第 5 章

数字系统的高级设计与综合

从基于原理图的设计转到硬件描述语言设计是电子设计的一次变革，它允许一个设计者从理论上以工艺无关的行为方式来描述所设计的数字系统模型。随着设计要求不断提高，复杂性不断增加，用硬件描述语言的数字电路设计在很多方面已经变成单调和费时的事情，设计者迫切需要更高层次抽象的设计与综合技术，为了适应技术的发展，大量高层次的设计技术与综合工具可提供给设计工程师使用。

对于结构比较清晰的数字系统，可以利用 HDL 硬件描述语言直接在寄存器传输级（即 RTL 级）对设计的系统进行描述，这种描述是对系统行为的描述，然后由综合工具进行综合，利用硬件来实现数字系统。

在第 3 章介绍 Verilog HDL 语言的基础上，第 4 章结合 Vivado 设计软件介绍了如何将描述系统行为的设计程序进行硬件实现，并加载到目标器件进行调试和验证。由于设计者的程序的编码风格和采用的设计技术直接影响系统模型的建立和综合的结果，本章将讨论如何使编写的程序能够建立正确的系统模型，并被软件综合成设计者设想的结构，包括编码风格的影响、综合工具优化的使用，以及同步设计技术的概念和措施。最后，按照数字系统的层次结构简要介绍综合可能采用的一些方法，以便读者理解从语言描述到硬件实现的过程。

5.1 Verilog 编程风格

对于使用硬件描述语言（HDL）编程的抽象级描述，综合优化技术仅能协助设计者满足设计要求。综合工具遵循编码构造和按照 RTL 中展开的结构在最基础的层次上映射逻辑。如果没有类似 FSM 和 RAM 等十分规则的结构，综合工具可以从代码中提取功能，识别可替代的结构，并相应地实现。

除了优化之外，为综合编码时的基本指导原则是不减少功能而使所写的结构和伪指令最小化，但是这样可能在仿真和综合之间产生不一致的结果。一个好的编码风格一般要保证 RTL 仿真与可综合的网表具有相同的性能。一类偏差是厂商支持的伪指令，它可以按照专门注释的形式（不考虑仿真工具）加入 RTL 代码，并引起综合工具按 RTL 代码本身不明显的方式推演一个逻辑结构。

由于综合工具只能对可综合的语句产生最终的硬件实现，如果设计者对语言规则和电路行为的理解不同，则可能使设计描述的编码风格直接影响 EDA 软件工具的综合结果。

例如，描述同一功能的两段 RTL 程序可能产生出时序和面积上完全不同的电路，好的描述方式就是综合器容易识别并可以综合出所期望的电路，电路的质量取决于工程师使用的描述风格和综合工具的能力。

5.1.1　逻辑推理

1. if-else 和 case 结构——特权与并行性

在 FPGA 设计的范围内，把一系列用来决定逻辑应该采取什么动作的条件称作一个判决树。通常，可以分类成 if-else 和 case 结构。考虑一个十分简单的寄存器写入的示例。

例 5-1

```
  module regwrite(
  output  reg   rout,
  input         clk,
  input  [3:0]  in,
  input  [3:0]  sel);
always @(posedge clk)
    if(sel[0])    rout <= in[0];
    else if (sel[1])  rout <= in[1];
    else if (sel[2])  rout <= in[2];
    else if (sel[3])  rout <= in[3];
  endmodule
```

这类 if-else 的结构可以推理成如图 5-1 所示的多路选择器的结构。

这类判决结构可以按许多不同的方式来实现，取决于速度/面积的权衡和要求的特权。下面介绍如何针对不同的综合结构对各种判决树进行编码和约束。

if-else 结构固有的性质是特权的概念。出现在 if-else 语句的条件所给予的特权超过判决树中的其他条件。所以，上述结构中更高的特权将对应靠近链的末尾和更接近寄存器的多路选择器。

在图 5-1 中，如果选择字的位 0 被设置，则不管选择字的其他位的状态，in0 将被寄存。如果选择字的位 0 没有被设置，则利用其他位的状态来决定通过寄存器的信号。通常，只有当某一位(在此情况是最低位 LSB)前面的所有位均没有被设置时，则利用该位来选择输出。这个特权多路选择器的真正实现如图 5-2 所示。

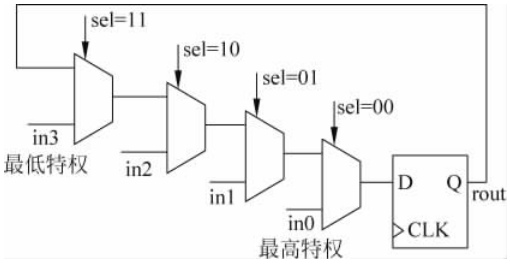


图 5-1　串行多路选择器结构的简单特权

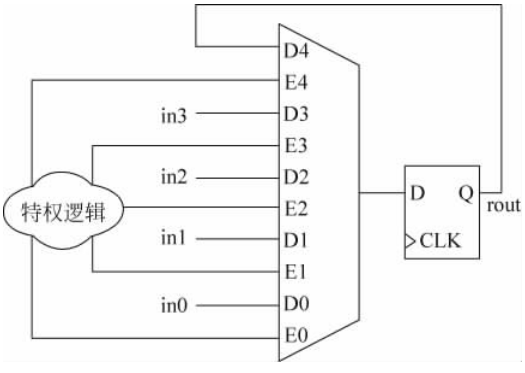


图 5-2　特权多路选择器

无论 if-else 结构最后如何实现，将赋予出现在任何给定的条件之前的条件语句更高的特权。所以，当判决树有特权编码时应该利用 if-else 结构。

另一方面，case 结构通常（不是总是）用于所有条件互不相容的情况。换言之，可以在任何时刻只有一个条件成立的情况下优化判决树。例如，根据其他多位网线或寄存器（例如加法器的译码器）进行判决时，在一个时刻只有一个条件成立。这与上述用 if-else 结构实现的译码操作是一样的。为了在 Verilog 中实现完全相同的功能，可以采用 case 语句。

例 5-2

```
case(1)
    sel[0]:  rout <= in[0];
    sel[1]:  rout <= in[1];
    sel[2]:  rout <= in[2];
    sel[3]:  rout <= in[3];
endcase
```

由于 case 语句是 if-else 结构的一种有效替代，许多初学者以为这是自动地无特权判决树的实现。对于更严格的 VHDL 语言，该想法恰巧是正确的；但是对于 Verilog 语言，却不是这种情况，可以从图 5-3 中 case 语句的实现看出。

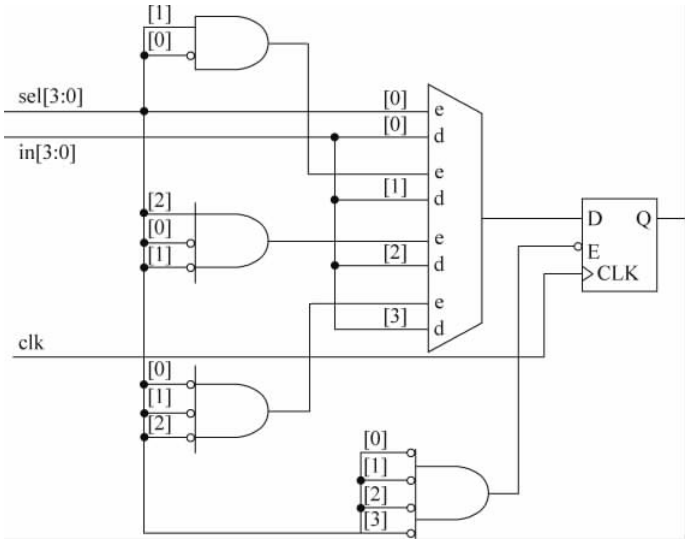


图 5-3　特权译码逻辑

如图 5-3 所示，缺省的部分是通过特权译码来设置多路选择器上相应的使能引脚，这导致许多设计者落入陷阱。如果综合工具报告 case 结构不是并行的，则 RTL 必须把它改变为并行的。如果特权条件是成立的，在相应的位置应该采用 if-else 结构。

2. 完全条件

目前为止检查的判决树中，如果 case 语句的条件没有一个是成立的，则综合工具将寄存器的输出返回到判决树作为一个默认条件（这个行为取决于综合工具默认的实现方式，但是本节假设它是成立的）。这个假设是如果没有条件满足，数值不改变。

建议设计者添加默认条件。这个默认值可以是也可以不是当前的数值，但是，为每个

case 条件分配输出一个数值，可避免工具自动地锁存当前值。用这个默认条件消除寄存器使能，如例 5-3 中修改后的 case 语句。

例 5-3

```
//不安全的 case 语句
module regwrite(
    output  reg   rout,
    input         clk,
    input  [3:0]  in,
    input  [3:0]  sel);
  always @(posedge clk)
    case(1)
        sel[0]:  rout <= in[0];
        sel[1]:  rout <= in[1];
        sel[2]:  rout <= in[2];
        sel[3]:  rout <= in[3];
        default:  rout <= 0;
    endcase
  endmodule
```

如图 5-4 所示，默认条件现在是明确的，作为多路选择器一个可供选择的输入实现。虽然触发器不再要求一个使能信号，总的逻辑资源不一定减少。同时应注意到，如果每个条件不对寄存器定义一个输出（通常发生在单个 case 语句分配多个输出时），默认条件和任何综合的特征位都不能防止产生一个锁存器。为了保证总有一个数值分配到寄存器，可以在 case 语句之前利用初始赋值分配给寄存器一个数值，如例 5-4 所示。

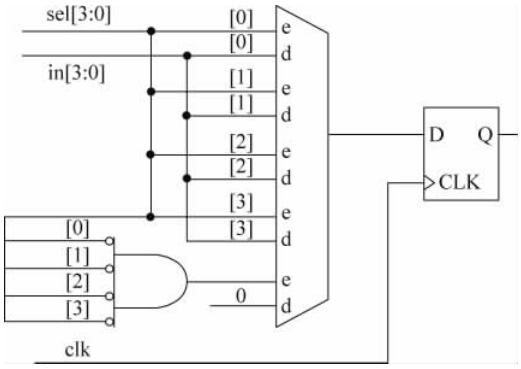


图 5-4　为默认条件编码

例 5-4

```
module regwrite (
    output   reg  rout,
    input         clk,
    input  [3:0]  in,
    input  [3:0]  sel);
  always @(posedge clk)
    rout <= 0;
    case(1)
```

```
        sel[0]:  rout <= in[0];
        sel[1]:  rout <= in[1];
        sel[2]:  rout <= in[2];
        sel[3]:  rout <= in[3];
    endcase
endmodule
```

这类编码风格消除了对默认情况的需要，也保证了如果没有其他赋值定义时，寄存器会分配到这个默认值。

完全条件可以用正确的编码方式来设计，推荐的方法是避免该约束仅由设计保证完全的覆盖，如前所述，即在 case 语句之前利用默认条件和设置默认数值。这将使代码具有更高的可移植性，减少不希望的失配可能性。设置 FPGA 综合选项时最大的风险之一是仅允许一个默认设置，因此所有的 case 语句自动地假设是 parallel_case、full_case 或二者兼之。厂商明确地提供了该选项。实际上，应该尽量不要使用这个选项，它会产生隐含的风险，以不正确的形式进行代码综合，并且用基本的系统内测试无法发现，在仿真中会产生不确定性。所以，parallel_case 和 full_case 可以引起仿真和综合的失配。

3. 多控制分支

设计者通常犯的错误（在较差的编码风格中）是对单个寄存器不连接控制分支。在例 5-5 中，oDat 被唯一的判决树中两个不同的数值赋值。

例 5-5

```
//不好的编程风格
module separated(
    output   reg  oDat,
    input         iclk,
    input          iDat1, iDat2, iCtrl1, iCtrl2);
  always @ (posedge iclk)  begin
    if (iCtrl2)   oDat <= iDat2;
    if (iCtrl1)   oDat <= iDat1;
  end
endmodule
```

因为无法说明 iCtrl1 和 iCtrl2 是否是互不相容的，因此该编码是模糊的，综合工具必须为实现做一定的假设。特别地，当二者的条件同时成立时，没有明显的方式管理特权。因此，综合工具必须基于这些条件发生的顺序赋予特权。在这种情况下，如果条件最后出现，它将获得优于第一个条件的特权。

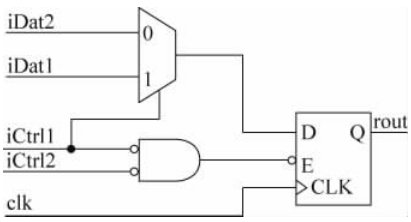


图 5-5　带明显特权的实现

基于图 5-5，iCtrl1 有优于 iCtrl2 的特权，如果交换它们的次序，特权同样也会交换。这与 if-else 结构的行为相反，if-else 结构通常把特权给予第一个条件。

所以，好的设计实践是把所有的寄存器赋值保持在单个控制结构内。

5.1.2　陷阱

在描述与可综合 RTL 结构特性有关的功能方面，行为 HDL 具有十分灵活的能力，自然地会产生大量的陷阱，当设计者不理解综合工具如何解释各种结构时可能会落入陷阱。本节识别大量的陷阱，并且讨论避免这些陷阱的设计方法。

1. 阻塞与非阻塞

在软件设计领域，按照预定的顺序执行规定的操作来产生功能。在 HDL 设计领域，这类执行可以想象为阻塞。这意味着当前的操作完成之后，进一步的操作才不被阻塞(之前它们没有被执行)。所有进一步的操作是在所有前面的操作已经完成和存储器中的所有变量已被更新的假设之下执行的。非阻塞的操作执行时与次序无关，更新是被专门的事件所触发的，当触发的事件发生时所有的更新同时发生。

Verilog 和 VHDL 等 HDL 语言提供为阻塞和非阻塞赋值的结构。如果不能理解何处和如何利用阻塞和非阻塞可能导致不期望的行为，还会使仿真和综合之间失配。例如，例 5-6 中的代码。

例 5-6

```
非阻塞模块
module blockingnonblocking(
(output   reg  out,
  input        clk,
  reg          in1, in2,in3);
  reg          logicfun;
  always @(posedge clk) begin
  logicfun <= in1 & in2;
  out    <= logicfun | in3;
  end
endmodule
```

该逻辑按照逻辑设计者预期的实现如图 5-6 所示。

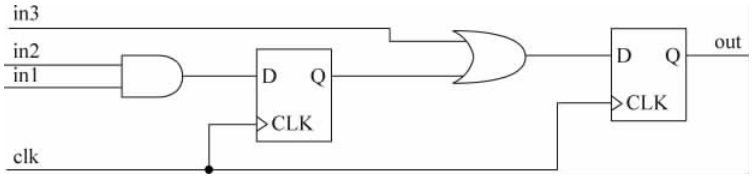


图 5-6　用非阻塞赋值的简单逻辑

在图 5-6 所示的实现中，信号 logicfun 和 out 是触发器输出，在 in1 和 in2 上的任何变化将花费两个时钟周期传播到 out。对于阻塞赋值，仅需做微小修改。

例 5-7

```
//不好的编程风格
logicfun   = in1 & in2;
out        = logicfun | in3;
```

在例 5-7 的修改中，非阻塞语句已经被改变为阻塞语句。这意味着 out 在 logicfun 更新

之后才被更新，二者的更新必须在同一个时钟周期内发生。

从图 5-7 可以看出，把赋值改变到阻塞方式，已经有效地消除了 logicfun 的寄存器，并改变了整个设计的时序。但是，并不是说相同的功能不可以用阻塞赋值来完成。考虑例 5-8 的修改。

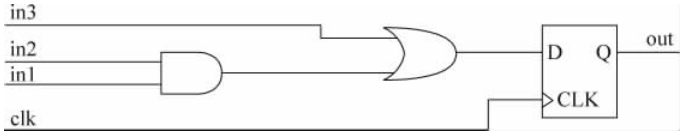


图 5-7　采用阻塞赋值的不正确的实现

例 5-8

```
//不好的编程风格
out         = logicfun | in3;
logicfun    = in1 & in2;
```

在例 5-8 的修改中，强迫 out 寄存器在 logicfun 之前更新，它迫使输入 in1 和 in2 经两个时钟周期的延时传播到 out。这将得到预期的逻辑实现，但是很少有直接的方法。事实上，对于具有相当复杂性的许多逻辑结构，这不是清晰的或者可行的方法。一个诱人的方法是为每个赋值使用独立的 always 语句。

例 5-9

```
//不好的编程风格
always @(posedge clk)
  logicfun = in1 & in2;
always @(posedge clk)
  out     = logicfun | in3;
```

尽管这些赋值被分解成看上去像并行的模块，但它们却不是这样仿真的。应该避免这类编码风格。

阻塞赋值常常出现在要求相对大量默认条件的操作的情况。在例 5-10 使用非阻塞赋值的代码中，控制信号 ctrl 定义哪个输入被赋值为相应的输出，其余的输出被赋值为零。

例 5-10

```
//不良的编程风格
module blockingnonblocking(
    output  reg  [3:0]  out;
    input               clk;
    input        [3:0]  ctrl, in);
    always @(posedge clk)
        if(ctrl[0]) begin
            out[0]  <= in[0];
            out[3:1]<= 0;
        end
        else if(ctrl[1]) begin
            out[1]  <= in[1];
            out[3:2]<= 0;
            out[0]  <= 0;
```

```
        end
        else if(ctrl[2]) begin
            out[2]  <= in[2];
            out[3]  <= 0;
            out[1:0]<= 0;
        end
        else if(ctrl[1]) begin
            out[3]  <= in[3];
            out[2:0]<= 0;
        end
        else
            out  <= 0;
    endmodule
```

在例 5-10 实现的每个判决分支中，不赋值的所有输出必须设置为零，每个分支包含单个输出(赋值为一个输入)以及三个零赋值语句。为了简化代码，阻塞语句有时候被用于初始赋值，如例 5-11 所示。

例 5-11

```
//不好的编程风格
module blockingnonblocking(
    output   reg  [3:0]  out,
    input                clk;
    input        [3:0]  ctrl, in);
    always @(posedge clk) begin
        Out                 = 0;
        if(ctrl[0])        out[0]  <= in[0];
        else if(ctrl[1])   out[1]  <= in[1];
        else if(ctrl[2])   out[2]  <= in[2];
        else if(ctrl[3])   out[3]  <= in[3];
    end
endmodule
```

在例 5-11 中，最后的赋值是“粘合”(stick)的赋值，因为设置了一个对所有输出位的初始值，需要时只改变一个输出。虽然这个代码将综合到与更复杂的非阻塞结构相同的逻辑结构，但仿真中可能出现竞争条件。非阻塞赋值可以完成相同的功能，如例 5-12 所示的类似的编码风格。

例 5-12

```
module blockingnonblocking(
    output   reg  [3:0]  out,
    input                clk;
    input        [3:0]  ctrl, in);
    always @(posedge clk) begin
        out                         <= 0;
        if(ctrl[0])         out[0]  <= in[0];
        else if(ctrl[1])    out[1]  <= in[1];
        else if(ctrl[2])    out[2]  <= in[2];
        else if(ctrl[3])    out[3]  <= in[3];
    end
endmodule
```

这类编码风格是普遍使用的，因为已经用非阻塞赋值消除竞争条件。

违反这些准则将导致仿真与综合的失配，程序可读性差，同时会降低仿真性能，较难诊断出硬件的错误。

2. for-loop 环路

类似于 C 语言的环路结构，for-loop 可能对有软件设计背景的设计者存在陷阱。与 C 语言不同，这些环路一般不可以在可综合代码中被算法迭代利用。然而，为了以最少的操作为大阵列赋值，HDL 设计者一般会使用这些环路结构。例如，软件设计者可能利用 for-loop 获得 X 的 N 次幂，如以下代码片段所示。

```
PowerX = 1;
for (i = 0; i < N; i++)  PowerX = PowerX * X;
```

该算法环路利用迭代执行 N 次乘法操作，每次通过该环路操作将变量更新。在软件中，该环路方法可以工作得很好，因为每次环路迭代中用 PowerX 的当前数值更新一个内部寄存器。

但是，可综合的 HDL 在迭代环路期间没有任何隐含的寄存器，所有的寄存器操作被清楚地定义。如果设计者试图用可综合的 HDL 以类似的方式产生上述结构，可能最后的结果看起来像例 5-13 的代码段。

例 5-13

```
//不好的编程风格
module forloop(
    output  reg  [7:0]  PowerX,
    input        [7:0]  X,  N);
    integer             I;
    always @ * begin
        PowerX = 1;
        for (i = 0; i < N; i = i + 1)
            PowerX = PowerX * X;
    end
endmodule
```

