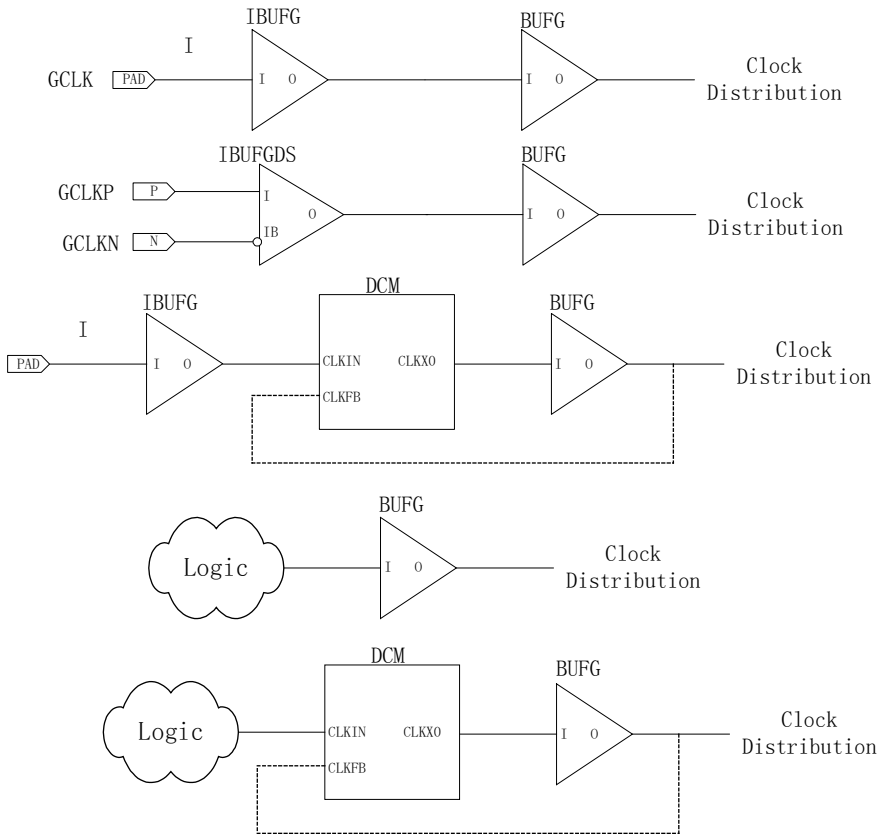


图1-29　Xilinx 全局时钟资源的 5 种常用方法

(1) IBUFG + BUFG 的使用方法:

IBUFG 后面连接 BUFG 的方法是最基本的全局时钟资源的使用方法，由于 IBUFG 组合 BUFG 相当于 BUFGP，所以在这种使用方法也称为 BUFGP 方法。

(2) IBUFGDS + BUFG 的使用方法:

当输入时钟信号为差分信号时，需要使用 IBUFGDS 代替 IBUFG。

(3) IBUFG + DCM+BUFG 的使用方法:

这种使用方法最为灵活，对全局时钟的控制更加有效。通过 DCM 模块不仅仅能对时钟进行同步、移相、分频、倍频等变换，而且可以使全局时钟的输出达到无抖动延迟（“0”skew）。

(4) Logic + BUFG 的使用方法:

BUFG 不但可以驱动 IBUFG 的输出，还可以驱动其它普通信号的输出。当某个信号（时钟、使能、快速路径）的扇出非常大，并且要求抖动延迟最小时，可以使用 BUFG 驱动该信号，使该信号利用全局时钟资源。但是需要注意的是，普通 IO 的输入或普通片内信号进入全局时钟布线层需要一个固有的延时，一般在 10ns 左右，也就是说普通 IO 和普通片内信号从输入到 BUFG 输

出有一个约 10ns 左右的固有延时，但是 BUFG 的输出到片内所有单元（IOB、CLB、Block Select RAM）的延时可以忽略不计为“0”ns。

(5) Logic + DCM+BUFG 的使用方法：

DCM 同样也可以控制并变换普通时钟信号。也就是说 DCM 的输入也可以是普通片内信号。

- **使用 Xilinx 全局时钟资源的注意事项**

  Xilinx 全局时钟资源必须满足的重要原则是：“使用 IBUFG 或 IBUFGDS 的***充分必要***条件是信号从专用全局时钟管脚输入。”换句话说，当某个信号从全局时钟管脚输入，不论它是否为时钟信号，都必须使用 IBUFG 或 IBUFGDS；如果对某个信号使用了 IBUFG 或 IBUFGDS 硬件原语，则这个信号必定是从全局时钟管脚输入的。如果违反了这条原则，那么在布局布线时会报错。这条规则使用由 Xilinx 的 FPGA 的内部结构决定的：IBUFG 和 IBUFGDS 的输入端仅仅与芯片的专用全局时钟输入管脚有物理连接，与普通 IO 和其它内部 CLB 等没有物理连接。

  另外由于 BUFGP 相当于 IBUFG 和 BUFG 的组合，所以 BUFGP 的使用也必须遵循上述的原则。

  全局时钟资源的例化方法大概可分为两种：一是在程序中直接例化全局时钟资源。二是通过综合阶段约束或者实现阶段约束实现对全局时钟资源的使用。

  第一种方法比较简单，用户只需按照前面讲述的 5 种全局时钟资源的基本使用方法编写代码或者绘制原理图即可。第二方法是通过综合阶段约束或实现阶段的约束完成对全局时钟资源的调用。这种方法根据综合工具和布局布线工具的不同而异。另外大多数综合工具会自动分析时钟信号的扇出数目，在全局时钟资源富于的情况下，将扇出数目最大的信号自动指定使用全局时钟资源。这时用户必须检查综合结果是否满足上述使用 IBUFG、IBUFGDS、BUFGP 的原则。如果某个信号因扇出很大，而被综合器自动指定使用 IBUFG、IBUFGDS、BUFGP 全局时钟资源，而该信号并没有从专用全局时钟管脚输入，在布局布线时会报错。这时应该在综合是指定该信号不使用全局时钟资源。具体的综合约束命令和操作因综合工具不同而异，下面仅仅举例 Synplify/Synplify Pro 和 FPGA Express/FPGA Compiler II 中指定不使用全局时钟资源的方法。

- Synplify/Synplify Pro 中，打开 SCOPE（Synthesis Constraints Optimization Environment，综合约束优化环境），选择约束属性（Attributes）选项卡。选择约束对象类型（Object Type）为“clock”，然后在约束对象（Object）为设计中的某个时钟，选择约束属性（Attribute）为“syn_noclockbuf”，设置约束值域（Value）为“1”。图 21 所示为在 Synplify Pro 的 SCOPE 中指定信号不使用全局时钟资源的方法。

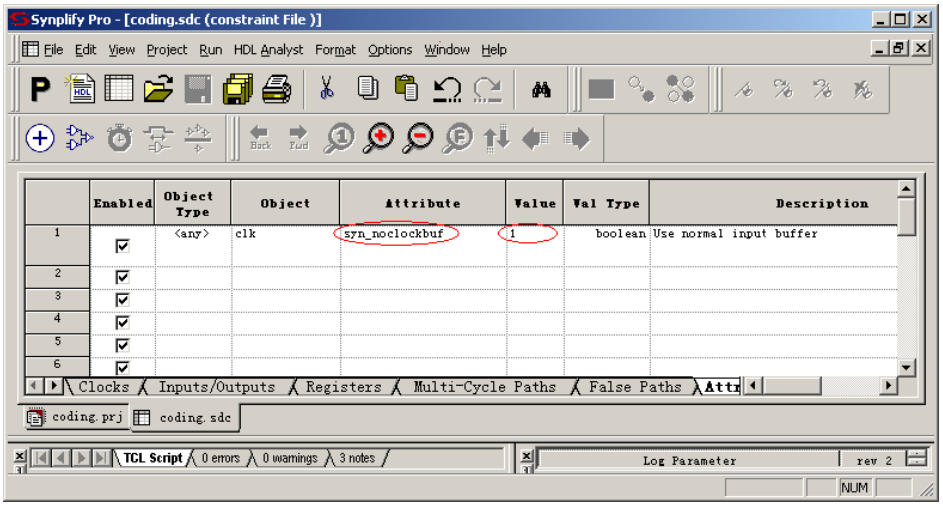


图1-30　在 SCOPE 中指定信号不使用全局时钟资源的方法

- 也可以在 Synplify/Synplify Pro 约束文件（扩展名为 SDC）或者源代码中使用约束命令指定不使用全局时钟资源，如表 1－3 所示。

表1－9.　Synplify/Synplify Pro 中不使用全局时钟资源的约束命令

文件类型	约束命令
SDC 的约束文件	define_attribute　　{clk} syn_noclockbuf {1}
Verilog 源代码	input clk　/* synthesis syn_noclockbuf = 1 */;
VHDL 源代码	attribute syn_noclockbuf : boolean; attribute syn_noclockbuf of clk : signal is true;

- 在 FPGA Express/FPGA Compiler II 中，用鼠标右键单击编译后的芯片图标，在弹出的命令对话框中选择“Edit Constraints”命令编辑综合约束文件（扩展名为 CTL），选择端口（Ports）选项卡，指定所需信号的全局时钟域为“DONT USE”。图 22 所示为在 FPGA Express 综合约束编辑器中指定信号不使用全局时钟资源的方法。

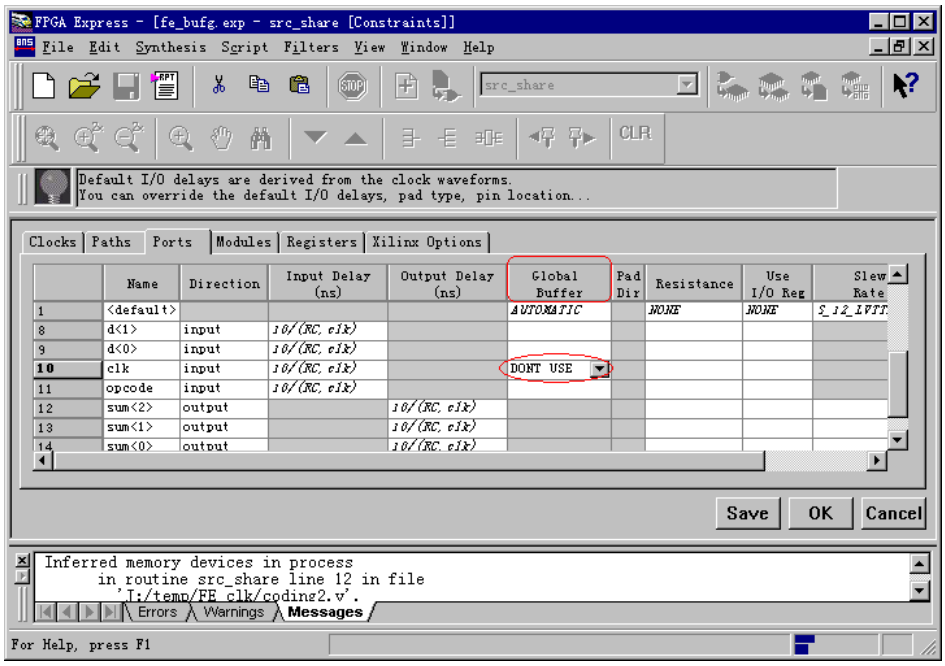


图1-31　在 FPGA Express 综合约束编辑器中指定信号不使用全局时钟资源的方法

- 也可以在 FPGA Express/FPGA Compiler II 综合约束文件（扩展名为 CTL）中直接使用 “port clk global_buffer "DONT USE";” 命令指定输入端口信号 “clk” 不使用全局时钟资源。

- Xilinx Virtex-II 和 Virtex-II Pro 器件族生成 DCM 的方法

  对于 Xilinx Virtex-II 和 Virtex-II Pro 器件族，除了按照上面介绍的 DLL 或 DCM 的使用方法，用代码或者原理图调用 DLL 或 DCM 外，还可以使用 ISE 5.1 以及其后版本附加的辅助设计工具——Architecture Wizard（设计结构向导）完成 DCM 的生成。Architecture Wizard 的使用非常简单，不论从界面风格还是使用方法上都与 Core Generator 十分相似。图 23～27 到简单的示例了 DCM 模块的生成方法：

1. 如图 23 所示，选择器件族为 Virtex-II 或 Virtex-II Pro。

图1-32　设置器件族为 Virtex2P

2. 在 Project Navigator 中新建 Architecture Wizard 资源，如图 24 所示。
3. 单击 下一步(N) > 按钮，显示新建资源信息，单击 完成 按钮确认新建资源信息，启动 Architecture Wizard，选择需要新建的模块类型，如图 25 所示。

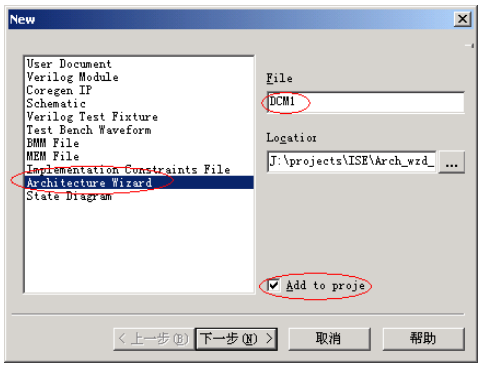


图1-33　新建 Architecture Wizard 资源

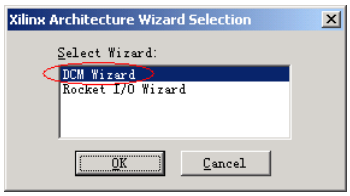


图 2-25　选择 DCM 类型

4. 单击 OK 按钮启动 DCM 向导。DCM 向导界面与 Core Generator 生成 IP 核的界面非常相似，如图 26 所示，设置 DCM 输入时钟频率、时钟源、反馈、移相、占空比整形和输出管脚等参数。

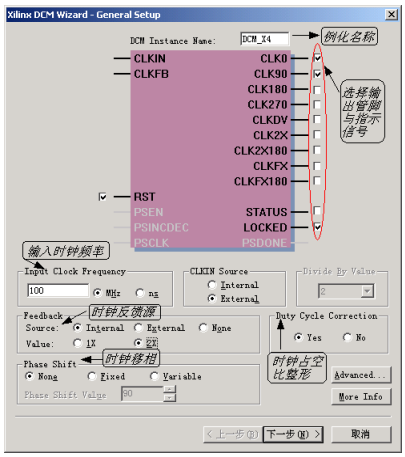


图1-26　DCM 参数设置

5. 单击 Advanced... 按钮，设置 DCM 锁定信号顺序和输入是否 2 分频等高级参数，如图 27 所示。

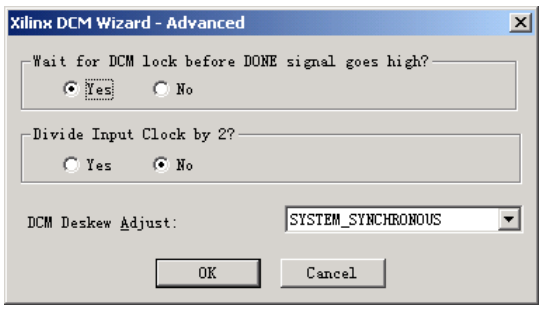


图1-27　设置高级参数

6. 单击 OK 按钮确认高级参数设置，然后单击 下一步(N) > 按钮确认其他参数设置，进入全局时钟设置界面。为了达到更好的性能与精度，Xilinx 强烈建议 DCM 模块时钟信号使用全局时钟缓冲（BUFGP，Global Buffer），单击 完成 按钮使用全局时钟资源，完成 DCM 模块的生成。

- Altera 器件 PLL 简介

  Altera 器件的 PLL 使用比较方便，一般是都是用 EDA 辅助工具比如 Magafunction 或 Maga Wizard 生成 IP，然后调用。Altera FPGA 不同器件可用的 PLL 单元不同，使用是需要根据器件类型选择相应的资源，如图 28，29，30 所示分别是 Enhanced PLL、Fast PLL、Cyclone PLL 等不同类型的 PLL 结构示意图。

