华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《FPGA 应用开发实战技巧精粹》

作者：华清远见

专业始于专注 卓识源于远见

第 4 章 逻辑电路设计技巧

专业始于专注 卓识源于远见

4.1 FPGA 设计的基本技巧

本节首先介绍 FPGA 设计的两种设计方法（Top-Down 和 Bottom-Up），然后分别介绍了利用 VHDL 和 Verilog HDL 两种语言进行 FPGA 设计的技巧，最后介绍了状态机设计的思想和技巧。

4.1.1 Top-Down 方式的设计技巧

❖ 技巧内容

本节将介绍 FPGA 的设计方法之一——Top-Down 的设计方法。

❖ 技巧详解

FPGA 传统的设计手段是采用原理图输入的方式进行的。通过调用 FPGA/EPLD 厂商所提供的相应物理元件库，在电路原理图中绘制所设计的系统，然后通过网表转换产生某一特定 FPGA/EPLD 厂商布局布线器所需网表，通过布局布线，完成设计。原理图绘制完成后可采用门级仿真器进行功能验证。

然而，工程师的最初设计思想不是一开始就考虑采用某一 FPGA/EPLD 厂商的某一特定型号器件，而是从功能描述开始的。设计工程师首先要考虑规划出能完成某一具体功能、满足自己产品系统设计要求的某一功能模块，利用某种方式（如 HDL 硬件描述语言）把功能描述出来，通过功能仿真（HDL 仿真器）以验证设计思路的正确性。当所设计功能满足需要时，再考虑以何种方式（即逻辑综合过程）完成所需要的设计，并能直接使用功能定义的描述。实际上这就是自顶而下设计方法。

与传统电原理图输入设计方法相比，Top-Down 设计方法具体有以下优点。

- 完全符合设计人员的设计思路，从功能描述开始，到物理实现的完成。
- 功能设计可完全独立于物理实现，在采用传统的电原理输入方法时，FPGA/EPLD 器件的采用受到器件库的制约。由于不同厂商 FPGA/EPLD 的结构完全不同，甚至同一厂商不同系列的产品也存在结构上的差别。因此，在设计一开始，工程师的设计思路就受到最终所采用器件的约束，大大限制了设计师的思路和器件选择的灵活性。而采用 Top-Down 设计方法，功能输入采用国际标准的 HDL 输入方法，HDL 可不含有任何器件的物理信息，因此工程师可以有更多的空间去集中精力进行功能描述，设计师可以在设计过程的最后阶段任意选择或更改物理器件。
- 设计可再利用，设计结果完全可以用一种知识产权（IP-Intellectual Property）的方式作为设计师或设计单位的设计成果，应用于不同的产品设计中，做到成果的再利用。
- 易于设计的更改，设计工程师可在极短的时间内修改。
- 设计、处理大规模、复杂电路，目前的 FPGA/EPLD 器件正向高集成度、深亚微米工艺发展。为设计系统的小型化、低功耗、高可靠性等提供了集成的手段。设计低于 10000 门左右的电路，Top-Down 设计方法具有很大的帮助，而设计更大规模的电路，Top-Down 设计方法则是必不可少的手段。
- 设计周期缩短，生产率大大提高，产品上市时间提前，性能明显提高，产品竞争力加强。据统计，采用 Top-Down 设计方法的生产率可达到传统设计方法的 2～4 倍。

Top-Down 设计流程如图 4.1 所示，其核心是采用 HDL 语言进行功能描述，由逻辑综合（Logic Synthesis）把行为（功能）描述转换成某一特定 FPGA/EPLD 的工艺网表，送到厂商的布局布线器完成物理实现。在设计过程的每一个环节，仿真器的功能验证和门级仿真技术保证设计功能和时序的正确性。

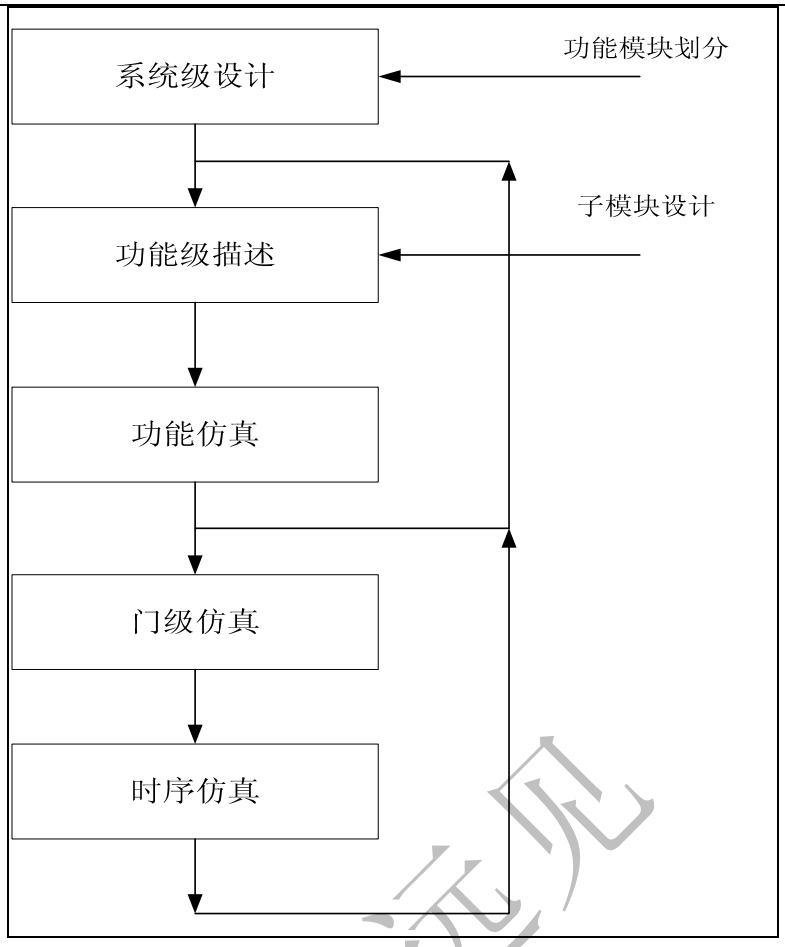

图 4.1　Top-Down 的设计流程

4.1.2　Bottom-Up 方式的设计技巧

❖ 技巧内容

本节将介绍 FPGA 的设计方法之一——Bottom-Up 的设计方法。

❖ 技巧详解

自下至上的硬件电路设计方法的主要步骤是：根据系统对硬件的要求，详细编制技术规格书，并画出系统控制流图；然后根据技术规格书和系统控制流图，对系统的功能进行细化，合理地划分功能模块，并画出系统的功能框图；接着就是进行各功能模块的细化和电路设计；各功能模块电路设计。调试完成后，将各功能模块的硬件电路连接起来再进行系统的调试，最后完成整个系统的硬件设计。

自下至上的设计方法在各功能模块的电路设计中的体现大概最能说明问题。下面以一个六进制计数器设计为例作一说明。

要设计一个六进制计数器，其方案是多种多样的，但是摆在设计者面前的一个首要问题是，如何选择现有的逻辑元、器件构成六进制计数器。那么，设计六进制计数器将首先从选择逻辑元、器件开始。

第一步，选择逻辑元、器件。由数字电路的基本知识可知，可以用与非门、或非门、D 触发器、JK 触发器等基本逻辑元、器件来构成一个计数器。

第二步，进行电路设计。假设六进制计数器采用约翰逊计数器。3 个触发器连接应该产生 8 个状态，现在只是用 6 个状态，将其中的 010 和 101 两种状态禁止掉。这样就产生一个 6 个状态的计数器。

第三步，逻辑元、器件连接。根据六进制状态的转移，对触发器和触发器之间，以及门电路和触发器之间进行连接。

这样就完成了对一个六进制计数器自下至上的设计。
与六进制计数器模块设计一样，系统的其他模块也按此方法进行设计。在所有硬件模块设计完成以后，再将各模块连接起来，进行调试，如有问题则进行局部修改，直至整个系统调试完毕为止。
从上述设计过程可以看到，系统硬件的设计是从选择具体元、器件开始的，并用这些元、器件进行逻辑电路设计，完成系统整个独立功能模块设计。然后再将各功能模块连接起来，完成整个系统的硬件设计。上述过程从最底层开始设计，直至到最高层设计完毕，故将这种设计方法称为自下至上的设计方法。

4.1.3 VHDL 设计 FPGA 的技巧

❖ 技巧内容

本节将介绍使用 VHDL 设计 FPGA 的一些常用技巧。

❖ 技巧详解

VHDL 设计是行为级的设计，所带来的问题是设计者的设计思考与实际电路结构是相脱节的。设计者主要是根据 VHDL 的语法规则，对系统目标的逻辑行为进行描述，然后通过综合工具进行电路结构的综合、编译、优化，通过仿真工具进行逻辑功能仿真和系统时延的仿真。实际设计过程中，由于每个设计工程师对语言规则、对电路行为的理解程度不同，每个人的编程风格不同，往往同样的系统功能，描述的方式是不一样的，综合出来的电路结构更是大相径庭。因此，即使最后综合出的电路都能实现相同的逻辑功能，其电路的复杂程度和时延特性都会有很大的差别，甚至某些臃肿的电路还会产生难以预料的问题。从这些问题出发，很有必要深入讨论在 VHDL 设计中电路结构和优化电路设计的问题。用 VHDL 进行设计，最终综合出的电路的复杂程度，除取决于设计要求实现的功能的难度外，还受设计工程师对电路的描述方法和对设计的规划水平的影响。
下面给出一些使用 VHDL 设计 FPGA 的一些实用的技巧。

- 寄存器代替锁存器：最常见的使电路复杂化的原因之一是在设计中存在许多本不必要的类似 LATCH 的结构。而且由于这些结构通常都由大量的触发器组成，不仅使电路更复杂、工作速度降低，而且由于时序配合的原因会导致不可预料的结果。既然锁存器存在不稳定，那么有必要找到一个好的替代，那就是寄存器。
- 使用括号描述想要的结构：如（1）Out1<=I1+I2+I3+I4;（2）Out2<=(I1+I2)+(I3+I4)。从结构上看，第一种描述中没有使用括号，电路结构为 3 层，加法一级一级地进行。它的特点是，4 个输入 I1、I2、I3、I4 到达加法器的路径不相等。第二种描述中的使用了括号，电路结构分为 2 层，它的特点是 4 个输入信号到达加法器的路径是相等的。这里并不能简单的评论两种结构的优劣，必须根据实际应用情况分析。
- 并行结构：在 VHDL 语言描述中，进程中的语句是顺序执行的。如果要设计一个并行的电路，不要使用顺序语句，要使用并行语句。
- 资源共享：如果有可能，尽量使资源共享。一块 FPGA 芯片中的资源是有限的。如果不有效利用仅有的资源，将会导致资源不够，系统冗余过多，稳定性也会变差。
- integer 和 std_logic_vector：在 VHDL 中，信号使用限制了位数的 std 在 VHDL 中，信号使用限制了位数的 std_logic_vector（标准逻辑向量）型，不能使用 integer（整型）。如果使用 integer，必须在后面加上约束条件，例如“a:integer range 0 to 7;”。如果使用了 integer 而没有对其进行位数的约束，那么在 32 位的 PC 机中，综合后将是一个 32bit 的信号，会造成很大的浪费。
- 不同的状态机描述：状态机的选型按状态机的输出方式分类，有限状态机可分为 Mealy 型和 Moore 型。从输出时序上看，状态机的输出是当前状态和所有输入信号的函数，它的输出是在输入变化后立即发生的，不依赖时钟的同步。Moore 型状态机输出则仅为当前状态的函数，状态机的输入发生变化还必须与状态机的时钟同步。由于 Mealy 状态机的输出不与时钟同步，所以当在状态译码比较复杂的时候，很容易在输出端产生大量的毛刺，这种情况是无法避免的。在一些特定的系统中，这些毛刺可能造成不可预料的结果。但是，由于输入变化可能出现在时钟周期内的任何时刻，这就使得 Mealy 状态机对输入的响应可以比 Moore 状态机对输入的响应要早一个时钟周期。Moore 状态机的输出与时钟

同步，可以在一定程度上剔除抖动。从稳定性的角度来讲，建议使用 Moore 状态机以提高系统的稳定性。在实际工程中，电路有具体的设计要求，可能适应于 Moore 或 Mealy 状态机，也可能两种都可以实现设计，所以设计者必须根据实际情况和经验综合决定采用哪种状态机。

4.1.4 Verilog HDL 设计 FPGA 的技巧

❖ 技巧内容

本节将介绍使用 Verilog HDL 设计 FPGA 的一些常用技巧。

❖ 技巧详解

随着电子技术的发展，芯片的复杂程度越来越高，人们对数万门乃至数百万门电路设计的需求也越来越多，采用硬件描述语言 HDL 的设计方式应运而生。而在利用硬件描述语言 HDL 进行 FPGA 设计时，高效的代码有利于得到较为理想的电路。

Verilog HDL 因其提供了非常精炼和易读的语法而受到广大硬件工程师的青睐。本文讨论了在进行 FPGA 设计中，如何通过编写 VerilogHDL 代码达到预期的设计要求。

- 减少关键路径上的组合逻辑单元数：在 FPGA 中每条关键路径上的逻辑单元都会增加一定的时延。因此为了保证关键路径能满足时序约束，设计时必须考虑在关键路径上如何减少逻辑单元的使用。下面的例子说明了如何减少关键路径上的逻辑单元个数。

假设“critical”所经的路径是一条关键路径，在下面的例子中“critical”经过了两个逻辑单元，如图 4.2 所示。

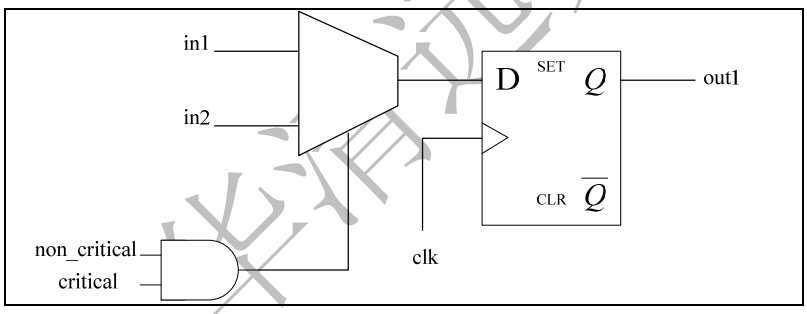

图 4.2 关键路径经过两个逻辑单元

```
reg   out_reg;
always @ (in1 or in2 or critical or non_critical)
    begin
      If (critical&non_critical)
        out_reg = in1;
      Else
        Out_reg=in2;
      End
always @ (posedge clk)
    begin
     out = out_reg;
    end
endmodule
```