程序可以在行为仿真中工作，取决于综合工具是否可以综合到门电路。Synplify 将基于最坏条件的 N 值综合这个环路。如果它确实可综合，最后的结果将是一个环路，它完全展开成一个运行极慢的大量的逻辑块。在环路每次迭代期间管理这些寄存器可能需要控制信号，如例 5-14 所示。

例 5-14

```
module forioop(
    output   reg  [7:0]  PowerX,
    output   reg         Done,
    input                clk, Start,
    input        [7:0]   X,N);
    integer              I;
    always @ (posedge Clk)
        if (Start) begin
```

```
            Power   <= 1;
            I       <= 1;
        Done <= 0;
    end
    else if (I < N) begin
            PowerX  <= PowerX * X;
            I <= i + 1;
    end
else
    Done <= 1;
endmodule
```

在例 5-14 的设计中，幂函数是一个比“类似软件”实现运行更快的和更小的数量级。

所以，for-loop 不应该用于实现类似软件的迭代算法。

理解 for-loop 的正确使用有助于产生可读的和有效的 HDL 代码。for-loop 通常使用较短的代码形式来减少并行代码段重复的长度。例如，例 5-15 中的代码取 X 的每一位与 Y 的偶数位进行异或操作产生一个输出。

例 5-15

```
out[0] <= Y[0] ^ X[0];
out[1] <= Y[2] ^ X[1];
out[2] <= Y[4] ^ X[2];
…
out[31] <= Y[62] ^ X[31];
```

以逻辑形式代码写 out 可能需要 32 行，需要反复地输入。为了压缩代码且更具有可读性，可采用 for-loop 来重复每一位的操作。

例 5-16

```
Always @ (posedge  Clk)
    For(i = 0; i<32; i = i+1)  out[i] = y[i*2] ^ X[i];
```

从例 5-16 可以看出，环路中没有反馈机构，for-loop 可以用来压缩类似的操作。

综合工具处理循环的方法是重复循环内的结构，在循环中包含不变化的表达式会使综合工具花费很多时间优化冗余逻辑。

例 5-17

```
//不好的编程风格
for( I = 0; i<4; i = i+1)   begin
    sig1 = sig2;           //不变化的表达式
    data_out(I) = data_in(I);
end
```

例 5-18

```
//正确的编程风格
sig1 = sig2;               //不变的表达式
for( I = 0; i<4; i = i+1)
    data_out(I) = data_in(I);
```

例 5-19

```
xor_reduce_func = 0;
for (I = N-1; I >= 0; I = I-1)
xor_reduce_func = XOR_reduce_func ^ data[I];
```

例 5-19 中的循环语句可综合成如图 5-8 所示的链状结构，对于长的组合链路，设计者应该在代码编写阶段就注意描述成树状结构，如例 5-20 所示，综合得到如图 5-9 所示的结果。

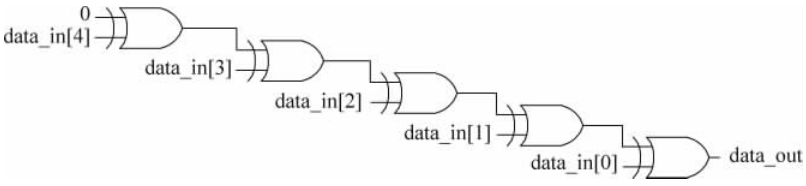


图 5-8　链状结构

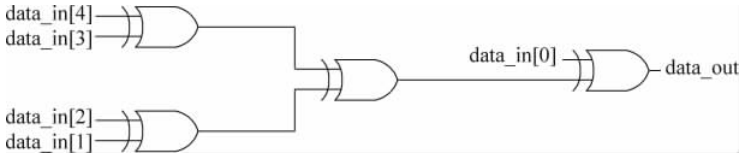


图 5-9　树状结构

例 5-20

```
i_data = data;
LEN = i_data`LENGTH;
if LEN = 1 then
    result = i_data(i_data`LEFT);
elsif LEN = 2 then
    result = i_data(i_data`LEFT) XOR i_data(i_data`RIGHT);
else
    MID = (LEN + 1)/2 + i_data`RIGHT;
    UPPER_TREE = XOR_tree_func(i_data(i_data`LEFT downto MID));
    LOWER_TREE = XOR_tree_func(i_data(MID-1 downto i_data`RIGHT));
    result = UPPER_TREE XOR LOWER_TREE;
end if;
```

3. 组合环路

组合环路是包含反馈的逻辑结构，其中没有任何的同步元件。从图 5-10 可以看出，当一组组合逻辑的输出不带中间寄存器反馈回自身时会出现组合环路。这类行为很少遇到，一般表示为设计和实现中的一个错误，这里要讨论可能产生这样一个结构的陷阱和如何避免它们。考虑以下的代码段。

例 5-21

```
//不好的编程风格
Module combfeedback(
```

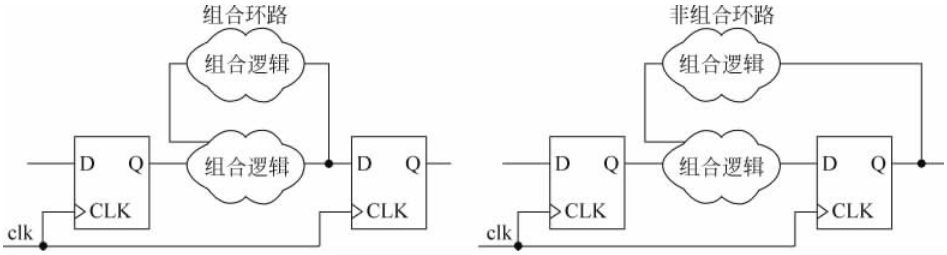


图 5-10　组合与时序环路

```
    output   out,
    input    a);
    reg      b;
//不好的编程风格:会将 b 反馈回 b
    assign out = b;
//不好的编程风格: 不完整的敏感信号列表
    always @(a)
    b = out ^ a;
endmodule
```

例 5-21 的模块表示一个行为描述,在仿真中它可能表现如下：当导线 a 改变时,输出被赋值为当前输出和 a 异或的结果。输出只在 a 改变时改变,不呈现任何反馈或振荡的行为。但是,在 FPGA 综合中,一个 always 结构描述了寄存器或组合逻辑的行为。在这种情况下,综合工具将扩展敏感清单(当前只包含 a),使敏感清单包含假设结构为组合的所有输入,当这些发生时,反馈环路关闭,将通过一个反馈到自身的异或门来实现,如图 5-11 所示。

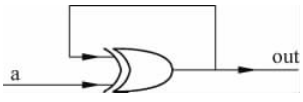


图 5-11　偶然的组合反馈

这类结构是有很大问题的,因为输入 a 为逻辑 1 的任何时刻它将振荡。例 5-21 列出的 Verilog 语句描述了一个编码风格很差的电路,设计者显然没有硬件的概念,在仿真和综合时将看到惊人的失配。作为好的编码实践,所有的组合结构应该编码,使得在 always 模块内的表达式中所包含的全部输入都列在敏感清单中。如果做到这点,在综合之前就能检测出问题。

4. 寄存器与锁存器

寄存器与锁存器都是用来暂存数据的器件。

寄存器是时钟沿触发的,输出端通常不随输入端的变化而变化,只在时钟上升沿(或下降沿)才将数据打入寄存器,输入端的数据送至输出端；锁存器是电平敏感的,只要使能信号有效,输出端总随输入端的变化而变化；相反地,使能信号无效时,输出保持已有的数值。

锁存器比寄存器快,有些场合适合使用锁存器,但是一定要保证输入信号的质量。锁存器的缺点是对输入信号的毛刺敏感,锁存器在 ASIC 设计中比寄存器简单,但是在 FPGA 的资源中,很多器件需要用查找表和寄存器来组成锁存器,浪费逻辑资源。锁存器在时序分析中也比较困难。程序中使用不完整的 if 语句结构或 case 语句结构,将导致综合软件综合出锁存器。所以要避免不必要的锁存器推论。

专门类型的组合反馈实际上可以推论出时序元件。例 5-22 以典型的方式模拟一个锁存器模块。

例 5-22

```
//锁存器接口
module latch (
    input    iClk, iDat,
    output  reg  oDat);
    always @ *
        if(iClk)   oDat <= iDat;
    endmodule
```

每当控制插入时,输入直接传递到输出;当控制释放时,锁存器被禁止。一个很常见的编码错误是产生组合的 if-else 树,忽略了对每个条件定义输出。实现时会包含一个锁存器,通常将指示一个编码错误。

对于 FPGA 设计,一般不推荐使用锁存器,但可以使用这些器件设计和执行时序分析。注意也有其他方式可偶然地推论出锁存器,更多的情况是无意的。在例 5-23 的赋值中,默认条件是信号本身。

例 5-23

```
//不好的编程风格
assign O = C ? I: O;
```

一些综合工具将推论出一个锁存器,而不是推论一个反馈到其输入之一的多路选择器(它不可以任何方式预测)。附带的问题是一个时序的终点(锁存器)被插入进一个路径,其中可能会有设计中介的时序元件。这类锁存器推论一般表示 HDL 描述中的一个错误。

对于 FPGA 设计,一般不推荐的锁存器可以十分容易地变成不正确的实现或完全不实现。例如通过使用函数调用实现,考虑锁存器封装进一个函数的典型例示,如例 5-24 所示。

例 5-24

```
//不好的编程风格
module latch (
    input     iClk, iDat,
    output  reg oDat);
always @ *
    oDat <= MyLatch(iDat, iClk);
    function D, G;
        if (G)   MyLatch = D;
    endfunction
endmodule
```

在这个示例中,输入到输出的条件赋值被放进一个函数,尽管锁存器的表示似乎精确,但是函数将总是判定组合逻辑,会把输入直接传递到输出。

敏感列表只对前面的仿真起作用,对综合器不起作用,而综合后生成的仿真模型是保证不遗漏敏感信号的,因此设计者不小心造成的敏感信号表的遗漏往往会导致前、后仿真的不一致。

引起硬件动作的信号应该都放在敏感信号列表中,包括:①组合电路描述中,所有被读取的信号;②时序电路中的时钟信号、异步控制信号。

例 5-25

```
//不完整的敏感信号列表
always @(d or clr)
```

```
    if (clr)
        q = 1'b0
    else if (e)
       q = d;
```

例 5-26

```
//完整的敏感信号列表
always @(d or clr or e)
    if (clr)
        q = 1'b0
    else if (e)
       q = d;
```

概括上面的分析，对于编程风格，列出以下注意要点或需要遵循的准则：

(1) 对希望形成组合逻辑的 if-else 和 case 语句，要完整地描述其各个分支，避免形成锁存器，一个可行的办法是在语句前为所有被赋值信号赋一个初始值。

(2) 有大量关于阻塞和非阻塞赋值为综合编码时广泛接受的准则：①利用阻塞赋值设计组合逻辑模型；②利用非阻塞赋值设计时序逻辑或混合逻辑模型。

(3) 从不把阻塞和非阻塞赋值混合在一个 always 模块中。

(4) 尽量使用简单的逻辑、简单的数学运算符。

(5) 进程的敏感列表应该列举完全，否则可能产生综合前后的仿真结果不同和引入锁存器的现象，可用 * 替代来自动识别全部敏感变量。

(6) 在循环中不要放置不随循环变化的表达式。

(7) 对于复杂的数学运算要充分进行资源共享，如采用 if 块等。

(8) 对于长的组合链路应该在代码编写阶段就注意描述成树状结构。

(9) 时序逻辑尽可能采用同步设计。

(10) 对具有不同的时序或面积限制的设计，应尽可能采用不同的代码描述以达到不同的要求(一般而言，面积与延时是相互冲突的)。

(11) 对于复杂系统设计，应尽量采用已有的算法和模块实现。

5.1.3　设计组织

所有与工程师队伍一起工作设计过大型 FPGA 的人都理解把设计组织成有用的功能约束，以及为重用和扩展做设计的重要性。在顶层上组织一个设计的目标是产生一个容易在模块基础上管理的设计，产生可读和可重用的代码，产生一个允许设计缩放的基础。本节讨论一些影响可读性、可重用性和综合效率的结构设计要考虑的内容。

1. 分割

分割是指依据模块、层次和其他功能约束组织设计。一个设计的分割应该预先考虑，因为设计组织的主要变化在设计进展时将变得更困难和更昂贵。设计者可以方便地围绕一部分功能交换他们的想法，允许他们以有效的方式设计、仿真和诊断自己的工作。

一般情况下，一个设计应按功能分割成较小的功能单元，每个功能单元都有一个公共的时钟域，并能独立进行验证。设计层次应能将时钟域分割开，以说明多个时钟之间的相互作用和对同步电路的需求，每个时钟域的逻辑在系统整合之前分别进行验证。

1）数据通道与控制

在第1章中对数字系统结构的分析可知，数字系统的许多结构可以分割成数据通道和控制结构的形式。数据通道一般是一个把数据从设计输入端运送到输出端并对数据执行必要操作的“管道”。控制结构通常不对设计数据处理或运送，但是为各种操作配置数据通道。按照数据通道和控制结构之间的逻辑分割，可以逻辑地将数据通道和控制结构设置在不同的模块，清楚地对各个设计者定义接口。对不同的逻辑设计者，这样不仅方便地划分设计活动，而且也可以优化可能要求的下游活动。

所以，*数据通道和控制结构应该分割成不同的模块。*

因为数据通道常常是设计的关键路径（设计的流量是与流水线的时序有关的），可能要求为这个通道达到最大性能设计布图。另一方面，通常将较慢的时序要求安排给控制逻辑，因为它不是主要的数据通道的一部分。

例如，通常用简单的SPI（串行外设接口）或I^2C（Intel-IC）类型总线来设置设计中的控制寄存器。如果流水线运行在几百兆赫兹，在两个时序要求之间将确实有大的偏差。因此，如果布图是应对数据通道要求的，控制逻辑的布图则可以保持空间无约束和散布地围绕流水线的合适的位置，像自动布局布线工具所实现的那样。

2）时钟与复位结构

好的设计实践表明，任何给定的模块只有一个类型的时钟和一个类型的复位。例如在许多设计案例中，当有多个时钟区域和（或）复位结构时，重要的是分割层次使得它们被不同的模块分开。由于混合时钟和程序描述复位类型会带来设计的风险，但是如果任意给定模块只有一个时钟和复位，这些问题几乎很少出现。

所以，*在每个模块中只利用一个时钟和一个类型的复位是好的设计实践。*

3）多个例示

如果有些逻辑操作在特定的模块中出现不止一次（或者横跨多个模块），则应自然地分割设计，把整块分成分开的模块，并放进多个例示的层次中。

图5-12描述的分割有许多优点。首先，整块的功能分配给相互独立的设计者，使设计更方便。一个设计者可以集中在顶层设计、组织和仿真，而另一个设计者可以集中于子模块的功能性技术要求。如果接口定义明确，这类分组设计可以获得更好的效果。但是，两个设计者在相同的模块内开发，可能发生更大的混淆和困难。此外，子模块可以在设计的其他区

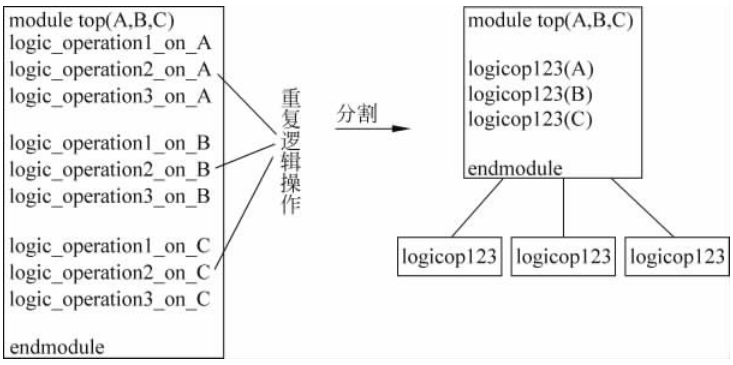


图5-12　模块设计

域重用，或者完全在不同的设计中。一般重新例示一个存在的模块更方便，而不是切割和连接更大的模块，并重新设计接口。

采用这样的策略可能引起的一个问题是各个模块的数据宽度、迭代次数等有细微的变化。这些案例强调参数化的设计方法，类型相似的模块可以共享公共的代码基，即在例示基础上参数化。下一节将更详细地讨论。

2. 参数化

在 FPGA 设计的范围内，参数是一个模块的特性，它可以在全局的意义上或者在每个例示的基础上改变，同时保持模块的基本功能。本节描述参数的形式，以及介绍它们如何加强综合的有效编码。

1）定义

参数和定义是类似的，在许多情况下可以相互交换使用。但是，在许多情况下指有效的、可读的和模块化设计。定义一般用于规定横跨所有模块的恒定的全局数值，或者为相容或不相容的部分代码提供编译时间的伪指令。在 Verilog 中，定义使用`define 语句，编译时间控制用一系列的`ifdef 语句。全局定义可定义全部设计的常数，如例 5-27 所示。

例 5-27

```
`define CHIPID 8`hC9              //全局芯片 ID
`define onems   90000             //使用 11ns 时钟近似的 1ms
`define ulimit16   65535          //一个 16 位无符号字的上限值
```

上面列出的定义是全局的“明确事情”的例子，从一个子模块到另一个子模块将不会发生改变。全局定义的另一个用途是为代码选择规定的编译时间伪指令。一个广泛的应用是 FPGA 中 ASIC 样机的使用。ASIC 和 FPGA 之间的不同常常需要对设计有细微地修改（特别是 I/O 和全局结构），例如，考虑例 5-28 中的定义。

例 5-28

```
`define FPGA
//`define ASIC
```

在顶层模块中，可能有例 5-29 的输入。

例 5-29

```
`ifdef ASIC
input TESTMODE;
output TESTOUT;
`endif
`ifdef FPGA
output DEBUGOUT;
`endif
```

在上面的代码例子中，为插入 ASIC 测试必须包含测试引脚，但是在 FPGA 实现中这样做并没有意义。因此，设计者只会包含那些在 ASIC 综合中的位置支架。类似地，设计者可能需要为 FPGA 样机诊断使用某个输出，但是却不包含在最后的 ASIC 实现中。全局定义允许设计者用行中包含的变化保持单个代码载体。所以，ifdef 伪指令应该为全局定义使用。

为了保证在全局意义上应用定义，并且不与另一个全局定义冲突，推荐编写一个可以包

括所有设计模块的全局定义文件。因此,任何全局参数都可以在集中的位置修改使其改变。

2) 参数

与全局定义不同,参数一般位于专门的模块,从一个例示到另一个例示可以改变,一个广泛应用的参数是尺寸或总线宽度,如例 5-30 的寄存器的例子中所示。

例 5-30

```
module paramreg #(parameter WIDTH = 8) (
    output  reg   [WIDTH - 1:0]  rout;
    input                        clk,
    input         [WIDTH - 1:0]  rst);
    always @(posedge clk)
        if (!rst)   rout <= 0;
        else        rout <= rin;
    endmodule
```

例 5-30 描述了一个具有可变宽度的简单参数化的寄存器。虽然参数的默认值是 8,但是可以只为这个例示修改宽度。例如,在更高层次中的模块可以例示例 5-31 所示的 2 位寄存器。

例 5-31

```
//正确,但是过时的参数传递
paramreg #(2)  r1(.clk(clk), .rin(rin), .rst(rst), rout(rout));
```

或者以下 22 位的寄存器:

```
//正确,但是过时的参数传递
paramreg #(22)  r2(.clk(clk), .rin(rin), .rst(rst), rout(rout));
```

从上面的例示可以看出,对于 paramreg,相同代码的载体可用来例示两个有不同特性的寄存器,同时注意到模块的基本功能在例示(寄存器)之间没有改变,只改变功能的专门特性(尺寸)。

所以,*参数应该被局部定义使用,从一个模块到另一个模块才改变。*

当要求类似功能的不同模块有细微的不同特性时,像这样的参数代码是十分有用的。如果没有参数化,设计者可能需要为相同模块编写冗长的代码载体,并且修改差异特性时容易出错。

上述参数定义可在 Verilog 中利用 defparam 命令替代,允许设计者在设计层次中规定任何参数。容易出现的问题是因为一般的参数是在专门的模块例示中使用,在特定例子的外部是看不到的(类似软件设计中的局部变量),容易混淆综合工具,产生仿真失配。所以,如果使用 defparam,应该将其包含在与定义的参数对应的模块例示中。

3) Verilog-2001 中的参数

参数改进的方法在 Verilog-2001 中引入。在 Verilog 的旧版本中,参数值的传递是含义模糊的,或者很难通过位置参数传递来读出,使用 defparam 会引起前述讨论的一些风险。理想情况下,设计者应该按照与模块的 I/O 之间传递信号相似的方式将一个参数值清单传递到模块。在 Verilog-2001 中,参数可以通过模块外部的名称识别,从而可消除可读性的问题以及使用 defparam 产生的风险。例如,使用 paramreg 的例示将包含参数的名称修改。

```
paramreg #(.WIDTH(22))  r2(.clk(clk), .rin(rin), .rst(rst), routy(rout));
```

