
Verilog
数字系统设计
教程

目　　录

第一章　Verilog 语言设计实践

随着科技的快速发展，数字电路设计工程师所面临的挑战发生了戏剧性的变化。设计过程变得更快，门电路的数量在不断增加，物理体积却在减小，芯片的引脚更多了，间距也更密了。然而，底层的设计要素却没有改变，将来也不会改变。所以设计者做出的设计必须遵循以下的要求：

- 清晰并易于理解。
- 逻辑的正确性。即设计必须准确无误地实现指定的逻辑功能。设计者应收集用户需求、器件参数并设计输入准则，然后做出满足最终用户需求的设计。
- 能在最恶劣的环境温度和工艺变化中正常运行。随着元器件的老化和温度的变化，电路元件的性能也会发生改变。温度的变化是由元器件自身或外界热源引起的。没有任意两个元器件是完全相同的，特别是在不同时间，不同的制造平台，或是在不同的设计规则下生产的元器件更是如此。器件与时序相关的诸变量，包括时钟偏移、寄存器建立和保持时间、传输时延和输出上升/下降时间等都必须给予考虑，做出说明。
- 可靠性。最终设计不能超出组件功耗的限制。每个元器件都有其工作的温度范围。例如，一个商业用途的器件其温度额定值是 0 ~ 70℃(32 ~ 160℉)。器件温度包括环境温度(即产品使用时周围空气的温度)、产品运行时因为内部热源的增加而产生温度和器件自身产生的温度。需注意内部温度的升高与门电路的数量及其逻辑状态改变的速度是成比例的。
- 不会产生满足工作需要及规定以外的 EMI/RFI(中文解释见书后附录，以下同)。
- 是可测试的且被证实满足用户需要。
- 不超出能量消耗的范围(例如，对电池组驱动的电路)。

无论设计的最终模式如何，也无论使用何种设计和测试工具，以上的要求都必须要遵守。

综　合

综合是高级设计描述向目标硬件的转换过程。在本书中，综合代表将 Verilog 代码转换为硬件可识别的设计网表的过程。

数字电路设计者的工作包括编写用于综合的 HDL 代码。这些 HDL 代码将在目标硬件中运行并规定产品的各项操作。设计者还需要编写激励代码用于测试系统输出。设计者编写的代码可读性强，而且要经过编译器的编译才能被硬件设备

最终识别。

为什么选用 HDL?

进行数字设计的方法可谓不少,例如:使用电路示意图。使用电路示意图容易实现更紧凑、更快速和更简明的 FPGA 设计。但是电路示意图方式不具备可移植性,而且在包含 10 或 20 层以上的设计中会变得难以处理。所以,对于大型可移植性系统而言,HDL 是最佳选择。

作为对比,程序列表 1-1 和程序列表 1-2 体现了其他书与本书 Verilog 设计的不同。

程序列表 1-1 其他书的设计

```
//将寄存器 b 的内容传送到寄存器 a
    a   <=   b;
```

程序列表 1-2 本书的设计

```
/* 作为在大规模设计中以 -3 速度级运行的器件部分,信号 b 需在 7.3ns
   的时间内变换为信号 a。此设计在旁置时电流应小于 80μA,在运行状
   态下应小于 800μA。整个设计在经过 CE 测试和近两个月的编写、调
   试、整合、编制文档并交付给用户以后,其成本应少于 1.47 美元。信
   号 a 必须与 75MHz 的系统时钟同步,可随全局系统复位而恢复。输入
   信号 b 应被设置在具有 208 条引脚封装结构的第 79 引脚或靠近第 79
   引脚处,这有助于满足寄存器 a 建立和保持时间的要求。 */
    a   <=   b;
```

为了说明设计过程,现再举一个常见的例子。此例说明了设计者在进行设计时所面临的问题。即使你对 Verilog 语言不熟悉,也无需担心,在本章稍后的篇章里对此将有详细说明。

1.1 小型过热探测器

以下是 Sarah(工程经理)写给 Sam(硬件设计师)的一封信:

收:sam@engineering

发:sarah@management

主题:热设计项目

一位客户要求设计一个装置:若按住开关或机器过热则红灯变亮,此装置由电池驱动。其最终成本不超过 0.02 美元,这样公司在以每个 9.95 美元出售时才能赢利。

首先,Sam 估计了设计的范围。根据以往的经验,他认为此系统与他去年所设计的电路极其相似。他计算了先前设计所用的门电路数目,并比较了这两个设计之间的差异,确定了这个系统需要 20 个门电路。他还考虑了系统应有的运行速度和其他重要因素,包括他在估计以前设计的复杂程度上可能出现的误差,甚至还考虑了他一周的假期和已定购的机票。他清楚地知道整个工作要包括设计、测试、整合和编制文档。他计算了所需的引脚数,包括一个按键输入、一个过热输入、一个过热输出、至少一个复位和一个时钟输入。根据他所估计的门电路数和引脚数,Sam 选择了一个拥有更多引脚数的设备和运行速度更快的 FPGA 器件以更好服务于用户需求的变化。Sam 把初步的时间计划和所选择的部件上报给老板后就开始工作了。老板对他的工作给予了肯定,并要求他尽快地完成设计工作,而且成本要足够低廉。

现在 Sam 已经完整地考虑了进行设计前的各个事项,可以开始进行设计了。他认为完成整个功能需要一个小型触发器电路,应该采用同步数字电路设计。Sam 编写的 Verilog 程序如下。

程序列表 1-3　过热探测器设计举例

```
module  overheat (clock, reset, overheat _ in, pushbutton _ in , overheat _
   out);
input  clock, reset, ocerheat _ in, pushbutton _ in;
output  overheat _ out;
reg  overheat _ out;
reg  pushbutton _ sync1, pushbutton _ sync2;
reg  ocerheat _ in _ sync1,ocerheat _ in _ sync2;
// 通常情况下,同步输入与系统时钟的相位无关。
// 外部信号使用双同步触发器以最大程度地减小系统的亚稳定性。
// 对于抖动和具有较慢的上升/下降时间的外部信号,进行过滤和锁存,
   效果会更好。

always @ (posedge clock or posedge reset )
begin
  if (reset)
```

```
        begin
        pushbutton_sync1 <= 1'b0;
        pushbutton_sync2 <= 1'b0;
        overheat_in_sync1 <= 1'b0;
        ocerheat_in_sync2 <= 1'b0;
        end
    else begin
        pushbutton_sync1 <= pushbutton_in;
        pushbutton_sync2 <= pushbutton_sync1;
        overheat_in_sync1 <= overheat_in;
        ocerheat_in_sync2 <= overheat_in_sync1;
        end
end

// 当用户发现过热并按下按钮时,锁存过热输出信号。

always @ (posedge clock or posedge reset )
begin
  if(reset)
      overheat_out   <=   1'b0;

// overheat_out 将一直保持(除非重启)。

   else if(overheat_in_sync2 && pushbutton_sync2)
       overheat_out <= 1'b1;
end
endmodule
```

第一个 always 部分看上去没有任何用处,好像应该可以删除。但是在以前的设计中,Sam 遇到过一些错误的逻辑行为(将在第二章中讨论),所以在实际应用中,他通常都采用双同步输入的方式。第二个 always 部分的作用是在 overheat_in_sync 和 pushbutton_sync 确定后再确定 pushbutton_out。

Sam 完成了设计中最有趣的部分:代码的实际设计。他很快就运行了编译器、仿真器以确定自己没有拼写错误和语法错误;然后他进行了测试,以检验自己的设计。其测试程序如程序列表 1-4。

程序列表 1-4 过热探测器测试程序

```
// 过热探测器测试程序。
// 由 Sam Stephens 编写。
`timescale 1ns/1ns
module oheat _ tf;
reg clock, system _ reset, overheat _ in, pushbutton _ in;
parameter clk _ period = 33.333;
overheat u1( clock, system _ reset, overheat _ in, pushbutton _ in, overheat _
  out);
always begin
    # clk _ period clock = ~clock; //产生系统时钟。
      end
initial
begin
      clock = 0;
      system _ reset = 1; // 复位确定。
      overheat _ in = 0;
      pushbutton _ in = 0;
 #75 system _ reset = 0;
end

// 切换输入确认 overheat _ out。
always
      begin
 #200 overheat _ in = 1;
 #100 pushbutton _ in = 1;
 #100 pushbutton _ in = 0;
 #200 overheat _ in = 0;
 #100 $ finish;
      end

endmodule
```

Sam 调用了他最喜欢的仿真工具检测了输出的波形,确认了输出逻辑的正确

性。输出波形如图 1-1 所示。

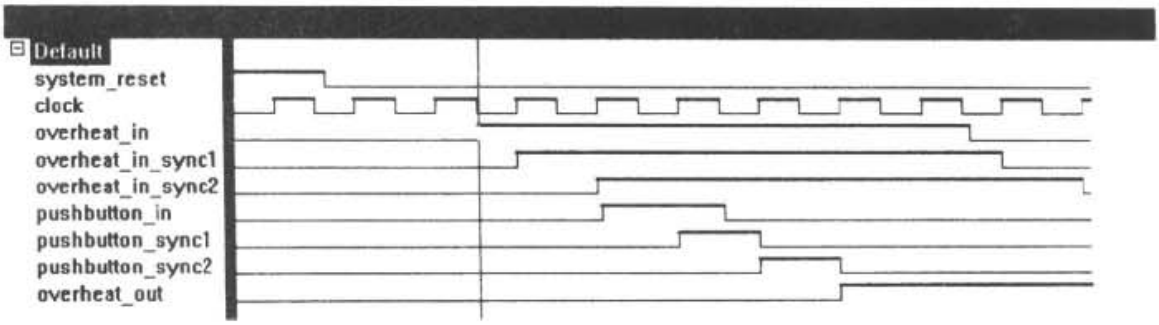

图 1-1 过热探测器测试输出波形

Sam 为他设计的器件分配了输入/输出引脚并规定了时序限制的条件。因为此系统不需要很快的运行速度,所以他选择了在许可范围内频率最低的石英振荡器来驱动时钟输入。这样,电池组的能量消耗最小。Sam 将其设计输入到 FPGA 编译器中进行校验,结果表明所选器件和时序限制条件完全符合要求。根据以往的经验,Sam 知道这个设计的运行速度不会有温度或 RFI 发射之类的问题,所以他发了电子邮件给老板,告诉她这个工作已完成了。

这个设计使用了 6 个触发器,看起来要做很多的事情,但实际上 Sam 是幸运的。他所选的器件、系统运行的速度都非常合适,该系统不涉及温度、EMI、RFI 等问题,而且用户的需求没有中途改变,他所使用的软件工具也没有崩溃。

关于工程进度表

一项管理制度能迫使工程师们免费加班。而工程师们虽然都有着丰富的经验,但在制定工作进度时依然会表现得过分乐观,所以他们常常会造成拖延。

对待这个问题,我们需要更加成熟:如果没有最后期限,什么工作都完不成。所以大多数工作在超时很少的情况下就能够完成。

无用输入和无用输出

很多人习惯于在彻底理解该设计要求前就进行编程。毕竟,对于一个工程师而言,编程是件乐事,而准备工作却截然相反。

但我并不在乎工作中能有多少乐趣,我建议在没有搞清楚你所想要的设计结果之前不要轻易地开始编程。

1.2 可综合的 Verilog 要素

Verilog 被设计成一种仿真语言,其很多要素都无需转换到硬件中。Verilog 是

一个大型的、完整的仿真语言，只有10%可以被综合；本节将就FPGA设计者所需要的这10%部分的一些基本特性进行说明。

Verilog要素中哪些是可被综合的？这是综合工具软件提供商所面临的设计问题。通常情况下，一个"非官方"的Verilog语言要素的子集是所有提供商都支持的，但是目前的Verilog描述并不包括任何可综合的语言要素。IEEE的一个工作组目前正在撰写一个名为IEEE Std 1364.1RTL的综合子集的规范，该规范将定义一个最小的可综合的Verilog语言要素的子集。此规范一旦发布，肯定会得到用户和综合工具提供商的拥护。

Verilog看起来与C语言很相像，但是请牢记C语言定义的是顺序过程（每次只有一个过程的一行代码被执行）；而Verilog则定义了顺序过程和并行过程。程序列表1-5列出了一些可以综合的Verilog语言要素所编写的代码段。

程序列表 1-5　Verilog程序范例

```
module hello ( in1, in2, in3, out1, out2,clk,rst,bidir_signal, output_ena-
    ble);// 见注释 1。
/* 见注释 2。跨越多行的注释可以如此标记。*/
input in1, in2, in3, clk, rst, output_enable; // 见注释 3。
output out1, out2;
inout bidir_signal;
reg out2; // 见注释 4。
wire out1;

assign out = in1 & in2; // 见注释 5。
assign bidir_signal = output_enable ? out2:1'bz ; // 见注释 6。

always @ (posedge clk or posedge rst) // 见注释 7。
    begin // 见注释 8。
    if (rst) out2 <= 1'b0; // 见注释 9。
    else out2 <= ( in3 & bidir_signal);
    end
endmodule
```

注释1：模块的第一个要素是模块名称。在本书中，模块的名称与文件的名称（扩展名为.v）是相同的，而每个文件只包括一个模块。这种做法并不是必要的，但

它有助于设计结构的清晰。

紧接在模块/文件名后的是端口列表。此列表列出了连接本模块和其他模块及外部世界的所有信号。而未在此列表中列出的信号则只属于该模块，并不与其他模块相连。Verilog 语言中用分号作为分隔符，但注意并非所有的行都以分号作为结束，尤其是编译指令(always 语句，if 和 case 语句等等)。

注释 2：说明和注释语句可写在“//”后面或在“/*”和“*/”之间。“/*”和“*/”必须成对出现，“/*”和“/*”的组合被认为是错误的。

注释 3：端口方向列表跟在端口列表之后。此列表定义了模块的输入端口、输出端口和双向端口。端口列表中的所有信号都是线网型。线网与印刷电路板上的内部连线相似。

注释 4：所有的信号可分为两类：线网型(即与电路板上的印刷线路相似的互连线)和寄存器型(即存储在锁存器或触发器中的信号)。线网型信号可被寄存器或组合赋值驱动。连接一个模块中的两个寄存器是不合法的。Verilog 中将代码中未定义的信号默认为 1 比特的线网型信号。

注释 5：强制赋值语句应是连续的(组合的)逻辑语句。

注释 6：赋值语句 bidir _ signal 使用了条件赋值，如果 output _ enable 为真，则 out2 的值赋给 bidir _ signal；反之，bidir _ signal 将呈三态 z 。

注释 7：always 程序块是按顺序执行的。在符号@后、圆括号中的信号列表称为事件灵敏度表。综合工具将从此表中提取出程序块的控制信号。灵敏度表的出现源于 Verilog 仿真的需要。仿真器保存了一个被监控信号的列表以减小仿真模型的复杂度，因为只有灵敏度表中的信号发生变化，逻辑赋值才会随之变化。当某些信号被提取为控制信号时，这些输入信号都必须出现在灵敏度表中。若此表不完整，则编译器会发出警告。

灵敏度表可以是一个信号列表(表中列出的任何信号都能被检测和使用)，这些信号可以是上升沿触发或下降沿触发的。如果一个上升沿触发或下降沿触发的信号被用作控制信号，那么该程序块中的所有控制信号都应是此类信号。

注释 8：begin /end 命令用来分离代码片断。如果代码段只使用一个分号，则 begin /end 是可选的。

注释 9：在 always 部分我们使用了“<=”进行赋值。若是使用“=”，则这些命令的顺序可能引起非预料的寄存器的综合，使得当变量更新时其值可以被保存。通常情况下，设计者希望时序(常用)程序块的各个要素能够同时更新，这样，“<=”赋值语句才能模拟 clock-to-Q(从出现时钟信号到出现输出信号)延时。这个延时将保证级联触发器(像移位寄存器)按预期工作。

若按照此种方式(即在同步模块中使用非同步信号)编写代码，则 rst 输入将作为异步复位信号。这并不是 IEEE Std 1364 的规定，只是一种习惯用法。

Verilog 语言要素对大小写是有区分的(例如 X 和 x 不相同)。同 C 语言一样，Verilog 对空格的用法比较随意，设计者可用空格来增加程序的易读性。但要注意，像下列的语句在语法上虽说是正确的，但可读性却很差。

```
a = b&c; d = e&f; g = h | I; j = k^m; n = o&p;
```

可移植的 Verilog 代码

在编译器中编译成功的代码在不同的硬件中运行应该有相同的结果，这是对 Verilog 程序的要求。不幸的是，在编写高性能(即所设计的系统要以高速运行)、高效率的(完成用户所需功能并使用最少的硬件资源)代码时，设计者必须使用结构化和编译器特定的命令。其可移植性通常不是实际设计要求，而是设计者应努力追求的目标。

设计者不能忽略操作符的优先级，可用圆括号来区分不同的优先级。使读者能从源程序中直接读出优先级，而不用强记或费力去查找。不要采用太复杂的结构，尽量使用简单和清晰的编码风格。程序列表 1-6 和程序列表 1-7 表现了具有相同编码结构和不同编码风格的两个程序。

程序列表 1-6　Casex 代码段(隐含"don't cares")

```
//使用隐含的"don't cares"。
reg     [ 7:0 ] test _ vector;
casex     (test _ vector)
8'bxxxx0001:
  begin
//此处插入代码。
//这种编码风格导致了并行的 case 结构(MUX)。
  end
endcase
```

程序列表 1-7　显式"don't care"代码段

```
// 使用显式的 "don't cares"。
reg     [ 7:0 ] test _ vector;
if     (test _ vector[ 3:0 ] = = 4'b0001)
    begin
```

```
//此处插入代码。
//这种编码风格导致了并行的 case 结构(MUX)。
end
```

使用 Verilog 的设计者需要首先判断是否采用优先编码结构或是 MUX 结构。内嵌的"if-then"语句将产生优先级编码式逻辑结构,而 case 语句则会产生 MUX 逻辑结构。在后面的章节中将对此进行详细论述。

不要想当然地把 Verilog 寄存器看成某种形式的触发器。在 Verilog 中,寄存器只是一种存储设备。这是 Verilog 设计者首先要掌握的第一个特点。一个寄存器可能会被综合为一个触发器(数字构建)、一个锁存器(模拟构建)或一根导连线,甚至可能在优化时被去掉。Verilog 中假设一个非显性变化的变量应该保持原值。与 Altera 的 AHDL 相比这是个优点,因为在 AHDL 中未记载的变量是要被清除的。所以设计者必须构造代码以便能被目标硬件设备的结构综合,同时还要注意锁存器被综合的可能性。Verilog 不要求综合器使用某种结构的指令。使用综合工具提供商定义的协议,在所有的输入都完全定义后,综合器就能得到正确的解释。

1.3 Verilog 的层次

一个 Verilog 设计实际上包括一个高层模块和一个或多个低层模块。高层模块可被仿真模块调用,将测试激励用于器件的引脚。高层器件模块通常包括与外界(设备引脚)相连的端口列表和低层模块之间的内部联结,以及控制双向 I/O 引脚和三态引脚的多态逻辑。模块实例定义如下:

```
module _ name instance _ name (port list);
```

例如,程序列表 1-8 中的代码段就描述了四个基本门电路实例。其综合后的电路示意图见图 1-2。

程序列表 1-8 结构化实例

```
module    gates (in1,in2,in3,out4);
input     in1, in2, in3;
output    out4;
wire      in1, in2, in3, out1, out2, out3, out4;
and       u1 (out1, in1, in2); // 结构的(原理图状的)。
or        u2 (out2, out1, in3); // 结构。
xor       u3 (out3, out1, out2);
```

```
not     u4 (out4, out3);
endmodule
```

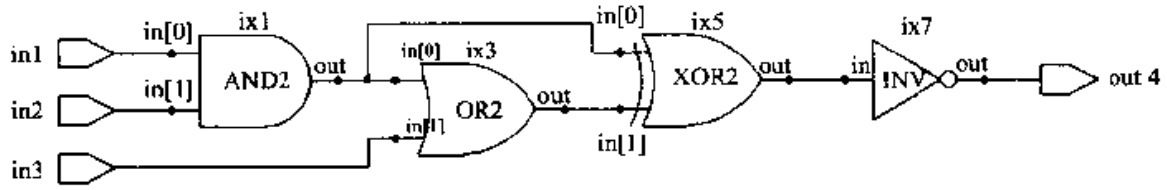

图 1-2 门电路示意图

本例使用了位置赋值。信号按照原始模块端口列表中的顺序进行连接。通常情况下,设计者要对端口列表进行剪切和拷贝以保证此顺序的完全相同。原始端口列表的一个要求是在表中先列输出,再列输入。

模块端口列表也可以使用命名赋值(需要位置赋值的原始模块例外),此时端口列表中的信号顺序是随机的。对于命名赋值,其格式是较低等级的信号名称(较高等级模块信号的名称)。程序列表 1-9 中的范例包括了命名赋值和位置赋值。

程序列表 1-9 命名和位置赋值实例

```
module      and _ top;
wire        test _ in1, test _ in2, test _ in3;
wire        test _ out1, test _ out2;
//无需考虑端口顺序时使用名称赋值。
user _ and u1 ( .out1(test _ out1), .in1(test _ in1), .in2(test _ in2));

//位置赋值。
user _ and u2 ((test _ out2, test _ in2), (test _ in3));
endmodule

module      user _ and (out1, in1, in2);
input       in1, in2;
output      out1;

assign      out1 = ( in1 & in2);
endmodule
```

1.4 内建逻辑原语

表 1-1 至表 1-12 描述了 Verilog 两输入端的功能。Verilog 原语并不只局限于两个输入，具有更多逻辑输入的原语可以从下列各表进行推算。

表 1-1 与门(AND gate logic)

与	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

表 1-2 与非门(NAND gate logic)

与非	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

表 1-3 或门(OR gate logic)

或	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

表 1-4 或非门(NOR gate logic)

或非	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

表 1-5 异或门(XOR gate logic)

异或	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

表 1-6 同或门(XNOR gate logic)

同或	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

表 1-7 缓冲门(buf gate logic)

输入	输出
0	0
1	1
x	x
z	x

表 1-8 非门(NOT gate logic)

输入	输出
0	1
1	0
x	x
z	x

表 1-9 低电平使能三态缓冲器(bufif 0 gate logic)

bufif 0	control = 0	control = 1	control = x	control = z
data = 0	0	z	x	x
data = 1	1	z	x	x
data = x	x	z	x	x
data = z	x	z	x	x

表 1-10 高电平使能三态缓冲器(bufif 1 gate logic)

bufif 0	control = 0	control = 1	control = x	control = z
data = 0	z	0	0 或 z	0 或 z
data = 1	z	1	1 或 z	1 或 z
data = x	z	x	x	x
data = z	z	x	x	x

表 1-11 低电平使能三态非门(NOTif 0 gate logic)

notif 0	control = 0	control = 1	control = x	control = z
data = 0	1	z	1 或 z	1 或 z
data = 1	0	z	0 或 z	0 或 z
data = x	x	z	x	x
data = z	x	z	x	x

表 1-12 高电平使能三态非门(NOTif 1 gate logic)

notif 1	control = 0	control = 1	control = x	control = z
data = 0	z	1	1 或 z	1 或 z
data = 1	z	0	0 或 z	x 或 z
data = x	z	x	x	x
data = z	z	x	x	x

程序列表 1-10 中的代码段说明了这些门电路的使用,图 1-3 是其示意图。

程序列表 1-10 初始化结构门的实例

```
module  struct1 (out1, out2, out3, out4, in1, in2, in3, in4, in5, in6,
buf _ control);
output  out1, out2, out3, out4;
```

```
input  in1, in2, in3, in4, in5, in6, buf_control;
bufif0 buf1(out1, in1, buf_control);
and and1(out2, in2, in3);
nor nor1(out3, in4, in5);
not not1(out4, in6);
endmodule
```

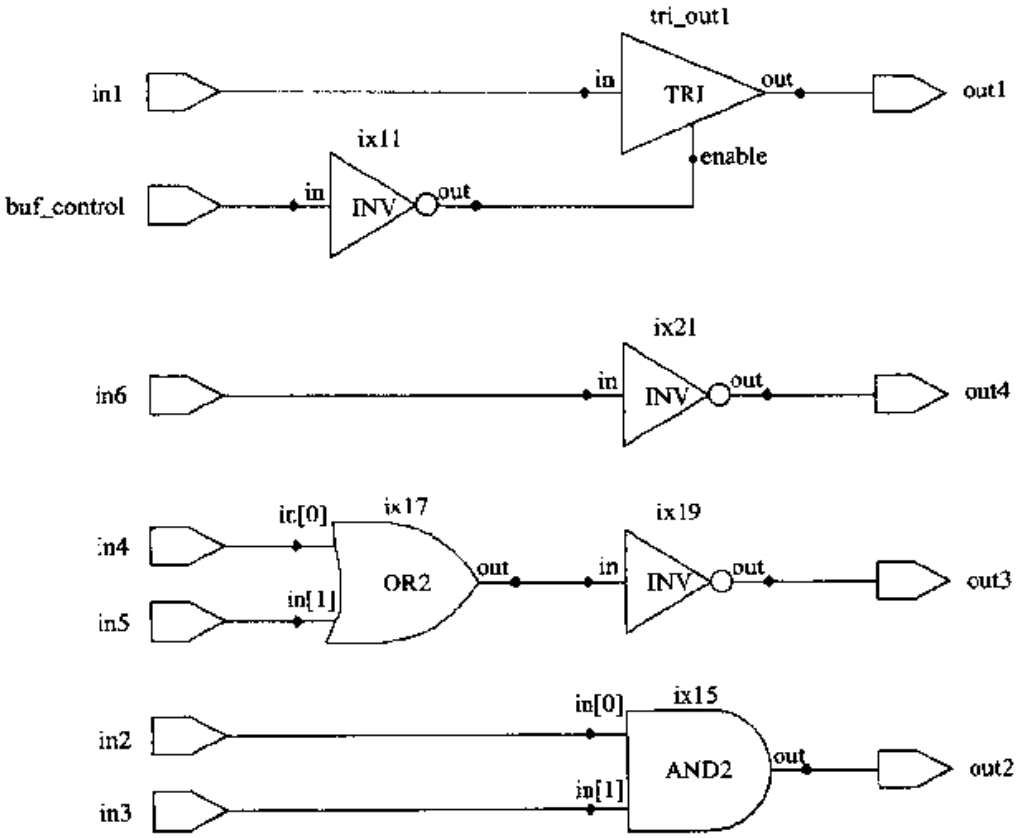

图 1-3 结构化门电路的示意图

1.5 锁存器和触发器

从技术的角度而言,触发器被定义为双稳态的多谐振荡器。多谐振荡器是具有两个或多个输出的模拟电路,且所有的输出中一次只能打开一个输出。双稳态的含义是每个输出都是二进制的数字信号,都具有两种输出状态:高电平或低电平。我们现在已将其扩展为三态(还包括高阻状态)。

触发器有很多种,但通常我们所讨论的都是 D 触发器,表 1-13 显示了一个普通的边缘 D 触发器的功能。注意表 1-13 中的启动和复位都是低电平有效。

表 1-13　7474 型 D 触发器的逻辑描述

/set	/reset	clock	data	Q	/Q	
0	1	x	x	1	0	
1	0	x	x	0	1	
0	0	x	x	1	1	注释
1	1	r	1	1	0	
1	1	r	0	0	1	
1	1	0	x	n	n	
1	1	1	x	n	n	

注释:此状态是不稳定的且是非法的。如果/set 和/reset 输入同时被去掉,则输出状态是未知的。
x = 不关心(无关)。
r = 时钟信号上升沿。
n = 不改变,保持先前状态。

典型的 FPGA 逻辑设计允许使用异步置位或复位,但不能同时采用。所以我们不用担心错误的输入。本书将重点强调同步设计技术,所以除了加电初始控制以外,我们不鼓励使用触发器异步置位或复位输入。

锁存器具有更多的模拟功能。需要牢记的是所有组成数字逻辑的底层电路都是模拟电路。基于晶体管的触发器加上正反馈构成的即是锁存器。

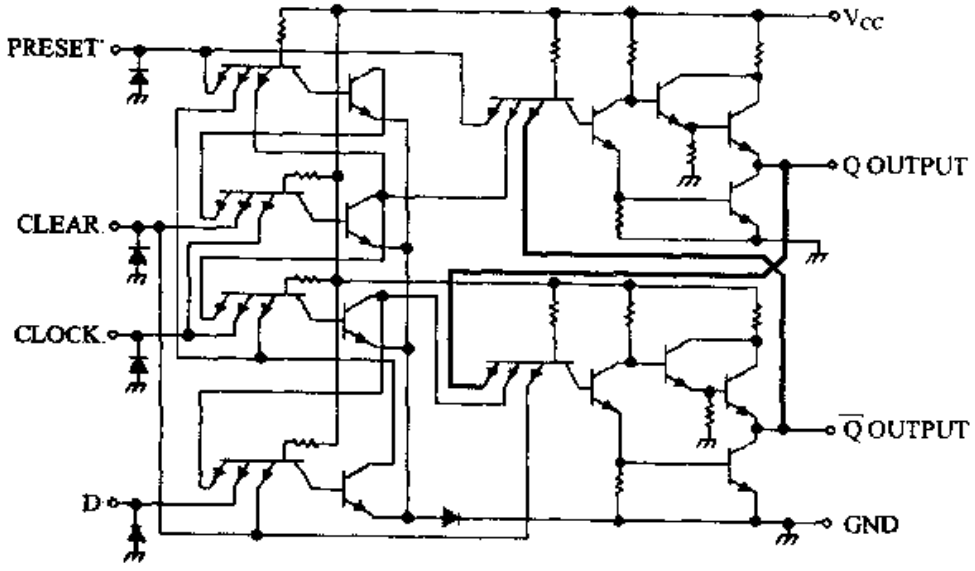

图 1-4　典型 CMOS D 触发器电路示意图

首先必须注意的是此 D 触发器是由晶体管等线性元件构成的。如果你牢记一点:所有的数字电路都是由模拟元件构成的,那么你就有可能成为最优秀的数字电路设计者。其次,要注意从 Q 或/Q 输出的反馈(注意高亮信号)。反馈信号能使

触发器保持其状态。

如果你更喜欢门电路的形式，图 1-5 给出了同一个 D 触发器的门电路示意图。这种形式的示意图是更高层次的抽象，晶体管和电阻器都被隐藏。但是，需要注意图中加粗表示的反馈路径。

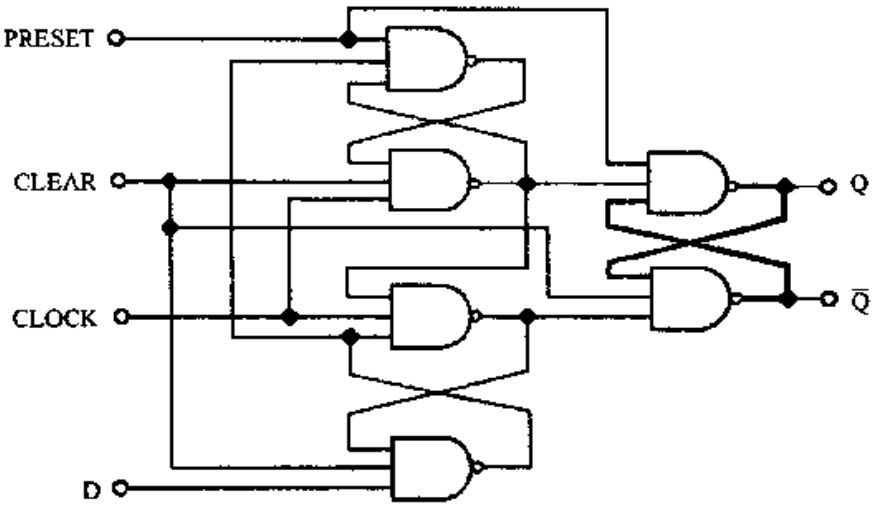

图 1-5 典型 CMOS D 触发器电路示意图(门电路)

程序列表 1-11 是一个锁存器的 Verilog 方案，图 1-6 是该 Verilog 设计的电路示意图。示意图中的电路使用了 RS 锁存器(LATRS)，其功能与图 1-5 中的电路相似。此电路不是数字电路。

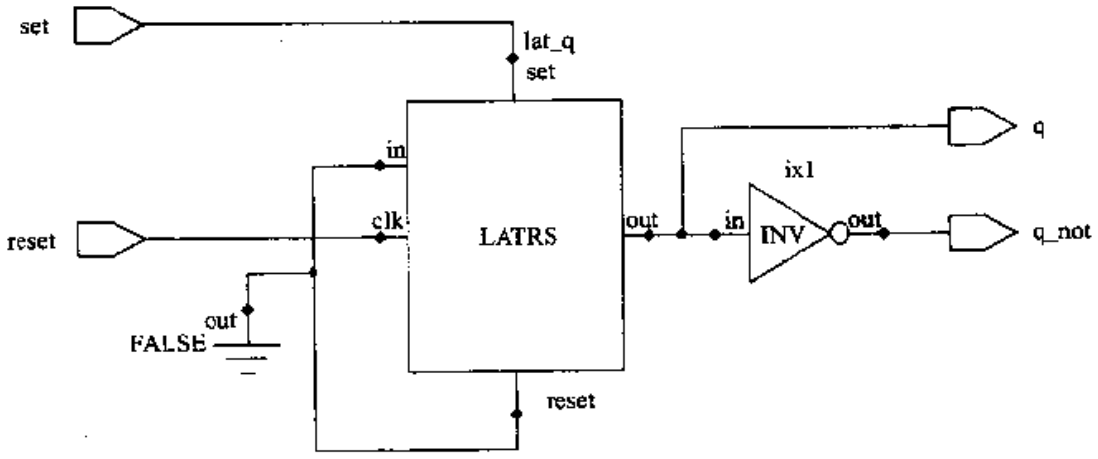

图 1-6 锁存器(利用触发器实现的)示意图

程序列表 1-11 锁存器的 Verilog 代码

```
// 基本寄存器。
// 这是一种不好的编程风格:不要采用这种方式来创建锁存器。
module latch(q, q_not, set, reset);
output q, q_not;
input set, reset;
reg q;

wire set, reset;

assign q_not = ~q;

    always @ (set or reset)
    begin
        if(set)
        q = 1;
        else if(reset)
        q = 0;
    end
endmodule
```

此锁存器使用反馈来控制其状态,隐含在程序列表 1-11 中的反馈是根据输入状态所定义的 q 来决定的,而这种 q 并不是根据所有输入状态的组合来确定。对于未定义的输入,q 将保持其以前的状态不变。决定锁存器输出状态的逻辑包括时钟信号等,它是电平敏感而不是边缘触发的结构。

程序列表 1-12 创建一个锁存器的 Verilog 代码

```
module lev_lat(test_in1, enable_input, test_out1);
input test_in1, enable_input;
output test_out1;
reg test_out1;

always @ (test_in1 or enable_input)
```

```
if (enable_input) begin
    test_out1 <= test_in1;
    end
endmodule
```

在此例中，只有在 enable_input 为高电平时 test_out1 才发生变化，然后 test_out1 将跟随 test_in1 的状态。这可以综合到一个组合锁存器中，如图 1-7 所示。除非在该锁存器由同步电路驱动同时也驱动同步电路的情况下（将导致伪同步设计），否则我们不鼓励这样的编码风格。

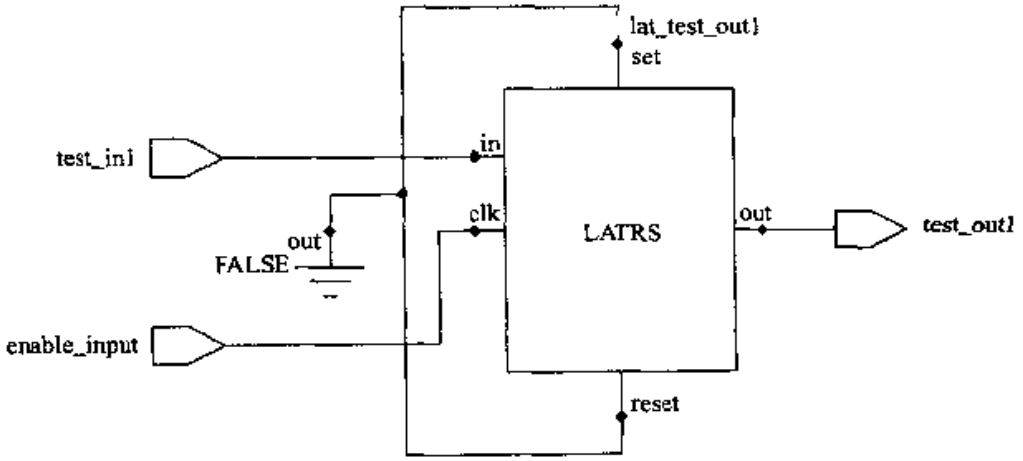

图 1-7 锁存器电路示意图（复位-启动型）

锁存器是否是一个好的设计构建？这要看设计者的意图。如果设计者希望生成一个锁存器，那么锁存器的构建就很好。如果设计者不希望这样做，那么锁存器就是不好的构建。总之，我们对锁存器最好是抱着怀疑的态度仔细考查。

一个较好的设计可以从中推断出时钟控制的触发器的结构，如程序列表 1-13 所示，其电路示意图见图 1-8。

程序列表 1-13 带有同步复位端的级联触发器

```
module edge_lat (clk, rst, test_in1, enable_input, test_out2);
input  clk, rst, test_in1, enable_input;
reg  test_out1, test_out2;
output  test_out2;

always @ (posedge clk or posedge rst)
```

```
begin
  if (rst) test_out1 <= 0;
  else if (enable_input) begin
          test_out2 <= test_out1;
          test_out1 <= test_in1;
  end
end
endmodule
```

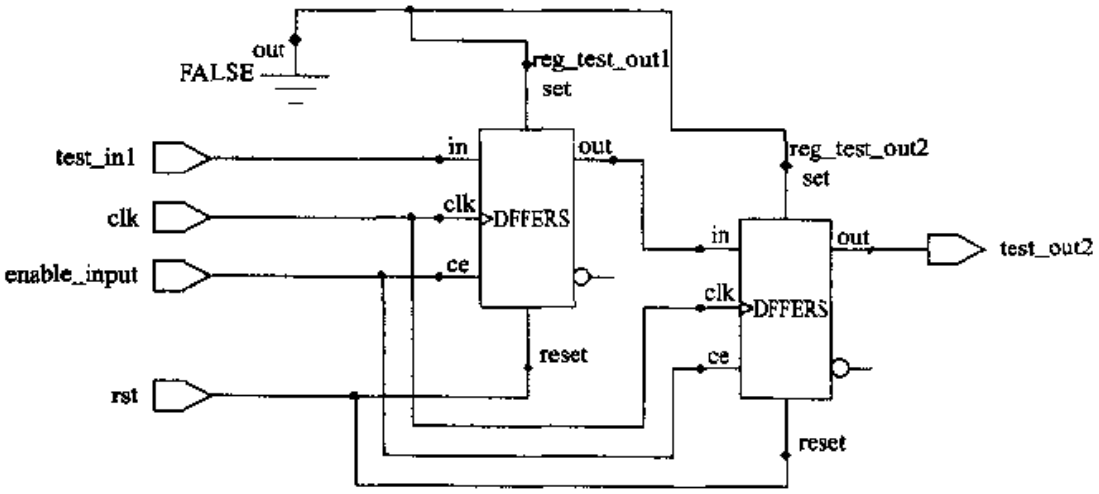

图 1-8　同步复位级联触发器电路示意图

程序列表 1-13 表明了一个同步复位触发器，其复位输入只在时钟边沿才有效。如果目标硬件不支持同步复位，在需要复位时，应该增加一个逻辑信号以使 D 输入为低电平，如图 1-9 所示。程序列表 1-14 给出了一个异步复位触发器的相关代码，其 rst 信号被连续检测。注意触发器专用的全局置位/复位资源(GSR)并没有被使用。综合一个同步复位信号并将其连至 GSR 是效率较高的方法。这种用法详见第五章。

程序列表 1-14　异步复位触发器的 Verilog 代码

```
module edgetrig (clk, rst, test_in1, enable_input, test_out2);
input   clk, rst, test_in1, enable_input;
reg test_out1, test_out1;
output   test_out2;
always @ (posedge clk)
```

```
begin
        if (rst)
          test _ out1 <= 0;
        else if (enable _ input) begin
          test _ out2 <= test _ out1;
          test _ out1 <= test _ in1;
        end
end
endmodule
```

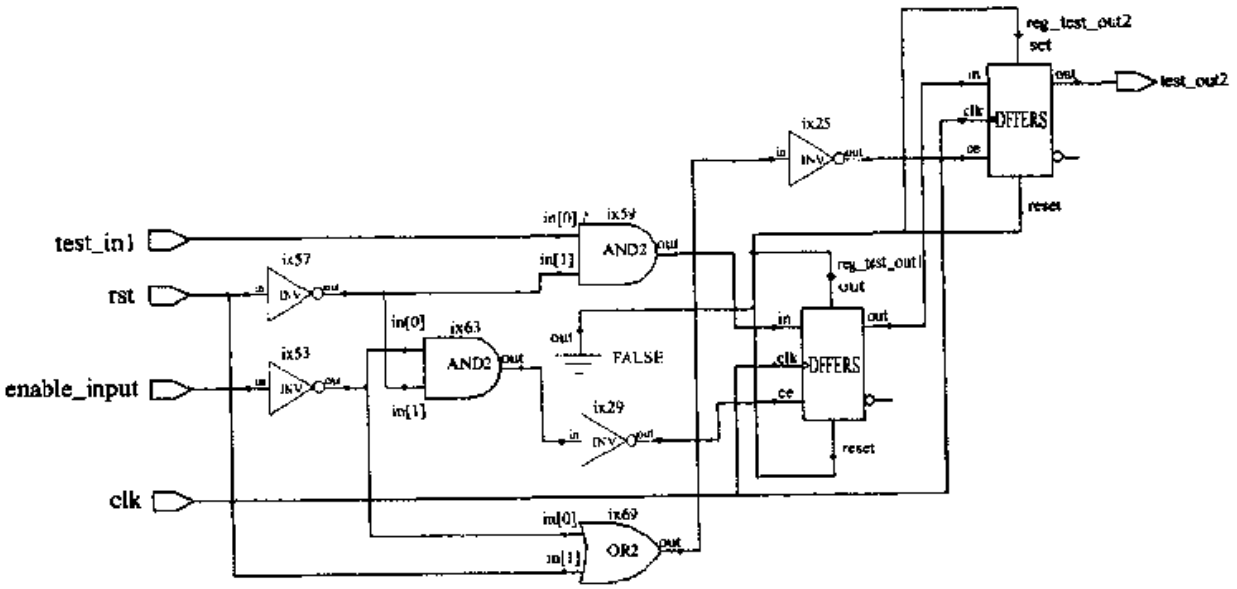

图 1-9 同步复位级联触发器电路示意图

1.6 阻塞性赋值与非阻塞性赋值

我们目前已经使用了非阻塞性赋值(<=)。对于阻塞性赋值(=),当变量在 always 语句外定义时,当前的赋值必须在以前的赋值完成之后才能进行。而硬件则是通过对原数值的锁存来延迟赋值的。这意味着阻塞性赋值是有顺序的,它在 begin/end 中被顺序执行。

程序列表 1-15 阻塞性赋值实例 1

```
/* 第一个阻塞性赋值语句必须在其后的赋值以前完成。本例中,创建了两
   组触发器(见图 1-10),因为 data _ out 要由中间变量来创建。*/
   module blocking (clock, reset, data _ in, data _ out);
```

```
input     clock, reset;
input     data _ in;
reg     data _ temp;
output     data _ out;
reg     data _ out;

    always @ (posedge clock or posedge reset)
    if (reset)
    begin
        data _ out   =   0;
        data _ temp   =   0;
    end
    else
    begin
        data _ out   =   data _ temp;
        data _ temp   =   data _ in;
    end
endmodule
```

与程序列表 1-15 相对应的综合逻辑电路见图 1-10，图中 data _ temp 和 data out 是阻塞性赋值：由一个触发器来创建中间变量 data _ temp。

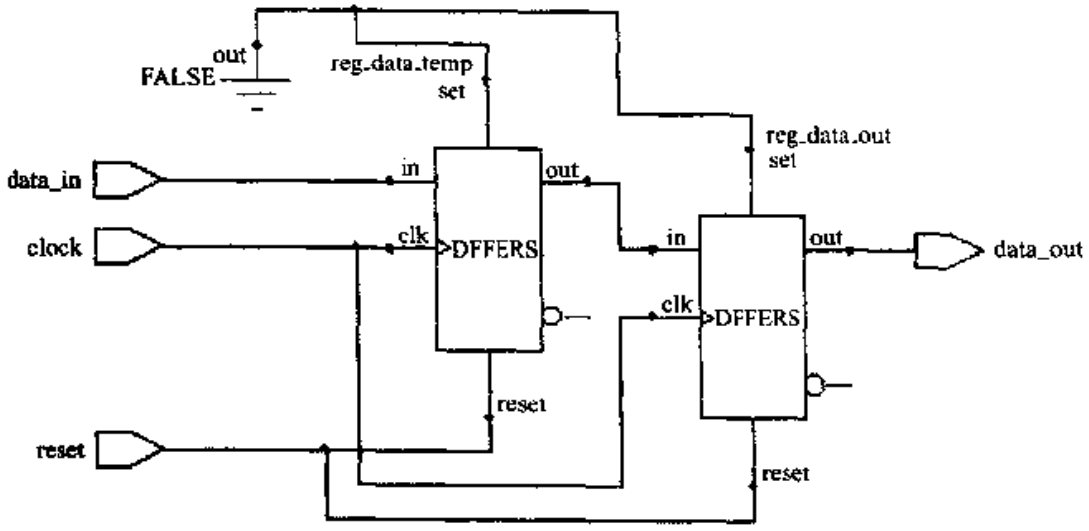

图 1-10 阻塞性赋值实例 1

在程序列表 1-16 中，阻塞性赋值语句是颠倒的。注意，图 1-11 中的结果与图 1-10 中的结果不同。

程序列表 1-16 阻塞性赋值实例 2

```
/* 阻塞性赋值语句是颠倒的，使 data_temp 变量冗余，则 data_temp 可获得
   优化。只需创建一组触发器(见图 1-11)，因为中间量是阻塞性的，所以不
   用创建 data_out。*/
 module  block2(clock, reset, data_in, data_out);
    input  clock, reset, data_in;
    reg data_temp;
    output   data_out;
    reg data_out;

    always @ (posedge clock or posedge reset)
    if (reset) begin
       data_out   =   0;
       data_temp   =   0;
    end
    else
    begin
       data_temp   =   data_in; //转换命令。
       data_out   =   data_temp;
    end
endmodule
```

关于阻塞性赋值和非阻塞性赋值

在同一个 always 模块中出现的多个阻塞性赋值语句的顺序如何排列是非常重要的。但非阻塞性语句的使用可以避免语句顺序的问题并可创建触发器。

如果我们用非阻塞性赋值来替换阻塞性赋值，那么指令的顺序就不再重要。所有的赋值语句的右侧项都在时钟上升沿进行计算，所有的赋值都可以在同一时间进行。程序列表 1-17 对应的逻辑电路图如图 1-12，此例中 data_temp 和 data_out 是非阻塞性赋值，但其结果与程序列表 1-16 等价。

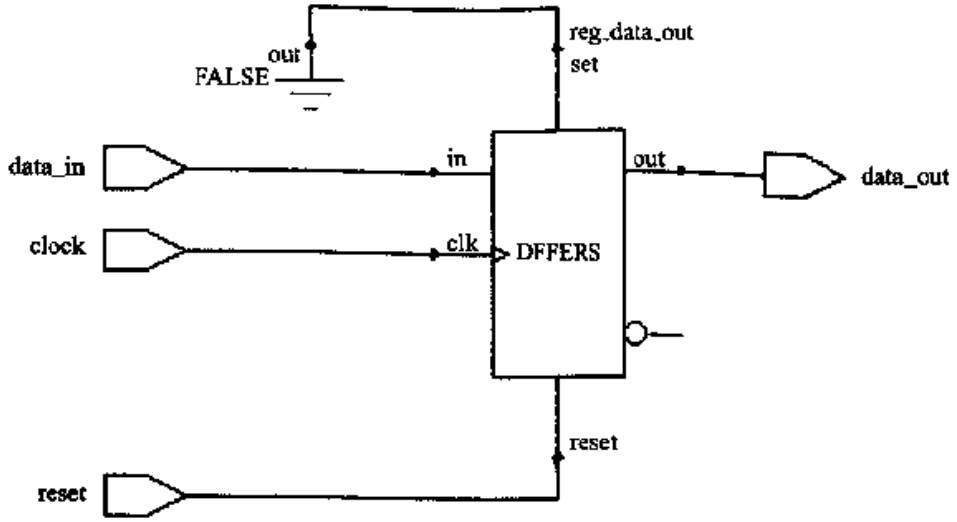

图 1-11 阻塞性赋值实例 2

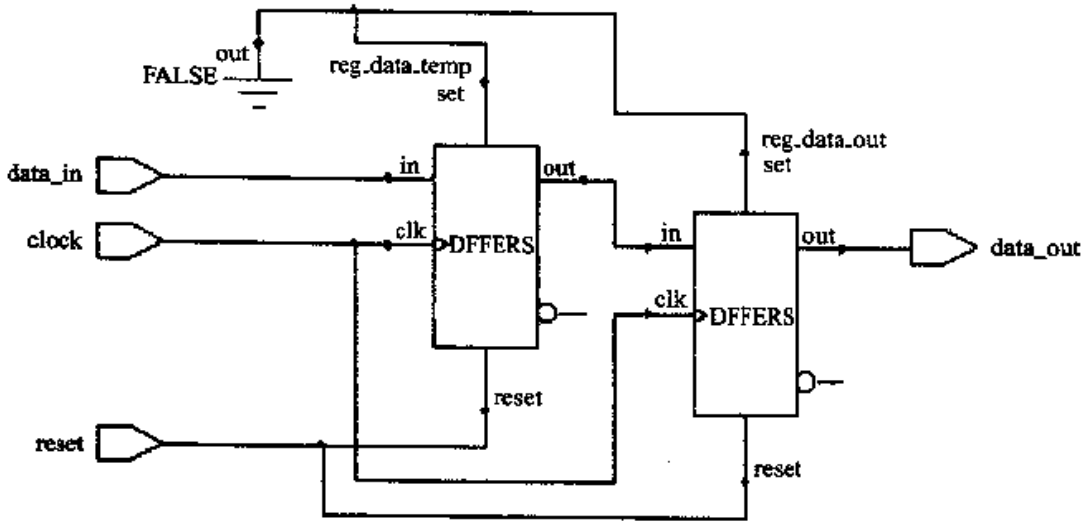

图 1-12 非阻塞性赋值实例 3

程序列表 1-17 非阻塞性赋值举例

```
// 非阻塞性逻辑实例。
// 非阻塞性赋值语句的顺序并不重要。

module nonblock(clock, reset, data_in, data_out);
input  clock, reset;
input  data_in;
```

```
reg   data _ temp;
output   data _ out;
reg   data _ out;

    always @ (posedge clock or posedge reset)
    if (reset)
    begin
        data _ out   <=   0;
        data _ temp   <=   0;
    end
    else
    begin
        data _ out   <=   data _ temp;
        data _ temp   <=   data _ in;
    end
endmodule
```

1.7 Verilog 语法

1.7.1 数

除非设计者重新定义,Verilog 数一般都默认为 32 位。Verilog 中数的格式是“大小'基数”(见下面实例)的形式。单引号 '用于 Verilog 数的表示,而单引号‘用于标明文本的替换和编译器指令,两者不能混淆。在 Verilog 中为增强可读性而在数字下面添加下划线是允许的。所有的数都必须从左边开始用 0,x's 或 z's (如果最左边的值定义为 x 或 z)来填充。如果一个数没有给出具体长度,则认为其大小满足运算中各数值的大小。X 或 x 是不定态,Z 或 z 为高阻态。Verilog 允许用?来代替 z。没有明确基数的数被认为是小数十进制数。除了被线网型数据驱动的,其余的线网型数据都将被认为是 Z。

数举例:

```
1'b0        // 1 位,值为 0。
'b0         // 32 位,全为 0。
32'b0       // 32 位,全为 0,
            // ( 0000 _ 0000 _ 0000 _ 0000 _ 0000 _ 0000 _ 0000 _ 0000 )。
```

```
4'ha      // 4位十六进制数(1010)。
5'h5      // 5位十六进制数(00101)。
4'hz      // zzzz。
4'h? zz?      // zzzz? 与 z 的意义相同。
4'bx      // xxxx。
9      // 32位数(从左边开始,前28位用0填充)。
a      // 非法数。
```

Verilog是一种松散的类型定义语言。例如,它可以接受一个类似4'hab的8比特数而不显示任何错误信息(即该数的前半个字节会被忽略而被识别为1011或b)。所以Verilog设计者对此类错误要格外注意。

1.7.2 逻辑非运算(NOT)的格式

! 是逻辑非运算符,其结果是一个单比特值:真(1)或假(0)。~ 是位逻辑非运算符。我们可以使用 ! 来转换一个单比特数值,其结果与使用 ~ 是一样的,但这是个坏习惯。当一个单比特数被转换为多比特向量时,这两个操作数将不再等价,而这是个不易发现的错误。

程序列表 1-18 非运算实例

```
module negation (clk, resetn);
input          clk, resetn;
reg [3:0]      c,d,e;

always @ (posedge clk or negedge resetn)
begin
    if ( ~ resetn)     //异步复位(低有效)。
    begin
    c   <=   5;// 这是一种不太好的异步设置数值的形式,
               //这个值称为魔数,其实应该是一个参数。
    d   <=   0;
    e   <=   0;
    end
    else
    begin
```

```
        d   <=   ! c;// d 被赋值为 0;
        e   <=   ~ c;// e 被赋值为 1010。
        end
    end
    endmodule
```

1.7.3 逻辑与/逻辑或运算(AND/OR)的格式

& 是与运算符。单 & 是位逻辑与运算符，&& 是逻辑(真/假)与运算符。这两种运算符格式在功能上并不相同。见程序列表 1-19。

程序列表 1-19 逻辑与运算和位逻辑与运算实例

```
a = 4'b1000  &   4'b0001 ;  //a = 4'b0000。
b = 4'b1000  &&  4'b0001 ;  //b = 1'b0。
```

| 是或运算符，单 | 是位逻辑或运算符。|| 是逻辑或运算符。这两种运算符格式在功能上不相同。见程序列表 1-20。

程序列表 1-20 逻辑或运算和位逻辑或运算实例

```
a = 4'b1000  |  4'b0001;  //a = 4'b1001。
b = 4'b1000  ||  4'b0001;  //b = 1'b1。
```

程序列表 1-21 与/或运算实例

```
module and_or (clk, resetn, and_test, or_test);
    input        clk, resetn, and_test, or_test;
    reg          a;
    reg[3:0]     b;
    reg[3:0]     c;
    reg[3:0]     d;
    reg[3:0]     e;
    reg[3:0]     g;

    always @ (posedge clk or negedge resetn)
```

```
begin
    if (~resetn)  //异步复位(低有效)。
    begin
    a   <=   0;
    b   <=   4'd4;  //这是一种不好的异步数值设置格式。应该
                    是一个参数。
    c   <=   4'd5;
    d   <=   0;
    e   <=   0;
    g   <=   0;
    end

    else if(and_test)
    begin
    d   <=   (c && ! a);        //d赋值为0。
    e   <=   (c & ! a);         //e赋值为1010。
    g   <=   (b & c);           //g赋值为0100。

    end
else if (or_test == 1) // 与 or_test 完全等价。
    begin

    e   <=   (c | ! a);  //e赋值为1111。
    g   <=   (b | c);  //g赋值为0101。
    end
    else
    begin
    d   <=   0;  // 分配默认值以避免不必要的锁存器。
    e   <=   0;
    g   <=   0;
    end
end
endmodule
```

在程序列表1-21中,最后一个else包含了一些注释,但并不是所有的输入状

态都包含在最后一个 else 条件语句之前。例如,若 and _ test 和 or _ test 都没有被声明,将会有什么样的输出呢? 如果没有定义最后一个 else,则 Verilog 会把已定义状态到未定义状态的变化视为状态保持(如果不要求输出,则最后一个数值将不变),这种情况下必须要创建锁存器。但通常这不是设计者所期望的情况,因此我们应该定义所有的状态。

1.7.4 相等运算符

= = 和 = = = 是逻辑运算符。 = = 称为逻辑相等运算符,通常其结果为真或假,但是若所比较的比特值是 x 或 z,则其结果为 x。 = = = 则称为 case 相等运算符,即对各个比特位的值进行比较,返回结果为真或假。在相等运算符前加上一个 ! ①表示不相等。在程序列表 1-22 中,有几个返回值为真的 if 语句。当程序从前到后执行时,只有第一个为真的状态被接受,其后的状态不再被考虑。我们称之为优先编码,与初始化锁存器一样,Verilog 倾向于使用这种结构,这将导致多级级联逻辑。要十分注意,你想要的选择可能是输入被并行检测的 MUX 类型结构。我们在以后将详细讨论相关内容。

程序列表 1-22 相等运算符实例

```
module eq _ test (clk, resetn, and _ test, or _ test);
input       clk, resetn, and _ test, or _ test;
reg         result;
reg[3:0]    b;
reg[3:0]    c;
reg[3:0]    d;
reg[3:0]    e;
reg[4:0]    g;
reg[3:0]    h, i, j;
always @ (posedge clk or negedge resetn)
begin
    if ( ~ resetn) // 异步复位(低有效)。
    begin

    result   <=   0; // 我们使用这个寄存器来映射等量的 result。
    b   <=   4'b1x00; // 这是一种不好的异步数值设置格式;这应该是
                      一个参数。
```

```
    c   <=   4'b1z00;
    d   <=   4'b1000;
    e   <=   4'b1001;
    g   <=   4'b01001;
    h   <=   4'b1z00;
    i   <=   4'b0110;
    j   <=   4'b011x;
    end

// 以下的测试失败了。
else if ((b==d)==1)
    result   <=   1'bx;

else if (b==d) // 此测试与前一行相同,失败了。
    result   <=   1'bx;

else if ((b==d)==0)//此测试通过了,因为 b 的值是 x。
    result   <=   1'bx;

else if ((b !  =d)==1) //此测试与前一行相同,失败了。
    result   <=   1'b0;

else if ((b==d)==1'bx) // 此测试通过了。因为 b 的值为 x。
    result   <=   1'b1; // 下列所有为真的条件都将被忽略。
else if (c===d) // 此测试失败。
    result   <=   1'b0;

else if (e==g)      // 此测试通过了,因为用 0 来填充 e,使其长度与
                    //g 相等。
    result   <=   1'b0; // 当变量长度不匹配时要注意。

else if (b==c)  // 此测试失败(返回 false)。
    result   <=   1'b0;

else if (b !  = c)  // 此测试通过(返回 true)。
```

```
        result   <=   1'b1;

    else if (d == e)   //此测试失败(返回 false)。
        result   <=   1'b0;

    else if (b !  == c)   //此测试通过(返回 true)。
        result   <=   1'b1;

    else if (c == h)   //此测试失败(返回 x)。
        result   <=   1'bx;

    else if (c === h)   //此测试通过(返回 true)。
        result   <=   1'b1;

    else if (e == ! i) //此测试通过(返回 true)。
        result   <=   1'b1;

    else if (e !  = j)
    //此测试失败(返回 x)。被取反的 x (未知)依然是未知。

        result   <=   1'bx;

    end
    endmodule
```

设计者可以选择如下的 if 语句格式:

```
if ( ~ resetn)…
if (resetn == 1'b0)
```

这两种格式的效果是相同的。哪一个是更容易阅读更容易理解的? 要因人而异。注意后缀"n"的使用,它表示一个低有效信号。有很多识别低有效信号的办法,例如,reset _ not,reset1,或 reset * ,resetN。

其他的操作如大于(>)、小于(<)、大于等于(>=)、小于等于(<=)也都被支持。

1.7.5 移位运算符

>> n 和 << n 是右移(相当于除以 2n)和左移(相当于乘以 2n)操作。移位操作用 0 来填充寄存器的高位或低位。包括 x 或 z 的移位操作会把所有的比特位都置为 x。程序列表 1-23 是使用移位操作的一些实例。

程序列表 1-23 移位操作实例

```
module shifter (clk, resetn, shift _ right _ test, shift _ left _ test);
input       clk, resetn;
input       shift _ right _ test;
input       shift _ left  test;
reg[3:0]a;
reg[3:0]b;
reg[3:0]c;
reg      d;
reg[3:0]e, f;

    always @ (posedge clk or negedge resetn)
    begin
        if ( ~ resetn) //异步复位(低有效)。
        begin
        a  <=  'b1001;
        b  <=  0;
        //这是一种不好的异步设置数值的格式。应该是一个参数。
        c  <=  0;
        d  <=  0;
        e  <=  'bx000;
        end
        else if (shift _ right _ test)
        begin

        c  <=  a >> 2;  // c 被赋值为 0010。
        d  <=  a >> 5;  // 无需考虑 a 的数值,d 总被赋值为 0。
        //虽然 Verilog 不会禁止这种做法,但使用时要谨慎。
```

```
        f   <=   e >> 1;   // 因为 e 中的 x 值,所以结果为 xxxx。
        end
        else if (shift _ left _ test)
        begin

        c   <=   a << 2;   // c 被赋值为 0100。
        d   <=   a << 5;   //无需考虑 a 的数值,d 总被赋值为 0。
        //虽然 Verilog 不会禁止这种做法,但使用时要谨慎。
        f   <=   e << 1;   //因为 e 中的 x 值,所以结果为 xxxx。
        end

        else
        begin
        d   <=   0;   // 赋予默认值以避免不必要的锁存器。
        e   <=   0;
        f   <=   0;
        end

    end
    endmodule
```

1.7.6 条件运算符

实现条件运算表达式的一个简单方法就是使用三段式格式。

```
output _ assignment <= expression ? true _ assignment : false _ assignment;
```

这是定义 MUX 的常用方法。如果该表达式的值为 x 或 z,则 output _ bus 将进行以比特为单位的计算,Verilog 将分析输出值。如果两个输入都是 1(即与输入条件无关),则输出为 1,两个输入都是 0 时情况也一样。任何不能被解出的比特值都将被置为 x。如果 true _ assignment 或 false _ assignment 的长度不足以填充 output _ assignment,则 output _ assignment 从左边开始用 0 填充。见程序列表 1-24。

程序列表 1-24 条件实例

```
module cond _ tst (clk, resetn, tristate _ control, input _ bus, output _ bus);
input              clk, resetn;
```

```
input          tristate_control;
input  [7:0] input_bus;
output         output_bus;
reg  [7:0] output_bus;

always @ (posedge clk or negedge resetn)
begin
    if ( ~resetn) //异步复位(低有效)。
    output_bus   <=   8'bz;

    else

// 若 tristate_control 为真则令 output_bus = input_bus ,
// 若 tristate_control 为假则令 output_bus 为高阻态。
  output_bus <= tristate_control ? input_bus:8'bz;
end
endmodule
```

1.7.7 数学运算符

Verilog支持一部分数学运算,包括加法(+)、减法(-)、乘法(*)、除法(/)、模运算(%)。但是综合工具可能限制了常数二次幂乘法和除法的使用(而左移位器或右移位器将被综合),并且可能不支持模运算。加法和减法运算将调用预先优化的加法器。Verilog认为所有的寄存器变量和线网型变量都是无符号的。

1.7.8 参数

使用参数是提高常量可读性的有效办法。参数只在其被定义的模块中使用,但可以被更高层次的模块改变。在运行时参数是不能被更改的,但在编译时可以被更改。在更改已定义的信号数目或更改使用某些结构的程序块的数目时,使用参数会很方便。并不是所有的参数都要求被赋值,但出现位置赋值列表时,参数不能被忽略。