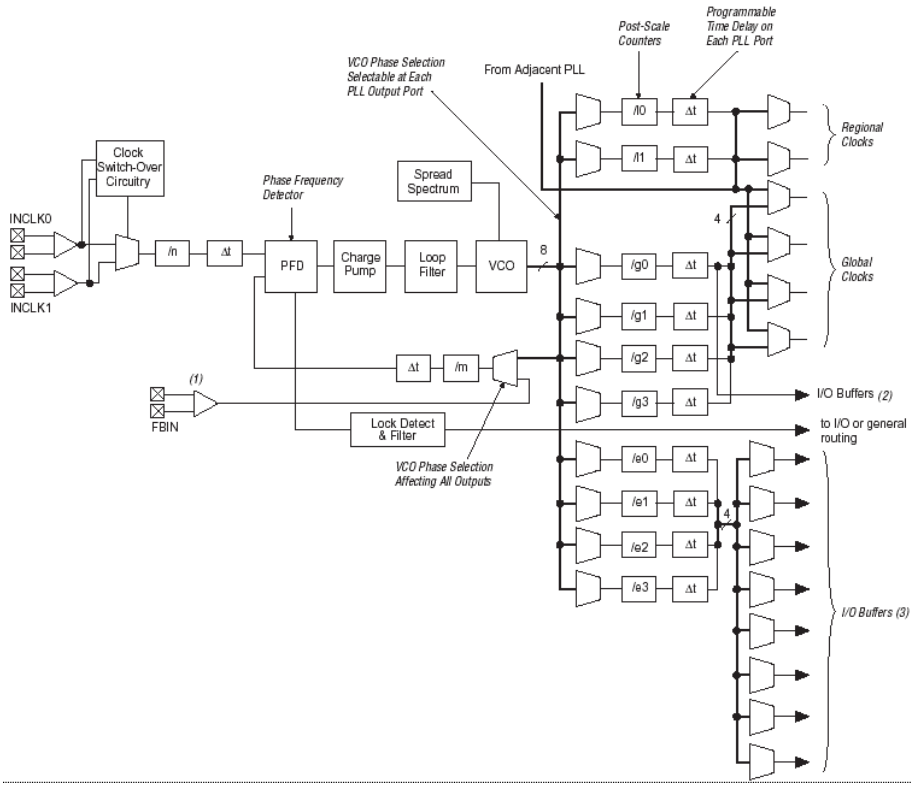


图1-28　Stratix/Stratix Gx 器件族的 Enhanced PLL 结构示意图

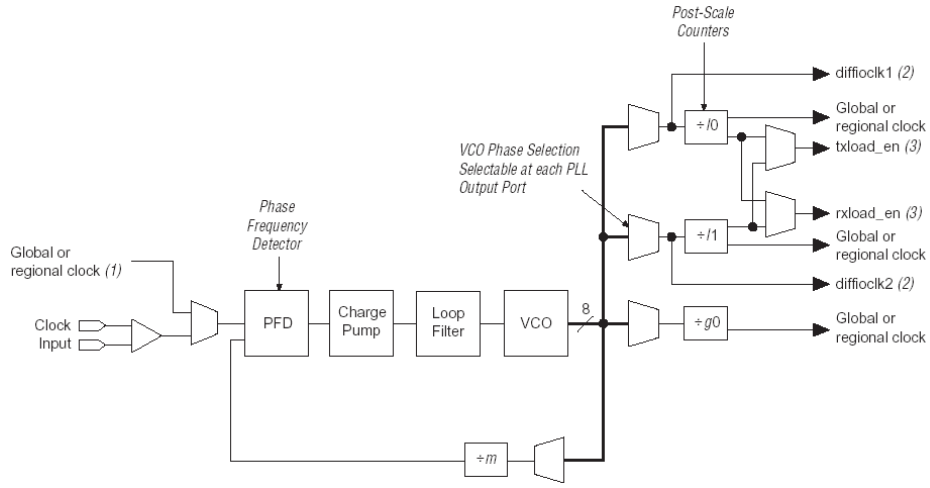


图1-29　Stratix/Stratix Gx 器件族的 Fast PLL 结构示意图

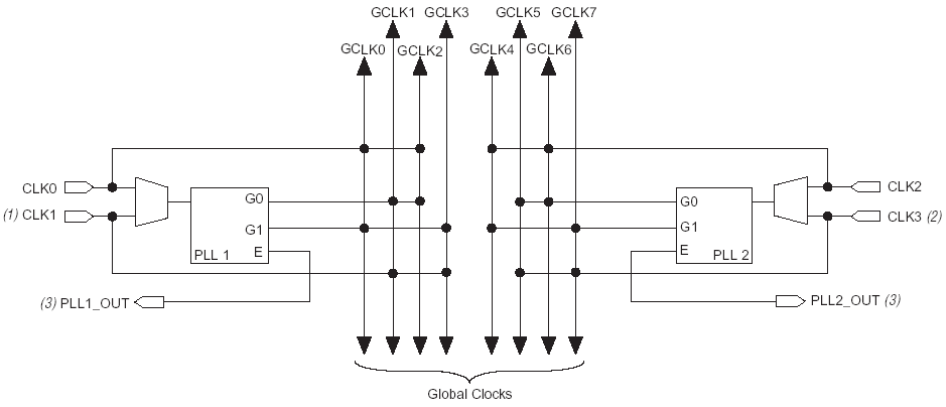


图1-30　Cyclone PLL 结构示意图

在使用 Altera PLL 的时候，首先应该根据自己所选的器件类型，察看该器件可用的 PLL 资源，然后根据需要设计 PLL 的参数，通过 Magafunction 或 Maga Wizard 设置 PLL 的参数并生成 IP core，在代码中调用即可。整个使用过程的关键在于弄清 Alter PLL 的常用参数含义。

- **Altera 器件 PLL 常用端口和参数简介**

下面对 Altera 器件 PLL 的常用端口和参数加以简介，不同器件具体支持的端口和参数略有不同，使用时请参看该器件的器件手册。

(1) pllena 它是 PLL 的使能端，高有效，当为低时，PLL 无输出。

(2) areset 它是 PLL 的异步复位端，高有效，异步复位 PLL。

(3) OPERATION_MODE 选择 PLL 的工作模式。Stratix 器件族的 PLL 通过选择 PLL 的工作模式为“normal”或者“zero delay buffer”，可以补偿片内和片外的时钟延迟。常用可配置的模式如下：“Normal”模式下，PLL 的输入引脚与

I/O 单元的输入时钟寄存器相关连；“Zero delay buffer”模式下，PLL 的输入引脚和 PLL 的输出引脚的延时相关连，通过 PLL 的调整，到达两者“零”延时；“External feedback”模式下，PLL 的输入管脚和 PLL 的反馈管脚延时相关联；“No compensation”模式下，不对 PLL 的输入管脚进行延时补偿。

(4) SCAN_CHAIN，用以动态重配 Stratix PLL 状态，有 Long 和 Short 两种不同宽度的扫描链。

(5) BANDWIDTH，用以指定 PLL 的相关带宽，默认情况下该参数被设置为“auto”。

(6) DOWN_SPREAD，SPREAD_FREQUENCY，这两个关于扩频的参数用以减少 PLL 的 EMI（电磁辐射）。

(7) clkswitch，clkloss，clkbad，这 3 个参数和 enhanced PLL 的输入时钟切换相关，通过 PLL 输入时钟切换使 PLL 的输入在两个不同的输入时钟间切换，使用该参数必须使能“inclock1”端口。clkloss 用以指定 PLL 在一个输入时钟丢失时自动切换到另一个输入时钟；clkbad 用以指定 PLL 在一个输入时钟质量便差时自动切换到另一个输入时钟；clkswitch 用以强迫指定 PLL 在 2 个输入时钟之间切换。

(8) CLK[ ]_MULTIPLY_BY，括号内为 CLKPLL 的标号，该参数是最为常用的参数之一，用以设置 CLKPLL 的倍频因子。

(9) CLK[ ]_DIVIDE_BY，括号内为 CLKPLL 的标号，该参数是最为常用的参数之一，用以设置 CLKPLL 的分频因子。

(10) CLK[ ]_PHASE_SHIFT，括号内为 CLKPLL 的标号，该参数用以设置输出时钟的相位偏移。

(11) CLK[ ]_TIME_DELAY，括号内为 CLKPLL 的标号，该参数用以设置输出时钟的时间延迟，延迟时间范围从－3ns 到 6ns。

(12) CLK[ ]_DUTY_CYCLE，括号内为 CLKPLL 的标号，该参数用以设置输出时钟的占空比中高电平的百分数，其可设置范围和输入时钟的频率相关。

(13) clkena[ ]，括号内为 CLKPLL 的标号，用以指定某个 CLKPLL 是否有效输出。高电平有效。当某个时钟被指定 clkena 为低时，其相关的 VCO 仍然振荡，但是输出时钟端口却被屏蔽了。

- 使用 MegaWizard 生成 altpll

Altera 器件的各种 altpll 一般都可以通过 MegaWizard 生成，根据用户设置，MegaWizard 会根据所选 altpll 类型（Enhanced PLL、Fast PLL、Cyclone PLL 等）引导用户完成该类型 PLL 的所有参数设置，非常实用、方便。下面简单演示一下 MegaWizard 的使用方法。

打开 Quartus，运行【Tools】/【MegaWizard plug-In Manager…】选项，在第一个对话框选择生成新的 Megafuntion，单击 NEXT 按钮进入 IP 功能选择框，如图 31 所示，在 I/O 功能分类目录下选择 ALTPLL，并在输出文件中输入模块的输出文件名。

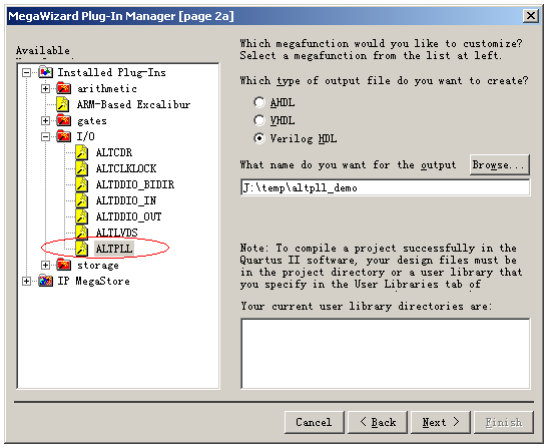


图1-31　在 MageWizard 中选择 altpll

单击 NEXT 按钮，进入 ALTPLL 的系列参数设置。根据如果选择 Enhanced PLL 一般会有 15 个对话框出现；选择 Fast PLL 会有 8 个对话框出现，根据提示依次在对话框中设置所需的 PLL 参数，就完成了 ALTPLL IP core 的生成。

如图 32 所示，第一个对话框用以设置 altpll 的基本参数和工作模式，如是否使用 Fast PLL，器件族，输入时钟的频率，设置使能、复位等可选管脚，设置 PLL 的时延补偿模式等。在对话框的最左边为 altpll 的参数和管脚示意图，该示意图会根据用户的参数设置动态调整。

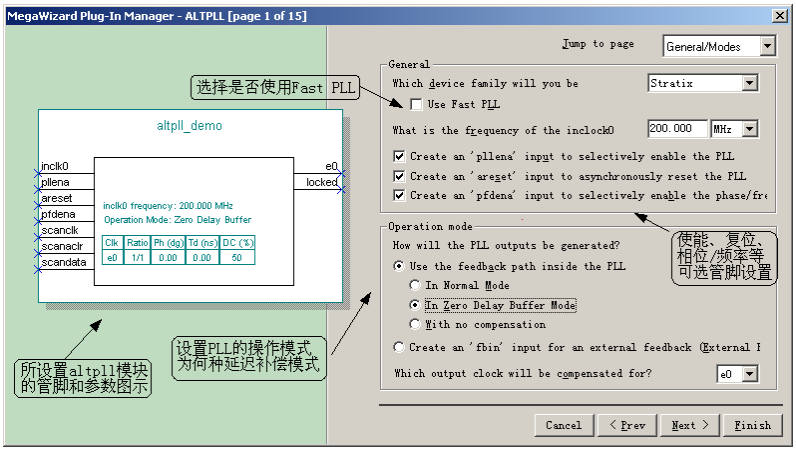


图1-32　altpll 基本参数和工作模式设置

如图 33 所示，为 altpll 设置的第二个对话框，用以设置 altpll 的动态重配相关参数和锁定输出时钟等。

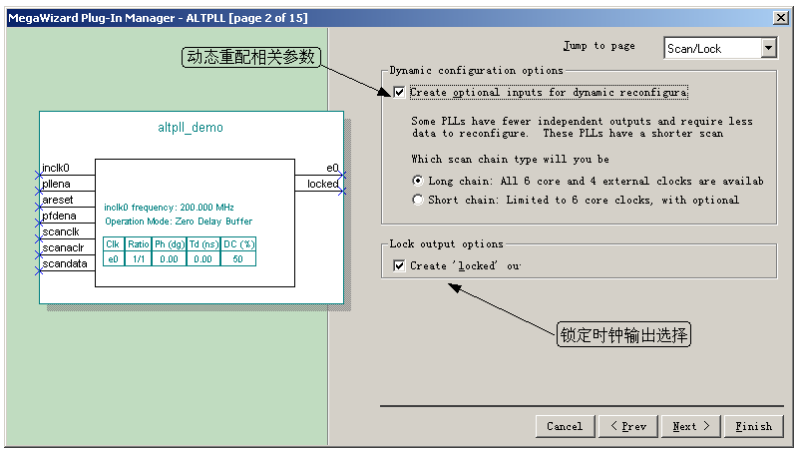


图1-33　altpll 动态重配和锁定输出时钟设置

图 34 为 altpll 的带宽和扩频参数设置。

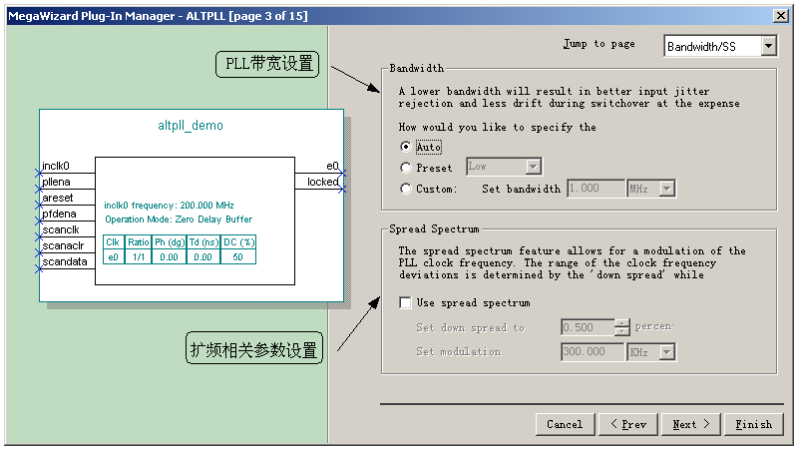


图1-34　altpll 的带宽和扩频参数设置

图 35 为 altpll 的输入时钟切换设置对话框。

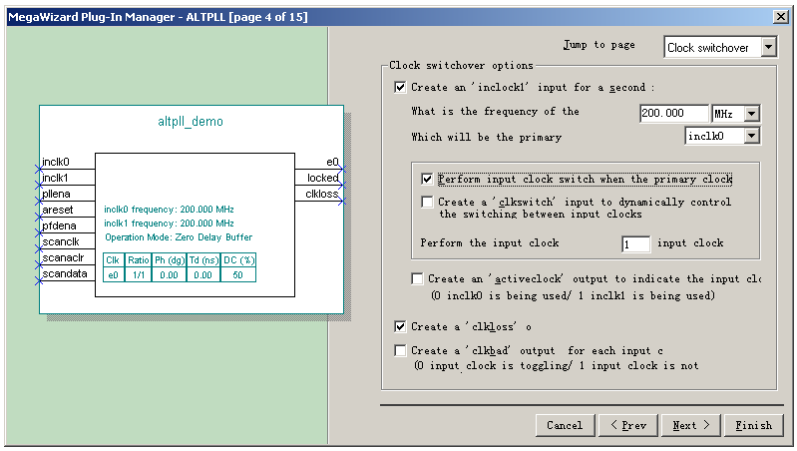


图1-35　altpll 的输入时钟切换设置

图 36 所示为设置 clock C0 的倍频、分频、移相、延迟、占空比调整等因子。该对话框非常重要，通过这个对话框的合理设置，用户可以生成满足小数分频/倍频，相位调整，占空比调整，时延调整等不同要求的 PLL 输出时钟。

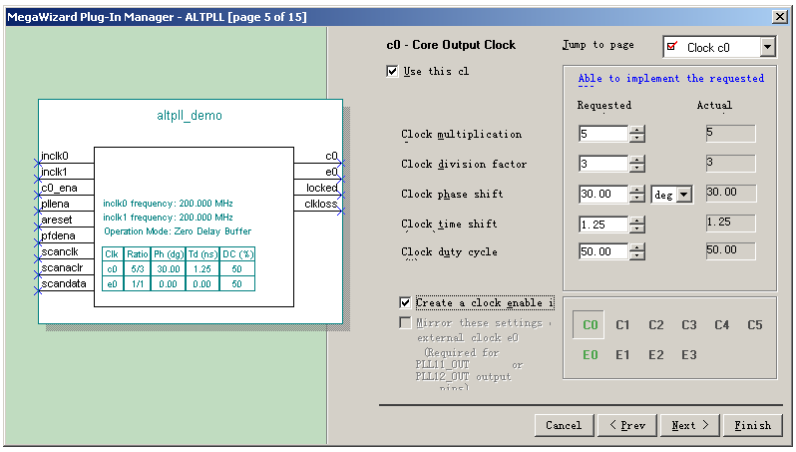


图1-36　倍频、分频、移相、延迟、占空比调整等参数设置

同样方法设可以设置其它时钟 C1~C5 和 E0~E3 的倍频、扩频等参数。

最后一个对话框对用户设置的管脚和参数在 altpll 示意图中加以显示，检查无误后，单击“Finish”按钮，Quartus 就会自动生成 altpll 的 IP core。Altpll 的仿真、例化、综合方法与一般 IP core 完全一致，在此不再累述。

- **不同时钟域数据交换时，使用 Altera PLL 时钟的注意事项**

在不同时钟域间交换数据时，除了需要注意 1.8“数据接口的同步方法”所述注意事项外，在使用 altpll 时，还要注意如下问题。

(1)　同步同型时钟域之间的数据交换

如果两个 PLL 时钟是由同一个 PLL 产生的同类型的时钟（如都是全局时钟，或者都是区域性时钟），则这两个时钟域之间的 register－to－register 数据交换最为简单，不需要附加任何逻辑。因为从前面“数据接口的同步方法”中我们已经知道，在这种情况下，使用一级寄存器采样就能完成数据的同步化过程，不会产生错误电平和非稳态的传播。