这样可以不锁定位置要求，增加代码的可读性，减少人为错误的概率。这类参数化的命名是极力推荐的。所以，命名参数传递优于位置参数传递或 defparam 语句。

在 Verilog-2001 中，参数化另一个主要的改进是 localparam。localparam 是局部变量的 Verilog 参数版本。localparam 可以通过其他参数的表达式推导出，被处在其内的模块的实际例示所约束。例如，考虑例 5-32 中参数化的乘法器。

例 5-32

```
//混合方式参数
module multiparam #( parameter WTDTH1 = 8, parameter WTDTH2 = 8)
    localparam             WIDTHOUT = WIDTH1 + WIDTH2;
    output  [WIDTH - 1:0]  oDat;
    input   [WIDTH - 1:0]  iDat1;
    input   [WIDTH - 1:0]  iDat2;
    always oDat = iDat1 + iDat2;
endmodule
```

在上面的例子中，需要外部定义的参数只是两个输入的宽度。因为设计者假设输出的宽度是输入宽度之和，因此这个参数可以从输入参数推导出来，没有冗余的外部计算。这使得设计者的工作更方便，消除了输出尺寸与输入尺寸之和不匹配的可能性。

通常，在模块的头部不支持 localparam，因此，如果在 I/O 清单中使用 localparam，端口清单必须额外说明(Verilog-1995 方式)。只要 localparam 可以从其他输入参数推导出来，就推荐使用，因为它将进一步减小人为错误的可能性。

5.1.4　针对 Xilinx FPGA 的 HDL 编码

由于没有一个完美的方式来产生设计，所以推荐使用编码技术的设计准则。对于 Xilinx FPGA 的架构，可以减少器件的使用，改善设计性能，使实现工具获得更好的布局和布线结果，进而得到更好的系统速度。

作为满足时序的设计策略，建议尽可能地使用专用资源，例如由 LUT 实现的移位寄存器 SRL、大量的 DSP 和 BRAM 等，达到提高速度、降低功耗和改善器件利用率的目的。

在可能的情况下，可以利用硬模块映射大的寄存器阵列的优点，SRL、BRAM 和 DSP Slice 可以有效地用于节省逻辑的 Slice 资源并改善性能。例如，可以利用 BRAM 实现有限状态机等。

7 系列 FPGA 的查找表 LUT 是 6 输入的，因此与 4 输入 LUT 相比，可以封装更多的逻辑。特别是当设计需要从 4 输入 LUT 架构优化的代码移植到 7 系列器件时，可能需要对设计进行重新编码，才能减少设计中流水线的级数，从而充分利用有效的资源。

当设计中 FPGA 占用比较满时，必须尝试不同的设计策略。如果没有将设计封装进器件，时序又无关，为减少设计中 CLB 资源的数量，可以使用更多的 DSP 和 BRAM 资源替代 CLB 资源。Xilinx 建议关闭 Logic Replication 的综合选项，可以帮助减少设计尺寸。

在某些情况下，优化面积设计可以改善性能，特别是在 RTL 级优化时，可以降低功耗。

1. 控制集

在 2.4 节中，图 2-19 SliceL 电路图中的圈 3 和圈 5 是八个触发器，每个触发器具有时钟 CK、时钟使能 CE 和同步/异步的置位/复位 SR 等三种控制端口，八个触发器共享相同

的时钟和控制信号，而时钟使能是四个触发器共享的高电平有效的信号。

驱动一个寄存器的这些控制信号被称为“控制集”。由于设计的编程风格和综合工具不同，设计可能有许多控制集。设计软件会以最优的性能智能地封装逻辑。

通常，Xilinx FPGA 对于每个设计提供足够的寄存器，但是在用户的设计中，某些时候寄存器不足，一些设计实践可能限制寄存器的有效利用。由于控制信号的限制，对于分组的 Slice 资源，在 CLB 模块中的资源对布局也有一定影响。FPGA 不推荐使用低电平有效的控制信号，因为低电平的控制信号不节省功耗，所以设计中低电平有效的控制信号的不合理使用也会造成资源的不足。

带有不同控制集的触发器不可以封装进相同的 Slice 中，但是通过把控制信号映射到 LUT 资源中，软件可以按指令减少控制集的数目，如图 5-13 所示，设计中不同的三个控制集，原来要映射到 3 个 Slice 中实现，但是使用同步的置位和复位，可以利用 LUT 将三个不同控制集变为一个相同的控制集，其中同步置位 Sset 与输入信号进行或运算，同步复位 SReset 与输入信号进行与运算，控制信号仅保留时钟信号 CK，这种映射增加了 LUT 的利用率，降低了 Slice 的利用率，节省了资源。

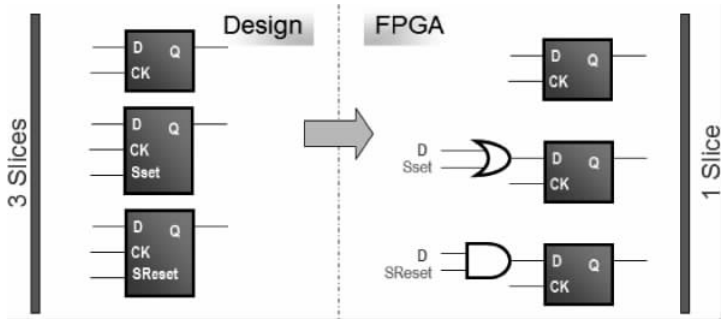


图 5-13　三个不同控制集变为一个相同的控制集

2. 控制信号设计技巧

控制端口的使用规则是时钟和异步的置位/复位信号总是占用触发器的控制端口，不可采用 LUT 的等效逻辑将它们移到数据通道；时钟使能和同步置位/复位信号与 Slice 中的大多数触发器共享相同的控制集时，它们将连接到触发器的控制端口，但是可以移到 LUT 输入端的数据通道。所以异步的置位/复位信号比同步的置位/复位信号有更高的特权来获取触发器的控制端口。

Xilinx 建议设计中要尽可能地利用同步置位和复位，也建议使用高电平有效的 CE 和置位及复位，并且尽量构造具有尽可能少的控制信号的设计。

3. 其他设计技巧

1) I/O 寄存器用法

IOB 模块的寄存器提供固定的建立时间和时钟至输出的时间 T_{CO}，这是捕获器件外的输入数据和时钟数据的最快的方式，但是，由于采用了 IOB 寄存器内部的时序，通道可能变得更长，导致内部逻辑的布线延时、加长。

Xilinx 建议只在必须满足 I/O 时序时使用 IOB 寄存器。

可选择性地利用 XDC 属性或 TCL 指令把寄存器移进 IOB：

```
set property IOB TRUE [get_cells b1_reg[ * ]]
set property ILOGIC_X0Y341 [get_cells b1_reg[0]]
```

每个寄存器例示的名称可以从综合工具的原理图视图或时序报告中获得。

2) Block RAM 的用法

为了得到 Block RAM 的最佳性能，可利用 IP 库工具的帮助例示存储器，避免使用 Block RAM 存储器的“写之前读”(read before write)模式，

Synplify 和其他第三方综合工具可以插入旁路逻辑防止 RTL 和硬件行为之间可能存在的失配错误。当读和写操作发生在相同的存储器地址时，强制 Block RAM 输出一个已知的数值。当确认不会出现失配错识时，可以不添加这个逻辑以免损害相关属性的性能。

```
attribute syn_ramstayle of mem: signal "no_rw_check"
```

3) 时钟使能

利用 HDL 代码对时钟使能的控制，只有当需要时才使用。如果需要低扇出的 CE，可利用独特网线的综合属性来利用控制信号或模块级的控制信号。

例 5-33

```
always @ (posedge CLK)
    If (CE = `1`)
        Q <= A;
```

例 5-33 的程序中，设计软件会将控制信号 CE 映射到相应的控制端口，但是对于低扇出的时钟使能信号，可以利用替代的编码方法，将控制信号 CE 映射到 LUT 的输入端，而不占用触发器的控制端口，如例 5-34 所示。

例 5-34

```
always @ (posedge CLK)
    Q <= (~CE & A)| (CE & Q);
```

Xilinx 不建议设计异步或不带 CE 的程序，应合理地采用低扇出 CE 的设计。

为了降低整个时钟域的功耗门控，利用时钟使能全局缓冲资源 BUFGCE 和 BUFHCE，对于暂时的小面积上的时钟的应用，可利用寄存器的时钟使能引脚。

5.2　综合优化

大多数为 FPGA 综合的实现工具都为设计者提供了很多优化选项，但是大多数设计者在运用中往往不清楚这些选项的准确功能，以及如何利用这些优化选项来优化一个设计。许多设计者都没有完全理解这些优化选项，通常会花费几小时、几天甚至几周时间来进行无尽头地组合，直到似乎找到一个最好的结果。只有少数设计者能恰好达到优化目的，甚至超过已有的方案。因此，对于许多设计者来说，由于缺少基本的理解，大多数优化选项变得无用，也很难开发一个完全的试探法的算法库。

本节介绍综合优化最重要的方面，基于尝试而成功的真实经验，为读者在实现层次提供实际的探索。

5.2.1 速度与面积

大多数综合工具都允许设计者在速度与面积优化之间转换。这似乎是轻而易举的：想要使设计运行得更快，就选择优化速度；想要使设计更小，就选择优化面积。实际上却并不简单，因为一些算法可能产生适得其反的结果，即在要求运行更快之后设计却变得更慢。所以，必须理解速度和面积优化在实际设计中是如何实现的。

在综合层次，速度和面积优化决定了实现 RTL 将要采用的逻辑拓扑。在抽象层次，很难知道 FPGA 有关的物理特性。目前的讨论与布局和布线的互连延时有关。综合工具采用所谓的导线负载模型，基于各种设计准则统计地估计互连延时。在 ASIC 中，设计者是可以理解这点的，但是使用 FPGA 设计时这些是隐藏在幕后的。综合工具最终得到的估计值，常常与实际的结果有显著的不同。由于缺少后端的知识，综合工具主要执行门级优化。在高级 FPGA 设计工具中，基于布局的综合流程会帮助关闭这个环路。

基于综合的门级优化包括状态机编码、并行与交错多路选择、逻辑复制等。作为一般的准则(虽然不总是一定成立)，更快的电路要求更高的并行性，相当于使用面积更大的电路，即速度与面积权衡的基本概念。但是，由于存在 FPGA 布局配置的二阶效应，并不总是能获得预期的效果。

直到布局布线完成后，工具才真正知道器件在布局布线过程的困难。但是，实际的逻辑拓扑已经被综合工具提交，因此，如果在综合级致力于优化速度，后端工具将发现器件过于拥挤。当器件过于拥挤，工具将别无选择，而将元件布局到适合的地方，而次优路径会引入更长的延时。实际上设计者常常出于经济的考虑使用尽可能小的 FPGA，导致一般的直观推断：*当资源利用率接近 100%时，在综合级进行速度优化不一定产生更快的设计。事实上，面积优化实际上可以获得更快速的设计。*

过分约束设计也会出现问题，如果目标频率设置太高(比最后的速度大 15%～20%)，设计可以按次优的方式实现，实际获得较低的最大速度。在实现初期，综合工具基于时序要求产生逻辑结构。如果在初始时序分析阶段，确定设计离达到的时序太远，工具可能提早放弃。但是，如果将约束设置到正确的目标，不高于最后的频率 20%(假设没有对时序进行初始化)，逻辑将用达到规定时序的最小面积来实现，在时序收敛期间有更多的灵活性。同时需要注意，由于 FPGA 实现的二阶效应，更小的设计能否改善时序与实际的环境有关。

在 FPGA 设计中，面积优化实际上是对 FPGA 的资源利用率的优化，设计要在各种资源利用之间达到一种平衡，最大限度地发挥器件的性能。尽量使用器件中的专用硬件模块(如 RAM、硬件乘法器)，这些硬件资源能够提高设计性能，并且它们已经存在于 FPGA 中，不用就浪费了。如果专用硬件模块不够用，而逻辑单元资源丰富，则可以用逻辑资源去实现这些硬件模块。DSP、I/O 单元中都有专用的触发器资源，应尽量使用，可以提升设计性能，减少可编程逻辑资源的消耗。表 5-1 给出了 Xilinx FPGA 的设计中从设计代码优化和软件设置选择等方面进行速度和面积优化的一些方法。优化设计代码要求深入理解整个设计，每个设计都有独特的地方，对设计进行优化时，需要充分理解设计的特点和需求(例如性能、成本、开发周期等)。而通过软件对设计进行约束和设置，目的是进行编译之后，可以根据编译结果对设计进行分析，找到设计中的真正瓶颈，从而有效地引导后续的优化过程。

表 5-1　速度和面积优化的方法

	速 度 优 化	面积(资源)优化
设计代码优化	对最高工作频率进行优化，最根本、最有效的方法是对设计代码的优化： (1) 编码风格直接影响组合逻辑的级数； (2) 按目标 FPGA 器件的结构特点编写代码； 几种速度优化方法： (1) 增加流水线级数； (2) 组合逻辑分割和平衡； (3) 复制高扇出的结点； (4) 状态机仅完成控制逻辑的功能； (5) 模块边界用寄存器	设计代码优化： 设计中最根本、最有效的优化方法是对设计输入(如 HDL 代码)的优化，与编码风格有关，常用的面积优化方法： (1) 模块的时分复用； (2) 改变状态机的编码方式； (3) 改变模块的实现方式
软件设置选择	综合软件中，进行“速度优化”设置； 状态机编码方式，使用 One-Hot； 使用全局信号，全局网络具有高扇出、低抖动等优点，而且可节省布线资源； 展平层次结构，可以使模块边界的时序路径充分优化； 不同的随机数种子，导致编译结果的小幅度变动； 使用逻辑锁定进行局部优化； 利用位置约束、手动布局和反标注来优化设计	逻辑综合对设计实现的结果影响很大，选择合适的综合约束条件和编译选项很重要； 可以设置“资源优化”或“面积优化”相关选项： (1) 较少复制逻辑； (2) 设置资源共享； (3) 状态机编码方式设置； (4) 展平设计的层次结构； (5) 用专用硬件资源(如 DSP 块)代替逻辑单元功能模块； (6) 网表面积优化； (7) 寄存器打包
	最高时钟频率优化： 当一个设计的 I/O 时序满足要求后，就可以优化设计的运行速度，即内部的最高工作频率； 这个最大的时钟频率由关键路径决定	资源重新分配： 专用硬件模块与可编程逻辑资源的平衡； 专用硬件资源间的平衡(如 Xilinx FPGA 中有分布式和 Block RAM，容量不同，适合不同应用目标)

5.2.2　资源共享

在编码风格上，只有在同一个条件语句(if-else 和 case)的不同分支中的算术操作才能资源共享。结构性资源共享可以采用不同功能模块的部分设计通过调整逻辑被重用。在高层次，这类结构可以显著地减少整个面积，如果操作不是互不相容的，可能包含流量的损失。在综合优化层次上，资源共享一般指在寄存器级之间对逻辑进行分组操作。这个较简单的结构可以归结为简单逻辑和经常的算术运算。

支持资源共享的综合引擎将识别互不相容的类似的算术运算，通过调整逻辑来组合这些运算。例如，考虑例 5-35 中的例子。

例 5-35

```
module addshare (
    output   oDat,
    input    iDat1, iDat2, iDat3,
```

```
    input    isel);
    assign   oDat = isel ? iDat1 + iDat2; iDat1 + iDat3;
endmodule
```

在上面的例子中，输出 oDat 要么被前两个输入之和赋值，要么被第一个和第三个输入之和赋值，这取决于选择位。这个逻辑的直接实现如图 5-14 所示。

在图 5-14 中，两个和是独立计算的，基于输入 iSel 来选择。这是从代码直接映射的，但不是最有效的方法。有经验的设计者将识别出输入 iDat1 是在两个加法运算中使用，用输入端多路选择输入 iDat2 和 iDat3 可以采用单个加法器。

这个结果也可以通过综合提供的资源共享选项获得。资源共享将两个加法操作识别为两个互不相容的事件。取决于选择位的状态(或其他条件运算符)，不是这个加法器被更新，就是另一个被更新。综合工具则能够组合这些加法器并实现多路选择器控制输入，如图 5-15 所示。

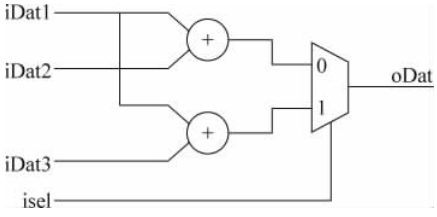


图 5-14　两个加法器的直接实现

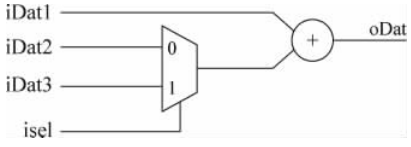


图 5-15　组合加法器资源

虽然上述实现的最大延时不受资源共享优化的影响，但有些情况资源共享要求在各个路径附加多路选择。考虑例 5-36。

例 5-36

```
    module addshare(
    output          oDat,
    input           iDat1, iDat2, iDat3,
    input  [1:0]    isel);
    assign  oDat = (isel == 0) ? iDat1 + iDat2;
                   (isel == 1) ? iDat1 + iDat3;
                   iDat2 + iDat3;
endmodule
```

直接的映射将产生如图 5-16 所示的结构。这个实现通过并行结构来产生所有的加法器和选择逻辑。最坏条件的延时是通过一个加法器和一个多路选择的路径。由于资源共享使能，加法器输入如图 5-17 所示被组合。在这个实现中，全部加法器已经减少到带多路选择输入的单个加法器。但是，应注意到关键路径被扩展成三层逻辑，会影响这个路径的时序，使时序不仅与被实现的逻辑规定有关，同时也与 FPGA 的可用资源有关。

当某路径不是关键路径时，智能的综合工具一般将利用资源共享，也就是运算中触发器到触发器的时序路径不是最坏情况。如果综合工具有这个能力，则总是会采用资源共享；如果不是，设计者必须分析关键路径，了解该优化是否会增加附加的延时。

如果资源共享被激活，则要验证没有添加延时到关键路径。

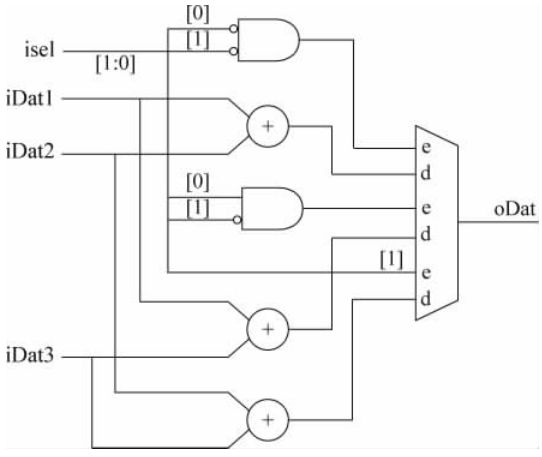


图 5-16　直接映射三个加法器

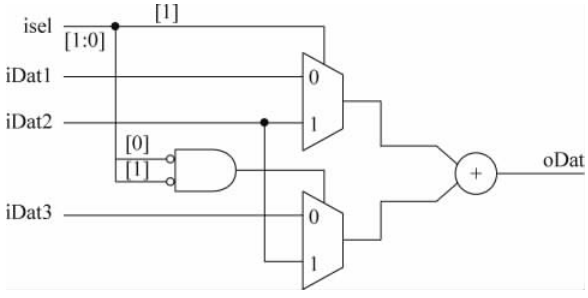


图 5-17　加法器共享时额外的逻辑层次

5.2.3　流水线、重新定时和寄存器平衡

在速度优化的设计中，流水线是一种方法，它在成组的逻辑之间添加寄存器级数，用于增加流量和触发器到触发器的时序。一个好的设计模块通常可以增加附加的寄存器级数流水，只影响总的时滞和小的面积损失。综合对相同结构的流水线、重新定时和寄存器平衡的操作进行选择，但不添加或移去寄存器本身。相反，围绕逻辑移动寄存器的优化可平衡任何两个寄存器级之间延时，从而使最坏条件下的延时最小化。流水线、重新定时和寄存器平衡在手段上是十分类似的，不同的厂商之间也只有细微的改变。图 5-18 从概念上进行了说明。

一般认为流水线起源于最广泛采用的负载平衡的方法，只要流水存储器或乘法器等规则结构可以被综合工具识别，就可以用重新分布逻辑来重新构造。在这个情况下，流水线要求规则的流水存在，并且该流水容易被工具重新组织。例如，例 5-37 中的代码定义一个可参数化的流水线乘法器。