一个参数也可以根据其他的常量或参数来定义。为了提高可读性,有些设计者用大写字符来标识参数。

程序列表1-25、程序列表1-26列出了Verilog的层次。其中,模块列表用分层表示时,分层从顶部开始,模块名用模块标识分开。

程序列表 1-25 参数实例,顶层模块

```
module top;
reg     clk, resetn;
parameter     byte_width   = 8;
defparam
        u1.reg_width         = 16;  // 此参数将替代在 reg_width 的 u1 实
                                    // 例中找到的第一个参数。
defparam
        u2.reg_width   =   byte_width * 2;
parm_tst u1 (clk, reset, output_bus);
  // 用 reg_width = 16 创建 parm_tst。
parm_tst u2 (clk, reset, output_bus);
  //此 parm_tst 的 reg_width 值也是 16。
parm_tst u3 (clk, reset, output_bus);
  // 此 parm_tst 的 reg_width 值为 8。
endmodule
```

程序列表 1-26 参数实例,低层模块

```
module          parm_tst (clk, resetn, output_bus);
input           clk, resetn;
parameter       reg_width   = 8;
// 此常量可以由一个传送到模块中的参数值替代。
parameter       byte_signal   = 8'd99;
parameter       byte_signal_true   = 8'hff;
parameter       byte_signal_false   = 8'h00;
output          [reg_width - 1:0] output_bus;
reg             [reg_width - 1:0] output_bus;
reg             [7:0] byte_count;

always @ (posedge clk or negedge resetn)
begin
    if (~resetn) //异步复位(低有效)。
    begin
```

```
        output_bus    <= 8'b0;
        byte_count    <= 8'b0;
        end
        else if (byte_count == byte_signal)
        output_bus    <=  byte  signal_true;
        else
        begin
        output_bus    <=  byte_signal_false;
        byte_count    <=  byte_count + 1;
        end
    end
    endmodule
```

1.7.9 级联

级联是信号或数值的组合,封闭在大括号{ }中,级联表达式用逗号(,)分离,如程序列表1-27所示。所有的级联数值都必须规定长度。注意使用中括号[ ]以确认比特选取或寄存器标识。定义类似backwards_reg的寄存器是合法的,而且,无需考虑所使用的数,最左边总是定义了最重要的比特。除非创建了一维数组(如RAM),否则最大数通常出现在冒号(:)的左边。

程序列表1-27 级联实例

```
module backward;
reg [0:2] backwards_reg;
reg [2:0] test;
/* {1'b0, test, 8'h55} 与
{1'b0, test[2], test[1], test[0], 1'b0, 1'b1, 1'b0, 1'b1, 1'b0, 1'b1, 1'
  b0, 1'b1}相同 。 */
always @ (test)
begin
    test   =   backwards_reg;
    // 上面的赋值与下面的赋值是等价的:
    test[2] = backwards_reg[0];
```

```
        test[1] = backwards_reg[1];
        test[0] = backwards_reg[2];
    end
    endmodule
```

第二章　数字设计的策略与技巧

从设计入门的观点来看，数字电路设计者应尽可能抽象地来描述一个设计。如果写下一行代码就能实现25 000个门电路设计的话(也许将来有一天250万个门电路是典型设计)，我们将认为这种抽象在做设计时是一种高效率的方法。我们使用先进的软件工具，可以将抽象的顶层设计转化为代表硬件和硬件连接的网表。

在用FPGA硬件实现前，顶层设计要分为多步来完成。在本章中，其中的每一步我们都将做详细讨论。

2.1　设计步骤

- 对设计进行语法分析。
- 按照目标结构对设计进行最小化和最优化处理。
- 用选定的库模块或核心模块代替已验证过的结构元素。
- 时序和资源需求的评估。
- 将设计转换为网表。
- 连接设计元素和模块，用库模块或内核模块网表来代替“黑匣子”模块。
- 平面规划并进行布线尝试，直至时序和资源的约束条件被满足。
- 从设计中提取时序和资源报告，创建一个时序标注网表以支持后布线仿真。
- 创建器件配置文件。

2.2　数字原语模拟模块的建立

同一个设计可以有很多思路，设计者要善于在不同的设计方法中挑选，直至将格式化的比特流文件配置到FPGA中。这能帮助我们牢记住数字设计单元都是由模拟部件实现的。没有什么神奇的元件可以直接完成与非门的功能，我们必须使用晶体管、二极管、电阻等模拟元件来实现数字逻辑功能。晶体管可以完成数字开关(开或关)或模拟转换门(通过模式)的功能。对于N型场效应晶体管(N-FET)来说，当栅为“1”时，为开启状态；但对于P型场效应晶体管(P-FET)来说，当栅为“0”时，为开启状态。见图2-1～图2-3所示。

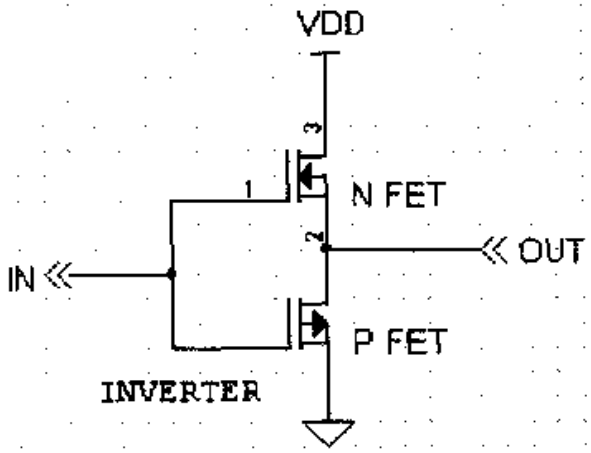

图 2-1 分立逻辑电路:简化的反相器

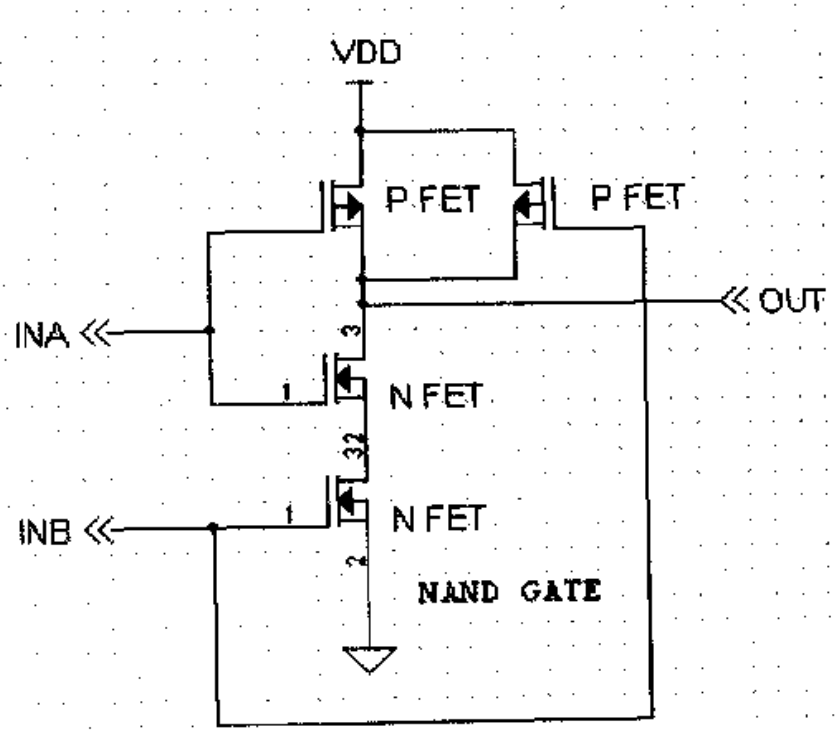

图 2-2 分立逻辑电路:简化的与非门

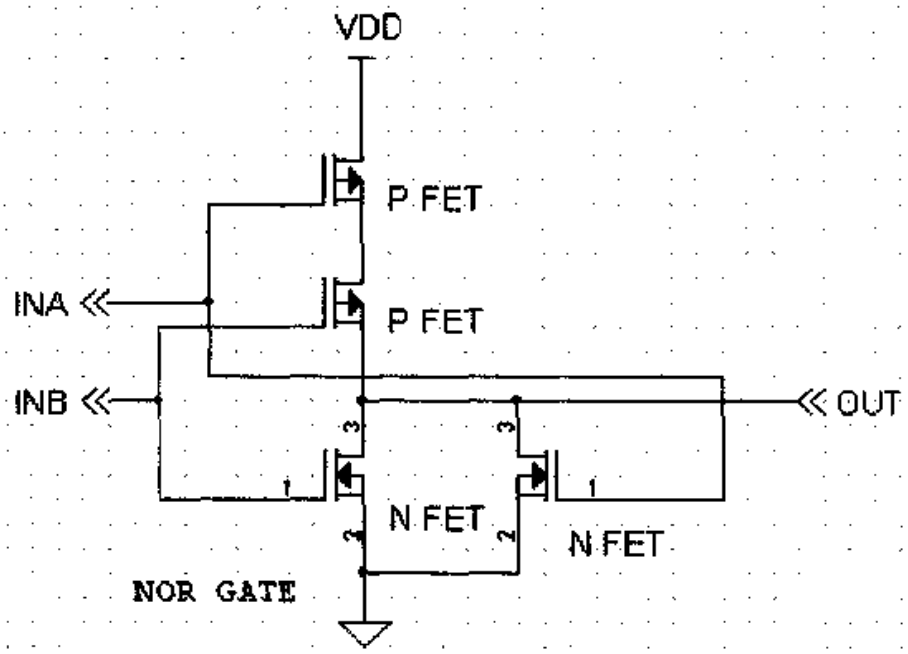

图 2-3　分立逻辑电路:简化的或非门

2.3　使用 LUT 来实现逻辑功能

大多数 FPGA 将多路转换器(MUX)查找表(LUT)作为基本的逻辑单元。这样做的原因有两点:

(1) LUT 是通用的(输入的任何功能都可实现)。

(2) LUT 能够在硅片上高效实现。

MUX 控制输入被用作逻辑输入,多路输入也可接入到逻辑层以实现所需的功能。

图 2-4 是一个用此方法实现的反相器。

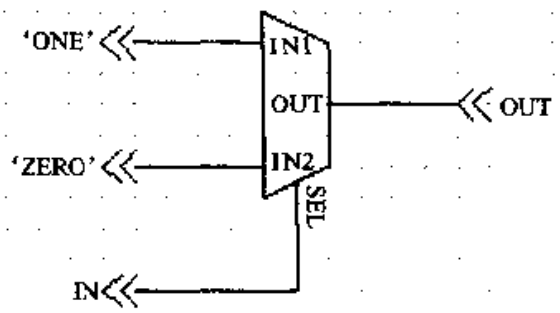

图 2-4　配置 MUX 而成的反相器

将 MUX LUT 作为逻辑单元还有一个内在的好处，这是由具有电容性负载和 MUX 输出的“先断后开”的开关特性提供的，即当输入改变时，输出信号先予保持，然后再平滑地改变，并且不产生任何脉冲干扰。

2.3.1 综合实例

再次关注第一章中的热探测器。

程序列表 2-1 过热探测器源代码

```
module  overheat (clock, reset, overheat _ in, pushbutton _ in, overheat _ out);
input   clock, reset, overheat _ in, pushbutton _ in;
output  overheat _ out;
reg     overheat _ out;
reg     pushbutton _ sync1, pushbutton _ sync2;
reg     overheat _ in _ sync1, overheat _ in _ sync2;

// 通常，同步输入的相位与系统时钟无关。
// 对外部信号使用双同步触发器以减少亚稳态问题。
// 对上升/下降时间较长和不稳定的外部信号进行某种滤波和锁存将能
// 使其有所改善。
always @ (posedge clock or posedge reset)
begin
    if (reset)
      begin
      pushbutton _ sync1   <=   1'b0;
      pushbutton _ sync2   <=   1'b0;
      overheat _ in _ sync1   <=   1'b0;
      overheat _ in _ sync2   <=   1'b0;
      end
    else begin
      pushbutton _ sync1   <=   pushbutton _ in;
      pushbutton _ sync2   <=   pushbutton _ sync1;
      overheat _ in _ sync1   <=   overheat _ in;
      overheat _ in _ sync2   <=   overheat _ in _ sync1;
      end
```

```
end

// 当确定过热时,按下按键,对过热输出信号进行锁存。
always @ (posedge clock or posedge reset)
begin
    if (reset)
        overheat_out    <=    1'b0;

// Overheat_out 被永久保存(直至复位)。
    else if (overheat_in_sync2 && pushbutton_sync2)
        overheat_out    <=    1'b1;
end

endmodule
```

综合工具将把这个简单程序的源代码转换为繁杂的、有 300 多行的 EDIF 网表。此网表保存了所有的设计元素和编译器的版本信息、目标和所有的设计约束。设计这种网表是为了兼容其他计算机上的程序,对设计者意义不大,所以不再举出相关实例。

图 2-5 所示网表的图形版本(电路图)对于我们更有价值,它可以帮助我们弄清综合工具到底创建了什么,尤其是当 HDL 功能不强时,更是如此。注意要正确利用时钟和复位的全局资源,因为 Verilog 不支持硬件资源的直接赋值(这是

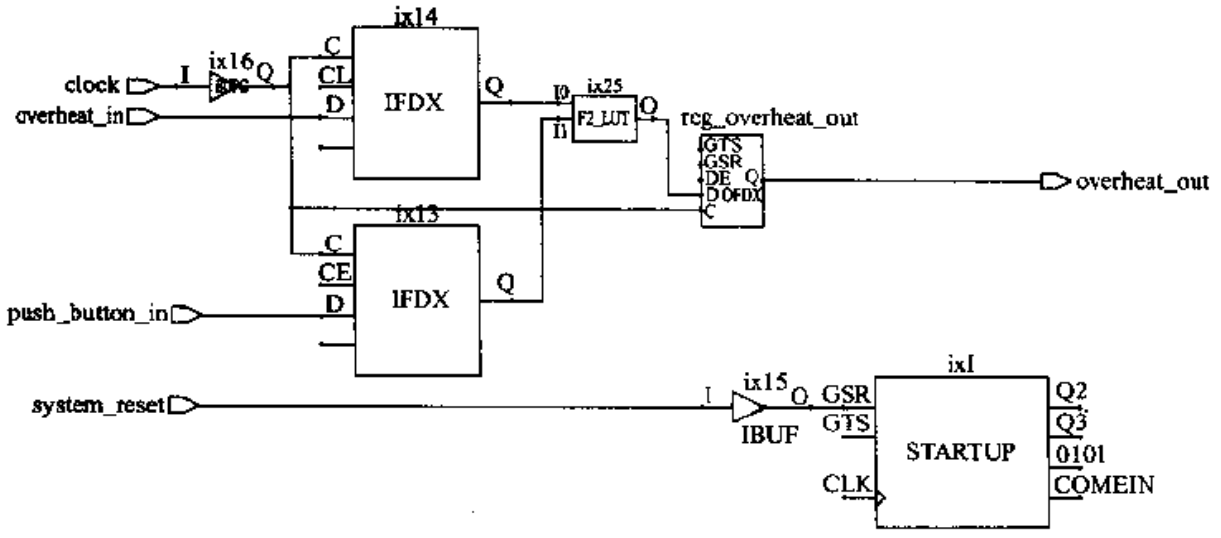

图 2-5　EDIF 网表示意图

Verilog FPGA设计者面临的最大问题)。

因为综合工具对目标结构有所了解,所以它可对时序设计和资源需求进行估算,见程序列表 2-2。这种估算不包括"黑匣子"模块,而该模块其后会由 FPGA 的布局和布线工具添加。

程序列表 2-2 综合器资源估计

```
*******************************************************

Cell: overheat     View: INTERFACE     Library: work

*******************************************************

 Number of ports :                          5
 Number of nets :                          13
 Number of instances :                     10
 Number of references to this view :        0

Total accumulated area :
 Number of BUFG :                           1
 Number of CLB Flip Flops :                 2
 Number of FG Function Generators :         1
 Number of IBUF :                           1
 Number of IOB Input Flip Flops :           2
 Number of IOB Output Flip Flops :          1
 Number of Packed CLBs :                    1
 Number of STARTUP :                        1

*************************************************
Device Utilization for 4010xlPQ100
*************************************************
Resource                  Used    Avail    Utilization
IOs                       5       77         6.49%
FG Function Generators    1       800        0.13%
H Function Generators     0       400        0.00%
CLB Flip Flops            2       800        0.25%
```

```
                    Clock Frequency Report

   Clock                 : Frequency
   ------------------------------------
   clock                 : 118.8 MHz
```

2.4 关于设计步骤

2.4.1 语法检验

第一步是将代码提交给编译器和仿真器,以及与/或 Lint 程序以识别语法错误、键盘输入错误和其他错误。每个程序检验代码的方式都是不同的。如果对语

法检查有疑问，则可以尝试其他检查程序。程序列表 2-3 ~ 程序列表 2-6 显示了插入到 overheat.v 中的一段程序代码，当发生同一个错误时的四种不同的报告方式。一个多余的分号添加在如下的 if 语句中：

```
if (overheat _ in _ sync & pushbutton _ sync);
```

程序列表 2-3　LeonardoSpectrum 综合工具的错误报告

```
35  always @ (posedge clock or posedge reset)
36      begin
37      if (reset)
38          overheat_out    <=  1'b0;
39
40  // Overheat_out is held forever (or until reset).
41      else if (overheat_in_sync2 && pushbutton_sync2);
42          overheat_out    <=  1'b1;
43      end
```

LeonadoSpectrum 标记“C:/Verilog /SourceCode /overheat.v”，第 37 行有错误，在异步过程中有一个以上的 if 语句不被支持。

程序列表 2-4　ModelSim 提供的错误报告

```
vlog C:/verilog/overheat.v
# – Compiling module overheat
# ERROR: C:/verilog/overheat.v(28): near "else": expecting: END
# ERROR: C:/verilog/overheat.v(29): near "end": expecting: ENDMODULE
```

ModelSim 在第 38 行报告错误，即在出错行的邻行进行报错。至少是在所出错误的右方进行报错。

程序列表 2-5　Silos Ⅲ 提供的错误报告

```
Reading "c:\verilog\verilog\overheat.v"
sim to 0
    Highest level modules (that have been auto-instantiated):
        (overheat overheat

c:\verilog\verilog\overheat.v (38) : error 3.229 : expecting
```

```
"end", or statement, not integer constant
error 2.188 : errors are too severe to simulate
```

程序列表 2-6 Verilint 提供的错误报告

```
Processing source file c:\verilog\verilog\overheat.v
  (E363)  c:\verilog\verilog\overheat.v, line  37: Syntax error:
  1 syntax error
  End of interHDL inc. Verilint (R) Version 3.14, 1 errors, 0
  warnings
```

列举这些实例的目的是说明不同的工具会显示不同的出错信息，所以采用几种工具尤其是 Lint 工具来检测你的代码是值得提倡的做法。Verilint（或其他的 Verilog Lint 工具）是很有用的检验工具，它运行速度快、容易操作、能检测出各种错误，并能节省很多时间。此外，这几个实例还提醒我们，一个放错位置的分号都会给我们带来很多麻烦。

2.4.2 设计的最小化和最优化

设计工作的最终结果是对硬件产生一个相应的配置。这些硬件可以是 FPGA、半定制 FPGA 或某些类型的 ASIC（标准单元、门阵列、全定制）。如果是一个 FPGA，

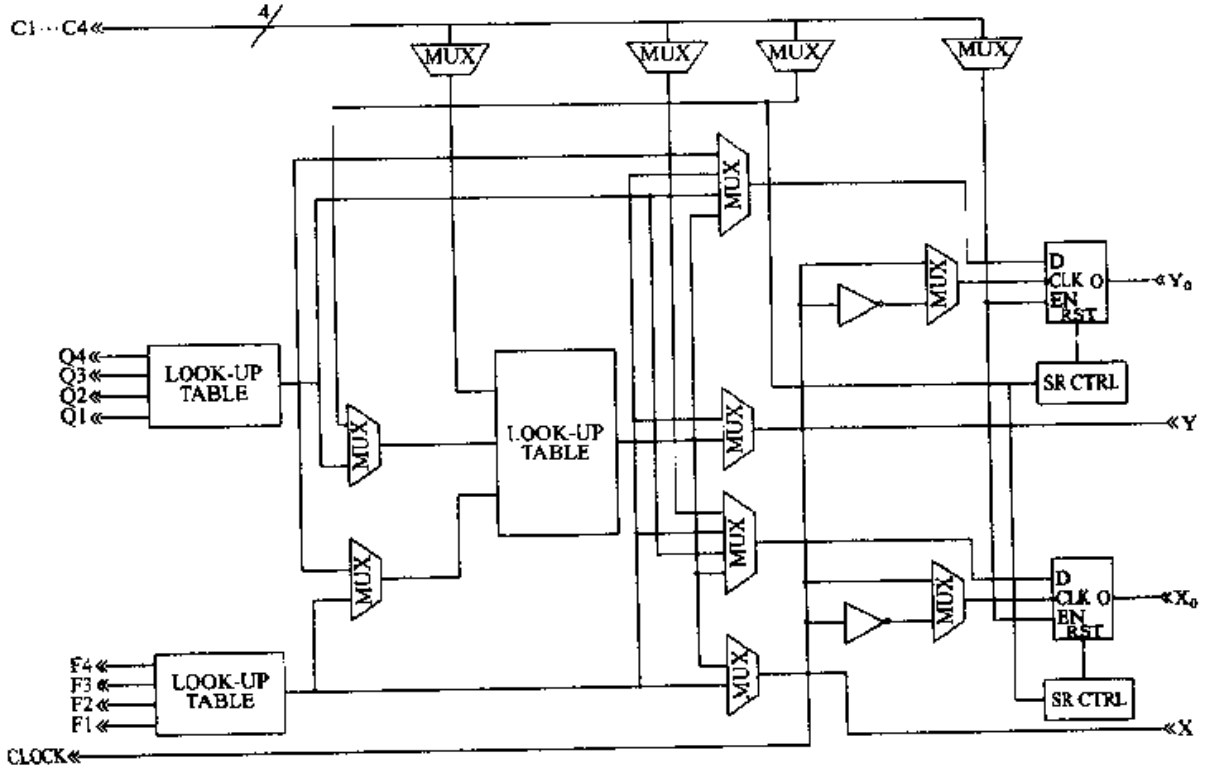

图 2-6 典型 Xilinx CLB 结构

则该硬件的底层结构会因 FPGA 设备制造商提供的设计方法的不同而不同。Xilinx 4K 系列的逻辑结构见图 2-6。

Xilinx 4K 系列可配置的逻辑模块(CLB)主要由两个四输入的 LUT 接到一对触发器为基础的。利用综合工具,我们所写的 Verilog 代码会映射到此结构中。

综合器使用以下的方法将设计转换成适合目标硬件的形式:

• 将设计转换为大布尔等式,一个等式对应一个模块输出、一个设计段的输出或一个寄存器输出。冗余寄存器可以被识别并被优化掉。例如,程序列表 2-7 的代码就可以转换成为程序列表 2-8 中的代码。

程序列表 2-7　简单加法器代码

```
// 简单加法器(不带输入)。
module adder (clock, reset, a, b, c);
input     clock, reset, a, b;
reg[1:0] c;
always @ (posedge clock or posedge reset)
   begin if (reset)
      c   =   2'b0;
   else
      c   =   a + b;   // 加法器。
   end
endmodule
```

为了表示此电路中的门电路,可以建立一个真值表(见表 2-1)来定义所有的输出和输入状态。

表 2-1　简单加法器真值表

a	b	c[1] - 进位	c[0] - 和
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

通过观察,我们发现 c[0](和)的输出可由异或门表示,c[1]的输出(进位)可由与门表示。转换后的加法器电路如图 2-7 所示。

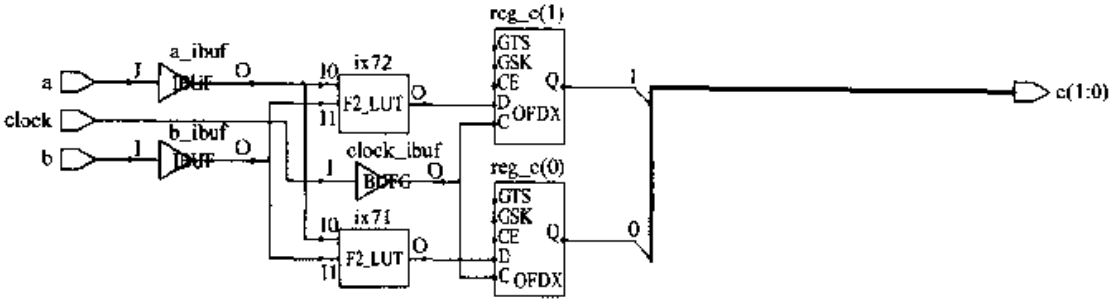

图 2-7 简单加法器电路示意图

程序列表 2-8 简单加法器布尔等式

```
c[0]  <=  a ^ b ; // ^是 Verilog 的布尔逻辑 XOR 运算符。
c[1]  <=  a & b ; // & 是 Verilog 的布尔逻辑 AND 运算符。
```

图中上面的 F2 _ LUT 用以创建 c[1](ix51) 的是(! I0 * I1) + (I0 * ! I1),相当于异或的功能。下面的 F2 _ LUT 用以创建 c[0](ix50)的是 (I0 * I1)。

• 布尔等式最小化,是通过确认并去掉冗余的逻辑项来实现的(即使是控制扇出或给出危害范围的情况,我们也希望保留它们)。

综合器不能识别跨越寄存器边界的冗余逻辑(虽然它能识别并去掉冗余的寄存器)。如果存在使逻辑最小化的方法,那么这种最小化的结果应当作为设计输入的一部分。这种逻辑项的删减工作最好由设计者自己来做。

• 已识别出的结构单元被选好的模块代替。例如,综合器把 a <= a - 1 作为递减计数器,并用面积最小、速度最快的、最适合目标硬件结构的优化电路来代替它。

• 估算时序和资源需求。编译器能估算时序和资源需求,因为设备制造商可能会改变时序参数(设备制造商总是领先于其他厂商,因为其他厂商都依靠制造商的数据)。时序估算可能不够准确的另一个原因是因为库文件和"黑匣子"单元还没有被写到设计网表中,而这两部分是在创建最终的网表并在转换后才插入的。

• 将设计转换为网表。有多种形式不同的网表,但目前最常用的格式是 EDIF。

• 链接各设计单元和模块,"黑匣子"模块由库模块网表代替。编译器创建的网表可以是平面型的(即将所有的模块都合并在一个网表中),也可以保留分层,即各个模块都保持分离状态。保留分层能使设计者更易于理解。

• 布局规划,并进行布线尝试直至时序和资源的约束条件被满足。画平面图,即是将各单元从器件逻辑中提取到电路图中。设计的布局和布线类似于印刷

电路板的布局和布线。布线的效率和速度取决于模块单元的排列，这些排列将影响模块间的互连。在 FPGA 中布线资源是有限的。当布线变得密集时，完成一个信号路径就需要较长的布线。这样一来，在紧凑的地方就容易产生布线问题。一些 FPGA 制造商宣称他们具有 100% 的布线能力，实际上其布线能力只能达到 65%（Altera）和 85%（Xilinx）。手工进行布局规划能增加可用逻辑电路密度。

• 从设计中提取时序和资源报告。创建一个具有时序信息的网表，可以支持布线后的仿真。这种网表通常是 SDF 格式，见程序列表 2-9。SDF 代表标准延迟格式。这个文件包括根据 FPGA 设计规则估算出的门延时。

程序列表 2-9　SDF 网表举例

```
(DELAYFILE
 (SDFVERSION "2.0")
 (DESIGN "adder")
 (DATE "08/31/99 09:21:34")
 (VENDOR "Exemplar Logic, Inc., Alameda")
 (PROGRAM "LeonardoSpectrum Level 3")
 (VERSION "v1999.1d")
 (DIVIDER /)
 (VOLTAGE)
 (PROCESS)
 (TEMPERATURE)
 (TIMESCALE 1 ns)
(CELL
 (CELLTYPE "F2_LUT")
 (INSTANCE ix72)
 (DELAY
  (ABSOLUTE
   (PORT I0 (::3.25) (::3.25))
   (PORT I1 (::3.25) (::3.25)))))
(CELL
 (CELLTYPE "F2_LUT")
 (INSTANCE ix71)
 (DELAY
  (ABSOLUTE
   (PORT I0 (::3.25) (::3.25))
   (PORT I1 (::3.25) (::3.25)))))
(CELL
 (CELLTYPE "BUFG")
 (INSTANCE clock_ibuf)
 (DELAY
  (ABSOLUTE
   (PORT I (::0.00) (::0.00)))))
(CELL
 (CELLTYPE "OFDX")
 (INSTANCE reg_c_1)
 (DELAY
  (ABSOLUTE
   (PORT C (::3.25) (::3.25))
   (PORT D (::2.77) (::2.77))))
  (CELL
```

```
 (CELLTYPE "OFDX")
 (INSTANCE reg_c_0)
 (DELAY
  (ABSOLUTE
   (PORT C (::3.25) (::3.25))
   (PORT D (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "IBUF")
 (INSTANCE reset_ibuf)
 (DELAY
  (ABSOLUTE
   (PORT I (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "IBUF")
 (INSTANCE a_ibuf)
 (DELAY
  (ABSOLUTE
   (PORT I (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "IBUF")
 (INSTANCE b_ibuf)
 (DELAY
  (ABSOLUTE
   (PORT I (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "STARTUP")
 (INSTANCE ix56)
 (DELAY
  (ABSOLUTE
   (PORT GSR (::2.77) (::2.77)))))
)
```

• 创建器件配置文件。下载文件可被编程写入串行 EPROM 中，通过串行或并行电缆下载；通过微处理器或 FPGA 产生的具有地址和数据控制的独立 EPROM 存入存储器，写入到器件。这个器件可能是 ISP 型或可重复编程型的（接入到编程器中，编程后，再安装到目标设计中）。

2.4.3 不稳定的逻辑电路

许多人在画一个双输入的或非门时，都会画出如图 2-8 所示的电路。根据作者的经验，这个电路是不稳定的。在输入改变时，其输出很可能出现脉冲干扰。我们是数字电路设计师，所以我们希望设计中的模拟部分越少越好。

图 2-8 双输入组合或非门

图 2-9 画出了一个有可能使用简单或门的典型电路。图中的电阻和电容不一定是电路板上的分立元件，它们也可能是由于信号布线和负载所产生的寄生值。

图 2-10 是示波器显示的图像，它显示了一个组合电路的波形。当一个输入固

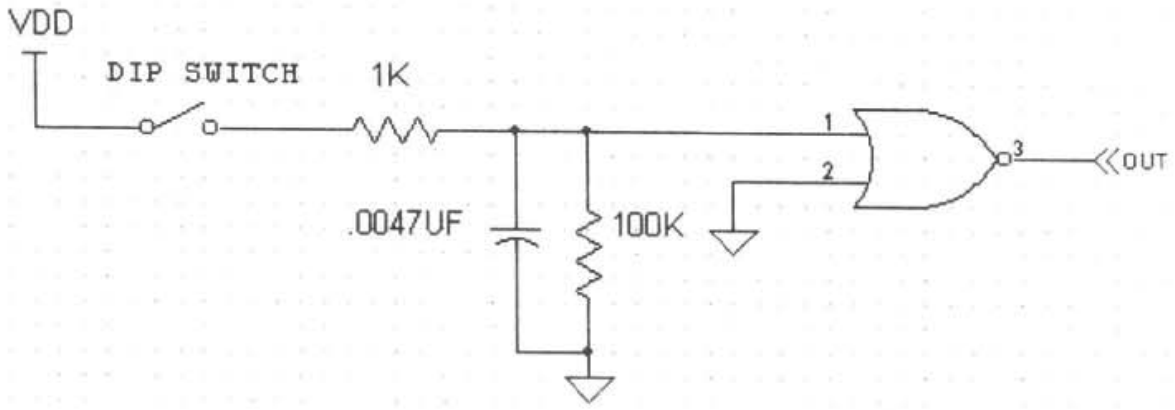

图 2-9　简单组合电路

定为低电平时，输出应该是另一个输入的反相，这正确吗？输出的那些令人讨厌的脉冲干扰来自何处呢？这是因为输入的噪声信号慢慢地超过输入阈值(即信号可被识别为 1 或 0 的范围)。而且 RC 网络放大了这个信号。对切换所引起的跳变进

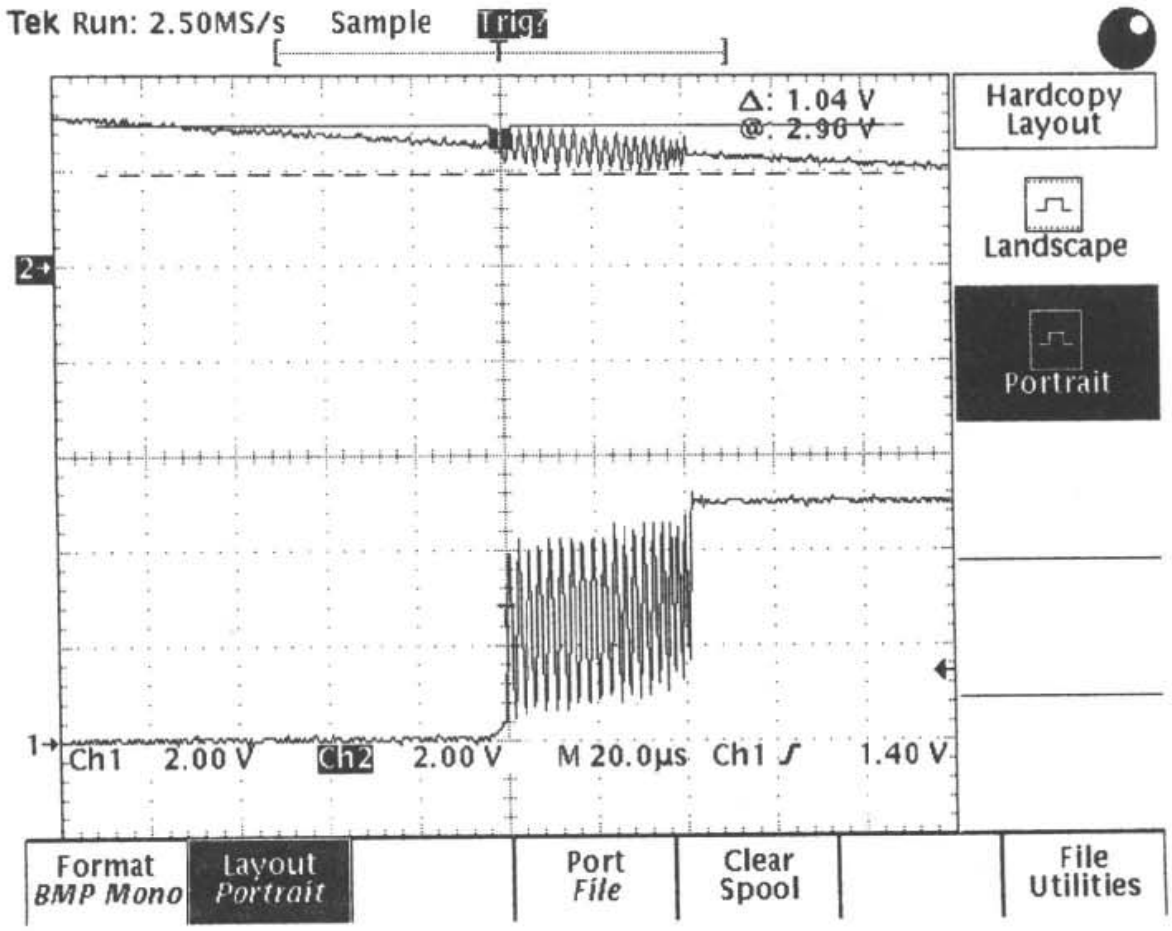

图 2-10　组合双输入或非门的输出瞬态波形

行滤波的正确方法是使用反馈(滞后现象)。

如果输入总是变化得很快,就能减少干扰脉冲。但是,即使能设计一个无限快速切换的电路,还是消除不了其他因素产生的干扰。例如当两个输入几乎同时改变时,输出是不确定的。图 2-11 说明了这个问题。加入一个 RC 网络以延迟输入信号,输出还是有讨厌的瞬态干扰现象。所以你的设计不能在输入处使用 RC 网络。RC 时延可以由输入之间的混合布线(造成信号偏移)产生,也可以由信号加载产生,这个时候,每个信号最终都会产生电容性负载。图 2-11 中 R 部分代表了信号源阻抗和布线阻抗(与线路长度成正比)之和,C 代表了网络负载(与网络中的负载数目成正比)。控制这个问题的惟一办法就是让信号具有低扇出系数。大多数综合工具都允许设置扇出限制来控制负载(信号是分离的,由不同的缓存器驱动)。

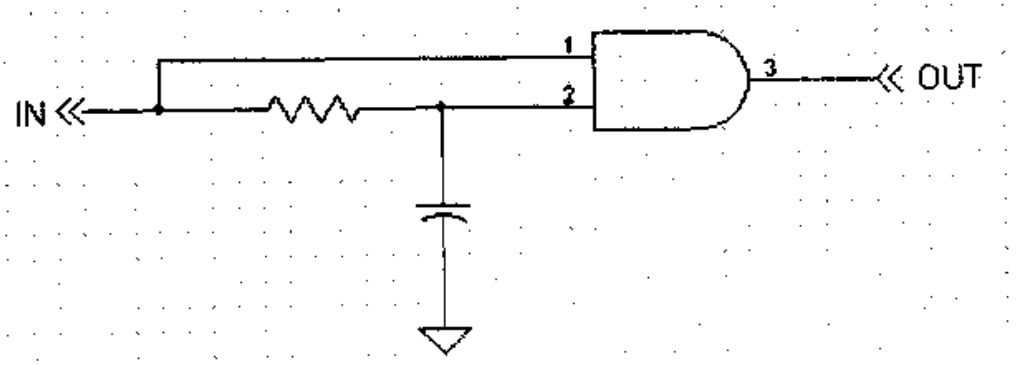

图 2-11 组合双输入带有 RC 网络的与门

如果有人要我设计一个双输入与门,我会画出如图 2-12 所示的电路,与图2-11的不同之处是添加了一个同步触发器。如果遵循同步逻辑规则,满足触发器的建立/保持时间的要求,那么此电路的输出就不会再有脉冲干扰。在输入同步信号时,这种做法尤其安全可靠。如果触发器 D 输入端的信号在所要求的建立时间内

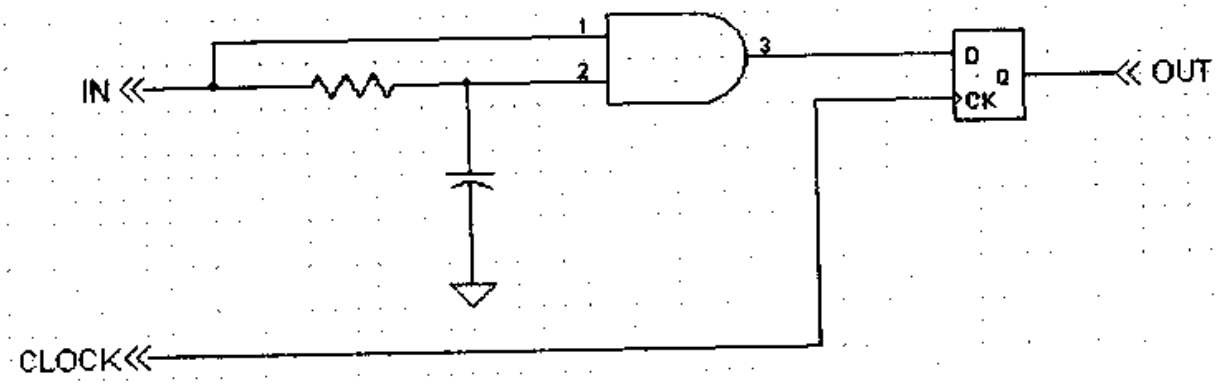

图 2-12 同步与门

是稳定的，并且其维持时间超出所要求的保持时间，那么一切都将是顺利的。

2.5　同步逻辑规则

2.5.1　亚稳态

从字面来看，亚稳态是不稳定的意思。如果信号是亚稳态的，则它是不稳定的，它介于 1 和 0 之间，或是经过振荡最终变为 1 或 0。作为一个数字电路设计师，希望你能仔细考虑到这一点。

当一个时钟沿因异步输入信号的变化而呈随机状态时，亚稳态就会发生。如果时钟和信号的关系确实是随机的，那么输入信号就会变得与时钟沿相距太近，从而不可避免地导致输出状态的不确定。解决的办法是将触发器的时延加长，使其长度比触发器数据表中列出的典型的 clock-to-Q(从出现时钟信号到出现输出信号)输出时延长得多。

图 2-13 显示了亚稳态的问题，如果信号在触发器的建立/保持时间范围内改变，则输出在一段时间内是未知的。这段时间有多长呢？这要取决于触发器的特性和周围的环境：触发器的速度、增益，系统噪声。这个问题有多严重？这取决于输入变化的频率和建立/保持相对于时钟周期的宽度。

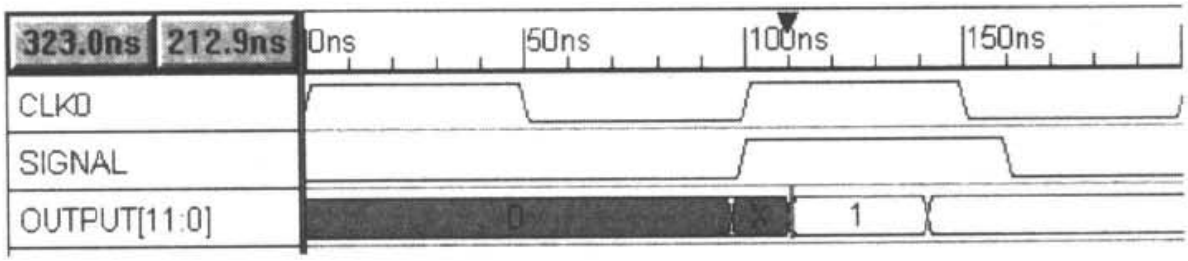

图 2-13　亚稳态输出

我们不可能消除亚稳态，但是我们希望能尽量避免亚稳态。

解决亚稳态问题的捷径是采用同步设计技术。这意味着用同步时钟来限定、选通或触发电路。在两个时钟沿之间的时间内允许信号传输和停留。这就像个游戏，如果你能在下一个时钟启动时间之前得到下一个触发器信号，那你就成功了。

2.5.2　建立和保持时间

因为触发器的输出是可以预测的(非亚稳态)，所以输入必须满足建立和保持的时间的要求。

- 建立时间通常用 T_{su} 来表示，是同步时钟沿之前的一段时间；在这个时间内要求输入信号保持稳定。如果建立时间被扰乱，那么输出信号就是不确定的。

• 保持时间通常用 T_h 来表示，是同步时钟沿之后的一段时间；在这个时间内要求输入信号保持稳定。如果保持时间被扰乱，那么输出信号的值就不能确定。

建立和保持的要求源于触发器设计中的模拟特性。触发器使用交叉耦合门电路来实现反馈以保持状态，而门电路也需要时间来达到稳定状态。在忽略这一切的理想情况下，边缘触发器的状态改变完全与时钟沿同步。时钟沿的出现是瞬时的，触发器也将瞬时地改变状态。但在实际中，时钟具有上升/下降的时延，触发器要求在建立/保持时间内有稳定的输入信号以确保输出的稳定。

只要信号和触发器的时钟之间有随机的相位关系，触发器的亚稳态问题就永远不会消失。但是眼下 IC 制造商在减小亚稳态时限上有了很大的进步。通过提高触发器速度，亚稳态时限会变到更短。实际上，设计者面临的大多数关于亚稳态问题都是在使用异步设计技术时遇到的。每个 FPGA 输入必须驱动一个且仅一个触发器，而单个触发器的输出可用来驱动另一个触发器以提高安全性或用来驱动异步系统的其他部分。当一个异步输入驱动多个触发器，并且这个输入在时钟沿附近变化时，一些触发器输出将会改变而其他的触发器输出不会改变。这不是亚稳态问题，而是异步输入的问题。

这个问题如图 2-14 所示。RC 延时代表了因 FPGA 的布线和负载而产生的信号延时。我们希望所有的触发器输出信号都是相同的，但是，因为输入信号的相位不同，输出信号常常是不相同的。如果我们用一个触发器来同步输入信号且不侵

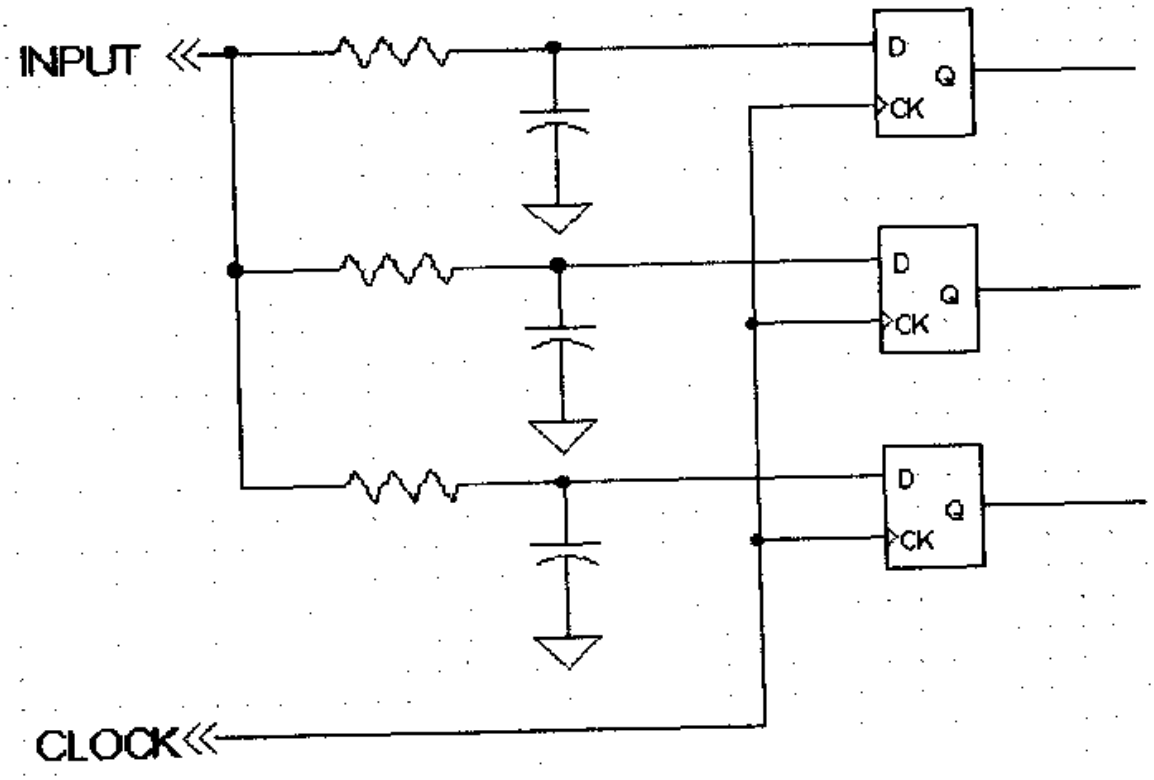

图 2-14 异步输入实例

犯其建立/保持时间，那么所有的输出就一定相同。

如何保证输入信号在触发器的建立和保持时间内不发生改变？如何实现一个没有问题的设计？

保持输入信号的同步！这意味着一个异步输入只驱动一个触发器。然后触发器的输出就可以安全地驱动同步电路的其他部分。

图 2-15 是一个具有同步输入信号的同步与门。

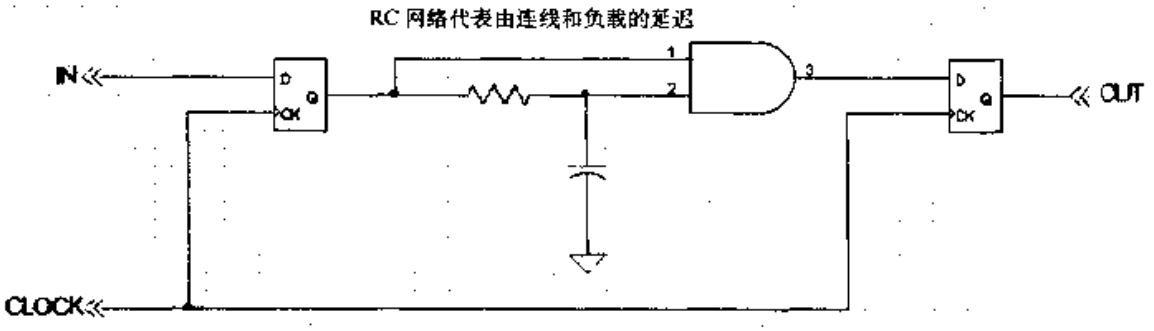

图 2-15　采用同步输入的同步与门

同步触发器的输出看上去好像在时钟跃变时发生变化，这对输出触发器而言不是问题吗？让我们先来考虑一个普通的电路。如果要求你设计一个被 2 除的电路，你可能就会做出如图 2-16 所示的设计。

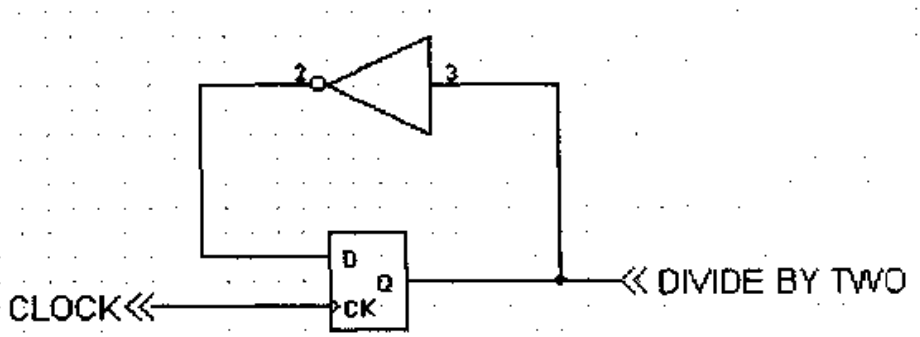

图 2-16　具有被 2 整除功能的电路

这个电路几乎不能再简单了，输出反相后反馈给 D 输入端，输出在每一个时钟沿处改变状态。需要考虑的是这个电路什么时候不工作。假设元件的技术指标都是一些简单的数字，例如设所有的延时都是 1ns。

触发器性能指标：

最小输入建立时间：　1ns

最小输入保持时间： 1ns

clock-to-Q(从出现时钟信号到出现输出信号的延迟) 最大值： 1ns

最大传输延迟时间(从 Q 输出端到 D 输入端)： 1ns

对于可重复的结果,D 输入必须在时钟上升沿到达前 1ns 到时钟上升沿到达后 1ns 的时间内都保持稳定。触发器的输出保证在时钟沿之后少于 1ns 的时间内达到终值。信号从 Q 输出端再返回到 D 输入端所用的时间起码要少于 1ns。那么这个电路不适用于什么样的时钟频率呢?

时钟上升沿的到来必须在 3ns 或一段时间之后(这段时间即是建立时间、输出延迟、路径延迟的总和)。这意味着输入的时钟频率不能大于 333.333MHz。只有使用当今先进的 ASIC 技术才可能达到这样高的频率。由于 FPGA 芯片有更长的延时(可能是长得多),所以其最大时钟频率也相应较低。

元器件的延迟是设备供应商提供的。FPGA 有多种延迟,包括 clock-to-Q 延迟、通过开关单元的布线延迟、通过信号转换器(即查找表)的延迟、与信号负载成正比的延迟等。例如 4000XL 型器件,Xilinx 就规定了 4 种速度级别的 41 个时序参数。存储这些参数,然后再加以测试。幸运的是,编译器知道这些已知的延迟,并能为电路设计计算出总的延迟时间。

再来看看串联的两个触发器,如图 2-17。

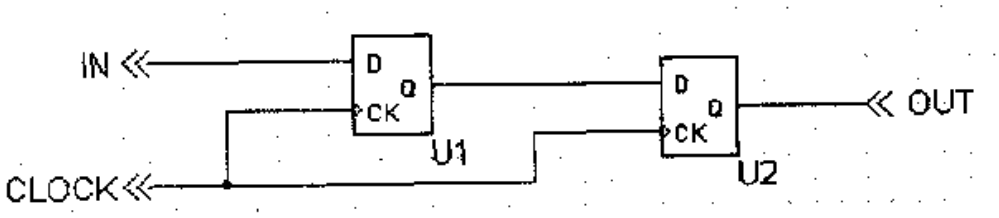

图 2-17 两个串联的触发器

这同样是个再简单不过的电路。这个电路是怎样可靠工作的呢? 如果 U1 的最小 clock-to-output 延迟(这个值很少做出规定,但通常都估算为 25%)小于 U2 的保持时间会有什么样的结果? 数据手册中规定保持时间为零,实际上所有的逻辑电路的保持时间都不可能是零。FPGA 逻辑单元可以延迟一段时间以保证逻辑线路延时小于时钟的线路延时。本质上,这种做法是在建立时间上再添加一个保持时间,然后延迟时钟以满足这个延长的建立时间。关于时钟沿,输入信号要用较长的时间到达 D 输入端并且保持一段较长的时间。即使输入信号的改变与时钟沿一致,逻辑单元内部的时钟延时也要足够长才能满足触发器保持时间的要求。这样就简化了 FPGA 设计的分析,并保证如图 2-18 的电路能够正常工作。

总之,FPGA 芯片的设计者创建了一个逻辑单元以保证如图 2-17 的电路可以

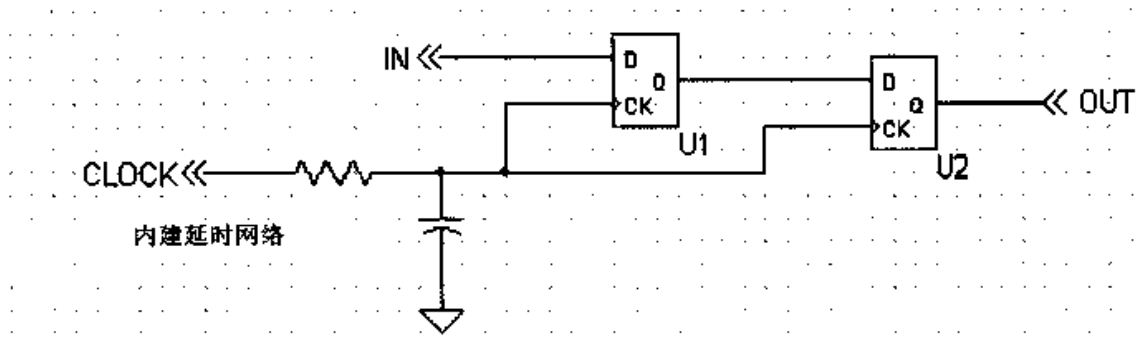

图 2-18　具有内部延迟的两个串联触发器

正常工作。ASIC 设计者就不具备这样的条件，他们必须计算设计中的延迟和容差。但在处理来自 FPGA 外部的信号时，我们就没有以上的便利条件了。必须检测外部信号的信号特征，如果出现信号减慢或干扰的迹象，那么我们就要用滞后电路（如 Schmitt 触发器）并使用双触发器同步电路以减少亚稳态。

滞后是指一个将正反馈加入到输入端的电路。这样，当输出变化时，加到输入端的反馈将有助于阻止振荡。反馈量应比输入端的噪声信号稍大。虽然 Xilinx 并未广泛宣传这个信息，但他们所有的 FPGA 输入端都有几百毫伏的滞后。这使得他们的输入更适应于噪声环境。

为了完成我们的分析，必须要考虑时钟偏移。在理想情况下，所有的触发器都接受完全同步的时钟信号。第一件必须明白的事是时钟偏移问题与运行频率无关。一个速度很慢的设计也可能出现时钟偏移的问题。

图 2-19 中的电路是将图 2-17 中的电路扩展而成的。假设设计时有意将两个触发器隔的远一些，第二个触发器的时钟相对于第一个触发器有延迟。

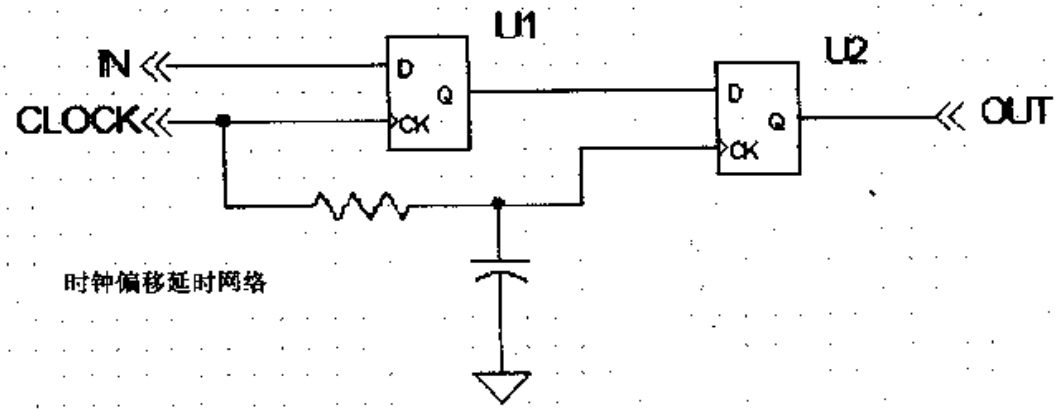

图 2-19　具有时钟偏移的两个串联触发器

用 t1 表示 clock-to-output 延时，t2 表示信号穿过器件到第二个触发器 D 输入端的传输延时。我们希望 U2 存储的值是 U1 的 Q 输出端的值。如果时钟偏移太多，则我们可能得到新的 U1 的 Q 值；也有可能 U2 的建立时间被扰乱，从 U2 得到一个未知的值。而我们不希望有未知的信号，解决的办法就是在 FPGA 设计中使用一个低时偏的时钟网络以保证最大的时钟偏移也不会超过最小的 clock-to-Q 和信号路径传输时间。如果你能使用低时偏的全局时钟网线，那么一切都会正常。如果你使用已选择路径的时钟如门控时钟、多路转换器时钟等来创建异步设计或一个 ASIC(时钟网线按常规设计)设计，那么你一定要保证满足上述条件。

2.5.3 处理外部信号

在 FPGA 设计中存在一个或多个信号不能被控制的情况，这是值得注意的问题。图 2-17 中触发器 U1 作为一个输入，它由 FPGA 提供触发器的时钟信号。如果 U1 是高速设备，则竞争现象就有可能出现。当信号在不同时间到达同步触发器时就会出现竞争，其输出结果是未知的，这是应该尽量避免的现象。

如果在输入端没有输入同步触发器的存在，信号竞争现象就会更加严重。因为在这种情况下，竞争现象会传播到整个设计的所有电路，这就非常糟糕。如果有输入同步触发器的话，该触发器就有了建立/保持时间的要求；那么一旦此设备开始同步操作，信号就能有较好的状态。这种情况下，最简单的解决方法就是保持外部元器件的时钟与内部的逻辑同步时钟是完全相同的；外部元器件是低速设备，其输出变化不是太快，所以不会引起竞争现象。另外，如何证明这种做法有效也是个问题，因为芯片制造商不提供最小的 clock-to-Q 输出时间。制造商不提供这个数据的原因是改进 IC 时他们不需要更换数据表。但是对于设计者而言，在分析同步时间时是很不方便的。

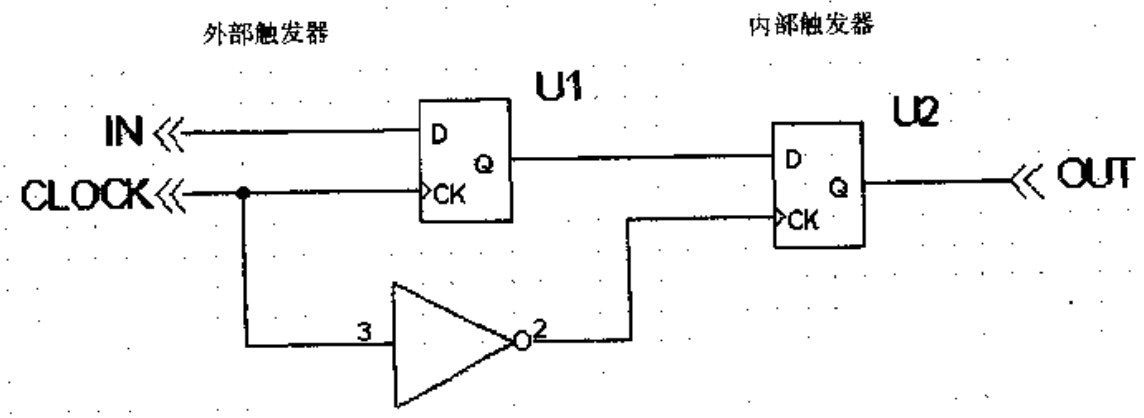

图 2-20 使用交替时钟沿的两个串联触发器

2.5.4　使用交替的时钟沿

进行外部器件时钟设定时，要使其时钟沿与 FPGA 内部的时钟沿相反。Xilinx 允许触发器在时钟的上升沿或下降沿进行触发。时钟偏移和建立时间必须小于 1/2个时钟周期。使用交替时钟沿的电路示意图如图 2-20，其同步波形如图 2-21。

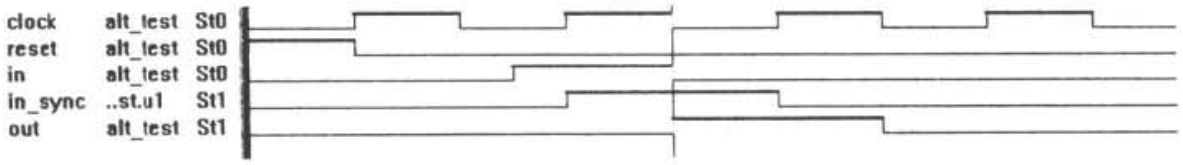

图 2-21　使用交替时钟沿的两个串联触发器的同步波形

2.6　时 钟 策 略

FPGA 设计者做出的最重要的决定

在一个设计中所做的最重要的决定就是采用何种时钟策略。必须仔细考虑这个问题。在时钟问题上的错误可能会毁掉整个设计。

我们已经决定了要创建同步设计，这意味着设计中至少存在一个时钟(可能只有一个)。对于没有瑕疵的设计，是使用一个主时钟。但是如果设计里必须有不同的时钟区域，应该怎么办呢？如果使用单个时钟导致了过度的能耗，又该怎么办呢？对于这类问题没有固定的答案，而是取决于设计者的想法和做法。以下是一些如何使用时钟的建议：

(1) 在设计 ASIC 时，如果能耗不是问题，那么最好在所有的触发器中使用一个主时钟，并且用时钟使能来代替低频时钟，它能够检测逻辑状态，并且时钟使能可以在低频时工作。整个电路只有一个时钟，因此有最简单的时序分析。对于这种设计，最易于做自动分析和测试，ATPG(自动测试程序)工作得最好。

(2) 在设计 ASIC 时，如果功耗是需要考虑的问题(使用电池组或电能损耗较大时)，则应在某些触发器中采用低频时钟。电路的功耗与时钟的频率以及在此频率上开关的门电路的数目成正比。

(3) 在设计 ASIC 时，将 FPGA 作为样机，这时最好使用一个主时钟，并且要求 FPGA 是高速运行的，类似于 ASIC(Quicklogic、Gatefield 或其他具有反熔丝的一次性编程器件)。

(4) FPGA 为样机，但使用的是基于 SRAM 的元器件(如 Xilinx 或 Altera)，则应该在所有的触发器中使用一个主时钟。但是所有的模块都要在可能的最低速度下运行，这样才能保证整个设计能正常工作。用多个时钟驱动触发器时，应对这个设

计进行划分，在中心时钟产生器模块中产生每个时钟，并将各时钟区域的交叉部分最小化，并确保跨越时钟区域的信号都是正确同步的。

(5) 设计快速 FPGA(可能成为 FPGA-to-ASIC 转换)时，最好的办法是充分利用全局时钟，将各时钟区域的交叉部分最小化，确保跨越时钟区域的信号都是正确同步的。

(6) 在做低速的 FPGA 设计时，上述任何一种方法都可以使用。

2.6.1 时钟使能

Verilog 硬件编程语言并不支持专用的时钟使能信号。有些硬件(如 FPGA 或 ASIC)具有专用的时钟使能资源，但 Verilog 并不直接控制这个信号的分配。但是，综合工具可通过编译器指令来支持这个功能。程序列表 2-9 中的代码能否用不同的方式来进行综合取决于目标硬件是否支持时钟使能的指令。如图 2-22 所示，若在 FPGA 逻辑模块设计中时钟是可以使能的，则该设计可由综合器解释。