为了减少“critical”所经过的逻辑单元数，对程序进行如下的修改，使得“critical”经过的逻辑单元数为 1，如图 4.3 所示。

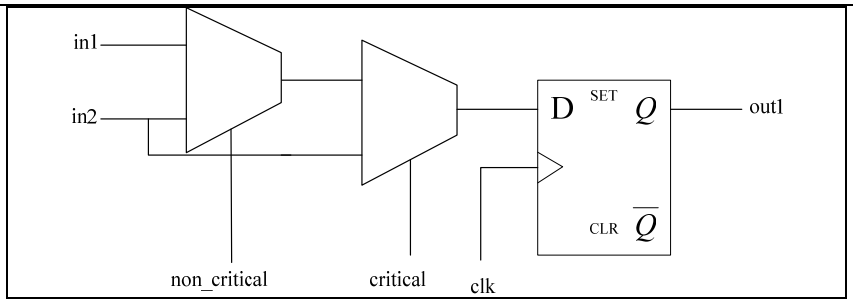

图 4.3　关键路径经过 1 个逻辑单元

```
reg   out_reg;
reg   out_tmp;
always @ (in1 or in2 or non_critical)
    begin
       If (non_critical)
          out_tmp = in1;
       Else
          out_tmp =in2;
       End
always @ (in2 or critical or out_tmp)
    begin
       If (non_critical)
          out_reg = out_tmp;
       Else
          out_reg =in2;
       End
always @ (posedge clk)
    begin
      out = out_reg;
    end
```

- 资源共享：资源共享能减少宏单元的使用数量，因此在设计时同样可以通过编写合适的程序来达到资源共享的目的。在下面的例子中，是 1 个二选一选择器和 2 个加法器，如图 4.4 所示。

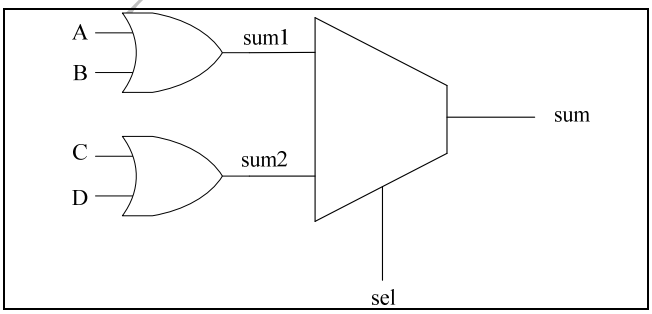

图 4.4　资源共享前

```
reg   sum;
reg   sum1;
reg   sum2;
always @ (a or b)
    begin
        sum1= a + b;
    End
always @ (c or d)
```

```
    begin
        sum2= c + d;
    End
always @ (sel or sum1 or sum2)
    begin
      If (sel)
        sum = sum1;
      Else
        sum =sum2;
     End
```

为了加大资源的利用率，重新书写代码以达到资源共享目的，如图 4.5 所示。

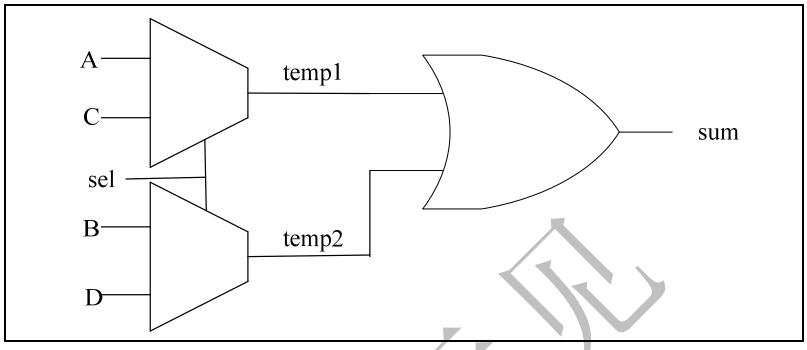

图 4.5　资源共享后

```
reg   temp1;
reg   temp2;
always @ (sel or a or c)
    begin
      If (sel)
        temp1 =a;
      Else
        temp1 =c;
    End
always @ (sel or b or d)
    begin
      If (sel)
        temp2 =b;
      Else
        temp2 =d;
    End
assign sum = temp1 + temp2;
```

- 为优化逻辑而进行的复制：设计人员在利用综合工具对可编程逻辑器件进行综合时，都会面临一个问题，即综合工具并不能对复杂的设计实现最佳的布局、布线结果。大多数综合工具都有一个扇出控制。因此，为了优化设计，建议在设计代码中产生复制逻辑，许多综合工具都可以优化复制，但必须告诉综合工具保持其重复逻辑。
- 复制组合逻辑：如果一个扇出大于 1 的组合逻辑不能在 CLB 内部实现，这时需要对组合逻辑进行复制。下面给出组合逻辑复制的例子，如图 4.6 所示。

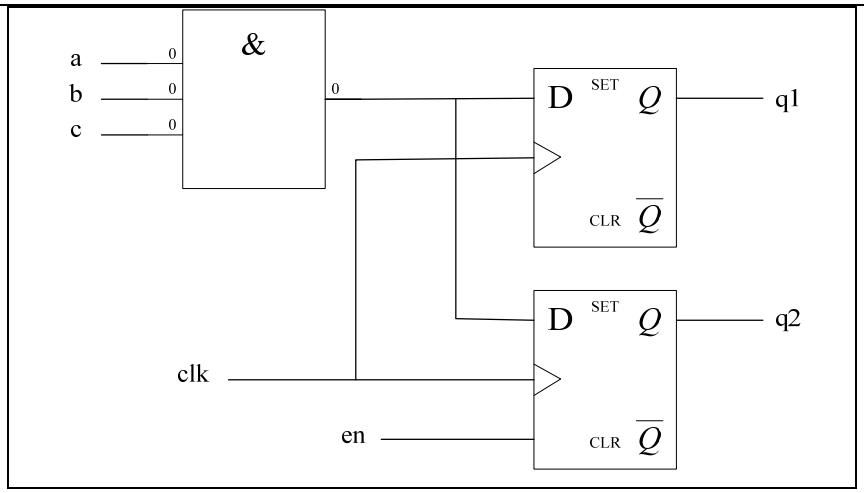

图 4.6　组合逻辑复制

```
wire   temp;
assign temp=a&b&c;
always @ (posedge clk)
    begin
       If (en)
          q2=temp;
    End
always @ (posedge clk)
    begin
         q1=temp;
    End
```

可以重新书写代码达到组合逻辑复制的目的，如图 4.7 所示。

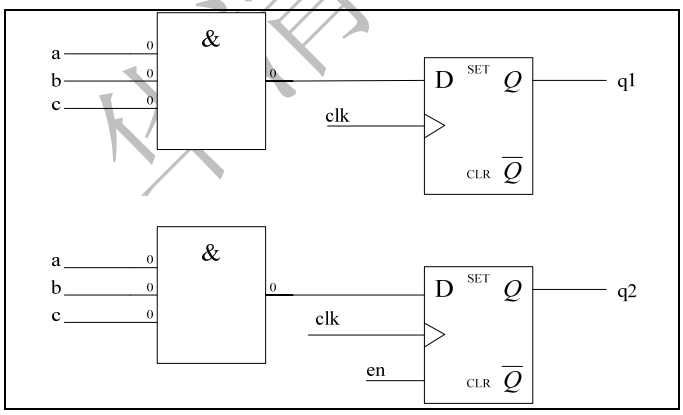

图 4.7　复制组合逻辑的效果

```
wire   temp1;
wire   temp2;
assign temp1=a&b&c;
assign temp2=a&b&c;
always @ (posedge clk)
    begin
       If (en)
          q2=temp2;
    End
always @ (posedge clk)
    begin
         q1=temp1;
```

```
    End
```

- 复制触发器：为了优化设计，可对大扇出信号的触发器进行复制。因为大扇出信号能减缓布线速度，并增加布线的难度。可以通过复制触发器解决两个问题，减小扇出，缩短布线延迟；复制后每个触发器可以驱动芯片的不同区域，有利于布线。下面给出复制触发器的例子，如图 4.8 所示。

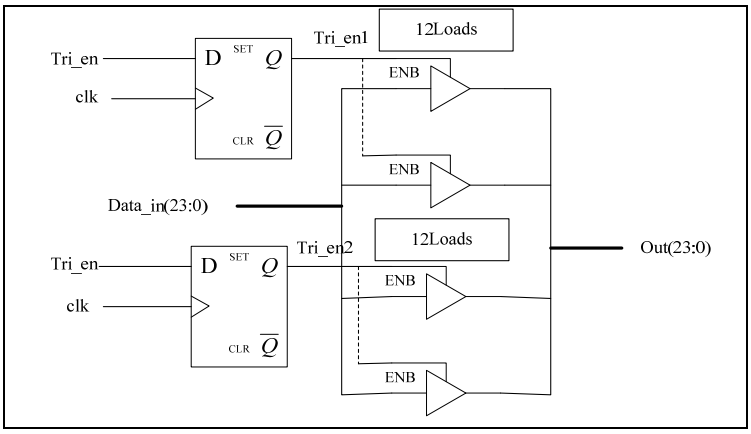

图 4.8　复制触发器前

```
reg  tri_end;
always @ (posedge clk)
    begin
        tri_end = tri_en;
    End
assign out=( tri_end)? Data_in : 'bz;
```

修改设计代码，可将触发器进行复制，降低 tri_end 的扇出，如图 4.9 所示。

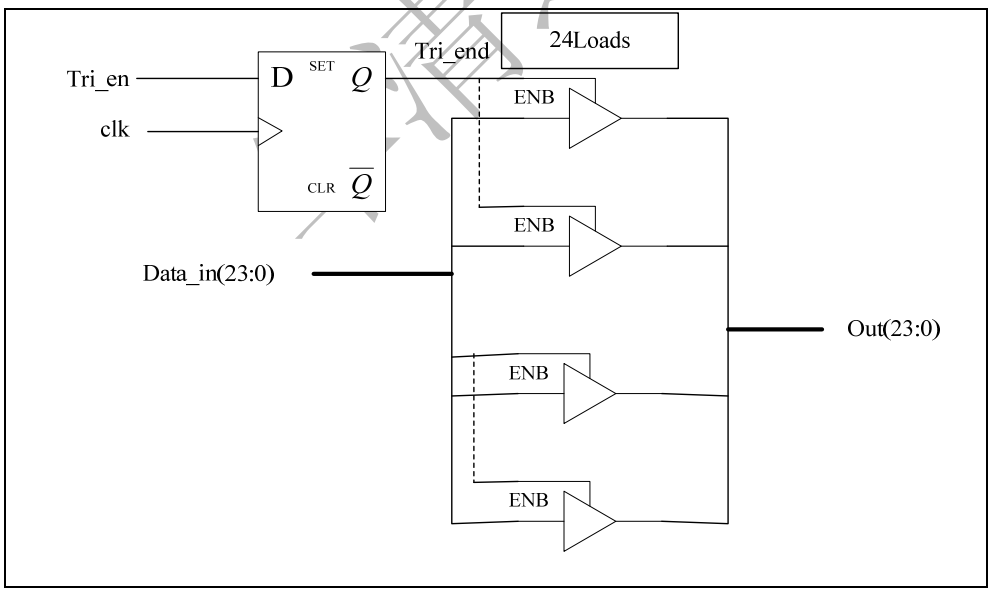

图 4.9　复制触发器后

```
reg  tri_en1, tri_en2;
always @ (posedge clk)
    begin
        tri_en1 = tri_en;
    End
always @ (posedge clk)
    begin
        tri_en2 = tri_en;
```

```
    End
assign out[11:0]=( tri_en1)? Data_in[11:0] : 'bz;
assign out23:12]=( tri_en2)? Data_in[23:12] : 'bz;
```

- 阶层化设计：随着人们设计的电路的逻辑越来越复杂，采用传统的平坦式设计来设计电路，已经不能满足设计人员的要求。因此出现了阶层化设计，即将设计任务分解到可控制模块中的方法形成阶层结构。采用阶层化设计有利设计的保存、继承。对每一个功能模块，设计人员可以建立通用的功能模块库，既便于与其他功能模块接口，又可以再次使用，避免重复劳动。在将两个设计划分为几个模块时，最好以寄存器作为划分模块的边界。这样有利于综合器综合出速度更快的电路。下面这个例子说明了如何寄存模块的输出，如图 4.10 所示。

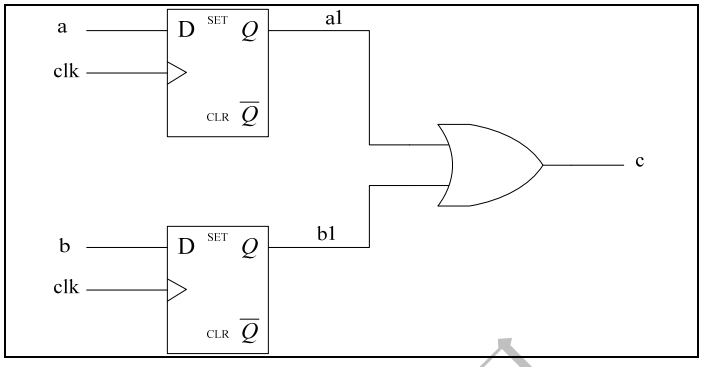

图 4.10　两个寄存器模块输出

```
reg  a1,b1;
always @ (posedge clk)
    begin
        a1 = a;
b1 = b;
    End
assign c=a1+b1;
```

将两个寄存器模块分为单一寄存器模块，如图 4.11 所示。

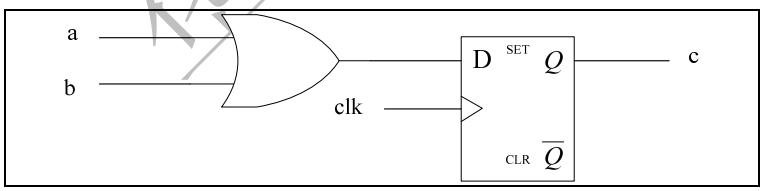

图 4.11　1 个寄存器模块输出

```
wire  sum;
assign sum=a+b;
always @ (posedge clk)
    begin
        c = sum;
    End
```

4.1.5　状态机设计的技巧

❖ 技巧内容

本节将讲解状态机设计的一些技巧。

❖ 技巧详解

状态机是逻辑设计中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。下面是一些适用于 Verilog 和 VHDL 等 HDL 语言编写状态机的一般性原则。

- 选择状态机的编码方式：Binary、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 更多地提供组合逻辑资源，而 FPGA 更多地提供触发器资源，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。
- 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、判断等写到一个模块中；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。