例 5-37

```
module multpipe #(parameter width = 8; parameter depth = 3) (
    output   [2 * width - 1: 0]   oProd,
    input    [width - 1: 0]       iIn1, iIn2,
    input                         iClk);
    reg      [2 * width - 1: 0]   ProdReg  [depth - 1: 0],
```

```
    integer                     I;
    assign   oProd   = ProdReg[depth-1:0];
    always @ (posedge iClk) begin
        ProdRey[0]    <= iIn1 * iIn2;
        for (i=1;I<depth;i=i+1)
            ProdReg[i]    <= ProdReg[i-1];
    end
endmodule
```

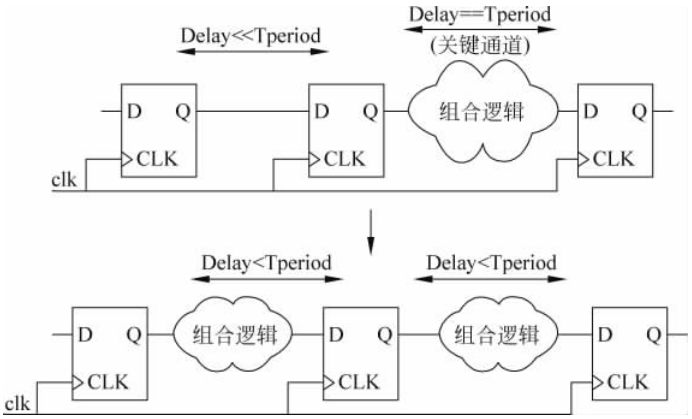


图 5-18　平衡组合逻辑

在例 5-37 的代码中，两个输入简单地相乘在一起，并由 parameter depth 定义的寄存器级寄存。没有自动流水线的直接映射将产生如图 5-19 所示的实现。

在图 5-19 所示的例子中，只有一个寄存器插进乘法器作为输出寄存器（由乘法器模块的数字 1 表示），其余的流水寄存器以不平衡的逻辑保留在输出端。启动流水线，可以把输出寄存器放进乘法器，如图 5-20 所示。符号中的数字 3 表示寄存器有三层内部的流水线。

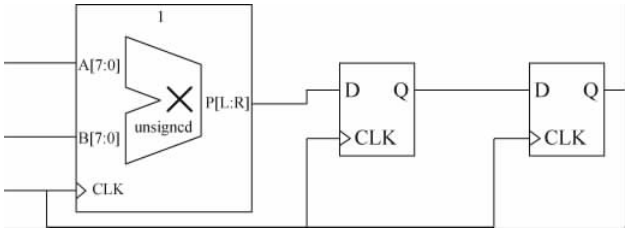


图 5-19　流水线在后的乘法器

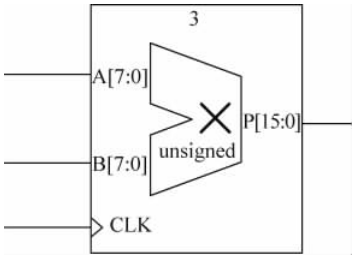


图 5-20　流水线移进乘法器

重新定时和寄存器平衡一般指的是更通常的情况，一个触发器围绕逻辑移动，同时对外部世界保持相同的逻辑功能。一般情况通过以下的示例说明。

例 5-38

```
module genpipe (
    output   reg  oProd,
    input    [7:0]  iIn1,
    input           iReset,
    input           iClk);
```

```
reg      [7: 0]  inreg1;

always @ (posedge iClk)
    if (iReset)   begin
        inreg1 <= 0;
        oProd <= 0;
    end
    else    begin
        inreg1 <= iIn1;
        oProd <= (inreg1[0] |inreg[1] & inreg[2] | inreg[3]) &
                 (inreg1[4] |inreg[5] & inreg[6] | inreg[7]) ;
    end
endmodule
```

综合运行是准确的寄存器和寄存器配对，产生如图 5-21 的实现。

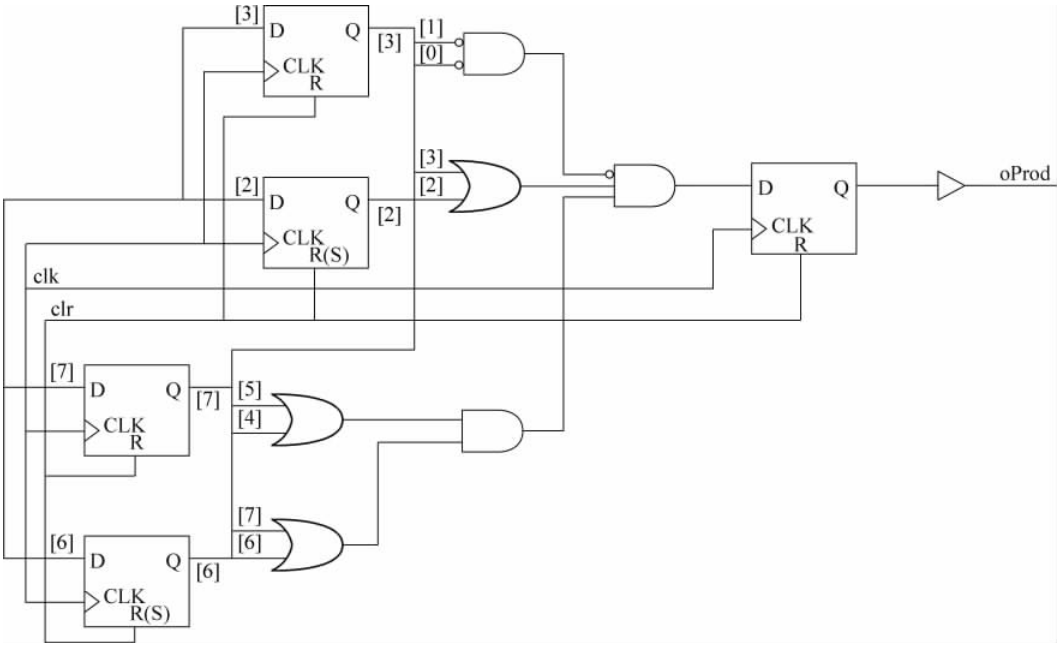


图 5-21　不平衡的逻辑

图 5-21 中，所有逻辑包含在代码中表示的两个寄存器级之间。如果启动寄存器平衡，把寄存器移进关键路径逻辑可以改善整个时序，如图 5-22 所示。

从图 5-22 的线路图可以看出，当利用寄存器平衡时，就时滞或流量而论没有损失。寄存器利用率可能增加或减少取决于应用，运行时间将会扩充。因此，如果为了时序一致，没有必要重新定时，智能的综合工具对非关键路径不执行这些操作。

寄存器平衡不应该应用于非关键路径。

1. 复位对寄存器平衡的影响

当有许多其他优化时，复位可能直接影响综合工具利用寄存器平衡的能力。如果要求用两个触发器组合来平衡逻辑负载，这两个触发器必须有相同的复位状态。例如，一个是同步复位，另一个是异步复位，或者一个是置位，另一个是复位，则二者是不能组合的，寄存器平衡不会有效。如果实际中存在这种情况，实现时应避免图 5-21 所示的连线（第 2 位和

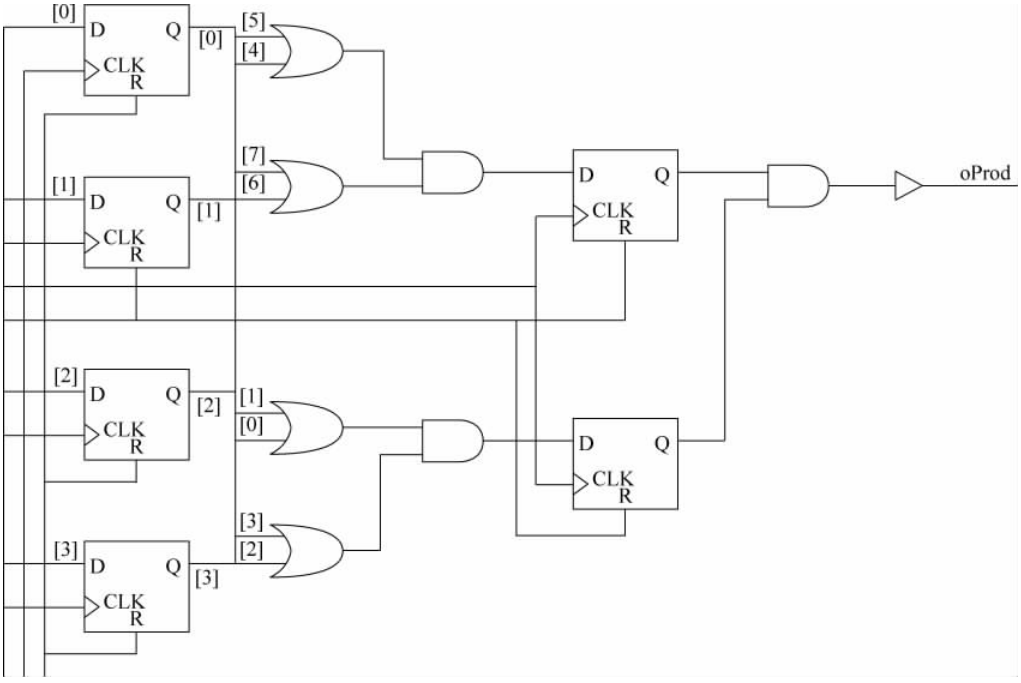


图 5-22　平衡的逻辑

第 6 位触发器置位端口接到置位端 S)。在这个实现中,驱动逻辑门的寄存器初始化为交替的 1-0-1-0 模式,可防止任意的寄存器平衡或由于不兼容的寄存器类型的重新组合。新式的综合工具可以分析路径,并通过相应的触发器输入和触发器输出一起反转复位类型从而改善整个时序。但是,该操作会引入延时从而打破寄存器平衡。综合工具使用直接映射并不会提供显著的优化效果。

所以,带有不同复位类型的相邻触发器可以阻止寄存器平衡发生。

2. 重新同步寄存器

寄存器平衡会在信号重新同步的区域引入问题。为了重新同步一个来自 FPGA 外部或另一个时钟域的异步信号可以采用双触发器的方法,如图 5-23 所示。双触发器的方法将在“同步设计中的异步问题”一节中详细讨论。

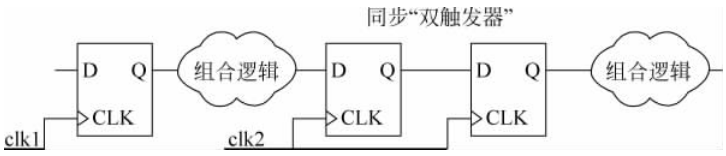


图 5-23　没有平衡的重新同步寄存器

如果启动寄存器平衡,跟随重新同步寄存器的逻辑可能推到这些寄存器之间,如图 5-24 所示。

通常不希望对潜在的准稳态信号执行任何逻辑操作,并且要提供尽可能多的时间使信号变成稳定,所以保证寄存器平衡不影响这些专门的电路十分重要。如果启动寄存器平衡,必须分析这些电路以保证对重新同步不存在影响。大多数综合工具有此能力限制设计防止

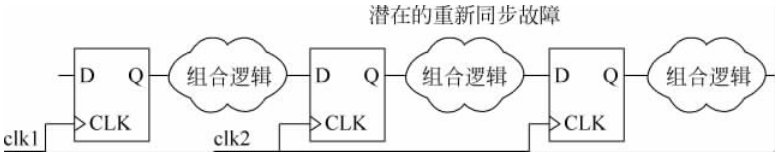


图 5-24　平衡应用于重新同步寄存器

对单个寄存器的寄存器平衡。

所以，要约束重新同步寄存器使得它们不受寄存器平衡的影响。

5.2.4　有限状态机编译

有限状态机(FSM)编译是指在 RTL 级自动识别有限状态机，并需要为速度/面积约束重新编码。这意味着只要使用标准的状态机结构，RTL 级是否准确编码并不重要。通过使用标准方式编码的状态机的规则结构，综合工具可以方便地提取状态传输和输出关系，并变换状态机使给定的设计和一组约束更加优化。

用标准编码的方式设计状态机，可以被综合工具识别和重新优化。

二进制和顺序编码将与状态表示中的所有触发器有关，因此状态解码是必需的。丰富的逻辑以及为译码逻辑设计多个输入门的 FPGA 技术将最佳地实现这些状态机。

对每个状态设置一个唯一的位，可实现一个有效(one-hot)编码。通过这个编码，不用使用状态译码，状态机通常运行更快。缺点是一个有效(one-hot)编码一般要求更多的寄存器。通常使用格雷码替代一个有效(one-hot)编码，因为其具有两个特点：①异步输出；②低功率器件。

如果状态机的输出或者状态机操作的任何逻辑是异步的，通常最好使用格雷码。由于异步电路不能防止竞争条件和毛刺，因此在状态寄存器中两位之间的路径不同可能引起不可预测的行为，并且该行为与布局配置和寄生参数有关。考虑图 5-25 所示的 Moore 机的编码输出。在这个例子中，单个位被清除和单个位被置位的地方将发生状态转移事件，因而产生潜在的竞争条件。说明该情形的波形表示如图 5-26 所示。

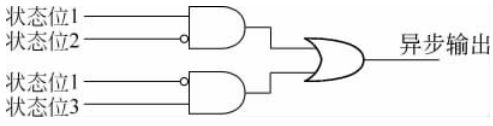


图 5-25　Moore 机输出

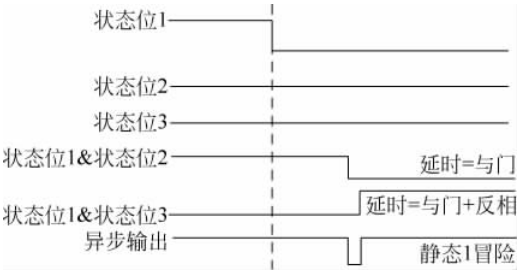


图 5-26　潜在的冒险

该问题的一个解决办法是采用格雷码。格雷码的任何转换只经历一个单个位的改变。在分析编码方案的结构之后，我们就能理解格雷码可以用来安全地驱动异步输出。为了构造格雷码，采用如下描述的镜像添加序列。

(1) 用一个“0”和一个“1”垂直排列开始。

(2) 镜像从底部数字开始编码。

(3) 添加“0”到代码的上半部(在镜像操作中这段被复制)。

(4) 添加“1”到代码的下半部(在镜像操作中这段被产生)。

这个序列在图 5-24 中说明。如在图 5-27 中可以看到,格雷码对每次状态转换只经历一个单个位的反转,因此消除在异步逻辑内的竞争条件。

所以,当驱动异步输出时使用格雷码。

除了上述情形外,FPGA 设计最好选择一个有效编码。因为 FPGA 有丰富的寄存器,不要求译码逻辑,通常也更快。因此,大多数状态机将用一个有效编码替代,所以为了减少运行时间,建议用一个有效编码设计所有的状态机。

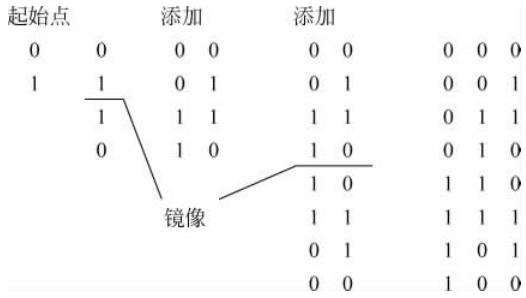


图 5-27 产生格雷码

大多数状态机编译器会去除无用的状态,并且智能到足以检测和去除不可达到的状态。对于大多数应用,这对优化速度和减少面积非常有帮助。要求保留不可达到的状态的主要应用是航空、军事或航天等领域要求的高可靠电路。由于几何尺寸极小,太阳系或核事件辐射的粒子可能引起触发器自发地改变状态。如果发生这种情况,在对人类生命有重要影响的电路中,确保寄存器状态的任何组合能够快速恢复路径十分重要。如果有限状态机中的状态没有被完全考虑,则可能把电路推进不可恢复的状态。因此,综合工具有“安全模式”来覆盖所有的状态,甚至能覆盖通过正常的操作不可达到的状态。

例 5-39 模块包含一个简单的有限状态机,在复位之后连续地顺序通过三个状态,模块的输出即状态本身。

例 5-39

```
    module safesm(
        output   [1;0]  oCtrl,
        input           iClk, iReset);
        reg      [1:0]  state;
//将状态赋值给 oCtrl
        assign oCtrl = state;
        parameters   STATE0 = 0,
                     STATE1 = 1,
                     STATE2 = 2,
                     STATE3 = 3,
    Always @ (posedge iClk)
        If (!iReset)  state = STATE0;
        else   case(state)
                     STATE0:   state <= STATE1;
                     STATE1:   state <= STATE2;
                     STATE2:   state <= STATE0;
                endcase
    endmodule
```

可通过如图 5-28 所示的移位寄存器简单地实现。注意到,如果位 1 和位 2 同时被错误地设置,这个错误将持续地循环,直到下一次复位才产生一个正确的输出。但是,如果启动

安全模式，该事件会引起一个立即的复位，如图 5-29 所示。

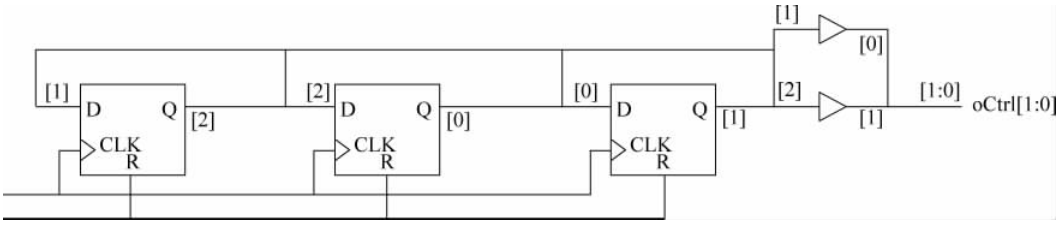


图 5-28　简单的状态机实现

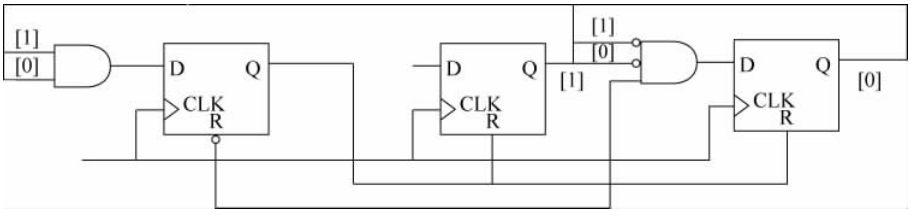


图 5-29　用安全模式状态机实现

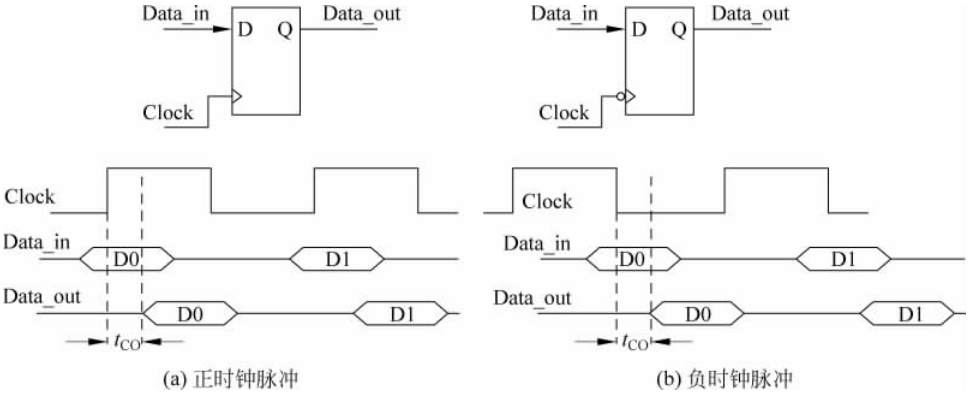
对于图 5-29 的实现，附加逻辑将检测不正确的状态，并迫使状态寄存器回到复位值。

5.3　数字系统的同步设计

数字系统由分层嵌套的有限状态机构成，而有限状态机在时钟信号的控制下才能发生状态的变化，所以同步设计成为数字系统设计遵循的主要设计原则。

5.3.1　同步设计基本原理

数字系统的输入时钟是周期信号，它控制同步器件中的全部时序特性。正时钟脉冲的脉冲上升沿形成由 0 到 1 的瞬时转换，而负时钟脉冲的瞬时转换是 1 到 0 的下降沿。图 5-30(a)和图 5-30(b)分别表示上升沿和下降沿触发寄存器的时序关系，寄存器的数据输入是 D，数据输出是 Q，定义时钟 Clock 到 Q 端的时间为 t_{CO}，它是寄存器固有的时钟输出延时。