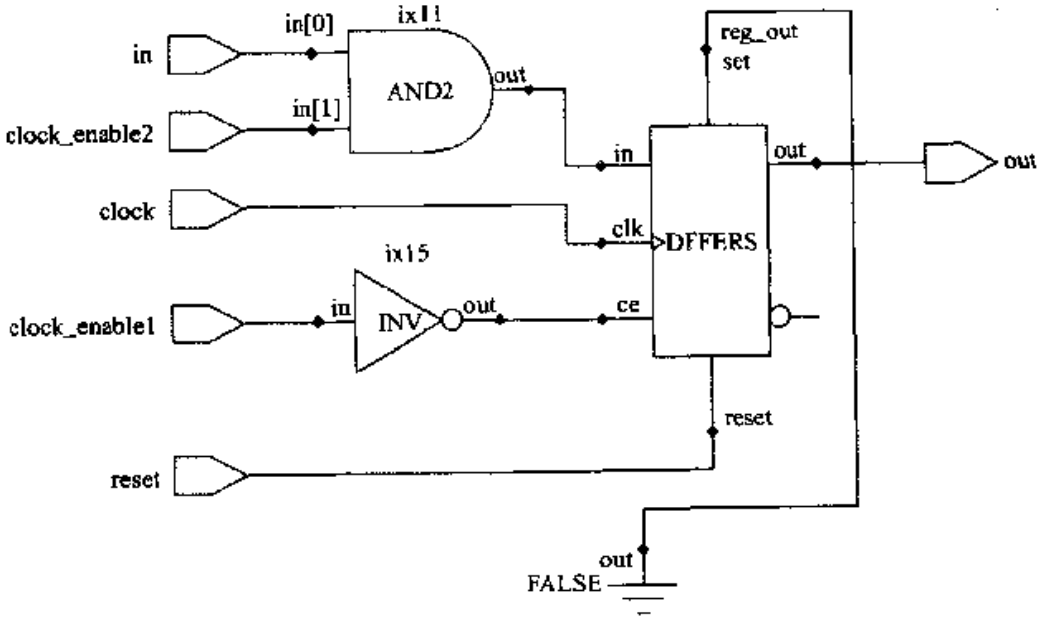

图 2-22 可综合的时钟使能

程序列表 2-10 时钟使能实例

```
module clock_en(out, in, clock, clock_enable1, clock_enable2, reset);

output  out;
input   in, clock, clock_enable1, clock_enable2, reset;
```

```
reg      out;

always @ (posedge clock or posedge reset)
   begin if (reset)
      out   <=   0;
   else if (clock _ enable1)
      out   <=   out; // 如果没有使能,则输出被保持。
   else
      out   <=   (in & clock _ enable2);
   end
endmodule
```

有些逻辑电路可包括在时钟使能的驱动电路中,如图 2-23 所示。此图与图 2-22的不同之处是:综合器可在时钟使能的路径上插入逻辑电路。

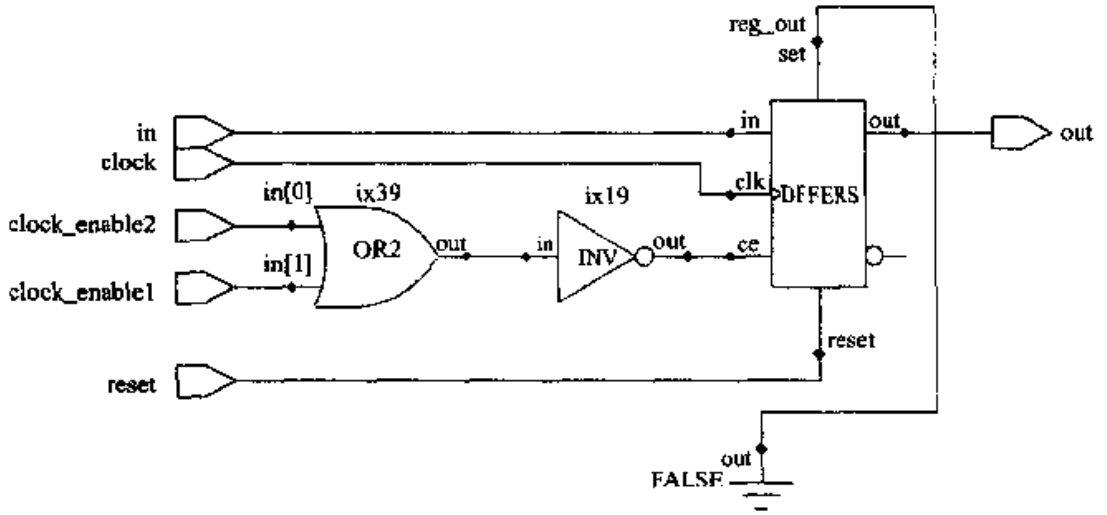

图 2-23　可综合的时钟使能(混合型)

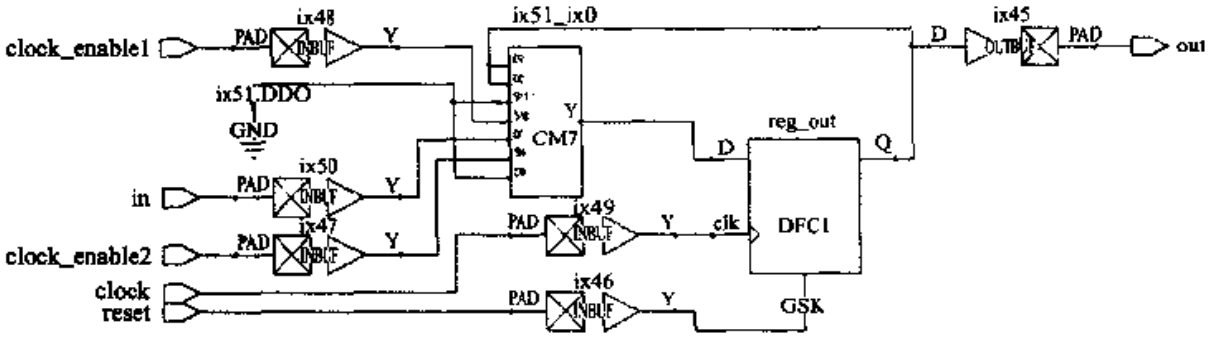

图 2-24　可综合的时钟使能(布线型)

下一个例子如图 2-24 所示，是用不具有时钟使能性质的硬件来实现时钟使能。在 clock _ enable 不确定时，用组合反馈来保持输出，用这种方式可以实现时钟使能。

2.7 逻辑化简

综合器能识别并去掉冗余逻辑。例如，程序列表 2-11 和 2-12 中的代码段是等效的。

程序列表 2-11 冗余逻辑实例 1 代码段

```
input  test1, test2, test3 ;
output sample ;

sample = (( test1 & test2 & test3 ) | ( test1 & ! test2 & test3)
| ( test1 & test2 & ! test3 ));
```

程序列表 2-12 冗余逻辑实例 2 代码段

```
input    test1, test2, test3 ;
output   sample ;
sample   =  ( test1 & ( test2 | test3 ));
```

即使是设计者有意添加冗余逻辑来消除设计隐患，这些冗余逻辑依然会被最简化。这种消除是附加在冗余逻辑之上的，本书不建议使用这种方法，而建议使用同步设计技巧以减少设计隐患。

编译器也能识别等效的逻辑等式。在 FPGA 中运行时，面积较小、层次较少的逻辑等式会被选择。编译器会变换等式形式以便等式能更好地满足设计需求。

德摩根定律

```
~(a & b)   =   (~a | ~b) ;
~(a | b)   =   (~a & ~b);
```

关于与/或的德摩根推论也可以应用到只有异或的形式。

```
(a^b)   =   ~a^~b;
~(a^b) = ~a^b = a^~b;
```

与/或运算互为对偶运算(如除法和乘法互为对偶运算)。德摩根定律定义了

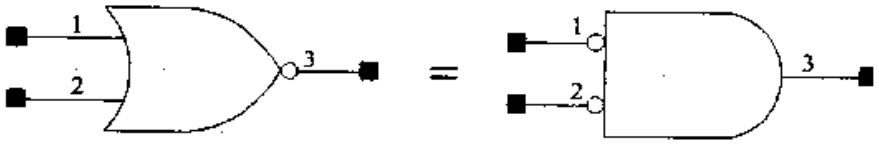

图 2-25　德摩根定律的电路示意图

与/或等式形式的转换。

编译器还可以使用布尔代数的定律来生成等式。这些定律包括：

交换律：a | b = b | a；

结合律：a | (b | c) = (a | b) | c；

分配律：a & (b | c) = (a & b) | (a & c)。

因为设计者使用同步技术并且不使用寄存器之间的复杂结构来设计，所以综合工具提取冗余逻辑项的能力会受到限制。可能有更简单的逻辑项，但如果该逻辑项跨越了寄存器的边界，综合器就无法提取它。如图 2-26 的电路，它是采用同步技术实现程序列表 2-10 中的逻辑电路。这时的综合器将找不出冗余项，除了一些传输延迟，图 2-26 中的两个电路是等效的。

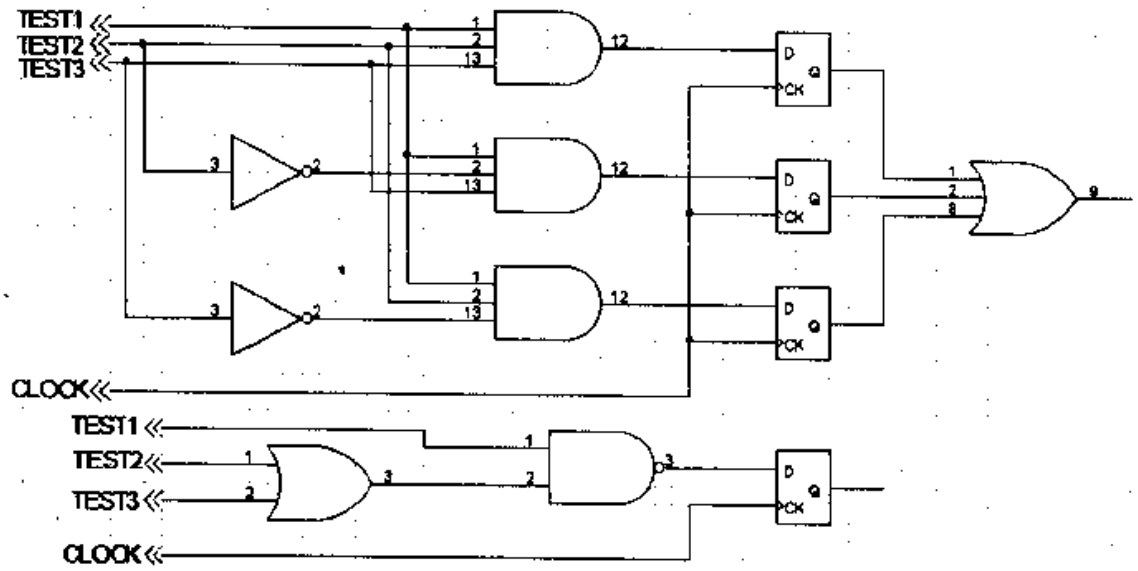

图 2-26　跨越寄存器边界的冗余逻辑项

最好的逻辑综合器就是人的大脑。无论使用多好的编译器，糟糕的设计始终都是糟糕的。在做设计时，水平高的设计者会始终在大脑中保持着综合逻辑的模型，因而不会使逻辑太复杂以至于影响到综合。利用综合工具的能力来简化和高

效压缩逻辑的一个办法就是永远不要创建单纯的组合模块。目前流行的任何FPGA都没有单纯的组合逻辑元素。如果 CLB 只用于组合逻辑里，那么触发器就是一种浪费。把组合逻辑和同步逻辑混合在一切，综合工具才能把它们合并到可用的设备中。逻辑块的结构都使用组合逻辑或 LUT(查找表)寄存器驱动。

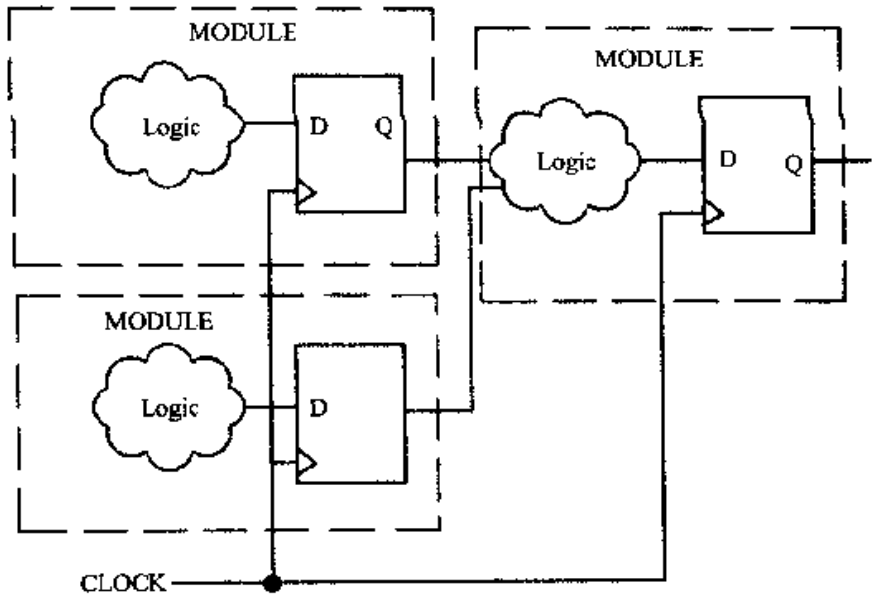

图 2-27 组合逻辑驱动触发器示意图

2.8 综合器做些什么

综合工具将 Verilog HDL 转化到目标硬件中。首先是通过去掉多余的逻辑项以简化逻辑等式，然后使整个设计成为一大组布尔等式。剩下的部分可以用最简单的除法描述：A/B。A 是整体设计，B 代表了目标 CPLD、FPGA 或 ASIC 等可用的硬件单元。通常，CPLD 的硬件结构是多输入逻辑单元(LE)，FPGA 的硬件结构是 3 输入或 4 输入的查找表(LUT)，而 ASIC 的硬件结构则是库单元的自由组合。假设基本逻辑单元是四输入的 LUT，则综合工具就会把复杂的分母分割为多个等式，每个等式都具有四个输入。有很多种等式的组合都能实现设计的功能，综合器会寻找适合设计目标和速度的组合。

真值表列出了所有的输入组合并定义了每种组合的输出状态。编译器从 HDL 代码中提取乘积和(SOP)等式。建立 SOP 等式的方法是查找结果为 1 的各项并对它们做或运算。

下面是一个七段解码器的 SOP 表达式。这个解码器类似于 CMOS 4513，可以将 4 位二进制编码对应的十进制数(BCD)转化到七段显示驱动器件的引脚中。

真值表如下：

输入				显示段						
BCD				a	b	c	d	e	f	g
b3	b2	b1	b0							
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	0	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

把产生“a”段的各输入项集中如下：

```
a = (( ! b3 & ! b2 & ! b1 & ! b0) | (! b3 & ! b2 & b1 & ! b0)
    | ( ! b3 & ! b2 & b1 & b0) | ( ! b3 & b2 & ! b1 & b0)
    | ( ! b3 & b2 & b1 & ! b0) | ( ! b3 & b2 & b1 & b0)
    | ( b3 & ! b2 & ! b1 & ! b0) | ( b3 & ! b2 & ! b1 & b0);
```

我们可以看出化简算法是如何提取和检验等式中的两个逻辑项：

(! b3 & ! b2 & ! b1 & ! b0) | (b3 & ! b2 & ! b1 & ! b0) = (! b2 & ! b1 & ! b0);

以上两项只在 b3 处不同，一个为低一个为高，所以 b3 是多余项，去掉它不会有任何影响。

下面我们将“a”段等式转换为标准的十进制和的形式，即用 0 来代替所有的负项，用 1 代替所有的正项。例如(! b3 & ! b2 & ! b1 & ! b0)将变为(0,0,0,0)，整项都用 0 代替。下一项(! b3 & ! b2 & b1 & ! b0) ,(0,0,1,0)就是 2，依此类推直到计算完“a”的所有逻辑项并确定“a”的值：

a= (0,2,3,5,6,7,8,9)

这种形式的布尔代数式在 Quine-McCluskey 算法中使用。Quine-McCluskey 的 DOS 版免费软件包括在本书附带的光盘中，名称是 QM.exe。

Quine-McCluskey 算法按照每个逻辑项所含的 0 的数目来排列各项。逻辑项之间只有在其负项的数目相差 1 以上才能结合。例如，(0,0,0,0)和(1,0,0,0)是可以结合的，因为第一项有四个 0，而第二项有三个 0。Quine-McCluskey 算法采用穷

举和结合逻辑项的方式来简化逻辑表达式。

用 QM 简化后的"a"段表达式如下：

a = ((! b3 & ! b2 & ! b0) |(b3 & ! b2 & ! b1) |(! b3 & b2 & ! b0) | (! b3 & b1));

根据这个表达式建立的 Verilog 设计见程序列表 2-13。

程序列表 2-13 七段显示解码器"a"段的 Verilog 设计

```
module seven_seg (clk, reset, bcd_input, a_segment);

input          clk, reset;
input    [3:0] bcd_input;
output         a_segment;
reg            a_segment;

always @ (posedge clk or posedge reset)
    if (reset)
        a_segment  <=  0;
    else
    begin case (bcd_input)
    {1'b0, 1'b0, 1'b0, 1'b0}:a_segment <= 1'b1;
    {1'b0, 1'b0, 1'b0, 1'b1}:a_segment <= 1'b0;
    {1'b0, 1'b0, 1'b1, 1'b0}:a_segment <= 1'b1;
    {1'b0, 1'b0, 1'b1, 1'b1}:a_segment <= 1'b1;
    {1'b0, 1'b1, 1'b0, 1'b0}:a_segment <= 1'b0;
    {1'b0, 1'b1, 1'b0, 1'b1}:a_segment <= 1'b1;
    {1'b0, 1'b1, 1'b1, 1'b0}:a_segment <= 1'b1;
    {1'b0, 1'b1, 1'b1, 1'b1}:a_segment <= 1'b1;
    {1'b1, 1'b0, 1'b0, 1'b0}:a_segment <= 1'b1;
    {1'b1, 1'b0, 1'b0, 1'b1}:a_segment <= 1'b1;
    default:a_segment <= 0;
    endcase
    end
endmodule
```

Xilinx 4xxx 逻辑电路将四输入的 LUT 作为原语并输出到触发器。综合器有效

地组织逻辑电路以便高效地利用 CLB 资源。综合器给出的"a"段的逻辑电路如图 2-28 所示。

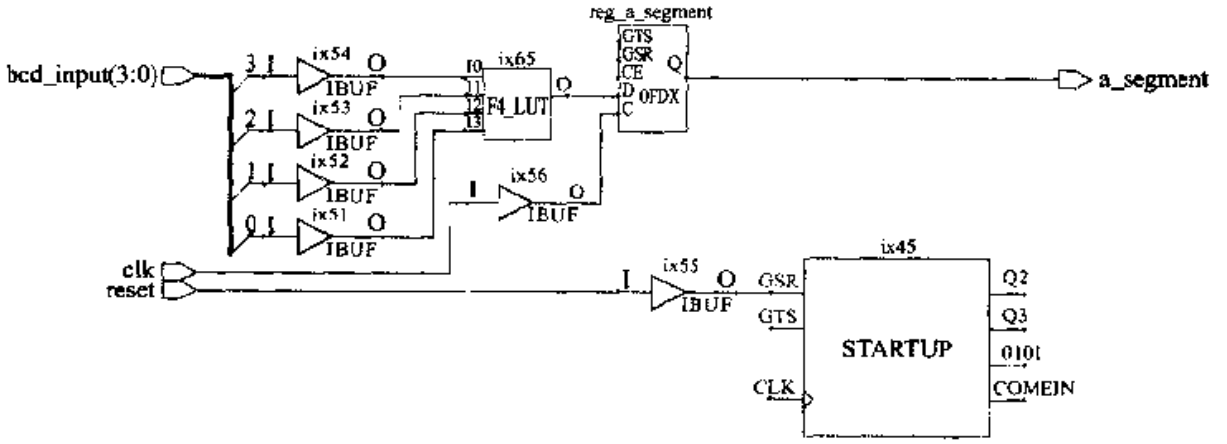

图 2-28　七段显示解码器"a"段的综合逻辑电路

2.9　面积/延时优化

在实现设计时，要综合考虑两个基本因素：设计的电路所占的空间和电路运行的速度。综合一个逻辑电路设计类似于在电路板上自动布线。在电路板上的布线有多种选择，信号会从哪条路径通过？具有优先权的信号与其他信号有何区别？这些问题都没有答案。信号的路径可以有很多条。需要强调的是，当一条路线符合设计需求时，它就是最合适的路线，即使可能还有在面积/延时方面性能更好的路线。设计者的工作并不需要完美无缺，而是要根据产品成本、设计成本、性能、可靠性、可维护性、推向市场所需时间等综合的需求来衡量。对设计的要求是"足够好"，而不是完美无缺。当然这并不意味着要做糟糕的设计。

设计成本的概念与面积和速度是互相影响的。通常最快的设计并不是最紧凑的，最紧凑的设计通常也不是最快的。如果设计者在工艺和速度选择上都能很好地满足需求，那么这个设计就是成功的。这个问题可能很简单，也可能很困难，这取决于多个因素：选定器件的尺寸、器件工艺的结构、所要求的系统运行速度、设计者的技巧和设计方法等。

有经验的设计者在做设计时总是留有余地，以便可以采用更快或更紧凑的器件来代替现有的某些器件。这样一来，就无需花费大量的成本和时间来重新设计电路板以容纳新的器件，更快或更密集的器件就可以很容易地进行替换。

第三章　数字电路工具箱

本章介绍了 Verilog 中的基本数字设计概念。

3.1　Verilog 层次回顾

Verilog 有一个强大的隔离和保持标识符的方法。没有被其他模块调用的模块被认为是顶层模块。顶层模块将调用其他各层的模块。顶层模块也被称为是根模块。设计标识符包括模块实例、任务、函数或 begin/end 程序块命名。

每个设计标识符都将在层次树上创建一个新的分支。树上的每个节点是不同的，包含不同的标识符，不会与其他分支上的标识符或其他层中的元素混淆。通过分离层次单元对信号进行层次描述，信号可以进入到设计的任何地方。程序列表 3-1 是一些文件名实例。

程序列表 3-1　层次命名实例

```
top.device_bus1[5:0]
top.device_bus2[3:0]
top.device_bus3[3:0]
top.s1[3:0]
top.s2[4:0]
top.s3[3:0]
top.s4[4:0]
top.s5[5:0]
top.msb1
top.msb2

top.u1.in1   // u1 由顶部模块调用。

top.u2.in1   // u2 由顶部模块调用。

top.u3.add1[4:0]
```

3.2　三态信号和总线

大多数的 FPGA 结构在器件输出端允许有三态总线。程序列表 3-2 提供了一个范例。此外，一些 FPGA 内部还允许有三态信号。在不同的控制信号和数据信号之间进行选择时使用内部三态可以节省大量的逻辑电路。换言之，不同的逻辑树每次用一个分枝上的逻辑电路来驱动一个三态总线。采用这种方法，整个逻辑结构可以快速切换。如果不允许内部三态，综合工具就可能进行控制，自动替代 MUX。这种替代需要很多的门电路，并且运行速度也比三态总线结构慢得多。对于 ASIC 而言，内部三态是个关键问题。商家可能并不提供内部三态，可能提供或不提供三态网线自动扩展为逻辑控制网线的功能。内部三态在进行仿真时也可能会出现问题。

程序列表 3-2　三态总线实例

```
module tristate (input _ bus, output _ bus, tri _ control);
input    [7:0]    input _ bus;
input             tri _ control;  // 三态控制信号。
output   [7:0]    output _ bus;

// 第一个状态是 tri _ control 为真的状态，第二个是 tri _ control 为假的状
   态。

assign output _ bus =  tri _ control ? input _ bus : 8'bz;
endmodule
```

在赋值语句的条件部分，input bus 和 tri control 可以是逻辑等式。对于内部三态，应使用程序列表 3-3 中的结点类型 tri。

程序列表 3-3　内部三态实例

```
module tristat2 (input _ bus1, input _ bus2, input _ bus3, input _ bus4, tri _ control,
output _ bus, output _ control);

input    [7:0]  input _ bus1, input _ bus2, input _ bus3, input _ bus4;
input    [1:0]  tri _ control;
```

```
input          output_control;
tri     [7:0]  tri_bus;
output  [7:0]  output_bus;

    parameter zero    =  2'b00;
    parameter one     =  2'b01;
    parameter two     =  2'b10;
    parameter three   =  2'b11;

assign tri_bus = (tri_control == zero) ? input_bus1 : 8'bz;
assign tri_bus = (tri_control == one) ? input_bus2 : 8'bz;
assign tri_bus = (tri_control == two) ? input_bus3 : 8'bz;
assign tri_bus = (tri_control == three)? input_bus4 : 8'bz;
assign output_bus =  output_control ? tri_bus : 8'b0;
endmodule
```

设计者有必要保证三态缓冲区使能相异，这样总线冲突才不会发生。即使是瞬间的三态总线冲突也会引起过量的功耗，在冲突时间过长或冲突次数过多时会使设备过热而损坏。

图 3-1 中的电路示意图有三级缓冲器。左边的是输入引脚缓冲器，中间的是内部三态缓冲器，右边的是输出引脚缓冲器。

图 3-3 的逻辑功能与图 3-1 相同，但内部三态被转换为 MUX。图 3-1 中的一级三态缓冲器转换为图 3-3 中的两级 MUX LUT。这就导致了运行速度的减缓。在设计被转换为 ASIC 时就需要进行这种变换。在用 LeonardoSpectrum 进行优化时，需要选择“Allow converting of internal tristates”选项。如图 3-2 所示。

图 3-3 里中部的方框是代替内部三态缓冲器的 MUX LUT。

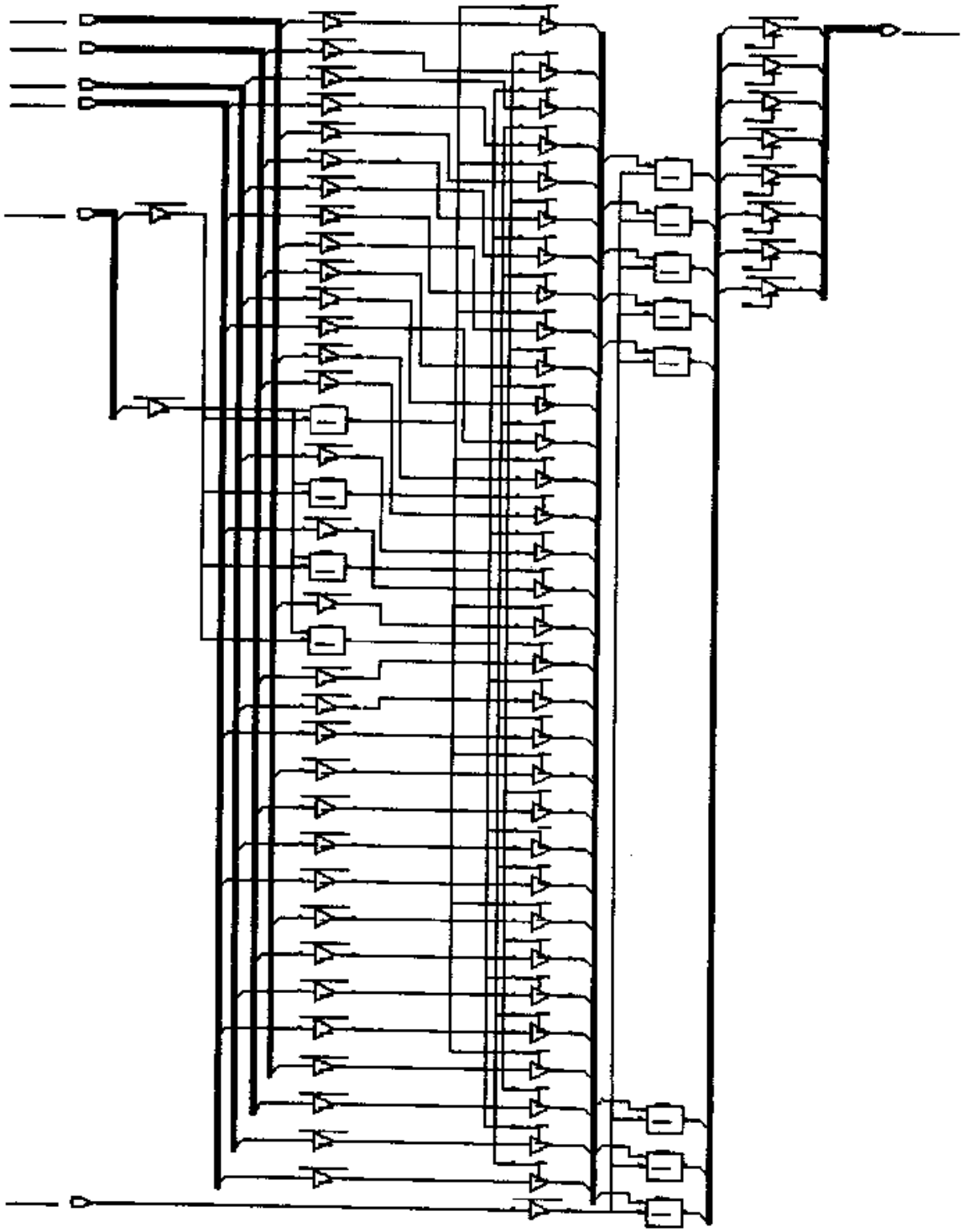

图 3-1　内部三态缓冲器设计示意图

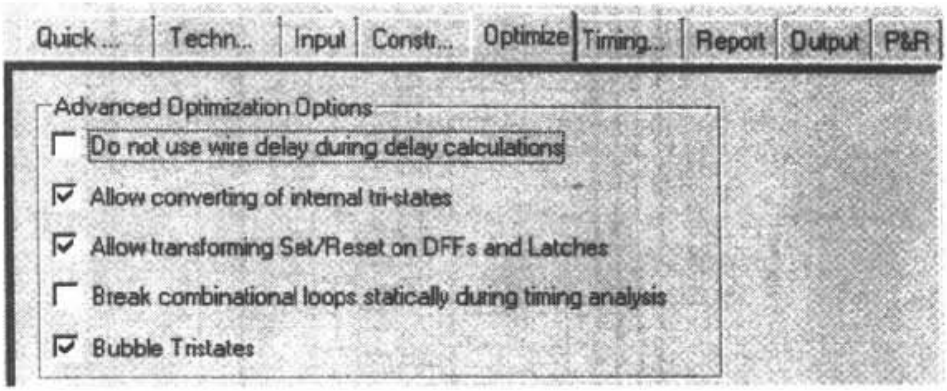

图 3-2 三态缓冲器被转换为 MUX

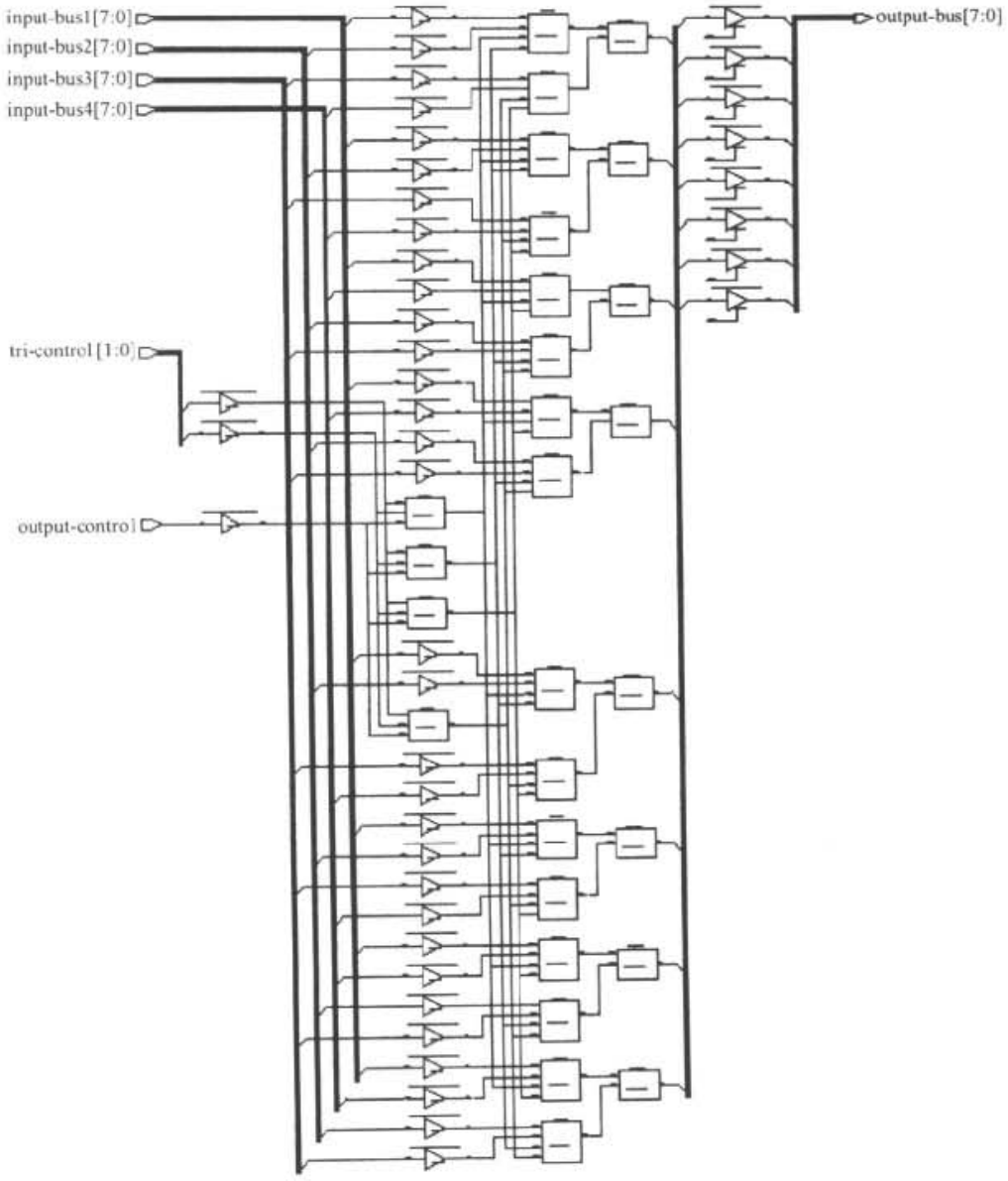

图 3-3 转换成 MUX 的三态缓冲器

3.3　双向总线

在 Verilog 中可方便地定义如图 3-4 和程序列表 3-4 的双向总线。信号被分为两部分:三态驱动器部分和输入部分。这两部分可用导线连接。模块端口必须被定义为“inout”类型。

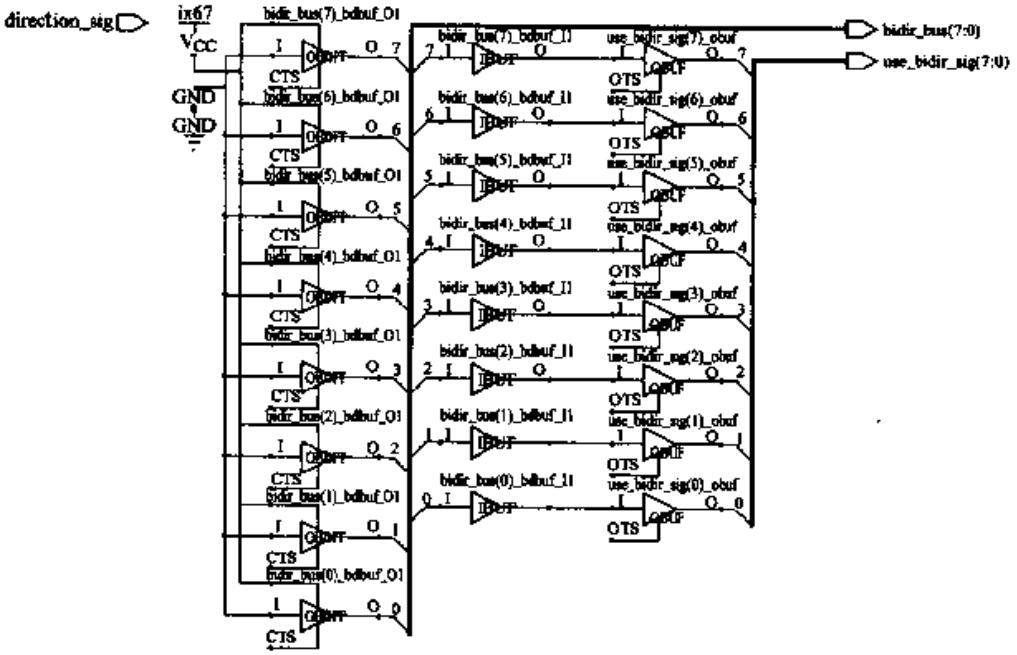

图 3-4　双向总线设计

程序列表 3-4　双向总线实例

```
module bidir (bidir_bus, direction_sig, use_bidir_sig);
inout   [7:0]    bidir_bus;
input            direction_sig;
output  [7:0]    use_bidir_sig;
reg     [7:0]    output_bus;
wire    [7:0]    bidir_input;

// 当 direction_sig 为真时,output_bus 驱动 bidir_bus 端口的引脚。
// bidir_bus 信号在设计内部的 bidir_input 总线上可以得到。

// 输出部分,MUX 形式。
```

```
assign bidir_bus = direction_sig ? output_bus : 8'bz;

// 输入部分。
assign bidir_input      = bidir_bus;
//对输入进行了赋值,所以该输入没有被综合工具优化掉。
assign use_bidir_sig    = bidir_input;
    endmodule
```

3.4 优先编码器

3.4.1 if/else 优先级编码

由于是 begin/end 语句内的第一个指令,所以 if-else 语句实际上具有隐含的优先权。程序列表 3-5 是一个优先编码器,其示意图如图 3-5。如果信号 a 是确定的,它就具有优先权。从时延的观点来看,信号 x 经过一级逻辑,而信号 z 要经过三级逻辑,所以 x 的速度比 z 快。

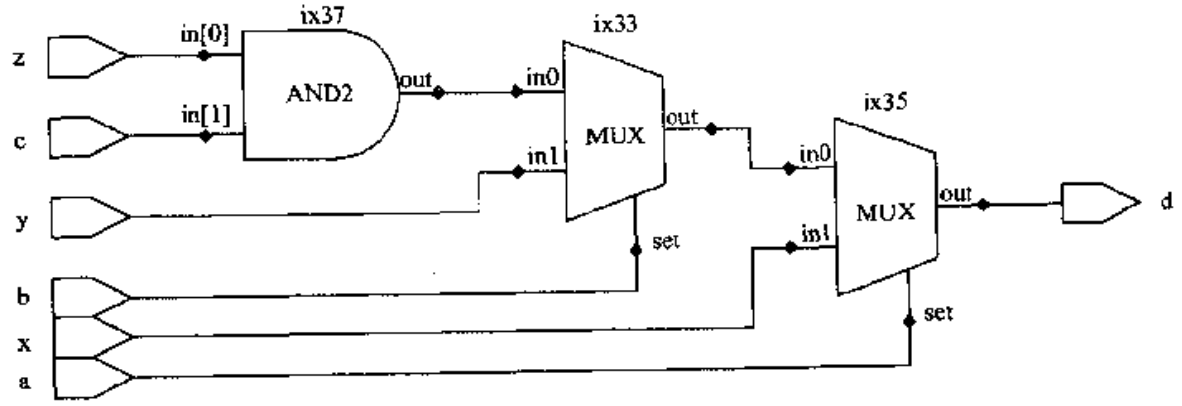

图 3-5 优先编码器电路示意图

程序列表 3-5 优先编码器实例

```
module priority (d, a, b, c, x, y, z);
input   a, b, c, x, y, z;
output  d;
reg   d;
```

```
always @ (a or b or c or x or y or z)
    begin
        if (a) d = x;
        else if (b)
            d = y;
        else if (c)
            d = z;
        else
          d = 1'b0;
    end
endmodule
```

3.4.2　case 语句中的优先级

在没有选择 full case 或所有的输入组合都定义了输出状态的情况下，同 if/else 语句一样，case 语句将创建一个优先编码器。选择 full case 状态意味着告诉编译器每个事件是互斥的，并不重叠。如果一个事件为真，则其他的事件就不能定义为真。既然并行事件的设计不存在冲突，则这些事件就不需要设置优先级；当事件有冲突时，才需要设置优先级。如果有一个以上的事件为真，那么编译器遇到的第一个事件具有最高的优先级（在计算高优先级的事件时，低优先级的事件被认为是不需要考虑的）。这意味着语句的顺序也是需要注意的。为了避免优先级编码，要确定所有的事件都涉及了，或是使用编译器指令来设置为 full case 型，还要避免事件冲突。

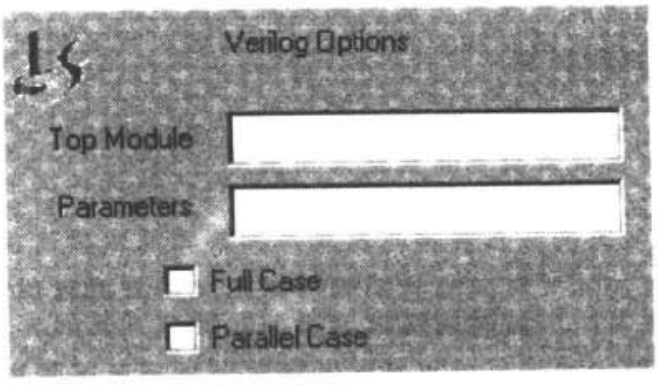

图 3-6　LeonardoSpectrum 中的 Case 选项

当输入状态列表不完全时，会创建如程序列表 3-6 和图 3-7 的锁存器。因为没有定义事件，所以保持先前的输出。而这不是设计者想期望的。要避免建立锁存

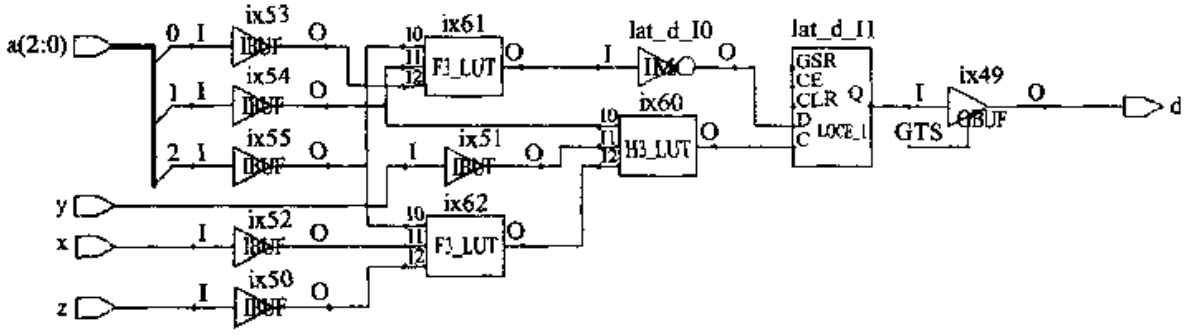

图 3-7 有锁存器的 Case 电路示意图

器,就应使用缺省的事件来涉及所有未定义的事件,如图 3-7 所示;或检查 LeonardoSpectrum 的 full case 选项,如图 3-6。用 full case 选项创建 MUX,如图 3-8。

并行事件复选框将创建锁存器并把所有未定义的输出强制为已知状态,如图 3-9 所示。

程序列表 3-6 创建锁存器的事件实例

```
module case1 (d, a, x, y, z);
input   [2:0]a;
input        x, y, z;
output       d;
reg          d;

always @ (a or x or y or z)
begin
    case (a)
    3'b001:  d  =  x;
    3'b010:  d  =  y;
    3'b100:  d  =  z;
    endcase
end
endmodule
```

程序列表 3-7 的不同之处是:如图 3-8 所示,所有未定义的事件都被置为缺省值"0"。所以,图 3-10 中没有锁存器。

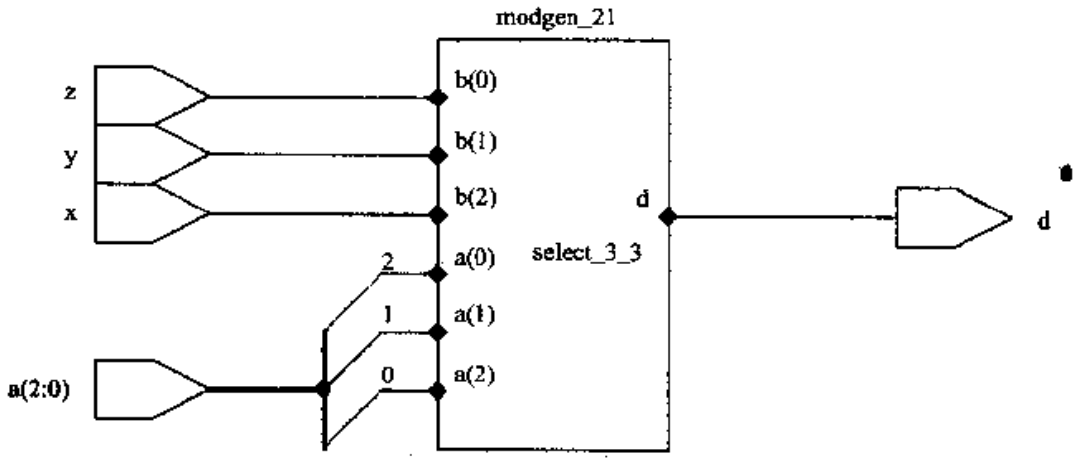

图 3-8　Full Case 方式下的 Case 电路示意图

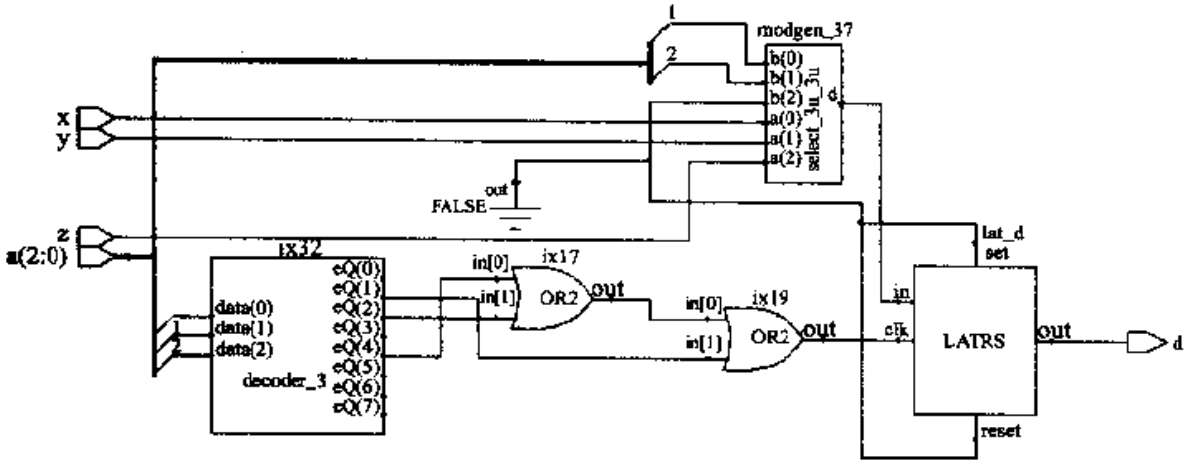

图 3-9　有锁存器的并行 Case 电路示意图

程序列表 3-7　缺省值事件实例

```
module case2 (d, a, x, y, z);
input     [2:0]a;
input          x, y, z;
output         d;
reg            d;

always @ (a or x or y or z)
```

```
begin
    case (a)
    3'b001:  d  =  x;
    3'b010:  d  =  y;
    3'b100:  d  =  z;
    default:  d  =  1'b0;
    endcase
end
endmodule
```

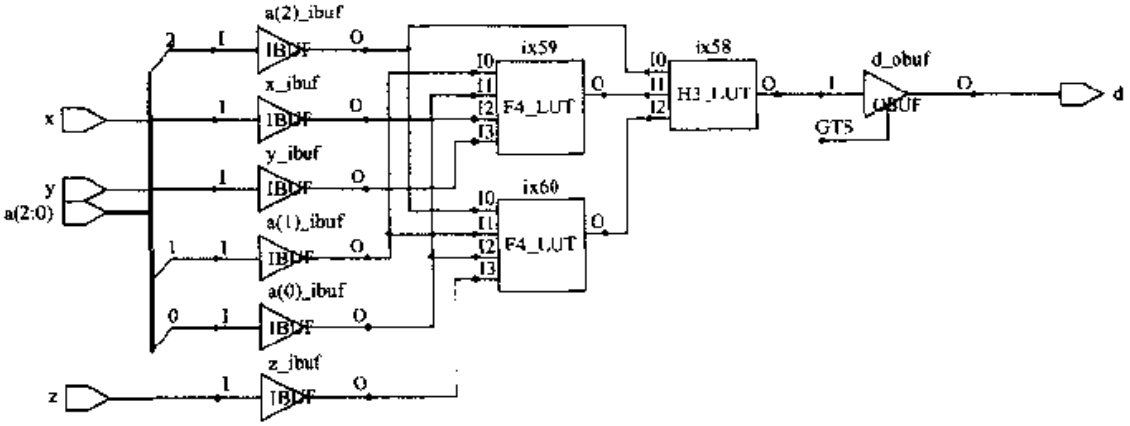

图 3-10　缺省值 Case 示意图

3.5　综合中面积/速度的优化

我们对综合器的功能已经有了初步的概念，现在来更深入地了解它。对 ASIC 和 FPGA 综合的优化处理可分为两个通用的策略，速度/延时或面积。很明显，设计必须满足所要求的运行速度。FPGA 的速度越快，其价格就越高。所以设计必须选择适合的元件。元件规模越大，则其价格越高。FPGA 设计者经常考虑被综合的逻辑电路的面积和速度。面积和速度会受到编码风格的影响，在稍后的内容里会有所介绍。

程序列表 3-8　被复制的逻辑实例

```
module optimize(a, b, c, d, e, f, g ,h , i, j, k, l, m, z);
output      a, b, c, d, z;
```

```
reg   a, b, c, d, z;
input      e, f, g, h, i, j, k, l, m;

always @ (e or f or g or h or i or j or k or l)
    begin
        a   =   e & f & g & h & i & j & k & l;
        z   =   m & a;
    end

always @ (e or f or g or h or i or j or k or l)
    begin
        b   =   e & f & g & h & i & j & k & l;
    end

always @ (e or f or g or h or i or j or k or l)
    begin
        c   =   e & f & g & h & i & j & k & l;
    end

always @ (e or f or g or h or i or j or k or l)
    begin
        d   =   e & f & g & h & i & j & k & l;
    end

endmodule
```

图 3-12 和图 3-13 中的电路具有细微的差别，但在逻辑功能上是完全一致的。这两个设计的差别在于 LeonardoSpectrum 中 Quick Setup 栏上面积/延时优化的选择。应该注意图 3-12 中信号 z 通过两级逻辑，而图 3-13 中的信号 z 则通过了三级逻辑。

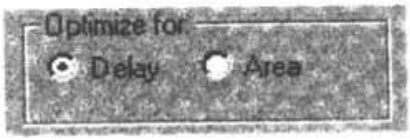

图 3-11　LeonardoSpectrum 的面积/延时优化选项

另一方面要注意此设计中的关键路径。关键路径是设计中时延最长的路径，LeonardoSpectrum 提取此路径并进行分析。关键路径如图 3-14 和图 3-15 所示。

图 3-15 显示了时延的改善。对于这个设计，这两条关键路径分别是 22.78ns

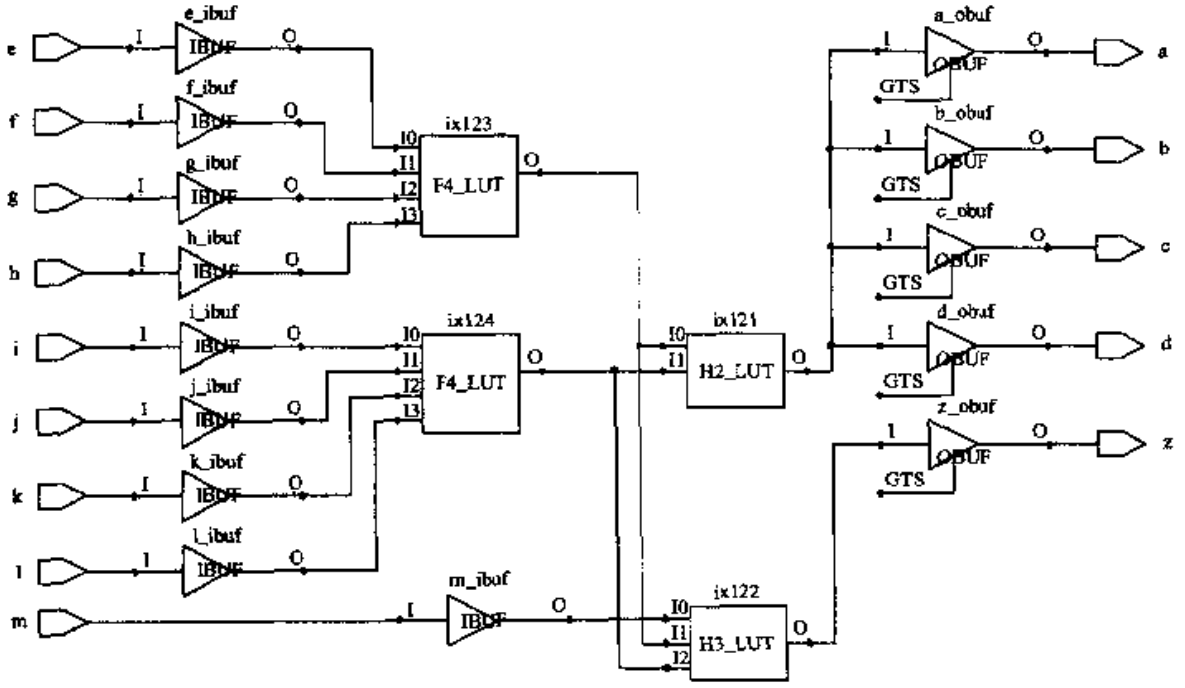

图 3-12 速度优化后的信号和布线实例

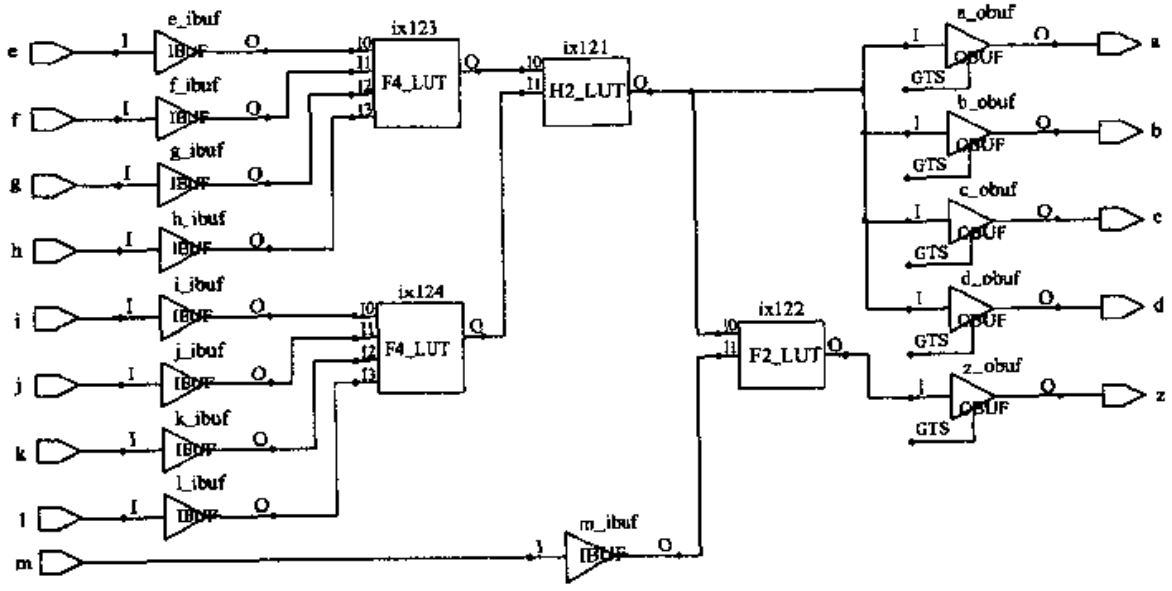

图 3-13 面积优化后的信号和布线实例

(时延优化)和 24.57ns(面积优化,相差 10%)。FPGA 设计者面临的挑战是使设计尽可能紧凑(便宜),使用速度尽可能慢的元器件,仍能具备所要求的性能。

从此例可看出,在一开始就应该明确实现一个逻辑功能的综合工具有多少选项。考虑到编码类型,此例表示将共享输入和/或输出逻辑分组对综合工具是有用

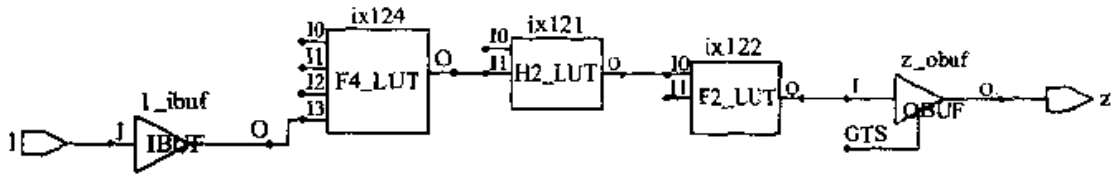

图 3-14　面积、优化后的关键路径

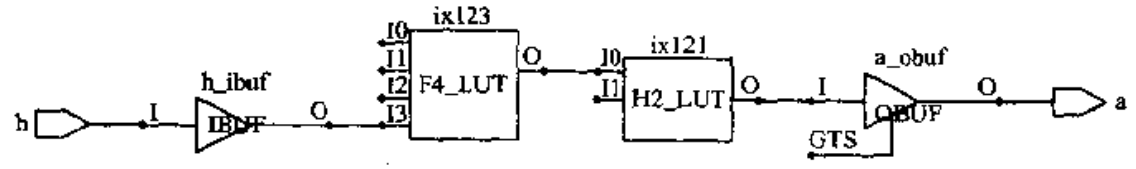

图 3-15　延时、优化后的关键路径

的。最好的划分应使它们尽可能在一个程序段或模块中。对于简单的输入和输出，不要使用综合器来完成分组功能。

还有一些有所帮助的设计方法，在一个设计工作组中，对于模块边缘触发器的安排应该达成一致的看法。图 3-16 是一个典型的实例。只要模块能保持这种形式，它们就能很好的工作。若没有达成一致的意见，就必须在输入端使用同步触发器，因为你不知道将面临什么样的情况。

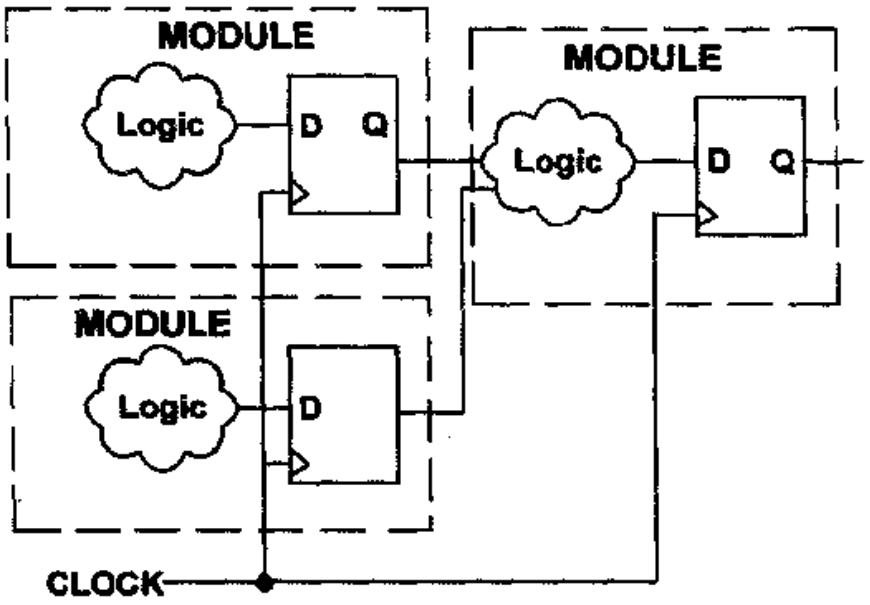

图 3-16　建议采用的模块边界选择和寄存器赋值

面积和时延有时是可以兼顾的。有时更紧凑的设计有更快的运行速度。如果电路有较少的门，那么它必然就具有更少的逻辑等级。

3.6 在运行速度和级联时间之间折中

面积和速度的另一种折中是在等待时间和电路运行速度之间。等待时间是信号从输入端经过电路传输到输出端所用的时间。只要吞吐量足够大，一个设计就能有多个等待时间。对于较大的吞吐量，设计者应该对逻辑进行分割，减少时钟之间的操作。程序列表 3-9、程序列表 3-10 是相关实例。这两个设计在功能上是相同的，但程序列表 3-10 中的设计具有更长的等待时间（输出结果出现在输出端需要三个时钟周期），运行在较高的时钟速率。程序列表 3-9 使用了较少的触发器，具有较短的等待时间（到达输出端只需要一个时钟周期），以较低的时钟速率运行。

程序列表 3-9 较短等待时间，较低运行速度的逻辑

```
module latency1 (clk, reset, a, b, c, d, e, f);
input                clk, reset;
input    [15:0]      a, b, c , d, e;
output               f;
reg      [15:0]      f;

always @ (posedge clk or posedge reset)
    begin
        if (reset)
            f     <=     0;
        else

        // 下一等式在一个时钟周期内解出。
            f   <=   ( a & b & c & d & e);
    end
endmodule
```

程序列表 3-10 较长等待时间和较高运行速度的逻辑

```
module latency2 (clk, reset, a, b, c, d, e, f);
input                clk, reset;
input    [15:0]      a, b, c, d, e;
```

```
output              f;
reg      [15:0]     ab, cd, f;

always @ (posedge clk or posedge reset)
    begin
        if (reset)
            begin
            ab   < =   0;
            cd   < =   0;
            f    < =   0;
            end
        else

    //剩余的每一个等式都应在一个时钟周期内解出。
            begin
              ab   < =   (a & b);
              cd   < =   (c & d);
              f    < =   (ab & cd & e);
            end
    end
endmodule
```

作为比较，设 Xilinx 4010XL-3 为目标器件，程序列表 3-9 中的电路使用 16 个 CLB，并在 78.9MHz 的频率上运行。程序列表 3-10 中的电路使用 24 个 CLB，在 87.1MHz上运行。我们添加一些硬件资源以提高操作速度。采用流水线方式来处理在一个时钟周期内必须完成的逻辑，几乎任何电路都可以通过这种方式加速。

综合工具为实现设计逻辑做了大量的工作，但设计者必须负责整个设计结构。如果一个设计需要在较高的时钟频率上运行，那么就要对逻辑进行分解以使两个时钟沿之间要完成的逻辑减少。只有不善此道的设计者才会对设计中所使用的工具的性能进行抱怨，因为他们不懂得如何寻找改善时序设计的方法。

3.7　FPGA 逻辑单元的延时

3.7.1　时序约束

提高设计性能有两种方法。第一种方法：通过应用时序约束条件来协助综合

工具识别关键逻辑。时序约束将在第六章具体讨论。这种方法可为需要高速运行的逻辑加上优先级。第二种方法:编写代码使综合工具要完成的工作更简单。大多数问题都能通过源代码的修改来完成。