其中较好的方式是后者，因为 FSM 和其他设计一样，最好使用同步时序方式设计，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中实现。这样做的好处不仅是便于阅读、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。另一方面状态机也应该有一个默认状态，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 和 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。

另外一个技巧是：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。

- 指定默认输出值：对状态机的所有输出变量指定一个默认的输出值，这样做的好处是能够防止无意生成的 Latch。另外所有的输出最好用寄存器打一拍，以获得更好的时序环境和状态的稳定。
- 状态机输出逻辑复用：如果在状态机中有多个状态都会执行某项操作，则在状态机外部定义这个操作的具体内容，然后在状态机中仅仅调用这个操作的最终输出值即可。

4.2 数字系统设计技巧

4.2.1 同步电路设计技巧

❖ 技巧内容

同步电路设计是 FPGA/CPLD 设计的重要原则之一。本节讲解为什么在 PLD 设计中采用同步电路设计的基础上重点讲述同步电路设计的技巧。

❖ 技巧详解

同步电路的特点如下所示。

- 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的。
- 同步时序电路可以很好地避免毛刺。布局布线后仿真和高速逻辑分析仪采样实际工作信号皆无毛刺。

- 利于器件的移植。
- 有利于静态时序分析、验证设计时序性能。

同步电路设计的基本原则是使用时钟沿触发所有的操作。如果所有寄存器的时序要求（Setup、Hold 时间等指标）都能够满足，则同步电路设计与异步电路设计相比，在不同 PVT 条件下能获得更佳的系统稳定性和可靠性。

同步电路设计中，稳定可靠的数据采样必须遵从以下两个基本原则。

- 在有效时钟沿到达前，数据输入至少已经稳定了采样寄存器的 Setup 时间，这条原则简称 Setup 时间原则。
- 在有效时钟沿到达后，数据输入至少还将稳定保持采样寄存器的 Hold 时间，这条原则简称 Hold 时间原则。

同步电路设计中，用户需要注意以下几个事项。

- 异步时钟域的数据转换。
- 组合逻辑电路的设计方法。
- 同步时序电路的时钟设计。
- 同步时序电路的延迟。同步电路设计中电路延迟最常用的设计方法是用分频或倍频的时钟或者同步计数器完成所需延迟。换言之，同步电路设计的延时被当作 1 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 1 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 触发器打一拍，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。
- Verilog 定义为 reg 型，不一定综合成寄存器。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。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触发器，这个电路是一个纯组合逻辑设计。

```
module reg_cmb(Reset, CS, Din, Addr, Dout);
input  Reset;
input  CS;
input  [7:0]  Din;
input  [1:0]  Addr;
output [1:0]  Dout;
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
```

4.2.2 异步时钟域数据同步的技巧

❖ 技巧内容

数据接口的同步是 FPGA/CPLD 设计的一个常见问题，也是一个重点和难点，很多设计不稳定都是源于数据接口的同步有问题。本节将介绍一些常用的数据同步的方法。

❖ 技巧详解

异步时钟域数据同步，也被称为数据接口同步，顾名思义，是指如何在两个时钟不同步的数据域之间可靠地进行数据交换的问题。数据的时钟域不同步主要有以下两种情况。

- 两个域的时钟频率相同，但是相位差不固定，或者相差固定但是不可测，简称同频异相问题。
- 两个时钟频率根本不同，简称异频问题。

为解决上述问题，在电路图设计阶段，一些工程师手工加入 buffer 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、保持时间要求。还有一些工程师为了有稳定的采样，生成了很多相差 90° 的时钟信号，用正沿打一下数据或用负沿打一下数据，用以调整数据的采样位置。这两种做法都十分不可取，因为一旦芯片更新换代或者移植到其他芯片组的芯片上，采样实现必须重新设计。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，或者有可能变动的条件下，完成数据同步的方法有以下几种。

对于数据的延迟不可测或变动，就需要建立同步机制，可以用一个同步使能或同步指示信号。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，但是不推荐这种做法。

- 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中非常普遍。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，完成数据的同步，并发现数据的“头”的方法如下。数据的同步方法完全可以采用上面的方法，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，则只有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，作用是有效防止亚稳态（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，这样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以下两点。

- 提高设计的工作频率，满足接口数据同步要求。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，满足接口数据同步要求。
- 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。

4.2.3 亚稳态

❖ 技巧内容

本节将介绍在 FPGA 设计中亚稳态的产生的原因，以及如何在设计中消除亚稳态的技巧。

❖ 技巧详解

在设计中，当用时钟去采样数据时，如果采样的数据不满足 setup 时间和 hold 时间，这样就可能产生亚稳态，此时触发器输出端 Q 在有效时钟沿之后比较长的一段时间内处于不确定的状态。在这段时间内 Q 端产生毛刺并不断振荡，最终固定在某一电压值。此电压值不一定等于原来数据输入端 D 的数值，这段时间称为决断时间（resolution time）。经过决断时间之后，Q 端将稳定到 0 或 1 上，但究竟是 0 还是 1，这是随机的，与输入没有必然的关系。

亚稳态的危害主要体现在破坏系统的稳定性。由于输出在稳定下来之前可能是毛刺、振荡、固定的某一电压值，因此亚稳态将导致逻辑误判，严重情况下输出 0～1 之间的中间电压值还会使下一级产生亚稳态，即导致亚稳态的传播。逻辑误判导致功能性错误，而亚稳态的传播则扩大了故障面。另外，在亚稳态状态下，任何诸如环境噪声、电源干扰等细微扰动都将导致更恶劣的状态不稳定。这时这个系统的传输延迟增大，状态输出错误，在某些情况下甚至会使寄存器在两个有效判定门限（VoL、VoH）之间长时间的振荡。

只要系统中有异步元件，亚稳态就无法避免，因此设计的电路首先要减少亚稳态导致的错误，其次要使系统对产生的错误不敏感。前者要靠同步设计来实现，而后者根据不同的设计应用有不同的处理方法。

使用两级寄存器采样可以有效地减少亚稳态继续传播的概率。但是并不能保证第二级输出的稳态电平就是正确电平。前面说过经过决断时间之后，寄存器输出的电平是一个不确定的稳态值。也就是说这种处理方法并不能排除采样错误的产生。这时就要求所设计的系统对采样错误有一定的容忍度。有些应用本身就对采样错误不敏感，如一帧图像编码、一段语音编码等。而有一些系统对错误采样比较敏感。这类由于亚稳态造成的采样是一些突发的错误，可以采用一些纠错编码手段完成错误的纠正。

4.2.4 系统原则的技巧

❖ 技巧内容

系统原则是FPGA/CPLD设计的重要原则之一。本节讲解在PLD设计中采用系统原则进行电路设计的技巧。

❖ 技巧详解

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，一个单板如何进行模块划分与任务分配，什么样的算法和功能适合放在传统FPGA中实现，什么样的算法和功能适合放在DSP、CPU中实现，或者在使用具有内嵌CPU和DSP资源的FPGA中如何划分软硬件功能，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、面积和速度等问题。

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、可用的设计资源有一个较深刻的认识。

FPGA基本由可编程I/O单元、基本可编程逻辑单元、嵌入式RAM、丰富的布线资源、底层嵌入功能单元和内嵌专用硬核等6部分组成。CPLD的结构相对比较简单，主要由可编程I/O单元、基本逻辑单元、布线池和其他辅助功能模块构成。系统原则要求设计者根据设计类型与资源评估合理地完成器件选型，然后充分发挥所选器件的各个部分的最大性能，对器件整体上有个优化的组合与配置方案。

- 存储器资源的使用：存储器资源的使用原则是根据设计中用到多少RAM或ROM，确定所选器件的嵌入式Block RAM的容量，并合理配置每块RAM的深度和　　宽度。
- 软核的使用：未来FPGA的一个发展趋势是越来越多的FPGA产品将包含DSP或CPU等软处理核，从而FPGA将由传统的硬件设计手段逐步过渡为系统级设计工具。如Altera的一些器件提供了DSP Core、NIOS等软核，Xilinx的一些器件提供了Power PC 450的CPU Core和MicroBlaze RISC处理器Core，Lattice的ECP系列内部也集成了DSP Core模块。这就为系统设计和单板设计提供了新的解决方案，传统的软硬件联合设计方案中软件部分在适当的情况下就可以是用内嵌的DSP、CPU Block的FPGA取代通用DSP和FPGA，从而简化了单板设计难点，节约了单板面积，提高了单板可靠性。
- 串行收发器的使用：很多高端的FPGA内嵌了SERDES以完成高速串行信号的收发。SERDES是SERializer和DESerializer的英文缩写，即串行收发器，它由两个部分构成：发端是串行发送单元SERializer，用高速时钟调制编码数据流；接收端为串行接收单元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，接收单元还有一个时钟数据恢复模块。目前三大FPGA生产商Altera、Xilinx、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，使得高速传输电路的设计变得简便、可靠。
- 其他结构的使用：对于可编程I/O资源，需要根据系统要求合理配置。通常选择I/O的标准有功耗、传输距离、抗干扰性和EMI等。根据设计的速度要求，要合理选择器件的速度等级，并在设计中正确地分频不同速度等级的布线资源与时钟资源。需要注意的是，选择高等级的器件和改善布线资源分配仅仅是提高芯片工作速度的辅助手段，设计速度主要由电路的整体结构、代码的Coding Style等因素决定。合理使用芯片内部的PLL、DLL或DCM资源完成时钟的分频、倍频、移相等操作不仅简化了设计，而且还能有效地提高系统的精度和工作稳定性。

下面给出FPGA系统规划的简化流程，如图4.12所示。

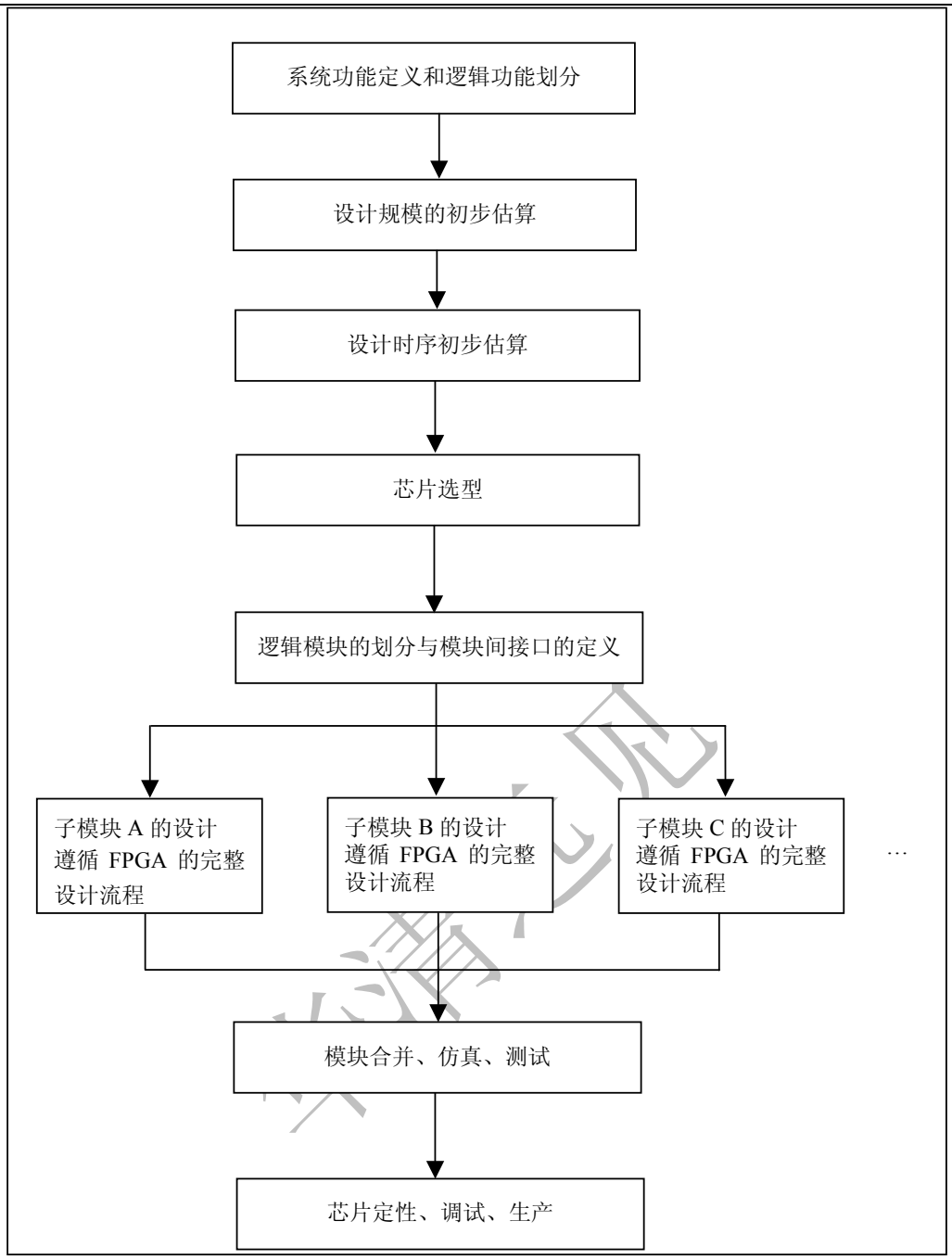

图 4.12　系统规划的简化流程

4.2.5　硬件设计原则的技巧

❖ 技巧内容

硬件原则是 FPGA/CPLD 设计的重要原则之一。本节讲解在 PLD 设计中采用硬件原则进行电路设计的技巧。

❖ 技巧详解

硬件原则主要针对 HDL 代码编写而言。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），同软件语言（如 C、C++等）是有本质区别的。以verilog 语言为例，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，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，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 代码这种表现实行转换得更流畅、更合理。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。

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

硬件原则的另外一个重要理解是“并行”和“串行”的概念。一般来说，硬件系统比软件系统速度快、实时性高。其中一个重要原因就是硬件系统中各个单元的运算是独立的，信号流是并行。而 C 语言编译后，其机器指令在 CPU 的高速缓冲队列中基本是顺序执行的，即使有一些并行处理的技术，也是在一定程度上十分有限的。所以在写 HDL 代码的时候，应该充分理解硬件系统的并行处理特点，合理安排数据流的时序，提高整个设计的效率。

另外，verilog 作为一种 HDL 语言，对系统行为的建模方式是分层次的。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）和电路开关级（switch）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 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。