(a) 正时钟脉冲　　(b) 负时钟脉冲

图 5-30　正负时钟脉冲

通过用统一的时钟控制一系列的逻辑门和寄存器等实现对数字系统的操作，称为同步逻辑。

同步逻辑保证所有的寄存器在同一时刻翻转，能够提高系统的整体性能，同步电路的分析也相对容易。为了保证时钟信号到达每个寄存器的延时一致，图 5-31 所示的典型时钟分配网络提供高速低偏移的时钟分配，分配时钟信号必须考虑以下几点：

（1）从参考时钟到存储元件只允许一个路径存在；

（2）不主张利用时钟分频器；

（3）不允许时钟信号的任何逻辑组合；

（4）只有在无竞争冒险的方式下选通时钟信号。

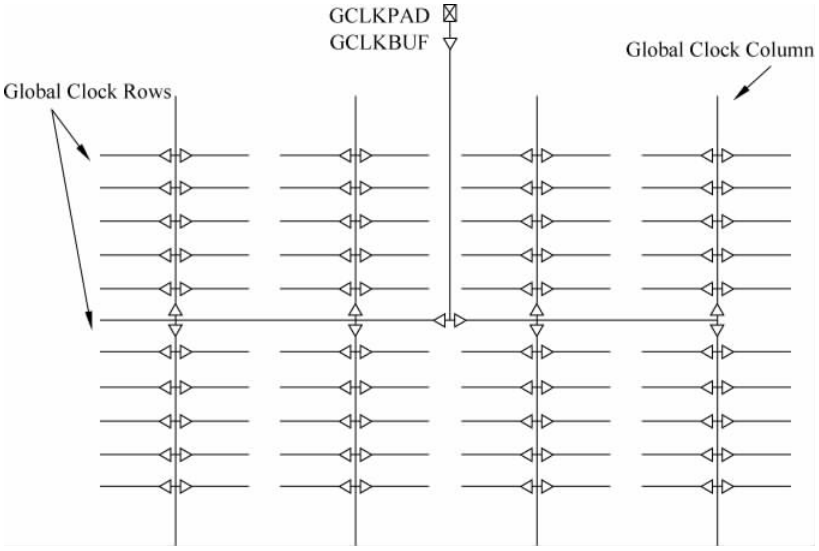


图 5-31　典型的时钟分配网络

包括触发器和逻辑门等在内，所有的逻辑器件都有一定的响应时间，任何器件在输入和输出之间都有一定的传输延时，同时由于多个门电路连接在一起，因此导致延时进一步增大。同步时序分析就是分析同步电路中的多种延时是如何组合的，以及对整个系统运行速度的影响。

在系统实现之前，综合过程利用的任何延时数值只是一个估计，元件产生延时的四个来源如图 5-32 所示：由输入信号上升和下降时间确定的转换率；门电路固有的 RC 负载确定的门延时；通过引线的传播延时和门电路驱动的 RC 负载延时；扇出负载增加了驱动器必须放电和充电的电容。这些部件产生的延时分量是系统进行时序分析的基础。

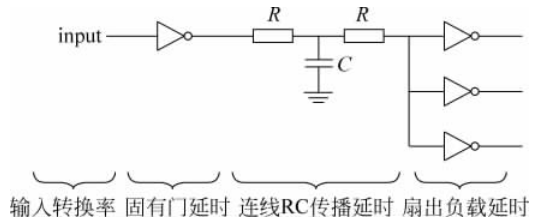


图 5-32　元件产生延时的四个分量

时钟将时间分成离散的时间间隔，每个间隔是单个时钟周期的持续期。从时序分析的角度来看，由于每个上升沿触发一个新的状态，每一个时钟周期等同于上一个时钟周期，因此同步时序分析就是考虑相继上升沿之间的一个时钟周期内电路的延时。

5.3.2　建立和保持时间

除去前面提到的触发器输出相对于时钟沿的延时时间 t_{CO} 之外，时序分析的基本参数还有输入建立时间 t_{SU} 和输入保持时间 t_H，如图 5-33 所示。

（1）输入建立时间 t_{SU} 是时钟沿到来前数据信号在输入端达到稳定需要的最小时间；

（2）输入保持时间 t_H 是时钟沿到达后数据信号在输入端保持稳定需要的最小时间。

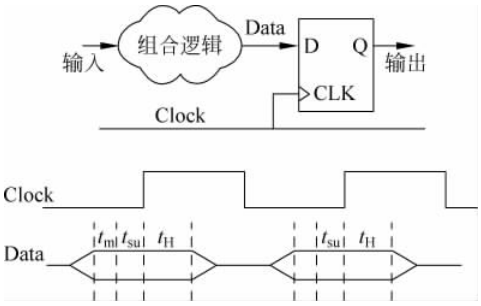


图 5-33　定义建立时间和保持时间

第 4 章指出数字系统设计的功能和性能在同步设计中是不相关的，为了达到要求的性能，必须提供时序约束文件，以使软件能够有实施的依据，并通过静态时序分析和时序分析报告，最终确定实现的设计是否达到期望的性能要求。

静态时序分析要计算每个静态时序通道在最坏条件下的建立时间和保持时间裕量，如果在考虑工艺、电压和温度的最坏条件下能够满足要求，则表示设计的时序收敛，达到性能要求。同步设计中的静态时序通道由起始定时元件启动，经过由组合电路的元件和网线构成的数据通道，到达捕获数据的目的定时元件，如图 5-34 所示。

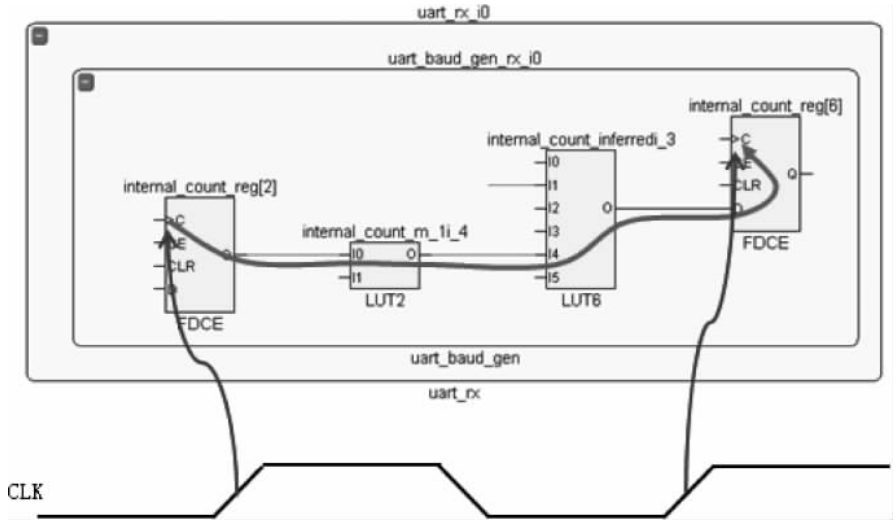


图 5-34　同步设计中的静态时序通道

按照 UltraFast 设计方法学中 Baseline 部分的建议，XDC 的时序约束包括首要的时钟约束、I/O 约束和时序例外约束三个主要部分。FPGA 片内的静态时序通道受到时钟周期的约束，所以要设置时钟信号。高质量的时钟信号允许更快的系统运行速度。但是抖动

(jitter)、偏移(skew)和持续期失真(Dute Cycle Distortion)降低了时钟的质量。

1) 时钟偏移

时钟信号的偏移是由时钟信号到达寄存器的时间不同而引起的,时钟的偏移造成电路的建立时间和保持时间被打破,对系统的影响是数据错误和输入/输出的时序容限减少。

前面对系统最高工作频率的推算是在假设源和目的触发器由同一时钟信号控制的条件下,但是各个触发器在时序上仍会存在一定的偏差,引线延时的差别是导致偏差的主要原因。

时钟偏移是在多个输入时钟沿之间的时序差,但是单个时钟源驱动许多个负载时也引起时钟偏移,此时需要多个时钟驱动器,每个驱动器之间电气特性会有小的变化,从而导致时钟偏移,并造成同步电路工作频率降低。

2) 时钟抖动

时钟信号的抖动是由时域的噪声信号引起的,抖动造成数字电路时序的预定容限的减少以及数据位的变形,破坏了电路的技术条件,对系统的影响是带宽减小、元件失效。

与时钟偏移一样,时钟抖动也使输入建立时间和保持时间的分析变坏,在确定电路实际的运行裕度时,必须从时间裕度的计算中减去时钟抖动。当运行频率增加时,时钟抖动因为占时钟周期和触发器时序要求的比例增大而变得更为严重。时钟抖动的要求是变化的,十分敏感的系统要求高质量的时钟电路来减少时钟抖动。

3) 持续期失真

时钟信号的持续期失真是指时钟信号的脉冲宽度的减少,以致时钟沿失去效用,造成电路无法对数据进行捕获,对系统的影响同样是数据错误和输入/输出的时序容限减少,关键的问题是数据是否在两个沿上读取。

1. 静态时序通道的建立和保持时间校验

同步电路设计中,静态时序通道的起始寄存器开始启动数据发送的时钟沿称为启动(launch)沿,目的寄存器"捕获"数据的时钟沿称为捕获(capture)沿。如图 5-35 所示,在进行建立时间的校验时,起始定时元件的启动沿与目的定时元件的捕获沿相差一个时钟周期。而且考虑主时钟信号从时钟输入引脚分别到起始和目的定时元件路径上不同的延时和偏移,所以要计算三条路径上的延时。其中,源时钟延时(Source Clock Delay)是从时钟输入端口到起始定时元件路径上的延时;数据通道延时(Data Path Delay)是起始和目的定时元件之间组合电路产生的延时;目的时钟延时(Destination Clock Delay)是从时钟输入端口到目的定时元件路径上的延时。

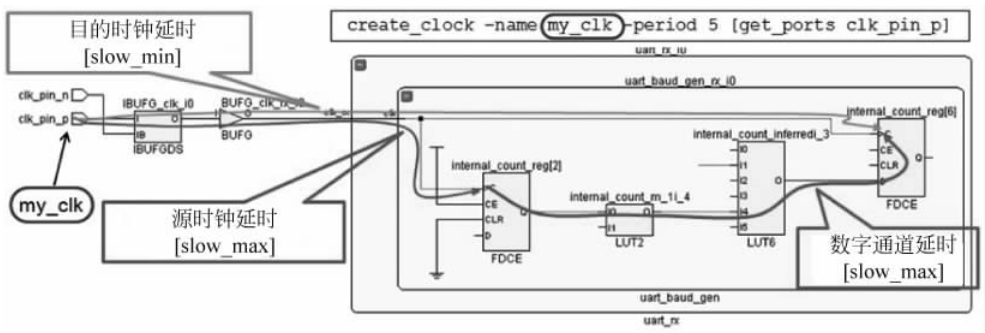


图 5-35　三条路径的延时

源时钟延时和数据通道延时之和称为数据的到达时间(Arrival Time),建立时间校验时取两条路径上受工艺、电压和温度等参数影响产生延时的最大值(slow_max),目的时钟延时加上建立时间起始沿和捕获沿间隔的一个时钟周期为数据的要求时间(Required Time),此路径上的延时要取最小值(slow_min),如果最小值的要求时间减去最大值的到达时间得到的建立时间裕量合格,则能够保证其他好于最坏情况的建立时间裕量也合格。

静态时序通道的建立时间的时序报告实例见第 4 章图 4-70“时序摘要报告”。

所以时序裕量(Slack)表示设计是否满足时序要求,正值表示满足时序要求,负值表示不满足时序要求。

建立时间的时序裕量为

Setup Slack = Smallest Data Required Time − Largest Data Arrival Time

对保持时间校验,起始定时元件的启动沿与目的定时元件的捕获沿是同一个沿。源时钟延时和数据通道延时得到的到达时间要取路径上延时的最小值,目的时钟延时要取路径上延时的最大值,保持时间的裕量是最小值的到达时间减去最大值的要求时间。

保持时间的时序裕量为

Hold Slack = Smallest Data Arrival Time − Largest Data Required Time

同步电路设计中,如果保持时间的裕量不合格,设计在硬件中不可能运行起来,但是建立时间裕量不合格,设计在硬件中有可能降频运行起来,但是达不到要求的运行速度。

在 Vivado 设计软件中,默认认为所有的时钟是相互有关的,不管源时钟和目的时钟是如何定义的,默认要对所有静态时序通道进行建立和保持时间的校验。因此要通过约束告知软件哪些是无关的时钟可以不做分析。对于不同时钟的起始沿和捕获沿,要取两个沿最紧的一对,一旦确定执行建立时间检验,对于保持时间也取两个沿最紧的一对做校验。

2. 输入和输出端口的建立和保持时间校验

输入端口的静态数据通道起始定时元件在 FPGA 外部,要计及外部定时元件的 Tco 和外部路径上的延时 Trce_delay,后者是数据通道的一部分,如图 5-36 所示,利用 TCL 指令 set_input_delay 设置外部上游器件到输入端口的延时,对建立时间的校验,Tco 和 trce_delay 都要取最大值,而对保持时间的校验,二者要取最小值。

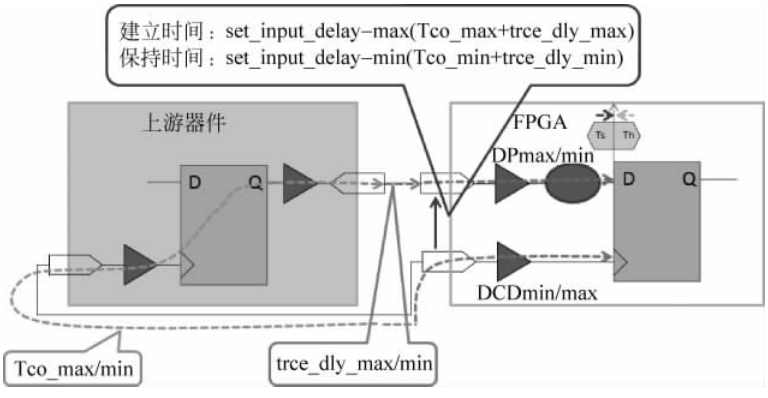


图 5-36　输入端口的时序约束

与输出端口有关的静态数据通道目的定时元件在 FPGA 外部，要计及外部定时元件的建立时间 Tsu 和保持时间 Thd，以及数据通道一部分的外部路径上的延时 Trce_delay，如图 5-37 所示，利用 TCL 指令 set_output_delay 设置外部输出端口到下游器件的延时，对建立时间的校验，取下游器件的 Tsu 和 trce_delay 最大值，而对保持时间的校验，取下游器件的 Thd 和外部 trce_delay 最小值。

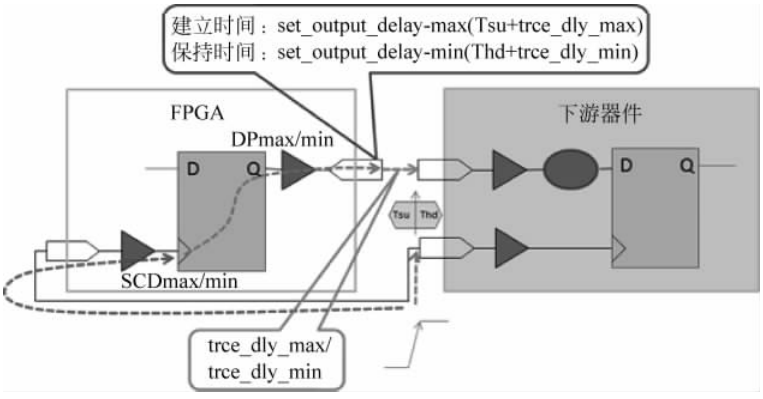


图 5-37　输出端口的时序约束

在输出端口的约束中，-min 后出现负数值，表示最小的延迟情况下，数据是在时钟采样沿之后还能保持的时间。同样地，-max 后的正数，表示最大的延迟情况下，数据是在时钟采样沿之前就到达的时间。

对于输入和输出端口的上游和下游器件的驱动时钟有两种情况，一种是上下游外部器件利用与 FPGA 相同的系统时钟，称为系统同步；另一种是时钟信号和数据一起同步输送到 FPGA，板卡上的引线延时可以视为 0，称为源同步。

在源同步的情况下，与数据同步传输的时钟信号到达 FPGA 后，可以直接驱动 I/O 端口的寄存器，或者经过 MMCM 进行频率的变换之后再驱动寄存器。

系统同步接口(System Synchronous Interface)的构建相对容易，以输入端口为例，上游器件仅传递数据信号到 FPGA 中，时钟信号则由系统主时钟来同步。同源的时钟信号在板卡的引线延时也要对准，数据传递的性能受到时钟在板卡上的引线延时和偏移以及数据路径延时的双重限制，无法达到更高速的设计要求，所以大部分情况也仅仅应用 SDR 方式。

为了改进系统同步接口中时钟频率受限的问题，设计了一种针对高速 I/O 的同步时序接口，在发送端将数据和时钟同步传输，在接收端用时钟沿脉冲来对数据进行锁存，重新使数据与时钟同步，这种电路就是源同步接口电路(Source Synchronous Interface)。

源同步接口最大的优点就是大大提升了总线的速度，理论上信号的传送可以不受传输延迟的影响，所以源同步接口也经常应用 DDR 方式，在相同时钟频率下提供双倍于 SDR 接口的数据带宽。

源同步接口的约束设置相对复杂，首先是因为有 SDR、DDR、中心对齐(Center Aligned)和边沿对齐(Edge Aligned)等多种方式；其次可以根据客观已知条件，选用与系统同步接口类似的系统级视角的方式，或是用源同步视角的方式来设置约束。

Vivado 设计软件提供对不同情境下规定 I/O 时序约束的 XDC 模板(XDC

Templates)，如图 5-38 所示(由 Tools→Language Templates→XDC→timing Constraints)。

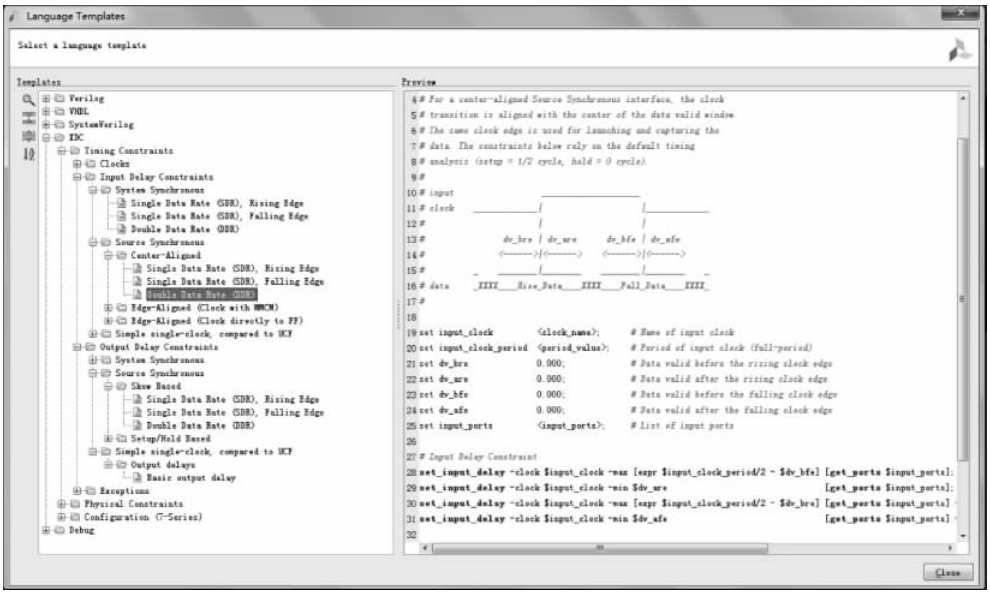


图 5-38　规定 I/O 时序约束的 XDC 模板

XDC 模板给出了 set_input_delay 和 set_output_delay 指令中延时数值计算利用的参数和公式。除前面用到的参数之外，对于源同步中心对准的情况，已知条件是数据相对于时钟沿的有效窗口值，下降沿分别为 dv_bfe(data valid_before fall edge)和 dv_afe(data valid_after fall edge)等；源同步沿对准时，已知条件是数据相对于时钟上升沿和下降沿的偏移，上升沿分别为 bre_skew(before rise edge_skew)和 afe_skew(after fall edge_skew)等。按照 XDC 模板给出的所要求的延时参数，再利用列出的公式可以计算出各种情况下到输入和输出端口的最大和最小延时数值，最大值为建立时间校验，最小值为保持时间校验。