(2)　同步异型时钟域之间的数据交换

如果两个时钟是由不同的 PLL 产生，或者是不同类型的时钟（一个是全局时钟，一个是区域性时钟），由同一个时钟反馈而无时延或者相位调整，则在进行数据交换时，必须插入一个 LE（Logic element，逻辑单元，Altera 的基本可配置模块）。如图 37 所示。

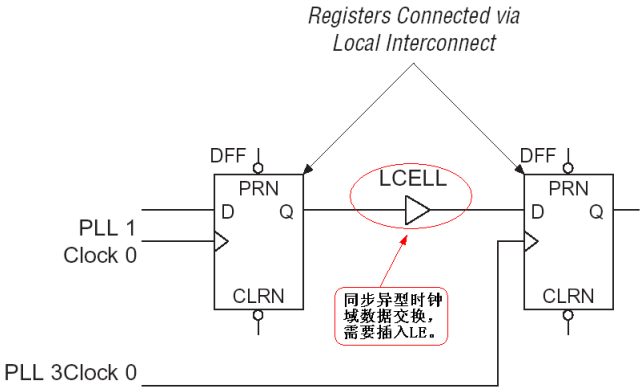


图1-37　同步异型时钟域之间的数据交换，需要插入 LE

(3)　异步时钟域之间的数据交换

如果是完全异步的时钟域间数据交换，请按照 1.8“数据接口的同步方法”所述方法，使用 DPRAM 或者 FIFO 完成，可以简单的绕过异步时钟域数据同步的握手和错误电平或者亚稳态等问题。Altera 举例使用 DCFIFO 完成异步时钟域之间的数据交换，如图 38 所示。

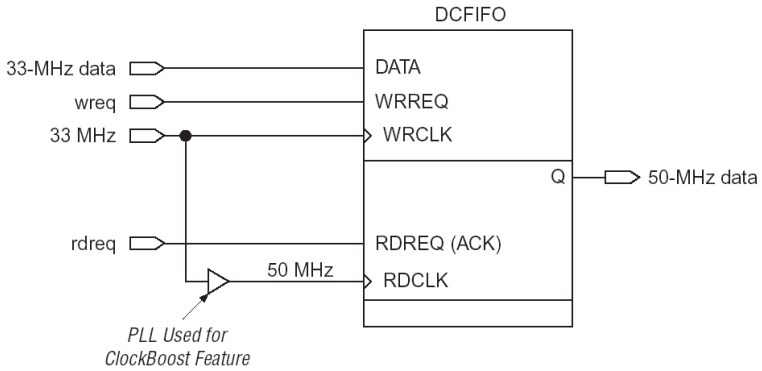


图1-38　使用 DCFIFO 完成异步时钟域之间的数据交换

- 门控时钟（gate clock）的危害

1.11　常用模块之三：全局复位/置位信号

很多设计者在使用全局复位（GSR）、全局置位（GTS）等 startup 资源时遇到了较多的困难，现在将关于 GSR/GTS 等 startup 资源的使用加以简介。

- 使用 GSR/GTS 前，设计者必须要搞清楚，您的设计中是否真正需要使用 GSR/GTS?

这一点非常重要，GSR/GTS 不像全局时钟资源是推荐使用的全局资源。特别在 Virtex/E/II/II Pro，Spartan-II/III 等器件族中，在用户设计资源富裕的情况下，甚至不推荐使用。主要因为 GSR/GTS 的 skew 和延时较大，速度很慢，会影响设计的速度。如果 GSR/GTS 的保持时间设计不够，还会导致电路并未真正的全局复位、置位，导致逻辑错误。而上述器件如果资源有富余，可以直接用长线资源（还有一种叫法为：第二全局时钟资源），它们的速度、skew 比 GSR/GTS 优异很多，能更好地满足设计需求。

- **GSR/GTS 的三种基本调用方法**

GSR/GTS 的三种基本调用方法是：直接例化、在综合的时候加约束属性、默认类推等。

(1) 直接例化 GSR/GTS 的方法直截了当，在代码中显化例化 GSR/GTS 资源，明确告诉综合器，综合器只有乖乖听话了。例化方法如下：

例 17.　VHDL 显化调用 GSR/GTS 的例子：

```
-- This example uses both GTS and GSR pins.
-- Unconnected STARTUP pins are omitted from
-- component declaration.
library IEEE;
use IEEE.std_logic_1164.all;
entity setreset is
     port (CLK: in std_logic;
       DIN1 : in STD_LOGIC;
       DIN2: in STD_LOGIC;
       RESET: in STD_LOGIC;
       GTSInput: in STD_LOGIC;
       DOUT1: out STD_LOGIC;
       DOUT2: out STD_LOGIC;
       DOUT3: out STD_LOGIC);
end setreset ;
architecture RTL of setreset is
component STARTUP_VIRTEX
     port( GSR, GTS: in std_logic);
end component;
begin
startup_inst: STARTUP_VIRTEX port map(GSR => RESET, GTS => GTSInput);
reset_process: process (CLK, RESET)
     begin
       if (RESET = '1') then
         DOUT1 <= '0';
```

```
        elsif ( CLK'event and CLK ='1') then
        DOUT1 <= DIN1;
        end if;
end process;
gtsprocess:process (GTSInput)
begin
if GTSInput = '0' then
DOUT3 <= '0';
DOUT2 <= DIN2;
else
DOUT2 <= 'Z';
DOUT3 <= 'Z';
end if;
end process;
end RTL;
```

例 18. Verilog 显化调用 GSR/GTS 的例子：

```
// This example uses both GTS and GSR pins
// Unused STARTUP pins are omitted from module
// declaration.
module setreset(CLK,DIN1, DIN2,RESET, GTSInput,
     DOUT1,DOUT2,DOUT3);
     input CLK;
     input DIN1;
     input DIN2;
     input RESET;
     input GTSInput;
     output DOUT1;
     output DOUT2;
     output DOUT3;
     reg DOUT1;
STARTUP_VIRTEX startup_inst(.GSR(RESET), .GTS(GTSInput));
always @(posedge CLK or posedge RESET)
begin
if (RESET)
     DOUT1 = 1'b0;
else
     DOUT1 = DIN1;
end
```

```
assign DOUT3 = (GTSInput == 1'b0)? 1'b0: 1'bZ;
assign DOUT2 = (GTSInput == 1'b0)? DIN2: 1'bZ;
endmodule
```

(2) 在综合器的综合属性中设置使用 GSR 的选项，这种方法因综合器不同而设置不同。一般叫"Force GSR usage"选项，比如 Synplicity 综合工具（Synplify/Synplify Pro、Amplify 等）里面的综合属性:

```
/* synthesis xc_isgsr = 1 */
```

另外也可以在综合属性图形设置界面添加使用 GSR 资源的属性。比如 Synplicity 综合工具的 SCOPE 中设置，以及在 Leonardo 的综合约束属性图形界面设置等。

(3) 目前，很多综合器会自动将起到全局作用的异步复位类推为 GSR，使用 GSR 资源，该过程是完全自动完成的，不需设计者干预，类推结果后会在综合报告中打印相关信息。但是对 Xilinx 的 Virtex/E/II/II Pro，Spartan-II/III 等器件并不自动类推 GSR，如需使用，必须按照第一种方法所示，在代码中显化调用 STARTUP_VIRTEX，STARTUP_VIRTEX2，或者 STARTUP_SPARTAN2 模块。

- GSR/GTS 的使用方法

  GSR/GTS 最常用的方法是用做异步全局复位/置位。当然也可以用做同步全局置位/复位，但是必须要考虑清楚一些相关的问题。另外用 GSR/GTS 复位/置位状态机，企图使状态机在启动后自动进入安全状态或者 IDEL 状态的时候，也必须要注意状态机的编码方法，由于 GSR 复位的时候，程序并没有发挥作用，所以必须要搞清楚自己的设计状态，如果使用 one－hot 编码，一般 one-hot with zero initial state 的编码方法，使 GSR 后自动进入 initial 的状态。

- GSR/GTS 的仿真

  使用了 GSR/GTS 等 startup 资源，在做仿真时必须指定加入 UniSim、SimPrim 等仿真库。而且需要在工程中加入 glbl.v 文件，在 load 时除了 load 自己的仿真 testbench 外，还要 load glbl.v 文件。测试激励中 GSR、GTS 的声明方法如下:

```
reg GSR;
assign glbl.GSR = GSR;
reg GTS;
assign glbl.GTS = GTS;

initial begin
GSR = 1; GTS = 1;
#100 GSR = 0; GTS = 0;
end
```

如果是早期 CPLD，声明方法如下:

```
reg PRLD;
```

```
assign glbl.PRLD = PRLD;

initial begin
PRLD = 1;
#100 PRLD = 0;
end
```

1.12 常用模块之四：高速串行收发器

本节重点讨论高速串行收发器的使用方法和注意事项。为了提供更加丰富多彩的业务，和高质量的服务，很多新型通讯标准对传输速率和信号带宽都提出了更高的要求，如 3G 移动通讯、10G 以太网、OC192 等等标准，其应用带宽都达到了 GHz 级。为了满足新技术的需求，设计者设计者必须找到高速率、高可靠性、低成本的解决方案。面对这些挑战，高速串行互连方法逐步成了首选解决方案。串行背板相对于并行互联背板有许多显著的优点。串行连接地第一个也是最重要的优点是高性能和高鲁棒性(Robust)。图 1-39 是并行数据传输和串行数据传输的示意图，两者的对比如下：

- **并行数据传输：**
- 多根线占用板面积；
- 每根线相互之间有干扰；
- 每根线需要自己的匹配电路；

- **串行数据传输：**
- 更少的走线占用的板面积减少；
- 信号线的干扰降到最小；
- 相对于并行传输只用了一小部分的匹配电路；
- 没有相位差(skew)的问题；

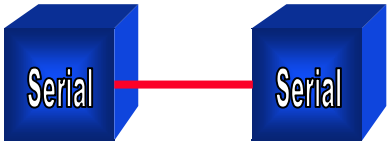


图1-39　平行传输和串行传输

正因为上述特点，使用串行传输方式比并行传输方式更适合目前高速领域的应用。早期的串行传输的速率过低，设计复杂，所以体现不出其优势，随着微电子技术与工艺的发展，FPGA 内嵌的高速串行收发器以其传输速率高、使用方便、性能可靠等优点为串行传输方案注入了新的活力。

串行收发器的英文缩写是 SERDES(SERializer & DESerializer)顾名思义，它由两部分构成：发端是串行发送单元 SERializer，用高速时钟调制编码数据流；接端为串行接收单元 DESerializer，其主要作用是从数据流中恢复出时钟信号，并解调还原数据，根据其功能，

接收单元还有一个名称叫时钟数据恢复器（CDR，Clock and data Recovery）。图 1－40 为 10 根数据线的串行传输和解串行接收示意图，10 根 100Mhz 的信号线进入 SERDES 器件产生串行码流，时钟也调制到码流内。反过来通过它恢复并行的数据和时钟。

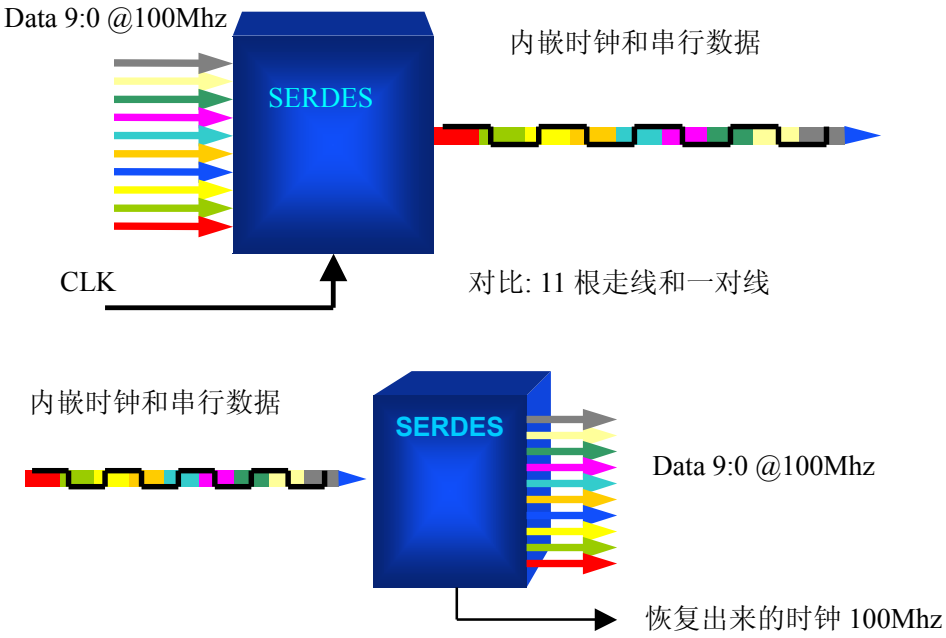


图1-40　SERDES 的功能

SERDES 的最重要的两个参数指标是传输速率和传输长度，即在符合误码率要求的以何种传输速率可以传输多长距离。更形像的评价方法是利用眼图，眼图的高和宽反映了信号的传输质量。图 1-41 所示为眼图实例。

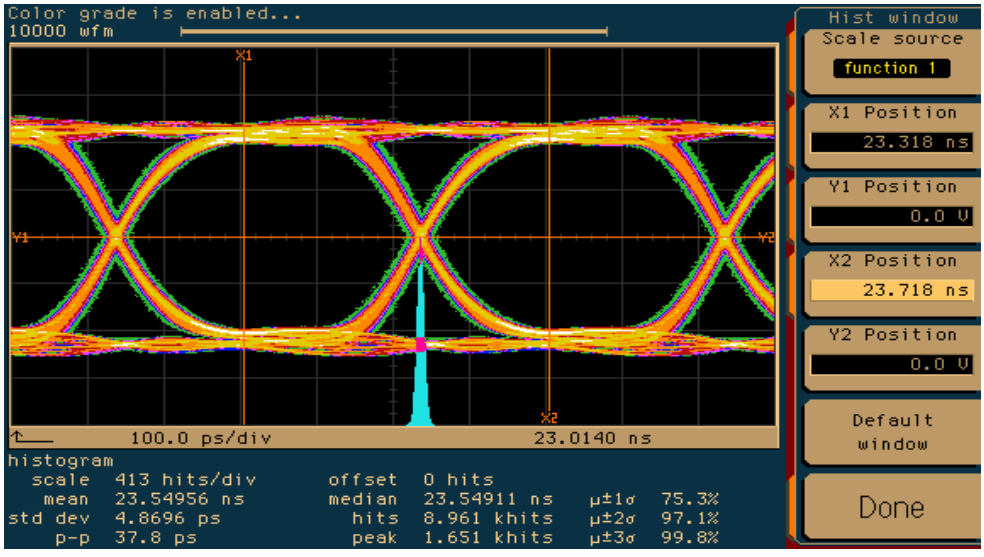


图1-41　眼图实例

图形化接收眼图模板如图 1-42，

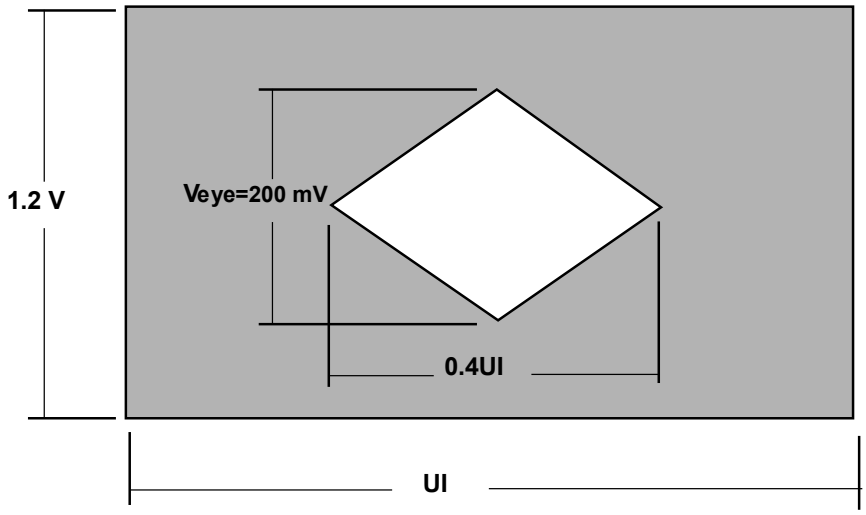


图1-42　接收眼图模板

其中，UI(Unit Interval)是 1bit 时间。例如对于 1Gbps 的眼图 1UI 是 1ns. 落在眼图外的数据被无误码接收（BER < 1E-12）。眼高用 mV, 眼宽用 Ui 或者 ps 来标示。