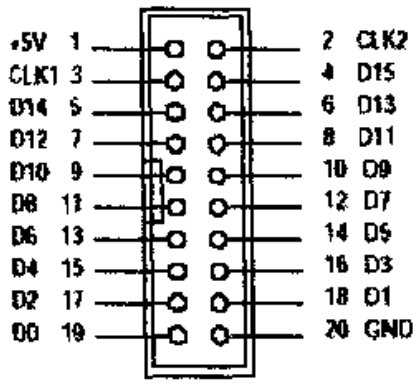

图 3-17 逻辑分析仪插槽图

3.7.2 测试点

即使是经验丰富的设计者也会出错。因为了解这一点,所以有经验的设计者所做的设计都是便于测试和编译的。故障查找的一个有效方法就是将测试点联结到易于接入的线路。例如,惠普的逻辑分析仪的插槽如图 3-17 所示。第 20 引脚接地,逻辑分析仪的时钟接到引脚 2 和引脚 3。每个引脚都已编号,一边是奇数,从 1 到 19;另一边是偶数,从 2 到 20。将测试值[15:0]分配到设备引脚(该引脚已连到测试连接装置),加入信号进行测试。具体实例见程序列表 3-11。

程序列表 3-11 测试点接线实例

```
module top_lev (test, clk, reset);
output          test;
wire    [15:0]  test;
wire    [15:0]  cnt;
input           clk, reset;

assign test[15:0]  =  cnt[15:0];

lower_level u1 (cnt, clk, reset);
endmodule

module low_level (cnt, clk, reset);
input           clk, reset;
output          cnt;
reg     [15:0]  cnt;

always @ (posedge clk or posedge reset)
begin
```

```
    if (reset)
        cnt      <=   0;
    else begin
        cnt      <=   cnt + 1;
        end
end
endmodule
```

20个引脚的惠普(p/n1251-8106)逻辑分析仪的插槽固定在一个终端网络上

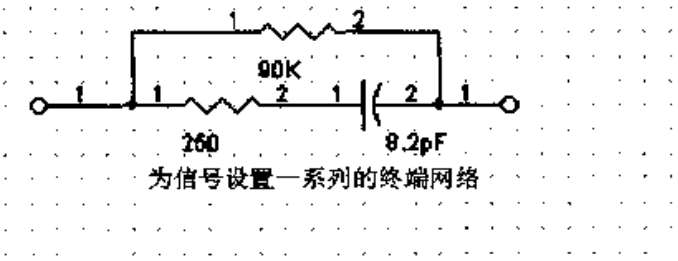

图 3-18　逻辑分析仪插头终端网络电路示意图

(如图 3-18),并插入到一个有双排引脚的0.1in[①]的插头上(如图 3-17 所示)。为了节省插槽的花费,为了能直接与 40 个引脚的逻辑分析仪插槽相连(如图 3-19),将一个双排 40 引脚、0.1in 的中心插头固定在底板上。

为了扩展测试中可获得的信号数量,可以使用 MUX 来选择测试点上的不同信号。设计者可能会陷入海森堡测不准原理中。意思是,进行测试的方法会影响正在被测试的数据。要明白,通过 MUX 的信号是从内部信号中删除了一个或多个逻辑级,而且时序精度也不完全相同。对于低速信号而言,这并不重要。在测试点加入逻辑或布线会使布线变得复杂,时序变慢。在你的实验室中要学习测试技术,注意了解你的测试设备。设计易于接入测试设备的电路

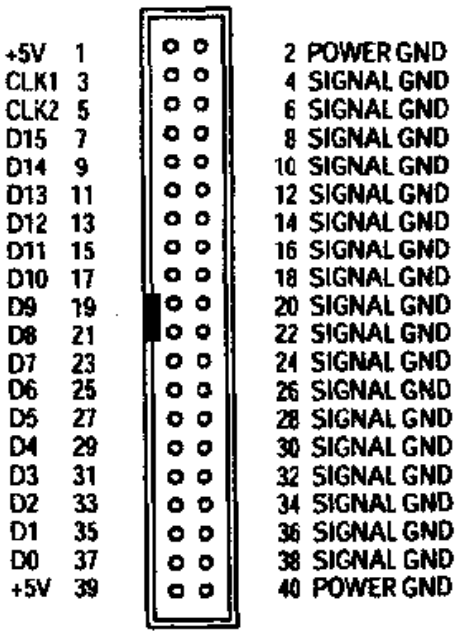

图 3-19　逻辑分析仪插头图

① 1in = 2.54cm。——译者注

板,并且使其便于连接到示波器或电压表;电路板应该合理布线以便使用测试线夹。

3.8 状 态 机

利用顺序过程来处理设计是很常见的。有限状态机(FSM)是用一组状态寄存器来识别当前机器状态。当前的状态取决于现在的输入和以前的输入。

作者认为有限状态机是一个技术上的奇迹。作为设计者,我们总是试图将复杂的问题分割成很多部分,这样每个小部分的解决就会相对简单。有限状态机能很好地满足这个要求。一旦确定了当前所在的状态,那么相关的输入也就明确了。

有限状态机有多种形式,在教科书中,它们被分为 Mealy 型和 Moore 型。在 Moore 型有限状态机中,输出只取决于状态。同步计数器就是 Moore 型有限状态机的一个例子,其输出只取决于机器的状态。Mealy 型有限状态机的输出则取决于机器状态和一些输入的信号。本书推荐类型是将输出逻辑和状态分配组合在同一个 always 块中。

用普通的办法来创建有限状态机是非常有用的。对于输出格雷码编码信号的计数器而言,每一个顺序输出都恰好只有一位数据不同。在本章稍后的内容中,我们将讨论格雷码。格雷码计数器是一个简单的有限状态机,下面用一些门电路和寄存器来设计一个这样的计数器。首先,创建一个当前状态/下一状态表,如程序列表 3-12 中,当前状态值在一个时钟沿之后将被下一状态值代替。

程序列表 3-12 状态机实例,3 位格雷码计数器

当前状态			下一状态		
d2	d1	d0	n2	n1	n0
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	0	1	0
0	1	0	1	1	0
1	1	0	1	1	1
1	1	1	1	0	1
1	0	1	1	0	0
1	0	0	0	0	0

收集所有决定下一状态其值为"1"的项,并做或运算以建立一个乘积和(SOP)来表示下一状态解码器逻辑,如程序列表 3-13 所示。

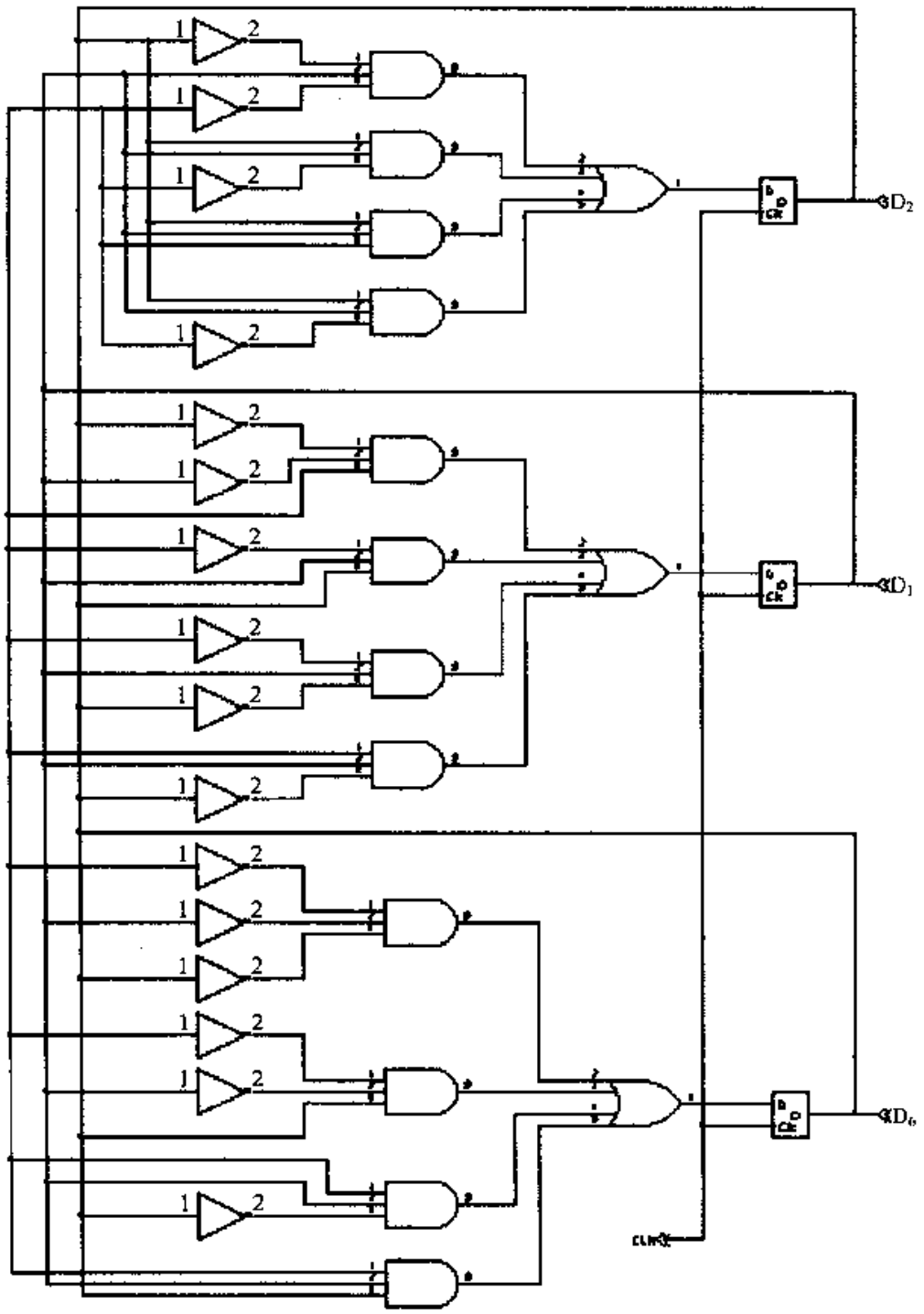

图 3-20　格雷码状态机逻辑

程序列表 3-13 有限状态机实例,下一状态逻辑

```
n2 <= (~d2 & d1 & ~d0) | (d2 & d1 & ~d0) | (d2 & d1 & d0) | (d2
& ~d1 & d0)

n1 <= (~d2 & ~d1 & d0) | (~d2 & d1 & d0) | (~d2 & d1 & ~d0) |
(d2 & d1 & ~d0)

n0 <= (~d2 & ~d1 & ~d0) | (~d2 & ~d1 & d0) | (d2 & d1 & ~d0)
| (d2 & d1 & d0)
```

图 3-20 中,左边是下一状态解码逻辑,右边是状态寄存器。用 Verilog 状态机来实现当前/下一状态逻辑,如程序列表 3-14 中使用 case 结构来创建下一状态解码器。虽然这些代码表面上看起来与上述的逻辑不同,但实际上是等价的。

程序列表 3-14 格雷码状态机的 Verilog 实例

```
module gray1 (clk, reset, cnt, flag_output);
input                clk, reset;
output               cnt;
wire       [2:0]     cnt;   // cnt 是当前逻辑状态。
reg        [2:0]     next_state;
output               flag_output;
reg                  flag_output;

assign cnt     =     next_state;

always @ (posedge clk or posedge reset)
    if (reset)  begin
        next_state      <=   3'b0;
        flag_output     <=   1'b0;
    end

    else begin case (next_state)
    3'b000:
```

```
        begin
            next_state      <=  3'b001;
            flag_output     <=  1'b0;
        end

    3'b001:
        begin
            next_state      <=  3'b011;
            flag_output     <=  1'b0;
        end

    3'b011:
        begin
            next_state      <=  3'b010;
            flag_output     <=  1'b1;
        end

    3'b010:
        begin
            next_state      <=  3'b110;
            flag_output     <=  1'b0;
        end

    3'b110:
        begin
            next_state      <=  3'b111;
            flag_output     <=  1'b0;
        end

    3'b111:
        begin
            next_state      <=  3'b101;
            flag_output     <=  1'b0;
        end
```

```
        3'b101:
            begin
                next _ state      < =   3'b100;
                flag _ output     < =   1'b0;
            end

        3'b100:
            begin
                next _ state      < =   3'b000;
                flag _ output     < =   1'b0;
            end
        default:
            begin
                next _ state      < =   3'b0;
                flag _ output     < =   1'b0;
            end
        endcase
        end
    endmodule
```

名为 flag _ output 的输出信号被加入到格雷码设计中，这将说明怎样为状态机添加一个输出。如果希望输出在 010 状态被确定，那么就应在前一状态 011 处将其设定。这样 flag _ output 信号在进入状态 010 时被确定，在退出状态 010 时被清除。此逻辑的波形如图 3-21 所示。

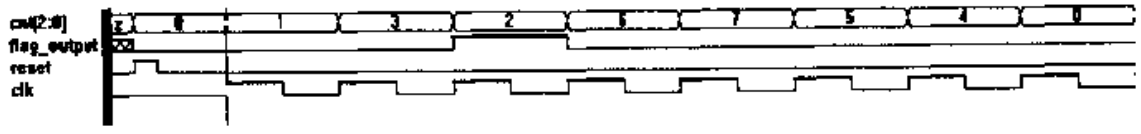

图 3-21 格雷码状态机波形

有多少个状态寄存器?

若 n 是状态寄存器的数目，则可使用的状态为：2^n

使用有限状态机是一种好方法。当状态机处于某个状态时，可以忽略时钟和

复位输入，除了逻辑和该状态明确涉及的输入，其他因素都可以忽略。怎样创建一个能高效综合的状态机呢？对于具有八个状态的寄存器和输入，作者认为将该状态机分为更小的部分更好些。

下面讨论另一种创建格雷码逻辑的方法。

四位格雷码
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000
0000

将二进制数转化为格雷码的算法如下：

格雷码 MSB　=　二进制数 MSB

下一位格雷码　=　（二进制数的相应位与下一个最高位异或）

下面的程序列表 3-15 是将二进制数转化为格雷码的算法。

程序列表 3-15　将二进制数转化为格雷码的 Verilog 代码

```
module binarytogray (clk, reset, binary _ input, gray _ output);
input          clk, reset;
input    [3:0] binary _ input;
output         gray _ output;
reg      [3:0] gray _ output;
```

```
always @ (posedge clk or posedge reset)
    if (reset)
    begin
        gray _ output < = 4'b0;
    end

    else begin
            gray _ output[3] < = binary _ input[3];
            gray _ output[2] < = binary _ input[3] ^ binary _ input[2];
            gray _ output[1] < = binary _ input[2] ^ binary _ input[1];
            gray _ output[0] < = binary _ input[1] ^ binary _ input[0];

    end
endmodule
```

将格雷码转化为二进制数的算法是:

二进制数 MSB = 格雷码 MSB
二进制数的下一位 = (前一位已确定的二进制数值与相应位的格雷码值异或)

将格雷码转化为二进制数的同步转换器如程序列表 3-16 所示。

程序列表 3-16 将二进制数转化为格雷码的 Verilog 代码

```
module gry2bin (clk, reset, gray _ in, binary _ output);
input               clk, reset;
input    [3:0]      gray _ in;
output              binary _ output;
reg      [3:0]      binary _ output;

always @ (posedge clk or posedge reset)
    if (reset)
    begin
        binary  output < = 4'b0;
    end
```

```
    else begin
            binary_output[3] <=      gray_in[3];
            binary_output[2] <=      gray_in[3]^gray_in[2];
            binary_output[1] <= (gray_in[3]^gray_in[2])^gray_in[1];
            binary_output[0] <= ((gray_in[3]^gray_in[2])^gray_in
            [1])^gray_in[0];

    end
endmodule
```

3.8.1 状态赋值

逻辑综合的效率与状态赋值有很大的关系。我们使用参数和“ifdef”语句在编码赋值中进行选择，如程序列表 3-17。二进制计数最便于测试和编译，但使用顺序格雷码会具有最高的综合效率。通过对状态寄存器和输出寄存器使用触发器，状态机的状态编码也可直接用于产生输出信号。这种方法能省去很多逻辑并能提高设计的运行速度。

程序列表 3-17　选择二进制/格雷码的状态赋值实例

```
module ifdef_test (clk, reset, count_output);
input          clk, reset;
output         count_output;
reg    [2:0]   count_output;

// 'define binary

'ifdef binary
parameter   state_zero   =   3'b000;
parameter   state_one    =   3'b001;
parameter   state_two    =   3'b010;
parameter   state_three  =   3'b011;
parameter   state_four   =   3'b100;
parameter   state_five   =   3'b101;
```

```
parameter  state_six   =  3'b110;
parameter  state_seven =  3'b111;

'else
parameter  state_zero  =  3'b000;
parameter  state_one   =  3'b001;
parameter  state_two   =  3'b011;
parameter  state_three =  3'b010;
parameter  state_four  =  3'b110;
parameter  state_five  =  3'b111;
parameter  state_six   =  3'b101;
parameter  state_seven =  3'b100;

'endif

always @ (posedge clk or posedge reset)
    if (reset)
    begin
        count_output  <=  state_zero;
    end

    else begin case (count_output)
    state_zero:     count_output <= state_one;
    state_one:      count_output <= state_two;
    state_two:      count_output <= state_three;
    state_three:    count_output <= state_four;
    state_four:     count_output <= state_five;
    state_five:     count_output <= state_six;
    state_six:      count_output <= state_seven;
    state_seven:    count_output <= state_zero;

    default:  count_output <= state_zero;

    endcase
```

```
    end
endmodule
```

3.8.2　One-Hot 状态赋值

One-Hot 状态赋值对一些设计有所帮助。One-Hot 的含义是每个状态只分配给一个触发器，此触发器只在其分配状态时才被激活。这种类型的编码将展开 FPGA 逻辑并能使逻辑的综合更简单。大多数 FPGA 结构都有许多寄存器，所以在 One-Hot 中占用一些寄存器并没有多大影响。一个 One-Hot 状态机所使用的寄存器比格雷码/二进制编码的状态机要多。但是，One-Hot 设计并不是完全固化的。在一些情况下，One-Hot 设计使用的逻辑比二进制或格雷码要多。所以在使用 One-Hot 前，一定要清楚它是否有益。

One-Hot 赋值要考虑的另一个方面是必须处理许多未使用的或默认的状态

程序列表 3-19 One-Cold 状态赋值代码段

```
parameter  state_zero   =  8'b11111110;
parameter  state_one    =  8'b11111101;
parameter  state_two    =  8'b11111011;
parameter  state_three  =  8'b11110111;
parameter  state_four   =  8'b11101111;
parameter  state_five   =  8'b11011111;
parameter  state_six    =  8'b10111111;
parameter  state_seven  =  8'b01111111;
```

3.9 加 法 器

Verilog 的综合器支持二进制加法器。综合工具检验每个“+”运算符并试图用预先优化的模块来实现。可能的优化是面积或速度/延时优化，编译环境也会影响优化的处理。如果进行一个规模小，运行速度慢的设计，那么通常可使用加法器，一般不会产生什么问题。但是，如果输入向量大，那么要执行代码 a < = b + c 就需要很多的逻辑。

设计者应该清楚综合工具会搜寻加法器，所以应尽量将加法器单独放置，而不是把它隐藏在设计中。

3.9.1 半加器逻辑

半加器对两个输入求和。之所以称为半加器是因为它忽略了进位输入信号。图 3-22 是一个半加器的真值表，其电路如图 3-23 所示。

a 输入端	b 输入端	进位输出	总和输出
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

图 3-22 半加器真值表

通过验证，可以算出应输出的进位是 a & b(即 a 与 b)，应输出的和是 a^b(a 异或 b)。

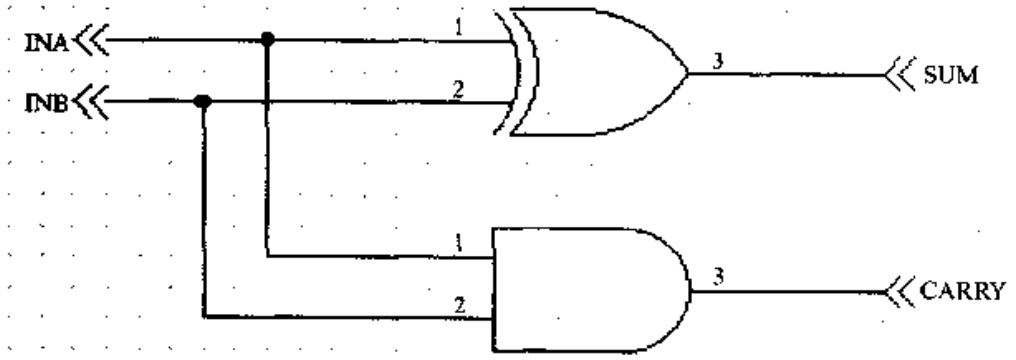

图 3-23　半加器电路示意图

如果认为可以比综合工具做得更好，那么就无需使用 Verilog 的半加综合器。在建立了字节宽度的加法器以后，我们可以将所得的结果同 LeonardoSpectrum 得到的结果相比，然后会发现我们比综合器聪明得多。

程序列表 3-20　半加器的 Verilog 方案

```
module half_adder (ina, inb, sum_out, carry_out, clk, reset);
    input       ina, inb, reset, clk;
    output      sum_out, carry_out;
    reg         sum_out, carry_out;
    always @ (posedge clk or posedge reset)
        begin
            if (reset)
            begin
                sum_out     <=  0;
                carry_out   <=  0;
            end
            else
            begin
                sum_out     <=  ina ^ inb;
                carry_out   <=  ina & inb;
            end
        end
endmodule
```

程序列表 3-20 中的半加器在对多比特数值求和时只能完成一半的工作。但实际中，很少有只对单比特数值求和的操作。要将半加器变为全加器，就要把一个半加器的输出连接到另一个半加器(图 3-25)。进位输出成为第二级的另一个输入。图 3-24 是全加器的真值表，其逻辑如程序列表 3-21。

a 输入端	b 输入端	进位输入	总和	进位输出
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

图 3-24 全加器真值表

程序列表 3-21 全加器的 Verilog 代码

```
// 全加器等式，级联两个半加器。
carry _ out   < =   ( ina & inb) | (( ina ^ inb ) & carry _ in )) ;
sum _ out     < =   ( ina ^ inb ) ^ carry _ in ;
```

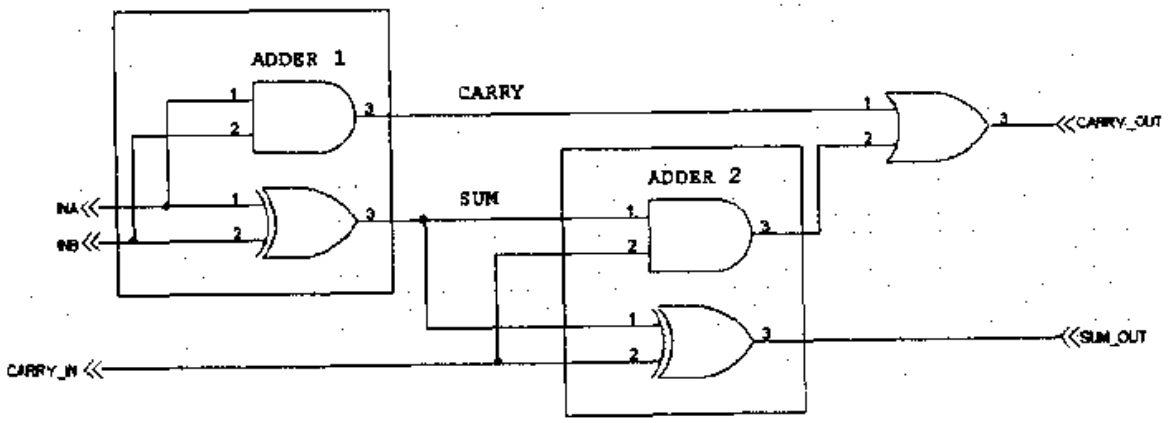

图 3-25 全加器电路示意图

程序列表 3-22　全加器的 Verilog 方案

```
module full_adder (carry_out, sum_out, ina, inb, carry_in, clk, reset);
  input      ina, inb, carry_in;
  input      clk, reset;
  output     carry_out, sum_out;
  reg        carry_out, sum_out;

  always @ (posedge clk or posedge reset)
    if (reset)
      begin
        carry_out <= 0;
        sum_out <= 0;
      end
    else

// 全加器等式,级联两个半加器。
      begin
        carry_out <= (ina & inb) + ((ina ^ inb) & carry_in);
        sum_out   <=  ((ina ^ inb) ^ carry_in);
      end
endmodule
```

读者可能会注意到,程序列表 3-22 和图 3-26 中,作者将组合型全加器设计变成了同步型。图 3-26 的 modgen 框图是一个或门。

为了创建宽度更大的加法器,可以将许多全加器进行级联。要创建大型的加法器,应该将设计分割成许多宽度小于四比特的小模块,然后再把它们结合在一起。这有助于综合器的综合并能提高面积和速度性能。

加法器的进位输出即是高一位比特的输入。注意:输出寄存器的长度应该大于输入寄存器,这样输出寄存器才能接收上一级的进位输出。但是若数据求和时不产生进位,则不会有上述要求。LSB 加法器的进位输入固定为 0。这种加法器的一个实例如程序列表 3-23。要使设计能正常工作,full_adder.v 必须包括在目标工程中。

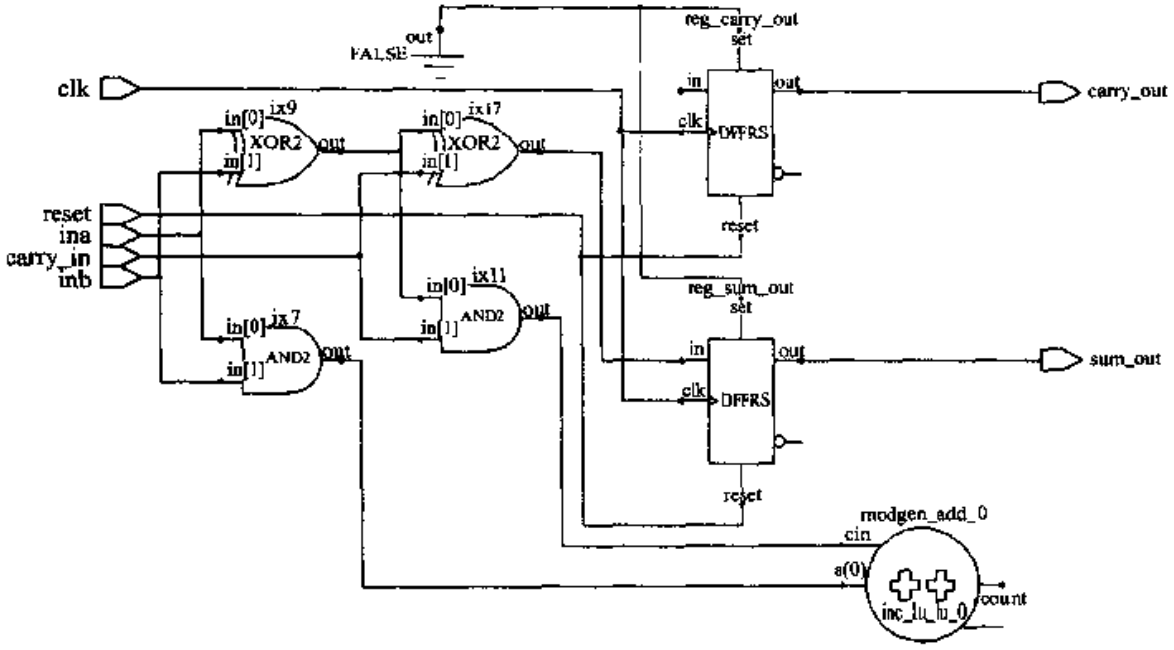

图 3-26 综合后的全加器电路示意图

程序列表 3-23 一个字节宽度的全加器结构方案

```
module byte _adder (byte _a, byte _ b, sum _ output, clk, reset);
  input    [7:0]    byte _ a, byte _ b;
  input             clk, reset;
  output   [8:0]    sum _ output;
  wire     [7:0]    carry _ output;

full _ adder u1 (carry _ output[0], sum _ output[0], byte _ a[0], byte _ b[0],
1'b0, clk, reset);
full _ adder u2 (carry _ output[1], sum _ output[1], byte _ a[1], byte _ b[1],
carry _ output[0], clk, reset);
full _ adder u3 (carry _ output[2], sum _ output[2], byte _ a[2], byte _ b[2],
carry _ output[1], clk, reset);
full _ adder u4 (carry _ output[3], sum _ output[3], byte _ a[3], byte _ b[3],
carry _ output[2], clk, reset);
full _ adder u5 (carry _ output[4], sum _ output[4], byte _ a[4], byte _ b[4],
carry _ output[3], clk, reset);
full _ adder u6 (carry _ output[5], sum _ output[5], byte _ a[5], byte _ b[5],
```

```
    carry_output[4], clk, reset);
    full_adder u7 (carry_output[6], sum_output[6], byte_a[6], byte_b[6],
    carry_output[5], clk, reset);
    full_adder u8 (carry_output[7], sum_output[7], byte_a[7], byte_b[7],
    carry_output[6], clk, reset);

    assign sum_output[8]  =  carry_output[7];

    endmodule
```

一个字节加法器设计的综合摘要如程序列表 3-24。

程序列表 3-24　一个字节宽度全加器综合摘要

```
Info, Instances dissolved by autodissolve in View
.work.byte_adder.INTERFACE
"C:/Verilog/SourceCode/byte_adder.v", line 7: u1 (full_adder)
"C:/Verilog/SourceCode/byte_adder.v", line 8: u2 (full_adder)
"C:/Verilog/SourceCode/byte_adder.v", line 9: u3 (full_adder)
"C:/Verilog/SourceCode/byte_adder.v", line 10: u4 (full_adder)
"C:/Verilog/SourceCode/byte_adder.v", line 11: u5 (full_adder)
"C:/Verilog/SourceCode/byte_adder.v", line 12: u6 (full_adder)
"C:/Verilog/SourceCode/byte_adder.v", line 13: u7 (full_adder)
"C:/Verilog/SourceCode/byte_adder.v", line 14: u8 (full_adder)
Using wire table: 4013xl-3_avg
Info, Inferred net 'reset' as GSR net.
-- Start optimization for design .work.byte_adder.INTERFACE
Using wire table: 4013xl-3_avg

      Pass     Area     Delay     DFFs  PIs   POs --CPU--
               (FGs)    (ns)                    min:sec
      1         16       12        16    18     9   00:00
Info, Added global buffer BUFG for port clk
Using wire table: 4013xl-3_avg
-- Start timing optimization for design .work.byte_adder.INTERFACE
No critical paths to optimize at this level

*******************************************************

Cell: byte_adder    View: INTERFACE    Library: work

*******************************************************

 Number of ports :                    27
 Number of nets :                     68
 Number of instances :                51
 Number of references to this view :   0
```

```
Total accumulated area :
 Number of BUFG :                          1
 Number of CLB Flip Flops :                7
 Number of FG Function Generators :       16
 Number of IBUF :                         17
 Number of IOB Output Flip Flops :         9
 Number of Packed CLBs :                   8
 Number of STARTUP :                       1

************************************************
Device Utilization for 4010xlPQ100
************************************************
Resource                  Used    Avail    Utilization
-------------------------------------------------
IOs                       27      77        35.06%
FG Function Generators    16      800        2.00%
H Function Generators     0       400        0.00%
CLB Flip Flops            7       800        0.88%

-------------------------------------------------
                          Clock Frequency Report
     Clock                  : Frequency
      ---------------------------------
     clk                    : 77.6 MHz
```

程序列表 3-25 是由综合器完成所有工作的实例。程序列表 3-26 提供了设计综合的统计数字。

程序列表 3-25 一个字节宽度全加器的综合工具方案

```
module byte_adder2 (byte_a, byte_b, sum_output, clk, reset);
    input    [7:0]    byte_a, byte_b;
    input             clk, reset;
    output            sum_output;
    reg      [8:0]    sum_output;

    always @ (posedge clk or posedge reset)
        begin
        if (reset)   sum_output  <=  0;
        else         sum_output  <=  byte_a + byte_b;
        end
endmodule
```

程序列表 3-26　一个字节宽度全加器的综合统计

```
Info, Inferred net 'reset' as GSR net.
-- Start optimization for design .work.byte_adder2.INTERFACE
Using wire table: 4013xl-3_avg

      Pass      Area     Delay      DFFs  PIs    POs --CPU--
                (FGs)     (ns)                      min:sec
      1           8         7          9    18      9   00:00
Info, Added global buffer BUFG for port clk
Using wire table: 4013xl-3_avg
-- Start timing optimization for design
.work.byte_adder2.INTERFACE
No critical paths to optimize at this level

*******************************************************

Cell: byte_adder2    View: INTERFACE    Library: work

*******************************************************

 Number of ports :                          27
 Number of nets :                          103
 Number of instances :                      48
 Number of references to this view :         0

Total accumulated area :
 Number of BUFG :                            1
 Number of CY4 :                             5
 Number of FG Function Generators :          8
 Number of IBUF :                           17
 Number of IOB Output Flip Flops :           9
 Number of STARTUP :                         1

*************************************************
Device Utilization for 4010xlPQ100
*************************************************
Resource                 Used    Avail    Utilization
-------------------------------------------------
IOs                      27      77        35.06%
FG Function Generators   8       800        1.00%
H Function Generators    0       400        0.00%
CLB Flip Flops           0       800        0.00%

-------------------------------------------------
                         Clock Frequency Report

     Clock                 : Frequency
      ---------------------------------
     clk                   : 135.1 MHz
```

将此字节宽度加法器方案与 LeonardoSpectrum 的方案进行比较，则此方案的面积要大 2 倍，而速度却慢 2 倍。

我们可以通过查找设计问题来改进此方案的性能。我们所创建的加法器是行波进位加法器(RCA)，它的面积较小，但速度较慢。RCA 的问题很容易看出。假

设,将两个二进制数 10101 和 11001 相加。那么最高位比特相加(1 + 1)的结果是什么? 在没有得到所有低位的进位前,这个问题回答不了。而这就是问题所在。在高一位的数据进行相加前,其低位的所有数据都必须已经计算完毕;直到所有位的计算都完成,才能得到最后的和与进位。如果能够用并行计算而不是串行计算,那么加法器的速度就会快得多,估算输入的进位并将其与部分和相加。用这种想法创建的加法器称为先行进位加法器(CLA 加法器)。

程序列表 3-27 进位产生和进位传输逻辑代码段

```
// propagate/generate 的定义。(不是 Verilog 的标准用法)
generate[i]   =  ( a[i] & b[i] );
propagate[i]  =  ( a[i] ^ b[i] );
```

按照产生/传输信号来描述一个可级联的(可扩展的)加法器需要添加前一级进位,如程序列表 3-28 所示。

程序列表 3-28 进位产生和进位传输逻辑,可扩展的加法器

```
// 可级联的 propagate/ generate 项。(不是 Verilog 的标准用法)

carry[i]      =     generate[i] | ( propagate[i] & carry[i - 1]) ;
propagate[i]  =     ( a[i] | b[i] ) ;
```

和仍然由级联半加器形成,并行地计算和与进位。为使每个模块都是统一的,在 s[0]级不允许有进位,其进位输入应直接连到"0"。

3.9.2 进位选择加法器

提高加法器速度的另一个方法是添加更多的硬件。无论有无进位都可以添加冗余的硬件来进行计算,并通过多路转换器来选择输出。

3.9.3 进位省略加法器

另一种提高加法器速度的办法,即在输入不相等时使用反相器。采用异或运算来计算输出,和位是进位位的反转,进位位是前一个进位的传递。这种类型的加法器通常分组实现,多半是四位一组。这种方式所做的估算比 CLA 少,速度也比较快。

还有其他一些加法器设计技巧，主要是：在使用 Verilog 加法运算符时对要综合的逻辑进行估算。使加法器的输入长度变短，然后找到能够减少重复设计的解决方案。如果可以采用 14 位的加法器，就不要用 16 位的加法器。同时，要在尽可能低的时钟频率下运行加法器。

3.10　减　法　器

减法器与加法器相似，减法器也有上述讨论的各种加法器的对应类型（借位减法器、借位省略减法器、借位选择减法器等）。首先我们讨论借位减法器，图 3-27 显示了输入 a 和 b 的半借位减法电路。

a 输入端	b 输入端	借位输出	差输出
0	0	0	0
0	1	1	0
1	0	0	1
1	1	0	0

图 3-27　半减器真值表[①]

程序列表 3-29　减法器逻辑代码段

```
Bout = ~ina & inb ; // 注意与加法器的相似之处：carry = ina & inb。
Diff = ina & ~inb ; // 注意与加法器 xor 的相似之处：
// sum = ( ina & ~inb ) | ( ~ina & inb )。
```

将半借位减法器扩展为全减器，其真值表如图 3-28。

借位输入	a 输入端	b 输入端	借位输出	差
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1

图 3-28　全减器真值表

① 原作者误写为“半加器真值表”。——译者注

3.11 乘 法 器

Verilog 语言通过二次幂支持无符号型乘法和除法。这并不是很复杂的工作，只需进行左移操作(二进制数乘以二的运算是将每个比特向左移一位)或右移操作(二进制数除以二是将每个比特向右移一位)。FPGA 能够高速完成很复杂的数学运算。设计者应该与系统工程师一起工作以减少要相乘的二进制数的长度。在需用 7 个比特时一定不要用 8 个。用 C 语言做一个模型来测试数据并检验其结果会好些。尽量采用最简单的解决方案，只要能获得可接受的结果即可。如果输入的变量与常数相乘，那么乘法器的操作就会比较简单。如果输入范围是有限制的，就能省去一些逻辑。

乘法器的结果需要 n + m 个比特的宽度，其中 n 和 m 都是输入变量的宽度。例如，当两个 4 比特的变量相乘时，需要一个 8 比特的寄存器来存储可能的最大值。

3.11.1 硬布线乘法器

说明乘法的最好办法就是举实例。假设用一个四比特的整型变量 n 要乘以一个 4 比特的整型常数 D(16)。

2^n	2^n	2^n	2^n	权值
n3	n2	n1	n0	与常数 D(16)相乘的 4 比特变量 n
1	1	0	1	与变量相乘的 4 比特常数 D(16)

这个乘法的处理过程是移位和相加。最左边的数字意味着与 n0 相加，n1 可以被忽略，n2 的含义是用 4 乘以 n(将 n 左移 2 位)并相加，n3 的含义是乘以 8(将 n 左移 3 位)并相加。

结果 = (n * 8) + (n * 4) + (n * 1)

设变量是 B(16)或 1011(2)并将其代入上述公式。

结果 = (1011 * 8) + (1011 * 4) + (1011 * 1)

结果 = 1011000 + 101100 + 1011

结果 = 10001111 = 8F(16)

可以看出，乘法只是一系列的移位和加法操作。此乘法器的 Verilog 方案如程序列表 3-30 所示。可以使用移位操作器，也可以使用结构型加法器。为了提高设计速度，可以使用流水线方式来处理中间结果，这样就能减少触发器之间可观的组合逻辑量。

程序列表 3-30　硬布线乘法器实例

```
module byte_mult (nibble_in, byte_out, clk, reset);
    input    [3:0]    nibble_in;
    input             clk, reset;
    output            byte_out;
    reg      [7:0]    byte_out;

    always @ (posedge clk or posedge reset)
        if (reset)
            byte_out <= 0;
        else
        begin

        // 在移位时利用 0 来填充。MSB 必须为 0 以使左边的宽度与右
        // 边的宽度匹配。

            byte_out <=  {1'b0, nibble_in[3:0], 3'b0}
                       + {2'b0, nibble_in[3:0], 2'b0}
                       + {4'b0, nibble_in[3:0]};
        end
endmodule
```

硬布线乘法器是完全定制的，并不常用，所以在使用时一定要小心。必须通过改变代码来改变所乘常数。如果在 FPGA 设计中存在资源问题，那么在一组新的系数中很难再有足够的空间来改变原有的代码。给输入或常数添加解决方案会增加大量逻辑。

3.11.2　通用乘法器

要将硬布线乘法器变为一般的四乘四乘法器，必须创建一些逻辑以便使用所有的移位和加法运算(是否使用这些运算取决于数据值)。这样做的方法很多，程序列表 3-31 中的实例就是其中之一。

程序列表 3-31 一般四乘以四乘法器实例

```
module byte_mult2 (nibble1, nibble2, byte_out, clk, reset);
    input   [3:0]   nibble1, nibble2;
    input           clk, reset;
    output          byte_out;
    reg     [7:0]   byte_out, stored3, stored2, stored1, stored0;

    always @ (posedge clk or posedge reset)
    if (reset)
    begin
        byte_out  <=  0;
        stored3   <=  0;
        stored2   <=  0;
        stored1   <=  0;
        stored0   <=  0;
    end
    else
    begin

// 在移位时利用 0 来填充。MSB 必须为 0 以使左边的宽度与右边的宽度
// 匹配。

        stored3 <= nibble1[3] ? {1'b0, nibble2[3:0], 3'b0} : 8'b0;
        stored2 <= nibble1[2] ? {2'b0, nibble2[3:0], 2'b0} : 8'b0;
        stored1 <= nibble1[1] ? {3'b0, nibble2[3:0], 3'b0} : 8'b0;
        stored0 <= nibble1[0] ? {4'b0, nibble2[3:0]} : 8'b0;
        byte_out <= stored3 + stored2 + stored1 + stored0;
    end
endmodule
```

关于程序列表 3-31 还应该提及的是，Verilog 或许不能对寄存器的宽度做检测以确定寄存器是否能接受计算出的数据。通常，设计者希望所有的求和寄存器都能比所计算出的最大数值的宽度多一个比特位。但是，因为作者知道存储寄存器中的数据的性质(MSB 肯定为 0)，所以可以令存储的宽度与 byte_out 的宽度相同。

我们知道存储两个 4 比特数相乘的结果最多需要 8 比特的宽度。但是，做这种假设时一定要小心。

3.11.3　与分数相乘

通常，许多常量的系数都是分数的形式。如果系统工程师要求一个 0.80 的系数，那么乘以 0.75(2^{-1}(或 1/2) + 2^{-2}(或 1/4))能否产生可接受的结果？如果是这样的话，这项工作就简单多了。

要使 4 比特的变量 n 乘以 0.75，首先意识到 $0.75 \times 4 = 2$，$(n \times 4)/4 = n$。所以用 4 乘以该系数，然后将所得的结果除以 4。

如果系统工程师坚持乘以 0.8 的话，那么要弄清所要求的精度，然后将 0.8 用二进制数表示($0.8 = 1/2 + 1/4 + 1/32 + \cdots$)，再做所要求的移位和相加运算。

第四章　更多的数字电路:计数器、只读存储器及随机存储器

本章将介绍几个具体的运用 Verilog 语言实现的数字设计实例。

4.1　行波计数器

最普通的计数器应属行波计数器(因为其输出总是一级一级波动,故此得名)。如果依照程序列表 4-1,并如图 4-1 所示。在 Exemplar Logic LeonardoSpectrum 的输入文件菜单中使用二进制计数器选项来构造一个 Verilog 语言描述的计数器的话,则结果将得到一个行波计数器。

程序列表 4-1　简单计数器的 Verilog 设计代码

```
module ripplel (count _ out, clk, reset);
    input          clk, reset;
    output         count _ out;
    reg      [3:0] count _ out;

    always @ (posedge clk or posedge reset)
    if (reset)
        Count _ out < = 0;
    else
        Count _ out < = count _ out + 1;
endmodule
```

图 4-1　计数器类型选择

在使用行波计数器时会遇到如下问题,即由于在同一时刻多个输出发生变化,因此在使用组合逻辑对输出状态进行译码时,会导致尖峰脉冲信号。使用格雷码或约翰逊计数器可以避免这个问题的产生。

4.2　约翰逊计数器

约翰逊计数器是一种移位计数器。由于一个移位计数器可以使用较少的组合逻辑器件实现计数逻辑功能,因此它可以做到高速运行(其运行速度仅受触发器状态切换速度和简单计数逻辑的传播时延所限制)。约翰逊计数器采用的是把最高位触发器的输出的非,反馈送到最低位触发器的输入端。如同格雷码计数器一样,约翰逊计数器在每个时钟下只有一个输出发生变化。这使得用组合逻辑对其输出进行译码时,不会在组合逻辑输出端产生尖峰脉冲。约翰逊计数器的缺点是:它需要更多的寄存器去存储计数变量(可以表示 2n 种循环状态,这里 n 为寄存器的个数),另外它缺乏错误修正功能。如果一个错误的计数模式被加载,它将重复循环直到寄存器被重新初始化(如果初始化可以产生的话)。约翰逊计数器的示意图如图 4-2 所示,相应的 Verilog 代码见程序列表 4-2,计数序列见程序列表 4-3。

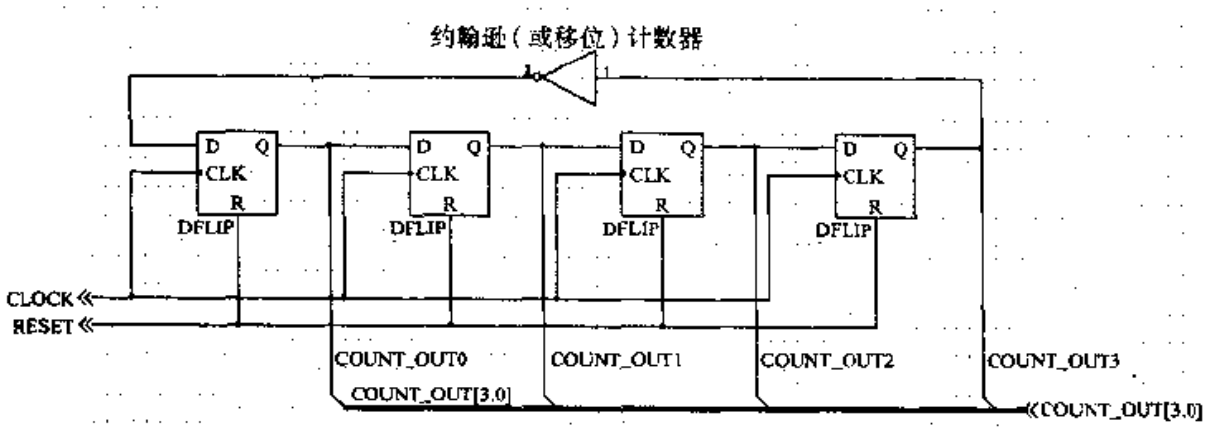

图 4-2　约翰逊计数器电路示意图

程序列表 4-2　约翰逊计数器的 Verilog 代码

```
module   johnson1 (clock, reset, count _ out);
    input              clock, reset;
    output             count _ out;
    reg        [3:0]   count _ out;
```

```
    always @ (clock or reset)
        if (reset)
            count_out <= 0;
        else begin
            count_out[3:1] <= count_out[2:0];
            count_out[0] <= ~count_out[3];
        end
endmodule
```

程序列表 4-3 约翰逊计数器输出序列

```
0000
1000
1100
1110
1111
0111
0011
0001
0000
Repeat...
```

细心的设计者会注意到在计数循环中并非所有的状态都会被用到。在这里，8个计数状态没有被使用。不难看出，约翰逊计数器的设计没有有效地利用寄存器的已有状态。这也许并不重要。然而，如果由于干扰噪声引入一个无效的计数状态，则其自身无法恢复到有效循环中去①。这种无法恢复的无效状态的存在将给设计者的设计带来麻烦。但如程序列表 4-4 所示，增加一些逻辑后，就可对这些无效的状态进行检测和恢复。这些逻辑的加入虽使得计数器的结构更加复杂，但这有助于构造一个输出译码中无尖峰脉冲的、具有很强抗干扰能力的计数器。

程序列表 4-4 具有错误恢复功能的约翰逊计数器

```
module johnson2 (clock, reset, count_out);
```

① 即不能自启动。——译者注

```
input          clock, reset;
output         count_out;
reg     [3:0]  count_out;

always  @  (posedge clock or posedge reset)
    if (reset)
        count_out <= 0;
    // 加入错误恢复。
    else if (count_out == 4'h2)  count_out <= 0;
    else if (count_out == 4'h4)  count_out <= 0;
    else if (count_out == 4'h5)  count_out <= 0;
    else if (count_out == 4'h6)  count_out <= 0;
    else if (count_out == 4'h9)  count_out <= 0;
    else if (count_out == 4'ha)  count_out <= 0;
    else if (count_out == 4'hb)  count_out <= 0;
    else if (count_out == 4'hd)  count_out <= 0;
    else begin
        count_out[3:1] <= count_out[2:0];
        count_out[0] <= ~count_out[3];
    end
endmodule
```

另外一种可以采用的错误纠正的方法是:当一个错误发生时,由外部的装置(比如一个微控制器)对计数器进行初始化。通常,如果硬件设计人员设计的逻辑器件不具备通过软件进行写入(或从已写入的器件中读出)操作的话,软件设计者将会感到十分不便。可读取和可写入的能力增强了硬件的可测试性,这通常是件好事。但同时,由于增加逻辑,使得设计规模增加并降低了运行速度,这又是这种设计的缺点。

4.3　线性反馈移位寄存器

一种十分有趣的计数器是线性反馈移位寄存器或者简称为 LFSR 计数器。它和约翰逊计数器十分相似,不同之处在于其最后一级不是直接反馈回第一级,并且它的部分抽头可以被循环使用。这种计数器的下一状态的逻辑是非常简单的(一些异或门或者同或门)。在逻辑为最大长度情况下(通过抽头的选择来给出最大的

计数),一个小规模的寄存器可以产生 2^n-1 个计数状态(二进制计数器的计数长度为 2^n)。一个最大长度 LSFR 计数器的状态序列中,反馈为异或时的全"0"状态,及反馈为同或时的全"1"状态都是不可恢复的状态。一个 LFSR 计数器因为其反馈逻辑很简单,所以它的执行速度比二进制计数器快。对于计数值是任意的情况(LFSR 计数序列是伪随机序列),采用 LFSR 计数器是个较好的选择。

下面用一个实例来说明这种计数器是如何工作的。一个最大长度为 4 比特位的 LFSR 计数器能够使用抽头[3,0](最大长度也可以用其他抽头来实现)。这些抽头是被反馈的寄存器输出端。图 4-3 中包含 4 个触发器和一个同或门。

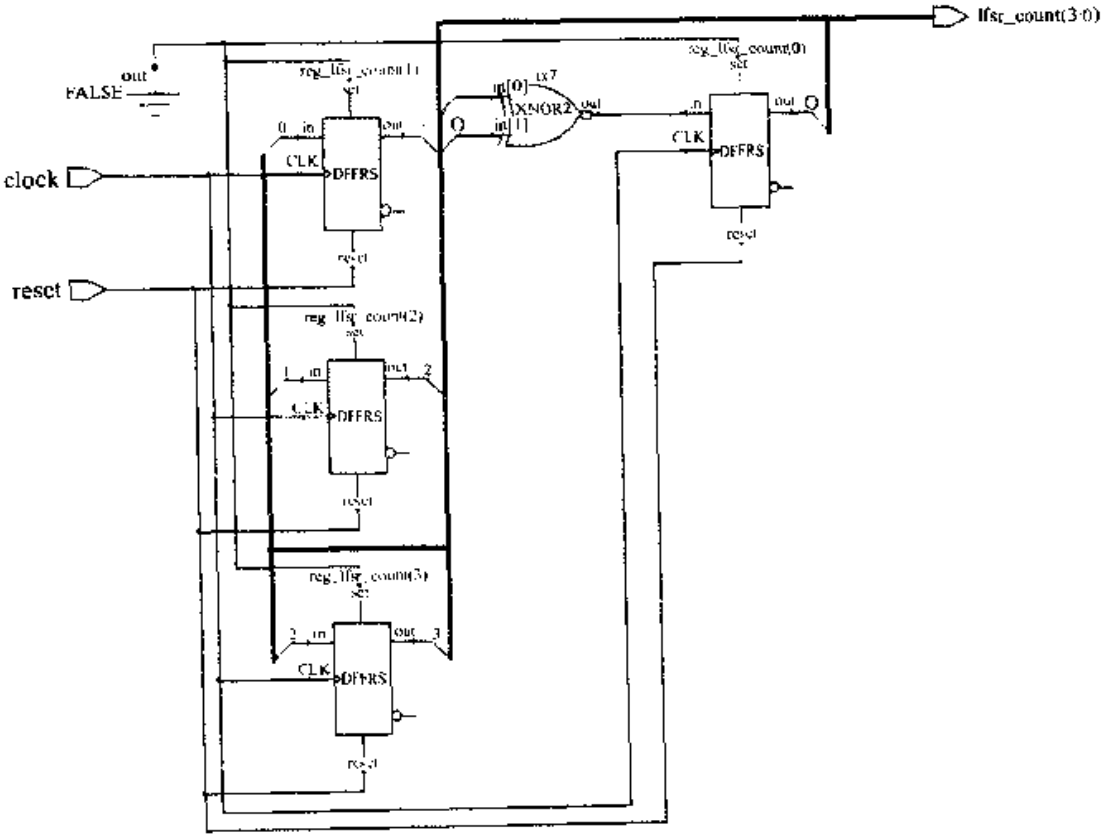

图 4-3 4 位 LFSR 计数器电路示意图

这种形式有时被称作"多对一";请注意抽头是如何从多个输出引出的。进行异或后又是如何反馈回输入端的。还有一种类型,称作"一对多",它是指所有的反馈项预先被组合后再进行反馈。

程序列表 4-5 所描述的计数器是一个同或形式。作者通常采用这种形式,因为其无效状态是全"1"状态;作者更愿意在上电时对所有的寄存器进行复位(而不是像采用异或形式时那样,对部分或全部寄存器进行预置)。程序列表 4-5 描述了 Verilog 语言设计的一个实例,程序列表 4-6 给出了对其进行测试的一段程序。

程序列表 4-5　4 位 LFSR 计数器 Verilog 描述

```
module  lfsr4 (clock,reset,lfsr _ count);
    input              clock, reset;
    output             lfsr _ count;
    reg        [3:0]   lfsr _ count;

    always @ (posedge clock or posedge reset)
        if (reset)
            lfsr _ count < = 0;
        else
        begin
            lfsr _ count [3:1]   < =   lfsr _ count [2:0];
            lfsr _ count[0]      < =   lfsr _ count [3] ~ ^ lfsr _ count [0];
        end
endmodule
```

程序列表 4-6　4 位 LFSR 计数器的简单测试程序

```
module  lfsr4 _ tf (clock, reset, lfsr _ count);
   'timescale              1ns / 1ns
   output                  clock, reset;
   reg                     clock, reset;
   input          [3:0]    lfsr _ count;
   parameter      clk _ period          = 20;

   lfsr4 u1 (clock, reset, lfsr _ count);

   always begin
            # (clk _ period / 2) clock =  ~ clock;
        end
   initial begin
              clock     =   0;
              reset     =   1;  //声明系统复位。
            # 75 reset   =   0;
```

```
    end
endmodule
```

图 4-4 给出了 4 位 LFSR 计数器的计数序列。

```
// 二进制          十六进制
// 0000            0
// 0001            1
// 0010            2
// 0101            5
// 1010            a
// 0100            4
// 1001            9
// 0011            3
// 0110            6
// 1101            d
// 1011            b
// 0111            7
// 1110            e
// 1100            c
// 1000            8

// 0000            0          重复序列
```

图 4-4 4 位 LFSR 计数器计数序列

看上去是不是有些杂乱？其实这正是 LFSR 计数器引人注目的一面。序列值之间有着松散的关联性，或者是伪随机的。这有助于减少时钟的谐波噪声。例如，在二进制计数器中，最低位随着每个时钟的到来而发生跳变，这将导致和系统时钟高度相关的噪声的产生，并且造成系统时钟谐波分量的能量增加，而这种谐波能量乃是产生系统噪声的主要来源。对于 LFSR 计数器，由于其计数值以一个更随机的方式变化，使其具有更低的峰值能量成分，因此这种计数器会产生频率范围更宽的噪声。

表 4-1 列出了最大长度 LFSR 计数器的抽头选择方法。对于某些计数器长度采用其他的抽头选择方法也是可行的。

表 4-1　最大长度 LFSR 计数器抽头

位数	循环长度	抽头
2*	3	[1,0]
3*	7	[2,0]
4	15	[3,0]
5*	31	[4,1]
6	63	[5,0]
7*	127	[6,0]
8	255	[7,3,2,1]
9	511	[8,3]
10	1023	[9,2]
11	2047	[10,1]
12	4095	[11,5,3,0]
13*	8191	[12,3,2,0]
14	16383	[13,4,2,0]
15	32767	[14,0]
16	65535	[15,4,2,1]
17*	131071	[16,2]
18	262143	[17,6]
19*	524287	[18,4,1,0]
20	1048575	[19,2]
21	2097151	[20,1]
22	4194303	[21,0]
23	8388607	[22,4]
24	16777215	[23,3,2,0]
25	33554431	[24,2]
26	67108863	[25,5,1,0]
27	134217727	[26,4,1,0]
28	268435455	[27,2]
29	536870911	[28,1]
30	1073741823	[29,5,3,0]
31*	2147483647	[30,2]
32	4294967295	[31,6,5,1]

* 表示长度为质数的序列。

注：①序列 2,3,5,7,13,17,19,31 的长度为质数。

②此表引自 Clive Maxfield 所著的 *Designus Maximus Unleashed*，已被 Butterworth-Heinemann 于 1998 年授权使用。

由表 4-1 所示,可以用 31 个寄存器和一个异或门构造一个 31 位的计数器。我们可以想像一下用逐级进位逻辑构造一个 31 位二进制计数器的情形。

LFSR 计数器的一个应用实例是构造一个实现除以 N 的简单逻辑电路。在设计中,末位的计数值也被用来作为一个输入,用于比较之用。程序列表 4-7 举例说明了一个 8 位除以 N 计数器的实现,程序列表 4-8 给出了一个测试程序,程序列表 4-9 是伪随机计数序列的输出结果,图 4-5 给出了它的计数翻转的波形图。

程序列表 4-7　8 位除 N　LFSR 计数器的 Verilog 描述

```
// 8 位除 N  LFSR 计数器。
module  fsr8 (clock, reset, lfsr _ count, terminal _ cnt, rollover);
    input                clock, reset;
    input     [7:0]      terminal _ cnt;
    output               lfsr _ count;
    reg       [7:0]      lfsr _ count;
    output               rollover;
    reg                  rollover;

    always  @  (posedge clock or posedge reset)
        if (reset)
        begin
                lfsr _ count      < =     0;
                rollover          < =     0;
        end
        else
            if (lfsr _ count = = terminal _ cnt) //末位计数检测。
            begin
                rollerover        < =     1;
                lfsr _ count      < =     0;
            end
            else begin
                    rollover          < =  0;
                    lfsr _ count[7:1]  < =  lfsr _ count [6:0];
                    lfsr _ count [0]   < =  lfsr _ count[7] ~^(lfsr _ count[3]
                                            ~^(lfsr _ count [2] ~^lfsr _ count[1]));
```

```
        end
endmodule
```

程序列表 4-8　8 位除 N　LFSR 计数器测试程序的 Verilog 描述

```
// 8 位除 N　LFSR 计数器测试程序。
module  lfsr8_tf (clock, reset, lfsr_count, terminal_cnt);
'timescale  1ns / 1ns
output                  clock, reset;
reg                     clock, reset;
input       [7:0]       lfsr_count;
output      [7:0]       terminal_cnt;
reg                     terminal_cnt;
wire                    rollover;

parameter   clk_period      =   20;

lfsr8 u1 (clock, reset, lfsr_count, terminal_cnt, rollover);

always
    begin
        # (clk_period / 2) clock = ~clock;
    end

    initial
    begin
        clock           =   0;
        reset           =   1;      // 声明系统复位。
        terminal_cnt    =   8       'd66; // 测试赋值。
        #75 reset       =   0;
    end
endmodule
```

程序列表 4-9 8 位除 N LFSR 计数器计数序列

```
lfsr_count = 00, rollover = 0  lfsr_count = 00, rollover = 0
lfsr_count = 00, rollover = 0  lfsr_count = 00, rollover = 0
lfsr_count = 01, rollover = 0  lfsr_count = 03, rollover = 0
lfsr_count = 06, rollover = 0  lfsr_count = 0d, rollover = 0
lfsr_count = 1b, rollover = 0  lfsr_count = 37, rollover = 0
lfsr_count = 6f, rollover = 0  lfsr_count = de, rollover = 0
lfsr_count = bd, rollover = 0  lfsr_count = 7a, rollover = 0
lfsr_count = f5, rollover = 0  lfsr_count = eb, rollover = 0
lfsr_count = d6, rollover = 0  lfsr_count = ac, rollover = 0
lfsr_count = 58, rollover = 0  lfsr_count = b0, rollover = 0
lfsr_count = 60, rollover = 0  lfsr_count = c1, rollover = 0
lfsr_count = 82, rollover = 0  lfsr_count = 05, rollover = 0
lfsr_count = 0a, rollover = 0  lfsr_count = 15, rollover = 0
lfsr_count = 2a, rollover = 0  lfsr_count = 55, rollover = 0
lfsr_count = aa, rollover = 0  lfsr_count = 54, rollover = 0
lfsr_count = a8, rollover = 0  lfsr_count = 51, rollover = 0
lfsr_count = a3, rollover = 0  lfsr_count = 47, rollover = 0
lfsr_count = 8f, rollover = 0  lfsr_count = 1f, rollover = 0
lfsr_count = 3e, rollover = 0  lfsr_count = 7c, rollover = 0
lfsr_count = f9, rollover = 0  lfsr_count = f3, rollover = 0
lfsr_count = e7, rollover = 0  lfsr_count = ce, rollover = 0
lfsr_count = 9d, rollover = 0  lfsr_count = 3a, rollover = 0
lfsr_count = 75, rollover = 0  lfsr_count = ea, rollover = 0
lfsr_count = d4, rollover = 0  lfsr_count = a9, rollover = 0
lfsr_count = 53, rollover = 0  lfsr_count = a6, rollover = 0
lfsr_count = 4c, rollover = 0  lfsr_count = 99, rollover = 0
lfsr_count = 33, rollover = 0  lfsr_count = 66, rollover = 0
lfsr_count = cd, rollover = 0  lfsr_count = 9a, rollover = 0
lfsr_count = 34, rollover = 0  lfsr count = 68, rollover = 0
lfsr_count = d0, rollover = 0  lfsr_count = a0, rollover = 0
lfsr_count = 40, rollover = 0  lfsr_count = 81, rollover = 0
lfsr_count = 02, rollover = 0  lfsr_count = 04, rollover = 0
lfsr_count = 08, rollover = 0  lfsr_count = 10, rollover = 0
```

lfsr_count = 21, rollover = 0　lfsr_count = 43, rollover = 0
lfsr_count = 86, rollover = 0　lfsr_count = 0c, rollover = 0
lfsr_count = 19, rollover = 0　lfsr_count = 32, rollover = 0
lfsr_count = 64, rollover = 0　lfsr_count = c8, rollover = 0
lfsr_count = 91, rollover = 0　lfsr_count = 22, rollover = 0
lfsr_count = 44, rollover = 0　lfsr_count = 88, rollover = 0
lfsr_count = 11, rollover = 0　lfsr_count = 23, rollover = 0
lfsr_count = 46, rollover = 0　lfsr_count = 8d, rollover = 0
lfsr_count = 1a, rollover = 0　lfsr_count = 35, rollover = 0
lfsr_count = 6a, rollover = 0　lfsr_count = d5, rollover = 0
lfsr_count = ab, rollover = 0　lfsr_count = 56, rollover = 0
lfsr_count = ad, rollover = 0　lfsr_count = 5a, rollover = 0
lfsr_count = b5, rollover = 0　lfsr_count = 6b, rollover = 0
lfsr_count = d7, rollover = 0　lfsr_count = ae, rollover = 0
lfsr_count = 5d, rollover = 0　lfsr_count = bb, rollover = 0
lfsr_count = 76, rollover = 0　lfsr_count = ed, rollover = 0
lfsr_count = da, rollover = 0　lfsr_count = b4, rollover = 0
lfsr_count = 69, rollover = 0　lfsr_count = d2, rollover = 0
lfsr_count = a5, rollover = 0　lfsr_count = 4b, rollover = 0
lfsr_count = 97, rollover = 0　lfsr_count = 2e, rollover = 0
lfsr_count = 5c, rollover = 0　lfsr_count = b9, rollover = 0
lfsr_count = 73, rollover = 0　lfsr_count = e6, rollover = 0
lfsr_count = cc, rollover = 0　lfsr_count = 98, rollover = 0
lfsr_count = 31, rollover = 0　lfsr_count = 63, rollover = 0
lfsr_count = c6, rollover = 0　lfsr_count = 8c, rollover = 0
lfsr_count = 18, rollover = 0　lfsr_count = 30, rollover = 0
lfsr_count = 61, rollover = 0　lfsr_count = c3, rollover = 0
lfsr_count = 87, rollover = 0　lfsr_count = 0e, rollover = 0
lfsr_count = 1c, rollover = 0　lfsr_count = 39, rollover = 0
lfsr_count = 72, rollover = 0　lfsr_count = e4, rollover = 0
lfsr_count = c9, rollover = 0　lfsr_count = 93, rollover = 0
lfsr_count = 27, rollover = 0　lfsr_count = 4f, rollover = 0
lfsr_count = 9e, rollover = 0　lfsr_count = 3d, rollover = 0
lfsr_count = 7b, rollover = 0　lfsr_count = f7, rollover = 0
lfsr_count = ee, rollover = 0　lfsr_count = dd, rollover = 0

lfsr _ count = ba, rollover = 0 lfsr _ count = 74, rollover = 0
lfsr _ count = e8, rollover = 0 lfsr _ count = d1, rollover = 0
lfsr _ count = a2, rollover = 0 lfsr _ count = 45, rollover = 0
lfsr _ count = 8a, rollover = 0 lfsr _ count = 14, rollover = 0
lfsr _ count = 28, rollover = 0 lfsr _ count = 50, rollover = 0
lfsr _ count = a1, rollover = 0 lfsr _ count = 42, rollover = 0
lfsr _ count = 00, rollover = 0 lfsr _ count = 01, rollover = 0