4.2.6 选择 if 语句与 case 语句的技巧

❖ 技巧内容

本节讲解在 FPGA 设计中选择使用 if 语句和 case 语句的技巧。

❖ 技巧详解

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

例用 IF-Then-Else 完成 8 选 1 多路选择器，如图 4.13 所示。

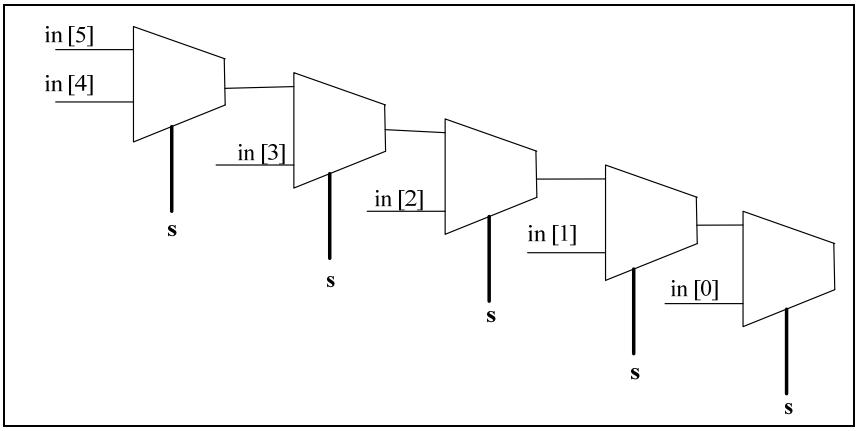

图 4.13 IF 语句实现电路选择

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

下面的例子是用 case 语句完成 8 选 1 多路选择器的 VHDL 实例。

在大多数 FPGA 结构中能够在单个 CLB 中完成一个 4 选 1 的多路选择器，Virtex 可以在单个 CLB 中完成一个 8 选 1 的多路选择器。而用 if-else 语句需要多个 CLB 才能完成相同功能因此 case 语句生成的设计速度更快延时更小。

用 case 实现 8 选 1 多路选择器，如图 4.14 所示。

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

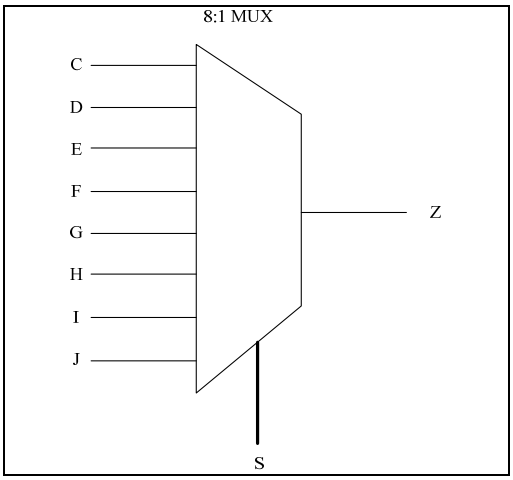

图 4.14 case 语句完成电路选择

4.2.7 分离组合电路与时序电路的技巧

❖ 技巧内容

本节讲解在 FPGA 设计中分离组合电路与时序电路的技巧。

❖ 技巧详解

包含寄存器的同步存储电路和异步组合逻辑应分别在独立的进程中完成，组合逻辑中关联性强的信号应放在一个进程中，这样在综合后面积和速度指标较高。这种方法常用在设计 Mealy 状态机中，Mealy 状态机的基本结构如图 4.15 所示。

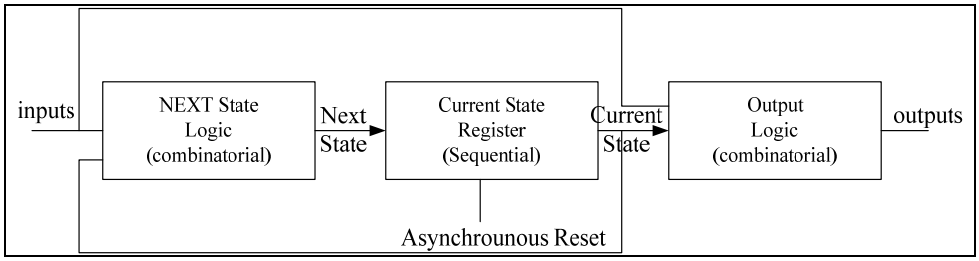

图 4.15 Mealy 型状态机结构

由图可看出，Mealy 状态机由次状态逻辑、当前状态寄存器和输出逻辑 3 部分组成，其中次状态逻辑和输出逻辑为组合逻辑，当前状态寄存器为时序逻辑。因此 Mealy 机可由 3 个进程实现，5 个状态的 Mealy 状态机如下面 VHDL 实例所示。

```
    -- Example of a 5-state Mealy FSM
    library ieee;
    use ieee.std_logic_1164.all;
entity mealy is
   port (clock, reset: in std_logic;
   data_out: out std_logic;
   data_in: in std_logic_vector (1 downto 0));
end mealy;

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

   -- FSM combinational block
   fsm: process (pres_state, data_in)
   begin
      case pres_state is
```

```
        when st0 =>
          case data_in is
            when "00" => next_state <= st0;
            when "01" => next_state <= st4;
            when "10" => next_state <= st1;
            when "11" => next_state <= st2;
            when others => next_state <= (others <= 'x');
          end case;
        when st1 =>
          case data_in is
            when "00" => next_state <= st0;
            when "10" => next_state <= st2;
            when others => next_state <= st1;
          end case;
        when st2 =>
          case data_in is
            when "00" => next_state <= st1;
            when "01" => next_state <= st1;
            when "10" => next_state <= st3;
            when "11" => next_state <= st3;
            when others => next_state <= (others <= 'x');
          end case;
        when st3 =>
          case data_in is
            when "01" => next_state <= st4;
            when "11" => next_state <= st4;
            when others => next_state <= st3;
          end case;
        when st4 =>
          case data_in is
            when "11" => next_state <= st4;
            when others => next_state <= st0;
        end case;
      when others => next_state <= st0;
      end case;
    end process fsm;

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

```
            when others => data_out <= '1';
          end case;
        when st3 => data_out <= '1';
        when st4 =>
          case data_in is
            when "10" => data_out <= '1';
            when "11" => data_out <= '1';
            when others => data_out <= '0';
          end case;
        when others => data_out <= '0';
      end case;
    end process outputs;
end behave;
```

4.2.8 乒乓操作的技巧

❖ 技巧内容

本节讲解在 FPGA 设计中一些常用设计思想之一——乒乓操作。

❖ 技巧详解

FPGA/CPLD 的设计思想与技巧是一个非常大的话题，本文将介绍一些常用的设计思想与技巧之一——乒乓操作。希望本文能引起工程师们的注意，如果能有意识地利用这些原则指导日后的设计工作，将取得“事半功倍”的效果。

“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 4.16 所示。

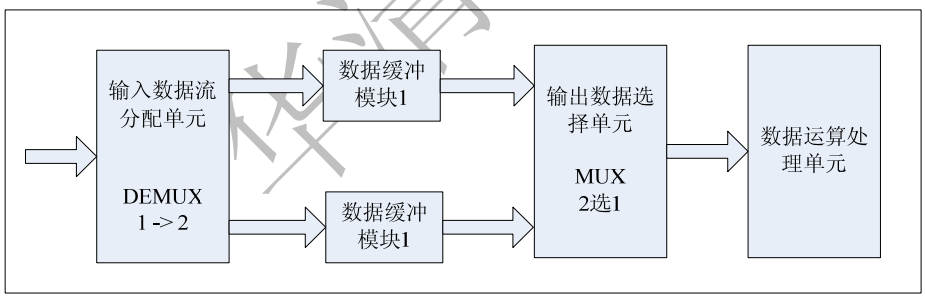

图 4.16 乒乓操作原理

乒乓操作的处理流程为：输入数据流通过“输入数据选择单元”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”，同时将“数据缓冲模块 1”缓存的第 1 个周期数据通过“输入数据选择单元”的选择，送到“数据流运算处理模块”进行运算处理；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”，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”切换，送到“数据流运算处理模块”进行运算处理，如此循环。

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停顿地送到“数据流运算处理模块”进行运算与处理。把乒乓操作模块当做一个整体，站在这个模块的两端看数据，输入数据流和输出数据流都是连续不断的，没有任何停顿，因此非常适合对数据流进行流水线式处理。所以乒乓操作常常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 个帧是由 15 个时隙组成的，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 3.84Mbit/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 个时隙数据的 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，两块 RAM 加起来也只有 5120Bit 的容量。

另外，巧妙地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 4.17 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的各种数据运算，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 100Mbit/s，乒乓操作的缓冲周期是 10ms。下面分析各个节点端口的数据速率。

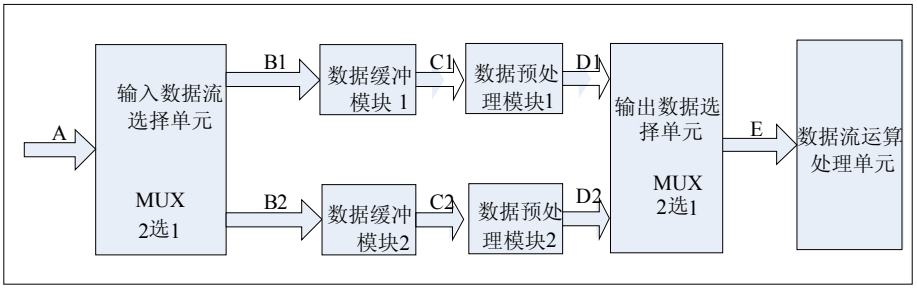

图 4.17　乒乓操作流程图

A 端口处输入数据流速率为 100Mbit/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 100Mbps，DPRAM1 要在 10ms 内写入 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 100Mbit/s，DPRAM2 在第 2 个 10ms 被写入 1MB 数据。在第 3 个 10ms，数据流又切换到 DPRAM1，DPRAM1 被写入 1MB 数据。

仔细分析就会发现到第 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。有的工程师困惑于 DPRAM1 的读数时间为什么是 20ms，这个时间是这样得来的：首先，在第 2 个缓冲周期向 DPRAM2 写数据的 10ms 内，DPRAM1 可以进行读操作；另外，在第 1 个缓冲周期的第 5ms 起（绝对时间为 5ms 时刻），DPRAM1 就可以一边向 500K 以后的地址写数据，一边从地址 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；在第 3 个缓冲周期的第 5ms 起（绝对时间为 35ms 时刻），同理可以一边向 500K 以后的地址写数据一边从地址 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 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=50Mbit/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 50Mbit/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 50Mbit/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体现。

4.2.9　串并转换的技巧

❖ 技巧内容

本节讲解在 PLD 设计中一些常用设计思想之一——串并转换设计技巧。

❖ 技巧详解

串并转换是 FPGA 设计的一个重要技巧，它是数据流处理的常用手段，也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。前面在乒乓操作的图例中，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得很大，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：

```
    X6Iz/6l1-cr3@lkoprl_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。串并转换的方法比较简单，在此不必赘述。

4.2.10 流水线操作设计的技巧

❖ 技巧内容

本节讲解在 PLD 设计中一些常用设计思想之一——流水线操作设计的技巧。

❖ 技巧详解

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”。

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来提高系统的工作频率。

流水线设计的结构示意图如图 4.18 所示。其基本结构为：将适当划分的 n 个操作步骤单流向串联起来。流水线操作的最大特点和要求是，数据流在各个步骤的处理从时间上看是连续的，如果将每个操作步骤简化假设为通过一个 D 触发器（就是用寄存器打一个节拍），那么流水线操作就类似一个移位寄存器组，数据流依次流经 D 触发器，完成每个步骤的操作。流水线设计时序如图 4.19 所示。

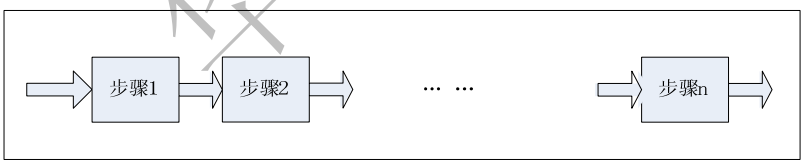

图 4.18 一般处理步骤

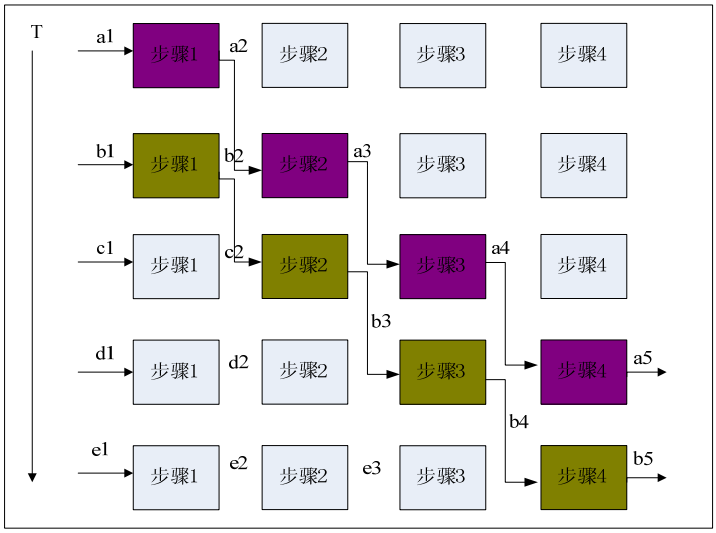

图 4.19　流水线设计流程

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

在 WCDMA 设计中经常使用到流水线处理的方法，如 RAKE 接收机、搜索器、前导捕获等。流水线处理方式之所以频率较高，是因为复制了处理模块，它是面积换取速度思想的又一种具体体现。

4.3　代码风格技巧

4.3.1　VHDL 的编码风格技巧

❖ 技巧内容

本节讲解在 FPGA 设计中 VHDL 语言编码风格的一些常用技巧。

❖ 技巧详解

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

（1）标识符（Identifiers）命名习惯。

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

标识符定义命名规定如下。

- 标识符第一个字符必须是字母，最后一个字符不能是下划线，不许出现连续两个下划线。
- 基本标识符只能由字母、数字和下划线组成。
- 标识符两词之间须用下划线连接，如 Packet_addr、Data_in、Mem_wr、Mem_ce。
- 标识符不得与保留字同名。
- 标识符大小写规定。
- 对常量、数据类型、实体名和结构体名采用全部大写。
- 对变量采用小写。
- 保留字一律小写。
- 信号名连贯缩写的规定。长的名字对书写和记忆会带来不便，甚至带来错误。采用缩写时应注意同一信号在模块中的一致性。一致性的缩写习惯有利于文件的阅读理解和交流。

部分缩写的统一规定如表 4.1 所示。

表 4.1

缩　　写	全　　称
Addr	address
Clk	clock
Clr	clear
Cnt	counter
En	enable
Inc	increase
Lch	latch