与 SERDES 有关的参数还有接收抖动容限，发送抖动和功耗。它们对于评估 SERDES 的性能也是非常重要的。

通过增加发送预加重补偿长距离传输高频分量的衰减，一般可以改善眼图的质量，但并这不是绝对的。图 1-43 以 Lattice 器件为例说明预加重的效果。

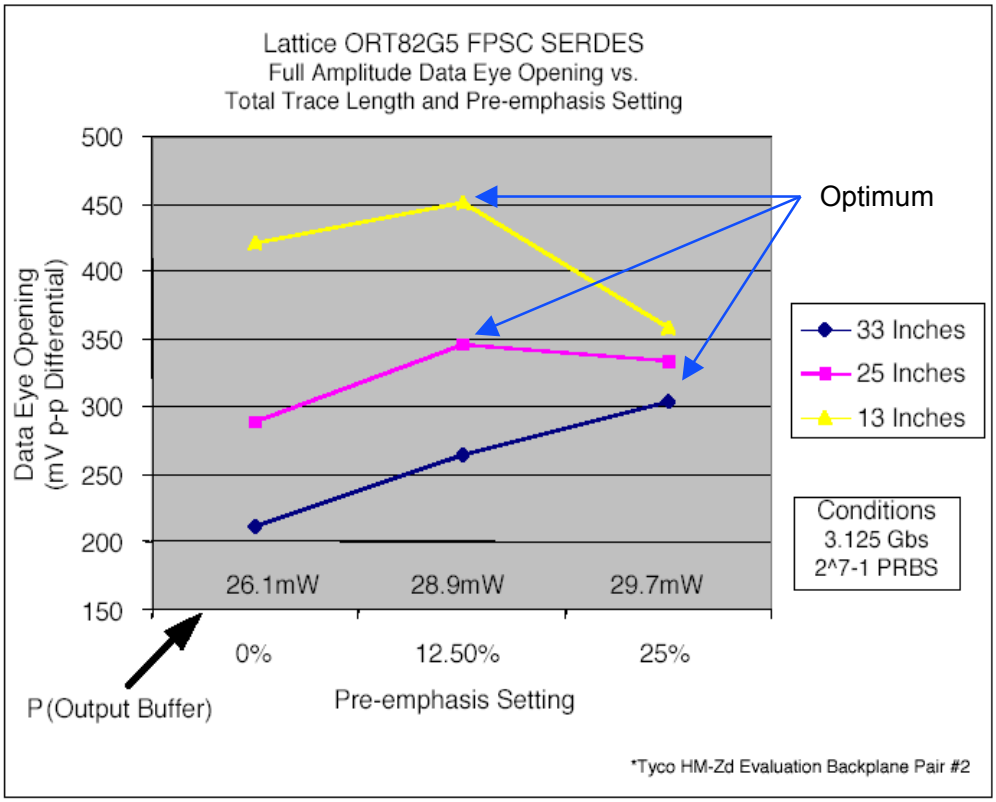


图1-43　预加重的效果

目前三大 FPGA 生产商 Xilinx、Altera、Lattice 的高端 FPGA 产品都包含有高速串行收发器的硬核，提供高达 3Gbit/s 的传输速率，并提供易于使用的设计软件和 IP 核，使高速传输电路的设计变得简便、可靠。

Xilinx 含有高速串行收发器的主流产品是 Virtex 2 Pro，它是 FPGA 中最早的高速串行收发器之一，其传输速率为 600Mb/s～3.125Gb/s；Lattice 含有高速串行收发器的主流产品是 ORT/ORS82G5、42G5 和 ORT8850 FPSC（Filed Programmable System on Chip）以及 GDX2 等器件族，它们提供丰富的软核和硬核专业致力于高速背板的设计，其传输速率为 126Mb/s～3.7Gb/s；；Altera 含有高速串行收发器的主流产品是 Stratix GX，它同 Lattice 的高速收发器相似，也提供了丰富的软核和硬核以满足不同系统的需求，其传输速率为 622M～3.1875GHz。

由于 Lattice 目前专业致力于 SERDES 的技术研究，其提供的软硬核比较丰富，器件指标也最具代表性，所以下面以 Lattice 为例，介绍 SERDES 的功能与性能。表 1-5 是 Lattice 公司提供的 SERDES 可编程器件。

表1－10.　Lattice Programmable SerDes

器件	可编程特性	高速接口特性
ORT4/82G5 8-Channel x 1.25 / 2.5 / 3.125 Gbit/s Transceiver	•ORCA Series 4 FPGA technology •Up to **400K programmable gates** •432 Programmable user I/O •111Kb Embedded RAM •I/O support includes HSTL, LVDS, SSTL, LVPECL, PECL, TTL, CMOS •Interface to three extra 2Kx32 dual-port RAMs (in the embedded section)	**•4/8 channels at 1.25/2.5/3.125 Gbit/s** •High-speed CML I/Os with internal termination •Total aggregate bandwidth – 10/20 Gbit/s •Power-down option on each HSI receiver **•Embedded 8B/10B coder/decoder** •Cell Processing •High-speed 680 PBGAM Packaging **•Multi-channel alignment FIFOs** •Two 4Kx36 Memory blocks in embedded macro
ORSO4/82G5 8-Channel x 1.35 / 2.7 Gbit/s Transceiver	•ORCA Series 4 FPGA technology •Up to **400K programmable gates** •432 Programmable user I/O •111Kb Embedded RAM •I/O support includes HSTL, LVDS, SSTL, LVPECL, PECL, TTL, CMOS •Interface to three extra 2Kx32 dual-port RAMs (in the embedded section)	**•4/8 channels at 1.35/2.7 Gbit/s** •High-speed CML I/Os with internal termination •Total aggregate bandwidth – 10/20 Gbit/s •Power-down option on each HSI receiver **•SONET scrambler** **•Cell Processing** interface to devices •High-speed 680 PBGAM Packaging **•Multi-channel alignment FIFOs** •Two 4Kx36 Memory blocks in embedded macro
ORT8850H 8-Channel x 850 Mbit/s Transceiver	•ORCA Series 4 FPGA technology •Up to **600K programmable gates** •536 Programmable user I/O •147Kb Embedded RAM	**•8 channels of 850 Mbit/s** •Total aggregate bandwidth - 6.8 Gbit/s •LVDS I/Os compliant with EIA-644 •3 full-duplex DDR I/O groups: RapidIO-like I/F •Powerdown option on each HSI receiver **•Pseudo-SONET framer including A1/A2** •SONET scrambler •In-band management & configuration **•Multi-channel alignment FIFOs**
ISPGDX2 (64,128 & 256) 4 To 16 Channels 400Mbs to 850Mbs Low Cost (>$2.00 / Ch)	•High speed high channel count •Up to **256 User** Programmable **I/O's•**I/O support includes HSTL, LVDS, SSTL, LVPECL, GTL+, TTL, CMOS	**•4/16 channels at 400Mbs/850Mbs** •High-speed, Low Cost LVDS I/Os •Total aggregate bandwidth – 3.4/13.6 Gbit/s **•Embedded 10B/12B coder/decoder** •Low Latency
ISPXPGA 4 To 20 Channels 400Mbs to 850Mbs Non-Volatile	•ISPXP FPGA technology •Up to **30,000 Registers** •496 Programmable user I/O •415Kb Embedded RAM •I/O support includes HSTL, LVDS, SSTL, LVPECL, TTL, CMOS •	**•4/20 channels at 400Mbs / 850Mbs** •High-speed LVDS I/O •Total aggregate bandwidth – 3.4/17 Gbit/s **•Embedded 10B/12B coder/decoder**
ISPXPIO 1 Channel 9.95Gbs to 10.7Gbs SFI -4	**•Parallel LVDS data range from 622 to 670 Mbps** **•Single-chip solution** **•0.13u CMOS technology**	**•Low jitter clock multiplier** **•On-chip clock data recovery** **•16:1 serialization and 1:16 deserialization** **•Embedded limiting amplifier** **•Built-In-Self-Test (BIST)** **•Lowest power consumption at 0.8W**

表 1-6 是 Lattice 与 Xilinx 的 SERDES 性能比较。

表1－11. SERDES 性能比较

Feature	Virtex II Pro	LatticeORT82G5	Lattice ORSO82G5

Data Rate Gbps per channel	0.6-3.125 (limited to 2.5G on wire bonded packages)	1.0-3.7	1.0-2.7
Wire-Bond Packages	0.6 - 2.5 Gbps	1.0 - 3.7 Gbps	1.0 - 2.7 Gb/s
Flip-Chip Packages	0.6 - 3.125 Gbps	N/A	N/A
Max. Length FR-4 at 3.125 Gb/s	20 inches	34 inches	34 inches
Number of channels	4 - 16	8	8
Data rate independence btn ch	Yes (for each Rx-Tx pair)	yes, 2 ref clk grps	yes, 2 ref clk grps
Parallel interface to FPGA	8, 16 or 32bit	32bit	32bit
8B10B encoder & decoder	yes	yes	no
Separate Tx and Rx PLLs per channel	no	yes	yes
Multi-channel alignment	yes	yes	yes
SONET scrambling & framing	no	no	yes
I/O power supply	2.5V	1.5 or 1.8V	1.5 or 1.8V
Power per channel @3.125Gbps	350 mW, typ	225 mW, worst case	210 mW worst case at 2.5 Gb/s
Internal I/O termination			
Tx Termination	50 or 75 ohm, prgmbl	86 ohm	86 ohm
Rx Termination	50 or 75 ohm, prgmbl	50 ohm	50 ohm
Programmable output amplitudes	5	2	2
Programmable pre-emph levels	4	3	3
CDR run-length tolerance	75b	75b	75b
CDR lock range	+/-100ppm	TBD- currently +/-100ppm (note 3)	TBD currently +/-100ppm (note 3)
Loopbacks			
High Speed Serial Line Loopback	yes	yes	yes
Low Speed Parallel Line Loopback	yes	yes	yes
Low Speed Parallel System Loopback	no	yes	yes
Standard compliance			
Fibre Channel	yes	yes (no support for forced negative disparity)	no
XAUI	yes	yes	no
Gigabit Ethernet	yes	yes	no
Jitter spec			
Rcvr total jitter tolerance	0.65 UI	0.734 UI	0.734 UI
Rcvr Determ. jitter tolerance	0.41 UI	0.5 UI	0.5 UI
Xmtr random jitter at 2.5 Gb/s	?	0.12 UI	0.12 UI
Xmtr total jitter at 2.5 Gb/s	?	0.2 UI	0.2 UI
Embedded Link State Machines			
Fibre Channel	yes	yes	no
XAUI (10 GbE)	no	yes	no
Cell Processing Capabilities			
Cell Delineation	yes	no	yes
Cell Striping Across Channels	no	no	yes
Idle Insertion/Deletion	partial (w/ FPGA gates)	no	yes

SERDES 的使用方法非常简便，根据用户需求设计 SERDES 参数，然后直接在器件商提供的 IP Core 生成软件中设置参数，生成 IP Core 即可，其使用方法和一般的 IP Core 使用

方法相同。其实，SERDES 设计的最大挑战在于高速 PCB 的设计，PCB 设计工程师必须按照高速准则，参考器件商提供的设计技巧，画出高质量的 PCB，才能真正体现 SERDES 高速传输的优势来。

第2章 FPGA 设计的具体准则

本章主要论述相对独立于综合器和布局布线器之外，一般意义上的 Coding Style 和设计准则。本章的主要内容如下：

- HDL 语言的层次含义;
- Coding Style 的含义;
- 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;
- 慎用锁存器（Latch）;
- 使用 Pipelining 方法优化时序;
- 模块复用与 Resource Sharing;
- 逻辑复制;
- 香农扩展;
- 信号敏感表;
- 复位逻辑;
- FSM 设计的一般型原则;
- 用 Verilog 语言设计 FSM 的技巧;
- CPLD 原理与设计方法。

2.1 HDL 语言的层次含义

HDL，是 Hardware Description Language 的缩写，即硬件描述语言。是目前数字 ASIC、FPGA、CPLD 等的最重要的一种设计输入方式。与较早的 EDA 原理图（Shematic）输入方式相比，HDL 设计输入方式具有：利于由顶向下设计，利于模块的划分与复用，可移植性好，通用性好，设计不因芯片的工艺与结构的变化而变化，更利于向 ASIC 的移植等优点。

HDL 语言是分层次、类型的，最常用的层次概念有系统与标准级（System level）、功能模块级（Functional model level）、行为级（Behavioral level）、寄存器传输级（RTL，Register transfer level）和门级（Gate level）。其概念如下：

- 系统与标准级（System level）

  在大型系统的设计与实现中，首先要进行详细的系统规划和描述。此时 HDL 描述侧重于整体系统的划分和实现。对系统级的仿真侧重于对整个系统的功能和性能指标的考衡。系统级常用 C 语言和抽象程度较高的 HDL 语言描述，如 SystemC、CoWare C、SystemVerilog、Superlog 等。

- 功能模块级（Functional model level）

  这个层次侧重用描述系统的功能划分。其主要内容为将系统整体功能划分为可实现的具体功能模块，大致确定模块间的接口，如时钟、读写信号、数据流、控制信号等。在有些情况下，还要对根据系统要求，描述每个模块或进程的时序约束。另外在此层次，必须权衡整个系统多种的实现方式之优劣，优选出系统性能指标优，并且可以高效实现的设计方案。这个层次的仿真主要是考察每个功能的功能和基本时序情况。这个层次适合用抽象程度较高的 HDL 语言描述。

- 行为级（Behavioral level）

  行为级模块描述的最大特点是必须明确每个模块间的所有接口和边界。此时模块内部的功能已经明确，模块间的所有接口，顶层的输入、输出信号等在行为级已经被清晰的描述出来。在 FPGA 设计流程中，常用行为级描述方式编写测试激励。延时描述、监视描述等命令都是在编写测试激励过程中常用的行为级语法。行为级描述常用 HDL 语言为 Verilog 和 VHDL 等。

- 寄存器传输级（RTL，Register transfer level）

  寄存器传输级描述的最大特点是可以直接用综合工具综合为门级网表。RTL 模块间的引脚接口、结构功能、时钟周期等都已经明确。通过综合工具的自动编译、映射等综合步骤，可以将 RTL 级模块用 FPGA 内部的查找表（LUT）、寄存器（Register）和基本的硬件原语（Primitive）实现。FPGA 的硬件原语因器件和厂商不同而异，是 FPGA 的基本功能模块和实现单元，如 Block RAM、DLL/PLL 等。目前最常见 FPGA/CPLD 设计方式即为：根据功能要求，设计 RTL 级代码；使用综合工具，综合生成逻辑网表；调用器件商的集成工具，完成布局布线等实现步骤，并生成器件配置文件。其中，RTL 输入是整个设计的第一步，直接决定着设计的功能和效率。好的 RTL 设计能在满足逻辑功能的前提下，使设计的速度和面积达到一种平衡的优化。RTL 级描述最常用的 HDL 语言是 Verilog 和 VHDL 语言。

- 门级（Gate level）

  由于目前 FPGA 设计，大多数依靠专业综合工具完成从 RTL 级代码向门级代码的转换，所有设计者直接用 HDL 语言描述门级模型的情况越来越少，高效的综合工具将设计者从复杂烦琐的门级描述中彻底解脱出来。目前直接使用门级描述的场合一般是 ASIC 和 FPGA 设计中某些面积或时序要求较高的模块。门级描述的特点是整个设计用逻辑门实现，通过逻辑门的组合显化描述了设计的引脚、功能、时钟周期等所有信息。

基于 HDL 的 FPGA 设计、仿真完整流程如图 2－1 所示。图中用虚线框表示的步骤可以根据项目的复杂度省略，而实线框表示的步骤为必须执行的步骤。

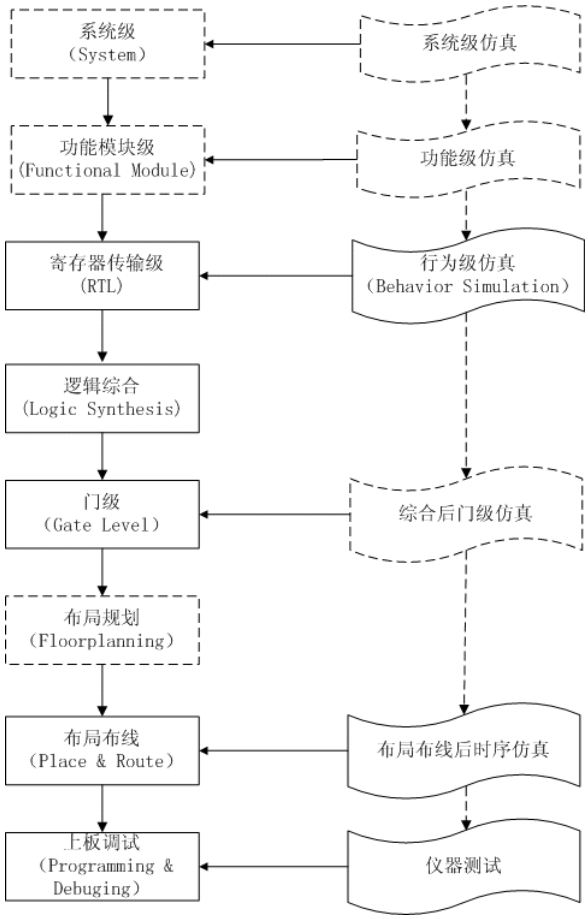


图1-44　基于 HDL 的 FPGA 设计、仿真流程

前面已经提到最流行的 HDL 语言是 Verilog 和 VHDL。在其基础上发展出了许多抽象程度更高的硬件描述语言，如 SystemVerilog、Superlog、SystemC、CoWare C，这些高级 HDL 语言的语法结构更加丰富，更适合作系统级、功能级等高层次的设计描述和仿真。HDL 语言的适用层次示意图如图 2－2 所示。其中实线框表示适用程度较高，虚线框表示适用程度较低。