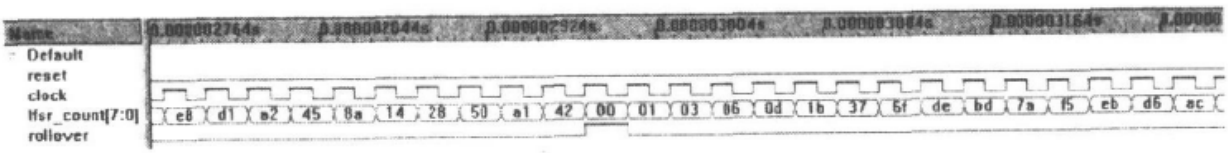

图 4-5 8 位除 N LFSR 翻转计数仿真

如程序列表 4-10 所示的“一对多”的电路模式将异或逻辑分解为多个双输入门,并使它们分布于寄存器阵列中。注意:相同的抽头被采用,仅仅形式不同而已。简言之,4 比特计数器的抽头[3,0]的意思是:图中寄存器 0 和寄存器 3 的输出进行异或(或者同或),然后将输出结果作为寄存器 1 的输入。最后一个寄存器的输出再回到寄存器 0。此结构仍可以产生最大长度序列,但计数序列(以及对于给定计数的最终计数值)是不同的。图 4-6 给出了和程序列表 4-10 对应的示意图。其输出波形如图 4-7 所示。

程序列表 4-10 4 位 LFSR 计数器的“一对多”形式的代码描述

```
module   lfsr4v2 (clock, reset, lfsr _ count);
        input                   clock, reset;
        output                  lfsr _ count;
        reg          [3:0]      lfsr _ count;

        always @ (posedge clock or posedge reset)
            if  (reset)
            lfsr _ count      < =      0;
            else      begin
                        lfsr _ count[0]    < =    lfsr _ count [3];
                        lfsr _ count[1]    < =    lfsr _ count [3] ~ ^ lfsr _
                        count [0];
```

```
        lfsr_count[3:2]  <=  lfsr_count[2:1];
    end
endmodule
```

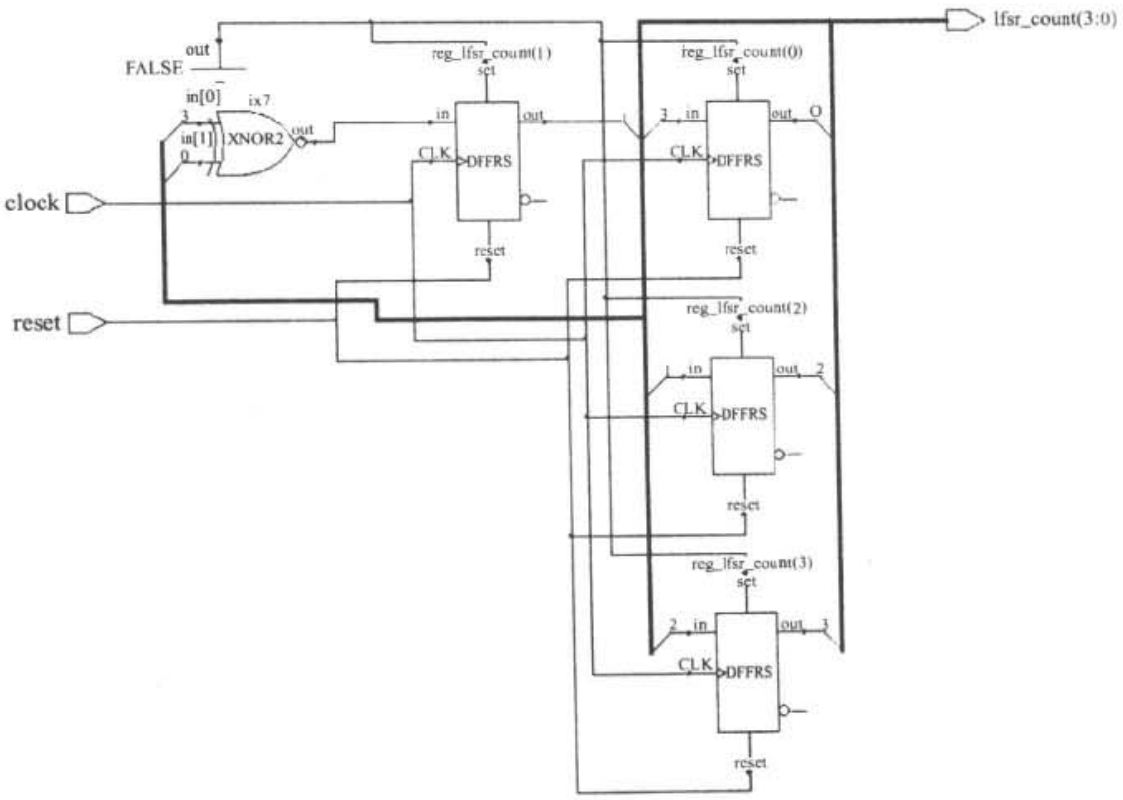

图 4-6　4 位 LFSR 计数器的“一对多”示意图

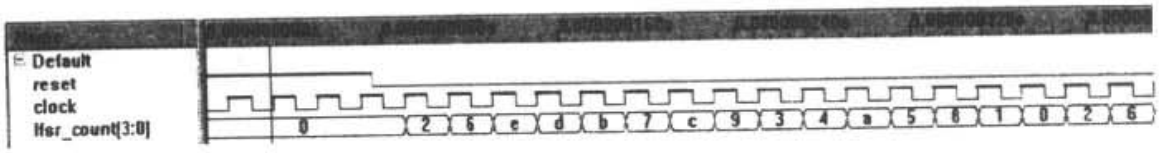

图 4-7　4 位 LFSR 计数器“一对多”模式输出波形

欲对 LFSR 计数器有更深入的了解,请参看 Max Maxfield 的 *Designus Maximus Unleashed*(有关这本书的细节在参考书目中可以找到)。

4.4　循环冗余校验

与 LFSR 计数器类似的逻辑可以用来构造循环冗余校验,或简称 CRC。CRC 用来测试一个数据包中是否有错误发生。一个普通的校验仅仅是把数据字节或把

数据字相加,并丢弃所有超过预设精度的进位。例如,一个 8 比特校验将使用模-256 的加法,并丢弃所有超过数值 255 的进位(255 用十六进制表示时为 FF)。

假设一个数据包含有下列 8 个字节:

十六进制数据

99
D0
AA
01
09
83
AF
BE

把以上所有数字相加,结果为 40D(十六进制),丢弃多于 8 位的进位,保留低 8 位的值,可得校验和为 0D(十六进制)。接收方的逻辑做相同的加法运算,计算接收到的数据的校验和是否也是 0D。这种方法对检验数据是否被正确接收提供了一种途径。更保险的办法是把发送的校验和的位数增加到 16 位,这将给一个 10 字节的数据包产生值为 40D 的校验和。此时,对于可能出现的多种错误,产生一个错误的概率是 1/65536,而不是原来的 1/256。如果一个错误使前一个字节产生大于实际值的数,而后续的错误又使后一个字节产生小于实际值的数,且前后相差的数又正好相同,则校验和将是匹配的,一个有错误的数据包将被误判为正确的数据包。引入更多的随机数字序列可以更好地执行错误检测。

CRC 的设计思想是用除法代替加法。数据包被看做一个多位二进制数。用这个二进制数除以一个选定的多项式,余数就是所要的校验和。余数的序列要比和的序列更具随机性。本文将不讨论有关 CRC 大量的数学公式推导,仅仅讨论采用多项式(这里借位被丢弃了)进行 CRC 除法的逻辑实现的问题,读者会发现这个逻辑和 LFSR 计数器的逻辑实现有很多相似之处。输入数据包由 N 位附加的零构成,并被串行移位输出(这里 N 表示 CRC 的长度)。在数据传送时,进行 CRC 的计算,并用其结果替换原来位置上的零,得到的就是所传送的数据包。

在接收端,对接收的数据包(包括 CRC 位)进行相同的 CRC 运算,若余数为零,则表示没有错误被检测到。这里用一个简单的例子对此进行说明。Xilinx 公司采用 16 位 CRC 对用于配置 FPGA 的串行数据进行校验。其逻辑示意图见图 4-8。Xilinx 采用的是异或逻辑,"一对多"的配置结构,以及[15,14,1,0]反馈抽头。

可以注意到:此逻辑和附加一个数据输入端的 LFSR 计数器的逻辑是十分相似的,此数据输入端被用作调制源。程序列表 4-11 是 CRC-16 的逻辑实现。

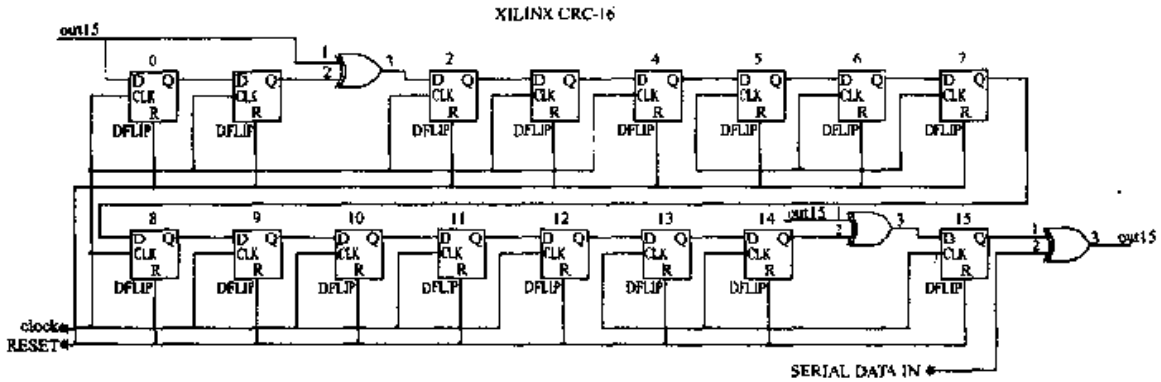

图 4-8　CRC-16 示意图

程序列表 4-11　CRC-16 逻辑的 Verilog 描述

```
module  crc16 (clock, reset, serial _ data _ in, serial _ data _ out );
input           clock, reset, serial _ data _ in;
output          serial _ data _ out;
reg     [15:0]  crc _ output;
assign          serial _ data _ out = serial _ data _ in ^ crc _ output[15];
   always  @  (posedge clock or posedge reset)
      if (reset) crc _ output < = 0;
      else begin
      crc _ output[14:3]     < =   crc _ output[13:2];
      crc _ output[1]        < =   crc _ output[0];
      crc _ output[2]        < =   crc _ output[1] ^ serial _ data _ out;
      crc _ output[15]       < =   crc _ output[14] ^ serial _ data _ out;
      crc _output[0]               crc _ output[15] ^ serial _ data _ out;
          end
endmodule
```

4.5　只读存储器(ROM)

ROM 的意思是只读存储器。当 FPGA 进行配置时,这个存储器被初始化,在配置之后将无法再进行擦写(可以进行擦写的存储器称作 RAM)。作为一个实例,我们采用 ROM 来实现 4 位 LFSR 计数器(这本没有什么实际意义,但为了解释和说

明,我们仍采用这个实例),参见程序列表 4-12 及图 4-9。

程序列表 4-12 LFSR 计数器的 ROM 描述

```
module  lfsr_rom (binary_in, lfsr_out, clk, reset);
input       [3:0]    binary_in;
input                clk, reset;
output      [3:0]    lfsr_out;
reg         [3:0]    lfsr_out;

always  @ (posedge clock or posedge reset)
    begin
        if (reset)
            lfsr_out <= 4'b0000;
        else  case (binary_in)
            4'b0000:    lfsr_out    <=    4'b0000;
            4'b0001:    lfsr_out    <=    4'b0001;
            4'b0010:    lfsr_out    <=    4'b0010;
            4'b0011:    lfsr_out    <=    4'b0101;
            4'b0100:    lfsr_out    <=    4'b1010;
            4'b0101:    lfsr_out    <=    4'b0100;
            4'b0110:    lfsr_out    <=    4'b1001;
            4'b0111:    lfsr_out    <=    4'b0011;
            4'b1000:    lfsr_out    <=    4'b0110;
            4'b1001:    lfsr_out    <=    4'b1101;
            4'b1010:    lfsr_out    <=    4'b1011;
            4'b1011:    lfsr_out    <=    4'b0111;
            4'b1100:    lfsr_out    <=    4'b1110;
            4'b1101:    lfsr_out    <=    4'b1100;
            4'b1110:    lfsr_out    <=    4'b1000;
            4'b1111:    lfsr_out    <=    4'b0000; // 未使用的
            组合。
        // 默认:lfsr_out <= 4'b0; 这是没必要的,因为上面已经包含
        // 了所有的组合。
        endcase
```

```
end
endmodule
```

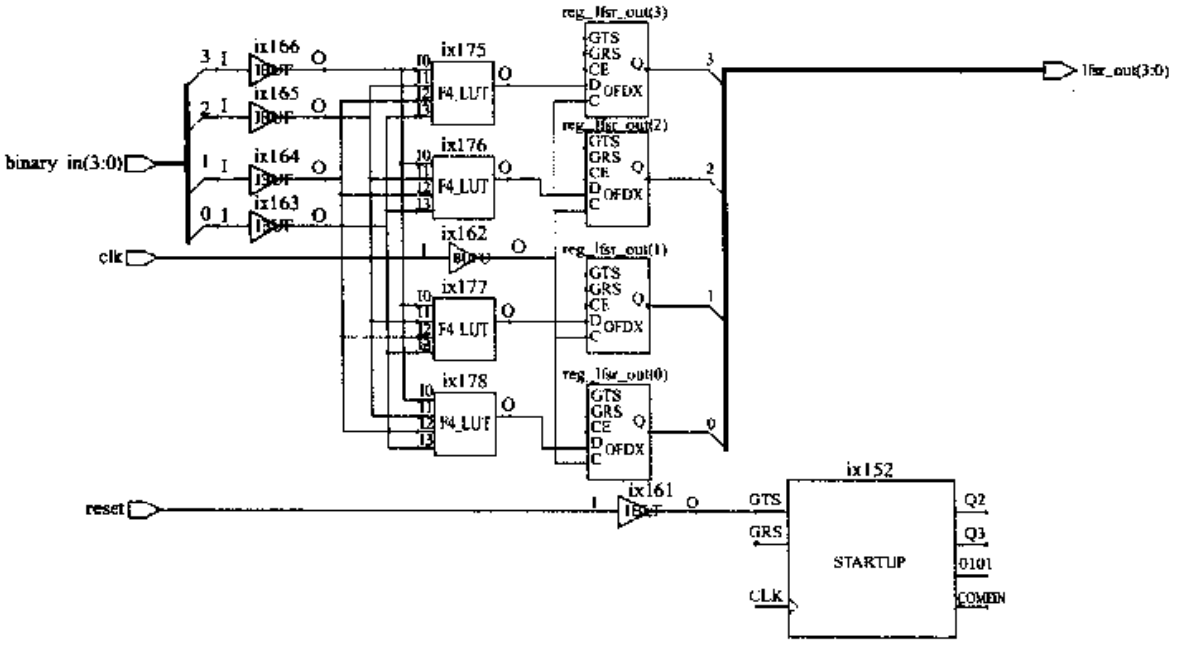

图 4-9　LFSR 计数器的 ROM 方式电路示意图

由于 Xilinx 可以有效地实现四输入逻辑的多种组合形式,所以此功能是高效的(但不如 LFSR 算法效率高。因为在这里,ROM 的功能是使用两组可配置逻辑块(CLB)来实现的,而早先的设计中仅采用一组可配置逻辑块)。然而,逻辑的复杂程度是按输入个数的平方增加的,因此,由 CLB 构成的 ROM 变得非常复杂。类似的 ROM 设计又被称作查找表(LUT)。

Xilinx 公司的许多 CLB 都具有一种 RAM 模式,在此模式下,一个 CLB 被用作一个 16 × 1 的存储元件。这是一种构建 RAM 或 ROM 模块的非常有效的方法。在第八章中将对 LogiBLOX 工具(Xilinx 公司的产品)和存储器模块的使用进行介绍。值得注意的是,许多专用集成电路(ASIC)技术都不具有 RAM 功能。在进行 ASIC 转换的过程中,ROM/RAM 单元将被随机逻辑替代,这将导致一个规模巨大的 ASIC 设计。

4.6　随机存储器(RAM)

RAM,即随机存储器,是一个可寻址的 M × N 容量的存储单元阵列,其中 N 表示数据单元的长度(即位数,如 × 4, × 8, × 16, × 32),M 表示 N 位数据单元的个数。

同样可以使用 CLB 来构造 RAM,下面我们通过构造一个简单的 16 × 1 大小的存储块(一个非常小的 RAM 块)来了解一下它的结构。在此设计中假定内部的三态驱动器是可控的。

注意:对于现场可编程门阵列(FPGA)中的 CLB RAM,一个可取之处在于它具有在复位后对 RAM 寄存器单元进行初始化的能力。

4.6.1 16×1 RAM 块

因为有 16 个存储单元,因此需要配置 4 根地址线,如程序列表 4-13 所示。

程序列表 4-13 用 CLB 构造 16 × 1RAM 的 Verilog 描述

```
module  ram16×1 (ram_data, ram_addr, ram_rwn, clock, reset);
inout           ram_data;
input   [3:0]   ram_addr;
input           ram_rwn, clock, reset;  //低电平有效时写入。
reg     [15:0]  ram_data-reg;
wire            ram_data_in;

assign  ram_data = ram_rwn ? ram_data_reg[ram_data_reg[ram_addr]] : 1'bz;
assign  ram_data_in     =     ram_data;

always @ ( posedge clock or posedge reset )
    if (reset) ram_data_reg <= 0;
    else case ( {ram_addr, ram_rwn} )
    {4'h0, 1'b0} : ram_data_reg[0]  <=  ram_data_in;
    {4'h1, 1'b0} : ram_data_reg[1]  <=  ram_data_in;
    {4'h2, 1'b0} : ram_data_reg[2]  <=  ram_data_in;
    {4'h3, 1'b0} : ram_data_reg[3]  <=  ram_data_in;
    {4'h4, 1'b0} : ram_data_reg[4]  <=  ram_data_in;
    {4'h5, 1'b0} : ram_data_reg[5]  <=  ram_data_in;
    {4'h6, 1'b0} : ram_data_reg[6]  <=  ram_data_in;
    {4'h7, 1'b0} : ram_data_reg[7]  <=  ram_data_in;
    {4'h8, 1'b0} : ram_data_reg[8]  <=  ram_data_in;
    {4'h9, 1'b0} : ram_data_reg[9]  <=  ram_data_in;
```

```
        {4'ha, 1'b0} : ram_data_reg[10]   <=   ram_data_in;
        {4'hb, 1'b0} : ram_data_reg[11]   <=   ram_data_in;
        {4'hc, 1'b0} : ram_data_reg[12]   <=   ram_data_in;
        {4'hd, 1'b0} : ram_data_reg[13]   <=   ram_data_in;
        {4'he, 1'b0} : ram_data_reg[14]   <=   ram_data_in;
        {4'hf, 1'b0} : ram_data_reg[15]   <=   ram_data_in;
        default:       ram_data_reg       <=   ram_data_reg;
        endcase
    endmodule
```

图 4-10 给出了由程序列表 4-13 得出的逻辑综合的示意图。程序列表 4-14 总结了此设计所用资源的情况。

程序列表 4-14　用 CLB 构造 16×1RAM 的 Verilog 描述设计小结

```
Total accumulated area :
 Number of BUFG :                          1
 Number of CLB Flip Flops :               16
 Number of FG Function Generators :       29
 Number of H Function Generators :         3
 Number of IBUF :                          7
 Number of OBUFT :                         1
 Number of Packed CLBs :                  15
 Number of STARTUP :                       1
*************************************************
Device Utilization for 4010xlPQ100
Resource                  Used    Avail    Utilization
-------------------------------------------------
IOs                       8       77        10.39%
FG Function Generators    29      800        3.62%
H Function Generators     3       400        0.75%
CLB Flip Flops            16      800        2.00%
-------------------------------------------------
     Clock Frequency Report
     Clock          : Frequency
     clock          : 41.1 MHz
```

从上图我们可以看到,使用 FPGA 的 CLB 去实现 RAM 功能的效率是不高的。CLB 是被用来实现随机逻辑功能的。如果采用一个 RAM 存储单元来替代同样逻辑的话,它将占用整个 CLB。

采用 Verilog 语言可以很容易地描述一个 RAM 单元。然而,由于 Verilog 语言不支持二维阵列的描述,因此,需采用一维向量阵列来对 RAM 进行建模。

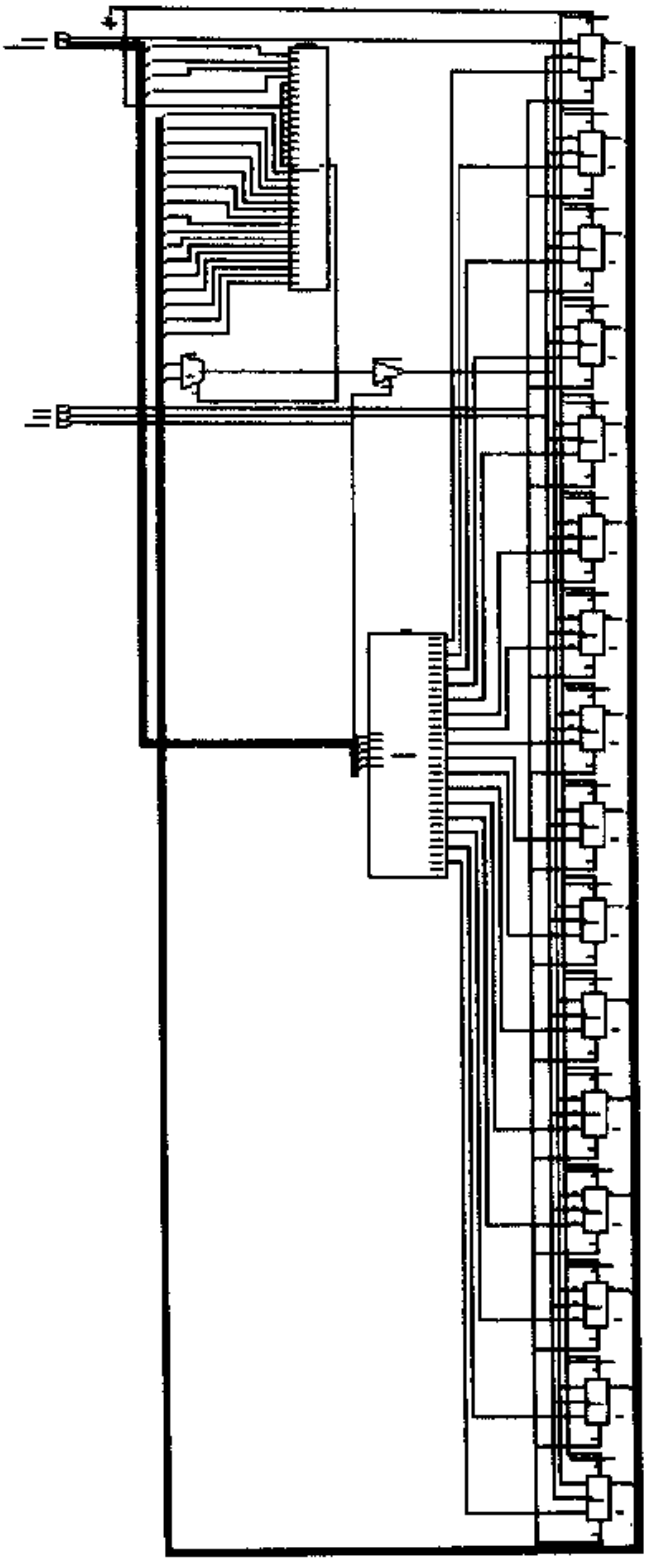

图 4-10 用 CLB 构建 16×1RAM 的 Verilog 电路示意图

程序列表 4-15 给出了一个 256 x 8 可综合的 RAM 模块的例子。

程序列表 4-15　RAM 的 Verilog 描述实例

```
module ram _ mod1(rwn, addr, data _ port);
input     rwn;
input     [7:0]   addr;
inout     [7:0]   data _ port;
reg       [7:0] ramdata [0:255];

assign data _ port = (rwn) ? ramdata[addr] : 8'hz;

always @ (rwn or addr)
    if ( ~ rwn) ramdata[addr] = data _ port ;
endmodule
```

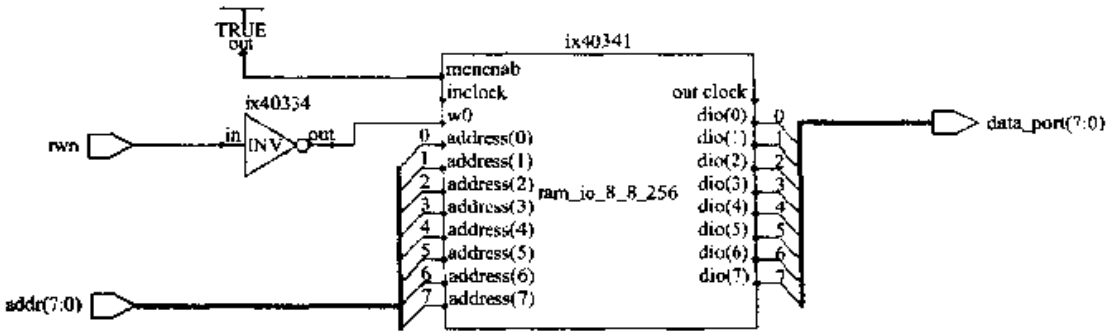

图 4-11　256 x 8RAM 的 Verilog 电路示意图

程序列表 4-15 所示的 RAM 描述是能工作的,但除非 FPGA 支持此类嵌入式的 RAM 块,否则,这种实现要占用大量的逻辑资源,并且其成本比选用任何一种你所能买到的静态随机存储器(SRAM)器件要高得多。对于小规模的 RAM 的实现(8 字节量级),采用 FPGA 的 CLB 实现的方法还是可取的,但对于大容量的 RAM 的实现,则需要考虑采用其他的解决方案。利用触发器去实现 RAM,其效率是非常低的。

从图 4-11 可以看到,利用 Exemplar Logic 的 LeonardoSpectrum 工具可由 Verilog 代码正确生成 RAM。Xilinx 的 4000XL 系列支持内嵌的 RAM 结构,参见图 4-12。此示意图和图 4-13 相比还算不上复杂。图 4-13 所示的设计是针对 XC3000 系列器件的,其体系结构中不具有分布式的 CLB RAM。它属于较早期的器件,其示意图有

39 页之多！而 4000XL 只需一页就够了。

为了存储输入/输出数据阵列，以及存储配置信息、表格和参数，设计者经常需

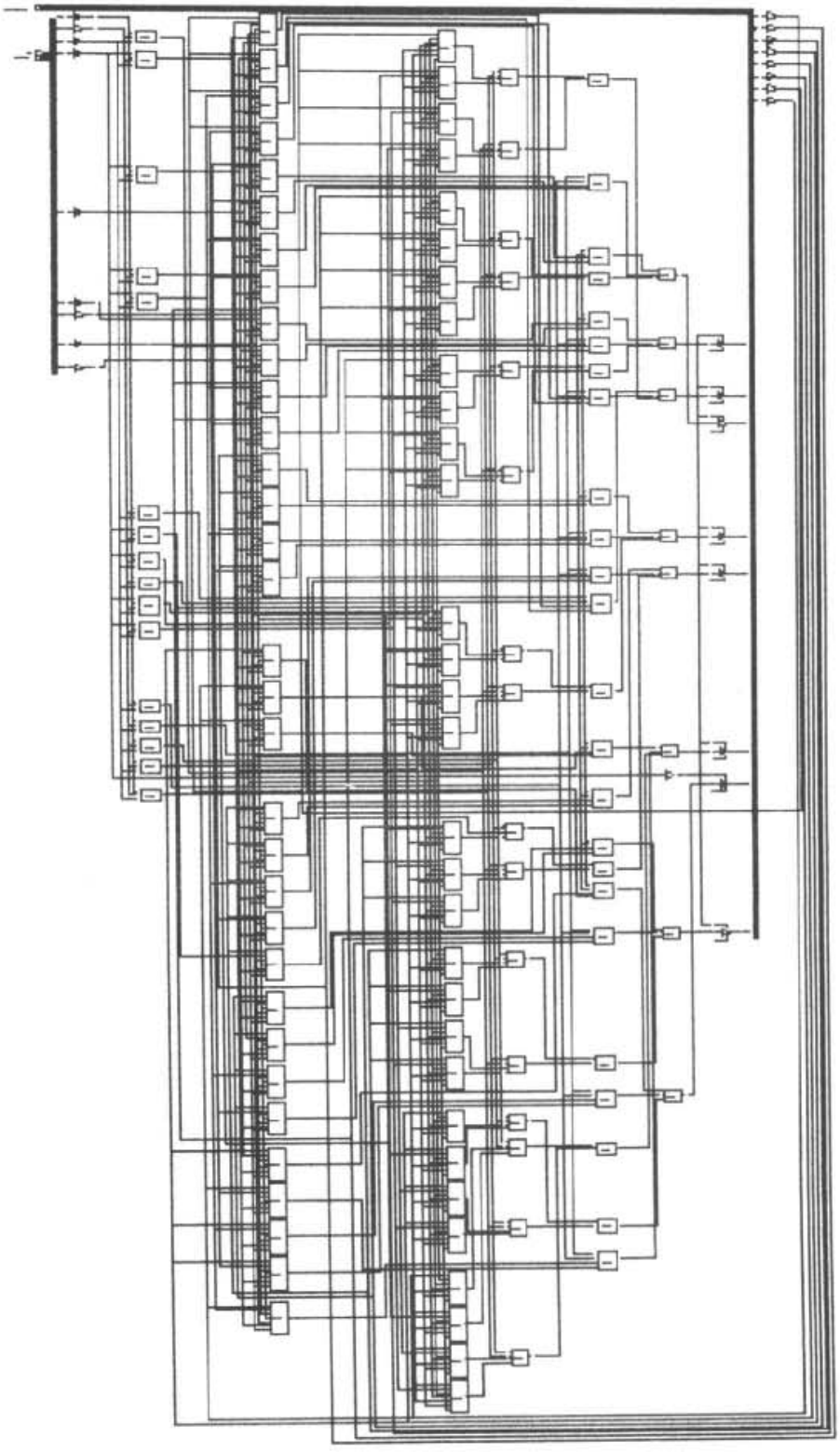

图 4-12　采用 4000XL 系列器件实现 256 × 8RAM(1/1 页)

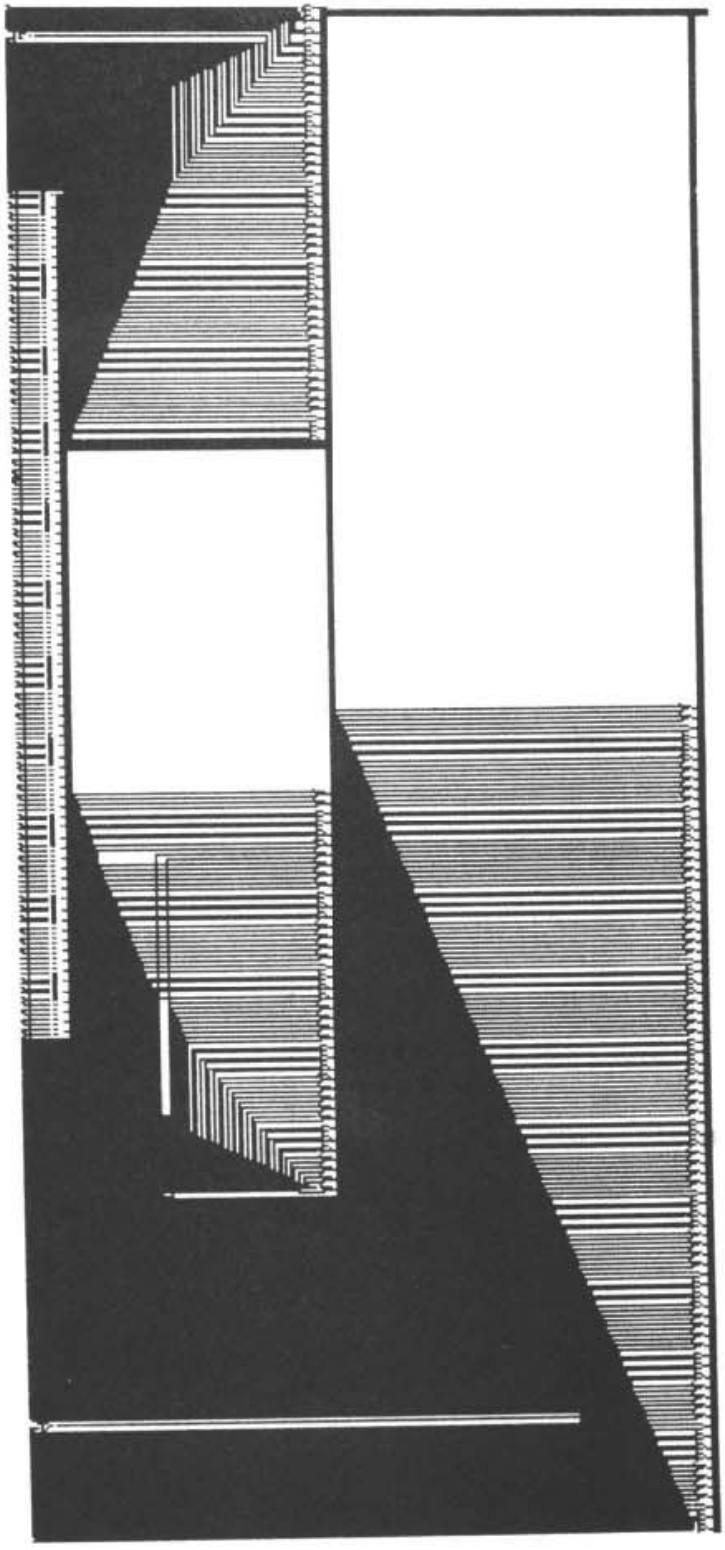

图 4-13　采用 XC3000 系列器件实现 256 × 8RAM(1/39 页)

要使用 RAM 块。许多流行的 FPGA/CPLD 体系结构中都包含有块结构的 RAM(典型的如 Altera 公司的产品)或可在器件设计过程中实现的分布式 RAM,这使得一个

CLB 可以被配置为一个查找表(LUT)或一个 RAM 单元(典型的如 Xilinx 公司的产品)。采用一个 CLB 作为一个 16×1 大小的 RAM 单元对设计者来说不失为一个好的方法,它速度快,还不会占用更多的 FPGA 资源。

当需要用到大量的 RAM 时又该如何处理呢?有两个解决方法。一是选取有足够多的具有内嵌结构 RAM 的 FPGA(选取时要注意适当留有一定的余量,比如,你需要用到 1K 的 RAM,则应至少选取具有 2K RAM 的器件);另一种方法是在设计中采用外接的 SRAM 器件。虽然现在的 FPGA 器件中都设置有 RAM 块或分布式的 RAM 存储单元,但采用 FPGA 去实现 RAM 比采用一般的 RAM 集成电路芯片的成本要高得多。另外值得考虑的是在向 ASIC 转换过程中可能由 RAM 所带来的一些问题。

4.6.2 内部 RAM 和外部 RAM 选取时的权衡策略

1. 内部 RAM 特性

速度 内部 RAM 组件不仅速度快(通常如此),而且可以避免驱动信号的开/关所造成的速度问题。

时序 由于使用内部 RAM 芯片,其时钟和数据的关系对于布局/布线工具来说是已知的。这减轻了时序分析的工作量。内部 FPGA 信号被调整使得寄存器保持时间为零。但外部 RAM 不一定能做到"零保持"。无论怎样,采用外部 RAM 时,必须考虑与器件 I/O 引脚有关的延时,并且也必须在设计中考虑由其引起的最小保持时间。

初始化 用 FPGA 实现 RAM 的另一个好处是可以在上电时对 RAM 内容进行初始化。此初始化可以写入全"0",或从一个文件中读出 RAM 值,然后写入 RAM 阵列。此种设计不需要配置其他辅助电路来完成 RAM 初始化的工作(比如,一个微处理器在上电时对每一个存储单元进行写操作)。

2. 内部 RAM 带来的问题

成本 内部 RAM 存储芯片比外部 RAM 存储芯片的成本高得多。不过,由于在电路板上要加入另外的器件并占用额外的 FPGA 引脚资源,因此会多多少少抵消采用外部 RAM 带来的成本优势。

4.6.3 RAM 使用实例

如何用一个实例来说明 RAM 模块呢?Xilinx 公司提供了一个称作 LogiBLOX 的工具,用来构造 RAM 模块。程序列表 4-16 给出了一个 LogiBLOX 模块的实例。有关 Xilinx LogiBLOX 模块的构造过程的更详细的介绍请参见第八章的相关内容。

程序列表 4-16　Xilinx 同步双端口 RAM

```
module x_ram1 (clk, ram_data, ram_a_addr, ram_b_addr, ram_a_data, ram_b_data, wr_strobe);
// 使用 Xilinx 公司的 LogiBLOX 构造的双端口 RAM。
input    [15:0]    ram_data;
input    [4:1]     ram_a_addr;
input    [4:1]     ram_b_addr;
input              wr_strobe;
input              clk;
output   [15:0]    ram_a_data;
output   [15:0]    ram_b_data;

//------------------------------------------------------------
// LogiBLOX DP_RAM Module "r16x16dp"
// Created by LogiBLOX version M1.3.7(由 LogiBLOX M1.3.7 版生成)
//  on Thu Feb 12 15:27:46 1998(生成日期)
// Attributes (属性)
//  MODTYPE = DP_RAM
//  BUS_WIDTH = 16
//  DEPTH = 16
//------------------------------------------------------------

r16x16dp  ramblk1
(.A({ram_a_addr[4], ram_a_addr[3], ram_a_addr[2], ram_a_addr[1]}),
 .SPO(ram_a_data),
 .DI(ram_data),
 .WR_EN(wr_strobe),
 .WR_CLK(clk),
 .DPO(ram_b_data),
 .DPRA({ram_b_addr[4], ram_b_addr[3], ram_b_addr[2], ram_b_addr[1]}));

endmodule
```

如程序列表 4-17 所示的 r16x16dp.vei 模块仅仅是预综合网表(r16x16dp.ngo)的一个占位,它将在布局/布线过程中被插入。它仅仅定义了模块的端口。这个自动生成文件的接口部分被剪切并粘贴到调用的占位模块中。该程序列表所述.vei 文件必须被包含在如程序列表 4-16 所示的 Exemplar Logic LeonardoSpectrum 的输入文件列表中。

程序列表 4-17 RAM 占位模块(r16x16dp.vei)

```
//-----------------------------------------------------------
// LogiBLOX DP_RAM Module "r16x16dp"
// Created by LogiBLOX version M1.5.19(由 LogiBLOX M1.5.19 生成)
//  on Sun May 30 14:19:03 1999(生成日期)
// Attributes(属性)
//  MODTYPE = DP_RAM
//  BUS_WIDTH = 4
//  DEPTH = 16
//  STYLE = MAX_SPEED
//  USE_RPM = FALSE
//-----------------------------------------------------------

module r16x16dp(A, SPO, DI, WR_EN, WR_CLK, DPO, DPRA);
input    [3:0]    A;
output   [3:0]    SPO;
input    [3:0]    DI;
input             WR_EN;
input             WR_CLK;
output   [3:0]    DPO;
input    [3:0]    DPRA;
endmodule
```

很容易设想用外部 RAM 替代 LogiBLOX RAM 的情形。不同之处在于外部 RAM 模块端口和器件之间必须进行实际的引脚连接。当多个模块需要访问 RAM 时,RAM 的接口要进行相应的扩展。在这种情况下,将启用一个仲裁方案对 RAM 的访问优先权进行判别和协商。程序列表 4-18 给出了一个具有简单判别器功能的外部 RAM 接口实例(允许多个源去访问该 RAM)。这仅是用于商业设计的一个

实例,也许还有更好的方法实现此类设计。

程序列表 4-18　RAM 访问接口及仲裁设计

```
// 引自 Advanced Technology Video 公司的 arbit1.v(1998)
// 该引用经过允许。

module  arbit1 (clk, reset, chan0_ramaddr, chan0_dat_from_ram, chan0
_dat_to_ram, chan1_ramaddr, chan1_dat_from_ram, chan1_dat_to_
ram, address_preset, ram_rwn, ram_addr, ram_data_pins, data_rd, data
_wr, up_data_to_ram, up_data_from_ram, sram_addr_strobe, rd_
ack, wr_ack, ram_data_oe);

// 系统输入。
input    clk;    // 系统时钟。
input    reset;  // 系统复位。

// 控制信号。
output   [2:0]   rd_ack;  // 读操作结束的确认信号。
reg      [2:0]   rd_ack;
output   [2:0]   wr_ack;  // 写操作结束的确认信号。
reg      [2:0]   wr_ack;

// RAM 接口。
input    [12:1]  chan0_ramaddr;// 通道 0 RAM 地址指针。
wire     [12:1]  chan0_ramaddr;
input    [12:1]  chan1_ramaddr;// 通道 1 RAM 地址指针。
wire     [12:1]  chan1_ramaddr;
output   [15:0]  chan0_dat_from_ram;// 通道 0 RAM 读数据。
reg      [15:0]  chan0_dat_from_ram;
output   [15:0]  chan1_dat_from_ram;// 通道 0 RAM 读数据。
reg      [15:0]  chan1_dat_from_ram;
input    [15:0]  chan0_dat_to_ram; // 通道 0 RAM 写数据。
wire     [15:0]  chan0_dat_to_ram;
output   [15:0]  chan1_dat_to_ram; // 通道 1 RAM 写数据。
```

```
wire    [15:0]  chan1_dat_to_ram;
input   [2:0]   data_rd;// RAM 读请求。
wire    [2:0]   data_rd;
input   [2:0]   data_wr;// RAM 写请求。
wire    [2:0]   data_wr;
input           sram_addr_strobe; //预先加载地址计数器信号。
input   [15:0]  up_data_to_ram; // 写入 RAM 的数据。
output  [15:0]  up_data_from_ram; // RAM 读出的数据。
reg     [15:0]  up_data_from_ram;
input   [12:0]  address_preset; //微处理器地址计数器预置输入。

// RAM 输入/输出端口。
output  ram_rwn; // SRAM 读/写控制引脚,高电平表示"读"。
output  [12:0]  ram_addr;  // SRAM 地址引脚。
reg     [12:0]  ram_addr;
inout   [7:0]   ram_data_pins;  // 被写入的 RAM 数据。
wire    [7:0]   ram_data_in;
reg     [7:0]   ram_data_out;
output          ram_data_oe;  // RAM 输出使能。
reg             ram_data_oe;

// 局部变量。
reg     [3:0]   ram_state;
reg             ram_rdn;
reg     [11:0]  ram_addr_ctr;
// 存储自动累加地址的寄存器,计数字。

parameter ram_state_idle    =  0;
parameter ram_state1        =  1;
parameter ram_state2        =  2;
parameter ram_state3        =  3;
parameter ram_state4        =  4;
parameter ram_state5        =  5;
parameter ram_state6        =  6;
parameter ram_state7        =  7;
```

```
parameter ram_state8        =  8;
parameter ram_state9        =  9;
parameter ram_state10       =  10;
parameter ram_state11       =  11;
parameter ram_state12       =  12;
parameter ram_state13       =  13;
parameter ram_state14       =  14;
parameter ram_state15       =  15;

assign ram_rwn = ~ram_rdn; //高有效的局部信号。

// SRAM 数据引脚的控制。
assign ram_data_pins = ram_data_oe ? ram_data_out : 8'bz;
assign ram_data_in = ram_data_pins;

always @ (posedge clk or posedge reset)
    begin
        if (reset)
        begin
            ram_state   <=   ram_state_idle;
            ram_rdn     <=   0;
            ram_addr    <=   0;
            ram_data_out <=  0;
            rd_ack      <=   0;
            wr_ack      <=   0;
            ram_data_oe <=   0;
        end
    else begin case (ram_state)
     ram_state_idle:begin
          begin
          ram   rdn   <=   0;
          ram_addr    <=   0;
          ram_data_out <=  0;
          ram_data_oe <=   0;
          end
```

```
if (data_rd[0])
begin
ram_rdn <= 0;
ram_addr <= {chan0_ramaddr, 1'b0};
ram_state <= ram_state1;
end

else if (data_rd[1])
begin
ram_rdn <=  0;
ram_addr <=  {chan1_ramaddr, 1'b0};
ram_state <=  ram_state3;
end

else if (data_wr[0])
begin
ram_rdn <=  1;
ram_addr <=  {chan0_ramaddr, 1'b0};
ram_data_out <= chan0_dat_to_ram[7:0];
ram_data_oe <= 1;
ram_state <= ram_state5;
end

else if (data_wr[1])
begin
ram_rdn <=  1;
ram_addr <=  {chan1_ramaddr, 1'b0};
ram_data_out <=  chan1_dat_to_ram[7:0];
ram_data_oe <=  1;
ram_state <=  ram_state8;
end

else if (data_rd[2])  // 微处理器读请求。
begin
```

```
            ram_rdn <= 0;
            ram_addr <= {ram_addr_ctr, 1'b0};
            ram_state <= ram_state11;
            end

            else if (data_wr[2]) //微处理器写请求。
            begin
            ram_rdn <= 1;
            ram_addr <= {ram_addr_ctr, 1'b0};
            ram_data_out <= up_data_from_ram[7:0];
            ram_data_oe <= 1;
            ram_state <= ram_state13;
            end

            else          // Default.
            ram_state <= ram_state_idle;

            end

// 读通道 0。
ram_state1:
      begin
      ram_rdn <= 0;
      ram_addr <= {chan0_ramaddr, 1'b1};
      chan0_dat_from_ram[7:0] <= ram_data_in;
      rd_ack[0] <= 1;      // 提前赋值。
      ram_state <= ram_state2;
      end

ram_state2:
      begin
      ram_rdn <= 0;
      ram_addr <= {chan0_ramaddr, 1'b1};
      chan0_dat_from_ram[15:8] <= ram_data_in;
      rd_ack[0] <= 1;  // 确认信号保持,直到读信号被释放。
```

```
        if (data_rd[0])
        ram_state <= ram_state2; //保持,直到 rd 信号被释放。
        else
        begin
        rd_ack[0] <= 0; // 释放确认信号。
        ram_state <= ram_state_idle;
        end
        end

// 读通道 1。
ram_state3:
        begin
        ram_rdn <= 0;
        ram_addr <= {chan1_ramaddr, 1'b1};
        chan1_dat_from_ram[7:0] <= ram_data_in;
        rd_ack[1] <= 1; // 提前赋值。
        ram_state <= ram_state4;
        end

ram_state4:
        begin
        ram_rdn <= 0;
        ram_addr <= {chan1_ramaddr, 1'b1};
        chan1_dat_from_ram[15:8] <= ram_data_in;
        rd_ack[1] <= 1; // 确认信号保持,直到读信号被释放。
        if (data_rd[1]) // 保持,直到 rd 信号被释放。
        ram_state <= ram_state4;
        else
        begin
        rd_ack[1] <= 0; // 释放确认信号。
        ram_state <= ram_state_idle;
        end
        end

// 写通道 0。
```

```
ram_state5:
    begin
    ram_rdn <= 0;
    ram_addr <= {chan0_ramaddr, 1'b0};
    ram_data_out <= chan1_dat_to_ram[7:0];
    ram_data_oe <= 1;
    ram_state <= ram_state6;
    end

ram_state6:
    begin
    ram_rdn <= 1;
    ram_addr <= {chan0_ramaddr, 1'b1};
    ram_data_out <= chan1_dat_to ram[15:8];
    ram_data_oe <= 1;
    wr_ack[0] <= 1; // 提前释放。
    ram_state <= ram_state7;
    end

ram_state7:
    begin
    ram_rdn <= 0;
    ram_addr <= {chan1_ramaddr, 1'b1};
    ram_data_out <= chan1_dat_to_ram[15:8];
    ram_data_oe <= 1;
    wr_ack[0] <= 1; // 确认信号一直保持到写信号被释放为止。
    if (data_wr[0]) // 保持,直到 wr 信号被释放。
    ram_state <= ram_state7;
    else
    begin
    wr_ack[0] <= 0; // 释放确认信号。
    ram_state <= ram_state_idle;
    end
    end
```

```
// 写通道 1。
ram _state8:
    begin
    ram _ rdn < =  0;
    ram _ addr < =  {chan1 _ ramaddr, 1'b0};
    ram _ data _ out < =  chan1 _ dat _ to _ ram[7:0];
    ram _ data _ oe < =  1;
    ram _ state < =  ram _ state9;
    end

ram _state9:
    begin
    ram _ rdn  < =  1;
    ram _ addr < =  {chan1 _ ramaddr, 1'b1};
    ram _ data _ out < =  chan1 _ dat _ to _ ram[15:8];
    ram _ data _ oe < =  1;
    wr _ ack[1]  < =  1;  // 提前释放。
    ram _ state < =  ram _ state10;
    end

ram _state10:
    begin
    ram _ rdn < =  0;
    ram _ addr < =  {chan1 _ ramaddr, 1'b1};
    ram _ data _ out < =  chan1 _ dat _ to _ ram[15:8];
    ram _ data _ oe < =  1;
    wr _ ack[1] < =  1; // 确认信号一直保持到写信号被释放。
    if (data _ wr[1]) // 保持,直到 wr 信号被释放。
    ram _ state < = ram _ state10;
    else
    begin
    wr _ ack[1]  < =0;  // 释放确认信号。
    ram _ state < = ram _ state _ idle;
    end
    end
```

```
// 微处理器初始化读操作。
ram_state11:
    begin
    ram_rdn <= 0;
    ram_addr <= {ram_addr_ctr, 1'b1};
    up_data_from_ram[7:0] <= ram_data_in;
    ram_state <= ram_state12;
    end

ram_state12:    // 在此状态下,进行地址计数器的累加。
    begin
    ram_rdn <= 0;
    ram_addr <= {ram_addr_ctr, 1'b1};
    rd_ack[2] <= 1;
    up_data_from_ram[15:8] <= ram_data_in;
    ram_state <= ram_state_idle;
    end

// 微处理器初始化写操作。
ram_state13:
    begin
    ram_rdn <= 0;
    ram_addr <= {ram_addr_ctr, 1'b0};
    ram_data_out <= up_data_to_ram[7:0];
    ram_data_oe <= 1;
    ram_state <= ram_state14;
    end

ram_state14:
    begin
    ram_rdn <= 1;
    ram_addr <= {ram_addr_ctr, 1'b1};
    ram_data_out <= up_data_to_ram[15:8];
    ram_data_oe <= 1;
```

```
        wr_ack[2] <= 1;  // 提前释放。
        ram_state <= ram_state15;
        end

ram_state15:  // 在此状态下进行地址计数器的累加。
        begin
        ram_rdn      <= 0;
        ram_addr     <= {ram_addr_ctr, 1'b1};
        ram_data_out      <= up_data_to_ram[15:8];
        ram_data_oe       <= 1;
        wr_ack[2]         <= 0;
        ram_state    <= ram_state_idle;
        end

        default:
ram_state     <= ram_state_idle;

        endcase
        end
        end

// 当微处理器读或写时地址计数器进行累加操作。
        always @ (posedge clk or posedge reset)
        begin
            if (reset)
                ram_addr_ctr <= 0;
            else if (sram_addr_strobe)
                ram_addr_ctr <= address_preset;
            else if ((ram_state == ram_state12) |
                    (ram_state == ram_state15))
                ram_addr_ctr <= ram_addr_ctr + 1;
        end

    endmodule
```

现在的综合工具可以从逻辑结构中提取出 RAM,只要它们掩藏的不是太深以致无法被编译器发现的话。这意味着可以对随机逻辑设计进行剖析,编译器将会设法采用更有效的 RAM 功能的模块。

4.7　先入先出存储器(FIFO)介绍

FIFO(先入先出存储器)用于调整系统间数据的传输速率。数据以一定速率被写入,并以一个不同的(相同或更快的)速率被读出。从表面上看,FIFO 好像一个可大可小的寄存器文件夹。然而,实际上它被设计成一个 RAM 块,并具有独立的写地址计数器和读地址计数器。对于每次 FIFO 写操作,写计数器值(通常是格雷码计数器)都会递增加 1;对于每次 FIFO 读操作,读计数器会递增加 1。最小的标志组包括一个空标志(当读指针和写指针变得相同时进行设置)和盈标志(当读指针和写指针再次重合时进行设置,此时的重合是因为写地址的绕回)。FIFO 系统设计的主要目的是为了防止"溢出"而造成数据的丢失,这种情况的发生是由于新数据没有被写入或旧数据被重复写入。造成"溢出"的因素是 FIFO 的深度和读/写的频率。

进行 FIFO 设计的一个难点是标志位的设计。例如盈标志必须在写时钟域内设置,而在读时钟域内被读取和清除。这要求两个域之间的时间要同步,而做到这点往往是困难的。

和 RAM 类似,FIFO 也可在寄存器之外进行构建。然而,除非所要求的 FIFO 非常小,否则对设计者来说没有必要在寄存器之外采用一个 FIFO(用 RAM 替代),因为这样的设计是不经济的。

第五章　Verilog 测试

基于 SRAM 的 FPGA 器件的设计者对他们的设计并不做充分的仿真，因为对他们来说烧制器件并对其进行试验是一件很容易的事情，这也是一种设计趋势。对这种设计方法的优点有着各种各样的看法，毕竟最终的设计结果必须能被人接受。尽管如此，任何有助于提高我们设计质量的工具都应当被采用。使用工作性能没有有效保证的器件进行试验，这对 ASIC 的设计者及使用反熔丝技术的设计者来说简直就是一种奢望（而且技术上也没有相应的支持手段允许他们这样做）。对这些设计者来说，他们的工作不像 FPGA 设计者那样下载一个配置文件就够了，他们要在一个昂贵的器件上进行编程（这个器件一旦无法工作就得报废），或者对一条 ASIC 生产线进行周密检查（这条生产线可能价值十分昂贵）。这也就是为什么 ASIC（及类似 ASIC 的器件）需要进行长时间仿真的原因，而这个过程的控制和管理是件令人头痛的事情。设计者们的工作不仅仅是像移交产品那么简单，而是一天到晚围着计算机忙得昏天黑地。

Verilog 被定位为一种仿真和测试语言。它具有许多优秀的特性并得到了精心的设计和不断的扩展。对于一个 FPGA 设计者来说，除了考虑仿真器软件的成本及担心陷入无休止的“分析—等待—分析”的循环中之外，没有其他理由拒绝经常性地使用一个仿真器。由于目前已有一些优秀的专著对 Verilog 语言的仿真进行了详细的介绍（请参见本书所附的参考文献），因此，本章将仅仅就此问题做一简单的回顾。

大多数仿真器配备有一个波形观察器，其界面一般都被设计得十分醒目。但问题在于需要由人工对产生的波形进行分析和解释。波形是复杂的，并且我们需要利用仿真器给出输入和输出信号。幸运的是，Verilog 支持自动测试。这个强大的功能有助于测试、验证一个设计。你可以对设计进行更改并对修改后的效果进行认真地评估。另外，你应当学会判断已做的修改是否会对设计中原本工作正常的部分造成了不利的影响。

自动测试并不是万能的，它也会时常带来麻烦。在自动测试中，你会发现很多时间是花费在做无用功上，因为有些根本就不需要测试，或者其测试条件过于苛刻了。

嵌入式自测试(BIST)

在制造过程中,一个FPGA能够被编程用于完成内部和外部的自测试,之后也可以通过编程支持嵌入式(可移植的)的应用。对诸如DRAM,SRAM,FIFO这样的外部硬件可以进行全面测试,测试结果可以通过发光二极管(LED)显示,或通过串行端口(可能仅作为测试专用)输出。

5.1 编译指令

在Verilog语言中有许多功能强大的编译指令。注意使用"`"(后斜杠上标或重音标记符)作为这些编译指令的一部分。

`define, `ifdef, `else, `endif, `undef Verilog语言支持条件编译和执行。代码可以支持仿真但不能被综合,或者可以被有条件的综合支持其可选特性。可以定义一个宏变量来控制编译,如程序列表5-1所示。

程序列表5-1 `ifdef实例

```
// 条件编译实例。
// 当综合时,将下一行注释掉。
`define    test_mode;          // 定义 test_mode 宏。
 ⋮
`ifdef     test_mode
// 在此处插入 test_mode 的代码。可能是一个测试模块定义或仿真指
   令,不仅仅是内嵌的代码。

data_bus     <=     test_points;

`else
// 在此处插入非测试代码。
//`else 是可选的。
data_bus     <=     internal_data;

`endif
// 以非条件性代码继续。
```

一个宏定义可以用**`undef**指令取消定义。

`include 文件名　这个指令和 C 语言中的 # include 指令相似。被文件名指定的文件(可以在当前路径,也可以用全路径名定义)在编译过程中被插入 Verilog 代码中的指定位置。这种包含可以是嵌套的。换句话说,即一个被包含的文件本身也可以包含另一个文件。

`timescale 时间单位 /精度　仿真器使用的时间单位是可以自行设定的。例如,在程序列表 5-3 中,有这样一条语句:

```
# 75 reset = 0;
```

数字 75 代表以 timescale 定义的以时间为单位的时延时间,这里为 75ns。这个时延告诉仿真器,在执行下一条指令前需等待一段时间,此段时间的长度由此时延值决定。

在逻辑综合中的时延

时延是就仿真过程而言,并不针对逻辑综合过程。

没有一个硬件结构可以被专门用来产生一个时延。如果没有使用 timescale 指令定义的话,缺省的时间单位为 1ns。精度决定了延迟时间的准确度及仿真的分辨率。精度值必须等于或小于 timescale 定义的时间单位。timescale 定义的时间的单位可以是 s(秒),ms(毫秒, 10^{-3}s),μs(微秒, 10^{-6}s),ns(纳秒, 10^{-9}s),ps(皮秒, 10^{-12}s)或 fs(飞秒,10^{-15}s)。多数情况下,使用 1ns / 1ns,即时延单位为 1ns,时延精度 1ns。在一个设计中仅允许有一个时标。

5.1.1 系统任务

Verilog 语言的系统任务以 $字符开头。

$finish　当在代码中遇到这个指令时,**$finish** 使仿真结束。如果没有设置终止点,仿真程序将一直执行下去(或者直到计算机内存溢出和崩溃)。**$finish** 指令控制计算机返回到操作系统。

$stop　此系统任务指令使仿真被挂起,但并不返回到操作系统。仿真可以从停止点重新启动继续执行,或者在当前的仿真时刻执行其他的系统命令。

$display**(参数列 1,参数列 2)　这个系统任务和 C 语言中的 printf 命令类似。Verilog 仅仅在非波形输出模式下才可以顺利运行。如果没有了波形,如何知道我们的设计正在做什么呢?你可以如程序列表 5-2 和程序列表 5-3 所示的那样,在设计中插入一些 **$display 命令以便观察变量和其他的信息(比如仿真的时间)。

程序列表 5-2 简单的 $display 实例

```
module display1 (clock, reset);
input      clock, reset;
reg        [7:0] count_val;
always @ (posedge clock or posedge reset)
    if (reset)
    count_val <= 0;
    else
    begin
    count_val <= count_val + 1;
    $display (count_val);
    end
endmodule
```

程序列表 5-3 简单的 $display 实例的测试程序

```
// 显示实例的测试。
module disp1_tf;
`timescale 1ns / 1ns
reg clock, reset;

parameter clk_period = 20;
display1 u1 (clock, reset);
always begin
    # (clk_period / 2) clock = ~clock;
    end
initial
begin
    clock      =    0;
    reset      =    1; // 声明系统复位。
    # 75 reset =    0;
    # 1000 $finish;
end
endmodule
```

程序列表 5-4 给出了 Silos Ⅲ 仿真运行的结果。开始的一些零是在“复位”期间内寄存器 count _ val 中的内容。**$display** 显示的数的缺省格式是十进制数,且在每次执行后输出一个回车(即换行)。若希望输出不换行,可以使用 **$write** 系统任务。

程序列表 5-4 简单的 $display 实例的输出列表

```
                S I L O S  I I I    Version 99.100
             DEMO COPY LIMITED TO 100 to 200 DEVICES
        Copyright (c) 1999 by SIMUCAD Inc. All rights reserved.
        No part of this program may be reproduced, transmitted,
        transcribed, or stored in a retrieval system, in any
        form or by any means without the prior written consent of
          SIMUCAD Inc., 32970 Alvarado-Niles Road, Union City,
                       California, 94587, U.S.A.
                 (510)-487-9700  Fax: (510)-487-9721
           Electronic Mail Address:   "silos@simucad.com"

!file .sav="display1"
!control .sav=3
!control .savcell=0
!control .disk=1000M

Reading "c:\verilog\sourcecode\disp1_tf.v"
Reading "c:\verilog\sourcecode\display1.v"
sim to 0
     Highest level modules (that have been auto-instantiated):
            (disp1_tf disp1_tf
     3 total devices.
     Linking ...

     3 nets total: 11 saved and 0 monitored.
     74 registers total: 74 saved.
     Done.

     0 State changes on observable nets.

     Simulation stopped at the end of time 0.000000000s.
Ready: sim
   0
   1
   2
   3
   4
   5
   6
   7
   8
   9
  10
  11
  12
```

```
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
    313 State changes on observable nets in 0.33 seconds.
    948 Events/second.

    Simulation stopped at the end of time 0.000001075s.
Ready:
```

文本和数字可以通过转义字串来定义格式。转义字串前冠以 \ 或 %符号。如下面的几个例子：

%h 以十六进制形式显示数

%d 以十进制形式(缺省值)显示数

%o 以八进制形式显示数

%b 以二进制形式显示数

%c 以 ASCⅡ码形式显示

%t 以当前时间格式显示

"string" 显示文本串

\n 换行
\t 制表符
\ literal literal 可以是另一个 \ (打印一个 \ 符),一个" (打印一个" 符),
或一个%(打印一个 % 符)

为了有个直观的理解,请参见程序列表 5-5、5-6 和 5-7。

程序列表 5-5 带有格式定义的简单 $display 实例的输出列表

```
// 显示带有格式的测试。
module time_setup;
initial
    begin
// 时间格式为:ns(-9),小数点后保留 2 位(2)。
// 时间单位后缀为 ns,显示的时间所占位数最少为 3 位。
    $timeformat (-9, 2, "ns", 3);
    end
endmodule

module disp2_tf;
`timescale 1ns / 1ns
reg    clock, reset;
parameter   clk_period = 20;

display2 u1 (clock, reset);
always begin
    #(clk_period / 2) clock = ~clock;
    end

initial
begin
    clock = 0;
    reset = 1; // 声明系统复位。
    #75 reset = 0;
    #1000 $finish;
```

```
end
endmodule
```