Mem	memory
Pntr	pointer
Pst	preset
Rst	reset
Reg	register
Rd	reader
Wr	write

- 常用多个单词的缩写 ROM、RAM、CPU、FIFO、ALU、CS、CE。
- 自定义的缩写必须在文件头注释。
- 信号名缩写的大小写规定。

单词的缩写若是信号名的第一个单词则首字符大写，如 Addr_in 中的 Addr。若该单词缩写不是第一个单词则小写，如 Addr_en 中的 en。

多个单词的首字符缩写都大写，不管该缩写在标识符的什么位置，如 RAM_addr、Rd_CPU_en。

- 信号名一致性规定：同一信号在不同层次应保持一致性。

信号命名有关建议：建议用有意义而有效的名字，能简单包含该信号的全部或部分信息，如输入输出信息 Data_in（总线数据输入）、Din（单根数据线输入）、FIFO_out（FIFO 数据总线输出），如宽度信息 Cnt8_q（8 位计数器输出信号的命名）。

建议添加有意义的后缀，使信号名更加明确，常用的后缀如表 4.2 所示。

表 4.2

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

说明：

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

模块内的反馈信号，在原信号名之后加后缀_s。“s”表示 same。

（2）数据对象和类型。

VHDL 是很强的类型语言，可综合的数据类型为标量类型（包含可枚举类型、整型、浮点型、物理类型）和组合类型（包含记录数组），模拟模型的数据类型为存取类型文件型，可综合的 VHDL 代码的编写不采用模拟类型浮点型物理类型。

不同基本类型的数据不能由另一类型赋值，不同类型间的赋值需使用运算符的重载。如 Cnt8_q 为 STD_LOGIC_VECTOR，类型若不对“+”运算符重载，则 Cnt8_q <= Cnt8_q + 1 语句在综合中将出错。可通过对“+”运算符进行重载，即使用“use IEEE.std_logic_arith.all”语句赋值语句是正确的。

常量名和数据类型必须用大写标识符表示。

- 数据及数据类型使用建议。为改善代码的可读性，建议可把常用的常量和自定义的数据类型在程序包中定义。

建议使用别名来标识一组数据类型有利于代码的清晰，如：

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

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

（3）信号和变量。

- 信号不许赋初值。
- 变量使用建议。变量主要用在高层次的模拟模型建模及用于运算的用途，但变量的综合较难定义，对于编写可综合的 VHDL 模块，在没有把握综合结果情况下建议不使用。
- 信号变量使用注意内容。在 VHDL 中，信号（signal）代表硬件连线，因此可以是逻辑门的输入输出。同时，信号也可表达存储元件的状态。端口也是信号。

在进程（process）中，信号是在进程结束时被赋值。因此在一个进程中，当一个信号被多个信号所赋值时，只有最后一个赋值语句起作用。如下：

```
Sig_p process(A, B , C)
begin
  D <= A ;
  X <= C or D;
  D <= B;
  Y <= C xor D;
End;
```

上面实际的结果是 B 赋值给 D，（C xor B）结果赋值给 X、Y。

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

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

实际结果是 X <= C or A Y<= C xor B。

（4）实体。

- 实体结构体使用规定：

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

除 IEEE 大写外，其余小写。

实体名和结构体名必须用大写标识，实体名必须与文件名同名。自定义的其他标识符如信号名、变量名、标号等不得与实体名、结构体名同名。

实体端口数据模式不准使用 buffer 模式。缓冲模式主要用在实体内部可读的端口，如计数器的输出。为简化大型设计各模块间接口的配合，要求不要使用。需要反馈的信号可定义内部信号来解决。如计数器端口 Count，可内部定义信号“signal Cnt8_q：STD_LOGIC_VECTOR(7 downto 0);”。“Cnt8_q”该信号可在内部反馈，最后通过赋值语句“Count <= Cnt8_q;”来实现端口的定义。

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

一个文件只对应一个实体，实体是设计文件的基本单元，其书写规范要求如下：

一条语句占用一行每行应限制在 80 个字符以内；

如果较长超出 80 个字符则要换行；

代码书写要有层次即层层缩进格式清晰美观；

要有必要的注释 25%。

实体开始处应注明文件名功能描述，引用、模块设计者、设计时间及版权信息等，如：

```
-- Filename ：
-- Author ：
-- Description ：
-- Called by ：Top module
-- Revision History ：99-08-01
-- Revision 1.0
-- Email ： M@163.com.cn
--Company ： Inc
--Copyright(c) 1999, Inc, All right reserved
library IEEE;
use IEEE.std_logic_1164.all;
entity ENTITY_NAME
port(
    Port1 : in STD_LOGIC;
    Port2 : in STD_LOGIC;
    Port3 : out STD_LOGIC;
…
    Portn : out STD_LOGIC
);
end ENTITY_NAME ;

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

- 实体使用建议：实体名的命名建议能大致反映该实体的功能，如 COUNTER8（8 位宽的计数器模块），DECODER38（3-8 线译码器模块）。

行为级的结构体名命名为 BEHAVIOR，结构级的结构体名命名为 STRUCTURE，若有多个结构体用后缀 A、B……命名。如：

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

一个实体可以有多个结构体，对单个结构体的实体，文件要包含结构体和实体说明，便于查阅。对多个结构体的实体建议把常用的结构体放在文件中，其余结构体用单独文件表示，使用时用 configuration 语句进行配置。

结构体的描述分为行为级描述、数据流描述和结构化描述。若无特殊要求，建议采用行为级描述和数据流的描述，不采用结构化描述或 BOOLEAN 数据流的描述。

VHDL 设计中，如果可以避免采用器件厂商的专用元件库（硬 Core），则尽量不要使用，除非只有采用该库元件才能实现你设计的性能指标。这是因为要充分利用 VHDL 独立于工艺且易于维护的优点。

- 实体使用注意内容。

VHDL 设计应是层级型的设计。VHDL 设计实体由实体说明和结构体组合而成。实体是一个设计的基本单元模块，即顶层的设计模块由次一级的实体构成，每个次一级实体又可由再下一层次的实体构成，最低层模块可以由表达式或最基本的实体模块构成。这种设计方法就是 Top-To-down 的设计方法。

实体端口模式为 in、out、buffer、inout。模式为 in 的信号不能被驱动，模式 out 的信号不能用于反馈，同时必须仅被一个信号所驱动，缓冲模式的端口不能被多重驱动（除非用决断函数解决外），同时仅可以连接内部信号或另一个实体的缓冲模式的某个端口。

VHDL 设计各模块接口定义时要考虑模块间配合的方便，如实体端口的模式、端口的数据类型等。

（5）语句。

- VHDL 各语句使用规定。

① with-select-when 语句书写规范规定。

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

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

with- select - when 语句的 selection_signal 的所有值必须具备完整性，若没写完整必须有一个 others 语句。如下 3 个写法，综合的效果是一致的。因为 S 的元素不是已知的逻辑值，X 将不被定义，但对 RTL 仿真而言，其结果是不一致的。这是因为 RTL 仿真支持多值元素。

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

建议不使用第三种写法。

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

when_else 语句书写规范规定：

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

```
Signal_name <= value_a when condition1 else
               value_b when condition2 else
               value_c when condition3 else
               ......
               value_x;
```

当条件是表达式时表达式须用括起来，使代码更为清晰，如 when（a＝b and C=‘1’）else。

if 语句必须有一个 else 对应，除在下面例子的情况下可不写 else 语句。例子代码为：

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

当没有 else 语句时，将产生不希望的存储器。

② case… when 语句书写规范规定。

该语句用于规定一组根据给定选择信号的值而执行的语句，可用 with-select-when 语句等效实现。代码的书写规范为：

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

case…when 语句必须有 when others 支项。

若信号在 if …else 或 case…when 语句作非完全赋值，必须给定一个缺省值。

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

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

```
Lab process Clk, Rst
list1 list2...
begin
```

每个 process 前须加个 lable。

不同逻辑功能采用不同的 process 进程块，把相同功能的放在同一进程中，如触发器组进程块：

```
D_p : process(Clk,Rst,D1,D2,...,Dn)
begin
   if (Rst = '1' ) then
      Q1 <= '0' ;
```

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

③ generate 语句书写规范规定。

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

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

生成机构必须有一个标号，如上例中的 Gen_lab1。

if -then 用在生成机构中，不能有 else 或 elseif 语句，如下例为复杂的生成机构语句：

```
G1 for I in 0 to 3 generate
    G2 for j in 0 to 7 generate
        G3 if (I <1 ) then generate
            Ua : thrst port map( Val(j),Rd,Val_out(j));
        end generate;
        G4 if (i = 1) then generate ......
```

④ port map 语句书写规范规定。

port map 语句书写规范如下：

```
Uxx Module_name
    port map(
        port1 => port 1,
        port2 => port 2,
        ......
        portn => port n);
```

为便于阅读 port map，采用名字对应（=>）映射方法。

port map 中总线到总线映射时（X downto Y）要写全。

向量采用降序方法，即 X downto Y 格式，向量有效位顺序的定义为从大数到小数。

port map 的 module（设计者自编写的 entity）名用 Uxx 标识，cell（如厂家提供的库元件、RAM、Core 等）名用 Vxx 标识。

- VHDL 语句使用建议。

作为可综合的代码编写，“–”值建议不用。如下例中的代码：

```
with tmp select
    X <= A when "1---",
         B when "-1--",
         C when "--1-",
```

```
        D when "---1",
        0 when others;
```

该代码在 RTL 级仿真中不会出错，但在综合过程中可能编译出错，视综合工具而定。
由于不同综合工具支持能力问题，建议不采用 wait 语句，即不使用隐式敏感表。

- VHDL 语句使用注意内容。

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

- 表达式书写规定。

为便于理解用表示逻辑运算符执行优先级，如“X <= （(A and B) and (C or not D))； ”，建议运算操作符两边都加上空格。
比较运算符规定：向量比较时，比较的向量的位宽要相等，否则会引起 warning 或 error。除非重载等值比较运算符（调用 numeric_std 库）。
（7）函数 function()。

- function 代码书写规范规定如下所示。

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

- function 使用建议。

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

- function 使用注意内容。

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

- procedure 书写规范规定如下所示。

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

- procedure 使用注意内容。

过程用于数值运算、类型转换、运算符重载或设计元件的最高层设计结构。

过程参数缺省模式为 in。

（8）类属（generic）。

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

```
Uxx : 参数化的实体名 generic map 实参
port map (
    端口映射表;
);
```

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

（9）包（package）。

- package 使用建议。对大型设计，建议把全局的常量（如数据宽度等）、指令状态编码、元件组函数和子程序组分别用元素包、器件包、函数包来构造。如通过调用元件程序包，实体的结构体说明区域中就不必再对调用的器件进行 component 说明。

程序包通过 use 语句使之可见。可通过保留字 all 使包中所有单元都可见，如：

```
use work..yourpacketname.all;
```

其中 yourpacketname 是 packet 名。

- package 使用注意内容。程序包由程序包说明和可选包体构成。程序包说明用来声明包中的类型、元件、函数和子程序；包体则用来存放说明中的函数和子程序。

不含有子程序和函数的程序包不需要包体。

程序包中的类型、常量、元件、函数和其他说明对其他设计单元是可见的。

4.3.2 Verilog HDL 的编码风格技巧

❖ 技巧内容

本节讲解在 FPGA 设计中 Verilog HDL 语言编码风格的一些常用技巧。

❖ 技巧详解

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

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

（1）用有意义而有效的名字，有效的命名有时并不是要求将功能描述出来，如以下代码所示。

```
For ( I = 0; I < 1024; I = I + 1 )
Mem[I] <= #1 32'b0;
```

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

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

addr： address

pntr： pointer

clk： clock

rst： reset

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

（3）大小写原则：名字一般首字符大写，其余小写（但 parameter、integer 定义的数值名可全部用大写），两个词之间要用下划线连接。如 Packet_addr、Data_in、Mem_wr、Mem_ce_。

（4）全局信号名字中应包含信号来源的一些信息，如 D_addr[7:2]。这里的“D”指明了地址是解码模块（Decoder module）中的地址。

（5）同一信号在不同层次应保持一致性。

（6）自己定义的常数类型等用大写标识，如“parameter CYCLE=100”。

（7）避免使用保留字，如 in，out，x，z 等不能够作为变量、端口或模块名。

（8）添加有意义的后缀，使信号名更加明确。

顶层模块应只是内部模块间的互连。

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

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

```
/* 文件头需要注释，说明文件信息
*Filename ： RX_MUX.v
*Author ：
*Description ：
*Called by ：Top module
*Revision History ：99-08-01
*Revision 1.0
*Email ： M@163.com.cn
*Company ： .Inc
*Copyright(c) 1999, Inc, All right reserved
*****************************/
```

不要对 Input 进行驱动，在 module 内不要存在没有驱动的信号，更不能在模块端口中出现没有驱动的输出信号，避免在仿真或综合时产生 warning，干扰错误定位。

每行应限制在 80 个字符以内，以保持代码的清晰、美观和层次感。一条语句占用一行如果较长（超出 80 个字符）则要换行。

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

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

统一用一个时钟的上沿或下沿采样信号，不能一会儿用上沿，一会儿用下沿。如果既要用上沿又要用下沿，则应分成两个模块设计。建议在顶层模块中对 Clock 做 1 个非门，在层次模块中如果要用时钟下沿就可以用非门产生的 Posedge Clk_，这样的好处是在整个设计中采用同一种时钟沿触发，有利于综合，基于时钟的综合策略。

在模块中增加注释：对信号、参量、引脚、模块、函数及进程等加以说明，便于阅读与维护。

Module 名要用大写标示且应与文件名保持一致。

如：