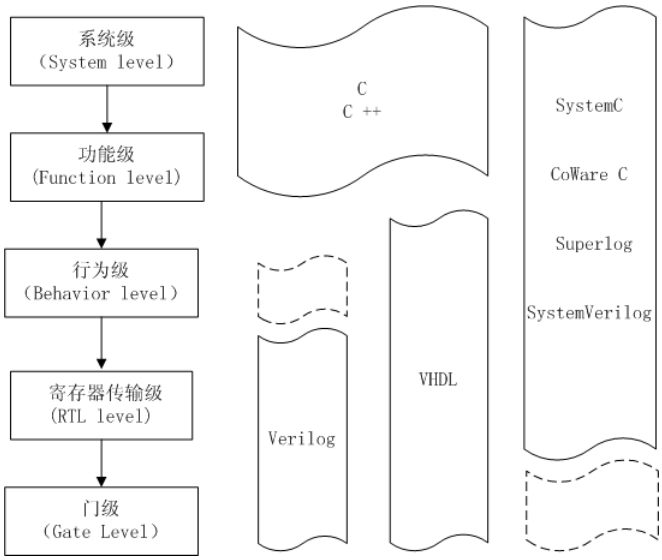


图1-45　HDL 语言的适用层次示意图

2.2 Coding Style 的含义

Coding Style，即所谓的代码风格，Coding Style 因对象不同而异。前面已经说过本章描述的重点是区别于综合、实现工具的一般意义的 RTL 级 Coding Style。这里有两层含义，首先本章所述为 RTL 级 Coding Style，重点在于如何优化 RTL 级代码，并非其它层次的描述方式。所以诸如业界炒的非常热的结构化设计方法（Architectural-based design）的代码风格的原则和方法与本章无关，甚至有很多原则和方法是与本章所述背道而驰的。

第二点，本章所描述的是不依赖于综合、实现工具的，一般意义上的代码风格。这一点深究起来比较容易混淆，首先大多数 Coding Style 都是和综合工具相关联的，善于利用某种综合工具的特点，采取该综合工具推荐的 Coding Style 编写 RTL 代码能够发挥该综合工具最大的潜能，达到事半功倍的效果。另一方面，不同的综合工具对一些语法细节的阐释略有不同，特别是那些关于优先级，实现的先后顺序等，所以不同的综合工具在个别细节上对 Coding Style 的解释有一定的差异。第三方面，有些 Coding Style 是针对某种器件的特点硬件结构的，用该厂商推荐的 Coding Style 能够正确的实例化这类底层单元，合理的适用其固有的硬件结构，这种 Coding Style 是与器件联系最紧密的。为了避免厚此薄彼，妄加评论，本章力图换一个角度，讨论一些一般综合工具都支持，独立于器件之外的一般意义上的 Coding Style 的建议和准则。

由于公司内部使用的 HDL 编码规范是基于 Verilog 的，所以本章讨论与示列将更侧重于 Verilog 语言。

2.3 结构层次化编码（Hierarchical Coding）

结构层次化编码是模块化设计思想的一种体现。目前大型设计中必须采用结构层次化编码风格，以提高代码的可读性，易于模块划分，易于分工协作，易于设计仿真测试激励。最基本的结构化层次是由一个顶层模块和若干个子模块构成，每个子模块根据需要还可以包含自己的子模块。结构层次化编码结构如图 2－3 所示。

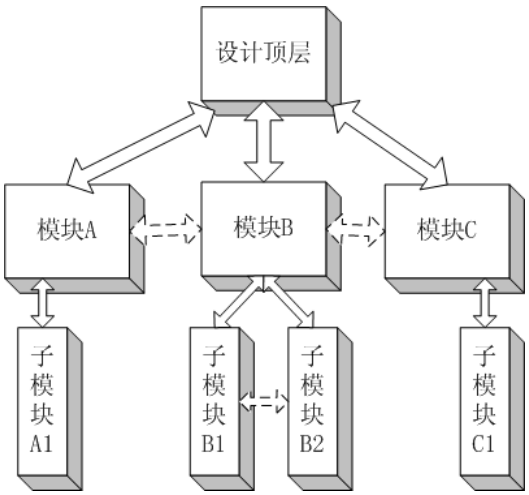


图1-46　结构层次化编码示意图

结构层次化编码有如下注意事项：

- 结构的层次不易太深，一般为 3 到 5 层即可。在综合时，一般综合工具为了获得更好的综合效果，特别是为了使综合结果所占用的面积更小，会默认将 RTL 代码的层次打平。而有时为了在综合后仿真和布局布线后的时序仿真中较方便的找出一些中间信号，比如子模块之间的接口信号等，可以在综合工具中设置保留结构层次，以便于仿真信号的查找和观察。
- 顶层模块最好仅仅包含对所有模块的组织和调用，而不应该完成比较复杂的逻辑功能。较为合理的顶层模块由：输入输出管脚声明、模块的调用与实例化、全局时钟资源、全局置位/复位、三态缓冲和一些简单的组合逻辑等构成。
- 所有的 IO 信号，如输入、输出、双向信号等的描述在顶层模块完成。
- 子模块之间也可以有接口，但是最好不要建立子模块间跨层次的接口，例如上图中模块 A1 到模块 B1 之间不宜直接连接，两者需要交换的信号可以通过模块 A，模块 B 的接口传递。这样做的好处是增加了设计的可读性和可维护性。
- 子模块的合理划分非常重要，应该综合考虑子模块的功能、结构、时序、复杂度等多方面因素。

2.4 模块的划分的技巧（Design partitioning）

模块划分非常重要，关系到是否最大程度上发挥项目成员协同设计的能力，更重要的是它直接决定着设计的综合、实现的耗时与效率。下面是一些模块划分的基本原则：

- **对每个同步时序设计的子模块的输出使用寄存器**（registering）。
  本原则也被称为用寄存器分割同步时序模块的原则。使用寄存器分割同步时序单元的好处有：便于综合工具权衡所分割的子模块中的组合电路部分和同步时序电路部分，从而达到更好的时序优化效果。而且这种模块划分符合时序约束的习惯，便于利用约束属性进行时序约束。
- **将相关的逻辑或者可以复用的逻辑划分在同一模块内**。
  该原则有时被称为呼应系统原则。这样做的好处有，一方面将相关的逻辑和可以复用的逻辑划分在同一模块，可在最大程度上的复用资源，减少设计所消耗的面积。同时也更利于综合工具优化某个具体功能的时序关键路径。其原因是，传统的综合工具只能同时优化某一部分的逻辑，而所能同时优化的逻辑的基本单元就是模块，所以将相关功能划分在同一模块将在时序和面积上获得更好的综合优化效果。
- **将不同优化目标的逻辑分开**。
  第一章第一个设计原则就说速度和面积平衡与互换。其中谈到合理的设计目标应该综合考虑面积最小和频率最高两个指标。好的设计，在规划阶段设计者就应该初步规划了设计的规模和时序关键路径，并对设计的优化目标有一个整体上的把握。对于时序紧张的部分，应该独立划分为一个模块，其优化目标为"speed"，这种划分方法便于设计者进行时序约束，也便于综合和实现工具进行优化。例如时序优化的利器 Amplify，以模块为单元进行物理区域约束，从而优化关键路径时序，以达到更高的系统工作频率就更为方便有效。另一类情况是：设计的矛盾主要集中在芯片的资源消耗上。这时应该将资源消耗过大的部分划分为独立的模块，这类模块的优化目标应该定为"Area"。同理，将他们规划到一起，更有利于区域布局与约束。这种根据优化目标进行优化的方法的最大好处是，对于某个模块综合器仅仅需要考虑一种优化目标和策略，从而比较容易达到较好的优化效果。相反的如果同时考虑两种优化目标，会使综合器陷入互相制约的困境，造成耗费巨大的综合优化时间也得不到令人满意的综合优化结果的局面。
- **将松约束的逻辑归到同一模块**。
  有些逻辑的时序非常宽松，不需要较高的时序约束，可以将这类逻辑归入同一模块，如多周期路径（multi-cycle path）等。将这些模块归类，并指定松约束，则可以让综合器尽量的节省面积资源。
- **将存储逻辑独立划分成模块**。
  RAM、ROM、CAM、FIFO 等存储单元应该独立划分模块。这样做的好处是

便于利用综合约束属性显化指定这些存储单元的结构和所使用的资源类型，也便于综合器将这些存储单元自动类推为指定器件的硬件原语。另一个好处是在仿真时消耗的内存也会少些，便于提高仿真速度。这是因为大多数仿真器对大面积的 RAM 都有独特的内存管理方式，以提高仿真效率。

- **合适的模块规模**。
  从理论上将，模块的规模越大，越利于模块资源共享（Resource Sharing）。但是庞大的模块，将要求对综合器同时处理更多的逻辑结构，这将对综合器的处理能力和计算机的配置提出了较高的要求。另外庞大的模块划分，不利于发挥目前非常流行的增量综合与实现技术的优势。

2.5 判断比较语句 case 和 if...else 的优先级

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，推荐使用 case 替换 if...else 结构。做两点补充：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。事实上，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。

2.6 慎用锁存器（Latch）

同步时序设计要尽量避免使用 Latch。综合出非目的性 Latch 的主要原因在于：不完全的条件判断语句，如 if...而没有 else...(这仅仅是一种可能，并不一定生成 Latch)。另外一种情况是设计中有组合逻辑的反馈环路（combinatorial feedback loops）等异步逻辑。

典型的生成 Latch 的 Verilog 和 VHDL 语句如下：

Verlog：

```
reg data_out;
always @(cond_1, data_in)
  begin
    if (cond_1)
        data_out <= data_in;
  end
```

VHDL：

```
process(cond1)
  begin
    if (cond_1 = '1') then
        data_out <= data_in;
    end if;
```

```
    end process;
```

上述描述，由于未指定在条件“cond_1”等于“0”时的动作，一般情况下会生成如图 2－4 所示的 Latch 结构。

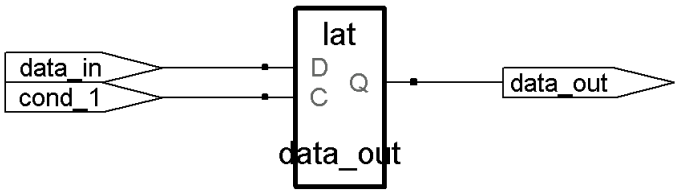


图1-47　Latch 的 RTL 示意图

防止产生非目的性的 Latch 的措施为：

- 使用完备的 if...else 语句。
- 检查设计中是否含有组合逻辑反馈环路。
- 为每个输入条件，设计输出操作，对 case 语句设置 default 操作。特别是在状态机设计中，最好有一个 default 的状态转移，而且每个状态最好也有一个 default 的操作。
- 如果使用 case 语句时，特别是在设计状态机时，尽量附加综合约束熟悉，综合为完全条件 case 语句（full case）。目前，大多数综合工具支持 full case 的综合约束熟悉，具体的语法请参考综合工具的约束属性指南。
- 仔细检查综合器的综合报告，目前大多数的综合器对所综合出的 Latch 都会报"warning"，通过综合报告可以较为方便的找出无意中生成的 Latch。

2.7 使用 Pipelining 技术优化时序

Pipelining，即流水线时序优化方法。其本质是调整一个较长的组合逻辑路径中的寄存器位置，用寄存器合理分割该组合逻辑路径，从而降低了对路径的 clock-to-output 和 setup 等时间参数的要求，达到提高设计频率的目的。但是必须要注意的是，使用 Pipelining 优化技术只能合理的调整寄存器位置，而不应该凭空增加寄存器级数，所以 Pipelining 有时也较 Register Balance。

目前一些先进的综合工具能根据用户参数配置，自动运用 Pipelining 技术，通过用寄存器平衡设计中的较长组合路径（Register Balance），在一定程度上提高设计的工作频率。这种时序优化手段对乘法器、ROM 等单元效果显著。

2.8 模块复用与 Resource Sharing

第一章系统原则中，已经讨论如何在系统层次上复用硬件模块，并通过例 4 举例说明。“硬件原则”是站在宏观的角度分析，而本章的模块复用和 Resource Sharing 主要站在微观的角度观察节约面积的问题。为了便于理解，首先我们看两例子。

例 19.　　Verilog Resource Sharing 的例子，一个补码平方器

这是一个补码平方器的例子，输入是 8bit 补码，求其平方和。由于输入是补码，所以当最高位是 1 时，表示原值是负数，需要按位取反，加 1 后再平方；当最高位是 0 时，表示原值是正数，直接求平方。

下面是两种描述方式，请读者判断一下优劣，并体会 Resource Sharing 的含义。

第一种实现方式：

```
module resource_share (data_in, square);
input [7: 0]   data_in;     //输入是补码
output [15: 0]  square;
wire  [7: 0]   data_bar;

assign data_bar = ~data_in + 1;
assign square=(data_in[7])? (data_bar*data_bar) :  (data_in*data_in);
endmodule
```

第二种实现方式：

```
module resource_share (data_in, square);
input [7: 0]   data_in;      //输入是补码
output [15: 0]  square;
wire  [7: 0]   data_tmp;

assign data_tmp = (data_in[7])? (~data_in + 1) :  data_in;
assign square = data_tmp * data_tmp;
endmodule
```

仔细观察一下可以发现：第一种实现方式需要两个 16bit 乘法器，同时平方，然后根据输入补码的符号选择输出结果，其关键在于使用了两个乘法器，选择器在乘法器之后。而第二种实现方法，首先根据输入补码的符号，换算为正数，然后做平方，其关键在于选择器在乘法器之前，仅仅使用了一个乘法器，节约了资源。第二种实现方式与第一种实现方式相比节约的资源有两部分：第一部分，节约了一个 16bit 乘法器；第二部分，后者的选择器是 1bit 判断 8bit 输出，而前者的是 1bit 判断 16bit 输出。

两种代码的硬件结构示意图如图 2－4，2－5 所示。

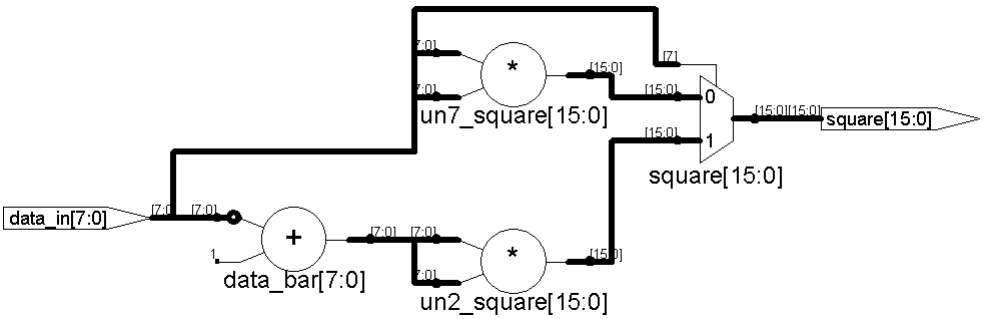


图1-48　未 Resource Sharing，2 个乘法器的实现方案

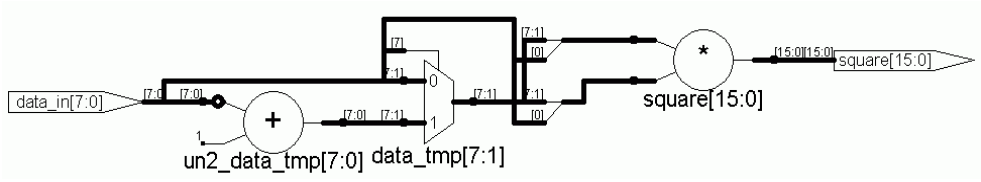


图1-49　Resource Sharing，1 个乘法器的实现方案

细心的读者也许已经注意到，本例所示的结构示意图是 Synplify Pro 综合后，提取的 RTL 视图。这里需要强调的有 5 点：

7. 本例综合选用的工具是 Synplify Pro 7.2，目标器件为：Altera Cyclone EP1C3 - 6 TC100，并关闭了“Resource Sharing”等所有优化参数。第一种实现方法，所占逻辑资源为 236 ATOMs；第一种实现方法，所占逻辑资源为 112 ATOMs。两者所占资源相差一倍以上！
8. 上例资源共享的单元是乘法器，通过 Resource Sharing，节省了一个乘法器和一些选择器占用的资源。其实如果拓广一下思维，将乘法器换成加法器，除法器等等，甚至推广到任何一个普通的模块，后续结构含有选择器，都可以使用本例的设计思想，通过 Resource Sharing 成倍的节省前级模块所消耗的资源。
9. 不同的综合工具、同一综合工具的不同版本、不同的优化参数、不同厂商的目标器件、同一厂商的不同器件族等因素都可能造成不同的综合结果。
10. 目前很多综合工具都“Resource Sharing”之类的优化参数，选择该参数，综合工具会自动考察设计中是否有可以资源共享的单元，在保证逻辑功能不变的情况下，进行 Resource Sharing，以获得面积更小的综合结果。例如上例中，打开 Synplify Pro 的“Resource Sharing”综合优化参数，Synplify Pro 会自动运用资源共享的优化算法，其综合结果将于第二种代码描述的综合结果完全一致。
11. 最后我们需要强调的是不能因为综合工具的优化能力增强，而片面依靠综合工具，放松对自己 Coding Style 的要求。这是因为：第一，综合工具的优化力度毕竟有限，很多情况不能智能的发现需要 Resource Sharing 的逻辑；第二，前面已经说过，“不同的综合工具、同一综合工具的不同版本、不同的优化参数、不同厂商的目标器件、同一厂商的不同器件族等因素”都会直接影响综合工具的优化能力和效果，所以依靠综合工具的优化能力十分不可靠；第三，在 ASIC 设计中，综合工具非常忠于作者意图，这时 Coding Style 更加重要。所以逻辑工程师必须要注意自己 Coding Style 方面的修养与提高。