程序列表 5-6 带有格式定义的简单 $display 实例的输出列表

```
module display2 (clock, reset);
input    clock, reset;
reg    [7:0] count _ val;
// 进行简单打印输出时,将下面的代码行注释掉。
`define verbose
always @ (posedge clock or posedge reset)
begin
    if (reset)
    count _ val < = 0;
    else
    begin
    count _ val < = count _ val + 1;
    `ifdef verbose
    $write ("count _ val = %h", count _ val);
    $display ("Current time = %t", $time);
    `else
    $write (":", count _ val);
    `endif
    end
end
endmodule
```

程序列表 5-7 带有格式定义的简单 $display 实例的输出列表

```
          S I L O S  I I I    Version 99.100
        DEMO COPY LIMITED TO 100 to 200 DEVICES
Copyright (c) 1999 by SIMUCAD Inc. All rights reserved.
No part of this program may be reproduced, transmitted,
transcribed,   or stored in a retrieval system, in any
form or by any means without the prior written consent of
  SIMUCAD Inc., 32970 Alvarado-Niles Road, Union City,
              California, 94587, U.S.A.
         (510)-487-9700  Fax: (510)-487-9721
   Electronic Mail Address:   "silos@simucad.com"
```

```
!file .sav="display2"
!control .sav=3
!control .savcell=0
!control .disk=1000M

Reading "c:\verilog\sourcecode\time_setup.v"
Reading "c:\verilog\sourcecode\display2.v"
sim to 0
     Highest level modules (that have been auto-instantiated):
            (time_setup time_setup
            (disp2_tf disp2_tf
     4 total devices.
     Linking ...
     3 nets total: 11 saved and 0 monitored.
     74 registers total: 74 saved.
     Done.

     0 State changes on observable nets.
     Simulation stopped at the end of time 0.000000000s.
Ready: sim
count_val = 00 Current time = 90.00 ns
count_val = 01 Current time = 110.00 ns
count_val = 02 Current time = 130.00 ns
count_val = 03 Current time = 150.00 ns
count_val = 04 Current time = 170.00 ns
count_val = 05 Current time = 190.00 ns
count_val = 06 Current time = 210.00 ns
count_val = 07 Current time = 230.00 ns
count_val = 08 Current time = 250.00 ns
count_val = 09 Current time = 270.00 ns
count_val = 0a Current time = 290.00 ns
count_val = 0b Current time = 310.00 ns
count_val = 0c Current time = 330.00 ns
count_val = 0d Current time = 350.00 ns
count_val = 0e Current time = 370.00 ns
count_val = 0f Current time = 390.00 ns
count_val = 10 Current time = 410.00 ns
count_val = 11 Current time = 430.00 ns
count_val = 12 Current time = 450.00 ns
count_val = 13 Current time = 470.00 ns
count_val = 14 Current time = 490.00 ns
count_val = 15 Current time = 510.00 ns
count_val = 16 Current time = 530.00 ns
count_val = 17 Current time = 550.00 ns
count_val = 18 Current time = 570.00 ns
count_val = 19 Current time = 590.00 ns
count_val = 1a Current time = 610.00 ns
count_val = 1b Current time = 630.00 ns
count_val = 1c Current time = 650.00 ns
count_val = 1d Current time = 670.00 ns
count_val = 1e Current time = 690.00 ns
count_val = 1f Current time = 710.00 ns
count_val = 20 Current time = 730.00 ns
count_val = 21 Current time = 750.00 ns
count_val = 22 Current time = 770.00 ns
count_val = 23 Current time = 790.00 ns
count_val = 24 Current time = 810.00 ns
count_val = 25 Current time = 830.00 ns
count_val = 26 Current time = 850.00 ns
count_val = 27 Current time = 870.00 ns
```

```
count_val = 28 Current time = 890.00 ns
count_val = 29 Current time = 910.00 ns
count_val = 2a Current time = 930.00 ns
count_val = 2b Current time = 950.00 ns
count_val = 2c Current time = 970.00 ns
count_val = 2d Current time = 990.00 ns
count_val = 2e Current time = 1010.00 ns
count_val = 2f Current time = 1030.00 ns
count_val = 30 Current time = 1050.00 ns
count_val = 31 Current time = 1070.00 ns
     313 State changes on observable nets in 0.76 seconds.
     411 Events/second.

     Simulation stopped at the end of time 0.000001075s.
Ready:
```

使用 **$monitor** 指令，对一组变量进行监控，当它们发生变化时，可以将它们的值全部显示出来。**$monitor** 的信号列表和格式控制和 **$display** 指令的相应部分的语法是类似的。程序列表 5-8 和程序列表 5-9 给出了使用 $monitor 指令的几个例子，它们对应的输出如程序列表 5-10 所示。

$monitor(信号列表和格式);

$monitoron/$monitoroff;

程序列表 5-8 $monitor 实例（display3.v）

```
module display3 (clock, reset);
input       clock, reset;
reg    [7:0]    count_val;

always @ (posedge clock or posedge reset)
begin
    if (reset)
    count_val <= 0;
    else
    begin
        count_val <= count_val + 1;
    end
end
endmodule
```

程序列表 5-9 $monitor 实例的测试程序（disp3 _ tf.v）

```
// 在测试程序中使用 $monitor。
module time _ setup2;
initial
    begin
    `timescale 1ns / 1ns
    $timeformat ( - 9, 2, "ns", 3);
    end
endmodule

module disp3 _ tf;

    reg       clock, reset;
    wire      [7:0] count _ val;
    parameter clk _ period = 20;

display3 u1 (clock, reset);

always
    begin
    # (clk _ period / 2) clock = ~ clock;
    end

initial
begin
    $monitor ($time, "Counter value: %h", u1.count _ val);
    clock = 0;
    reset = 1; // 声明系统复位。
    # 75 reset = 0;
    # 1000 $finish;
end
endmodule
```

程序列表 5-10 $monitor 实例的输出列表

```
90000000000.00 ns Counter value: 01
110000000000.00 ns Counter value: 02
130000000000.00 ns Counter value: 03
150000000000.00 ns Counter value: 04
170000000000.00 ns Counter value: 05
190000000000.00 ns Counter value: 06
210000000000.00 ns Counter value: 07
230000000000.00 ns Counter value: 08
250000000000.00 ns Counter value: 09
270000000000.00 ns Counter value: 0a
290000000000.00 ns Counter value: 0b
310000000000.00 ns Counter value: 0c
330000000000.00 ns Counter value: 0d
350000000000.00 ns Counter value: 0e
370000000000.00 ns Counter value: 0f
390000000000.00 ns Counter value: 10
410000000000.00 ns Counter value: 11
430000000000.00 ns Counter value: 12
450000000000.00 ns Counter value: 13
470000000000.00 ns Counter value: 14
490000000000.00 ns Counter value: 15
510000000000.00 ns Counter value: 16
530000000000.00 ns Counter value: 17
550000000000.00 ns Counter value: 18
570000000000.00 ns Counter value: 19
590000000000.00 ns Counter value: 1a
610000000000.00 ns Counter value: 1b
630000000000.00 ns Counter value: 1c
650000000000.00 ns Counter value: 1d
670000000000.00 ns Counter value: 1e
690000000000.00 ns Counter value: 1f
710000000000.00 ns Counter value: 20
730000000000.00 ns Counter value: 21
750000000000.00 ns Counter value: 22
770000000000.00 ns Counter value: 23
790000000000.00 ns Counter value: 24
810000000000.00 ns Counter value: 25
830000000000.00 ns Counter value: 26
850000000000.00 ns Counter value: 27
870000000000.00 ns Counter value: 28
890000000000.00 ns Counter value: 29
910000000000.00 ns Counter value: 2a
930000000000.00 ns Counter value: 2b
950000000000.00 ns Counter value: 2c
970000000000.00 ns Counter value: 2d
990000000000.00 ns Counter value: 2e
1010000000000.00 ns Counter value: 2f
1030000000000.00 ns Counter value: 30
1050000000000.00 ns Counter value: 31
1070000000000.00 ns Counter value: 32
```

以下是有关文件操作的命令,用黑字体表示。

$dumpfile("文件名");

```
$dumpvars(层次,变量所在的模块);
$dumpvars(0,变量所在的模块);
$dumpvars;
$dumpon;
$dumpoff;
$dumpall;
$dumplimit(以字节为单位的文件长度);
```

Verilog 语言能够以 ASCⅡ码格式存储仿真结果。这种文件的格式称作值变转储或 VCD。当一个变量发生变化时,其变化的信息被转储进 VCD 文件中。不带自变量列表的 $dumpvars 指令转储设计中的所有变量。$dumpvars(0,模块名)指令转储所列模块及被其调用的所有模块中的变量。变量可以在文件列表中按层次进行区分(模块 1.模块 2.变量名)。

VCD 文件可以非常大。用 $dumplimitng 指定 VCD 文件的最大长度,当文件长度达到此最大长度时,转储停止。程序列表 5-11 和程序列表 5-12 给出了 $dump 指令的一些实例,部分输出结果如程序列表 5-13 所示。

程序列表 5-11 $dump 选项

```
#100  $dumpon ;     // 延迟 100 个单位时间后转储所有的变量。
#100  $dumpoff ;    // 延迟 100 个单位时间后将转储任务挂起。
#100  $dumpall ;    // 转储所有变量的当前值。
```

程序列表 5-12 $dumpvars 实例

```
// 值变转储实例。
module time _ setup3;
initial begin
    `timescale 10ns / 1ns
    $timeformat ( -9, 2, "ns", 3);
    $dumpfile ("bigdump.dmp"); // 打开转储文件。
    end
endmodule

module disp4 _ tf;
```

```
    reg    clock, reset;
    wire   [7:0] count_val;
    parameter   clk_period = 20;
display4 u1 (clock, reset);
always begin
    #(clk_period / 2) clock = ~clock;
    end
initial begin
      $timeformat (-9, 2, "ns", 3);
      $dumpvars;
          clock = 0;
          reset = 1; // 声明系统复位。
     #75 reset = 0;
    #1000 $finish;
end
endmodule

module display4 (clock, reset);
input clock, reset;

reg [7:0] count_val;

always @ (posedge clock or posedge reset)
begin
    if (reset)
    count_val <= 0;
    else
    begin
    count_val <= count_val + 1;
    end
end
endmodule
```

程序列表 5-13 $dumpvars 输出列表中的一段

```
$scope module disp4_tf $end
$var reg    1 !    reset $end
$var reg    1 "    clock $end

$scope module u1 $end
$var wire   1 #    clock $end
$var wire   1 $    reset $end
$var reg    8 %    count_val [7:0] $end
$upscope $end

$upscope $end

$enddefinitions $end
#0
$dumpvars
1!
0"
0#
1$
b00000000 %
$end
```

程序列表 5-13 是转储文件 digdump.dmp 的一小部分。注意，在 **$dumpvars** 指令作用范围内的每一个信号(由于 **$dumpvars** 对作用范围没有限制，所以设计中的所有信号都被转储)都被分配了一个关键字符(比如！代表复位)，且这种速记方式被应用在转储文件中。VCD 文件表述方式不够友好，但它是一种可以被其他工具所读取的格式。

$readmemh("文件名",存储器名); 从指定文件中读取十六进制值
$readmemb("文件名",存储器名); 从指定文件中读取二进制值

可选的起始地址和结束地址可被用来对从文件中读取的数据进行限制。地址是阵列中第 n 个数据单元位置的索引。可以仅指定起始地址，而不指定结束地址。在这种情况下，从指定的起始地址处开始加载数据，直至存储器阵列的末端为止。

$readmemh("文件名",存储器名,起始地址,结束地址);

5.2 自动测试

确保一个设计得到全面测试的惟一途径是实现任务的自动化。可以创建一个校验表。当修改过源代码后，所有的自动测试都应被再次执行。这个过程是令人

头疼的,因为大部分的工作量不是用在解决设计本身的错误方面,而是花在了对测试程序所出错误的调试上。尽管如此,对于一个设计来说,没有其他更好的测试方法。

程序列表 5-14 给出了一个简单的数字滤波器设计和测试实例。设计的源代码如程序列表 5-15 所示。输出值在图 5-1 中给出。这个设计实现了一个一维低通 Pyramidal 滤波器,它使用了 5 个采样值,系数分别为:0.0625,0.125,0.625,0.125,0.625(注意:所有系数之和为 1)。这个滤波器的设计是简单的,且存在截断误差,但它作为一个自动测试的实例还是适宜的。

程序列表 5-14 Pyramidal 滤波器测试程序

```
// Pyramidal 滤波器实例。

module time _ setup4;
initial
    begin
    `timescale 1ns / 1ns
    $timeformat ( -9, 2, "ns", 3);
    end
endmodule

module pf1 _ tf;

    reg     clock, reset;
    reg [7:0]     tap _ unfilt;

// 定义测试值存储阵列。
    reg [7:0] test _ pattern[0:31];

// 定义滤波器输出测试阵列。
    reg [7:0] verify _ pattern[0:31];
    reg [4:0] mem _ index;
    wire [7:0] tap _ filt;
    reg [7:0] tap _ test, filt _ test;
    reg    flag;
```

```
    parameter clk_period = 20;

pfilt1 u1 (clock, reset, tap_unfilt, tap_filt);

always begin
    # (clk_period / 2)
    clock       = ~clock;
    tap_unfilt  = test_pattern [mem_index];
// filt_test    = tap_filt [mem_index];
    tap_test    = verify_pattern [mem_index];
    mem_index   = mem_index + 1;
    if (mem_index == 0) $finish;
    end

always begin
    # (clk_period)
    if (! reset & (tap_filt != tap_test))
      begin
$display ($time, "ERROR! tap_filt = %h tap_test = %h", tap_filt, tap
        _test);
    flag <= 1;
    end
// else $display ("All is okay.");
    end

initial
begin
clock       = 0;
mem_index   = 0;
tap_test    = 0;
filt_test   = 0;
tap_unfilt  = 0;
flag        = 0;
reset       = 1; // 声明系统复位。
```

```
// 从 test _ pattern 阵列存储的文件中读取测试模式数据。
$readmemh ("pfilt1.tst", test _ pattern);
$readmemh ("verify1.tst", verify _ pattern);

#(clk _ period * 2) reset = 0;
end
endmodule
```

程序列表 5-15 Pyramidal 滤波器 Verilog 描述

```
// Pyramidal 滤波器实例。
// 这实现了一个低通滤波器,其系数为:
// .0625   .125   .625   .125   .0625
// 增益 = 1。
module pfilt1 (clock, reset, tap4, tap _ out[8:1]);
input    clock, reset;
input    [7:0]    tap4;
output          tap _ out;
reg    [8:0]    tap _ out;
reg    [7:0]    tap0, tap1, tap2, tap3;

// 中间累加(流水线)寄存器。
reg [4:0] sum1;
reg [5:0] sum2;
reg [7:0] sum3;

always @ (posedge clock or posedge reset)
begin
    if (reset)
    begin
    tap0    < = 0;
    tap1    < = 0;
    tap2    < = 0;
    tap3    < = 0;
```

```
        tap_out    <= 0;
        sum1       <= 0;
        sum2       <= 0;
        sum3       <= 0;
        end
        else
        begin
        tap0       <= tap1;
        tap1       <= tap2;
        tap2       <= tap3;
        tap3       <= tap4;

    // 乘以 0.0625(相当于除以 16):
    // 左移两位。
    // 结果寄存器必须比用于保存进位的输入寄存器大 1。
        sum1 <= tap0[7:4] + tap4[7:4];

    // 乘以 0.125 (相当于除以 8):
        sum2 <= tap1[7:3] + tap3[7:3];

    // 乘以 0.625 (5/8) 相当于:除以 2 + 除以 8。
        sum3 <= tap2[7:1] + {2'b0, tap2[7:3]};

    // 最终结果为: sum1 + sum2 + sum3。
    // 如果要求设计的执行速度更快,则可采用更多的流水线,用以展开加法
       逻辑。
    // 通过截断处理以得到 8 位结果。
    // 逻辑可以相加,如果必要,可以进行舍入,以得到 8 位结果。

        tap_out[8:0] <= {3'b0, sum1} + {2'b0, sum2} + sum3;
        end
    end
    endmodule
```

这个 Pyramidal 滤波器设计从两个外部文件中读取数据。用于滤波器仿真的

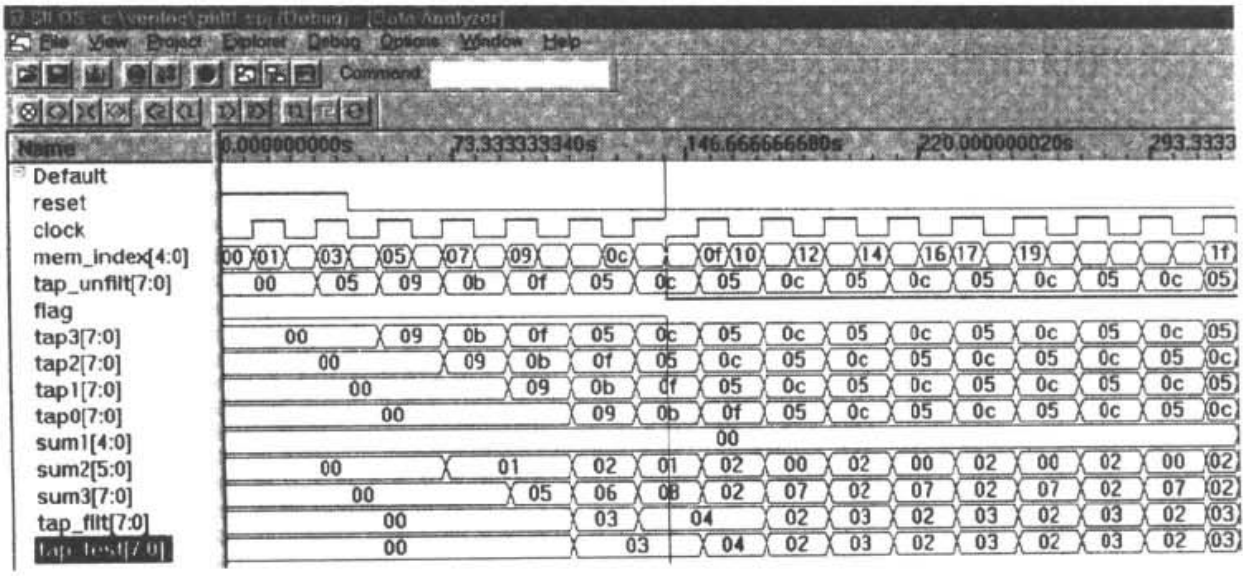

图 5-1　Pyramidal 滤波器波形

激励数据从文件 pfilt1.tst 中获取(文件 pfilt1.tst 见程序列表 5-16),用于验证滤波器输出的数据(不包含为测试而故意设置的故障点)可从程序列表 5-17 所示的 verify1.tst 文件中获取。出现错误时显示的格式如下所示:

140000000000.00 ns ERROR! tap_filt = 04 tap_test = 03

被 Verilog 读取的文件允许使用三种输入类型:数(十六进制或二进制),空格和注释。

程序列表 5-16　Pyramidal 滤波器输入数据列表(pfilt1.tst)

```
0 //由于数据值在每个时钟边沿被加载,所以这里的每个数值均出现两
    次;
0 //这两个数值都被要求在一个完整的时钟周期内保持有效。
5
5
9
9
b
b
f
```

```
f
5
5
c
c
5
5
c
c
5
5
c
c
5
5
c
c
5
5
c
c
5
5
```

程序列表 5-17 Pyramidal 滤波器测试数据列表(verify1 . tst)

```
0
0
0
0
0
0
0
0
0
```

```
0
3
3
3 // 添加的错误,应当为 4。
3 // 添加的错误,应当为 4。
4
4
2
2
3
3
2
2
3
3
2
2
3
3
2
2
```

第六章　实用设计:工具、技术及权衡策略

本书的内容并不是泛泛而写的。从理论上讲,编写可移植代码是可行的,这些代码可运行在任何一家供应商所提供的硬件上(包括 ASIC 工序)。但在性能和效率之间的取舍没必要给予过多地考虑。在完成一个实用(快速和小型)设计时,一般都采用提供商所提供的库和技术。作为有经验的 FPGA 设计人员,会对这些东西都很熟悉。

在实际设计中,我们需选择一家 FPGA 提供商。两大实力雄厚的公司:Xilinx 和 Altera 占据了目前 80% 的 FPGA 市场份额。两者都是拥有强大技术实力的公司。本书重点介绍 Xilinx 公司的 FPGA 器件。在第七章中,将对这两家公司的产品体系结构间的具体差异进行简要的介绍。在编译器领域,至少到作者写书时,市场的领先者是 Exemplar Logic 和 Synplicity。二者都是优秀的产品。此外,备受注目的与 Xilinx 设计管理器绑定的 Synopsis 公司的 FPGA Express 也即将投放市场,它是目前市场上可以得到的价格最便宜的设计工具包,而这并非是其受到关注的惟一原因。本书叙述中,将采用 Exemplar Logic 的 LeonardoSpectrum 软件。

本书使用到的设计流程和设计工具是:

• 设计定义。在一个任务被明确定义前就开始进行代码的编写是不明智的。在实际中,我们经常在一项任务的市场定位被确定前,就不得不展开工作。但即便如此,我们也应尽可能做到首先确定任务范围。

• 设计划分。把一个任务进行划分。尽可能采用已有的成熟设计。当设计一个 5000 ~ 10 000 门的模块时,预计每行代码大约生成 20 个门(这个值可以根据实际情况调整),因此每个模块将有 250 ~ 500 行代码(等于分号的数目)。

• 编写代码。使用彩色代码编辑器以避免语法错误(这种用不同颜色书写代码的方法具有实时语法检查的功能,并且是相当有效的),完成面积、时序,时钟/复位资源的分配,给出引脚分配的约束。

• 查找语法错误,利用每一个你所能找到的工具(包括不同的仿真器)对你的设计进行编译。你将发现不同的提供商在产品中对错误信息及帮助等级的显示方式是不同的。

• 如果可能,使用像 Verilint 一样的 lint 程序。Verilog 编译器可以检查出几种错误,如出现失配的向量,产生不该有的锁存器等。对于这些错误,Verilint 都可以捕捉到。要密切注意逻辑综合过程中出现的警告信息,它们可能预示着某些存在于逻辑综合方面的隐患。

• 设计仿真。编写测试程序并利用自动测试和波形图验证设计的正确性。本书中，使用 Simucad 公司的 Silos Ⅲ，在尽可能的高水平上对设计进行仿真。

• 编译代码。本书中，使用 Exemplar Logic 的 LeonardoSpectrum 生成设计网表。对逻辑门的数目进行监视，对速度进行估计。使用原理图观察器以确保所编代码按照预期的方式执行。检查时钟及复位的执行情况。确保全局信号按预期的方式被检测和控制。

• 网表的布局和布线。本书中，将使用 Xilinx 的设计管理器生成一个可下载的配置文件。为满足设计要求，需对布局/布线进行控制，并尝试做尽可能多次的布局/布线。

• 下载设计及在目标硬件上的测试。FPGA 设计者如果手头没有合适的工具，或者缺乏耐性，很多人都倾向于过早地跳到这一步。设计者应当确保在电路测试之前其原理设计的正确性。

6.1　使用 LeonardoSpectrum 进行编译

6.1.1　使用 LeonardoSpectrum 脚本

LeonardoSpectrum 提供图形用户界面和设计向导。然而，设计者很快会发现使用脚本是构造一个网表更快和更有效的方法。由设计向导生成的设计脚本可以被调用和执行。程序列表 6-1 是一个由设计向导所生成的脚本实例。

程序列表 6-1　LeonardoSpectrum 脚本实例

```
set register2register 50
set input2register 50
set input2output 50
set register2output 50
set output_file "C:/verilog/latch.edf"
set novendor_constraint_file FALSE
_gc_read_init
_gc_run_init
set input_file_list {"C:/verilog/latch.v"}
set part 4013xlPQ160
set process 3
set wire_table 4013x1 -3 _avg
set nowrite_eqn FALSE
```

```
set chip TRUE
set area TRUE
set report brief
set global _ sr reset
set output _ file "C:/verilog/latch.edf"
set target xi4xl
_ gc _ read
set register2register 50
set input2register 50
set input2output 50
set register2output 50
set output _ file ''''
```

下面对以上脚本逐行进行解释。

set register2register 50 在设计向导中,选择 20MHz 作为全局约束条件,即时钟周期为 50ns。这个约束条件表示在寄存器间的所有信号(从一个寄存器的输出到另一个寄存器的输入)必须在 50ns 内被判定。

set input2register 50 基于运行在 20MHz 时钟下的总体设计要求,器件输入和寄存器之间的所有信号必须在 50ns 内被判定。设计者必须考虑如何确保在器件外部逻辑中满足此设计要求。依据外部电路的时序情况,对这些节点可能需要采用更加严格的约束条件。当使用具有 I/O 寄存器的器件时,此问题是易于解决的。

set input2output 50 基于 20MHz 时钟的总体设计要求,器件逻辑及由此逻辑驱动的器件引脚之间的所有信号必须在 50ns 内被判定。为了满足器件外部电路的要求,此约束条件可能需加严。

set register2output 50 基于 20MHz 时钟的总体设计要求,器件内部寄存器和器件输出引脚之间的所有信号必须在 50ns 内被判定。为了满足器件外部电路的要求,此条件可能需加严。当使用具有 I/O 寄存器的器件时,此问题是易于解决的。

set global _ sr reset 将全局置位/复位资源和复位信号相连接。Xilinx 支持

用户自定义的全局复位资源的连接，它可以被器件中任何寄存器所使用。此信号需要识别并应用于每一个需要复位信号的块中。

lut _ max _ fanout 4　此语句用来控制输出负载（它影响到设计面积及工作速度）。LeonardoSpectrum 允许设计者对连接到 CLB 上的负载的最大数目进行控制。在此例中，采用的是数目为 4 的轻负载。这导致为减少负载而采用多个缓冲器。

set output _ file "C:/verilog/latch.edf"　由编译器生成采用 EDIF(.edf)文件格式的网表，按指定的路径保存。注意在路径的表示中采用了 UNIX 书写格式的正斜杠(/)。文件输出格式包括：.edf(edif)，.edif(edif)，.eds(edif)，.sdf(标准延迟格式)，.v(verilog)，.verilog(verilog)，.vhd(vhdl)，.vhdl(vhdl)，.xdb(二进制转储)，.xnf(Xilinx 网表格式)。

set novendor _ constraint _ file FALSE　这个双重否定表示将生成一个 FPGA 提供商(此例中是指 Xilinx 公司)约束文件，并用它来指导逻辑的布局和布线。

set input _ file _ list {"C:/verilog/latch.v"}　这是被用于连接的输入文件列表。在此例中，设计仅用到了一个文件。注意在路径的表示中运用了 UNIX 书写格式的正斜杠(/)。文件输出的格式包括：.edf(edif)，.edif(edif)，.eds(edif)，.sdf(标准延迟格式)，.v(verilog)，.verilog(verilog)，.vhd(vhdl)，.vhdl(vhdl)，.xdb(二进制转储)，.xnf(Xilinx 网表格式)。

set part 4013xlPQ160　设计所使用的器件为 Xilinx 公司的 4013XL 产品(大约有 13000 门)，采用的是 PQ160(160 引脚的表贴工艺)封装形式。

set process 3　设置使用的是 LeonardoSpectrum 的第三级设计流程。第一级和第二级是第三级的子集。第一级针对的是采用单个提供商的 FPGA 的设计流程；第二级针对的是采用多个提供商的 FPGA 的设计流程；第三级针对采用多个提供商产品并包含 ASIC 设计的流程。

set wire _ table 4013x1-3 _ avg　-3 速度级器件平均负载情况下(和最坏情况相比)的时延。

set nowrite _ eqn FALSE　这是另一个双重否定，它表示从设计网表中提取

电路图时,将器件方程式写入电路图中。

set chip TRUE 对应一个具体的器件,设计网表被进行编译,并且在网表中将包含和顶层引脚相对应的 I/O 引脚。

set area TRUE 本设计将按面积最优化进行编译。这个选项表示针对速度进行编译。LeonardoSpectrum 的第三级允许单个模块针对面积或速度进行编译(一个强大的功能)。

set report brief 报告将采用简洁格式。

hierarchy _ preserve TRUE LeonardoSpectrum 在对模块进行组合时试图通过保持其原有的层次结构来减少逻辑。这种缩减是不允许的。而在调试过程中设置此值为 TRUE 是有用的,因为此种情况下信号名将很有可能被保留。

set target xi4xl 使用 Xilinx 4000XL 元件库中的原语来实现此设计。

程序列表 6-2 给出了此设计的实现方法。这个设计存在一个问题:即会附带生成出一个原设计中不应存在的锁存器。LeonardoSpectrum 将以友好的方式指出存在的问题。所采用的信息记录格式如程序列表 6-3 所示。

程序列表 6-2 Verilog 锁存器

```
// 基本锁存器。
module latch2(q, q_not, set, reset);
    output    q, q_not;
    reg    q;
    input    set, reset;
    wire    set, reset;

    assign q_not = ~q;
    always @ (set or reset)
    begin
        if (set)
        q = 1;
        else if (reset)
```

```
        q = 0;
    end
endmodule
```

程序列表 6-3　对于 Verilog 锁存器的 LeonardoSpectrum 信息记录

```
-- Reading target technology xi4xl
Reading library file
`C:\EXEMPLAR\LEOSPEC\V19991D\lib\xi4xl.syn`...
Library version = 1.8
Delays assume: Process=3
-- read -tech xi4xl { "C:/Verilog/SourceCode/latch2.v" }
-- Reading file 'C:/Verilog/SourceCode/latch2.v'...
-- Loading module latch2
-- Compiling root module 'latch2'
"C:/Verilog/SourceCode/latch2.v",line 4: Warning, q is not always
assigned. latches could be needed.
-- Pre Optimizing Design .work.latch2.INTERFACE
Info: Finished reading design
->_gc_run
-- Run Started On Mon Sep 06 10:42:20 Pacific Daylight Time 1999
--
-- optimize -target xi4xl -effort quick -chip -area -
hierarchy=auto
Using wire table: 4013xl-3_avg
Info, Inferred net 'set' as GSR net.
-- Start optimization for design .work.latch2.INTERFACE
Using wire table: 4013xl-3_avg

      Pass      Area     Delay      DFFs  PIs   POs --CPU--
                (FGs)     (ns)                      min:sec
      1            0        7          0     2     2   00:00
Info, Added global buffer BUFG for port reset
Using wire table: 4013xl-3_avg
-- Start timing optimization for design .work.latch2.INTERFACE
No critical paths to optimize at this level

*******************************************************

Cell: latch2    View: INTERFACE    Library: work

*******************************************************

  Number of ports :                  4
  Number of nets :                  10
  Number of instances :              9
  Number of references to this view :  0
Total accumulated area :
  Number of BUFG :                   1
  Number of CLB Latches :            1
  Number of IBUF :                   1
  Number of OBUF :                   2
  Number of STARTUP :                1
```

```
*************************************************
Device Utilization for 4010xlPQ100
*************************************************
Resource                Used    Avail    Utilization
-------------------------------------------------
IOs                     4       77       5.19%
FG Function Generators  0       800      0.00%
H Function Generators   0       400      0.00%
CLB Flip Flops          0       800      0.00%

-------------------------------------------------
                        Clock Frequency Report

   Clock                 : Frequency
    ------------------------------------

   reset                 : 3333.3 MHz
```

以上信息记录的部分项目注释如下。

• Reading library file `C: \ EXEMPLAR \ LEOSPEC \ V19991D \ lib \ xi4xl.syn`...

LeonardoSpectrum 采用针对 Xilinx 公司 4xxxXL 系列产品的 xi4xl 元件库来实现这个锁存器设计。

• "C:/verilog/latch.v", line 6 : Warning, q is not always assigned. latches could be needed.

LeonardoSpectrum 友好地给出了一个锁存器被生成的警告。通常情况下,这是一个由于在代码中没能对所有输出条件进行完整定义而造成的错误。

• optimize-target xi4xl-effort quick-chip-area-flatten = TRUE

我们选取 Xilinx 公司的 4010XL 作为目标器件。与扩展(多路径)编译(即对多种尝试进行评估)相比,我们选择的是快速优化编译模式。由于选取的是芯片模式,所以器件引脚的分配在顶层完成。这里,我们选择的是面积优化而不是针对速度的优化。设计网表文件被转化为一个单层的合并网表,取消原有的层次结构(层次结构中每一个模块分别对应网表中的一个不同部分)。

• info, Inferred net 'set' as GSR net

LeonardoSpectrum 选取这个"置位"信号作为全局置位(GSR 代表 Global Set-Reset,即全局置位/复位)资源。Xilinx 具有一个全局布线信号,它可以在不占用器件的一般布线资源的情况下用作置位或复位。通常,这个网线被用于全局复位。

Pass	Area (FGs)	Delay (ns)	DFFs	PIs	POs	--CPU-- min:sec
1	0	7	0	2	2	00:00

这里，我们给出了通道 1 的最优化情况。这个通道产生了 7ns 的时延。此设计不使用 D 触发器，而使用两个输入端口和两个输出端口，并且编译时间为零。当然，所用时间不可能为 0 秒，只是表示编译速度很快而已。

• Info, Added globle buffer BUFG for port reset

除了全局置位/复位资源外，4xxxXL 系列产品还提供了 8 个可用的全局信号(BUFG)。通常它们被用作时钟信号，但 LeonardoSpectrum 可以自动提取复位信号，并把它分配给一个全局缓冲器(Global Buffer)。

• Info, setting outputs in top level view 'INTERFACE' to fast

此模块的输出引脚采用高速输出缓冲器。通常，设计者应当选取低速缓冲器，因为这样可减少能量损耗和降低噪声。

• Using wire table : 4013xl-3 _ avg

在分析中采用的是平均负载。另一种选择就是考虑最恶劣条件下的负载(包括最恶劣的温度、电源供电电压情况下的影响)。缺省模式是针对快速的和条件恶劣的实验室测试环境。缺省模式也可用于对速度大小影响不敏感的场合，例如，用 FPGA 对一个使用更快的技术(如 ASIC)实现的设计进行模拟时，也可以采用此缺省模式。

• IOs　　4　　77　　5.19%

这部分表明仅使用了 4010XL 器件资源的很少一部分。

• Writing file C:/verilog/latch.edf

LeonardoSpectrum 工具的输出是 EDIF 格式的网表文件。Xilinx 的布局和布线工具将使用此网表文件生成器件的配置文件(.bit 文件)。

为了进行 LeonardoSpectrum 的结构配置，可以查阅此工具下方的工具栏。在此工具栏中可以找到一个名为“变量编辑器”(Variable Editor)的标签，其下拉菜单显示了 LeonardoSpectrum 工具的所有的设置。其中一些(像 xlx _ fast _ slew 项，它表示了除非有其他方面的限制，否则设置引脚驱动的缺省方式为工作在快速摆动率下)

在图形用户界面下是不可用的。

6.1.2 在批处理模式下运用 LeonardoSpectrum 工具

如果你熟悉了 LeonardoSpectrum 工具的使用,并希望更快地完成一个设计工作,同时期望设计过程是可重复的和可控的(和使用图形用户界面相比),则你可以在批处理模式下运行 spectrum 可执行程序(此程序在早期的 LeonardoSpectrum 版本中称作 elsyn)。此时要确保在 autoexec.bat 文件 DOS 路径环境设置中正确设置 spectrum 程序的路径。此路径可以是 c: \ exemplar \ LeoSpec \ v1999.1d \ bin \ win32。

例如,可用来对上述基本锁存器设计进行编译的简单命令模式如下:

```
spectrum -source basiclatch.v -edif _ file basiclatch.edf -ta xi4e
```

另一个方法是在图形用户界面的筛选命令窗口中进行剪切和粘贴操作,以构造如程序列表 6-4 所示的类似 basiclatch.run 的文件。

程序列表 6-4 LeonardoSpectrum 可执行脚本文件实例

```
restore_project_script C:/Verilog/verilog/basiclatch.scr
_gc_read_init
_gc_run_init
set input_file_list { "C:/Verilog/verilog/basiclatch.v" }
set part 4013xlPQ160
set process 3
set wire_table 4000xl-default
set pack_clbs FALSE
set timespec_generate FALSE
set nowrite_eqn FALSE
set chip TRUE
set macro FALSE
set area TRUE
set delay FALSE
set report brief
set hierarchy_preserve FALSE
set output_file "C:/Verilog/verilog/basiclatch.edf"
set novendor_constraint_file FALSE
set target xl4xl
_gc_read
_gc_run
```

此文件通过如下命令行方式被调用:spectrum -file basiclatch.scr。键入"spectrum -batchhelp"可以列出所有的命令行选项(类似于程序列表 6-5)。

程序列表 6-5　LeonardoSpectrum 批处理模式命令

-nomap _ global _ bufs
对时钟及其他的全局信号不使用全局缓冲器(Xilinx/Actel)。

-use _ qclk _ bufs
对 Actel 公司的 3200dx 结构使用四相时钟。

-insert _ global _ bufs
对时钟及其他的全局信号应用全局缓冲器(Xilinx/Actel)。

-max _ cap _ load < float >
若在库中被指定,则覆盖缺省的 max _ cap _ load

-max _ fanout _ load < float >
若在库中被指定,则覆盖缺省的 max _ fanout _ load

-lut _ max _ fanout < integer >
针对查找表技术(Xilinx,Altera Flex 系列,Lucent ORCA)指定网线的扇出数。

-noenable _ dff _ map
禁用硬件描述语言 HDL 中的时钟使能检测功能。

-enable _ dff _ map _ optimize
开启从随机逻辑中提取触发器时钟使能的应用。

-exclude < list >
在映射过程中不使用所列出的逻辑门。

-include < list >
映射到指定的同步 D 触发器和 D 型锁存器。

-pal _ device
对 Actel 公司产品,禁用其映射到复杂输入/输出单元的功能。

-wire _ tree < string >
线网树互连:best | balanced | worst = default。

-wire _ table < string >
针对互连时延而采用的线网负载模型。

-nowire _ table
在时延分析中忽略互连产生的延迟时间。

-nobreak _ loops _ in _ delay
不要企图断开组合环路来进行时序分析。

-crit _ path _ analysis _ mode < string >
maximum(报告建立时间冲突) | minimum(报告保持时间冲突) | both = default

-num _ crit _ paths < integer >
报告关键路径的数目,由 < integer > 给出。

-crit _ path _ slack < float >
以纳秒(ns)为单位的驰豫阈值。

-crit _ path _ arrival < float >
以纳秒(ns)为单位的到达阈值。

-crit _ path _ longest
给出最长的路径而不是关键路径。

-crit _ path _ detail < string >
full(点到点的详细描述)(缺省) | short(起点—终点)

-crit _ path _ no _ io _ terminals
不报告终止于原始输出的路径。

-crit _ path _ no _ int _ terminals

不报告终止于内部端点的路径。

-crit _ paths _ from < list >
仅报告以 < list > 所列端口，或者 port _ inst 或 instance 为起点的路径。

-crit _ paths _ to < list >
仅报告以 < list > 所列端口，或者 port _ inst 或 instance 为终点的路径。

-crit _ paths _ thru < list >
仅报告穿过 < list > 所列网线的关键路径。

-crit _ paths _ not _ thru < list >
仅报告不穿过 < list > 所列网线的关键路径。

-crit _ path _ report _ input _ pins
报告门的输入引脚，缺省情况下为 off。

-crit _ path _ report _ nets
报告网线名，缺省情况下为 off。

-nocounter _ extract
禁用计数器的自动提取功能。

-noram _ extract
禁用随机存储器的自动提取功能。

-nodecoder _ extract
禁用译码器的自动提取功能。

-optimize _ cpu _ limit < integer >
设置 CPU 优化限制。

-notimespec _ generate
不从用户约束条件中生成 TIMESPEC 信息。这仅针对 Xilinx 公司的产品。

-nopack _ clbs
不要把查找表放入 CLB 中。仅针对 Xilinx 的 4K 系列产品。

-write _ clb _ packing
打印 CLB 封装(HBLKNM)信息,如果可以的话,以 XNF/EDIF 格式打印。

-crit _ path _ rpt < string >
在此文件中写入关键路径报表。

-nocrit _ path _ rpt
不生成一个关键路径报表文件。

-report _ brief | -report _ full
生成一个简单的设计总结或生成一个详细的设计总结,缺省值为 full。

-map _ area _ weight < float >
一个介于 0 和 1.0 之间的数。该数越大,用于最小化面积的映射就越多。

-map _ delay _ weight < float >
一个介于 0 和 1.0 之间的数。该数越大,用于最小化时延的映射就越多。

-simple _ port _ names
对向量端口进行简单的命名:使用%s%d 形式,而不使用%s(%d)形式。

-bus _ name _ style < string >
向量端口和网线的命名格式:default%s(%d) | simple%s%d | old _ galileo%s _%d。

-nobus
以扩展方式写总线。在 Xilinx 的 EDIF 阅读器中可能需要。

-nowrite _ eqn
在输出中不写入等式,而是使用技术原语。

-nopld _ xor _ decomp
对 Altera 的 MAX 系列及 Xilinx 的 CPLD 技术,不执行异或分解操作。

-noglobal _ symbol
删除启动(全局置位/复位)块。

-notime _ opt
不执行时序优化。

-max _ frequency < float >
预期的最高工作频率，单位为 MHz。

-edifin _ ground _ net _ names < list >
指定由 < list > 所列出的网线为接地网线。

-edifin _ power _ net _ names < list >
指定由 < list > 所列出的网线为电源网线。

-edifin _ ground _ port _ names < list >
指定由 < list > 所列出的端口为接地端口。

-edifin _ power _ port _ names < list >
指定由 < list > 所列出的端口为电源端口。

-edifin _ ignore _ port _ names < list >
指定由 < list > 所列出的端口为无效端口。

-edifout _ power _ ground _ style _ is _ net
写出无驱动的电源和接地网线名，它采用提取的或隐含的网线名来命名。

-edifout _ power _ net _ name < string >
当 'edifout _ power _ ground _ style _ is _ net' 值为 **TRUE** 时，使用 < string > 所给出的名称对电源网线进行命名，缺省为 'VCC'。

-edifout _ ground _ net _ name < string >
当 'edifout _ power _ ground _ style _ is _ net' 值为 **TRUE** 时，使用 < string > 所给出的名称对接地网线进行命名，缺省为 'GND'。

6.2 完整的设计流程,8 位相等比较器

从第一章到现在,我们仅完成了 FPGA 一半的设计任务,即设计输入和逻辑综合。对于完整的设计,还需运行 Xilinx 的布局/布线工具和设计管理器。为了介绍这些工具的使用方法,我们采用一个具体的设计实例贯穿整个叙述过程。这个实例和一个 8 位相等比较器——HC688 类似。这个设计对两个字节进行比较,如果相同,则产生一个"相等"(equal)信号。另外,采用一个级联输入方式以扩展被比较的输入信号;此种情况下,若级联方式没有被声明,则"相等"输出信号将无法给出。出于作者个人的喜好,对原有设计进行了部分的改变,即所有的信号均为高有效,并且同步"相等"(equal)输出信号。此设计的 Verilog 代码实现方法见程序列表6-6。

程序列表 6-6 8 位相等比较器

```
// 同步的 8 位相等比较器。
// 所有信号均被变换为高有效。
// 输出同步。

module hc688s (equal, clock, reset, cascade, a, b);
output          equal;
input           clock, reset;
input           cascade;
input   [7:0]    a, b;
reg              equal;

always @ (posedge clock or posedge reset)
    begin
    if (reset)
    equal    < = 0;
    else if ( ~ cascade)
    equal    < = 0;
    else if (a = = b)
    equal    < = 1;
    else
    equal    < = 0;    // 确保覆盖所有可能的输入。
```

```
end
endmodule
```

Verilog 代码是十分简洁的。“相等”(equal)信号仅仅在级联信号为高且 a,b 两输入字节相等时才变为高电平。让我们来看看 LeonardoSpectrum 工具是如何处理这个设计的,图 6-1 是部分示意图。

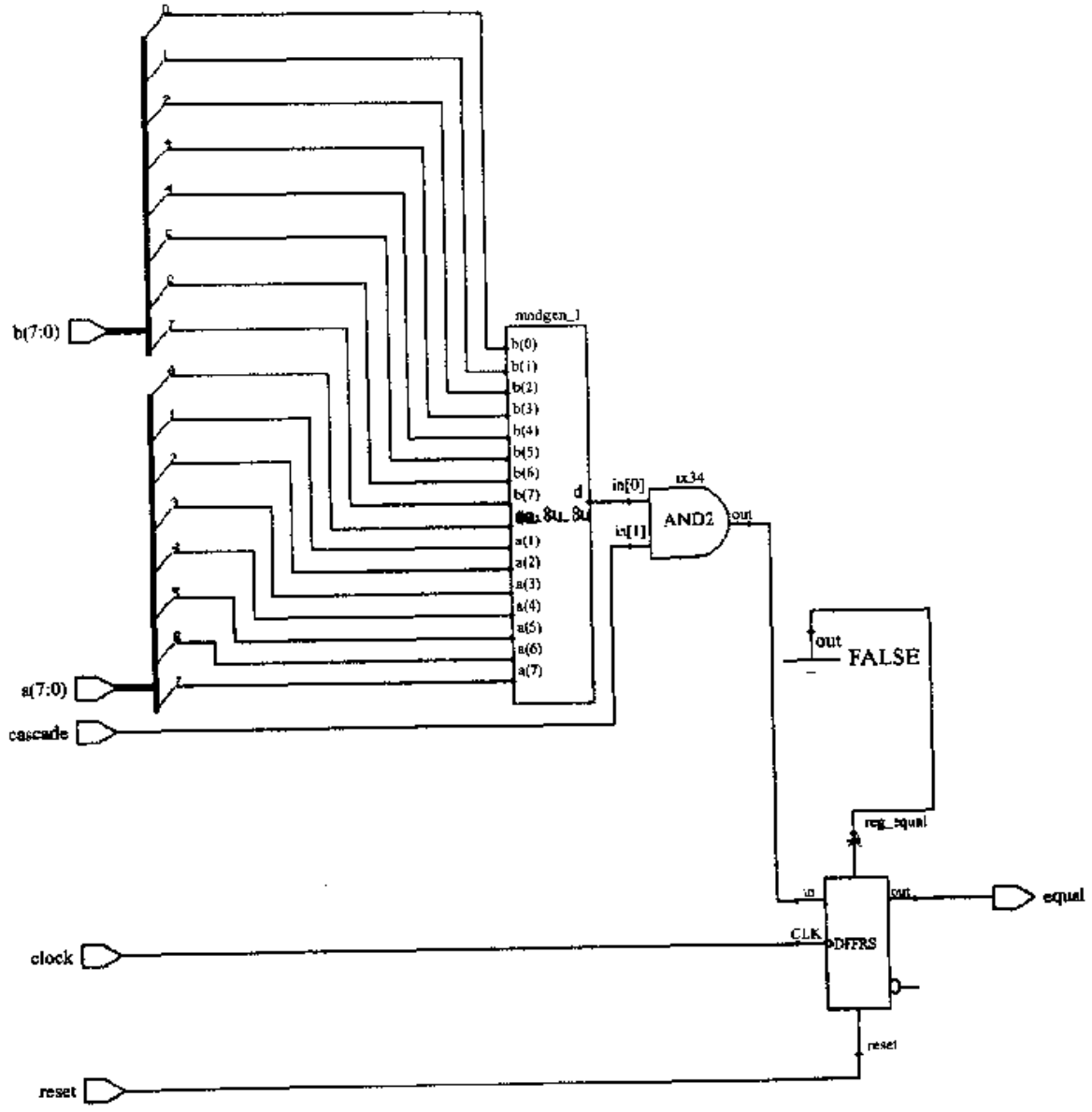

图 6-1　HC688s 的 LeonardoSpectrum 寄存器传输级(RTL)电路示意图

从图 6-1 可以看出,“相等”(equal)输出信号是由触发器实现的,并且时钟和复位信号均按预期的形式被实现。LeonardoSpectrum 使用模块生成器(modgen)来说明了一个元件库的作用,在这里,模块生成器被用来实现相等测试的逻辑功能。为了

解更多的设计细节，LeonardoSpectrum 提供了另一个示意图查看选项，即门级示意图，参见图 6-2。

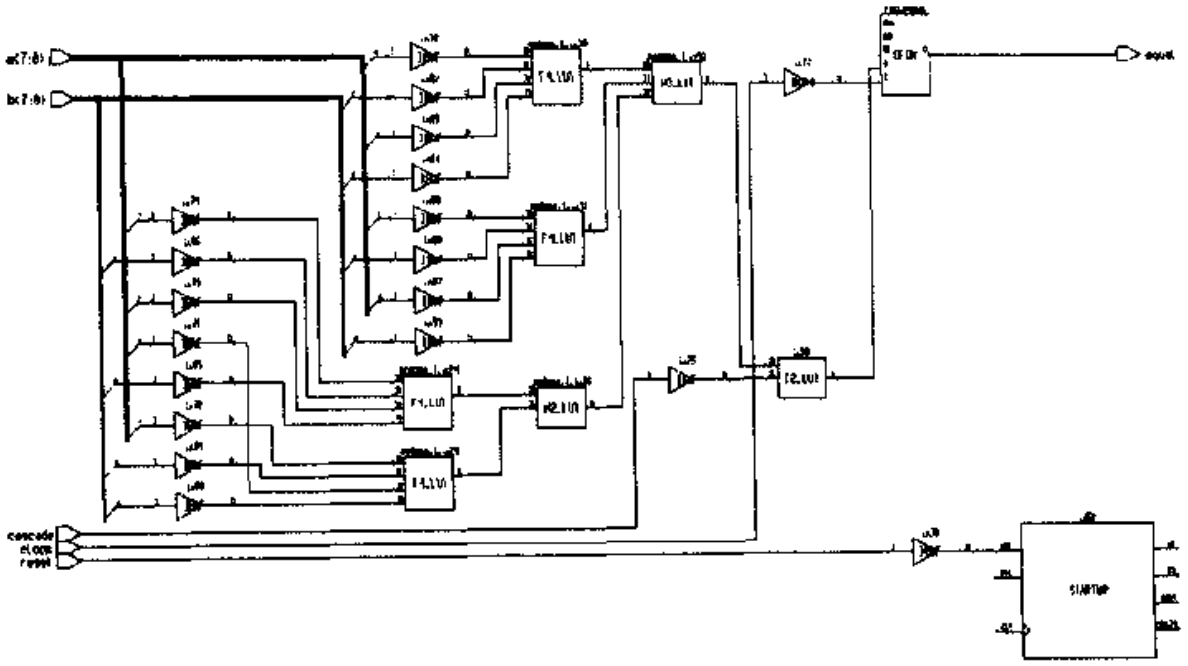

图 6-2 HC688s 的 LeonardoSpectrum 门级电路示意图

门级示意图给出了当映射到 Xilinx 硬件时的实际逻辑形式。有关 Xilinx 的可配置逻辑块(CLB)的相关内容将在第七章中详细介绍。因此，目前我们所关注的是以下三方面：设计中双输入及四输入查找表的分配问题；对时钟和复位信号使用全局缓冲器的问题；驱动“相等”(equal)输出信号的触发器的问题。

对此示意图有几个问题值得注意。首先，此图是设计网表的图形化表示形式，此网表是由 LeonardoSpectrum 生成的。设计上仍有一些需由 Xilinx 的设计管理器(Design Manager)(布局/布线工具)来完成。使用示意图是为了在核查时可以做到一目了然，如果设计没能被进行有效的逻辑综合处理，设计者可以尝试采用不同的编译方式或使用更结构化的方法来进行设计。例如，设计者可以用结构化形式的逻辑门来代替一个高层逻辑相等运算符(= =)，这可以使得设计综合过程更加清晰。

LeonardoSpectrum 提供了对示意图的另一种表示方式，其关键路径如图 6-3 所示。

此关键路径是设计中存在的最长时延路径。如果设计需要进行优化以得到更高的速度的话，设计者应当把重点集中在对这条路径的重新设计上，其目标是去掉逻辑层次。从图中可以看到，最长的路径是从 b[4]输入到“相等”(equal)输出，并且在此路径上分为四个逻辑层次。如同我们早期学过的加法器一样，可以找到一

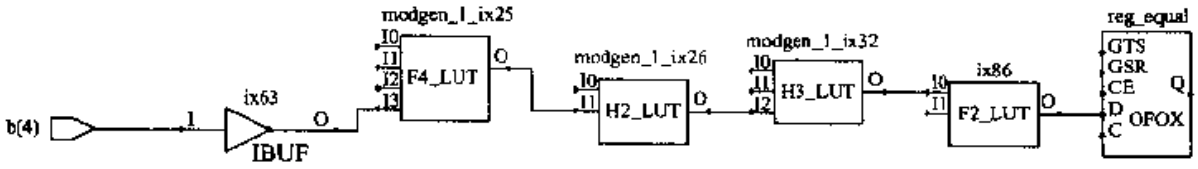

图 6-3　HC688s 的 LeonardoSpectrum 关键路径电路示意图

种方法，即在必要时加入额外的逻辑，以实现逻辑的预测和并行流水处理。

针对时延的优化编译并不会给原设计带来任何改动。但对于大多数设计来说，在设计的说明部分将会产生一个变化，并且往往是一个有益的变化。

逻辑综合器的输出是一个设计网表，此处为 EDIF(.edf)格式文件，但此格式的文件不适宜人工阅读。LeonardoSpectrum 也可以生成以 Verilog 语言格式描述的结构性设计网表。事实上，LeonardoSpectrum 的一大特点就是具有在不同类型的设计网表之间进行转化的能力。毕竟，我们正在学习 Verilog 语言，因此在程序列表 6-7 中给出了这个网表的 Verilog 语言描述形式。

程序列表 6-7　8 位相等比较器的结构化设计网表

```
//
// 对 hc688s 单元的 Verilog 语言描述。
// 09/06/99 11:00:40。
//

module hc688s ( equal, clock, reset, cascade, a, b ) ;

    output equal ;
    input clock ;
    input reset ;
    input cascade ;
    input [7:0]a ;
    input [7:0]b ;

    wire nx12, modgen _ eq _ 2 _ nx21, modgen _ eq _ 2 _ nx22, modgen _ eq
        _ 2 _ nx23, modgen _ eq _ 2 _ nx28, modgen _ eq _ 2 _ nx29, clock _
        int, reset _ int, cascade _ int, a _ 7 __ int, a _ 6 __ int, a _ 5 __ int,
        a _ 4 __ int, a _ 3 __ int, a _ 2 __ int, a _ 1 __ int, a _ 0 __ int, b _
```

```
    7__int, b_6__int, b_5__int, b_4__int, b_3__int, b_2
    __int, b_1__int, b_0__int, nx15;
wire [8:0] \$dummy;

assign modgen_eq_2_nx22 = (~a_7__int && ~b_7__int &&
    ~a_6__int && ~b_6__int) || (~a_7__int && ~b_7
    __int && a_6__int && b_6__int) || (a_7__int && b_7__
    int && ~a_6__int && ~b_6__int) || (a_7__int && b_7
    __int && a_6__int && b_6__int);

assign modgen_eq_2_nx23 = (~a_5__int && ~b_5__int &&
    ~a_4__int && ~b_4__int) || (~a_5__int && ~b_5
    __int && a_4__int && b_4__int) || (a_5__int && b_5__
    int && ~a_4__int && ~b_4__int) || (a_5__int && b_
    5__int && a_4__int && b_4__int);

assign modgen_eq_2_nx21 = (modgen_eq_2_nx22 && modgen_eq
    _2_nx23);

assign modgen_eq_2_nx28 = (~a_3__int && ~b_3__int &&
    ~a_2__int && ~b_2__int) || (~a_3__int && ~b_3
    __int && a_2__int && b_2__int) || (a_3__int && b_3__
    int && ~a_2__int && ~b_2__int) || (a_3__int && b_3
    __int && a_2__int && b_2__int);

assign modgen_eq_2_nx29 = (~a_1__int && ~b_1__int &&
    ~a_0__int && ~b_0__int) || (~a_1__int && ~b_1
    __int && a_0__int && b_0__int) || (a_1__int && b_1__
    int && ~a_0__int && ~b_0__int) || (a_1__int && b_1
    __int && a_0__int && b_0__int);

assign nx12 = (modgen_eq_2_nx28 && modgen_eq_2_nx29 &&
    modgen_eq_2_nx21);

STARTUP ix63 (.Q2(\$dummy[0]), .Q3(\$dummy[1]),
```

```
.Q1Q4 ( \ $ dummy [2]), .DONEIN ( \ $ dummy [3]), .GSR (reset _
int), .GTS ( \ S dummy [4]), .CLK
      ( \ $ dummy [5])) ;
    IBUF b _ 0 __ ibuf ( .O (b _ 0 __ int), .I (b[0])) ;
    IBUF b _ 1 __ ibuf ( .O (b _ 1 __ int), .I (b[1])) ;
    IBUF b _ 2 __ ibuf ( .O (b _ 2 __ int), .I (b[2])) ;
    IBUF b _ 3 __ ibuf ( .O (b _ 3 __ int), .I (b[3])) ;
    IBUF b _ 4 __ ibuf ( .O (b _ 4 __ int), .I (b[4])) ;
    IBUF b _ 5 __ ibuf ( .O (b _ 5 __ int), .I (b[5])) ;
    IBUF b _ 6 __ ibuf ( .O (b _ 6 __ int), .I (b[6])) ;
    IBUF b _ 7 __ ibuf ( .O (b _ 7 __ int), .I (b[7])) ;
    IBUF a _ 0 __ ibuf ( .O (a _ 0 __ int), .I (a[0])) ;
    IBUF a _ 1 __ ibuf ( .O (a _ 1 __ int), .I (a[1])) ;
    IBUF a _ 2 __ ibuf ( .O (a _ 2 __ int), .I (a[2])) ;
    IBUF a _ 3 __ ibuf ( .O (a _ 3 __ int), .I (a[3])) ;
    IBUF a _ 4 __ ibuf ( .O (a _ 4 __ int), .I (a[4])) ;
    IBUF a _ 5 __ ibuf ( .O (a _ 5 __ int), .I (a[5])) ;
    IBUF a _ 6 __ ibuf ( .O (a _ 6 __ int), .I (a[6])) ;
    IBUF a _ 7 __ ibuf ( .O (a _ 7 __ int), .I (a[7])) ;
    IBUF cascade _ ibuf ( .O (cascade _ int), .I (cascade)) ;
    IBUF reset _ ibuf ( .O (reset _ int), .I (reset)) ;
    OFDX reg _ equal ( .Q (equal), .C (clock _ int), .D (nx15), .CE
( \ $ dummy [6]), .GSR ( \ $ dummy [7]), .GTS ( \ $ dummy [8])) ;

    BUFG clock _ ibuf ( .O (clock _ int), .I (clock)) ;

    assign nx15 = (nx12 && cascade _ int) ;
endmodule
```

以上的描述看起来有些杂乱，但从中可以得出一些有用的东西。注意附加在内部信号上的_ int 标志。这样表示是一种友好的方式，因为一些逻辑综合器可能会把一个有用信号名如时钟转化成名为 ifght _ 2746 的信号，而不是 clock _ int，这使得查找设计网表变得十分困难。当一个信号参与布线时，我们期望所用的综合器能够使该信号与其他信号隔离，但同时又希望保留我们在其他地方曾经使用过的信号名中的一部分。相等模块是 modgen _ 2，将它连接到输入缓冲器(ibufs)。“相

等"寄存器是一个输出 D 触发器(OFDX);注意对 Q 输出端及时钟/数据/时钟使能端的分配方式。GTS 代表全局三态控制,GSR 代表全局置位/复位控制。

布局/布线工具的运行是以从输入设计中提取的设计网表为基础的,并且要受到设计约束条件及逻辑综合控制的影响。若综合后的逻辑存在某些问题的话,则该工具可以用来分析该网表文件,并确保以一个合理的方式对设计进行逻辑综合。

网表文件的另一种格式是.xnf(Xilinx 网表格式)形式,它具有很好的可读性。但令人遗憾的是,Xilinx 公司却想把 EDIF 格式标准化,其实 EDIF 格式的可读性很差。

6.3 使用层次设计法设计 8 位相等比较器

下面让我们来看看对于相等比较器使用层次设计法后,其所得的设计网表会有什么不同。程序列表 6-8 所示的 hier688 设计说明了如何使用三个 hc688s 来构造一个 24 位的地址译码器。

程序列表 6-8 8 位相等比较器的层次设计实例

```
module hier688(chip_select, output_enable, addr, rwn, clock, reset);
output    chip_select, output_enable;
input   [23:0] addr;
input          rwn, clock, reset;
wire           low, middle, high;
reg            chip_select, output_enable;
parameter      low_range = 8'h80;
parameter      mid_range = 8'ha0;
parameter      high_range = 8'hff;

// 对于低位地址比较器,把级联输入端捆绑在一起。
hc688s u1 (low, clock, reset, 1'b1, addr[7:0], low_range);
hc688s u2 (middle, clock, reset, low, addr[15:8], mid_range);
hc688s u3 (high, clock, reset, middle, addr[23:16], high_range);

//使模块输出同步。
always @ (posedge clock or posedge reset)
    begin
```

```
    if (reset)
        begin
        chip _ select          < = 0;
        output _ enable  < = 0;
        end
    else
        begin
        chip _ select          < = high;
        output _ enable  < = (high & ~rwn);
        end
  end
endmodule
```

图 6-4 所示的示意图逻辑关系不是非常清晰，但从中可以看到 HC688 解码器的结构层次所形成的级联逻辑形式。此设计的工作速度不是很快，但由于对原有 HC688 模块的重复使用，因此易于进行组合。我们不打算分析其关键路径，但很明显，其路径是从低阶地址输入端到"输出使能"输出信号端。

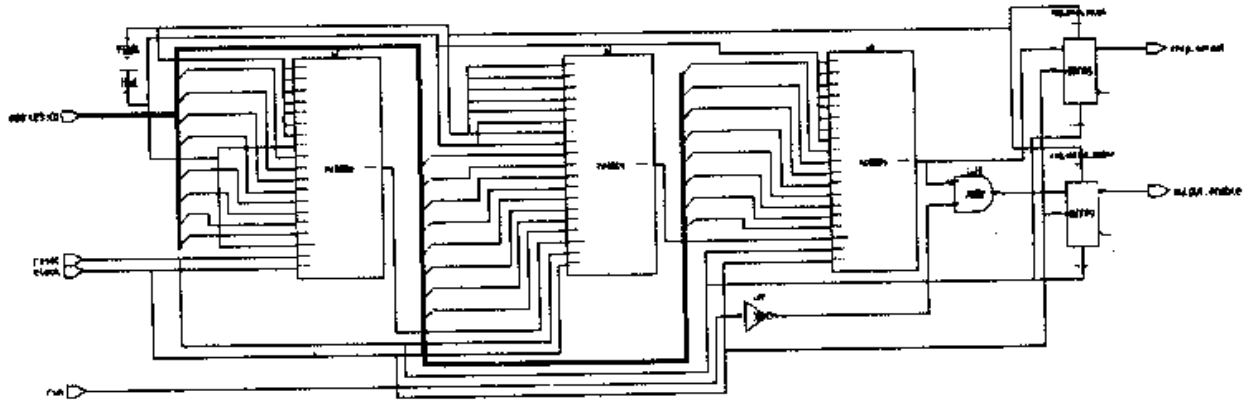

图 6-4　分层次的 HC688s 门级电路示意图

下面讨论在实际的器件上实现此设计的问题。首先，要对设计进行布局和布线处理，生成应用在最终 Xilinx 器件上的配置文件。然后，打开设计管理器，创建一个新的任务（见图 6-5），通过浏览，查找到 hier688.edf 网表文件。设计管理器提供点击操作。下面，我们仅执行缺省方式的设计管理器流程，看看将得到什么样的结果。

开启设计管理器程序的一个快速方法是在 windows 桌面上建立一个快捷方式。例如，命令行可以是：C：\ Xilinx \ bin \ nt \ dsgnmgr.exe。