```
Module DFF_ASYNC_RST(
   Reset,
   Clk,
   Data,

                                  ********************** ********
                                                                Qout);
```

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

模块输出寄存器化：对所有模块的输出加以寄存，如图 4.20 所示，使得输出的驱动强度和输入的延迟可以预测，从而使得模块的综合过程更简单。

输出驱动的强度都等于平均的触发器驱动强度。

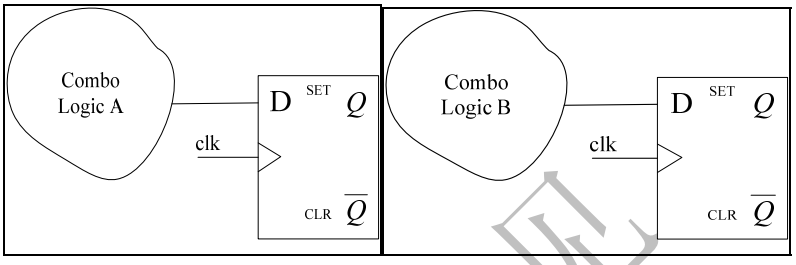

图 4.20　输出加以寄存器

将关键路径逻辑和非关键路径逻辑放在不同模块：保证 DC 可以对关键路径模块实现速度优化，而对非关键路径模块实施面积优化。在同一模块 DC 无法实现不同的综合策略。

将相关的组合逻辑放在同一模块，有助于 DC 对其进行优化，因为 DC 通常不能越过模块的边界来优化逻辑。

Net and Register：一个 reg 变量只能在一个 always 语句中赋值。

向量有效位顺序的定义一般是从大数到小数。

尽管定义有效位的顺序很自由，但如果采用毫无规则的定义势必会给作者和读代码的人带来困惑，如 Data[-4 0]，则 LSB[0][-1][-2][-3][-4]MSB，或 Data[0 4]，则 LSB[4][3][2][1][0]MSB，这两种情况的定义都不太好，推荐 Data[4 0]这种格式的定义。

对 net 和 register 类型的输出要做声明（在 PORT 中）。如果一个信号名没做声明，Verilog 将假定它为一位宽的 wire 变量。

线网的多种类型，寄存器的类型。

Expressions：用括号来表示执行的优先级，尽管操作符本身有优先顺序，但用括号来表示优先级对读者更清晰，更有意义。

“if ((alpha < beta) && (gamma >= delta))”比“if（alpha < beta && gamma >= delta）表达更合意。

用一个函数（function）来代替表达式的多次重复，如果代码中发现多次使用一个特殊的表达式，那么就用一个函数来代替，这样在以后的版本升级时更便利，这种概念在做行为级的代码设计时同样使用，经常使用的一组描述可以写到一个任务（task）中。

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

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

```
If (Abc = = 8'b0) begin
```

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

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

以上语句 DataOut 会综合成锁存器。
应注意 if ...else if ...else if ...else 的优先级。
如果变量在 if...else 或 case 语句中做非完全赋值，则应给变量一个缺省值，即：

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

case 语句：case 语句通常综合成一级多路复用器，而 if...then...else 则综合成优先编码的串接的多个多路复用器。通常，使用 case 语句要比 if 语句快，优先编器的结构仅在信号的到达有先后时使用。条件赋值语句也能综合成多路复用器，而 case 语句仿真要比条件赋值语句快。
所有的 case 应该有一个 default case，允许空语句如下。

```
Default : ;
```

Writing functions：在 function 的最后给 function 赋值，范例如下。

```
Function CompareVectors; // (Vector1, Vector2, Length)
  Input [199:0] Vector1, Vector2;
  Input [31:0] Length;
  //local variables
  Integer i;
  Reg Equal;
Begin
  i = 0;
  Equal = 1;
  While ((i<Length) && Equal) begin
    If (Vector 2[i] !== 1'bx) begin
      If (Vector1[i] !== Vector2[i])
        Equal = 0;
      Else ;
    End
```

```
      i = i + 1;
    End
    CompareVectors = Equal;
  End
  Endfunction //compareVectors
```

函数中避免使用全局变量，否则容易引起 HDL 行为级仿真和门级仿真的差异。范例如下。

```
  function ByteCompare
     input [15:0] Vector1
     input [15:0] Vector2
     input [7:0] Length
  begin
     if (ByteSel)
        // compare the upper byte
     else
        // compare the lower byte
     end
  endfunction // ByteCompare
```

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

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

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

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

以上语句在行为级仿真时 e 的变化将不会使仿真器进入该进程，导致仿真结果错误。

- Assign/deassign 仅用于仿真加速，仅对寄存器有用。
- Force/release 仅用于 debug，对寄存器和线网均有用。
- 避免使用 Disable。

对任何 reg 赋值用非阻塞赋值代替阻塞赋值，reg 的非阻塞赋值要加单位延迟，但异步复位可加可不加。

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

```
      Else if (Load_init)
         Begin
            Rega <= #u_dly init_rega;
            Regb <= #u_dly init_regb;
         End
      Else
         Begin
            Rega <= #u_dly Rega << 1;
            Regb <= #u_dly St_1;
         End
   End // end Rega, Regb assignment.
```

Combinatorial Vs Sequential Logic：如果一个事件持续几个时钟周期，设计时就用时序逻辑代替组合逻辑。如：

```
Wire Ct_24_e4;     //it ccarries info. Last over several clock cycles
Assign Ct_24_e4 = (count8bit [7:0] >= 8'h24) & (count8bit [7:0] <= 8'he4);
```

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

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

内部总线不要悬空，在 default 状态要把它上拉或下拉，代码如下。

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

Macros：为了保持代码的可读性，常用“define”做常数声明。
把“define”放在一个独立的文件中，参数 parameter 必须在一个模块中定义不要传替参数到模块（仿真测试向量例外），“define”可以在任何地方定义，要把所有的“define”定义在一个文件中，在编译原代码时首先要把这个文件读入。如果希望宏的作用域仅在一个模块中，就用参数来代替。
Comments：对更新的内容更新要做注释。在语法块的结尾做标记。代码如下。

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

```
    //style 2 --- identical lables on begin and end

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

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

4.3.3 命名的技巧

❖ 技巧内容

本节讲解在 FPGA 设计中信号命名的一些常用技巧。

❖ 技巧详解

一个好的命名规则能够提供较好的可读性。首先在 VHDL 语言中大写字母和小写字母是没有区别的，也就是说，在所有的语句中写大写字母可以，写小写字母也可以，混合起来写也可以。但是，有两种情况是例外，这就是用单引号括起来的字符常数和用双引号括起来的字符。这时大写字母和小写字母是有区别的。例如，在 STD_LOGIC 和 STD_LOGIC_VECTOR 代入不定值“X”时应注意。

```
SIGNAL a : STD_LOGIC;
SIGNAL b : STD_LOGIC_VECTOR（3 downto 0）;
a<='X'; --X 用小写字母是错误的
b<="XXXX"; --X 用小写字母是错误的
```

在 VHDL 语言中所使用的名字（名称），如信号名、实体名、构造体名、变量名等，在命名时应遵守如下规则：

- 名字的最前面应该是英文字母；
- 能使用的字符只有英文字母、数字和‘—’；
- 不能连续使用‘—’符号，在名字的最后也不能使用‘—’符号。

例如：

```
SIGNAL a_bus : STD_LOGIC_VECTOR（7 downto 0）;
SIGNAL 302_bus: … --数字开头的名字是错误的
SIGNAL b_@bus: … --@符号不能作为名称的字母，是错误的
SIGNAL a_ _bus: … -- ‘—’ 符号在名称中不能连着使用，是错误的
SIGNAL a _bus_: … -- ‘—’ 符号不能在名称最后使用，是错误的
```

此外，本节还给出了 Verilog HDL 语言中信号命名的一些规则和技巧。

- 命名方式分类。

底线分隔型：xxx_yyy_zzz。
大写底线分隔型：XXX_YYY_ZZZ。
首字大写型：AbcDefGhi。
首字小写型：avcDefGhi。

- 各种元素所使用的命名。

文件名称：底线分隔型，例如 xxx_yyy_zzz.v。

module 名称：底线分隔型，例如 xxx_yyy_zzz。
module instance 名称：底线分隔型，例如 xxx_yyy_zzz。
local wire 名称：底线分隔型：例如 xxx_yyy_zzz。
local reg 名称：底线分隔型，例如 xxx_yyy_zzz。
input signal 名称：前置 i_的底线分隔型，例如 i_xxx_yyy_zzz。
output signal 名称：前置 o_的底线分隔型，例如 o_xxx_yyy_zzz。
input/output signal 名称：前置 io_的底线分隔型，例如 io_xxx_yyy_zzz。
常数名称：大写底线分隔型，例如 XXX_YYY_ZZZ。
parameter 参数名称：大写底线分隔型，例如 XXX_YYY_ZZZ。
block 名称：大写底线分隔型，例如 XXX_YYY_ZZZ。

- 特殊讯号名称。

单一的 clock signal：clk。
多个 clock signal：clk_xxx。
负沿触发的 clock signal：clk_n, clk_xxx_n。
单一的 reset signal：rst。
多个 reset signal：rst_xxx。
负沿触发的 reset signal：rst_n，rst_xxx_n。
单一的 set signal：set。
多个 set signals：set_xxx。
负沿触发的 set signals：set_n，set_xxx_n。
使能信号：en_xxx。
除能信号：dis_xxx。

4.3.4 添加注释的技巧

❖ 技巧内容

本节讲解在 FPGA 设计中添加注释的一些常用技巧。

❖ 技巧详解

无论是针对 VHDL 语言还是 Verilog HDL 语言，添加注释是十分必要的。不仅可以帮助作者很容易地了解模块的功能，对于后继者理解代码也发挥着十分重要的作用。使用注释时，需要注意以下事项。

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

同其他的高级语言一样，HDL 语言的程序有注释栏目，可以对所编写语句进行注释。
对 VHDL 语言来说，注释从“——”符号开始到该项末尾（回车、换行符）结束。注释文字虽然不作为 VHDL 的语句予以处理，但是有时也用于其他工具和接口。
对 Verilog HDL 语言来说，注释从‘//’符号开始到该项末尾（回车、换行符）结束，也可以用‘/*……*/’符号选择要注释的内容。

4.3.5 模块划分的技巧

❖ 技巧内容

本节将讲解模块划分的一些常用技巧。

❖ 技巧详解

模块划分关系到能否最大程度上发挥项目成员协同设计的能力，更重要的是它决定着设计的综合、实现的耗时与效率。模块划分的基本原则如下。

- 对每个同步时序设计的子模块的输出使用寄存器：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；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。
- 降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个模块，可以最大程度上复用资源，减少设计所消耗的面积；更利于综合工具优化某个具体功能的时序关键路径，因为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个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：在介绍速度和面积的平衡与互换原则时，谈到合理的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“Aera”。同理，将它们规划到一起，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，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，耗费巨大的综合优化时间也得不到令人满意的综合优化效果。
- 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，如多周期路径等。将这些模块归类，并制定松约束，则可以让综合器尽量节省面积资源。
- 将存储逻辑独立划分成模块：RAM、ROM、CAM 和 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：从理论上讲，模块的规模越大，越利于模块资源共享。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，庞大的规模划分，不利于发挥目前非常流行的增量综合与实现技术的优势。

4.3.6 模块重用的技巧

❖ 技巧内容

本节将讲解模块重用的一些常用技巧。

❖ 技巧详解

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，首先看一个例子。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，请大家判断一下优劣，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;
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;
output [15:0]  square;
Wire  [7:0]  data_temp;

Assign data_temp = (data_in[7])? (~data_in +1) : data_in;
Assign square = data_temp * data_temp;
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；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 的乘法器；第二部分，后者的选择器是 1bit 判断 8bit 输出，而前者的 1bit 判断 16bit 输出。

将上述两种实现方式通过综合工具综合后发现，第二种实现方式所需资源比第一种实现方式相差了一倍以上。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。其实如果拓展一个思维，将乘法器换成加法器、除法器等，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 成倍地节省前级模块所消耗的资源。

目前很多综合工具都有“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。但是，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，放松对编码风格的要求。这是因为：第一，综合工具的优化力度毕竟有限，很多情况下不能智能地发现需要 Resource Sharing 的逻辑；第二，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不十分可靠；第三，在 ASIC 设计中，综合工具非常忠于用户意图，这时编码风格更加重要。所以工程师必须注意并不断提高自己在编码风格方面的修养。

4.3.7 编写可综合代码的技巧

❖ 技巧内容

本节将讲解编写可综合代码的一些常用技巧。

❖ 技巧详解

首先介绍一下针对综合的编码准则。

（1）寄存器推断。

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

时序逻辑的 VHDL 模板如下例所示。

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

（2）避免使用 LATCH。

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

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

if 语句中缺乏 else 子句，case 语句中各个条件所处理的变量不同。在综合时推出 LATCH，使用下述方法可避免 LATCH 在每个 process 的开始给信号赋初始值：对所有的输入条件都给出输出；在最终优先级的出发上使用 else 子句，而不用 else…if。

如果必须使用 LATCH，建议使用下述电路使得 LATCH 可测，如图 4.21 所示。

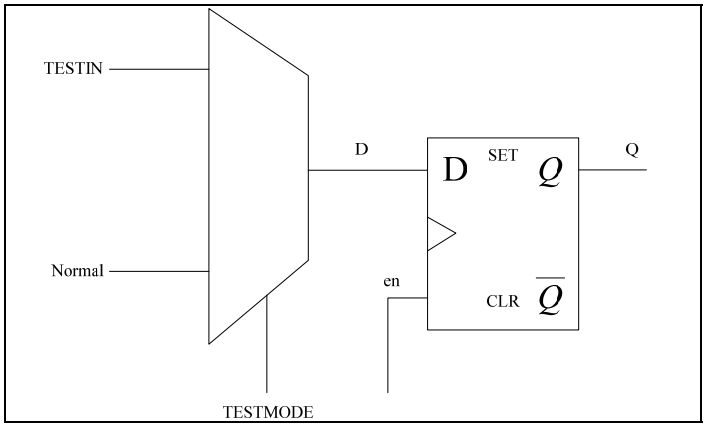

图 4.21　Latch 使用电路

（3）避免组合反馈。

在设计中，避免组合反馈电路。

（4）定义完整的敏感列表。

在每一个 process（VHDL 中）要求定义完整的敏感列表。对于组合模块，敏感列表中必须包含被 process 所利用的所有信号。这通常意味着包含所有出现在赋值语句右边和条件表述式中的信号；对于时序模块敏感列表中，必须包含时钟，如果有异步复位还要包含复位信号。
确保敏感表中没有包含不必要的信号，敏感列表中不必要的信号会降低仿真的性能。

4.4 提高速度的技巧

4.4.1 设置速度约束的技巧

❖ 技巧内容

本节将讲解在布局布线时，如何对布线工具设置速度约束。

❖ 技巧详解

本节主要针对 Xilinx 和 Altera 两个厂商的布线工具，介绍如何设置速度约束。
对于 Altera 厂商提供的布线工具 Quartus II：在菜单栏中选择【Assignments】/【settings】命令，弹出如图 4.22 所示对话框。在下面的界面中，选择【Analysis&Synthesis settings】选项。然后在“Optimization Technology”栏中选择“Speed”，最后单击【OK】按钮即可。