(还有一个 Xilinx、Lattice 不同综合工具的 VHDL 例子，另外还有一个好的 resource share，改代码前，改代码后，使用参数 resource share 前，使用后等的四种情况。)

2.9 逻辑复制

逻辑复制，是一种通过增加面积而改善时序条件的优化手段。逻辑复制最常使用的场合是调整信号的扇出。如果某个信号需要驱动后级的很多单元，换句话说，也就是其扇出非常大，那么为了增加这个信号的驱动能力，必须插入很多级 buffer，这样就在一定程度上增加了这个信号路径的延时。这时我们可以复制生成这个信号的逻辑，使多路同频同相的信号驱动后续电路，平均到每路的扇出变低，不需要加 buffer 也能满足驱动能力的要求，这样就节约了该信号的路径时延。如图 2－6 所示。

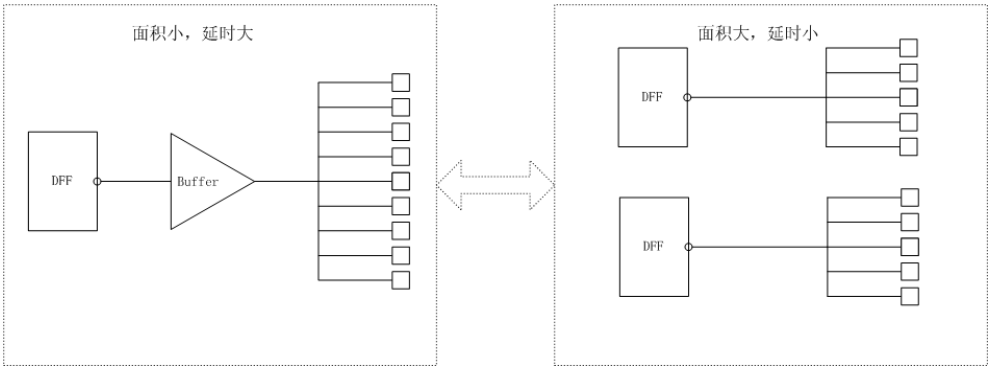


图1-50　用逻辑复制改善扇出

需要说明的是，现在很多综合工具都可以自动设置最大扇出值（Max Fanout），如果某个信号的扇出值大于最大扇出，则该信自动被综合工具复制。最大扇出值和器件的工艺密切相关，其合理值应该根据器件手册的声明和工程经验设置。这里举例用逻辑复制手段调整扇出，达到优化路径时延仅仅是为了讲述逻辑复制的概念，其实逻辑复制还有其它很多形式。例如香农扩展（Shannon Expansion）等时序优化技术。香农扩展在后面将会有详细的介绍。

有的读者感到，逻辑复制和资源共享是两个矛盾的概念，既然使用了资源共享优化技术，为什么还要做逻辑复制呢？

其实这个问题的本质，还是面积与速度的平衡。逻辑复制与前面的 Resource Sharing 是两个对立同一的概念。Resource Sharing 的目的是为了节省面积资源，而逻辑复制的目的是为了提高工作频率。当使用逻辑复制手段提高工作频率的时候，必然会增加面积资源，这是与资源共享相对立的方面；但是正如前面第一章面积与速度的对立统一一样，逻辑复制和资源共享都是要到的设计目标的两种手段，一个侧重于速度目标，一个侧重于面积目标，两者存在一种转换与平衡的关系，所以两者又是统一的。

首先看下面的一个例子。

例 20.　　一个加法器的资源共享例子。

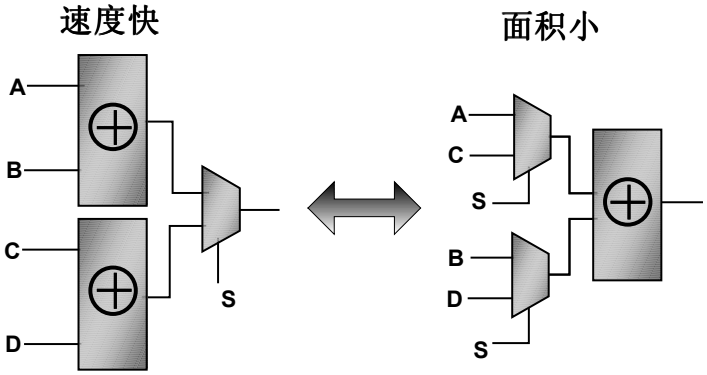
这个例子和前面乘法器的例子非常相似，只是将平方器换成了加法器。实现这个加法器也有两种代码写法，对应两种不同的硬件结构，如图所示。

第一种写法，对应左边的 RTL 结构示意图：

```
assign data_out= (sel)? (a+b) :  (c+d) ;
```

第二种写法，对应右边的 RTL 结构示意图：

```
wire temp1，temp2;
assign temp1 = (sel)? (a) :  (c) ;
assign temp1 = (sel)? (a) :  (c) ;
assign data_out = temp1+temp2;
```



第一种写法比第二种省了一个加法器。但是严格的来讲，第一种写法比第二种写法耗时略微小一些，这在本例还不算十分明显，当运算模块不是加法器而是一些较复杂的逻辑时，会比较明显。当设计的时序满足要求，或者设计的面积紧张时，我们一般会采用资源共享的优化方法，将第一种设计转换为第二种设计，绝大多数情况如是；但是在某些特殊情况下，时序非常紧张，我们会反其道而行之，将第二种设计转换为第一种设计，从而便于调整组合逻辑信号的到达时间，提高这个加法选择器的工作频率。

2.10　香农扩展运算

前面已经讲到，香农扩展（Shannon Expansion）也是一种逻辑复制，增加面积，提高频率的时序优化手段。

其概念如下，布尔逻辑可以做如下扩展，

$$F(a,b,c) = aF(1,b,c) + \bar{a}F(0,b,c)\,.$$

从上面的定义可以看到，香农扩展即布尔逻辑扩展，是卡诺逻辑化简反向运算，香农扩展相当于逻辑复制，提高频率；而卡诺逻辑化简相当于资源共享，节约面积。

香农扩展通过增加 MUX，从而缩短了某个优先级高，但是组合路径长的信号的路径时延，从而提高了该关键路径的工作频率。通过下面的例子，读者会对香农扩展有一个更全面的理解。

例 21.　　使用香农扩展优化组合逻辑时序

设所需运算的逻辑表达式为：

F = ((({8{late}} | in0) + in1) = = in2) & en;

其中信号 in0，in1，in2 都是 8bit 的数据，信号 late 和信号 en 是控制信号，信号 late 是

本逻辑运算的关键路径信号，延时最大。

使用香农扩展，

F = late.F(late=1) + ~late.F(late=0)

= late.[((({8{1'b1}} | in0) + in1) = = in2) & en] +

~late.[((({8{1'b0}} | in0) + in1) = = in2) & en]

= late.[(8'b1+in1) = = in2) & en] + ~late[((in0 + in1) = = in2) & en]

这相当于一个以 late 为选择信号，以[(8'b 1+in1) = = in2) & en]和[((in0 + in1) = = in2) & en]为两个输入信号的 2 选 1 的 MUX。因此，late 信号的优先级被提高，其信号路径的延时降低，但是其代价是设计的面积增加了，并且需要两个比较运算符。

未 shannon 扩展前的 verilog 代码描述如下：

```
module un_shannon (in0, in1, in2, late, en, out);
input [7: 0] in0, in1, in2;
input late, en;
output out;
  assign out = ((({8{late}} | in0) + in1) == in2) & en;
endmodule
```

使用 Synplify Pro 7.2 综合，选择目标器件为 Lattice 的 ORCA Series4 4E02，所得到的 RTL 视图如 2－7 所示。

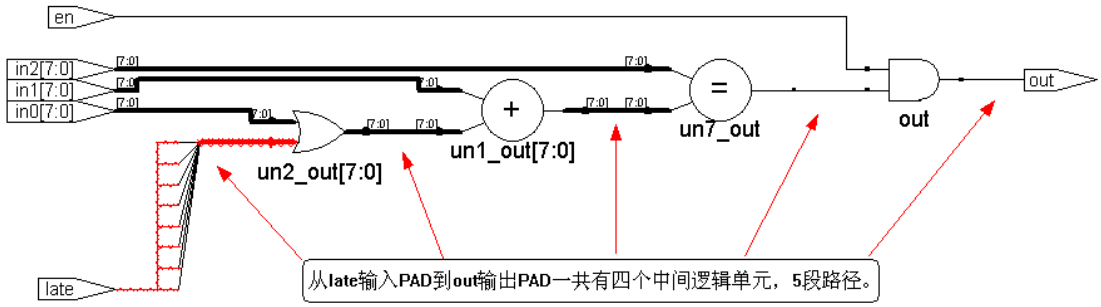


图1-51 未使用香农扩展前的逻辑表达式对应的 RTL 视图

图中可以清晰的看到，未使用 Shannon 扩展前，从输入 PAD late 到输出 PAD out 之间共有 4 个逻辑单元，5 段路径。其综合结果使用了 1 个 8bit 或门，1 个 8bit 输入加法器，1 个 8bit 比较器，1 个 2 输入与门。采用默认参数，用 Synplify Pro 在上述 Lattice 目标器件上综合，其结果的工作频率约为 50.8Mhz。

而使用 Shannon 扩展的 Verilog 代码如下，

```
module shannon_fast (in0, in1, in2, late, en, out);
input [7: 0] in0, in1, in2;
input late, en;
output out;
wire late_eq_0, late_eq_1;
assign late_eq_0 = ((in0+in1) == in2) & en;
```

```
//assign late_eq_0 = ((({8{1'b0}} | in0) + in1) == in2) & en;
assign late_eq_1 = ((8'b1+in1) == in2) & en;
//assign late_eq_1 = ((({8{1'b1}} | in0) + in1) == in2) & en;
assign out = (late) ? late_eq_1 :  late_eq_0;
endmodule
```

为了方便读者，在此一并提供对应的 VHDL 代码如下，

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity shannon_fast is
port( in0,  in1,  in2 :  in std_logic_vector (7 downto 0);
      late,  en :  in std_logic;
      out1 :  out std_logic);
end shannon_fast;
architecture arch_shannon_fast of shannon_fast is
signal late_eq_0,  late_eq_1,  late_0,  late_1 :  std_logic;
begin
    late_0 <= '1' when ((("00000000" or in0) + in1) = in2) else '0';
    late_eq_0 <= late_0 and en;
    late_1 <= '1' when ((("11111111" or in0) + in1) = in2) else '0';
    late_eq_1 <= late_1 and en;
    out1 <= late_eq_1 when (late = '1') else late_eq_0;
end arch_shannon_fast;
```

同样，使用 Synplify Pro 7.2 综合，选择器件目标为 Lattice 的 ORCA Series4 4E02，所得到的 RTL 视图如 2－8 所示。

在图中可以清晰的看到，使用 Shannon 扩展后，从输入 PAD late 到输出 PAD out 之间共有 1 个逻辑单元，2 段路径。其综合结果使用了 2 个 8bit 输入加法器，2 个 8bit 比较器，2 个输入与门和一个 2 输入选择器。采用默认参数，用 Synplify Pro 在上述 Lattice 目标器件上综合，其结果的工作频率约为 67.4Mhz。

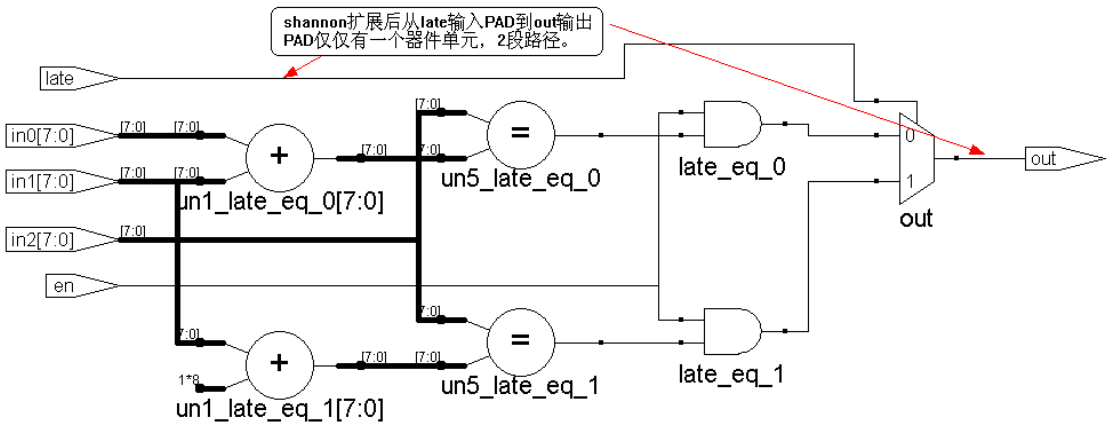


图1-52　香农扩展后的逻辑表达式对应的 RTL 视图

虽然使用不同的器件族，综合结果的工作频率不一致，但是从 RTL 视图可以清晰的看到，采用香农扩展后，对于 late 信号这一关键路径，消除了 3 个逻辑层次，从而大幅度的提高了设计的工作频率。作为提高工作频率的代价，是多用 1 个加法器和选择器，消耗了更多的面积。

正如前面反复强调的面积和速度的平衡关系所述，是否使用香农扩展时序优化手段，关键要看被优化对象的优化目标是面积还是路径。

2.11　信号敏感表

几乎所有的 Coding Style 指导手册有关于信号敏感表的论述。时序逻辑的信号敏感表比较好写，在信号敏感表写明时钟信号的正负触发沿即可，关于信号敏感表的主要问题集中在组合逻辑的信号敏感表的写法。在此，我们仅仅强调一下组合逻辑的信号敏感表的相关要点：

- 正确的信号敏感表设计方法是将操作进程（Verilog 的 always block 或 VHDL 的 process block）中使用到的所有输入信号和条件判断信号都列在信号敏感表中。
- 希望通过信号敏感表的增减完成某项逻辑功能是大错特错的。
- 不完整的信号敏感表会造成前仿真结果和综合、实现后仿真结果不一致。
- 一般综合工具对于不完整的信号敏感表的默认做法是，将处理进程中用到的所有输入和判断条件信号都默认添加到综合结果的信号敏感表中，并对原设计代码敏感表中遗漏的的信号报警告（warning）信息。

简单的解释一下上面这些要点。有些初学者发现在信号敏感表中增减一些信号，会得到不同的仿真结果，于是企图依靠修改信号敏感表，而方便地完成某些逻辑的设计，这种做法是大错特错的。其实一般综合工具的默认操作都是，将操作进程（Verilog 的 always block 或 VHDL 的 process block）中使用到的所有输入信号和条件判断信号都当做触发信号，综合到信号敏感表中，所以增减信号敏感表，其实得到的综合结果完全一致。而增减信号敏感表，

得到的仿真结果不同是因为仿真器的工作机制决定的，大多数仿真器是数据流和时钟周期驱动的，如果信号敏感表中没有某个信号，则无法触发和该信号相关的仿真进程，从而得到的仿真结果不同。

2.12　复位逻辑

复位逻辑是一个比较复杂的问题，“采用何种复位方式最合理？”这个问题和设计内容以及所选用的目标器件的底层结构息息相关，深入探讨起来内容很多。在此简单的介绍一下服务逻辑的分类。复位简单的分有同步复位逻辑、异步复位逻辑、全局复位逻辑。

同步复位、异步复位的设计方法的最大区别在于信号敏感表的列项上，如果信号敏感表中含有复位信号的正沿或者负沿触发，则属于异步复位逻辑。第一章对全局复位/置位信号已经做了较为详尽的论述，在此不再累述。总的来讲，采用同步复位逻辑的 FPGA 设计的工作频率较高，采用异步逻辑复位的 FPGA 设计较为简单。关于全局复位逻辑，不同厂商的器件的推荐设计不同，CPLD 一般推荐使用异步的全局复位逻辑；Xilinx 的 Virtex/E/II/II Pro，Spartan-II/III 等较新型 FPGA 器件族不推荐使用全局复位逻辑，因为上述器件族的 Xilinx 器件的 GSR 资源速度较低，skew 较大，不如直接采用全局时钟资源或者第二全局时钟资源驱动复位逻辑更高速；Lattice 的 FPGS 和 ORCA 系列 FPGA 恰恰相反，强烈推荐使用全局复位资源，使用 Lattice 的全局复位资源，可以达到最佳的设计性能和频率。

典型的同步复位逻辑如下：

```
module reset_form (clk, reset, data_in, data_out);
input        clk, reset;
input   [3: 0] data_in;
output  [3: 0] data_out;
reg     [3: 0] data_out;
always @ (posedge clk)
 if (!reset)
    data_out <= 4'h f;
 else
    data_out <= data_in;
endmodule
```

典型的异步复位逻辑如下：