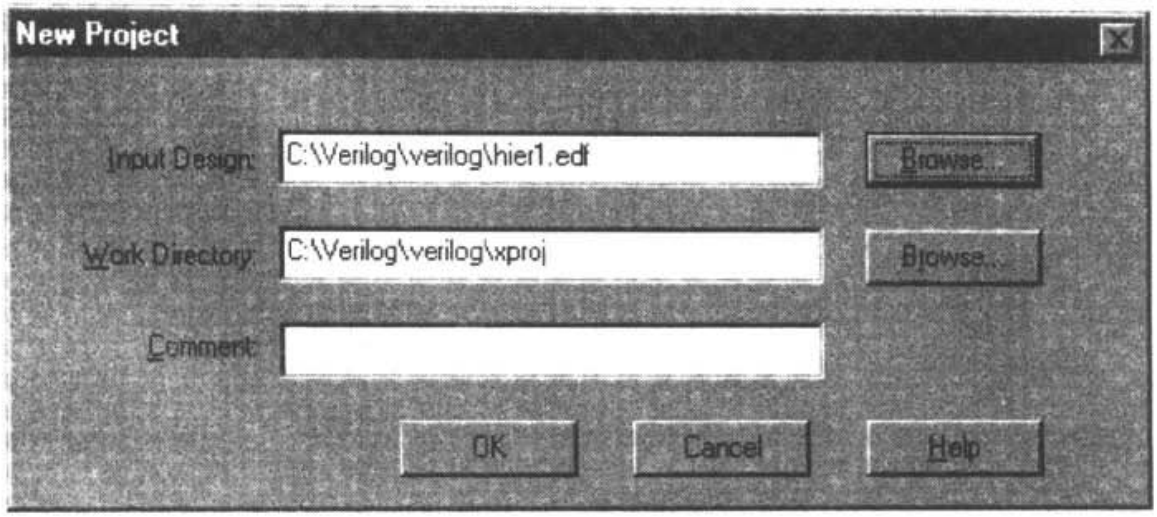

图 6-5 使用 Xilinx 的设计管理器打开一个设计

程序列表 6-9 8 位相等比较器分层设计实例,Xilinx 转换报表

```
ngdbuild:  version M1.5.19
Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Command Line: ngdbuild -p xc4010xl-3-pq100 -dd ..
C:\Verilog\SourceCode\hier688.edf hier688.ngd

Launcher: Executing edif2ngd "C:\Verilog\SourceCode\hier688.edf"
"C:\Verilog\SourceCode\xproj\ver1\hier688.ngo"
Reading NGO file "C:/Verilog/SourceCode/xproj/ver1/hier688.ngo"
...
Reading component libraries for design expansion...

Checking timing specifications ...

Checking expanded design ...

NGDBUILD Design Results Summary:
  Number of errors:     0
  Number of warnings:   0

Writing NGD file "hier688.ngd" ...

Writing NGDBUILD log file "hier688.bld"...
```

图 6-6 显示的是报表浏览器窗口。选中 Translation Report 时,将会显示如程序列表 6-9 所示的报表内容,从中可以看到输入设计被正确读取,EDIF 格式网表被转换成了 Xilinx 的二进制网表格式文件(扩展名为 .ngo 文件)。

程序列表 6-10 截取了 Xilinx 布局/布线报表的一部分。类似于印刷电路板自动布线系统,此布局/布线工具可以做到对不同的布局方案进行尝试,并从中选取

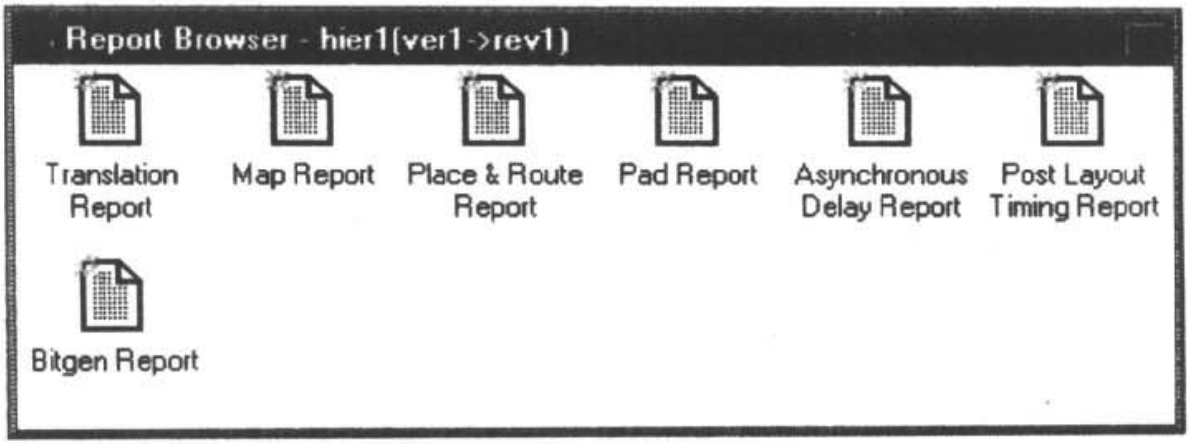

图 6-6　设计管理器报表

一个最佳的方式，从中我们可以知道时序是怎样的。

程序列表 6-10　8 位相等比较器的分层设计实例，Xilinx 布局/布线报表

```
Starting Constructive Placer.  REAL time: 7 secs
Placer score = 13350
Placer score = 9810
Placer score = 6780
Placer score = 5730
Placer score = 5190
Placer score = 4440
Placer score = 3720
Placer score = 3570
Placer score = 3480
Placer score = 3270
Placer score = 3090
Finished Constructive Placer.  REAL time: 7 secs
```

程序列表 6-11　8 位相等比较器分层设计实例，Xilinx 器件平均时延报表

```
The Number of signals not completely routed for this design is: 0

   The Average Connection Delay for this design is:        1.929 ns
   The Average Connection Delay on critical nets is:       0.000 ns
   The Average Clock Skew for this design is:              0.098 ns
   The Maximum Pin Delay is:                               5.937 ns
   The Average Connection Delay on the 10 Worst Nets is:   2.983 ns

   Listing Pin Delays by value: (ns)

d <= 10   < d <= 20   < d <= 30   < d <= 40   < d <= 50   d > 50
-------   ---------   ---------   ---------   ---------   -------
   37           0           0           0           0         0
```

信号的时延可以从程序列表 6-11 中体现。本设计的运行速度较快(看起来它将以 100MHz 运行),其原因是它所用到的器件资源很少。用到的器件资源越多,需要占用布线资源的逻辑越多时,运行速度将会越慢。

程序列表 6-12 8 位相等比较器分层设计实例,Xilinx 引脚定义报表

```
# Pinout constraints listing
# These constraints are in PCF grammar format
# and may be cut and pasted into the PCF file
# after the "SCHEMATIC END ;" statement to
# preserve this pinout for future design iterations.
#
COMP "addr(0)" LOCATE = SITE "P90" ;
COMP "addr(1)" LOCATE = SITE "P89" ;
COMP "addr(10)" LOCATE = SITE "P36" ;
COMP "addr(11)" LOCATE = SITE "P35" ;
COMP "addr(12)" LOCATE = SITE "P37" ;
COMP "addr(13)" LOCATE = SITE "P39" ;
COMP "addr(14)" LOCATE = SITE "P44" ;
COMP "addr(15)" LOCATE = SITE "P42" ;
COMP "addr(16)" LOCATE = SITE "P32" ;
COMP "addr(17)" LOCATE = SITE "P22" ;
COMP "addr(18)" LOCATE = SITE "P30" ;
COMP "addr(19)" LOCATE = SITE "P31" ;
COMP "addr(2)" LOCATE = SITE "P93" ;
COMP "addr(20)" LOCATE = SITE "P23" ;
COMP "addr(21)" LOCATE = SITE "P21" ;
COMP "addr(22)" LOCATE = SITE "P24" ;
COMP "addr(23)" LOCATE = SITE "P33" ;
COMP "addr(3)" LOCATE = SITE "P95" ;
COMP "addr(4)" LOCATE = SITE "P97" ;
COMP "addr(5)" LOCATE = SITE "P94" ;
COMP "addr(6)" LOCATE = SITE "P88" ;
COMP "addr(7)" LOCATE = SITE "P96" ;
COMP "addr(8)" LOCATE = SITE "P38" ;
COMP "addr(9)" LOCATE = SITE "P43" ;
COMP "chip_select" LOCATE = SITE "P20" ;
COMP "clock" LOCATE = SITE "P5" ;
COMP "output_enable" LOCATE = SITE "P18" ;
COMP "reset" LOCATE = SITE "P56" ;
COMP "rwn" LOCATE = SITE "P17" ;
```

在输入设计中不进行引脚位置的分配。在进行第一次引脚分配时,使用布局/布线工具不失为一个好的方法(特别是对 Altera 的器件)。FPGA 设计旨在以通用的模式分配引脚(即对设计者自定义的引脚使用不敏感方式,且任一个 I/O 引脚均可被芯片上的任一个逻辑所使用),但也可以做一些事先的约定,例如,数据流为水平方向(和器件上引脚 1 的位置有关),控制信号为垂直方向。另外,对于 PWB(印刷线路板)设计,你可能希望控制引脚的位置,尽量使地址线集中布置,对其他形式的布线也类似处理。一旦电路板的设计工作完成后,我们就不希望编译器对引脚

再进行重新分配，因此需要限制引脚的位置。由 Xilinx 的布局/布线工具分配的引脚的位置可以通过如程序列表 6-12 所示的引脚报表予以确定。此文件可以被剪切、粘贴并编辑到 LeonardoSpectrum 的限制文件中，从而确定引脚的分配，如程序列表 6-13 所示。此工作也可以在 Xilinx 的设计管理器中完成，但我更愿意在设计开始阶段中确定这些引脚的定义。

程序列表 6-13　8 位相等比较器分层设计实例，Xilinx 引脚分配表

```
addr(0)          INPUT        P90
addr(1)          INPUT        P89
addr(10)         INPUT        P36
addr(11)         INPUT        P35
addr(12)         INPUT        P37
addr(13)         INPUT        P39
addr(14)         INPUT        P44
addr(15)         INPUT        P42
addr(16)         INPUT        P32
addr(17)         INPUT        P22
addr(18)         INPUT        P30
addr(19)         INPUT        P31
addr(2)          INPUT        P93
addr(20)         INPUT        P23
addr(21)         INPUT        P21
addr(22)         INPUT        P24
addr(23)         INPUT        P33
addr(3)          INPUT        P95
addr(4)          INPUT        P97
addr(5)          INPUT        P94
addr(6)          INPUT        P88
addr(7)          INPUT        P96
addr(8)          INPUT        P38
addr(9)          INPUT        P43
chip_select      OUTPUT       P20
clock            INPUT        P5
output_enable    OUTPUT       P18
reset            INPUT        P56
rwn              INPUT        P17
```

这些引脚也可以在 LeonardoSpectrum 环境下进行分配，即打开 Constraints 对话框，找到 Input 和 Output 标签，在 Pin Location 输入框中填写相应数值。当所有引脚分配值都填写好后（图 6-7），点击 Apply 按钮确认。这些操作也可以通过批处理模式来完成，如下所示：

```
set_attribute -port { <hierarchical net name> } -name PIN_NUMBER -value PXX
```

注意：XX 表示实际的引脚数值。

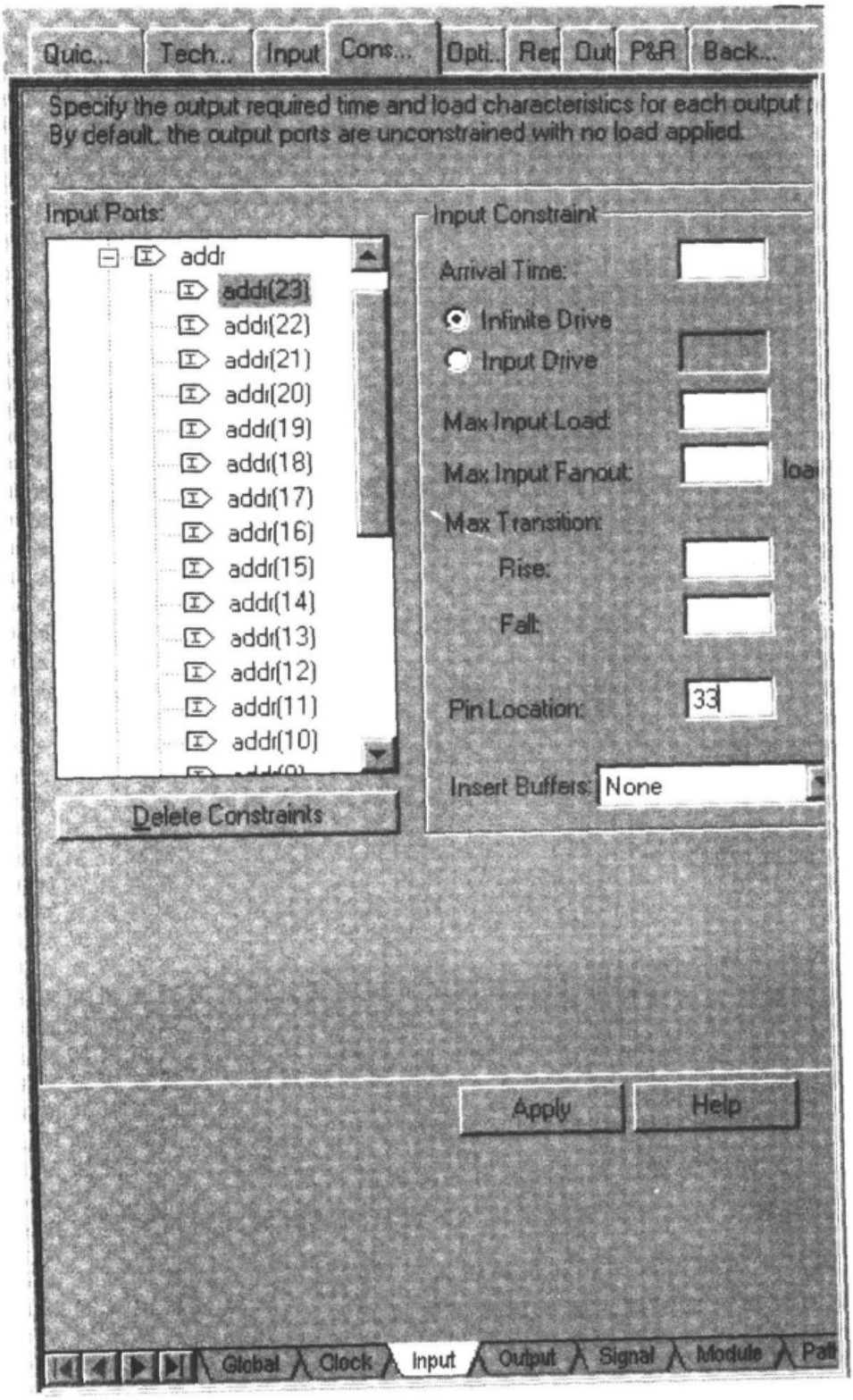

图 6-7 使用图形用户界面的 LeonardoSpectrum 引脚分配

这些并不是电路板上需要进行引脚分配的全部。在电路板极示意图上，我们必须为供电电源，接地及配置信号等专用信号进行引脚的分配。

程序列表 6-14　8 位相等比较器分层设计实例，Xilinx 的异步时延报表

```
The 20 Worst Net Delays are:
------------------------------
| Max Delay (ns)   | Netname    |
------------------------------
   5.937             low
   4.154             middle
   3.314             high
   2.751             clock_int
   2.508             reset_int
   2.490             addr(23)_int
   2.228             addr(17)_int
   2.187             addr(21)_int
   2.179             addr(4)_int
   2.097             addr(1)_int
   2.085             addr(7)_int
   1.823             addr(14)_int
   1.823             addr(10)_int
   1.767             addr(9)_int
   1.754             addr(22)_int
   1.739             addr(0)_int
   1.705             addr(15)_int
   1.693             addr(11)_int
   1.637             addr(6)_int
   1.557             addr(16)_int
```

以上的 20 个时延数据可以从程序列表 6-14 所示的异步时延报表中体现。从中可以推测出此设计运行的工作频率为 168MHz，这对于一个低速的 -3 速度级器件来说已经是不错了。另外，我们所用到的仅仅是器件资源的很小一部分。并且，这个所得的时延并不是完整的，它只是单个节点之间的时延而已。为了得到完整的时延数据，我们不得不利用以上数据执行完整的时序分析，结果如下：

```
Timing constraint: Default period analysis
 34 items analyzed, 0 timing errors detected.
 Minimum period is   9.967ns.

Delay:      9.967ns low to middle (8.027ns delay plus 1.940ns
setup)
Path low to middle contains 2 levels of logic:
Path starting from Comp: CLB_R1C10.K (from clock_int)
To          Delay type          Delay(ns)      Physical
                                               Resource
Logical Resources
-------------------------------------------------  --------
CLB_R1C10.XQ       Tcko                  2.090R     low
                                                    ul_reg_equal
CLB_R20C10.C2      net (fanout=1)        5.937R     low
```

```
CLB_R20C10.K      Thh1ck              1.940R          middle
modgen_eq_3_ix18                                    u2_reg_equal
-------------------------------------------------
Total (4.030ns logic, 5.937ns route)        9.967ns (to clock_int)
      (40.4% logic, 59.6% route)
```

从以上的显示内容可以看到,最坏情况下,从一个触发器到另一个触发器之间的时延为 9.967ns,因此时钟频率只要设置在 100MHz 就可以了。

6.4 Xilinx 环境下的优化选项

Xilinx 的布局/布线工具(Design Manager,设计管理器)的作用是把 EDIF 格式的网表文件转换为可以下载到最终目标器件上的配置文件。布局/布线工具中的部分优化参数允许设计者自行设置。进入 options 菜单的方法是从 Implement 菜单(图 6-8)中点击 options 按钮。

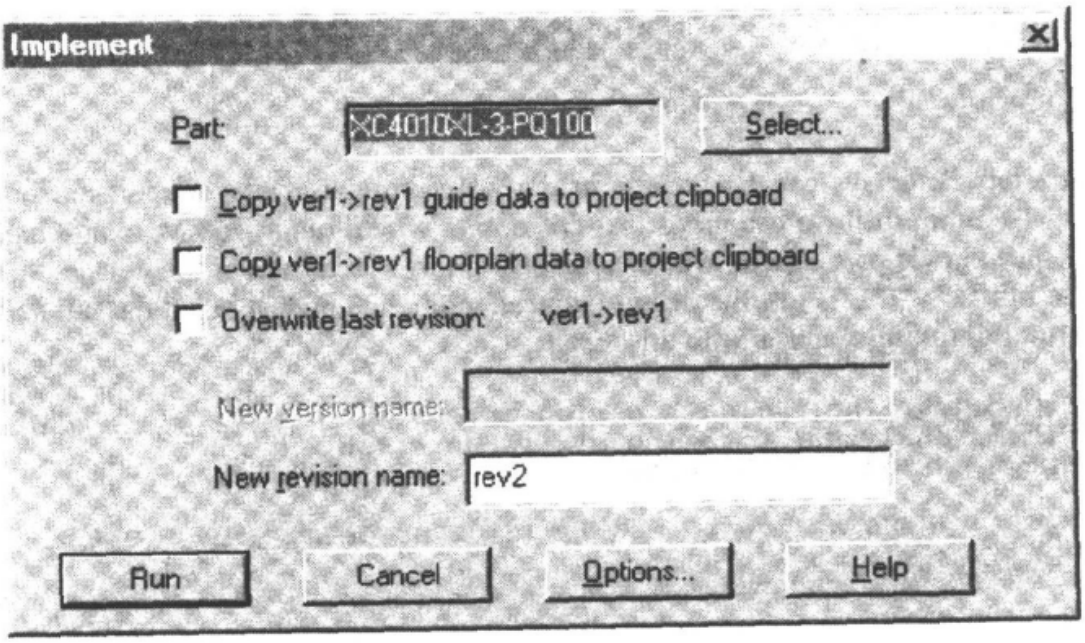

图 6-8 Xilinx 设计管理器选项

6.5 映 射 选 项

在逻辑综合后的网表文件中为预编译的库单元留有一些占位符。映射器会找到库单元(.ngo 文件,二进制网表格式)并把它们合并入原文件中。然后,映射器将合并后的网表转换为物理网表,在此物理网表中,所有的网表逻辑单元都和具体的硬件单元相对应。映射器的输出是一个.ncf 文件(物理网表格式)。用户可以用

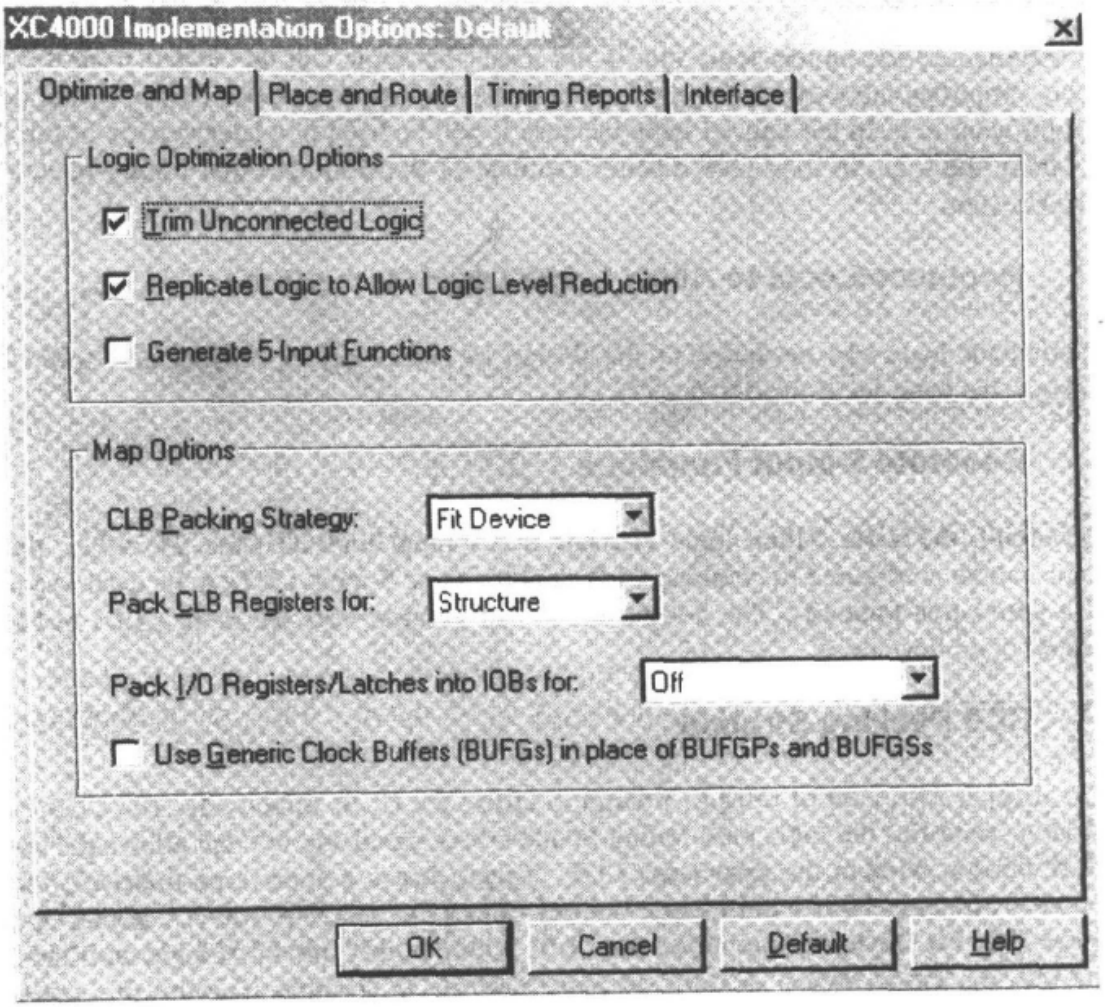

图 6-9　Xilinx 设计管理器执行选项

如图 6-9 所示的 Implementation Option 窗口中的相应选项对映射过程进行配置。下面分别对窗口中的各个复选框选项做一些解释。

Trim Unconnected Logic（裁减无用的逻辑）　若映射器遇到没有被使用到的逻辑，则此逻辑可以从设计中删除。这将使逻辑变得简单并可提高布局/布线的执行速度。然而，设计者可能希望保留这些没有使用到的逻辑，因为它们可能会在此设计的后续版本中被用到。保留这个逻辑可能有助于获得最终设计在资源及时序方面的更好的评估结果。

Replicate Logic to Allow Logic Level Reduction（复制逻辑以实现逻辑层次的缩减）　设计中可以加入冗余逻辑以减轻驱动器负载，提高设计的执行速度（这是基本的面积/速度折中策略）。

Generate 5-input Function（生成五输入的函数） 通常，基本的 Xilinx 逻辑单元是一个四输入的查找表。然而，CLB 逻辑可以被配置组合以构成五输入的查找表。这也是一个有关面积/速度的折中的方法。五输入的 CLB 配置允许使用更多的 CLB，但允许更高的运行速度。

CLB Packing Strategy（CLB 整合策略） 通过应用一套规则，映射器可以有效的利用 CLB 组。CLB 整合策略可以修改逻辑的划分方法，以减少公共信号的数量并允许使用和查找表无关的 CLB 触发器。这里采用的同样是一种速度/面积折中的方法。CLB 整合策略可以使用更多的逻辑，但要求工作在更高的速度上。Fit Device 选项把可能无关的逻辑和 CLB 整合在一起，直至目标硬件和设计相匹配或再无其他东西可以归整在一起为止。此选项(CLB Packing Strategy)若设为 off，则仅仅允许将相关的逻辑(指具有公共输入的逻辑)整合进一个 CLB 中。

Pack CLB Registers for Minimum Area or Structure（为使面积或结构最小化而整合 CLB 寄存器） 此选项除了可以用来分辨相连的信号名称外，还可以用来控制寄存器的排序。使用 Minimum Area 选项将得到一个更加紧凑的设计，它将以一个更加随机的方式进行寄存器的映射。Structure 选项将开启寄存器排序分析功能。

Pack I/O Registers/Latches into IOBs for Input only, Output only, Inputs and Outputs, and Off(整合 I/O 寄存器/锁存器入 IOB 中，可选项为：输入，输出，输入和输出，关闭） 正常情况下，从逻辑单元到 I/O 缓冲器(IOB)的分配工作由综合工具完成。但此选项允许使用映射器对 IOB 进行分配，这将生成一个功能更强大的 CLB。若选择 Off，则改由综合工具控制 IOB 的分配。

Use Generic Clock Buffers(BUFGs) in Place of BUFGPs and BUFGSs(使用通用时钟缓冲器代替一级和二级全局缓冲器） 较早的 Xilinx 器件在处理全局信号时使用的是一级(BUFGP)和二级(BUFGS)全局缓冲器，因此一些综合工具可以进行这方面的处理。较新的 Xilinx 器件使用的是通用全局缓冲器(BUFG)。这样的选择表示允许使用 BUFG 来代替 BUFGS 和 BUFGP。

6.6 布局/布线选项

Place & Route Effort Level（布局/布线效果等级） 在优化设计时所花费的时

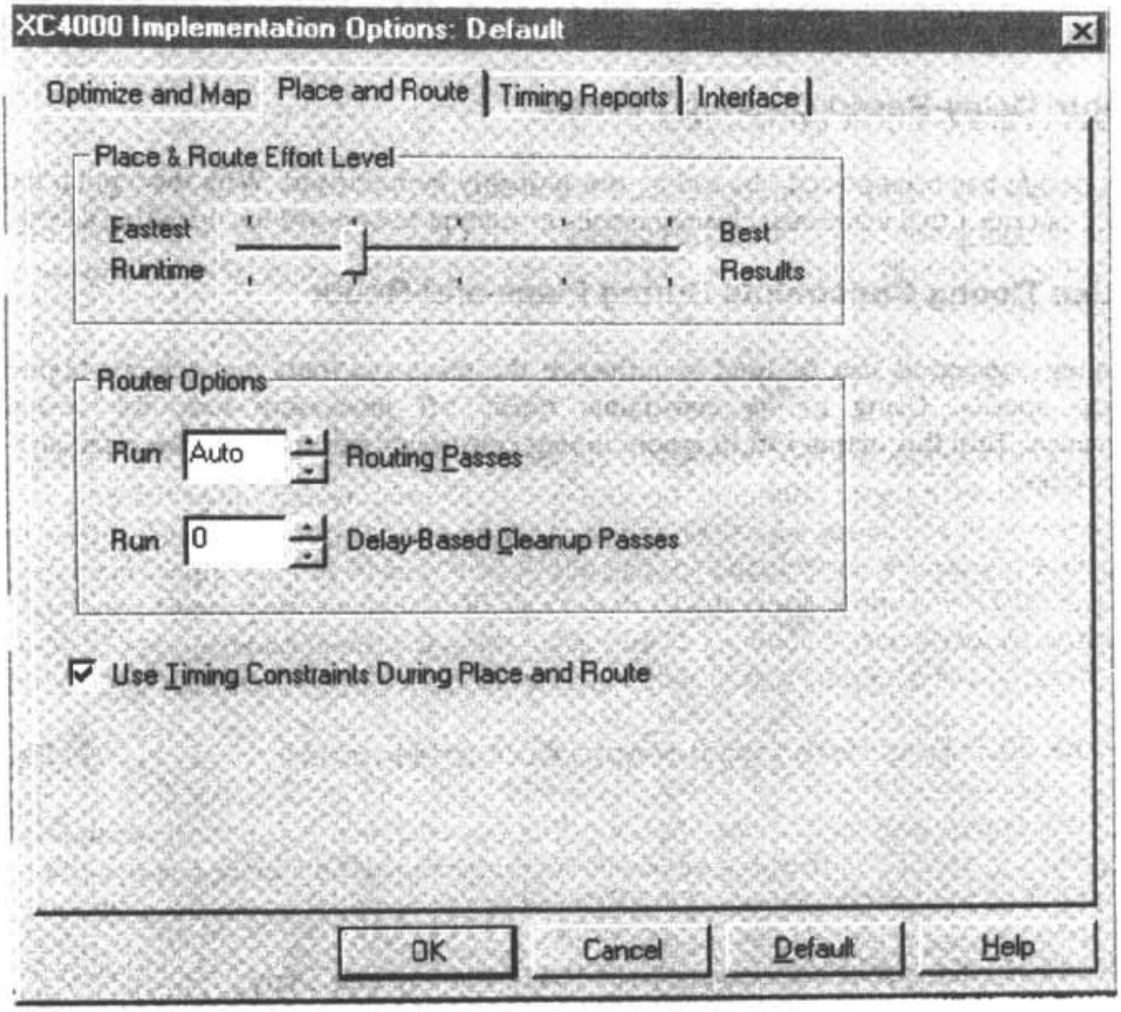

图 6-10　Xilinx 设计管理器的布局/布线选项

间和优化后的效果两者之间也存在一个折中的问题，在图 6-10 所示的布局/布线菜单中可以对此进行调节。若布局/布线工具运行时间越长，它可使用的优化选项就越多，则其面积/速度的优化效果也就可能越好。但更好的效果将花费更多的运行时间。

Router Option，Run Routing Passes（布线程序选项，执行布线路径尝试）设计者能够选择布线路径的数目。在布局过程中，对每一条布线路径都要进行完整的尝试。一旦布线程序找到了满足设计要求的布线路径（此时设计和器件相匹配，满足所有时序约束条件），则该布线程序终止运行。

Run Delay-Based Cleanup Passes（执行基于时延的清除路径的尝试）　一旦完成了设计布局，则设计的时序将有望得到改善。使用这个选项，设计者能够通过

设置 1～5 个附加的清除路径，以提高运行的速度。

Use Timing Constraints During Place-and-Route（在布局/布线过程中使用时序约束条件） 时序约束条件被用来对布局/布线施加影响，并实现更高的运行速度。使用时序约束条件来权衡设计执行的处理时间。此项设置为 off，将会忽略时序约束条件，以此来提高布局/布线的处理速度。

6.7 逻辑级时序分析报表/版图设计后的时序分析报告

Xilinx 设计管理器有关设计实现的选项界面如图 6-11 所示。

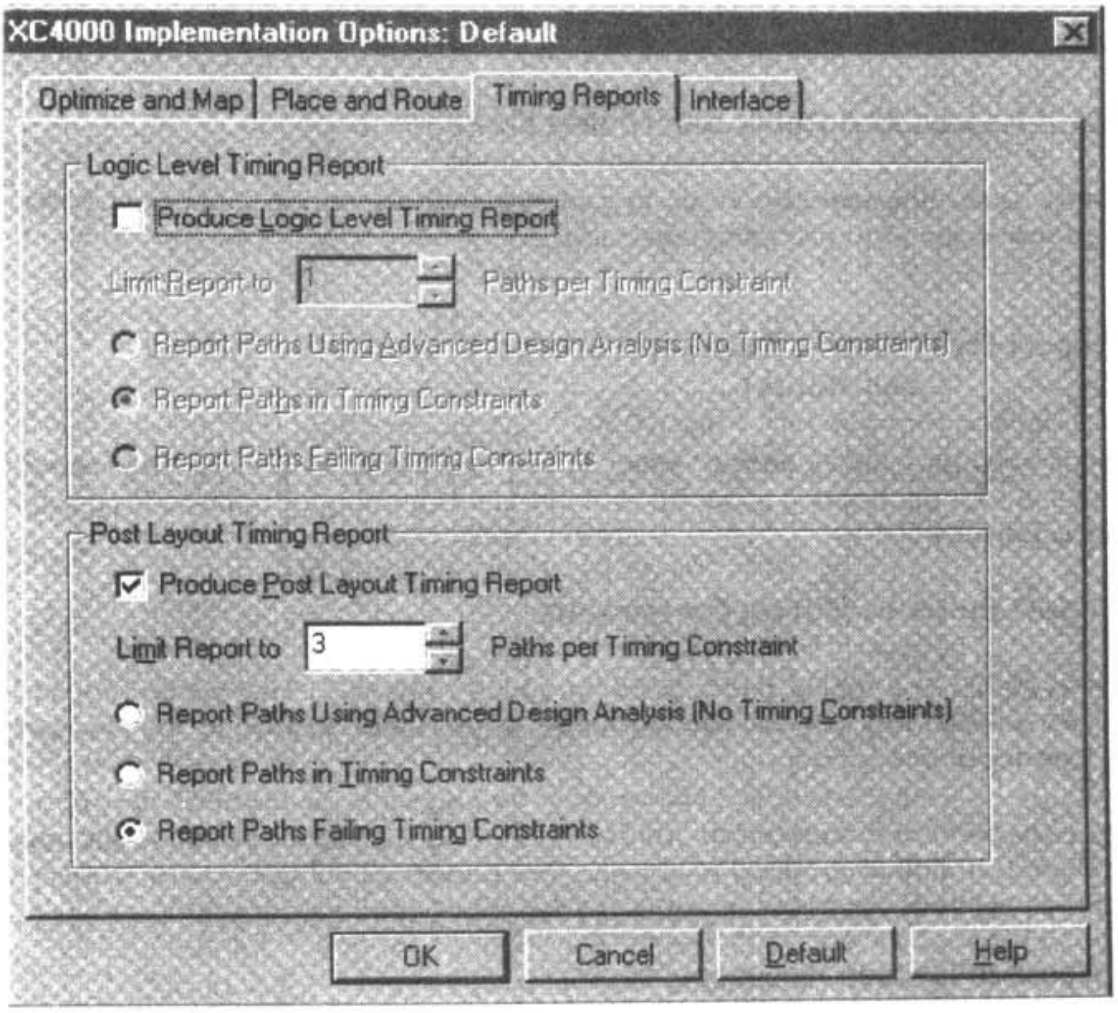

图 6-11 Xilinx 设计管理器的设计实现选项

Produce Logic Level Timing Report（生成逻辑层时序报告） 为了获取设计的有关时序执行方面的信息，可以通过选择图中的复选框来生成一个逻辑层的时

序分析。此分析可以在无需进行完整的(经常是非常耗时的)布局/布线过程的情况下得出结果。

Produce Post Layout Timing Report(生成版图设计后期的时序报告)　器件的顶层时序报告可以由该选项生成的简单时序报告得到。最大的时钟值被给出。对于错误和路径报告,在该选项下面给出了有关约束条件和时延值方面的三个子选项。如果迟延时间为负则表明有一个约束条件没有被满足。

Limit Report to n Paths per Timing Constraint(对每个时序约束条件,限制被报告的路径数目)　对于每个时序约束条件,用此选项(可以选取 Summary, No Limit,或者是 1 到 10 之间的一个数字)来设置给出的最恶劣状况路径的数目。

Report Paths Using Advanced Design Analysis (No Timing Constraints)(在无时序约束条件情况下,使用高级设计分析方法给出路径报告)　此选项给出了在无用户约束条件情况下的时序分析结果。这个分析中包含所有的时钟、每个时钟所允许的偏移量及一个按时延值分类的组合路径列表。

Report paths in Timing Constraints(生成时序约束条件下的路径报告)　此选项生成一个基于时序约束条件的时序分析报表。对于每个限制条件,被给出的路径数和 **Limit Report to n Paths per Timing Constraint** 对话框中所设置的值相同。

程序列表 6-15 给出了 hier688.v 设计中一个信号的时序报告。在 rwn 和outpu _ enable 之间的所有产生时延的路径都被列了出来,所有弛豫时间均为正值。注意:时延的 80%是由逻辑本身引起的。当设计变得更紧凑,并且逻辑对布线资源的使用进行优化后,这个比例将减小(可能减小很多)。

程序列表 6-15　Xilinx 时序报告实例

```
================================================================================
Timing constraint: TS01 = MAXDELAY FROM TIMEGRP "PADS" TO TIMEGRP
"FFS" 50nS;
 30 items analyzed, 0 timing errors detected.
 Maximum delay is  13.354ns.
--------------------------------------------------------------------------------
Slack:    36.646ns path rwn to output_enable relative to
          50.000ns delay constraint

Path rwn to output_enable contains 3 levels of logic:
Path starting from Comp: P102.PAD
To                    Delay type         Delay(ns)   Physical
Resource
```

```
                                                          Logical
Resource(s)
-------------------------------------------------------   --------
P102.I1                 Tpid                    3.000R    rwn
                                                          IPAD_rwn
                                                          ix46
CLB_R7C14.F2            net (fanout=1)          1.215R    rwn_int
CLB_R7C14.X             Tilo                    2.700R    D
                                                          ix79
P99.O                   net (fanout=1)          1.439R    D
P99.OK                  Took                    5.000R    output_enable

reg_output_enable
-------------------------------------------------------
Total (10.700ns logic, 2.654ns route)      13.354ns (to clock_int)
      (80.1% logic, 19.9% route)
```

Report Paths Failing Timing Constraints(生成不满足时序约束条件的路径报告)　此选项将得出不满足时序约束条件的信号和路径的报告(按从差到好的顺序)。对逻辑时延和布线时延进行检查,找出那些不合理的时延路径。全面的时延分析将为设计者提供一些有用的线索,这些线索有助于发现那些可以通过流水线处理和简化处理而提升整个设计的执行速度的区域,以及约束条件设置得过于严格的区域。

对于每个限制条件,被给出的路径数和 **Limit Report to n Paths per Timing Constraint** 对话框中所设置的值相同。

6.8 接口选项[①]

Macro Search Path(宏搜索路径)　当网表文件被合并,.ngo 文件被插入时,编译器为要插入的文件搜寻正确的路径。用户可以添加其他的搜索路径。可以输入多个搜索路径,之间以分号作为路径分隔符。

Rules File(转换文件格式)　合并入 ncf 格式的网表文件中的文件类型必须是.ngo 格式。**rules file** 路径指向一个可以将其他格式的网表文件转换为.ngo 格式文件的应用程序的位置。

Create I/O Pads from Ports(从端口生成 I/O 引脚)　某些设计工具可以将 PAD(器件引脚)符号转变为模块端口符号。这个复选框选项用来将顶层模块端口转化为 PAD。

① 此选项见图 6-9 ~ 图 6-11 中的 interface 标签。——译者注

6.9　VHDL/VERILOG 仿真选项

6.9.1　仿真选项

Simulation Data Option(仿真数据选项)　Xilinx 能够以三种格式，即 EDIF，VHDL 和 Verilog 来创建具有时序标注功能的网表文件。这里我们采用 Verilog 选项来支持 Verilog 仿真。供应商支持的 Xilinx 布局/布线工具包括常见的 EDIF，Verilog，VHDL，还有 Active VHDL，Concept NC-Verilog，Concept Verilog-XL，Foundation EDIF，ModelSim Verilog，ModelSim VHDL，NC-Verilog，Quicksim，Verilog-XL，Viewsim-XL，Viewsim-EDIF，VSS 和缺省格式。本书将采用 ModelSim Verilog。

Correlate Simulation Data to Input Design(使仿真数据和输入设计相关联)　为了在被优化的网表文件中使用自己的逻辑门及信号名称，而不是使用由布局/布线工具分配的名称，请选取这个复选框选项。

Simulation Netlist Name(仿真网表的名称)　为仿真输出文件定义文件名。如果你想保留这个仿真文件的多个版本，则应在此处输入多个文件名，否则，新文件将覆盖旧文件。

Bring Out Global Set/Reset Net as a Port(将全局置位/复位网络作为一个端口)　为了仿真的需要，可以手工将内部可用的置位/复位节点用作设计顶层的一个端口。驱动全局置位/复位(GSR)资源的信号名称能够被填入对话框中以便和 HDL 设计相匹配。

Bring Out Global Tristate Net as a Port(将全局三态结点作为一个端口)　为了仿真的需要，可以手工将内部可用的三态控制网线用作设计顶层的一个端口。驱动全局三态(GTS)的信号名称能够被写入对话框中以便和 HDL 设计相匹配。这种三态状态控制着所有器件的输出，当进行接受外部设备测试时，用于将器件与电路板进行隔离。

Generate Test Fixture/Testbench File(生成测试程序/测试平台文件)　选择这个复选框将生成一个 Verilog 测试程序(.tv)临时文件。

**Include `uselib Directive in Verilog File**(在 Verilog 文件中包含`uselib 指令)　Xilinx 提供一组具有时序注解功能的 SIMPRIM(SIMulation PRIMitive 仿真原语)文

件。通过选择这个复选框,这些文件的路径能够被自动的插入到 Verilog 测试程序中去。

Generate Pin File(生成引脚文件) 选择这个复选框将生成信号到引脚(.pin)的映射文件。

Retain Hierarchy in Netlist(保留网表文件中的层次结构) Verilog 测试文件可以保留输入设计的层次结构,也可以把原网表文件转换成单层的较大文件。选择这个复选框将保留输入设计的层次结构。

6.9.2 配置选项

Xilinx 器件是基于 SRAM 原理的,必须在每次上电后进行配置加载。有多种配置模式,包括:串行 PROM、主动并行模式、从动并行模式、电缆下载等。

Configuration Rate(配置速率) **Slow**(1MHz)或 **Fast**(8MHz)内部配置时钟(主动模式)。这些速度均为近似值。

Threshold Levels(阈值等级,仅适用于 **XC4000E** 和 **XC4000EX**) 在 TTL 兼容电平的输入阈值(标称值为电源供电值的 30%)或 CMOS 阈值(标称值为电源供电值的 50%)和输出驱动间进行选择。若选择 **Read from Design**,则将使用在物理性限制(PCF)文件中已经定义了的 TTL/CMOS 输入电平。

Configuration Pins(配置引脚) 对于 TDO, Mode, Done 配置引脚,提供了不同的选项(包括一个三态模式)。

Perform CRC During Configuration(在配置过程中执行 CRC 校验) Xilinx 的内部配置逻辑可以对配置数据帧执行一个长度为 4 位的局部 CRC 校验,或者仅仅在每一帧的末端进行 0110 模式的简单校验。

Produce ASCⅡ Configuration File(生成 **ASC**Ⅱ码配置文件) 标准的配置文件是一个二进制的.bit 格式的文件。但也可生成对应此比特流配置文件的 ASCⅡ码格式的文件。

5V Tolerant I/Os(输入/输出容许电压为 **5V**,仅适用于 **XC4000XLA** 和 **XC4000XV**) 在混合电源供电环境下,工作于低电压的器件的 I/O 引脚可以通

过配置以承受高一些的驱动电压。

6.9.3　启动选项

Start-up Clock(启动时钟)　配置任务可以使用内部时钟源(CCLK)或外部时钟(用户时钟)源来驱动。

Synchronize Start-up to DONE Input Pin(使启动和 DONE 输入引脚同步)　当多个器件以菊花链模式配置时，一次只能对一个器件进行配置。当前一个器件配置结束后，下一个器件的配置工作才能开始。

Output Events(输出项)　控制信号能够在不同的时间段被确定或释放。这些状态信号包括：Done，Enable Output，和 Release Set/Reset。

Readback(回读)　当回读功能开启时，器件的配置信息可以被读出(考虑到设计安全的问题，回读功能也可以被禁止)。此标签包括与回读时钟源(内部或外部)和回读过程终止有关的选项。

Tie Unused Interconnect(未使用的内部连线)　空闲的引脚能够被设置为高电平或低电平，用以减少噪声和能量损耗。

Advanced Options(高级选项)　在主动并行配置模式下，FPGA 通过生成地址线去控制一个并行存储器件，这些配置地址线可以用来对 18 或 22 条线进行配置。

6.10　其他的设计管理器工具

设计管理器工具包括：Flow Engine(流程引擎，用其执行布局/布线过程)，Timing Analyzer(时序分析器)，Floor Planner(资源分配器)，PROM 文件格式化工具，Hardware Debugger(硬件调试器，包括 FPGA 下载程序)，和 EPIC Design Editor(EPIC 设计编辑器)。

6.10.1　时序分析器

时序分析器将提供对设计中选定路径的报告。例如，可以对设计中所有的时钟进行检查。具体的路径可以不予考虑。

程序列表 6-16 Xilinx 时序报告实例

```
==========================================================================
Timing constraint: Default period analysis
 12 items analyzed, 0 timing errors detected.
 Maximum delay is  11.647ns.
--------------------------------------------------------------------------
Delay:    11.647ns device_bus2(0) to device_bus1(2)

Path device_bus2(0) to device_bus1(2) contains 3 levels of logic:
Path starting from Comp: P46.PAD
To                    Delay type         Delay(ns)  Physical
Resource
                                                    Logical
Resource(s)
-------------------------------------------------  --------
P46.I2                Tpid                 1.560R  device_bus2(0)

IPAD_device_bus2(0)
                                                    ix57
CLB_R24C1.G2          net (fanout=3)       2.016R
device_bus2(0)_int
CLB_R24C1.Y           Tilo                 1.590R
device_bus1_dup0(3)
                                                    ix66
P44.O                 net (fanout=1)       2.441R
device_bus1_dup0(2)
P44.PAD               Topf                 4.040R  device_bus1(2)
                                                    ix50

OPAD_device_bus1(2)
-------------------------------------------------
Total (7.190ns logic, 4.457ns route)       11.647ns
      (61.7% logic, 38.3% route)
```

程序列表 6-16 给出了有关 hier688 设计最坏的路径(关键路径)的一般时序报告。此路径的最大延迟时间是 11.647ns。注意列表底部分别列出的逻辑及布线之间的时间分配比例。设计密度越高,布线所造成的时延在总时延中所占的比例就会越大。

6.10.2 资源分配器

资源分配器是一个用来对 FPGA 内部逻辑的排列及定位进行处理和优化的工具。图 6-12 给出了一个用此工具生成的典型的器件平面布置图。这个设计的某些方面对于设计者来说是显而易见的,同时对自动布局/布线工具来说,可以识别,但也可能无法对其进行识别。哪些部分是设计的关键并应当和其他的逻辑单元相邻?能够通过采取一定的措施来找出一个执行速度更快,效率更高的设计方案吗?和计算机相比,人更能胜任此类工作。

图 6-13 给出了 hier688 逻辑、引脚分配、可配置逻辑块(CLB)及信号布线放大

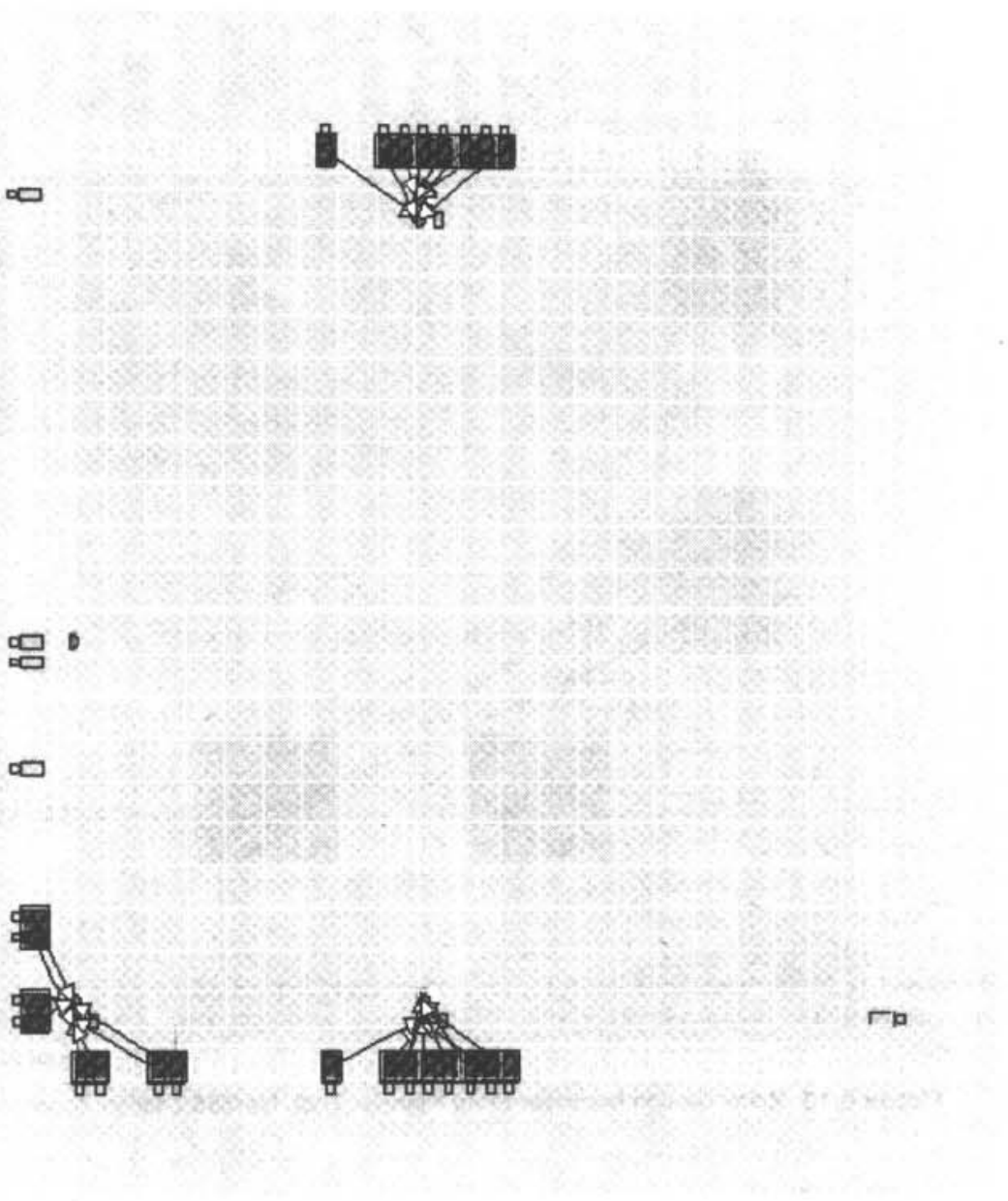

图 6-12　Xilinx 设计管理器资源分配器工具

视图。

6.10.3　PROM 文件格式化工具

Xilinx 支持串行和并行配置 PROM 的模式。一个文件可以被生成并存放于一个微控制器的 PROM 中。大型的器件可能需要多个 PROM。此 PROM 文件格式化工具允许将一个设计配置文件分割成多个部分，用来分别配置不同的器件，如图 6-14所示。

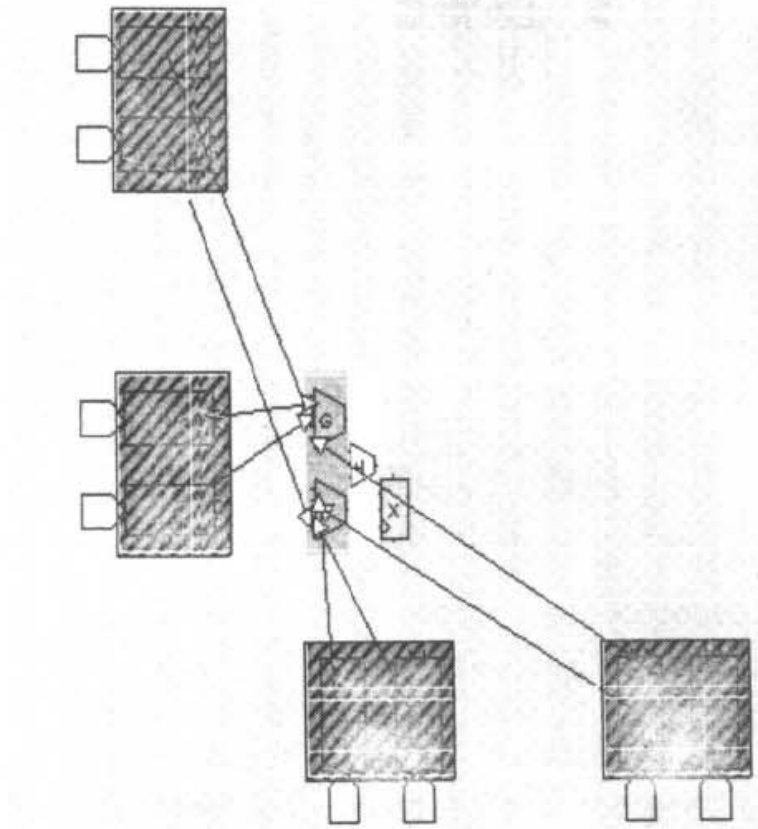

图 6-13 Xilinx 设计管理器资源分配器工具,hier688 设计的放大视图

6.10.4 硬件调试器

硬件调试器提供通讯设置选项,如图 6-15 所示,一个器件可以通过 PC 机的串行端口、并行端口,或 Xilinx 的 Xchecker 电缆(和 PC 机的并行端口相连)进行数据的下载。图 6-16 给出了电缆的连接方式。Xilinx 也支持四线(TDI,TMS,TCK,TDO)的 JTAG 串行端口编程方式。

6.10.5 EPIC 设计编辑器

这个工具提供了此设计对应的实际物理器件本身的俯视图(图 6-17)。图中可以看到引脚、引脚缓冲器、寄存器及全局信号、信号布线和 CLB。某些布线操作可

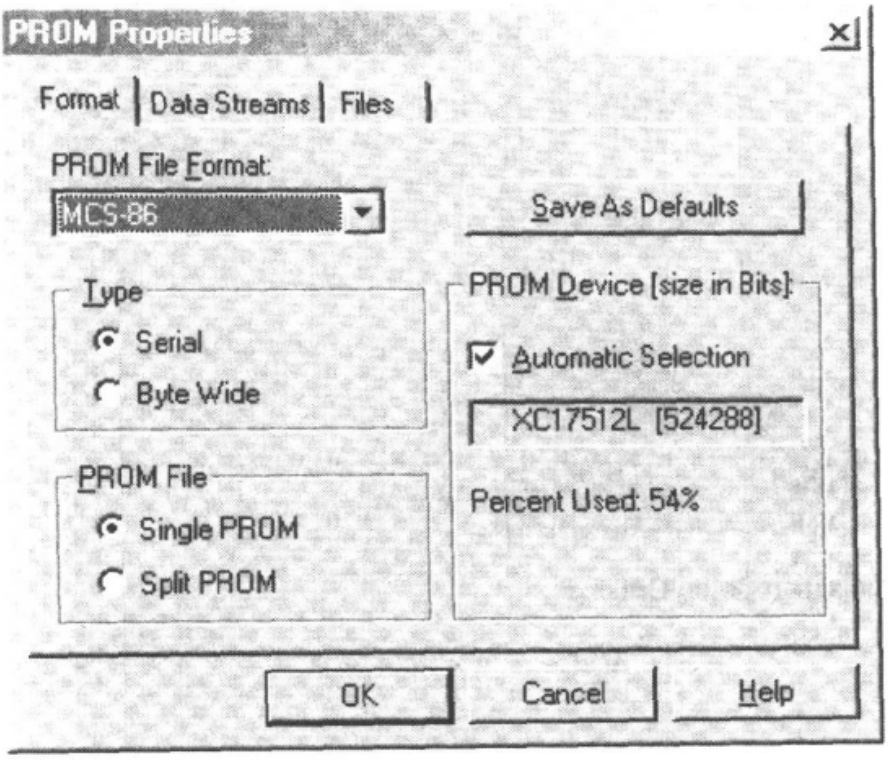

图 6-14　Xilinx 设计管理器 PROM 文件格式化选项

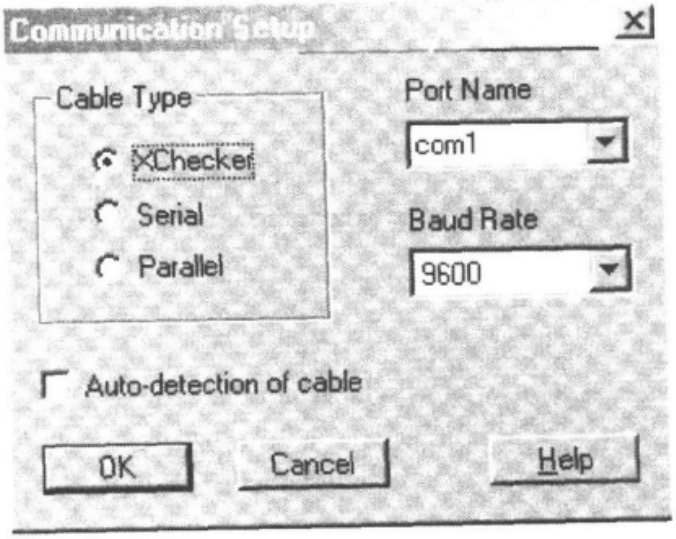

图 6-15　通讯设置选项

以在此环境下完成。例如，在不必进行重新综合和编译的情况下，连接测试点是完全可能的。

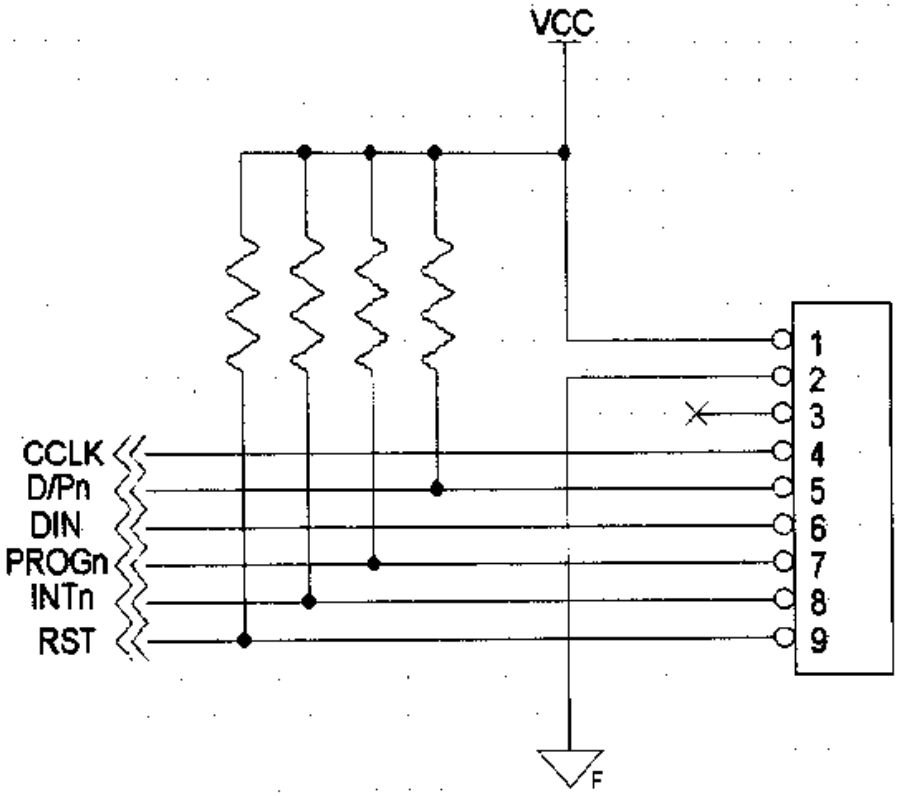

图 6-16　Xilinx 的 Xchecker 电缆接线

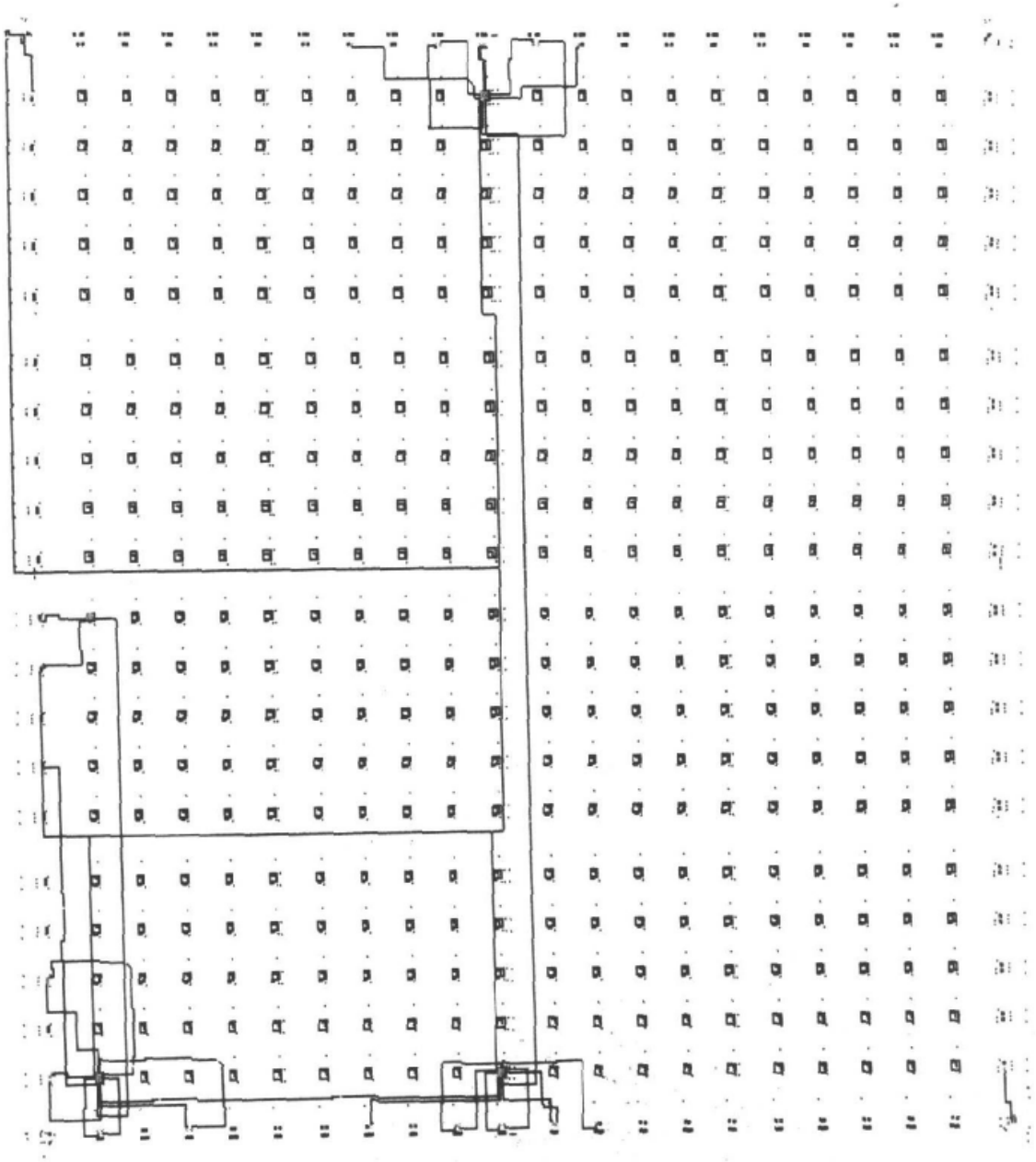

图 6-17　hier688 设计的 EPIC 图示

第七章　几种架构的比较

FPGA 器件厂商曾经被认为神通广大，他们生产的产品往往可以获得令人惊讶的销售量，但实际上他们都无一例外进行着商业运作。FPGA 器件生产商，包括 Xilinx，Altera，Lucent，Actel，Lattice/Vantis，Quicklogic，Atmel，以及其他的厂商，都从同样的渠道以几乎相同的价格购进他们需要的组件。大多数（Lucent 和 Atmel 除外）公司都在压缩其具有相同经营模式的生产规模。

7.1　决定集成电路价格的因素

封装　集成电路的成本中，用于封装部分的花费占据很大的比例（就像生产土豆片的成本花费一样）。

硅片　这里指的是“晶片尺寸”和布线层数和光刻版精度。这和一个仓库的情况相似，仓库的价值与它的库容大小有关。至于其他因素，如 IC 加工过程是否属于“主流”，也能够对价格产生影响。一些可编程的器件使用 EEPROM 技术或特殊的材料，如 Actel 公司就在电路层间的互连上使用钨质接插件。

产量或规模经济　要想得到一个价格便宜的 IC，要么进行大批量选购业界业已大量使用到的产品（换句话说，若你想省钱，那就买大多数其他公司正在购买使用的产品）。要么它必须进行大批量的生产。FPGA 产品使用灵活，已经被许多设计者所采用。ASIC 则具有特殊性，针对的是有特殊用途的用户（虽然这些特殊的厂商用户可能会把其购得的产品重新卖给更广大的市场。例如，从 Intel 公司的观点来看，微处理器就是一种专用集成电路）。

这些因素只有在竞争的局面形成后才会发挥作用。在竞争的形式下，资源受到特殊的工艺过程、技术所有权及小规模市场的限制，经济形势是不同的，价格是在增长的。

FPGA 市场是动态的和激动人心的，但因其标准和产品种类多也往往令人不知如何去选择而感到无所适从。每个 FPGA 供应商为了帮助购买其产品的用户解决所遇到的问题，在产品设计上都有自己的一套设计策略。他们都具有自己的产品定位、策略及相关技术。但要记住的是，对于每一个设计目标，都不存在终极的、理想的解决方案，每一个 FPGA 设计都有其强项及不足之处。一个 Xilinx 的 FPGA 可能是针对一个问题的最好的解决方案（如混合函数的随机阵列）；Altera 的 CPLD 可能是解决另一个问题的最好方法（如包括数字滤波器的数据通路函数）；而一个

Actel 器件对于一个要求其具有 ASIC 性能的设计来说可能是最好的选择。

7.2　FPGA 器件设计

有一些事情，包括专利权问题，会影响 FPGA 器件的设计工作。使用一个已经获得专利的系统是件令人头痛的问题，因为这会使你的竞争对手赢利，而它仅仅是因为具有该专利的特许权。你能够依据一个专利开展设计吗？你能够想出解决困难的其他方法吗？仅仅发放专利的使用权，状况就更好吗？另外，你所关注的目标市场具有什么特征？竞争对手提供的系列产品中是否存在某些缺陷而使我们有可乘之机？

一个使用 Verilog 进行 FPGA 设计的设计者应当对将要使用到的 FPGA 器件的架构有一个透彻的了解。需要记住的是，你的设计是要通过查表来实现的，而这会带来不可忽视的布线传输时延。Verilog 是一种具有可移植性的语言，但即使最高性能的设计最终也要努力去适应目标 FPGA 器件的特性。FPGA 设计单元到底是怎样的？Xilinx 公司的 4000XL 型号 FPGA 器件可以被看做是一个由多个四输入查找表组成的阵列，每个查找表为 16 × 1 的 RAM。Altera 公司的 Flex 8K 器件是一个由逻辑阵列块(LAB)构成的阵列，每个逻辑阵列块由 8 个专用 RAM 块(逻辑单元)构成，每个逻辑单元又各含有一个四输入的查找表。

在 FPGA 中的一个芯片上，Altera 的器件可以做到 20 000 门，而其尺寸却仅约相当于具有相同门数的 Xilinx 器件的 2/3。这是否意味着 Altera 器件在芯片集成度上比 Xilinx 更高？还是意味着 Xilinx 公司相信具有更多的布线资源对于充分地支持 CLB 的使用是必要的？是否支持内部的三态信号？有多少全局低时偏时钟网络是可用的？哪一种架构更好呢？使用能够提供更多的门和更多的布线资源的芯片就一定更好吗？对于以上这些问题并不存在一个普遍的和一般性的答案。

7.3　在选择 FPGA 器件时需要考虑的问题

FPGA 器件可以满足任何实际工作的需要，只要选用的器件具有足够的引脚和门电路。通常，一个器件往往在没有任何具体理由的情况下就被选定了，而选择时考虑的仅仅是诸如：公司有什么备件，哪一种器件的使用经验值得设计者保存等。我们不打算谈论可用门的数目问题；每个供应商在计算可用门数目时采用的方法都不一样。以下列出的问题比对器件的逐一介绍要重要得多，因为技术总是在不断进步的。

- 器件中包含多少逻辑块和触发器？
- 器件能提供多大容量的 RAM(如果有的话)？是双端还是单端？分布式的

还是集中在块中的？逻辑块有多大？它们能够以何种组合方式被使用（×1，×2，×4，×8，×16，×32 等）？

• 器件具有多少可用的 I/O 引脚？

要记住的是，每一种器件都有电源/接地及其他专用的引脚，它们是不能被 Verilog 的设计者分配给其他信号使用的。

• 可以提供多少可用的低偏全局时钟/置位/预置网线？

将时钟及其他全局信号沿着信号通道进行布线是可能的，但这样一来，由于时钟波形失真的影响，将给原有设计带来不少的困难。

• 支持内部三态总线吗？

和多路选择器（MUX）相比，三态总线（通路）能够大幅度地提高译码器的运行速度。

• 在同一套组件中，是否提供引脚输出兼容且密集度更高的器件？

如果设计规模增大，电路板是否有必要进行重新设计以容纳更高密集度的器件？

• 在 FPGA 设计完成之前，器件引脚输出不得不被锁定吗？

CPLD 器件和 FPGA 器件相比，完成一个原先做印刷线路板设计（PWB）时所需要的条件对 FPGA 器件（器件引脚到随机逻辑单元间的布线能力更强）来说更加适用。

• FPGA 在上电后能够被重新加载或上电后即可投入使用吗？

想像一下为微处理器进行存储器译码的 FPGA 设计。如果 FPGA 被微处理器初始化，并且 FPGA 不对存储器进行译码，以致微处理器可以正确访问存储器的话，这可能是一个加电的问题。一个用于存储器解码和时钟产生的快速 PLD 设计会经常伴随着 FPGA 的使用。

• 对于现场升级或电路板的专业化定制要求，FPGA 是否支持相应的内电路配置功能？

• FPGA 是否具有可用于输入引脚的触发器？

在器件的外围，靠近输入引脚的位置配置可用的触发器，可以做到对输入信号进行快速的和可预测的锁存。

• FPGA 是否具有可用于输出引脚的触发器？

在器件的外围，靠近输出引脚的位置配置可用的触发器，则可以产生快速和预期的时钟-输出次数。

• 有向 ASIC 方向进行转化的必要吗？

如果有这种可能性的话，则应考虑选择和 ASIC 架构及速度类似的反熔丝器件。

• 需要配置接插件吗?