注意时序约束中，默认的是-max 和 rise edge，而-min 和 fall edge 必须标注，对于多个时序约束的语句，后加的要添加-add_delay 说明，防止后者覆盖前者。

5.3.3　时序例外约束

时序约束对设计的编译过程有重要的影响，布局布线工具要对最差的时序路径花费最多的努力，以使其满足约束的要求，如果经过努力，仍有达不到约束要求的路径，综合工具将以红色告警给予提示。时序约束一般包括内部和 I/O 时序约束，以及最大和最小时序约束。

时序约束的设置考虑通常是先全局，后局部。首先制定设计项目中普遍适用的全局性的时序约束要求，然后对特殊的结点、路径或分组指定局部性的时序约束要求，如果局部性的时序约束要求与全局性的时序约束要求有冲突，则局部性的时序约束要求的优先级更高。

1. 多时钟域约束

在一个设计中可以存在多个工作时钟。

(1) 独立时钟(Absolute Clock)：要指定独立时钟的 f_{max}和占空比。

(2) 衍生时钟(Derived Clock)：由独立时钟派生出来的时钟，通过独立时钟信号进行

分频、倍频、占空比调整、偏移和反相等操作而生成的时钟信号。

2. 多周期路径约束

多周期路径是指数据稳定时间需要一个时钟周期以上的路径。例如一个寄存器需要每2个或每3个时钟周期锁存一次数据，对建立时间设置多周期数m，则对保持时间设置的多周期数(m－1)，是为了将捕获沿与起始沿对齐。

第4章的uart_led的例子中，设置了多周期约束为

```
set_multicycle_path -setup -from[get_pins led_ctl_i0/led_o_reg*/C] -to [get_ports led_pins*] 2
set_multicycle_path -hold -from[get_pins led_ctl_i0/led_o_reg*/C] -to [get_ports led_pins*] 1
```

3. 伪路径约束

伪路径是指不必关心其时序的路径。用户可以通过各种方法(例如软件设置、附加约束)将伪路径排除在时序分析之外。

伪路径时序例外约束set_false_path在时序约束中具有最高的特权。

4. 最大延时和最小延时

要对所有的静态时序通道进行建立和保持时间的校验，校验的要求由时钟之间的关系决定，应利用起始沿和捕获沿导致的最紧要求。在某些情况中，对于通道的默认要求不一定是正确的，例如不同时钟域之间的通道，为避免准稳态而使用的触发器之间的通道，是从输入端口直接到输出端口不经过任何定时元件的逻辑通道。在这些情况下，对建立和保持校验的要求需要利用set_max_delay和set_min_delay指令重写并应用于完全的静态时序通道。

建立时间校验可以利用set_max_delay重写，保持时间校验可以利用set_min_delay重写。例如，为避免准稳态的单比特同步器中，图5-39(b)中的双触发器通道2时序约束重写为

```
set_max_delay -from [get_cells REGB0] -to [get_cells REGb1] 2
```

设置一个小的最大值，保证两个触发器映射到一个Slice中，延长平均无故障时间。

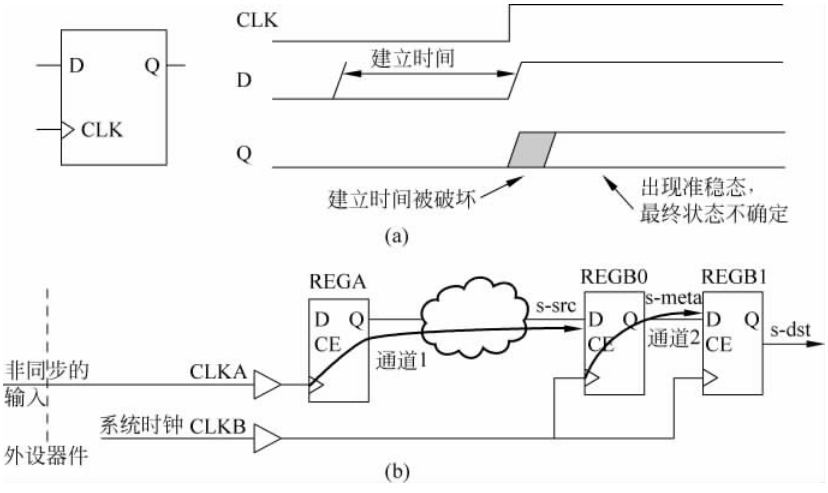


图5-39 准稳态的产生和消除

图 5-39(b)中的不同时钟域之间的通道 1 时序约束重写为

```
set_max_delay -from [get_cells REGA] -to [get_cells REGB0] 5 -datapath_only]
```

对于从输入端口 CombIn 到输出端口 CombOut 之间无定时元件的组合通道重写为

```
set_max_delay -from [get_ports CombIn] -to [get_ports CombOut] 8
```

5.3.4　同步设计中的异步问题

在对 FPGA 进行同步设计时，如何保证系统的时钟信号不产生相位偏移，从而使系统能够正常工作是至关重要的。同步设计要遵循的一个重要准则是不要通过组合逻辑电路来产生时钟信号以及异步的置位和复位信号，否则就会带来异步设计的一系列问题。例如，一个曾经通过实际电路调试的设计，在重新布线以后又不能工作了，或者一个通过时序仿真测试的设计，却不能通过实际电路的调试等。

此外，许多系统要求在一个设计中采用多个时钟，在时钟系统中，两个时钟信号之间要求一定的建立和保持时间，所以在应用中要对时钟信号引入附加的时序约束条件，也会要求对一些异步信号进行同步。两个异步微处理器之间的接口或微处理器与异步通信通道之间的接口都是多时钟系统的例子。

下面介绍一些消除同步设计中引起异步问题的方法。

1. 相位控制的方法

如图 5-40 所示的移位寄存器，因为存在时钟的相位偏移所以不能正常工作，在 FPGA 中要利用全局的时钟缓冲器，避免时钟相位偏移的出现，确保能满足触发器保持时间的要求，使系统正常工作。

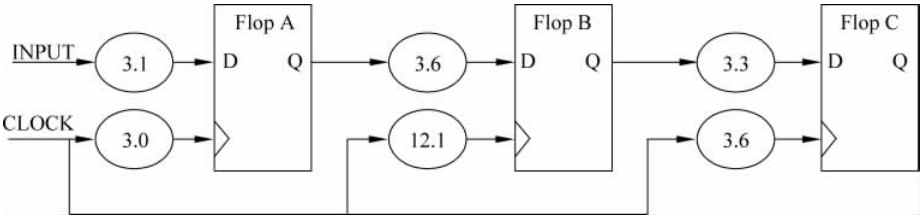


图 5-40　时钟相位偏移影响触发器正常工作

如图 5-41 所示的多时钟系统，其中一个信号在两个时钟区域间传递，两个时钟有一定的倍数关系。图 5-41(a)表示快慢时钟有不匹配的相位关系，数据的建立时间被破坏，出现准稳态的现象。对于不同周期有任意相位关系的两个时钟区域，如果其中至少一个时钟是在 FPGA 内部通过 PLL(锁相环)或 DLL(延迟锁相环)可控制的，其中一个时钟与在 PLL 或 DLL 解决方案中另一个时钟有倍数关系(此例为两倍)的周期，如图 5-41(b)所示，相位匹配可以用来消除时序冲突。PLL 调整(捕捉)较快时钟区域的相位，并与较慢的时钟区域(传送)相匹配。

2. 双触发器技术

图 5-39(a)说明当触发器的输入信号在时钟上升沿到达的相同时刻发生变化，触发器的

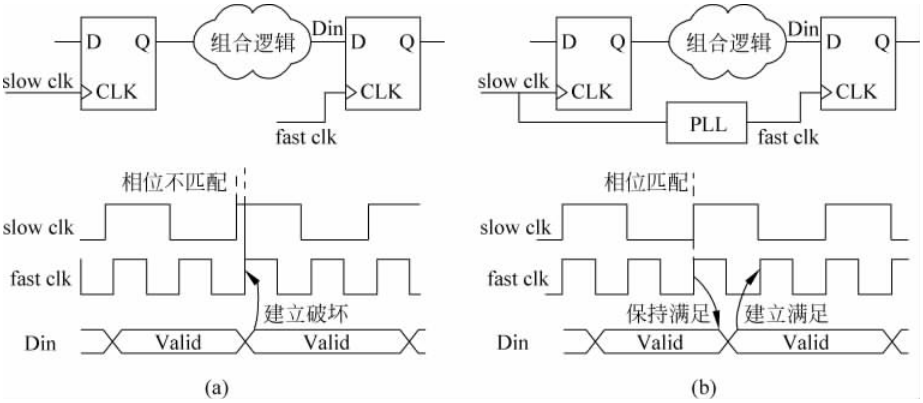


图 5-41　相位控制方法消除准稳态

建立时间被破坏，此时可能出现准稳态或称为亚稳态，FPGA 可以从准稳态很快恢复，但是最终状态可能取得其中间值。

在两个异步时钟区域之间，双触发技术是一项可以用于单比特信号传输的技术。由于建立或保持的时序冲突可以引起触发器中的一个结点变到准稳态。在信号稳定于有效电平之前，会有一个不确定数量的逗留时间存在。这个逗留时间会附加于时钟到输出(T_{co})时间(同时连接到路径的传播延迟)之上，并且可能会在下一级中引起时序冲突。如果这个信号被传入控制分支或判决树中，它就会变得极其危险。不幸的是，既没有好办法预测准稳态会持续多长时间，同样，也没有好办法将这个信息返回注释到时序分析和优化工具。假定两个时钟区域是完全异步的(不可能用相位控制)，一个减小准稳态发生概率的简单的方法是使用双触发技术。

为了避免准稳态的出现，使外设器件的异步输入电路也能安全地工作，可以用图 5-39(b)所示的方法进行同步，在非同步输入的触发器后增加一个触发器，将非同步输入变为同步输入，确保不产生准稳态，使系统正常工作。即双倍同步一个异步的信号到新的时钟域，在采样异步信号 s_src 之后，s_meta 很大概率为准稳态，第二次采样使 s_dst 以十分低的概率成为准稳态，Verilog 程序为：

```
always @(posedge clk_dst)
begin
    if (rst_dst)
    begin
        signal_meta <= 1'b0;
        signal_dst <= 1'b0;
    end
    else
    begin
        signal_meta <= signal_src;
        signal_dst <= signal_meta;
    end
end
```

双触发器可防止准稳态传播的原理如下：假设第一级触发器的输入不满足其建立保持

时间，它在第一个脉冲沿到来后输出的数据就为准稳态，那么在下一个脉冲沿到来之前，其输出的准稳态数据在一段恢复时间后必须稳定下来，而且稳定的数据必须满足第二级触发器的建立时间，如果都满足了，在下一个脉冲沿到来时，第二级触发器将不会出现准稳态，因为其输入端的数据满足其建立保持时间。

同步器有效的条件为：

第一级触发器进入准稳态后的恢复时间＋第二级触发器的建立时间≤时钟周期

这种情况下，为了更长的平均无故障时间 MTBF(Mean Time Between Failures)，需要配合一个 ASYNC_REG 的约束，把用作单比特同步器的多个寄存器放入同一个 Slice，以降低引线延时的不一致和不确定性，以图 5-39(b)为例有

```
set_property ASYNC_REG TRUE [get_cells [list REGB0 REGB1]]
```

或者利用 5.3.3 节的最大延时和最小延时，设置双触发器的数据通道路径的延时尽可能小，保证被软件推论映射到一个 Slice 中，加大平均无故障时间来避免准稳态，即 set_max_delay_from[get_cells REGB0]_to[get_cells REGB1]2。

3. 异步数据传输——FIFO 结构

在异步时钟域之间传递信号更灵活的方式是通过先进先出(FIFO)的方式。当在异步时钟域之间传递多位信号时可以利用 FIFO。FIFO 最通常的应用包括在标准总线接口之间传递数据和读写突发存储器。FIFO 是各种应用中十分有用的数据结构，利用 FIFO 也是解决许多类型的数据传输最好的方法之一。数据可能在一个时钟域以随机的时间间隔到达，可能包含大的突发量。在这个情况中，接收器件设置在不同的时钟域上，只可以处理特定速率的数据。发生在器件内部形成的队列以 FIFO 的方式进行存取，如图 5-42 所示。

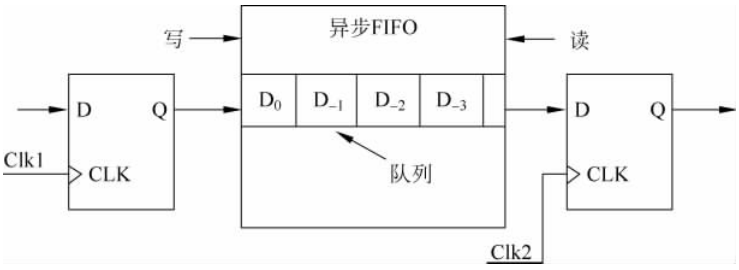


图 5-42　异步 FIFO

使用异步 FIFO，数据可以在任意时间间隔在发送端运送，接收端把数据推出队列，因为它有处理数据的带宽。由于使用 FIFO 实现有限尺寸的任意队列，需要一定的控制来适当地防止溢出。因此要预先设置 FIFO 的深度和 FIFO 的握手控制，包括发送速率的先验知识(突发或不突发)、最小的接收速率和相应的最大队列尺寸。

4. 应用时钟使能信号

同步设计中可以采用时钟使能信号来保证系统时序的同步和正确。图 5-43(a)说明如何产生一个与系统时钟完全同步的时钟使能信号来代替输入信号，但是要求输入信号的宽度一定要大于一个时钟的宽度。

如果数字系统的设计需要用到多个时钟，时钟使能信号能保持一个设计的同步，如图 5-43(b)中虚线框部分所示，可用一个时钟作为主时钟，并由它来驱动其他时钟。典型的

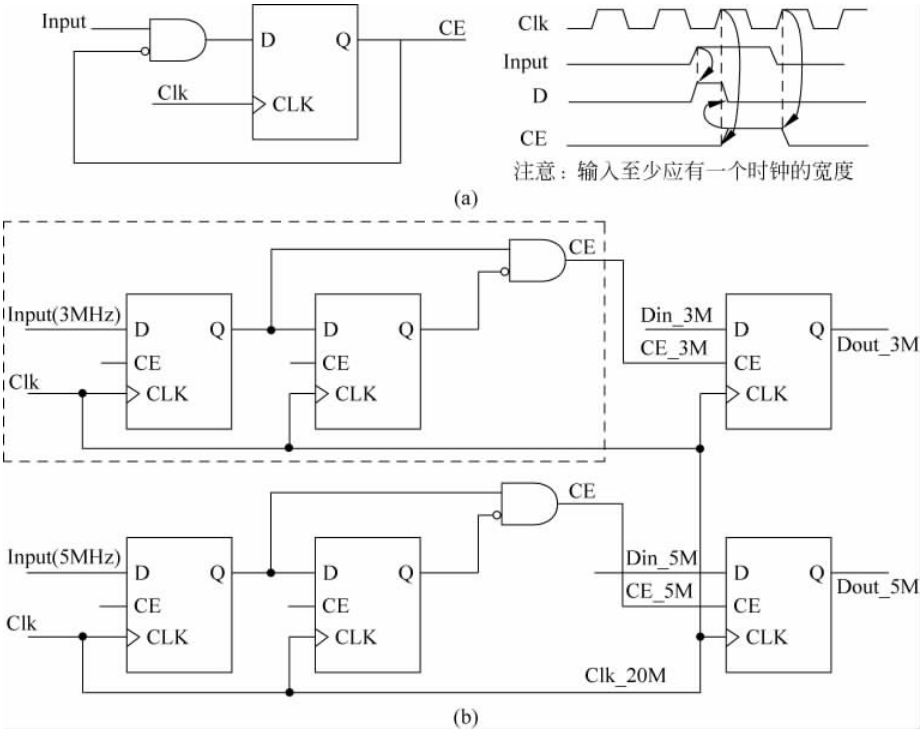


图 5-43　生成时钟使能信号

FPGA 的延时，50％由布线引起，50％由逻辑引起。不要忘记时钟输出和时钟建立的时间。上面提到的一些同步设计的方法，在采用硬件描述语言 HDL 设计数字系统时，也是应遵从的。

在许多应用中只将异步信号同步化还是不够的，当系统中有两个或两个以上不同信源的时钟时，数据的建立和保持时间很难得到保证，时序分析也变得十分复杂。解决的办法是同步不同信源的时钟，需要利用带使能端的 D 触发器，以及引入一个高频率时钟来实现不同信源时钟的同步。如图 5-43(b)所示，图中 Input 信号为不同信源的时钟(此例为 3MHz 和 5MHz)，Clk 为 20MHz 或更高的系统时钟频率，同步后的 3MHz 和 5MHz 使能信号输入到数据输入寄存器的使能端，输出数据就与系统时钟频率同步。

5. 消除组合电路毛刺

组合电路产生的毛刺脉冲也是数字系统不能正常工作的原因之一，图 5-44(a)给出一个产生毛刺脉冲的例子，并说明了毛刺脉冲产生的原因，因为电路的最高位布线最短，计数从 0111 到 1000 时，最高位先于其他位发生翻转，变成从 0111 经过 1111 再到达 1000，使与门动作产生毛刺脉冲。采用图 5-44(b)的电路，可以避免产生毛刺脉冲，使系统稳定地同步系统的时钟。

图 5-45 以对比的方式说明同步设计的方法，图 5-45(a)和图 5-45(b)是同步设计的好的示例，它们分别克服了图 5-45(c)和图 5-45(d)中异步设计的缺点。图 5-45(b)也是利用时钟使能的示例。

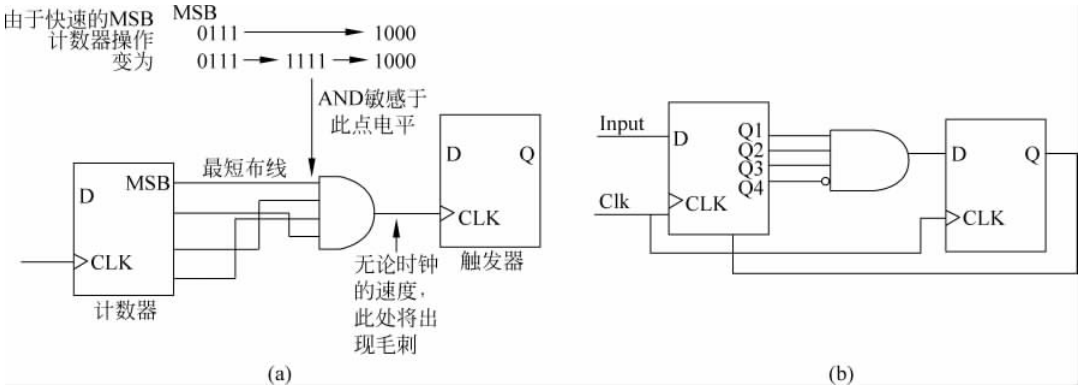


图 5-44　消除产生的毛刺信号

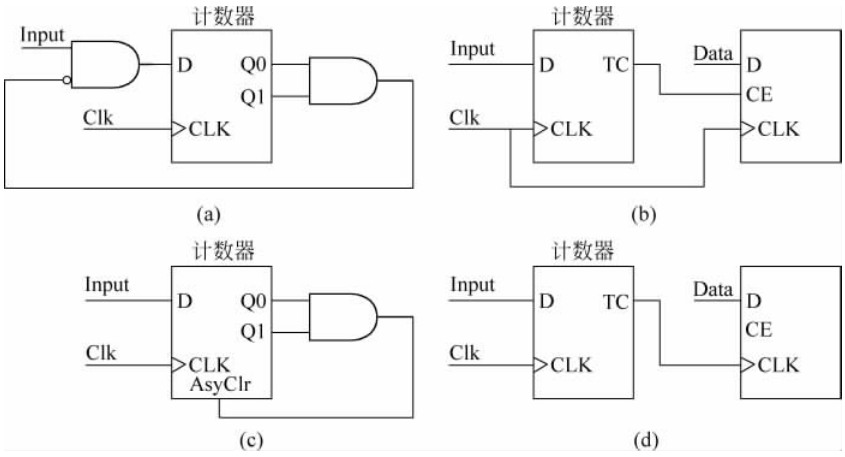


图 5-45　同步设计示例(a)(b)以及异步设计示例(c)(d)

5.4　数字系统的综合

5.4.1　数字系统综合概述

通过 1.2 节对数字系统的层次化结构进行分析之后，本节再来讨论数字系统设计的综合方法。按照层次化的概念，数字系统的集成电路设计过程定义为从硬件的高层次抽象描述向低层次物理描述的一系列转换过程，利用计算机辅助设计的自动化工具，将一个层次的描述形式转换为另一个层次的描述形式，这个过程称为自动综合，简称为综合。