```
module reset_form (clk, reset, data_in, data_out);
input        clk, reset;
input   [3: 0] data_in;
output  [3: 0] data_out;
reg     [3: 0] data_out;
always @ (posedge clk or negedge reset)
 if (!reset)
    data_out <= 4'h f;
```

```
    else
        data_out <= data_in;
endmodule
```

下面是采用不同厂商的器件下，Synplify Pro 在默认条件下类推的复位资源的结构视图。

图 2－10 是目标器件族为 Lattice ORCA 4，同步复位逻辑结构示意图。图中可以看到，Lattice ORCA 4 器件族使用了其底层带置位/复位触发器的 PD 同步复位端，完成上面代码的同步复位功能。

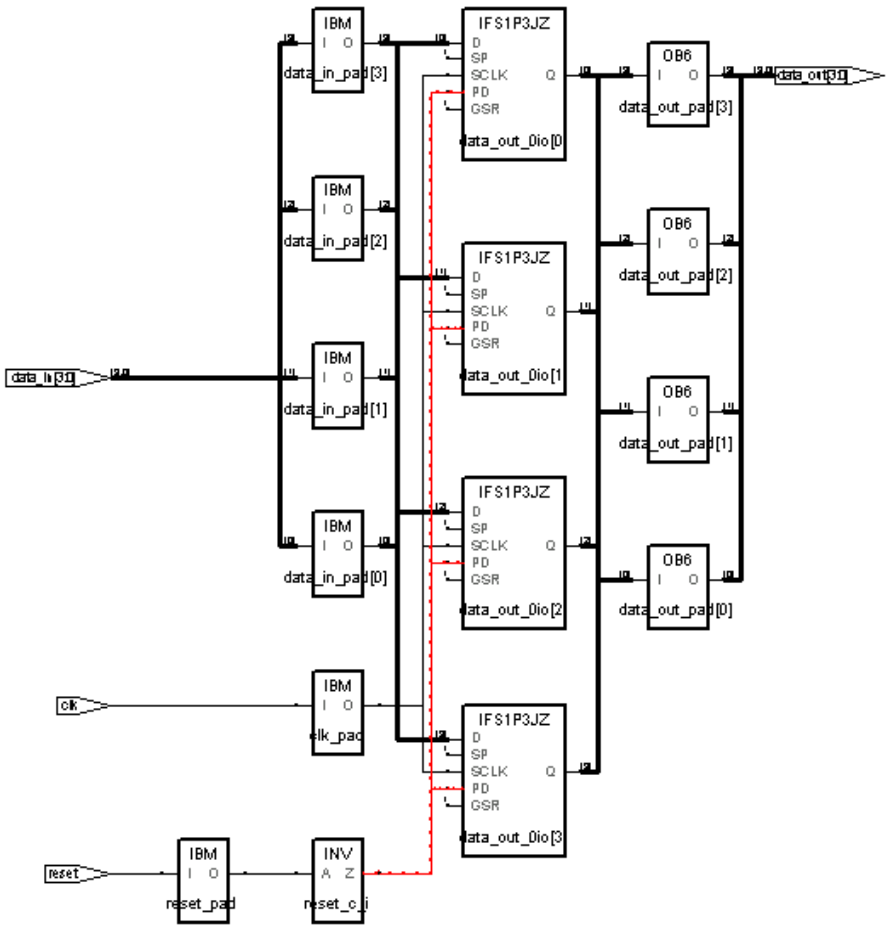


图1-53　Lattice ORCA 4 器件同步复位结构视图

图 2－11 是目标器件族为 Lattice ORCA 4，异步复位逻辑结构示意图。图中可以看到，Lattice ORCA 4 器件族使用了其底层带置位/复位触发器的 GSR 全局复位端，完成上面代码的异步复位功能。使用 Lattice 器件，采用异步 GSR，可以达到较高的工作频率。

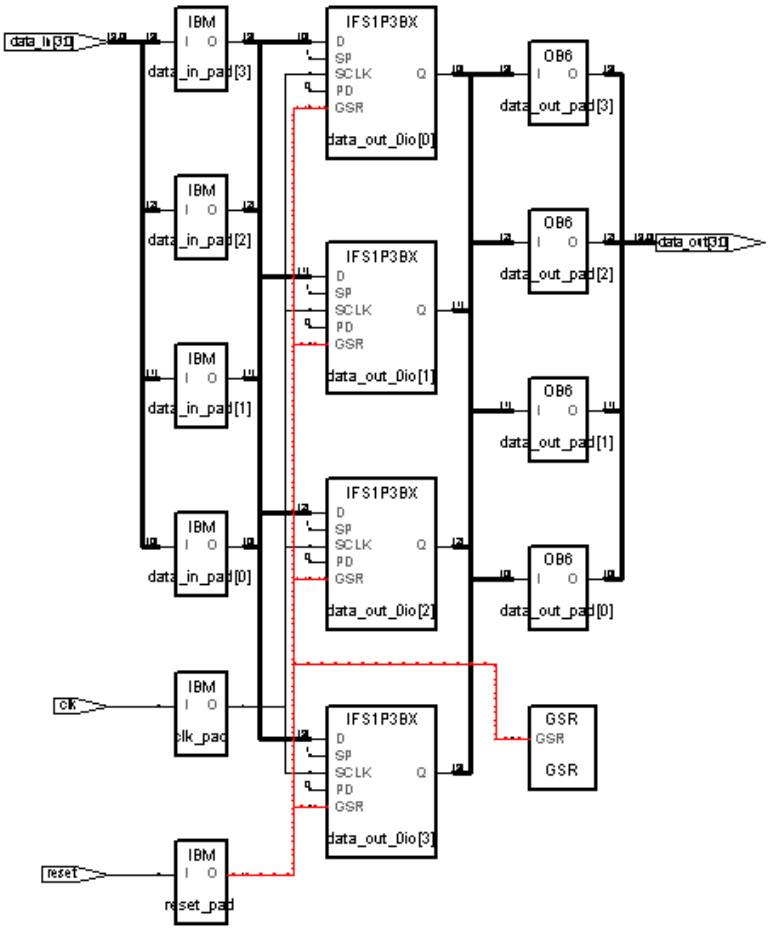


图1-54　Lattice ORCA 4 器件异步复位结构视图

图 2－12 是目标器件族为 Xilinx Virtex 2 器件族，同步复位逻辑结构示意图。图中可以看到，Xilinx Virtex 2 器件族使用了其底层带置位/复位 D 触发器的同步复位端 S，完成上面代码的同步复位功能。

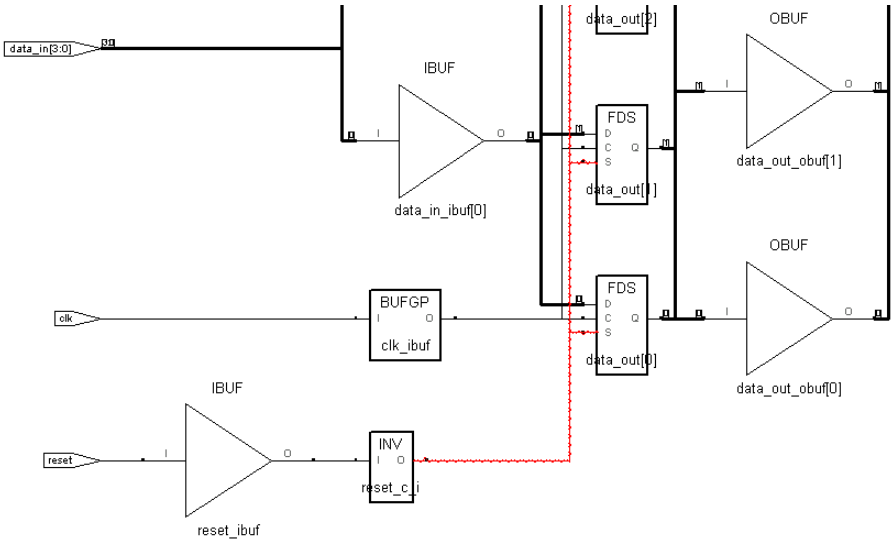


图1-55　Xilinx Virtex 2 器件同步复位结构视图

图 2－13 是目标器件族为 Xilinx Virtex 2 器件族，异步复位逻辑结构示意图。图中可以看到，Xilinx Virtex 2 器件族使用了其底层带置位/复位 D 触发器的异步置位端 PRE，完成上面代码的异步复位功能。另一方面，Xilinx Virtex 2 器件族并不推荐使用 GSR 端完成异步复位，如果复位逻辑的扇出很大或者时序要求较高时，可以使用全局时钟资源（BUFG）或者第二全局时钟资源（Low Skew Lines）驱动复位逻辑。

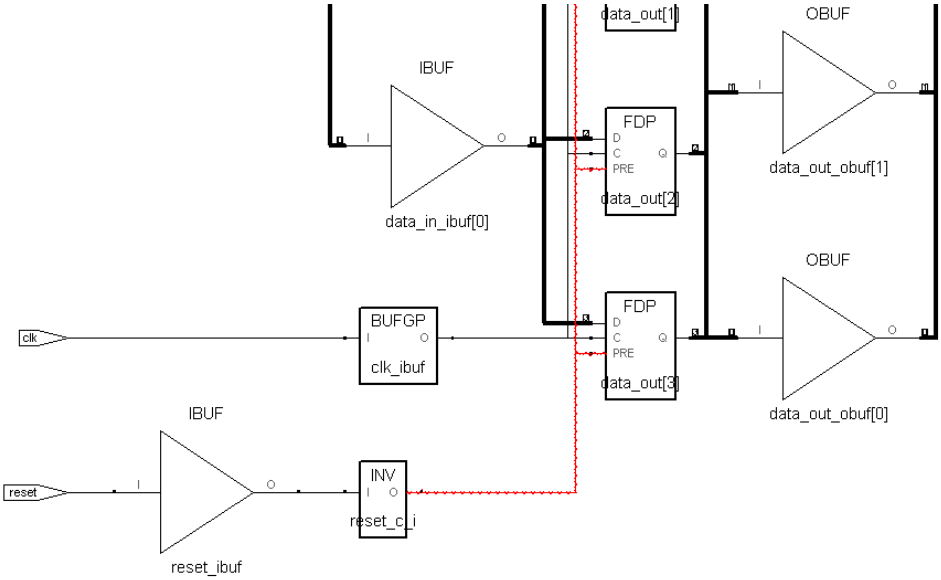


图1-56　Xilinx Virtex 2 器件异步复位结构视图

图 2－14 是目标器件族为 Altera Stratix 器件族，同步复位逻辑结构示意图。图中可以看到，Altera Stratix 器件族使用了其底层触发器的逻辑端 datab，完成上面代码的同步复位功能。

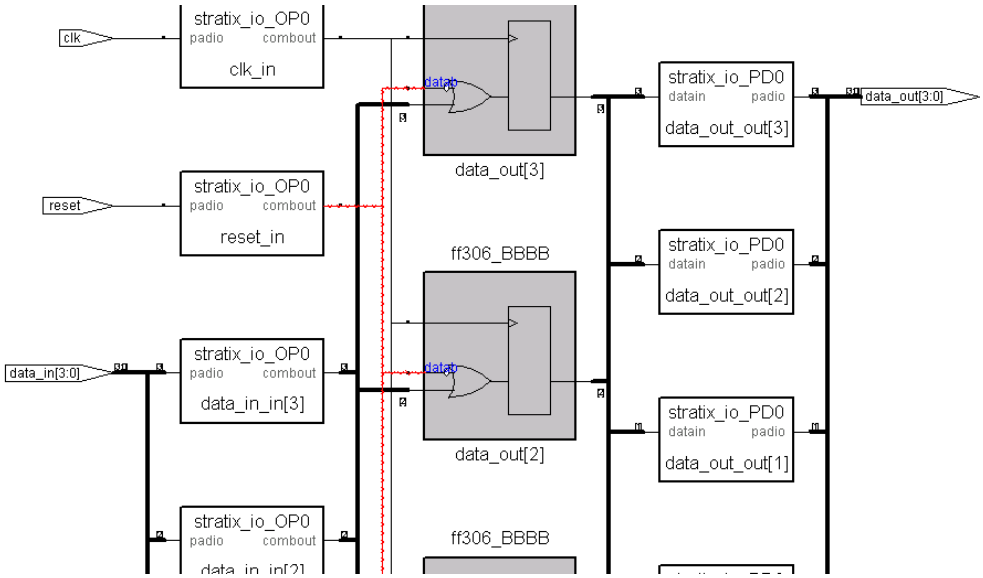


图1-57　Altera Stratix 器件同步复位结构视图

图 2－15 是目标器件族为 Altera Stratix 器件族，异步复位逻辑结构示意图。图中可以看到，Altera Stratix 器件族使用了其底层触发器的异步复位端 aclr，完成上面代码的异步复位功能。Altera 不同器件族的生产工艺差异较大，推荐的复位结构也各式各样，在此不一一叙述。

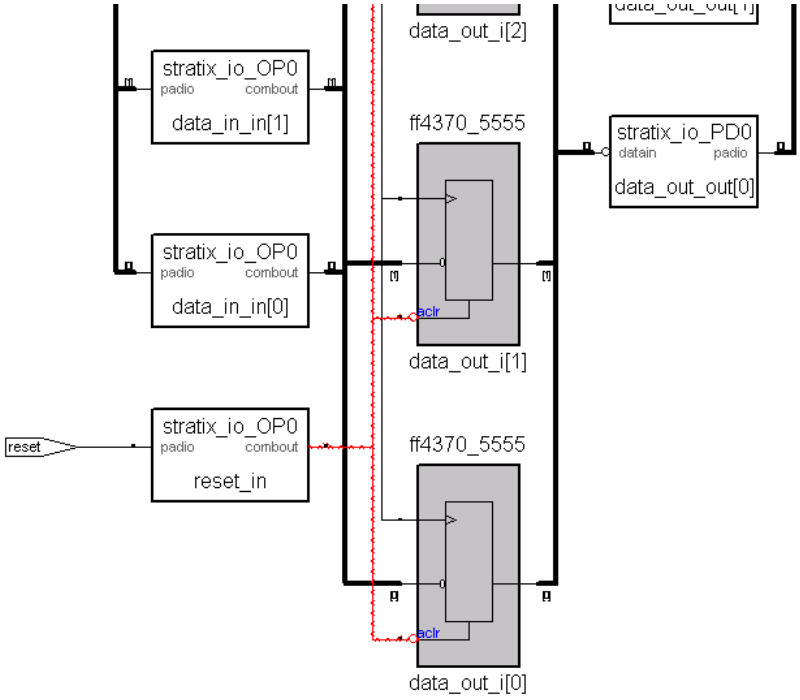


图1-58　Altera Stratix 器件同步复位结构视图

采用异步复位时，设计方法较为简单，只需注意复位信号的布线延时与 skew 必须满足

复位时间的要求即可，但是工作频率较低。采用同步复位逻辑时，由于时钟信号到来时才统一复位，必须注意电路的逻辑功能，特别是电路是否真正的完成了复位。

2.13 状态机设计的一般原则

状态机是逻辑设计中的最重要的设计内容之一，通过状态转移图设计手段可以将复杂的控制时序图形化表示，分解为状态之间的转换关系，将问题简化。使用 HDL 语言高效、完备、安全地描述状态机在一定程度上是一件体现代码功底的设计项目。下面是一些编写状态机的一般性原则。

- **状态机的编码**

  Biary、gray-code 编码使用最少的触发器，较多的组合逻辑。而 one-hot 编码反之。由于 CPLD 更多的提供组合逻辑资源，而 FPGA 更多的提供触发器资源，所以 CPLD 多使用 gray-code，而 FPGA 多使用 one-hot 编码。另一方面，对于小型设计使用 gray-code 和 binary 编码更有效，而大型状态机使用 one-hot 更高效。

  在代码中添加综合器的综合约束属性或者在图形界面下设置综合约束属性可以比较方便的改变状态的编码。Synplicity， Synopsys， Exemplar，XST 等综合工具的不同 FSM 综合属性。

  FSM 分两大类：米勒型和摩尔型。组成要素有输入（包括复位），状态（包括当前状态的操作），状态转移条件，状态的输出条件。

- **两段式状态机的设计方法**

  设计 FSM 的方法和技巧多种多样，但是总结起来有两大类：第一种，将状态转移和状态的操作和判断等写到一个模块（process、block）中。另一种是将状态转移单独写成一个模块，将状态的操作和判断等写到另一个模块中（在 Verilog 代码中，相当于使用两个"always" block）。

  其中较好的方式是后者。其原因如下。FSM 和其他设计一样，最好使用同步时序方式设计，其好处不再累述。而状态机实现后，状态转移是用寄存器实现的，是同步时序部分。状态的转移条件的判断是通过组合逻辑判断实现的，之所以第二种比第一种编码方式合理，就在于第二种编码将同步时序和组合逻辑分别放到不同的程序块（process，block）中实现。这样做的好处不仅仅是便于阅读、理解、维护，更重要的是利于综合器优化代码，利于用户添加合适的时序约束条件，利于布局布线器实现设计。

- **初始化状态和默认状态。**

  一个完备的状态机（健壮性强）应该具备初始化状态和默认状态。当芯片加电或者复位后，状态机应该能够自动将所有判断条件复位，并进入初始化状态。需要注明的一点是，大多数 FPGA 有 GSR（Global Set/Reset）信号，当 FPGA 加电后，GSR 信号拉高，对所有的寄存器，RAM 等单元复位/置位，这时配置于 FPGA 的逻辑并未生效，所以不能保证正确的进入初始化状态。所

以使用 GSR 企图进入 FPGA 的初始化状态，常常会产生种种不必一定的麻烦。一般的方法是采用异步复位信号，当然也可以使用同步复位，但是要注意同步复位逻辑的设计。解决这个问题的另一种方法是将默认的初始状态的编码设为全零，这样 GSR 复位后，状态机自动进入初始状态。

另一方面状态机也应该有一个默认（default）状态，当转移条件不满足，或者状态发生了突变时，要能保证逻辑不会陷入"死循环"。这是对状态机健壮性的一个重要要求，也就是常说的要具备"自恢复"功能。对应于编码就是对 case，if…else 语句要特别注意，尽量使用完备的条件判断语句。VHDL 中，当使用 CASE 语句的时候，要使用"When Others"建立默认状态。使用"IF...THEN...ELSE"语句的时候，要用在"ELSE"指定默认状态；Verilog 中，使用"case"语句的时候要用"default"建立默认状态，使用"if...else"语句也要力求完备。

另外提一个技巧：大多数综合器都支持 Verilog 编码状态机的完备状态属性－－"full case"。这个属性用于指定将状态机综合成完备的状态，如 Synplicity 的综合工具 Synplify/Synplify Pro， Amplify，和 Synopys 的综合工具 FPGA Compiler II 等.

Synplicity 综合工具还有一个关于状态机的综合属性，叫"// synthesis parallel_case"其功能是检查所有的状态是"并行的"（parallel），也就是说在同一时间只有一个状态能够成立。

2.14　Verilog 的 FSM 设计技巧

下面是使用 Verilog 语言设计 FSM 的一些具体技巧。

- 状态机的定义可以用 parameter 定义，但是不推荐使用`define 宏定义的方式，因为'define 宏定义在编译时自动替换整个设计中所定义的宏，而 parameter 仅仅定义模块内部的参数，定义的参数不会与模块外的其他状态机混淆。
- 设计时序 always 模块时一定要使用"=>"非阻塞赋值方式。采用非阻塞赋值方式消除了很多竞争冒险的隐患。
- 组合逻辑模块，也就是前面说的两段式 FSM 描述方法中将判断"nextstate"状态转移条件的组合逻辑单独提炼成一个模块，在该模块的描述中要注意下面一些问题。该模块可以用 always 加电平敏感的组合逻辑实现，一般采用" = "阻塞赋值方式，虽然常常定义为"reg"型，但是并非同步的寄存器。其敏感表为所有用以控制状态转移的转移条件，和输入变量。一般在该组合模块中，首先有一个默认的 next state 的描述，然后真实的状态转移由内部的 case 或者 if...else 条件判断确定。推荐的方式是用默认的 idle 状态或者其他的安全、等待状态或者就停留在当前状态。