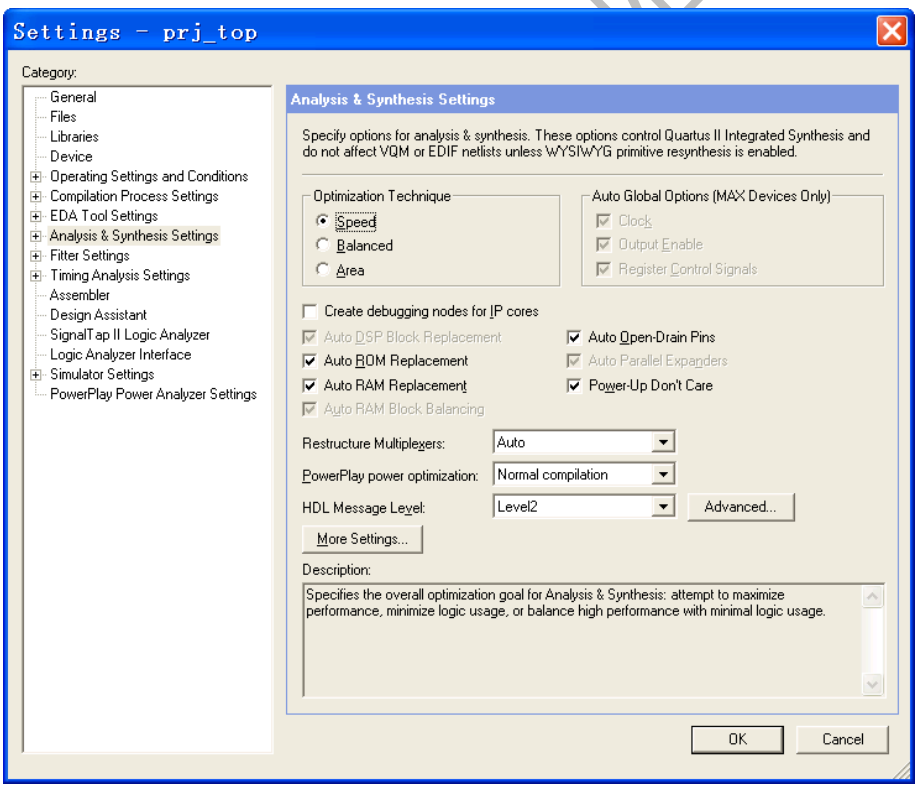

图 4.22 Quartus II 速度约束

对于 Xilinx 厂商提供的布线工具 ISE：在“process”栏中选择“Implement Design”，单击右键选择“Properties”命令，弹出如图 4.23 所示对话框。在下面的界面中，选择“Map Properties”选项。然后在“Optimization Strategy”栏中选择“Speed”，最后单击【OK】按钮即可。

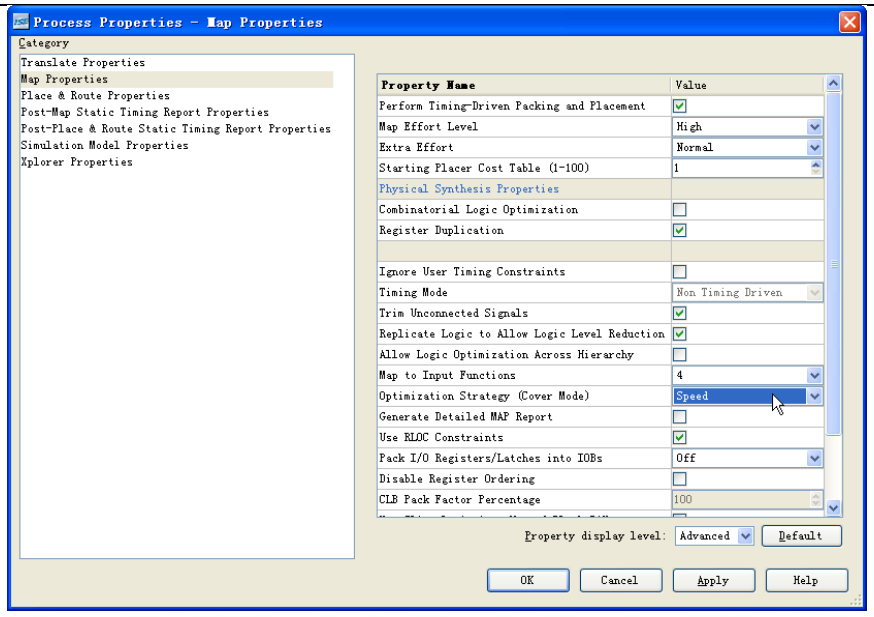

图 4.23　ISE 速度约束

4.4.2　专用资源提高速度的技巧

❖ 技巧内容

本节将讲解使用专用资源提高速度的一些常用技巧。

❖ 技巧详解

FPGA 厂商为客户提供了一些专用资源，如进位链 MUX、SRL、乘法器等。客户可利用 Coregen 工具产生宏单元，来利用专有资源。

利用专有资源虽然可以提高速度，但有一缺点，就是降低了代码的可移植性。如果是准备转 ASIC，则需对专有资源进行代码改动。这样，就增加了出错的可能性。因此，在 ASIC 设计时，采用这种方法要仔细权衡。

4.4.3　分配关键路径的技巧

❖ 技巧内容

本节将讲解如何分配关键路径的一些常用技巧。

❖ 技巧详解

首先，对于关键路径的处理要遵守以下原则。

- 关键路径在同一个 Module，这样在综合时可以获得最佳效果。
- 关键路径不与其他模块放在一起综合，对关键路径所在模块采取速度优先策略，对非关键路径模块采用面积优先策略。
- 针对关键路径进行位置约束，如果发现关键路径相关 LUT 距离太远，可通过 floorplanner 手工布线，并形成位置约束文件以指导布局布线。
- 迂回策略，降低非关键路径上的面积，为关键路径节约空间。
- 尽可能优化非关键路径上的面积，以尽量多给关键路径预留空间，以便将关键路径相关 LUT 压缩在一起，降低布线延时。该方法体现了向非关键路径要面积，向关键路径要速度的设计思想。

- 提高关键路径速度的一个常用方法是复制电路，减少关键路径的扇出。当一个信号网络所带负载增加时，其路径延时也相应增加。这对复位信号网络可能影响不大，但对像三态使能信号这类的信号是不能容忍的。扇出对关键路径延时的影响甚至超过了逻辑的级延时，确保一个网络的扇出少于一定值，例如 16 表示某个信号所驱动的基本器件不超过 16 个是很重要的。

下面的 VHDL 实例中信号 Tri_en 的扇出为 24，如图 4.24 所示。

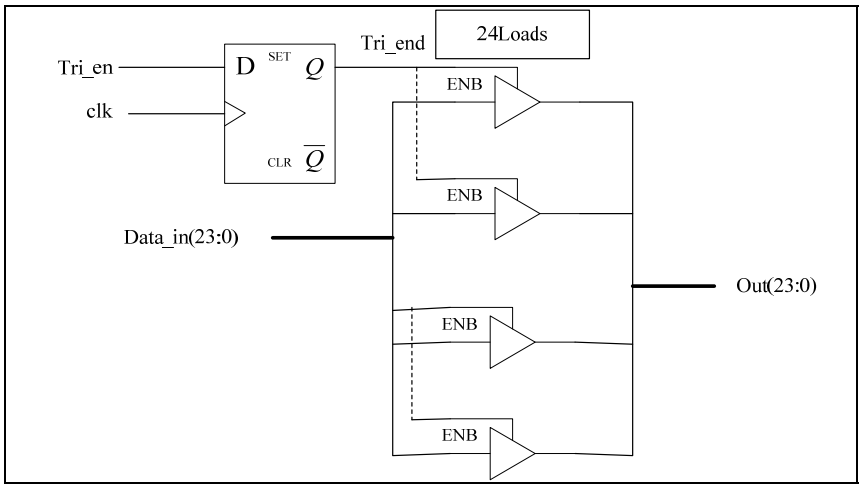

图 4.24　扇出较大

为了将扇出降低一半，增加一个寄存器使负载分成两部分，每个寄存器扇出为 12，如图 4.25 所示。

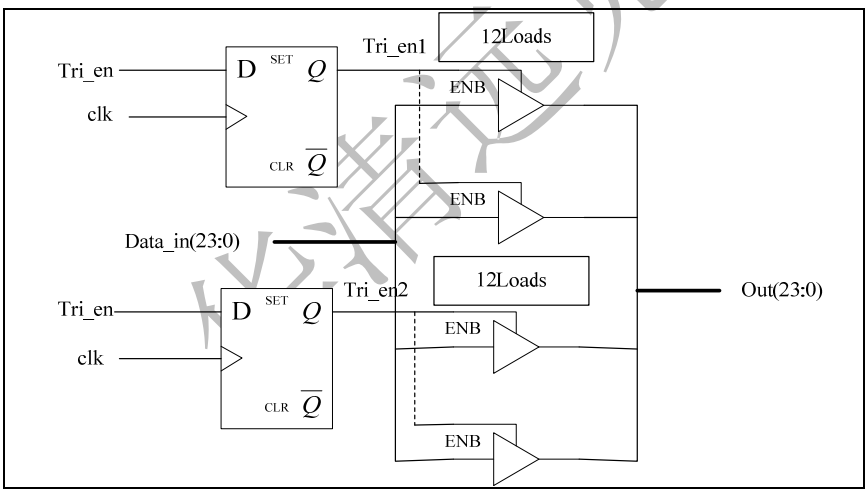

图 4.25　扇出较小

类似地，还可以使用复制组合逻辑电路、网络上插入 buffer 等手段减少扇出，提高速度。

4.4.4　进行特殊约束的技巧

❖ 技巧内容

本节将讲解进行特殊约束的一些常用技巧。

❖ 技巧详解

许多设计者觉得设置 TIG 和 Multi-Cycle-Path 意义不大，因为它们不直接对关键路径发生作用，这种想法是错误的。虽然它们不直接对关键路径发生作用，但可以起到让非关键路径散开的作用。让这些非关键路径离得越远越好，这样就为关键路径节约出空间。与关键路径相关的 LUT 可以尽量压缩在一起，从而到达压缩关键路径上线延时的目的。这实际是一种“曲线救国”的策略。
实践证明这种方法非常行之有效，而且它的一个最大好处是不用更改设计。

4.4.5 减少逻辑级数的技巧

❖ 技巧内容

本节将讲解减少逻辑级数的一些常用技巧。

❖ 技巧详解

在 FPGA 中关键路径（critical path）上的每一级逻辑都会增加延时。为了保证能满足时间约束，就必须在对设计的行为进行描述时，考虑逻辑的级数减少关键路径的延时。常用方法是：给最迟到达的信号最高的优先级，这样能减少关键路径的逻辑级数。
下面的实例描述了如何减少关键路径上的逻辑级数。
通过等效电路赋予关键路径最高优先级，此例中 critical 信号经过了二级逻辑，如图 4.26 所示。

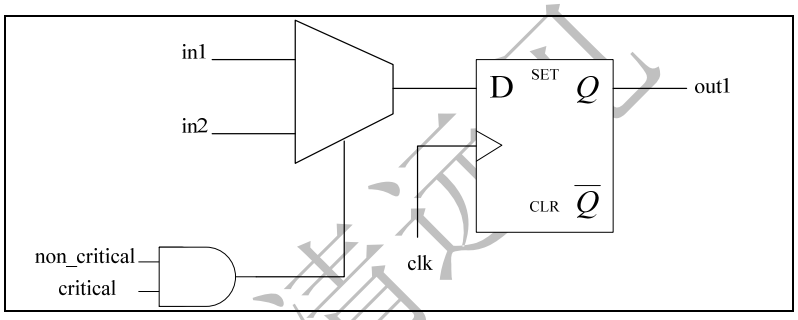

图 4.26　critical 信号经过二级逻辑

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

为了减少 critical 路径的逻辑级数，将电路修改如图 4.27 所示，critical 信号只经过了一级逻辑。

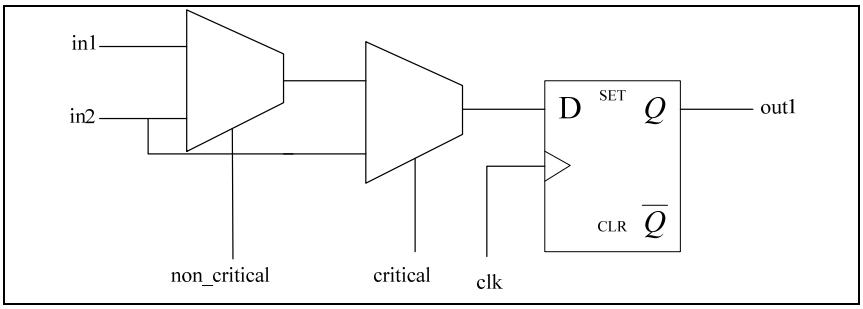

图 4.27　critical 信号只经过一级逻辑

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

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

4.4.6 分割组合逻辑的技巧

❖ 技巧内容

本节将讲解对组合逻辑进行分割的一些常用技巧。

❖ 技巧详解

在功能等价的情况下，可以根据时序需要，安排组合逻辑电路在寄存器前后的位置，合理地分配延时。本例中组合逻辑在寄存器之后，如图 4.28 所示，假定 a、b 信号的延时非常大。

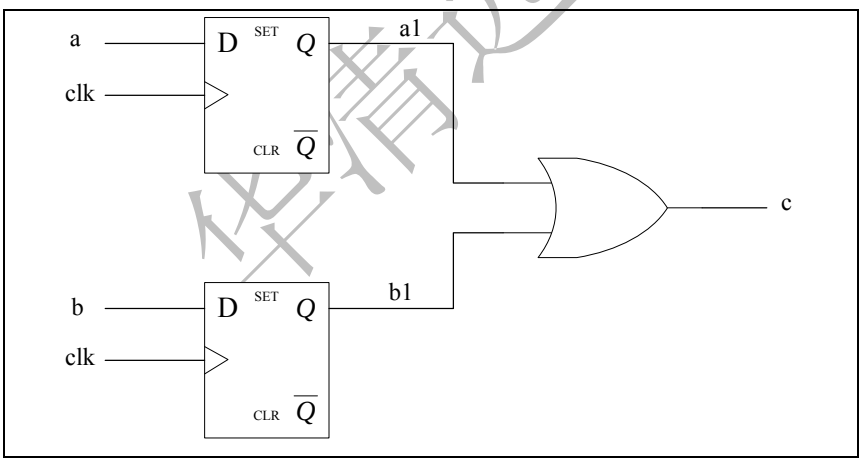

图 4.28 组合逻辑在后

组合逻辑放在寄存器之前如图 4.29 所示，如果 a、b 信号的延时并不大，而寄存器 c 信号经过的逻辑比较多，则延时大些。

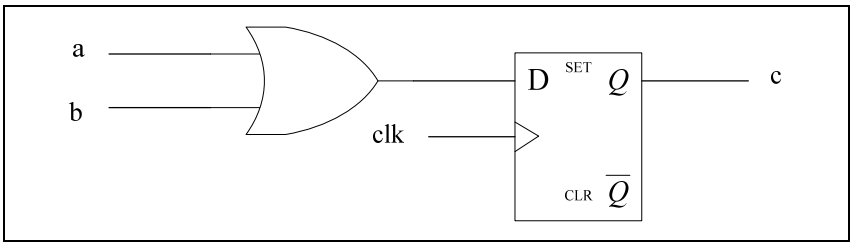

图 4.29 组合逻辑在前

这种处理方法的实质是，将关键路径中的部分延时挪到其他非关键路径上。

4.4.7 转移组合逻辑的技巧

❖ 技巧内容

本节将讲解对组合逻辑进行转移的一些常用技巧。

❖ 技巧详解

由于一般同步电路都不只一级锁存，而要使电路稳定工作，时钟周期必须满足最大延时要求，缩短最长延时路径，才可提高电路的工作频率。可以将较大的组合逻辑分解为较小的几块，中间插入触发器，这样可以提高电路的工作频率。这也是所谓“流水线”（pipelining）技术的基本原理。