如图 1-1 的电子设计的 Y 图所示，构成数字系统的集成电路可以在不同的级别(或层次)上研究，自底层向上将其分为电路级、逻辑级、寄存器传输级、算法级和系统级。设计从底部的电路级开始时，复杂度随设计的最终完成而不断地增加。集成数字系统的芯片性能取决于每一级单独的特性和各个级别链接在一起的方式。

系统级集成芯片的设计常采用自顶而下的设计方法，用硬件描述语言(HDL)在行为级进行描述，再通过编译器将 HDL 描述的设计表示转换成对系统级综合更有用的内部表示，

内部表示通常是图和语法树，但绝大多数系统级综合利用数据流图或控制流图。

广义地说，由行为描述变换到结构描述的过程称为数字系统的设计综合。为了定义和区分综合的类型，可以由三部分组成的设计表示来说明，如图 1-1 所示。这三部分设计表示描述三个不同的范畴，如同前面在介绍反相器设计中所提到的，它们分别是行为域、结构域和物理域。

在所描述的范畴内，由顶向下是构成系统的不同级别或层次，由底向上移动时，随着系统构成的升级，各级别的设计描述变得越来越抽象。设计工具可以表示为在两个范畴或两个域之间所要做的转换，由所要完成的转换可以说明每个工具利用什么信息和使用这个设计工具产生什么信息。

综合工具可以把所设计电路的高层次描述自动地转换为低层次的描述，也可以把同一层次的行为描述自动地转换为该层次的结构描述。由综合器完成的这些转换，类似于软件开发时，利用编译器把高级语言编写的程序转换为可执行的机器代码。

数字系统的设计描述从行为域变换到结构域在不同设计层次的综合过程分别是：

（1）系统级：系统级综合。

（2）算法级：算法综合＋高级综合。

（3）寄存器传输级（微结构级）：RTL 综合。

（4）逻辑级：逻辑综合。

（5）电路级：版图综合（物理域）。

如上所述，“综合”的术语用来定义把一个数字系统从行为描述技术条件到硬件实现结构设计的过程。通常来说，技术条件包含一定形式的抽象，即对一些设计判决没有进行限制；另一方面，硬件实现必须在给定的抽象级别上详细地描述完整的结构设计。因此，综合可以看作产生技术条件之外所留下的硬件实现细节的处理过程。例如，微处理器的纯粹行为描述可能只规定在一般的指令周期应该做什么，而把是否应该利用中央总线、采用什么技术实现控制功能、应该支持多少并行性等问题留待综合过程去处理。

由于数字系统的复杂性，特别是在 VLSI 工艺中实现它们时，综合过程通常分成几个步骤来完成，这些步骤包括系统级综合、算法综合和高级综合构成的算法级综合、寄存器传输级综合以及逻辑综合等。

系统级首先要给出所设计的整个系统的技术条件，将这些技术条件作为综合工具的输入进行系统级综合。对于复杂的数字系统，一般不适宜直接将系统技术条件的高级语言描述转换成同一层次的结构描述。所以，系统级综合完成对系统技术条件的高级语言描述自动转换成系统技术要求的算法描述，提供给下一层次的算法级作为综合的输入。而算法级可以包括算法综合和高级综合。算法综合是在行为域对系统的算法进行建模、仿真和优化，产生系统的一般性结构，所以是系统的算法设计；高级综合是将算法综合后优化的行为描述转化为同一层次的结构描述，所以高级综合是对算法综合确定的优化算法进行调度、分配和优化。所以，算法综合得到的算法设计结果是高级综合的输入，也是高级综合的出发点。随着系统规模的扩大和性能要求的提高，为了获得一个好的系统实现方案，必须对系统的功能通过行为描述确定一个优化的算法，在后面讨论的 DSP 系统设计中，由于不可能对设计的 DSP 系统直接给出优化的 RTL 级描述，就要先不考虑结构实现，先进行算法设计的建模、优化和仿真，然后再进行实现。利用电路输入-输出行为模型的算法描述要比 RTL 描述

更加抽象，模型的算法描述与实现的硬件结构之间没有明显的映射关系，也不具有寄存器、数据通道和计算资源的隐形结构，所以算法描述可读性好、容易理解，其语句按照顺序执行，没有明显的结构形式。这就是为什么要突出算法级的算法综合，它是与结构无关的算法建模、仿真和优化，之后再转换为结构设计的高级综合。

系统级综合处理系统实现的基本结构的组成成分是这级的输入，是系统级的技术条件，它根据一组交互的过程描述整个系统的行为。这样一个系统可以由一组协同操作的处理器实现，例如 ASIC、专用控制器、FPGA 和 DSP 处理器等。分配这组实际的处理器和把行为技术条件的过程映射到这些处理器是在系统级综合步骤要做出的最关键的判决。系统级的重要特性是综合技术和设计要求与应用密切有关。例如，实时嵌入式控制器应用领域中使用的系统级综合是完全不同于 DSP 系统中的系统级综合。这个特性也导致构成系统级综合步骤的设计任务有不同的定义，作为系统级综合技术的共同特性，要考虑的与系统级综合有关的课题是系统分割、硬件/软件协同设计以及互连结构设计等。系统级综合步骤的输出是一组带有明确定义的接口的过程，每个过程由行为技术条件规定。

系统级设计是硬件实现过程的第一步，综合过程要把系统技术要求的高级语言描述转换为真值表、状态图或算法模型等。将高级语言描述转换为真值表、状态图要求硬件的功能有明确的结构形式或可以映射到相应的 IP 核，所以对硬件结构会有限定。而将高级语言描述转换为算法模型时，不会对硬件结构或电路形式预先作任何限制。实际上，对于硬件结构预先不能确定的复杂数字系统的设计，可以先不考虑硬件结构的限制，将高级语言描述的技术条件先转换为算法的要求，由算法综合对算法进行建模、仿真和优化后，再由高级综合将算法模型转换为硬件实现的结构描述。

高级综合则将这个过程的行为技术条件转换成与工艺无关的结构描述，这个结构描述通常是根据寄存器传输级的网表清单给出的。

系统级综合、算法综合和高级综合形成数字系统设计前端的综合方法，在执行了系统级综合、算法综合和高级综合后，使用逻辑综合和物理设计把寄存器传输级的结构实现映射到最后实现的布图描述。逻辑综合和物理设计形成数字系统设计后端的综合方法。

把综合过程分成前端系统综合和后端综合的一个重要原因是因为前端综合处理具有一定的良好特性，而与后端综合有关的半导体工艺通常寿命较短。当前端系统综合不受实际工艺的影响时，可以在几个不同的设计环境中使用，需要时可以很快适应新工艺，但是后端综合因为工艺变化、寿命周期十分短，需要适应工艺的变化而变化。

本节主要介绍各层次综合的行为描述到结构描述的转换，算法综合将在第 6 章介绍。

5.4.2　系统级综合

系统级综合是在最高抽象层次表示系统的设计步骤，在这一级，技术条件是作为一组交互过程构成的，在该步骤期间考虑的基本系统元件是处理器、ASIC、存储器和总线等。因此，系统级综合是系统设计的最高级别操作，其中基本的判决是确定哪些因素对被设计系统的结构、成本和性能有大的影响。

系统级综合的输入是交互进程的可执行技术条件和一组设计约束。约束可以与功能技术条件一起定义，或者分别给出，它们是诸如成本、速度、输入/输出速率、功耗和可测性等系统的特性。事实上，输入是可执行的技术条件，对设计具有很重要的实际意义，因为它允许

在较早的阶段用仿真和诊断进行系统校验，避免多次重新设计产生更多成本。

为系统技术条件选择适当的语言是设计方法的十分重要的方面，例如选择交互的VHDL进程描述系统。

系统级综合产生的输出是一组行为技术条件，每个技术条件对应于分配到系统元件的模块，每一个模块都可以利用高级综合工具直接综合到硬件，或者当它分配到一个可编程元件时，它可以对软件进行编译。因此，系统级综合涉及硬件/软件协同综合的问题。

系统级综合可以判决全部进程是否应该在ASIC的硬件中实现，这些进程可以作为带数据通道的单个控制器进行综合，也可以判决为几个控制器在ASIC上实现这些进程。它们可以通过共享的数据通道元件或通过一条总线相互通信，也可以分配一些进程到硬件实现，而另一些进程到处理器上进行软件实现，处理器可以是流行的微处理器、微控制器、DSP或专用的集成处理器(ASIP)，更复杂的处理器结构可以选择由几个处理器、ASIC、存储器、总线等组成来实现。

系统级综合必须决定系统元件的种类和数量，以及在这些元件上分配的专门的功能。这个判决过程由确定的优化策略指导，也要考虑由设计约束给出的限制。系统功能分布到不同的元件上导致这些元件之间有通信的需要，为通信而设计的硬件和软件必须在系统级综合期间产生，并放置在整个系统的文本中。

对于系统级综合没有确切定义的步骤，也没有强制的算法和协议，但是有一些系统级综合特殊课题强调的算法和方法。

系统级综合期间必须执行三个主要的步骤：

(1) 分配系统元件：这个任务定义系统级的结构视图，要选择一组可以实现系统功能的处理、存储和通信元件，这些元件可以是微处理器、微控制器、DSP或专用的集成处理器(ASIP)、ASIC、FPGA、存储器或总线等。

(2) 分割(partitioning)：系统分割把技术条件获得的功能分布到所分配的系统元件中，因此，由进程规定的行为在微处理器、微控制器、DSP或专用的集成处理器(ASIP)、ASIC、FPGA之间分割，而变量被分割和映射到寄存器和存储器。

(3) 通信综合：通信综合产生为系统模块通信需要的硬件和软件，以及它们的通信协议，作为这个任务的一部分，通信通道必须分配到共享的寄存器、总线或端口。

三个任务是交织的，为了获得最佳的实现，理想情况是三个任务应该一起执行，而没有预先规定的顺序，但是理想的方法比分开执行极大地增加了复杂度。首先分配元件，然后分割功能到固定的结构，可以减少问题的复杂性，但是限制了方案可行性的空间。为了减少设计过程的复杂性，通常是一次解决一个设计任务，在估计这个结果的性能和成本之后，可以更新元件和任务开始分配一个新的设计迭代，由极快速的分割和估计工具以及高速的仿真器运行几次这样的设计循环才能完成。另一个设计策略是从一个初始设计开始，在考虑通信课题的同时，变换执行分配和分割进行优化，这是典型的变换方法，解的质量取决于搜索非常大的设计空间而使用的启发式算法的效率。无论系统级综合任务采用哪种专门的策略，整个任务的目标是在设计空间找到对应于接近最佳解的实现，并满足施加的设计约束。约束和最优准则可以指成本、速度、芯片面积、引脚数、功耗、可测性、存储器尺寸等。

1. 分配系统元件

通过分配决定实现系统采用的元件种类和数量，可以分配的三类元件是

(1) 处理元件：微处理器、微控制器、DSP 或专用的集成处理器(ASIP)、ASIC、FPGA。

(2) 存储元件：存储器、寄存器堆、寄存器。

(3) 通信元件：总线。

由于分配系统元件的过程十分复杂，按照当前流行的方式，借鉴有经验的设计者的技巧来执行这个任务，可能替代的选择数量十分庞大，所以必须考虑设计约束、有效的诊断、可获取的工艺和预期的生产数量来仔细地分析。例如，设计者首先应基于流行的处理器或处理器核考虑一个实现，这样基于软件的方案才是有利于优化成本的，因为定制的硬件可以减少一些粘附逻辑；为了改善速度，可以将几个处理器连接成 MIMD 或 SIMD 处理机。如果所有软件方案不能满足性能约束，则可以在 ASIC 上硬件实现部分或全部的功能。另一种选择是，为了满足设计要求，可以基于 ASIP 实现，专门的指令集结构通常可以提供比流行处理器更好的性能，尤其是期望大批量生产时，这种方案可能是成本最有效的选择。

2. 系统分割

系统分割的目的是把适当的对象指定成类，使得给定对象的功能是优化的，从而满足设计的约束。系统分割不仅是为了成本/性能的优化，也是为了降低系统综合的复杂性，通过把设计划分成更小的元件，可以有效地被设计工具管理。分割可以在设计进程的各个抽象级别上执行。对于较低的抽象级别，一般采用结构性分割，在这种级别上，分割结构设计决定了硬件对象必须如何分组才能满足确定的封装约束。在布图级，使用优化面积或传播延时为目的的分割方法。但是，在功能技术条件被综合到结构之前，要做出对被实现的系统最后质量有更大影响的基本判决。在系统级，功能性分割是为了将系统的行为在多个元件之间进行划分。硬件对象不一定在这一级分类，但是，行为、变量和通道要分类。行为必须分类，分配到处理单元，使得满足对硅面积、存储器尺寸或执行时间的约束，在分类之间的互连数量是最少的；变量被分类，分配到存储器，并要考虑各种对象，例如，由并行性行为同时存取到相同存储器模块的可能性最小；对于通信综合，通道被分组，分配到总线，以减少总线冲突的数量和连接到给定总线的行为数目。功能性分割的结果是产生行为模块，在以后的设计步骤中，这些模块将综合到软件或硬件结构。

为了产生高质量的结果，分割算法必须依赖定量测量的质量，这个可以由不同的属性表征，称为度量，必须由定量的表达式表示。

有两种基本类型的分割方法：结构的(分类)和迭代的(基于变换)。结构的算法利用由底向上的方法：每个对象初始属于它自身的类，类别则逐步合并或增长，直到找到要求的分割。关于对象分类的判决是基于相互接近的程度，它不要求系统的全局视图，只依靠对象之间的局部关系。对对象分组选择正确的策略和适当的度量可以获得要求质量的最后分割。迭代策略是基于设计空间的搜索，由反映分割的全局质量的目标函数直接指导。启动的方案被迭代地修改，从一个候选方案传递到基于目标函数估计的另一个方案，最后的目标是达到接近最佳的解。设计空间的搜索是基于在可行性解的集合上定义所谓的邻近结构实现的。

将一个给定的图分割成两个相等尺寸的类，试图使两个类之间的边总成本最小化，这就是极小截分割算法，是典型的布图优化中的结构性分割。实际的电路通常呈现高度的局部性，好的布局应该能够考察局部以群聚连接紧密的元件，使它们能够接近在一起。识别群集的一个方法是将元件分割成两个相等或几乎相等的集合或区域，而使横跨两个分割的边的

数目最小，如果一根网线连接一个集合中的元件和另一个集合的元件，此网线称为截，网线截的集合称为截集(cutset)，网线截的数目是截尺寸(cutsize)。分割的目标是使截尺寸最小，同时平衡各集合的尺寸。

简化的基于极小截的布局可以如下说明。一个电路的逻辑元件分割成如图 5-46(a)所示的两个区域，电路图的结点 a、b、c、d、e、f 和 g 是一个电路的逻辑元件，边是互连线，逻辑元件则在两个区域 $P1$ 和 $P2$ 之间移动，目标是横跨分割之间的边界的互连线数目最小化。当元件 g 选择从 $P2$ 移到 $P1$，截尺寸从 3 减少为 1，如图 5-46(b)所示。

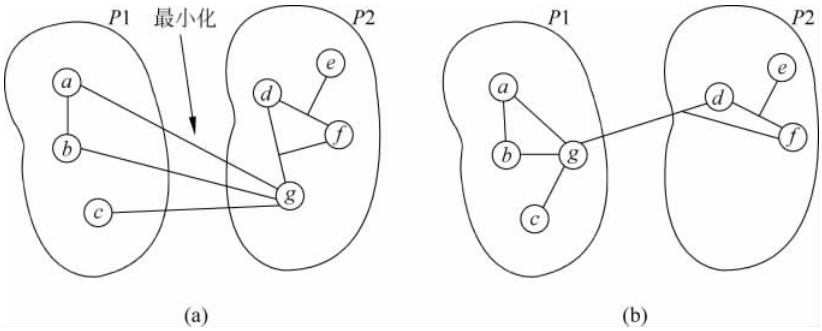


图 5-46　基于最小数目截分割的图示说明

如果某个区域又一次分割，从两个区域之间最小的通信来分配逻辑功能块，整个分割一直重复到某个分割的截尺寸为 1，然后将每个逻辑元件分配到芯片的物理元件上。

整个过程已经简化。首先，初始分割是随机产生的；其次，截尺寸不会总是很幸运地减少。Kernighan 和 Lin 给出了对图等分而使边缘的截最小化的推断。他们的算法的基本思想是从随机分割出发，然后对一系列分割之间的结点(元件)执行成对地不厌其烦地试探性交换。试探性交换的结点是要使截尺寸最大地减少或最少地增加的结点。试探性交换的最好的那部分子序列则选作实际的交换，新的等分形成之后，整个过程重复进行直到新方案不再比旧方案更好。一般地，算法采用几个不同的随机初始分割进行重复。

3. 通信综合

作为系统分割的结果，规定系统行为的进程已经分配到元件，这些元件可以是执行软件进程的处理器和专用的硬件元件。按照它们的系统级技术条件，进程是通过抽象的通道通信的，它们也必须与外设器件和专用接口交互作用。系统级通信是通过类似同步或异步消息传递、会合、遥远过程呼叫和监控器规定的，全部实现细节在这一级是隐含的。

通信综合产生连接系统元件的硬件和软件，使进程能够相互通信，同时能与外设器件和其他外部接口通信。

作为由顶向下的设计任务，通信综合包含三个主要的步骤。

1) 通道绑定

由系统技术条件定义的抽象通道必须利用实际的通信元件实现，这些元件可以是点到点的通信线或共享总线。因此，第一个任务是给通过系统的通信保障分配资源；然后，抽象的通道必须分割；最终的分组受分配的资源约束，在一个分组中对应通道的消息是在共享的通信元件上进行多路转换，这个任务非常类似于高级综合中的分配/绑定。类似于其他系统元件，以成本/性能取舍进行通信元件的分配。

2）通信完善

在通道绑定中分配和绑定之后，通信仍然是在高抽象层次描述，对于通信的实现仍然有几个可采用的方案，技术条件必须在几个细节上完善，直到最后的实现。在前面的步骤之后，基本能够知道系统的互连拓扑，包括基于点到点的通信、共享总线或混合的方案，也确定通道受给定的通信元件约束。为了确定通信保障的实际特性，现在必须做出几个判决：①根据数据传输率、有效引脚数和成本等的限制，确定通信链的宽度；②对于共享的通信总线，确定相当的控制策略，实现要求总线从元件之间的每次通信通过总线主元件执行，可能使性能严重受限；③为了以一致的方式提供系统元件之间的通信，确定每个通信链的通信协议，协议定义通信保障的数据传输的准确的机制，完全握手、半握手、固定延时、硬线端口或多层协议等。

3）接口产生

在前面两步中，系统技术条件中通信保障的细节逐步完善，根据提供的这些信息，可以产生系统功能需要的接口，意味着以下项目的综合：①通信进程内的存取路由；②缓冲器、FIFO 队列、逻辑属性组成的控制器；③利用通信协议不一致的接口元件需要的适配器；④存取到外设器件和专用接口的器件驱动；⑤中断控制、DMA、存储器映射 I/O 等有关的低层次通信保障。

5.4.3　高级综合

高级综合接收数字系统的行为描述，产生一个 RTL 级的实现。通常，高级综合由三个主要任务组成：调度(Schedule)、分配(Allocation)和装配或绑定(Binding)。调度要处理的问题是把每个操作调度到与时钟周期或时间间隔对应的时间间隙。分配和绑定要对给定的设计完成硬件资源的选择和分配，而绑定要把被选择的硬件元件分配到一个给定的数据通道结点。这里的分配含有分配和绑定两个任务的含义。

高级综合系统顺序地执行基本的综合步骤。首先，行为描述编译成内部的设计表示，在这个步骤期间，可以利用编译器优化技术。然后，要完成对内部的设计表示分配元件，对给定的技术条件所利用的元件确定其实现的类型和数量。接着，调度将操作调整到时钟的节拍。最后，绑定步骤分配功能单元、寄存器和总线等实际元件到设计表示的数据通道单元上。

上述高级综合任务相互之间不是独立的，调度调整操作到时间间隙，所以限制了分配的自由度。例如，分配两个“加法”操作到相同的时钟节拍，意味着它们要并行地执行，所以不可能共享相同的加法器。因为这两个操作必须约束到不同的加法器或其他功能单元，这限制了绑定的自由度。另一方面，先执行分配也将以类似的方式限制调度。

可以利用优化算法来执行调度、分配和绑定，从整数线性规划的经典方法到为综合目的专门建议的最优启发式方法，例如清单调度、受力方向调度或为分配和绑定设计的左边沿算法等，最优算法在不同方式和范围内有很多选择。某些情况下，也使用基于规则的方案。主要的问题是如何达到最优的结果，最优结果并不一定总能获得，因为已经证明为调度和分配寻找最优解的问题是完全 NP 的问题。这意味着对于实际问题，使用保证最优解的算法处理这些问题不是很难。