- FSM 的输出逻辑的生成。输出逻辑的编码有两种形式，第一种方法是，将这部分逻辑混合入描述状态转移的组合逻辑中；第二种方法是，将这部分逻辑独立出来，单独写一个部分。如果混合写的话，最好将输出逻辑写成一个 “task”

任务，该 task 可以被每个状态转移的 case 语句调用，另外要注意，在 always 后面要写一个默认的 output 赋值，道理和写一个默认的状态转移一样。单独写一个逻辑块，程序比较清晰，并且可以避免歧意代码生成 Latches。task 的命名和功能参考"图 5output_task 所示。"这种命名和描述的方法一方面可以节省一部分资源（相当于复用了 task），并且在输出有所改动的时候非常方便。

- Mealy 状态机，输出逻辑可以用"？语句"描述，或者使用 case 语句判断转移条件与输入信号即可。
- 所有的 output 最好用 register 打一下，以获得更好的时序环境和状态的稳定。可以用一个或者多个时序 always 模块完成这个功能。

补充两点注意事项：

- 将 FSM 所有的逻辑用一个状态机描述有如下缺点：时序约束、更改、调试等问题。而且不能很好的表示 MealyFSM 的输出，容易写出 Latches，容易出错。
- 不是所有的 FSM 都必须设计成为 full case 和 parallel case，有时这样做会影响状态机的功能，并会比用二进制编码或者 gray 码耗费更多的资源。

2.15 CPLD 的原理与设计方法

前面的整个篇幅都是在围绕着可编程逻辑器件的主体 FPGA，可编程逻辑器件还有另外一个重要组成部分 CPLD。相比之下 CPLD 的复杂度和设计难度都较为简单，所以指导原则对 CPLD 的设计方法并未特别强调，为了保证论述的完整性，仅仅在本章加以说明。

CPDL 是 Complex Programmable Logic Device 的缩写，即复杂的可编程逻辑器件，不同的器件商对这种器件加以不同的商品名称，如 XCPLD、CPLD、ECPLD 等，其实它们的本质与设计方法是一致的，为了便于论述，我们统称为 CPLD。

- CPLD 和 FPGA 在结构与工艺上的区别

CPLD 在结构和工艺上都与 FPGA 有很大的区别，FPGA 一般都是 SRAM 工艺的，如 Xilinx、Altera、 Lattice 的 FPGA 系列器件，它们都是基于查找表结构的。而 CPLD，一般都是基于乘积项结构的，如 Lattice 的 LC4000、ispLSI5000、ispLSI2000（EECMOS 工艺）系列器件，Altera 的 MAX7000、MAX3000（EEPROM 工艺）系列器件，Xilinx 的 XC9500（Flash 工艺）系列器件等，它们都是基于乘积项的 CPLD。

我们以 Lattice LC4000 为例，观察一下 CPLD 的总体结构，其他厂商的 CPLD 结构与之非常相似，LC4000 的结构示意图如图 2－16 所示。

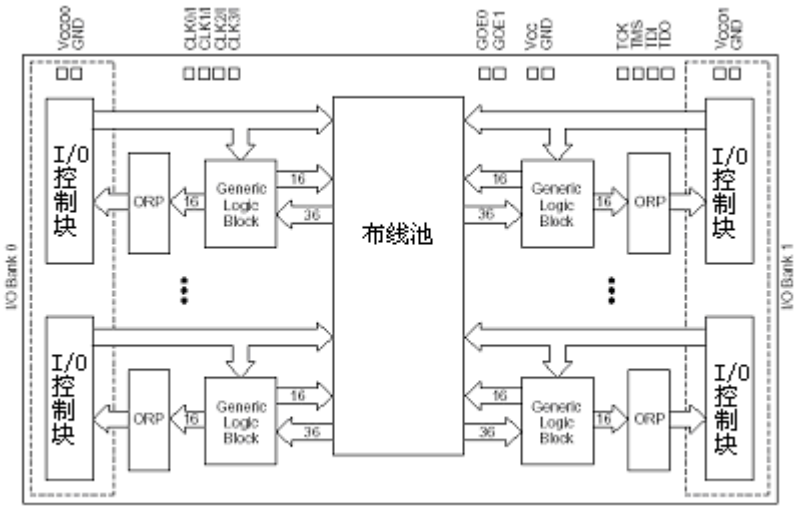


图1-59　Lattice LC4000 CPLD 的结构示意图

这种 CPLD 可分为三块结构：逻辑部分（GLB/LAB），布线部分（GRP/PIA），I/O 口部分。逻辑部分是器件的核心部分，其基本单元是宏单元（Macrocell），图中 GLB 是多个宏单元的集合，对于其中宏单元的数量，不同厂商的器件有不同的规定，如表 2－1 所示。

表1－12.　不同厂商器件的每个 CLB 内 Micro Cell 的数量

	Altera		Lattice		Xilinx	
系列	Max7000	MAX3000	IspLSI2000	IspLSI5000	LC4000	XC9500
MC 数量/GLB	16	16	4	32	16	18

由于每个厂家的 GLB 大小不一样，所以从布线池到 GLB 的信号数也不一样，具体数据如表 2－2 所示。

表1－13.　不同厂商器件的每个 CLB 中的信号数目

	Altera		Lattice			Xilinx
系列	Max7000	MAX3000	IspLSI2000	IspLSI5000	LC4000	XC9500
GLB 输入信号数	36	36	16+2	68	36	54

图中上面的信号 CLK0/I、CLK1/I、CLK2/I、CLK3/I、GOE0、GOE1 是全局时钟和全局输出使能信号，这几个信号有专用联线与 CPLD 中的每个宏单元相连，使信号到每个宏单元的延时相同并且延时与抖动都最小。

GLB 内部，其结构都是由与阵列、或阵列和 D 触发器构成的。继续以 LC4000 器件为例，LC4000 的 CLB 内部结构如图 2－17 所示。

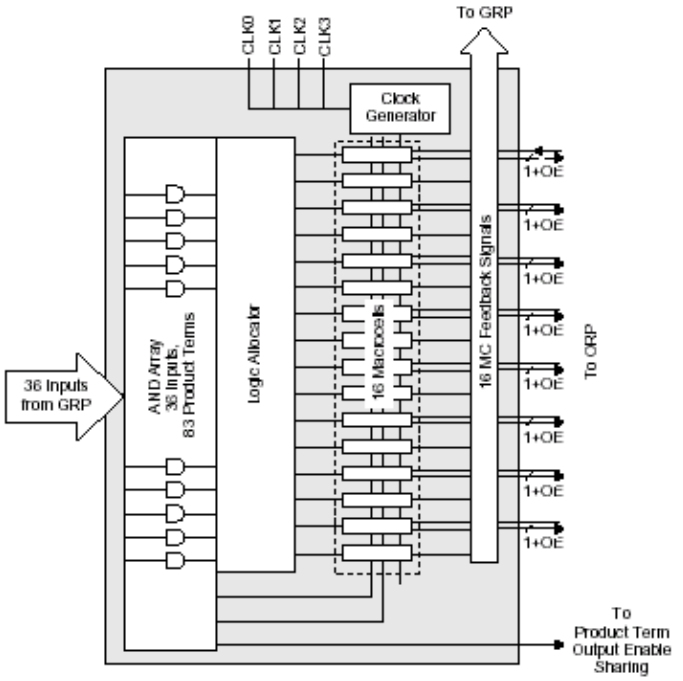


图1-60　LC4000 的 CLB 内部结构示意图

所谓乘积项即与阵列的输出，其数量标志了 CPLD 容量，对 CPLD 的性能也有一定的影响，不同厂商的 CPLD 定制的乘积项数目不同。图中，左侧的乘积项阵列，实际上就是一个与或阵列，每一个交叉点都是一个可编程熔丝，如果导通就是实现“与“逻辑；后面的逻辑分配器是一个“或”阵列。两者配合工作，完成复杂的组合逻辑功能。图中，右侧是可编程的 D 触发器，它的时钟、复位/置位、时钟使能都可编程选择。其触发源可以使用全局的信号，也可以使用内部逻辑（乘积项阵列）产生的信号，如果不需要触发器，也可以将此触发器旁路，信号直接输出到 I/O 端口。

事实上，目前所有厂商的 CPLD 都是这种主流结构，只不过在内部结构上有细微的区别，可以说在选择哪一家器件时是一个相对比较困难的问题，需要将设计需要、性能指标和成本等因素综合考虑。目前业界的主流 CPLD 产品是 Lattice 的 LC4000 系列和 Altera 的 MAX3000 系列。这两个器件族各有特点：Lattice 公司是一家 CPLD 主导公司，LC4000 系列器件是其精心推出的产品，采用 0.18um 工艺，并使用其擅长的 EECMOS 技术，集成了其早期 CPLD 产品的优良性能，可观的评价是一个比较成功的产品；对于 Altera，目前其主流产品是 FPGA，花费在 CPLD 产品的精力相对少些，但其 MAX3000 系列，采用 0.30um 工艺，继承了 MAX7000 的优异性能，而在成本上比 MAX7000 有更大的优势；Xilinx 由于公司战略策略的转移，目前正逐步淡出 CPLD 等低端可编程器件领域。具体选择器件时，需要综合考虑器件应用场合、宏单元数量、I/O 端口数量、器件封装、器件性能指标、成本等众多因素。

- **乘积项结构的 CPLD 逻辑实现原理**

首先我们用一个简单的电路为例，具体说明 CPLD 是如何利用以上结构实现逻辑的，电路如图 2－18 所示：

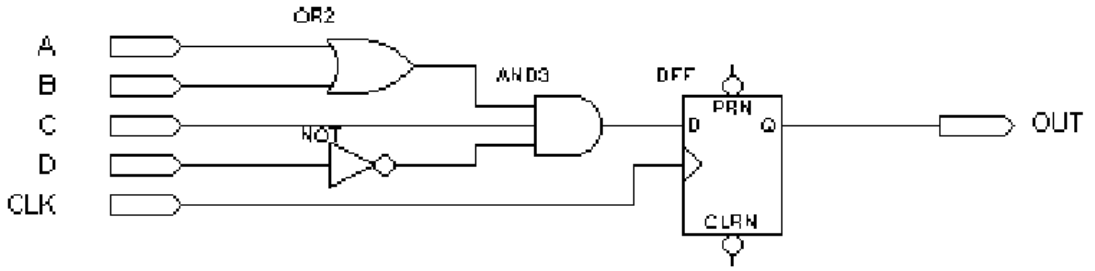


图1-61　某逻辑电路原理图

假设组合逻辑的输出（三输入与门的输出）为 F，则 F=(A+B)*C*(!D)=A*C*(!D)+B*C*(!D)，CPLD 将用如图 2－19 所示的方法实现该组合逻辑 F。

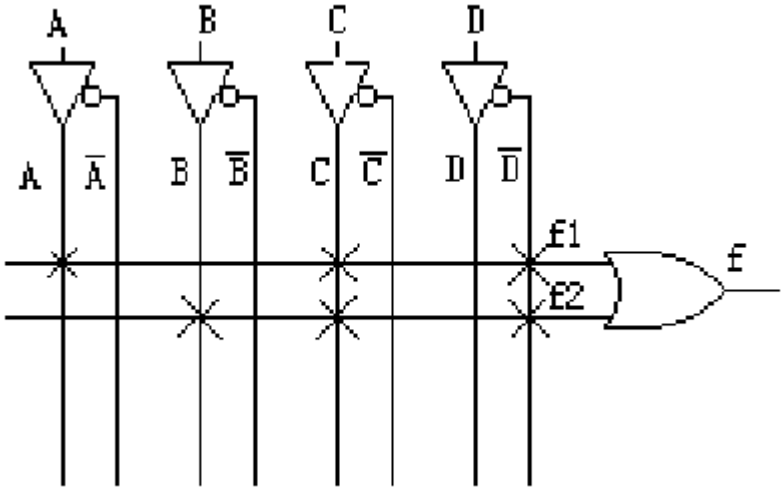


图1-62　组合逻辑“F”的 CPLD 实现方法

图中清晰的看到 CPLD 是如何利用与或阵列来实现组合逻辑的。而对于电路中 D 触发器的实现就更为简单，直接利用宏单元的可编程 D 触发器来实现。时钟信号 CLK 由 I/O 交输入后进入芯片的全局时钟专用通道，直接连接到可编程触发器的时钟端。可编程触发器的输出与 I/O 脚相连，把结果输出到芯片管脚。这样就在 CPLD 器件内部完成了图中的电路功能。

本例仅仅实现了一个非常简单逻辑功能，只需要一个宏单元就可以完成。但是对于一个复杂的电路，一个宏单元是不可能实现的，这就需要更多乘积项，也可能需要将更多的宏单元相连，这样 CPLD 就可以实现更复杂的逻辑功能。

- CPLD 和 FPGA 的区别与联系

最后作为总结，我们用表格的形式对比一下 CPLD 和 FPGA 的区别与联系，以加深读者对 CPLD 特点的整体把握。

表1－14.　　CPLD 与 FPGA 区别与联系对比表

项目	FPGA	CPLD	备注
工艺	SRAM	乘积项，物理上有多种实现方法：EPROM、EEPROM、Flash、反熔丝等	
触发器数量	多	少	所以 FPGA 更适合实现时序逻辑，CPLD 多用于实现组合逻辑。
Pin to Pin 延时	不可预测	固定	所以对 FPGA 而言时序约束和仿真非常重要。
规模与逻辑复杂度	规模大，逻辑复杂度高，新型器件高达千万门级	规模小，逻辑复杂度低	FPGA 用以实现复杂设计，CPLD 用以实现简单设计
成本与价格	成本高，价格高	成本低，价格低	CPLD 用于实现低成本设计
编程与配置	方法多，一般为 RAM 型	一般为 ROM 型。由两种编程方式，一种是通过编程器烧写 ROM，另一种较方便的方式是通过 ISP 模式。	FPGA 如果不外挂存储器（EPROM 等），掉电将丢失逻辑功能。
保密性	差	好	除非工业极和一些高端器件，FPGA 不容易实现加密。
互联结构，连线资源	分布式，丰富的连线资源。	集总式，相对连线资源有限	
适用的设计类型	复杂的时序功能	简单的逻辑功能	

至此，关于设计准则的讨论暂时告一段落。限于篇幅，这两章中我们只能对设计中一般性原则和规律加以分析与介绍，未能深入讨论，其实这些设计准则也都不是绝对的，随着微电子制造与 EDA 技术的发展，特别是制造工艺与 EDA 综合实现技术的突飞猛进，很多设计准则在发生着质的变化。归根结底，设计准则和规律是前人经验的总结。所以要想很好的理解可编程逻辑的设计方法，熟练掌握其设计规律，必须在日后自己的实践中勤于思考和总结。