把时钟频率受制于较大的组合逻辑中延时的组合逻辑部分，通过适当的方法平均分配组合逻辑，可以避免在两个触发器之间出现过大的延时，消除速度瓶颈。

那么在设计中如何拆分组合逻辑呢，更好的方法要在实践中不断的积累，但是一些良好的设计思想和方法也需要掌握。目前大部分 FPGA 都基于 4 输入 LUT 的，如果一个输出对应的判断条件大于 4 输入的话，就要由多个 LUT 级联才能完成。这样就引入一级组合逻辑时延。要减少组合逻辑，无非就是要输入条件尽可能少。这样就可以使级联的 LUT 更少，从而减少了组合逻辑引起的时延。

“流水线”就是一种通过切割大的组合逻辑（在其中插入一级或多级 D 触发器，从而使寄存器与寄存器之间的组合逻辑减少）来提高工作频率的方法。比如一个 32 位的计数器，该计数器的进位链很长，必然会降低工作频率。可以将其分割成 4 位和 8 位的计数，每当 4 位的计数器计到 15 后触发一次 8 位的计数器，这样就实现了计数器的切割，也提高了工作频率。

在状态机中，一般也要将大的计数器移到状态机外，因为计数器这东西一般经常是大于 4 输入的，如果再和其他条件一起做为状态的跳变判据的话，必然会增加 LUT 的级联，从而增大组合逻辑。以一个 6 输入的计数器为例，原希望当计数器计到 111100 后状态跳变。现在将计数器放到状态机外，当计数器计到 111011 后产生个 enable 信号去触发状态跳变，这样就将组合逻辑减少了。

状态机一般包含 3 个模块，一个输出模块，一个决定下个状态是什么的模块和一个保存当前状态的模块。组成 3 个模块所采用的逻辑也各不相同。输出模块通常既包含组合逻辑又包含时序逻辑；决定下一个状态是什么的模块通常又组合逻辑构成；保存现在状态的通常由时序逻辑构成。因此，合理的划分这 3 个模块以及模块中的组合逻辑部分将成为设计状态机的关键。

4.5 降低面积的技巧

4.5.1 模块划分的技巧

❖ 技巧内容

本节将讲解利用模块划分以达到降低面积的技巧。

❖ 技巧详解

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

- 有利于模块的可重用性：模块设计得好，可节省大量的重复工作，并且为以后的设计带来方便。
- 在组合电路设计中应当没有层次：可提高代码的可读性，另一方面是综合的时候方便，并且时序较易满足。
- 每个模块输出尽量采用寄存器输出形式：这样设计有利于时序的满足。
- 模块的按功能进行划分，划分要合理。

- 模块大小应适中，不能太大，也不能太小，一般为 2 000 门左右。具体情况应当依据综合工具的性能而定。

模块的层次应当至少有 3 级，可将一个设计划分为 3 个层次：TOP、MID 和功能 CORE。

- TOP：包括实例化的 MID 和输入输出定义（如果用综合工具插入引脚则可不要此层次）。
- MID：由两部分组成，如分频电路和倍频电路。
- 功能 CORE：包括各种功能电路的设计。一个复杂的功能可以分成多个子功能来实现，即再划分子层。

4.5.2 复用模块的技巧

❖ 技巧内容

本节将讲解利用模块复用以达到降低面积的技巧。

❖ 技巧详解

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，首先看两个例子。

例 1，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，请大家判断一下优劣，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;
   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;
   output [15:0]  square;
   Wire  [7:0]  data_temp;

   Assign data_temp = (data_in[7])? (~data_in +1) : data_in;
   Assign square = data_temp * data_temp;
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；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 的乘法器；第二部分，后者的选择器是 1bit 判断 8Bit 输出，而前者的 1Bit 判断 16Bit 输出。

将上述两种实现方式通过综合工具综合后发现，第二种实现方式所需资源比第一种实现方式相差了一倍以上。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。其实如果拓展一个思维，将乘法器换成加法器、除法器等，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 成倍地节省前级模块所消耗的资源。

目前很多综合工具都有“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。但是，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，很多情况下不能智能地发现需要 Resource Sharing 的逻辑；第二，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不十分可靠；第三，在 ASIC 设计中，综合工具非常忠于用户意图，这时 Coding Style（编码风格）更加重要。所以工程师必须注意自己 Coding Style 方面的修养并不断提高。

4.5.3 利用代码风格降低面积的技巧

❖ 技巧内容

本节将讲解如何利用代码风格以达到降低面积的技巧。

❖ 技巧详解

代码风格有两层含义：其一是 HDL 的代码书写习惯；另一个则是对于特定电路。用哪一种形式的语言描述，才能将电路描述得更清楚，综合以后产生的电路更为合理。

代码风格有通用风格和专用风格两大类，前者指不依赖于 FPGA 开发的 EDA 软件工具和 FPGA 芯片类型，仅仅是从 HDL 语言出发的代码风格；后者指和开发软件以及硬件芯片密切相关的风格，不仅需要关注 EDA 软件在语法细节上的差异，还要紧密依赖于固有的硬件结构。显然，前者具有较好的通用性，但性能未必最优先。在使用时，如果有后续的进一步开发，建议使用通用风格；否则就可采用后者，以便极大地挖掘芯片潜力。

“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。

下面介绍利用代码风格降低面积的常用手段。

- Distributed RAM 代替 BlockRAM：在设计中如果 LUT 足够多而 BlockRAM 不够则可考虑采用 Distributed RAM 代替 BlockRAM，这种情况在设计中经常会碰到。
- Distributed RAM 代替通道计数器：这个在下面的章节已经给出详细的应用说明，这里就不再多说。
- 专有资源的利用：如进位链 MUX、SRL、乘法器等可利用 Coregen 产生宏单元。利用专有资源虽然可以降低面积，但有一缺点是降低了代码的可移植性。如果是准备转 ASIC，则需对专有资源进行代码改动，增加出错的可能性。因此，在做 ASIC 设计时采用这种方法要仔细权衡。
- 基本设计技巧：加法器处理；if 代替 case 语句；利用 LUT 四输入特点进行优化；减少关键路径上的 LUT 级数；资源共享；高效利用 IOB；采用单端口 BlockRAM。

综合性能对代码风格的要求如下。

- 资源共享的应用限制在同一个 Modul 里。这样，综合工具才能最大限度地发挥其资源共享综合作用。
- 尽可能将 critical path 上所有相关逻辑放在同一个 Module 里。这样，综合工具能够发挥其最佳综合效果。
- critical path 所在的 Module 与其他 Module 分别综合，对 critical path 采用速度优先的综合策略对其他 Module 采用面积优先的综合策略。
- 尽可能 Register 所有的 Output。做到这一点，对加约束比较方便；同时，一条路径上的组合逻辑不可能分散在各个 Module 里，这对综合非常有利。可以比较方便地达到面积速度双赢的目的。
- 一个 Module 的 size 不能太大。具体大小由各综合工具而定。

- 一个 Module 尽量只有一个时钟，或者整个设计只有一个时钟。

4.5.4 使用分布式 RAM 的技巧

❖ 技巧内容

本节将讲解如何使用 Xilinx 分布式 RAM 的一些常用技巧。

❖ 技巧详解

分布式 RAM 是 Xilinx 产品所提供的一种特殊的资源。一般地，我们将 Distributed RAM 用作同步 RAM。虽然 Xilinx 的 Virtex 系列器件里有许多 BlockRAM 可供设计者使用。但是，其资源毕竟是有限的。许多情况下，我们需要应用 Distributed RAM 来补充 BlockRAM 的不足。

由于 Distributed RAM 是采用 LUT 实现的，当 RAM 容量大到一定程度时，例如 64X32 的同 RAM，此时每个地址线驱动的基本单元都比较多，电路扇出很大，并且实现 RAM 的 LUT 比较分散，线延时也会增加许多，因此有如下建议。

- 地址线直接来自专门地址寄存器。
- RAM 的输出直接接寄存器。

这样，保证地址线的扇出较小，且按照流水线设计，可获得较高频率。

图 4.30 所示的是一个采用 Distributed RAM 实现多路（多通道）加 1 计数器，其中的地址就是通道号。这种电路经常用在这样的设计中：设计的通道比较多（例如 256），每个通道需要一个地址指针（例如 8bit）加 1 计数器。如果采用 BlockRAM，则十分浪费。

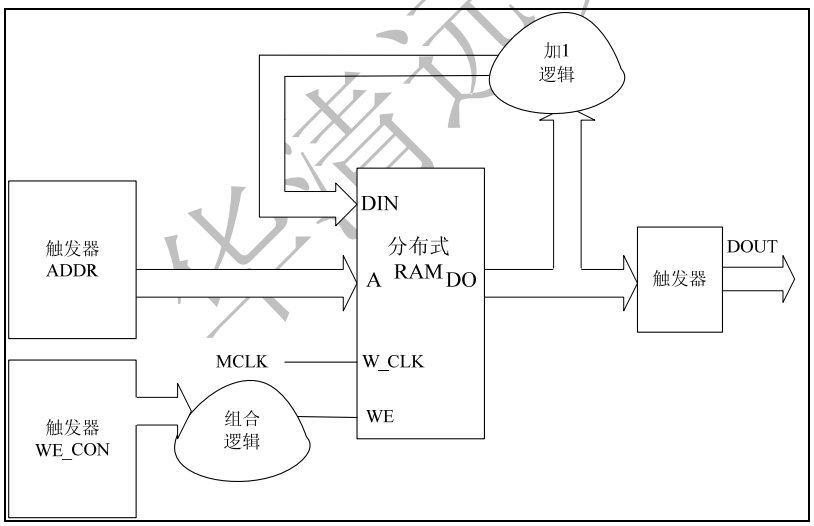

图 4.30 采用分布式 RAM 实现多路加 1 计数器

当然，是否在 Distributed RAM 前后加寄存器，要具体情况具体分析。因此，要求设计者在设计之前，必须了解所采用 RAM 最快速率是多少，以决定是否加和如何加寄存器。

注意，如果能够采用 RAM 来做计数器，会大大降低资源占用率。例如：一个 LUT 可以替代 16 个 register。

4.5.5 布局布线的技巧

❖ 技巧内容

本节将讲解如何对布局布线进行调整的注意事项和一些常用的技巧。

❖ 技巧详解

布局布线的作用是什么，已经在前面有过详细的介绍了，这里主要介绍一些注意事项和常用的技巧。

（1）设计前期（设计方案阶段），对关键电路的处理。

一个设计能否成功，关键是在设计方案阶段相关问题是否考虑完善。其中，一个非常重要的工作是确定关键路径（或者关键模块、关键算法等）能否在芯片中实现，其实现的结果如何（如速度是否满足？面积是否太大等）。

因此，要求大家在做方案时，要对所有的可能的关键路径或关键部分心中有数，而且一定要在正式开始编代码之前，要将这一部分的“评估代码”完成，并经过布局布线的检验，以考察其可实现性。从而对设计方案的风险有一个确定的认识。

（2）布局布线策略，以及如何做第一次布局布线。

一个设计想要成功完成布局布线，必需满足以下条件。

- 设计规模在芯片容量限制之内。
- 布线资源等资源不能超过芯片现有资源。
- 时序要满足要求。

目前，在进行布局布线时绝大部分设计者做法是：设定时钟约束及必要的引脚约束，选定器件，然后开始布局布线。这种做法在小规模设计时，一般可以。不过，当面对大型设计时，却往往行不通，浪费设计时间。

在这种情况下，比较好的做法如下。

- 第一次布线时，不加任何约束或者放松时钟约束：这么做的目的是，确保我所选的器件是符合容量要求的，并快速提供一个参考结果。目前，经常遇到布线几十个小时，仍然没有结果的情况。因为没有结果，也就无法利用相关的工具进行分析，只能干等。同时，也不知道这么长时间没结果是布线布不通呢？还是资源不够用？结果，白白浪费时间。
- 在第一次布线结果分析基础上，适当增加约束条件，如时钟约束。注意，时钟约束要根据上一次分析结果确定是否一步到位。约束条件如果设置不好，则会浪费大量的时间，尤其是对规模庞大的设计。因此，约束要恰到好处，既能发现关键路径不满足约束条件，又能较快地布出一个结果来。
- 如果布线结果仍不满意，则应当努力找到尽可能多的“放松”约束（TIG、Multi-Cycle-Path 等）。同时，根据实际情况，决定是否要进行设计修改。

需要注意的是，约束文件（如 ucf）只是一种微调作用。若要从根本上解决延时问题，应多从设计本身考虑。当时序实现与要求差别很大时，是不能依靠约束来解决问题。

- 正确看待 map 之后的资源占用报告：在使用 Virtex 系列进行 FPGA 设计时，经常发现 map 报告说资源利用率已经到达 100%。然而，真实情况确实如此吗？未必！

Slice 内部包含 LUT Register 和快速进位链及其他快速性能电路。其中，对资源占用影响最深的应当是 LUT 和 Register。因此，在看报告时，应当看 LUT 占用了多少，Register 占用了多少，当然，也应当看 Block RAM 占用多少，时钟资源占用多少。这些东西，才是我们下决策时要考虑的因素。

因此，在估计一个设计是否能被某个器件装下时，不能笼统地只看 Slice 使用状况。

4.5.6 面积和速度的平衡与互换技巧

❖ 技巧内容

可编程逻辑设计有许多内在的规律可循，总结并掌握这些规律对于深刻地理解可编程逻辑设计技术非常重要。本节将讲解一个基本的设计原则——面积和速度的平衡与互换原则。

❖ 技巧详解

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，不应该简单地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，能达到的工作频率远远高于设计要求，那么设计者可以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，一般的方法达不到设计频率，那么设计者可以通过将数据流串并转换，并行复制多个操作模块，对整个设计采取“乒乓操作”和“串并转换”的思想进行运作，在芯片输出模块再对数据进行“并串转换”，是从宏观上看整个芯片满足了处理速度的要求，这相当于用面积复制换来速度的提高。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）。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218