高级综合进程的输入是算法级的技术条件给出的，例如行为描述的 HDL，这类技术条

件给出从输入序列映射到输出序列的要求。技术条件应该尽可能少地对被设计的系统内部结构进行约束。由输入的技术条件，综合系统产生一个数据通道的描述，即寄存器网络、功能单元、多路选择器和总线。如果不集成到数据通道中，也应该产生控制器部件。在同步设计中，控制部件可以作为微码、PLA 阵列或随机逻辑提供。

数据通道的基本元件将由给定工艺有效的一些物理模块来实现，这些物理模块的硅面积、操作延时和功耗等工艺参数通常存储在模块库中，对高级综合算法是有效的，这种方法中，相同的高级综合算法可以利用不同的模块库基于不同的工艺对设计进行综合。

在高级综合中执行的基本任务是行为分析、设计方式选择、操作调度、数据通道分配、控制分配、模块绑定和优化等。

完成综合任务意味着做出设计选择，通常有一组替代的结构可以用来实现给定的行为。例如，在给定的行为技术条件中，一个加法操作可以由专门的加法器实现，或者与其他操作共享 ALU。

综合算法的功能是分析各种替代方案的全集或子集，选择最好的满足设计约束的结构，例如满足对周期时间、面积或功率的限制。综合的难处是不同设计方面之间的取舍是高度依赖的。因此，对综合任务进行设计判决时，其他任务必须依次按照优化的一些设计准则来考虑，例如实现的成本。因而，每个综合任务不可能在不影响设计全局优化的情况下独立地完成。

1. 调度

调度是对每个操作进行处理，将操作分配到时钟周期或时间间隔。一般地，这个任务的输入是由控制数据流图(CDFG)、一组有效的硬件资源和性能约束组成，调度产生的结果应该不破坏由 CDFG 定义的数据/控制关系、性能约束得到满足。

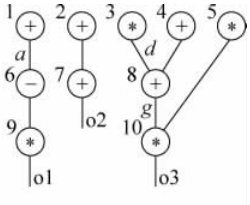
调度决定哪些操作可以分配到相同的时间间隙上，这影响到最终设计的并行程度，进而影响性能。在一个调度中，一个确定类型的并行操作数目最大化是对这个操作要求更低数量硬件资源的约束。所以，调度的选择影响实现的成本，因此调度在高级综合中起重要的作用。

调度问题取决于所作的基本假设可以由几种方式构成。一个直接的方式是假设行为描述不包含条件的或循环的结构，每个操作准确地花费一个控制步骤，且仅由一个类型的功能单元来执行。这是资源限制(Resource-Constrained)调度。

图 5-47 给出一个简单的行为描述和与它对应的数据流图(DFG)。DFG 由分析不同操作的数据依赖关系的算法产生，数据关系分析基于以下原则执行：如果第一个操作数的结果被第二个操作数使用，则第二个操作数必须在第一个操作数完成后才能开始。这两个操

```
a = i1 + i2
o1 = (a - i3) *3;
o2 = i4 + i5 + i6
d = i7 + i8
g = d + i9 + i10
o3 = i11 * g
```

(a) 行为技术要求



(b) 数据流图

图 5-47　行为技术要求和其数据流图

作数之间有数据关系，这个数据关系在综合过程中作为两个操作数之间顺序的约束，必须由调度来满足。图 5-47(b)说明从图 5-47(a)的 HDL 代码产生的数据流图，图 5-47(b)中假设所有的数据向下流动，所以定向边只画了边，而没有画表示方向的箭头。

最简单的调度技术是基于尽可能快(ASAP)原则的启发式贪婪算法。为了利用 ASAP 算法调度图中的数据流图，要按照它们的局部次序的拓扑先存储操作数，然后进行 ASAP 算法调度操作，以尽可能早的控制步骤按照存储次序的拓扑依次放置它们。如图 5-48(a)所示，图中标识操作的数字表示拓扑的存储次序，与图 5-47(b)中所示一致。

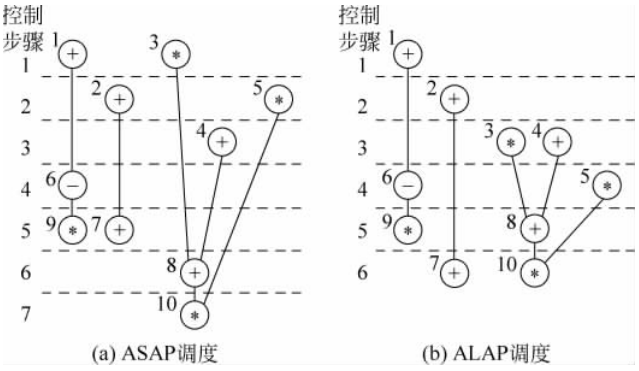


图 5-48　ASAP 和 ALAP 调度的示例

简单调度问题的类似方法是利用尽可能晚(ALAP)的原则，采用 ALAP 算法时，也是先存储数据/控制关系的拓扑，与 ASAP 的情况一样。但是，调度操作按照最晚可能的控制步骤向后放置它们，如图 5-48(b)所示。

通常要求产生的调度是一个最优的调度，即花费最少的控制步骤执行规定的行为。但是，调度的问题是完全 NP 的，不能保证最优结果的启发式算法被广泛用于产生满意的解。较广泛利用的调度算法有推定技术的调度类型和基于变换的调度类型。

ASAP 调度、ALAP 调度、清单调度和受力方向调度等是推定技术的调度类型。ASAP 调度和 ALAP 调度是一步接一步推定，直到所有操作被调度；清单调度从控制步骤到控制步骤地按照特权顺序进行调度，这些属于限制资源的调度。而受力方向调度是限制时间的调度，它的基本策略是在不同的控制步骤放置类似的操作，以平衡分配到功能单元操作的并行性，不增加总的执行时间。平衡操作的并行性，保证每个功能单元都有较高的利用率，所以要求的单元总数减少。受力调度算法由三个主要步骤组成：①确定每个操作的时间框架；②产生分布图；③计算与每个分配有关的受力。由于受力方向调度算法在每个迭代中仅调度一个操作，所以它也是推定的。但是受力方向调度在选择下一被调度的操作时，要对操作和控制步骤作全局分析，所以受力方向调度更耗费计算时间。

基于变换的调度是从一个初始的调度开始，通常是最大的串行或最大的并行，再把变换运用于其上来获得其他调度。基本的变换是把串行操作或操作模块变换到并行操作，或相反地变换并行操作到串行操作。基于变换的算法如何选择采用的变换和变换的次序是不同的。基于变换的方法的一个重要的优点是在每次迭代中，存在完全的调度，可以根据不同的准则进行设计的准确估计，可以直接扩展这类算法来管理许多与调度有关的高级课题。

2. 分配和绑定

数据通道的分配和绑定通常是解决利用什么资源来进行物理实现的问题，这样的资源包括寄存器、存储器单元和不同的功能单元以及它们的通信通道。基本的原则是在性能和其他设计准则可以满足的条件下尽可能地共享资源。分配和绑定为给定的设计选择和分配硬件资源。分配决定给定设计的硬件资源的种类和数量，绑定把分配的硬件资源的实例赋予给定的数据通道结点。不同的数据通道如果不同时执行可以共享相同的硬件资源。例如，两个加法操作不在相同的时钟周期期间执行时，可以共享一个加法器；如果变量的寿命周期不重叠，一个寄存器可以用来存储两个变量的数值。有时分配和绑定都用分配来表示。

在分配步骤期间，选择硬件资源的种类和数量通常公式化为优化的问题，主要的目标是在满足给定的面积/性能约束的同时，找到最少数量的资源。许多高级综合系统考虑绑定所作的基本假设是，每个数据通道结点至少有一个模块库中的模块实现数据通道结点的功能。例如，执行加法操作的结点可能对应于一个加法器或模块库中的一个 ALU，不同的模块可能有不同的面积或时滞，就有可能在不同的实现之间进行舍取。绑定问题也是最优化问题，可以利用存在的优化方法公式化。例如，可以使用整数线性规划或图形分类技术来进行求解，也可以利用启发式方法来求解。

在组合最优化问题的实际情况中，线性规划公式常常可以在未知数被约束在整数值的条件下应用，所以称为整数线性规划。

最小化 $\mathbf{c}^{\mathrm{T}}\mathbf{x}$ 以使

$$\mathbf{Ax} \geqslant \mathbf{b}$$

$$\mathbf{x} \geqslant \mathbf{0} \qquad x \in Z^n$$

问题的离散特性使得它是不可控制的。整数线性规划求解的下限可以通过放宽整数约束找到，把一个实数解矢量舍入成整数不保证达到最优。当整数线性规划模型判定问题和变量被限制是二进制值 0 和 1 时，这种真实情况的整数线性规划称为 0-1 整数线性规划或 ZOLP 或二进制线性规划。整数线性规划可用各种算法求解，例如分支定界算法，放宽整数约束推导线性规划的解常常用来作为这些算法的起始点。

5.4.4 寄存器传输级综合

根据带数据通道的有限状态机(FSMD)的构成，寄存器传输级(RTL)的综合包括数据通道综合和控制器综合。

1. 数据通道综合

数据通道是由具有一定拓扑关系的互连的几个功能元件组成，一般采用同步工作方式，由时钟沿触发进行同步。功能元件实现由行为描述定义的各种操作，包括逻辑操作、算术运算、关系运算、移位运算和复杂运算等。数据存储在存储元件中，不同功能元件之间的数据交换通过通信元件进行。数据通道单元通常用数据流图描述。

有两个主要的数据通道形式，取决于数据通道的元件之间的通信组织的方法，寄存器、操作数和外部部件之间的通信可以通过总线或者通过多路选择器进行，如图 1-10 所示。

流水线是充分利用硬件内部的并行性、增加数据处理能力的有效方法，使用流水线，硬件一定要执行某种迭代形式的操作。在数据通道流水线中，为了允许并行执行子序列语句，要复制一部分数据通道。

数据通道通常由以下几部分组成：①包括算术逻辑单元 ALU 和数据存储寄存器的计算部分；②数据通过系统的逻辑；③数据在计算单元和内部寄存器之间移动的逻辑；④移动数据进出外部系统的通道等。

2. 控制器综合

系统综合通常把系统规定为一组通信的并发进程来对待，在这种情况下，控制器综合不仅与控制器的功能而且与交互和同步要求都有很大的关系。实现控制器可以通过有限状态机和微程序方式两种方法，其中有效的方法是用有限状态机表示。

1）控制器形式选择

有几种设计形式可以用来实现复杂的控制器，这些方法的主要思想是用几个较小的控制器实现复杂控制结构，以简化生成的有限状态机的设计。由单个 FSM 模型化的控制器是单个控制器，也是最简单的解决方案；由大量控制器按层次组织起来的控制器是层次控制器，当控制架构中有过程和环路结构的算法语言时可选择该方法；控制器的功能分布在几个较小的并行执行的控制器，这种结构是并行控制器，这些并行控制器可以通过通信交换数据或同步执行。

2）控制器产生

当控制器形式已经选定，FSM 的数量确定，就需要产生这些 FSM。控制器产生的首要任务是确定使用 Moore 机还是 Mealy 机，这个判决主要取决于所使用的设计表示和后端逻辑综合工具的有效性，但是，Mealy 机和 Moore 机可以等效变换。

3）控制器实现

对于不同的控制器的形式，要实现不同的控制结构，但是基本的结构是根据单个的 FSM 来实现的，通过这个基本实现和复杂控制器附加的同步和通信硬件一起实现其他形式的控制器。单个 FSM 控制器实现可以采用随机逻辑、微码或 PLA。

无论使用什么技术，一般的实现结构都是十分类似的，一个状态寄存器用于存储当前状态，而根据当前状态和数据通道输入的条件组合逻辑用来产生下一状态，当前状态也用来产生数据通道的控制信号，某些情况下，附加的解码和编码可以分别用于控制信号和条件，如图 5-49 所示。

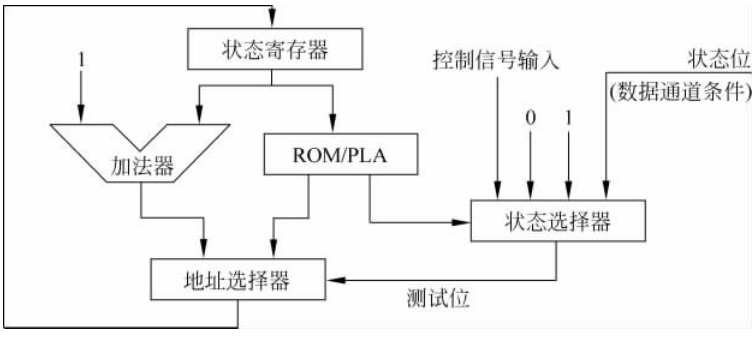


图 5-49　基本的控制器结构

用硬件描述语言描述状态机的一个特性是大量的设计者可用的有效状态机编码，主要为紧凑的设计优化的状态机编码，优化的编码也更适合高速设计的要求。设计过程中的许多变化必须根据设计者的优先选择来进行，但是，这也引入了人为错误的可能性，更别提在

识别状态机描述时，软件翻译的错误。经常地，直到综合已经执行后设计者也不能确定状态机编码是否需要优化以及如何优化。修改状态机编码是费时的事情，并且通常与状态机的功能无关。实际上，大多数综合工具可识别一个状态机，并且基于实际的设计约束对它重新编码，但是，高层次描述关于实现细节提供给综合工具的就是最灵活的优化方法。

5.4.5 逻辑级综合

逻辑级综合可以分成组合逻辑综合和时序逻辑综合，组合逻辑综合的输入是布尔方程描述的行为要求，时序逻辑综合的输入是某种类型的有限状态机的描述，通过逻辑级综合产生门级网表文件。

1. 组合逻辑综合

积之和形式的逻辑表达式可以利用两级与或逻辑阵列的硬件来实现，如果积之和形式的布尔表达式包含最少的乘积项和最少的字符变量，则积之和形式的布尔表达式是最简的。一个最简的积之和表达式对应一个两级逻辑电路，这个电路包含最少的逻辑门和最少的逻辑门输入数目。

表 5-2 给出了化简布尔代数表达式得到有效的硬件电路实现的重要定理，如果将布尔化简法所使用的定理嵌入逻辑综合工具的程序中，利用逻辑综合工具和程序就可以进行逻辑化简，并有效地实现两级和多级逻辑电路的综合。

表 5-2 化简布尔代数表达式的定理

定　　理	积之和形式	和之积形式
逻辑相邻性	$ab+ab'=a$	$(a+b)(a+b')=a$
吸收性	$aA+ab=a, ab'+b=a+b, a+ab'=a+b$	$a(a+b)=a, (a+b')b=ab, (a'+b)a=ab$
乘法运算与分解	$(a+b)(a'+c)=ac+ab'$	$Ab+a'c=(a+c)(a'+b)$
同一性	$ab+bc+a'c = ab+a'c$	$(a+b)(b+c)(a'+c)$

2. 时序逻辑综合

组合逻辑的输出只是其当前输入的瞬时函数，而时序逻辑的输出不仅取决于当前的输入，还与输入信号的历史有关，这种关联性可以用“状态”的概念来表示，并需要存储状态信息的存储元件。时序逻辑的综合工具只支持同步确定型时序电路的设计，此时可以利用一个公共的时钟作为同步其运行的同步信号，在通过电路传播信号时建立可预测的时间间隔，以便得到更可靠的设计和更简单的设计方法。

如果时序逻辑的两个状态对所有可能的输入序列都具有相同的输出序列，则认为这两个状态是等价的。时序逻辑的等价状态无法通过观察输出序列的异同对其加以区分，合并等价状态也不会改变状态机的输入-输出特性。通过识别和合并等价状态可以简化时序逻辑的状态表和状态转移图，而且无须考虑电路的综合功能就可以减少硬件开销，一般情况下，对每个有限状态机都有一个唯一的最简化的等价状态机存在。

由门级网表文件可以在给定的工艺中产生设计最终的实现，相应的综合取决于实现的工艺。如果采用全定制方式，要进行物理设计，产生版图，还要投片进行加工；如果采用 FPGA 的实现方式，利用 FPGA 的设计工具进行映射、布局和布线，完成设计的硬件实现。

本章小结

本章结合具体实例讨论了数字系统设计中编程风格和综合优化对硬件实现的影响，分析和讨论了数字系统同步设计的原理和要求，以及异步设计产生的问题和克服的办法，最后给出了数字系统在各层次综合的概念，以便对读者的设计编程有所帮助。

习题

5.1　判别以下哪种情况将综合产生组合逻辑的结果：

(1) 结构化的基本门网表；

(2) 一系列连续赋值语句中，带有反馈的条件操作符和无反馈的情况；

(3) 一个电平敏感的周期性行为中，隐含或不隐含对存储器结构要求的情况；

(4) 对于所有可能的输入数值给出了或没有完全给出输出赋值。

5.2　综合工具分别在什么情况下产生组合逻辑、透明的锁存器和沿触发的时序电路？

5.3　判别以下各种情况综合产生的结果对特权的处理：

(1) 在 Verilog 语句中，综合工具判别 case 语句中的分支选择项，当它们互不相同时，综合工具产生什么结果？

(2) 在 Verilog 语句中，case 语句是否通常隐含对首先解码的项赋予较高的特权？

(3) 在 Verilog 语句中，if 语句是否隐含了指定第一个分支有比其他分支较高的特权？

(4) 在 Verilog 语句中，当 if 语句中的分支是用互不相同的条件指定时，综合产生什么结果？

5.4　以下沿敏感行为中的寄存器变量是否将综合成一个触发器？

(1) 寄存器变量在行为描述的范围之外使用；

(2) 寄存器变量在未被赋值之前在行为描述中使用到它；

(3) 寄存器变量仅在行为描述动作的一些分支上被赋值。

5.5　确定状态机是由明确定义的状态寄存器和一个能够在输入作用下控制状态演变的逻辑构成的，对确定状态机采用以下编码风格是否可行？

(1) 用两个周期性行为描述确定状态机，一个电平敏感行为来描述下一状态和输出组合逻辑，一个沿敏感行为来同步状态的转移；

(2) 在电平敏感的周期性行为中，用阻塞赋值操作符(=)描述有限状态机的组合逻辑；

(3) 在边沿敏感的周期性行为中，用非阻塞赋值操作符(<=)描述有限状态机的状态转移和时序机数据通道的寄存器传输；

(4) 描述确定状态机的下一状态和输出组合逻辑的电平敏感行为时，要对所有可能的状态译码。

5.6　以下设计分割是否可行？

(1) 设计按照功能分割成较小的功能单元，每个功能单元都有一个公共的时钟域，并都能独立进行验证；

(2) 在分割中，功能相关的逻辑组合在一起，便于综合工具有可能利用共享逻辑，使模

块间的连线最短。单独优化在多个位置利用的模块，再根据需要对模块例化；

(3) 为使综合工具不受无关逻辑的影响对状态机进行逻辑优化，采用一个模块一个状态机的结构，不同时钟域对状态机相互作用的逻辑分割进不同的模块，同步器件放在信号在这些域中通过的地方；

(4) 对寄存器及其逻辑进行分组，有效地实现它们的控制逻辑。

5.7 阻塞赋值和非阻塞赋值有什么区别？为什么阻塞赋值通常要求相对大量的默认条件的操作？非阻塞赋值取代阻塞赋值会产生什么样的结果？

5.8 应该如何正确地使用 for-loop 语句？对于迭代次数由运算中的某个变量决定的 for-loop 循环结构，这种与数据有关的循环是否可以综合？在将循环动作分布到多个时钟周期后，是否可以被综合？

5.9 建立时间和保持时间与最大工作频率的关系？已知输入到寄存器、寄存器到寄存器和寄存器到输出的数据最大延时为 $B=7.445$ns，从时钟到目的寄存器的最短时钟路径为 $E=3.70$ns，从时钟到源寄存器的最短时钟路径为 $C=3.70$ns，寄存器时钟到输出时间 $t_{CO}=0.384$ns，寄存器固有建立时间和保持时间 $t_{SU}=t_H=0.18$ns，证明最大时钟频率为 124.86MHz。计算 I/O 的建立时间、I/O 保持时间和 I/O 时钟到输出时间。

5.10 为什么强调在数字系统设计中要采用同步设计？哪些技术可以解决同步设计中的异步问题？

5.11 数字系统的集成电路可以在哪些不同的级别上进行设计综合？结合第 1 章的数字系统结构了解各个级别上设计综合的特点。

5.12 高级综合的行为综合工具有哪些基本功能？