对于那些一次性编程器件,在设计被固化之前,为了适应器件的变化,设置一个接插件是十分必要的。这绝不是一件微不足道的事,当然,密集的引脚间距给接插件的安排带来了困难。

Xilinx 和 Altera 两大公司的产品大约占有 FPGA 市场 80% 的份额。哪一家的市场更大? 这个问题一直在争论着,要回答这个问题还必须要看这个数字是如何被计算出来的。如果不了解这两家公司的产品,就不能算是一个真正的 FPGA 设计者(个人履历中若注明对这两家公司的产品均具有开发经验时,会使你在求职时更具竞争力)。其他的公司也有其强大的产品和相应的软件! 但至今没有一家可以达到像 Xilinx 和 Altera 公司那样的市场份额。本书后面列出了器件制造商的网站地址,读者可以访问这些网站以了解最新的相关信息。

7.4　Xilinx 公司 FPGA 器件的架构

7.4.1　XC3000/XC3100 系列 FPGA 器件

这是一个较早的基于 SRAM 的体系结构。每个可编程逻辑块(CLB)有 5 个逻辑输入、2 个触发器、1 个公共时钟、1 个直接复位端及 1 个时钟使能。请注意时钟

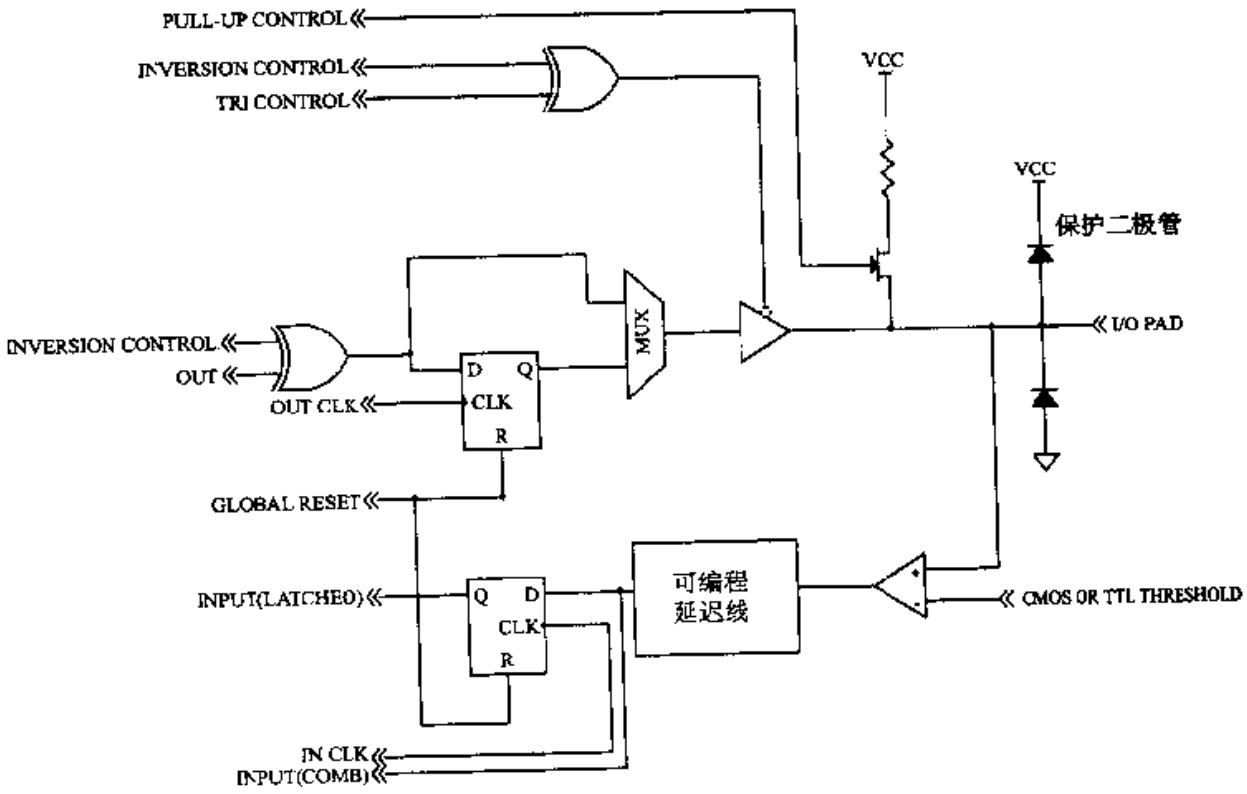

图 7-1　Xilinx 的 3K 系列 I/O 结构

使能的实现方法(图 7-2)。一个时钟使能多路转换器(MUX)用于在查找表的输出及锁存器之间进行选择。当处在时钟禁止模式下时,锁存器的输出将被反馈回输入端。

如图 7-1 所示,XC3000 的可编程输入/输出块(IOB)包括有输入和输出触发器、可编程的三态及上拉电阻输出控制。此种结构不具有如 SRAM、快速进位输入/输出等更新的技术特性。

FPGA 器件中信号的路由选择是通过称为可编程内部连线点(PIP)的旁路晶体管电路来实现的。旁路晶体管电路由基于 RAM 结构的配置位来控制。在 CLB 间的布线包括非直接的连接、在水平方向和垂直方向长线间的驱动器、在垂直方向和水平方向长线间的驱动器及其他特殊类型的驱动器。长线绕过 PIP,用于传送贯穿整个芯片,即长距离传输的信号。3K 系列器件包含一个可以用来支持振荡电路(使用一个晶体的 RC 网络,或谐振器去设置工作频率)的缓冲器。

Xilinx 的 3K 系列器件采用基于 SRAM 的架构。上电后,器件的配置文件被加

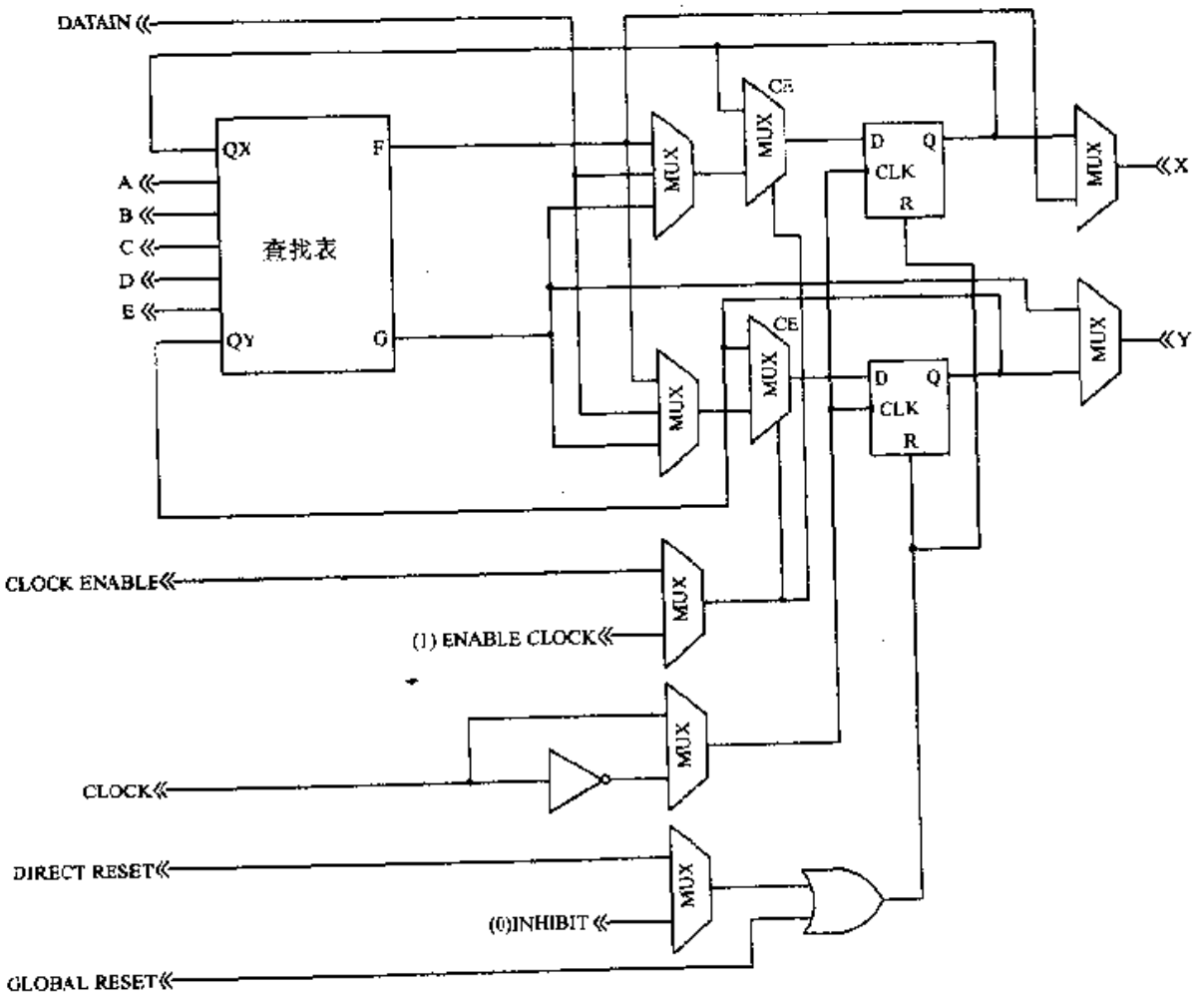

图 7-2 Xilinx 的 3K 系列 CLB 结构

载到器件中，加载是通过串行 EPROM 或串行下载电缆来实现，或是由微处理器(从模式)或字节宽度/字宽度的 EPROM 利用并行方式或其他方式来实现。基于 SRAM 的器件在断电后，其配置将丢失。在上电时，Xilinx 器件将按可编程模式引脚所定义的方式自动加载配置数据。

7.4.2　XC4000 系列 FPGA 器件

和 Xilinx 的 3K 系列相比，4K 系列器件具有更高的集成度、更高的速度、以及其他新增的特性。特别是在 4000E 和 4000X 系列器件中，可以将一个 CLB 定义配置为一个 16 × 1 大小的分布式 RAM 单元来使用，这个特性对于设计者来说是非常有用的(图 7.3)。

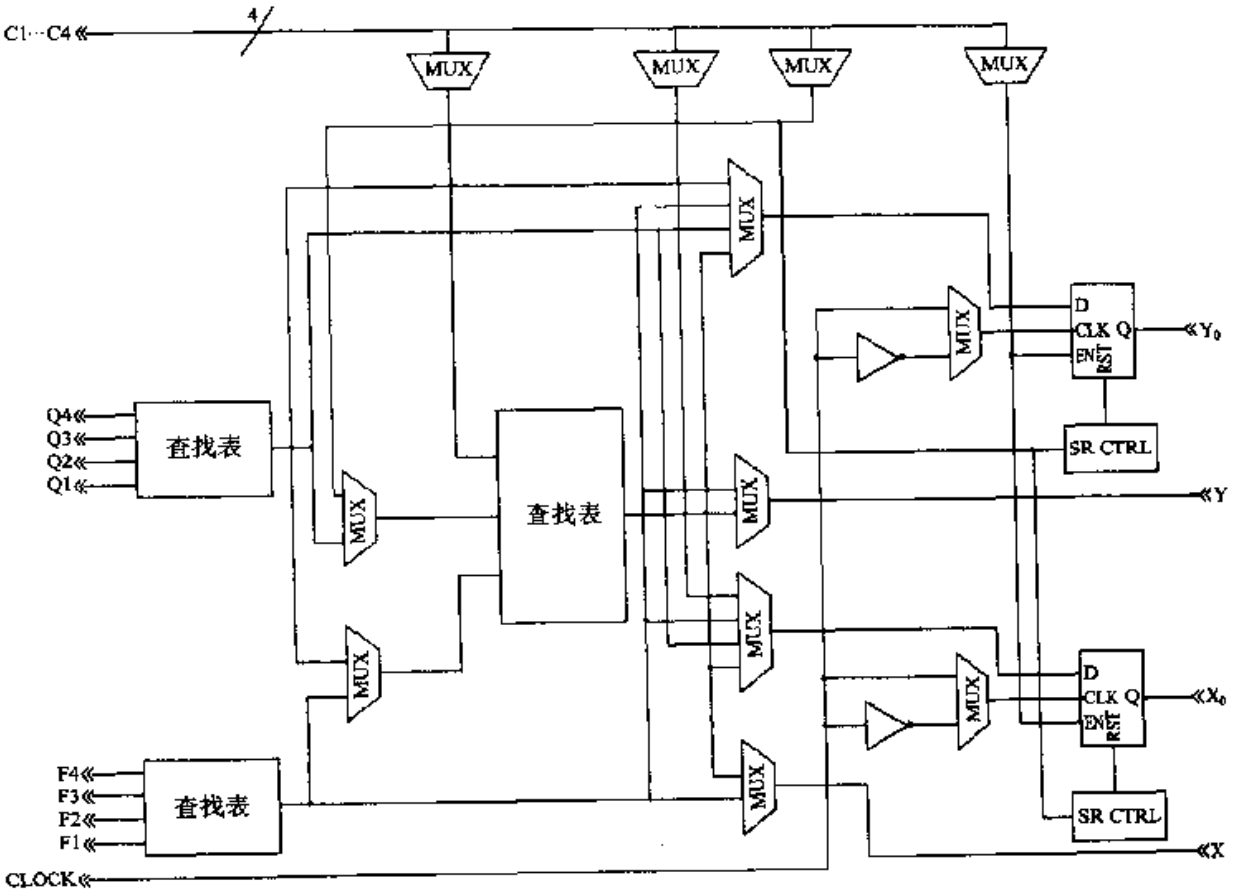

图 7-3　Xilinx 的 4K 系列器件 CLB 结构

Xilinx 的 CLB 结构中包括两个四输入查找表，两个具有专用时钟使能控制的 D 型触发器、置位或复位端、极性可配置的时钟，以及快速进位输入和进位输出信号通路。在 4000E/XL 器件中的每一个 CLB 都可被分别充作：两个 16 × 1 大小的单端口 RAM 单元，或一个 16 × 1 大小的双端口 RAM 单元，或一个单端口的 32 × 1 大小

的 RAM 单元来使用。双端口 RAM 配置是同步的,其他方式的配置可以是非同步的(电平触发)。

Xilinx 提供的其他模块还有:输入/输出块(IOB),它包括 I/O 寄存器和可配置的终端负载(上拉式或下拉式)及引脚缓冲器(快速的或慢速的);宽位解码器块,它对于输入端达到 9 个的快速解码器有辅助促进作用。4000 系列器件具有一个片上振荡器和专用的低偏移网络,可以提供给时钟信号和其他快速全局信号使用。4000 系列还支持内部的三态信号和总线。

7.4.3 硬布线(HardWire)器件

Xilinx 公司为其 FPGAs 系列提供了一套 HardWire 版本的产品,对于那些使用器件数量有限,转化为全定制 ASIC 并不划算的应用来说,采用此产品可以为用户节省一些成本。在此技术中,Xilinx 采用了相同的 CLB 结构,但以金属布线层代替原有的 SRAM 布线和开关转化阵列。和 FPGA 相比,这种改进将使用更少的硅材料(更小的芯片面积),同时获得同等的或更好的时序。转换为 HardWire 器件的一个好处是,FPGA 设计者对这种变换所承受的压力并不大;Xilinx 公司可以确保此定制器件的时序及功能与 FPGA 器件相匹配。向 HardWire 器件转化的过程是一个非向量测试过程,Xilinx 不断扩展自动化测试的覆盖范围,并且可以保证器件在用户的实际应用中正常工作。这就使得设计者可以摆脱需要另外进行同步设计及做彻底的性能测试之苦。例如,所有的异步逻辑都需要对竞争条件进行检查,因为使用 HardWire 器件的设计在多数情况下可能比使用 FPGA 器件的设计的运行速度更快。

7.4.4 Virtex 系列 FPGA 器件

Xilinx 公司最新的器件采用了 0.22μm 光刻技术(线间距可达到 0.18μm)和五层金属层布线工艺。百万门级的器件内部集成了 7500 万个晶体管。它有一些值得注意的新特性(图 7-4 和图 7-5),其中包括混合电压 I/O(包括低电压差分输入以支持类似 GTL 的总线)、专用的 4096 比特位双端口 SRAM 块、分布式 RAM 单元、使用多 DLL(延迟锁相环)以提供时延可控的时钟网络及基于向量特性的布线方式(允许在 CLB 之间进行灵活的上/下/左/右四方向的布线操作)。此类器件工作电压为 2.5V,不过其 I/O 端口可承受更高的接口电压的冲激。

Xilinx 公司在广告中宣称其百万门的器件内部含有 27 648 个逻辑单元,131 072个块 RAM 比特位,以及 660 个用户 I/O 引脚。而 8031 微控制器核心仅仅包含不足 600 个可用门(相当于不到 100 个 CLB),256 位 RAM 及 32 个用户 I/O 引脚。这真是一个惊人的消息。

Virtex 的 I/O 块(图 7-6)包括可编程的上拉和下拉电阻、一个弱保持电路(当驱

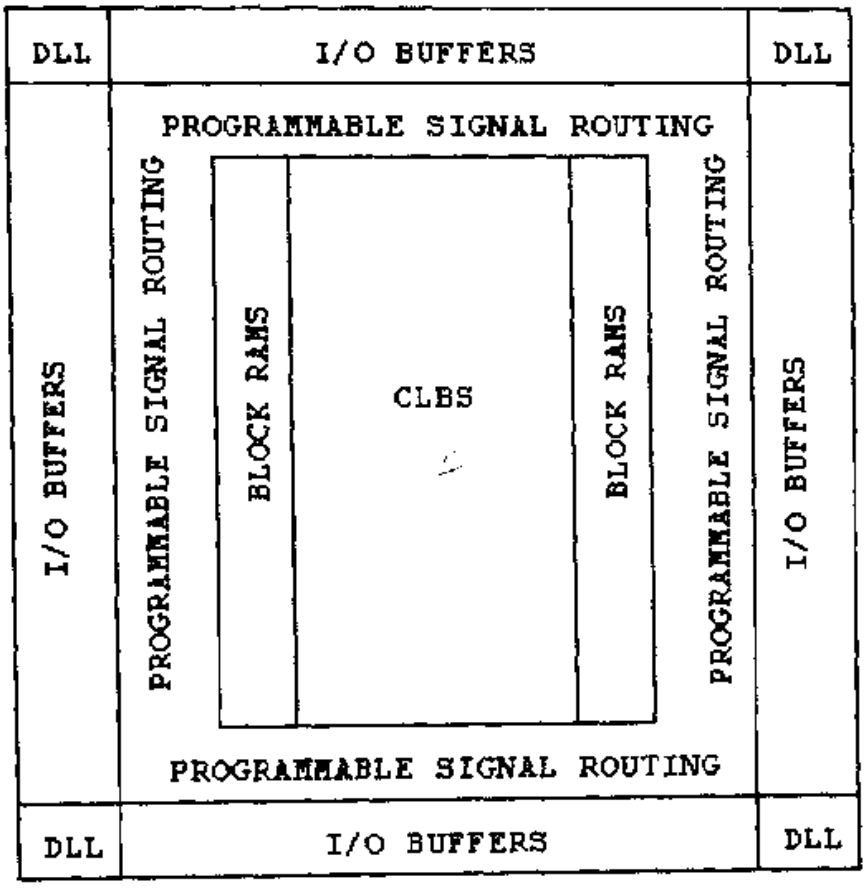

图 7-4　Xilinx 的 Virtex 系列器件体系结构

动信号消失时，可用来保持原有信号值不变）、三态控制、I/O 锁存器，以及时延可编程的输入端（可用来依据时钟边沿转换输入信号）。

7.4.5　配置器件

在开发研制过程中，使用下载电缆连接 PC 机是进行 FPGA 配置的最简便方法。一旦设计结束，一个串行配置器件（串行 PROM）就可派上用场。一次性编程（OTP）器件（来自于 Altera，Xilinx，Lucent 公司）和可重复编程的器件（来自于 Atmel 公司）都可以使用。为了节省串行器件的成本，利用微处理器进行并行下载也是可行的。这就要求处理器必须在 FPGA 被配置前进行初始化并开始运行。另外，Xilinx公司的器件还可以通过 JTAG 串行端口进行编程。

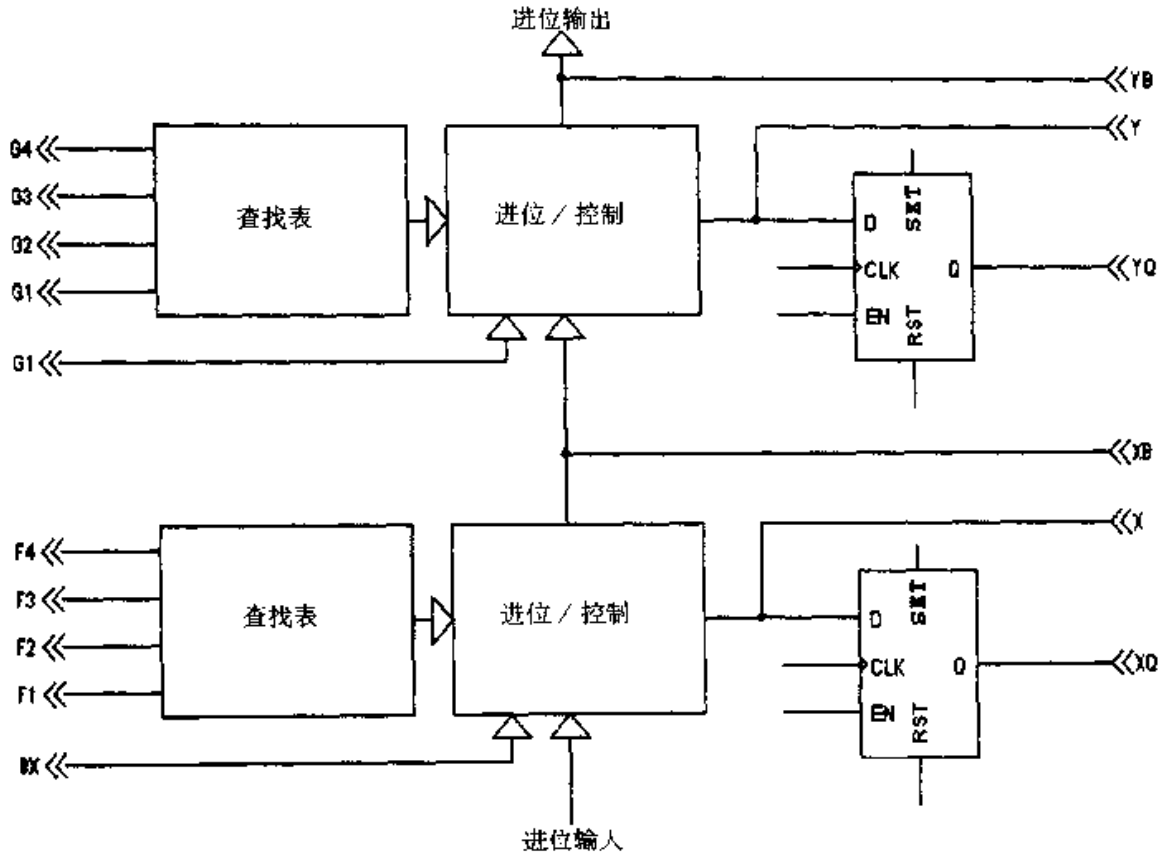

图 7-5 Xilinx 的 Virtex 系列器件 CLB 结构(给出的是每个 CLB 单元中两片中任一个的结构)

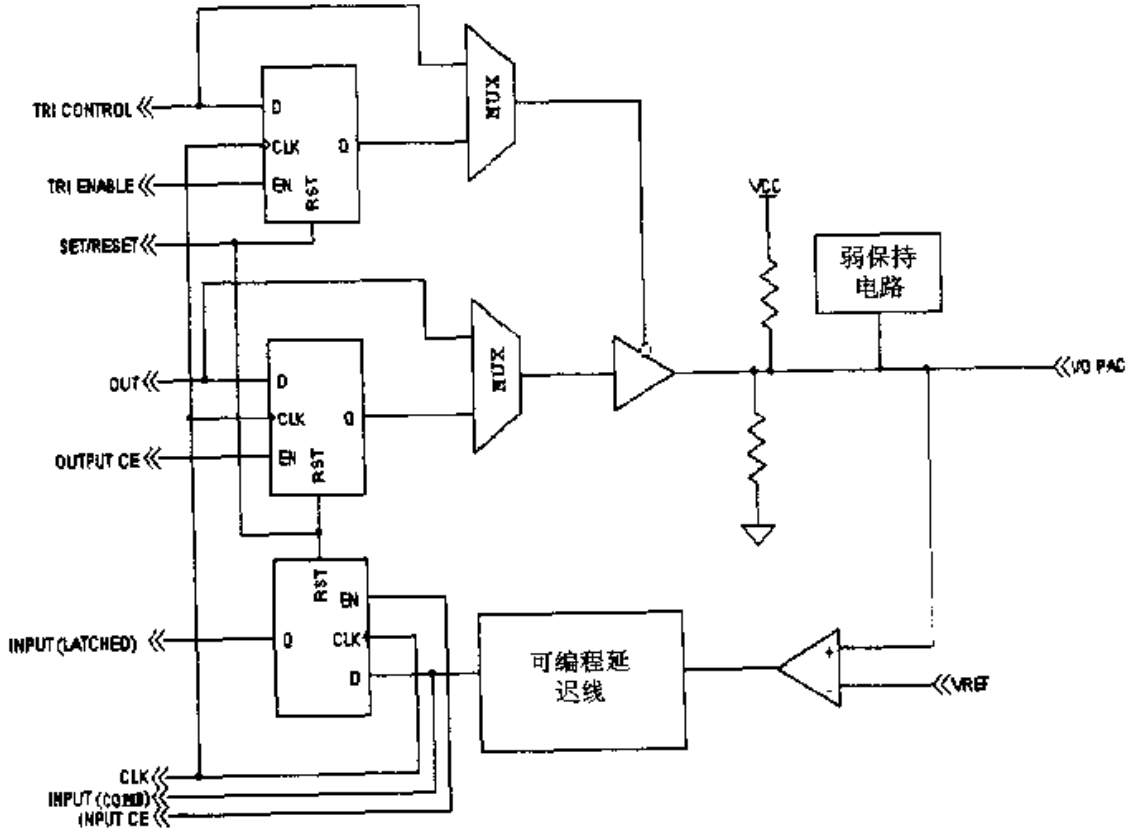

图 7-6 Xilinx 的 Virtex 系列器件输入/输出块

7.5　Altera 公司 CPLD 器件架构

Altera 公司使用“复杂可编程逻辑器件”(CPLD)这个名称来描述其设计的实现方法。与 Xilinx 的器件相比,Altera 的器件使用更少的布线资源。它使用的 LAB(逻辑阵列块)比 Xilinx 的 CLB 要复杂,并且在其芯片上可用的 LAB 的数量更少。那么,使用 Altera 公司的器件就一定比使用 Xilinx 公司的器件更好吗?这要看你打算做什么。部分设计人员偏爱使用 Altera 公司器件的一个原因是,由于不需要更多的布线资源,因此其布局/布线软件比 Xilinx 公司的“设计管理器”软件更易于使用。Altera 公司的软件具有速度快和很好的确定性;同样的设计,用相同的编译设置进行多次编译,每次都可得到相同的结果。在 Usenet 网站上,有人这么说:“我喜欢 Altera 公司的软件,同时喜爱 Xilinx 公司的芯片。”简单的陈述有深刻的含义。

由于 Altera 布线资源有限,因此完成一个设计的布线不是一件容易的事。设计者在设计过程中应注意不可过早地对引脚做出功能定义。随着设计过程的深化,有可能需要对原来已分配好的引脚进行再定义,但对于 Altera 的器件来说,这却是不太可能实现的。而对于 Xilinx/Lucent/Actel 公司的器件,这个问题要好办得多。

7.5.1　Altera 的 FLEX8K 器件架构

FLEX8K(灵活的逻辑单元矩阵)器件是基于 SRAM 的粗粒度结构(以大的逻辑单元块为基础),并且包含一个逻辑阵列块(LAB)阵列。每一个 LAB 由 8 个 LE(逻辑单元)、4 个控制输入信号(被用做时钟、置位/复位、进位输入、级联输入)、24 个全局逻辑输入、8 个局部反馈输入和 8 个输出组成。和 Xilinx 类似器件相比,本架构在 LAB 之间有很少的内部连线。例如,LAB 的快速进位和级联输出仅与 LAB 的右部相连,而快速进位和级联输入则仅被连接于 LAB 的左部。FLEX8000 系列器件不支持内部三态总线(编译器自动用 MUX 来替代内部的三态功能)。每个 LE 包括一个四输入的查找表和一个触发器。

在图 7-7 中,IOE 表示输入/输出单元(引脚缓冲器和寄存器)。IOE[①]的结构如图 7-8 所示。配置模式包括 JTAG、JAM(一个 Altera 公司正在扶植推广的串行配置标准)、主动/被动串行模式、主动/被动并行同步模式及异步模式。

7.5.2　Altera 的 FLEX10K 器件架构

FLEX10K 系列是 Altera 公司对原有的 FLEX8000 系列进行升级而得到的。FLEX10K 系列包含嵌入式 2048 比特位的双端口 RAM 块(嵌入式阵列块,即 EAB)。

① 应为 IE 的结构。——译者注

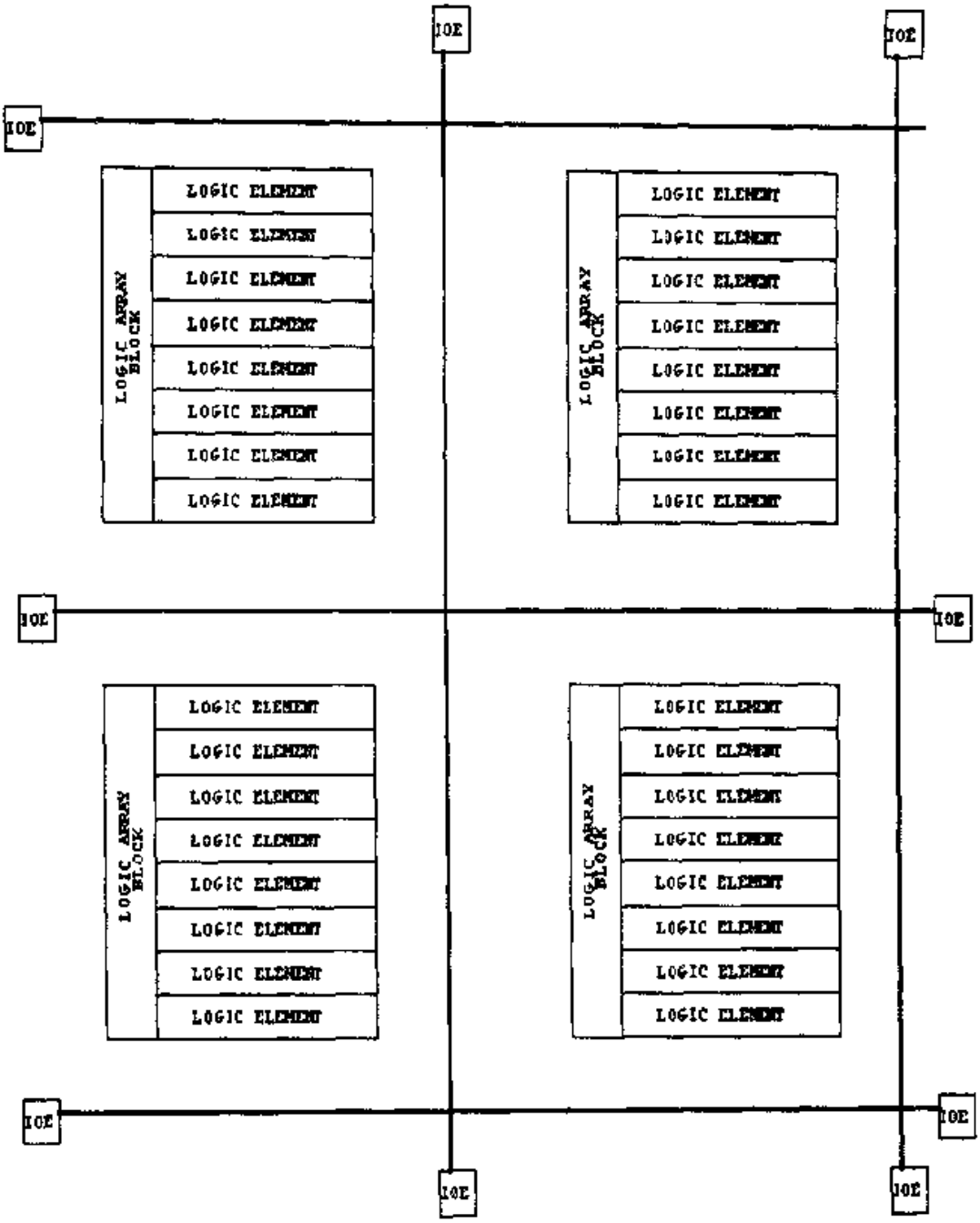

图 7-7 FLEX8000 器件的逻辑结构

这些 2048 比特位的 EAB 可以被分别用做 2048 × 1,1024 × 2,512 × 4 或 256 × 8 的阵列。EAB 可进行组合,形成规模更大,结构更复杂的存储单元。EAB 也可以利用其逻辑功能构造更大的查找表。EAB 还包括输入和输出寄存器。

FLEX10K 和 FLEX8K 的 LE 是类似的,其输出连接到快速布线通道(称作快速通道 FastTrack)上。此外,在 LE 的触发器上增加了时钟使能。

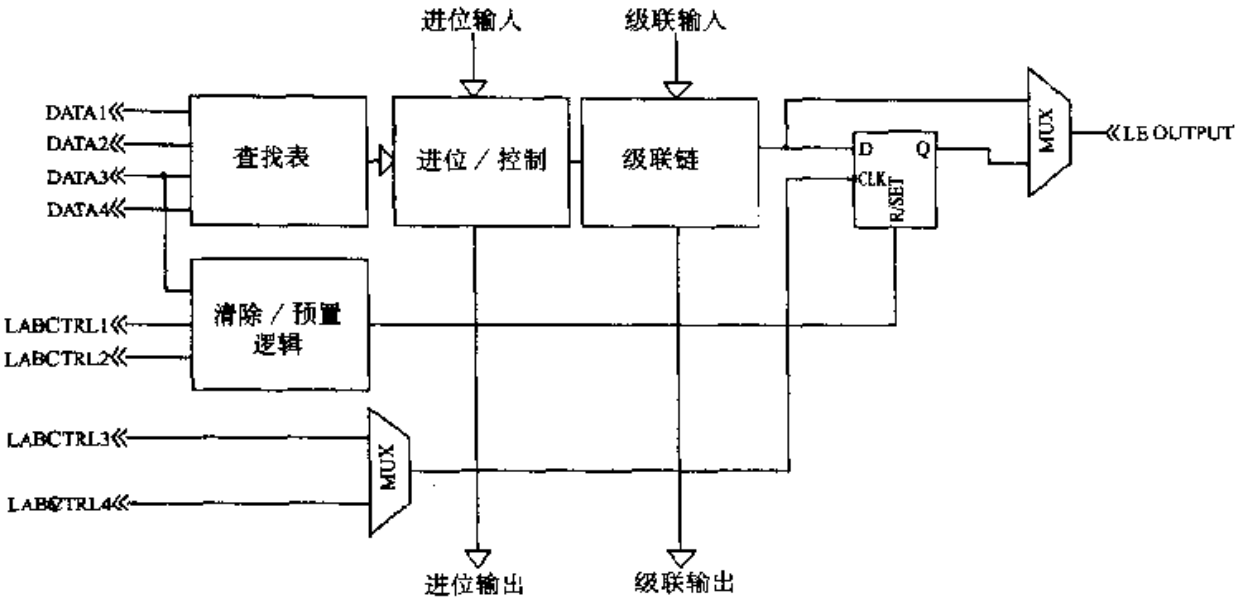

图 7-8　FLEX8000 器件的逻辑单元(LE)

7.5.3　Altera 的 APEX 20K 器件的架构

Altera 公司的 APEX 20K 系列是构建在 8K/10K 系列架构之上的，其逻辑结构如图 7-9 所示。此系列器件具有更高的集成度及许多先进的特性，如时钟锁相环(应用于时钟锁定，时钟乘法器和时钟相位变换)、多种逻辑结构的混合(包括 RAM

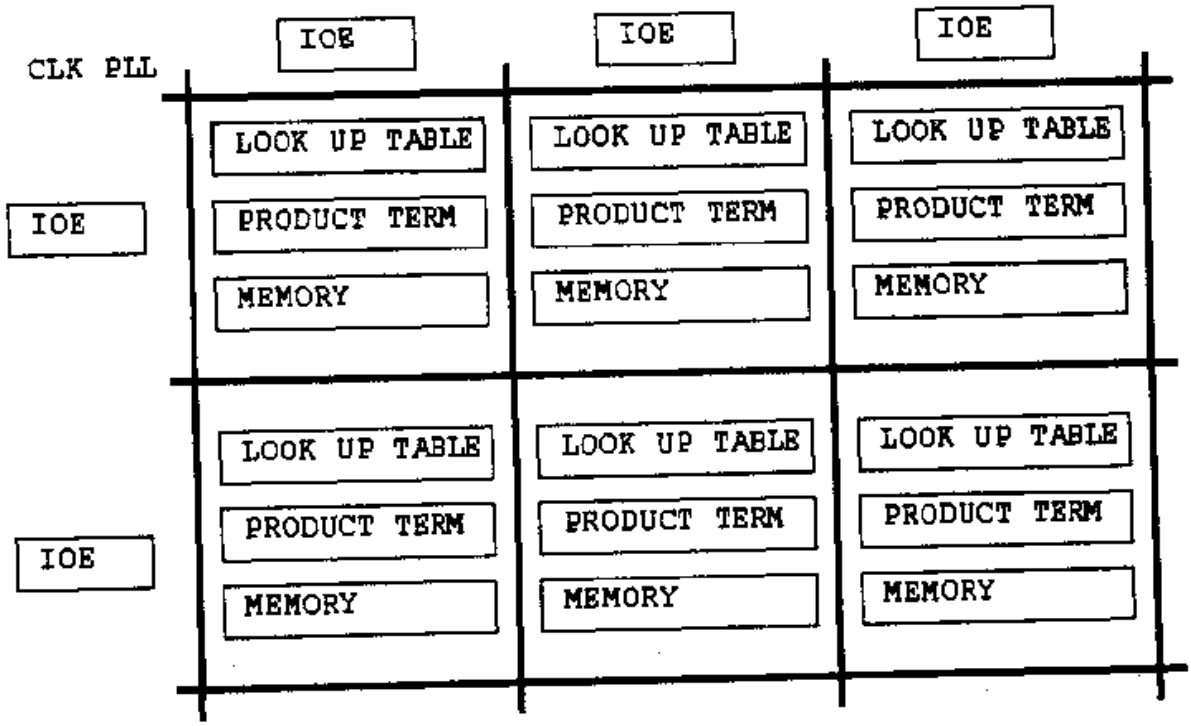

图 7-9　APEX 20K 系列的逻辑结构

和针对译码器的宽位 PLD 模式逻辑)、多种输入/输出模式。LAB 中的逻辑单元(LE)的数量由 FLEX 8K/10K 系列的 8 个增加为现在的 10 个。APEX 20K 系列和 FLEX 10K 系列的逻辑单元类似,但增加了同步负载和逻辑清除以及更多的时钟选择。

第八章　元件库、可再用模块及 IP

电子设计集成度的不断提高是一个发展趋势，近 40 年来这个趋势一直没有停止过。人们对于不断涌现的、精密的、尖端的小型电子产品(无论是 GPS、手机、游戏、家庭自动化、网络、Internet 商务、音频/视频娱乐，还是信息处理计算)的迷恋好像是永无止境的。人们更多的空闲时间是在和电子玩具玩耍。随着诸如此类需求的增长，业界提供晶体管和门电路的能力好像也变得无止境了。对于 FPGA 设计者来说，这意味着设计将包含更多的可用逻辑门。

设想一个工程师按一天约 100 个门的速度进行设计工作。这相当于用 Verilog 语言书写 10 行代码的工作量(这包括了用于测试和文档编写的开销)。用不了多久，FPGA 设计的平均水平将达到 20 万门。这意味着，一个用两个人干的活，除非改变设计方法，否则要花费 1000 天！设计任务的复杂度逐年增加，但设计流程却始终是一成不变。要想提高工作进度，不得不采取一些其他的措施。这里有几种方法可供选择。

8.1　生产率提高的关键

8.1.1　设计队伍的规模必须增大

具有最高生产率的团队是由 1～3 个专业设计人员构成的。如果公司允许有充足的时间等待产品问世的话，那就是最省钱的方法了。然而，大多数公司对效率和生产能力并不感兴趣，它们感兴趣的是如何尽可能快地把产品推向市场。因此，扩大团队是必然的。扩大团队的困难在于，随着团队中成员人数的线性增加，成员之间沟通的难度将按指数形式增加。如果团队中有两个成员，即 Jack 和 Jill，在工作上他们之间必须进行必要的协调，有两条沟通的途径(Jack 对 Jill 和 Jill 对 Jack)。如果又增加个成员 Jerry，那么 Jack 就不得不和 Jill 及 Jerry 两人沟通，同时 Jill 也要和 Jack 及 Jerry 进行沟通，依此类推。这你总该明白了吧？这个简单例子的结果就已经够令人头疼的了，其实在实际设计中的沟通比这还要繁杂和烦人得多。

为了使团队中所有的成员为了共同的目标而努力需要花费巨大的心血，最终团队中的每个成员的设计工作必须和该团队中其他成员的工作协调一致。这个效果的产生绝不是偶然的。为此，将需要开更多的会议(这将降低生产率)、编写更多的工作报告(这将降低生产率)、制定更多的技术规范，以确保设计中的各个部分之间能够彼此协调。但往往是旧矛盾解决了，新矛盾又来了，这肯定会降低生产率。

管理一个团队更像一门艺术而不是一门科学。虽然典型设计中的可用门的数量在以指数的形式增长，但能够把所有员工凝聚在一起却不是一件容易的事情。

请记住作者这句话：

> **COFFMAN(作者自己)定律**
> 一所房屋中所孕育的智慧，从平均意义上来看，和屋里的人的数量成反比。

8.1.2 单个设计者必须编写更多的代码

在硬件设计领域，20 世纪 40 年代的电子设计人员在进行设计时使用的是真空管。到了 50 年代，晶体管取代了真空管。60 年代，晶体管被集成电路(晶体管的 100 倍)所代替。如今，集成上百万上千万个晶体管的 IC 已相当普遍。因此，拿别人事先设计好的电路模块进行组合和匹配就可构造出属于你自己的设计，这对于硬件设计者来说不能不算是一件幸事。我们可以这样做：

(1) 使每行代码能够描述更多的实际电路元件。当综合工具变得越来越智能化，FPGA 设计变得越来越紧凑(因此，实现一个设计所需要用到的门的数量已变得不重要了，如果以牺牲门的数量为代价，换取设计的高速特性，这也是可以承受的)时，更高级别的电路结构的实现将成为可能。今后，我们将可能通过编写如下所示的一行代码来实现一个工作于 100MHz 频率的 1024 位加法器：

```
a = b + c;
```

设计一个超前进位加法器并不是仅仅依靠手工来完成，综合工具将会根据你的设计约束条件来推断采用何种加法器可以做到更加有效。

(2) 设计的重复使用。早期设计中所用到的模块将被包含在你现在的程序代码(这是最一般的重用方式)中，或通过购买或经由他人的许可而获得。业界中大量的工作集中在向 ASIC 设计者出售知识产权(IP)设计上，并且供货商也愿意向 FPGA 市场供应 IP 产品。这不难理解，因为就 ASIC 及设计自动化领域的上层人物看来，通过向希望减少产品进入市场时间的公司出售已有设计产品的方法可以给他们带来巨大的利润。

只要 IP 使用的“先尝后买”的市场模式存在，这个市场就会一直存在下去。这个市场的发展和集成电路的使用有着一定的联系。这种模式已经被成功应用了 30 多年，因此是成熟的。从设计者的观点来看，设计规范的详细程度、成本及不同 IP 产品的交付方式都在考虑之内，最终将选取一个最合适的产品。对于器件制造商来说，商业模式是值得考虑的问题。需要花费很大的投资，器件才能够被设计出来并投向市场。设计的产品化过程的前期投入(可能达到数百万美元)需要通过电子制造商购买该器件后才能收回来，而这个过程是漫长的。如果该设计被广泛采用的话，这个策略将带来丰厚的回报。好比这是一场游戏，IP 供应商玩得起这个

花费巨大的游戏吗?

IP 供应商必须能提供表征产品性能的完整数据,包括生产能力、延迟时间、信号的 I/O 要求、模块的尺寸、以及能量损耗。只有这样,才能确保设计满足实际要求,才能与其他产品进行较量。成功的 IP 产品应该是一个和操作系统无关的独立的并能满足设计者某种特定功能要求的模块,比如 FIFOs 和其他类型的基于内存的模块、微控制器、滤波器、压缩/解压缩功能块以及通信端口(UARTs、Ethernet、USB 等)。对于具有可行性的 IP 类型的深入的思考(请参阅后记)。

在我们为 IP 的使用欣喜若狂之前,让我们先来看看一个可以提高设计效率的最简单的方法:使用内嵌的库单元。

8.2 库 单 元

每个 FPGA 供应商都提供一组基本的库单元。Verilog 设计通过使用和此类似的原语被映射到硬件上。原语依据底层硬件的特性,以一种有效的形式得以实现。FPGA 提供商每年都会加进新的功能单元,原语的功能变得越来越强,这将对推广库的使用起到促进作用。

熟练的设计人员对于不同抽象层的设计,包括用来实现该设计的库单元的类型都应能做到心中有数。

下面是可能会遇到的各种库单元:

AND	与门
BPAD	双向端口
CKBUF	时钟缓冲器
FF	具有异步置位、复位及时钟使能端的 D 型触发器
INV	反相器
IPAD	输入端口
KEEPER	弱状态保持器(驱动源断开后保持最后一个状态值)
LATCH	具有异步置位和复位端的 D 型锁存器
LATCHE	具有异步置位/复位及门控制使能端的 D 型锁存器
LUT	查找表
MUX	多路转换器
ONE	逻辑"1"发生器
OPAD	输出端口
OR	或门
PD	下拉电阻
PU	上拉电阻

RAM	读/写存储器(可被用作只读存储器)
SFF	具有异步/同步置位/复位端的 D 型触发器
SRL16E	移位寄存器查找表
TRI	三态缓冲器
UPAD	未焊引脚
XOR	异或门
ZERO	逻辑"0"发生器

这些原语都有着其不同的用法。例如,以下是与门的几种用法:

AND2	无反相输入端的双输入与门
AND3	无反相输入端的三输入与门
AND4	无反相输入端的四输入与门
AND5	无反相输入端的五输入与门
AND6	无反相输入端的六输入与门
AND7	无反相输入端的七输入与门
AND8	无反相输入端的八输入与门
AND16	无反相输入端的十六输入与门
AND32	无反相输入端的三十二输入与门

Verilog 编译器也使用一套原语。它们和 FPGA 供应商提供的原语库有许多相似之处。

FALSE
TRUE
INV
BUF
AND2
OR2
XOR2
NAND2
NOR2
MUX
DFFRS
DFFERS
LATRS
RSLAT
TRI
PULLUP

PULLDN
TRSTMEM
DON'T _ CARE

以下是 Exemplar 的器件专有库列表，适用于 Xilinx 的 4000XL 系列。一般的原语都具有类似的元件库。

IBUF	输入缓冲器
OBUF	输出缓冲器
OBUF _ NG	反相输出缓冲器
OBUFT	三态输出缓冲器
OBUFT _ NG	反相三态输出缓冲器
OUTFF	输出触发器
OFDX	OC 高有效的输出 D 触发器
OFDX1	OC 低有效的输出 D 触发器
OFDTX	具有三态输出的输出触发器
OFDTXI	OC 低有效的三态输出 D 触发器
OFD	输出 D 触发器
OFD _ NG	具有反相输出端的输出 D 触发器
OFDX _ NG	具有反相三态输出端的输出 D 触发器
OFDI	
OFDI _ NG	
OFDXI _ NG	
OFDT	
OFDT _ NG	
OFDTX _ NG	
OUTFFT	
OFDTI	
OFDTI _ NG	
OFDTXI _ NG	
IFD	输入 D 触发器
IFDX	
INFF	输入触发器
IFD _ NG	
IFDX _ NG	
IFDI	
IFDXI	

IFDI _ NG	
IFDXI _ NG	
INLAT	输入锁存器
ILD _ 1	
ILDX _ 1	
ILD _ 1 _ NG	
ILDX _ 1 _ NG	
ILD	
ILDX	
ILDI	
ILDXI	
ILDI _ 1	
ILDXI _ 1	
ILDI _ 1 _ NG	
ILDXI _ 1 _ NG	
INREG	输入寄存器
DFF	D 触发器
FDPE	
FDCE	
FD	
FD _ GP	
FDP	
FD _ NGP	
FD _ NG	
FDC	
FDC _ NG	
FDCE _ NG	
FDE	
FDE _ GP	
FDE _ NGP	
FDE _ NG	
FDP _ NG	
FDPE _ NG	
OUTFFT _ IBUF	三态输出触发器和输入缓冲器
OUTFFTX _ IBUF	

OBUFT _ INFF _ 1	具有输入触发器的三态输出缓冲器
OBUFT _ INFFX _ 1	
OBUFT _ INLAT _ 1	具有输入锁存器的三态输出缓存器
OUTFFT _ INFF _ 1	具有输入触发器的三态输出缓存器
OUTFFTX _ INFF _ 1	
OUTFFT _ INFFX _ 1	
OUTFFTX _ INFFX _ 1	
OUTFFT _ INLAT _ 1	具有输入锁存器的三态输出触发器
OUTFFTX _ INLAT _ 1	
DLAT	D 锁存器
LDCE _ 1	
LDPE _ 1	
LD	
LD _ NG	
LD _ 1	
LDC	
LDC _ NG	
LDCE	
LDPE	
LDP	
LDP _ NG	
LD _ NGP	
BUFGLS	
BUFGE	具有使能端的全局缓存器
BUFFCLK	时钟缓存器
BUFCS	全局资源缓存器(二级)
BUFGP	全局资源缓存器(一级)
BUFG	全局网络缓存器
BDBUF	双向缓存器

8.3　结构化编程模式

如果你登录数字设计新闻组(见本书后的资源部分)网站的话,你将会不时发现一些偏爱原理图设计的人,他们认为,使用原理图通常可以实现有效的设计。这个观点可能是正确的,因此,对于那些只使用 Verilog 语言的人来说,通过将原语组

合为文本文件的方式绘制原理图还是有一定意义的。正确使用这个方法,将得到一个非常紧凑和快速的逻辑设计。然而很快就会发现它并不那么容易使用,所以我们仅在必要时才采用这种设计方法。

当 IP 作为商品提供给设计领域并在设计中应用时,则要求其具有一定的可移植性,因为 IP 技术和硬件描述语言(HDL)彼此应该是相互依赖的。而原理图设计的一个明显缺点恰恰是:可移植性不好。

程序列表 8-1 是一个库原语的结构化应用实例。图 8-1 给出了相应的逻辑综合后的电路图,Xilinx 的结构资源分配情况见图 8-2。

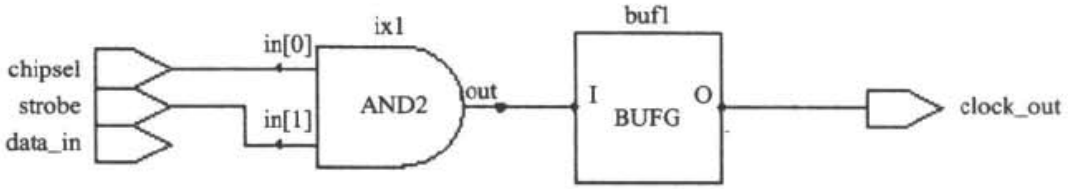

图 8-1 BUFG 结构化资源分配

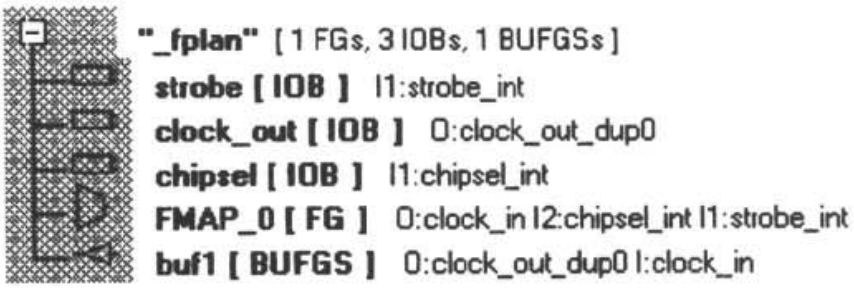

图 8-2 BUFG 结构化资源分配(以上资料来自于 Xilinx 的布局规划工具)

程序列表 8-1 使用库原语的 Verilog 结构性设计

```
//库原语的结构化实例。

module 全局缓冲器 (clock _ out, data _ out, chipsel, strobe, data _ in);

input  chipsel, strobe, data _ in;
output  data _ out;
reg  data _ out;
wire  clock _ in;
```

```
output    clock _ out;

assign    clock _ in  =  chipsel & strobe;

BUFG bufl  (.I(clock _ in), .O(clock _ out));

endmodule

// 为缓冲器创建黑匣子。
module   BUFG (I, O);
input    I;
output   O;
endmodule
```

图 8-2 是全局缓存器设计中用到的资源列表,从中可以看到 BUF1 被定义实现为一个 BUFGS(这是在 Xilinx 4000XL 系列器件中惟一可用的全局缓冲器类型)。

8.4　原理图设计和 Verilog 语言设计的比较

图 8-3 给出了一个简单 RAM 器件实现的示意图。将此示意图和相同设计的 Verilog 结构化设计程序(见程序列表 8-2)相比较是件有趣的事情。有关 LogiBLOX 工具更详细的介绍参见 8.5 节。

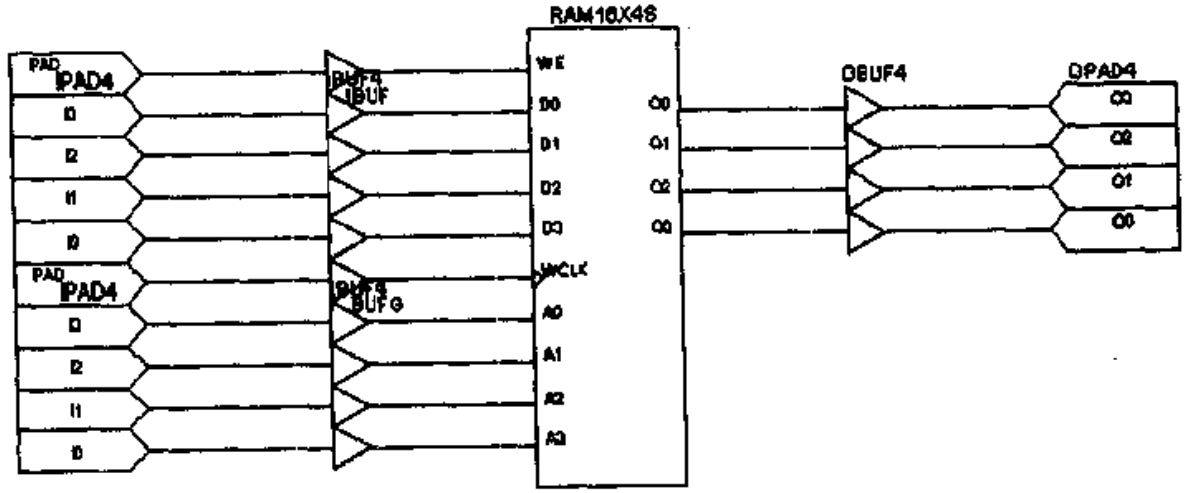

图 8-3　使用库原语模块的电路示意图

程序列表 8-2 Verilog 结构化示意图实例

```
// 结构化示意图设计实例。
module schematic(out_data, in_data, in_addr, clock, write_enable);
input [3:0] in_data, in_addr;
input clock, write_enable;
output [3:0] out_data;
// 定义和"黑匣子"ram_module 的界面。
// 此空匣子将用预定义的网表描述来填充。
// 由 Xilinx 的 LogiBLOX 工具生成的 RAM 块。
//-----------------------------------------
// LogiBLOX SYNC_RAM Module" ram_module"
// Created by LogiBLOX version M1.5.19(由 LogiBLOX M1.5.19 版生成)。
//   on Mon Dec 28 17:21:11 1998(生成日期)。
// Attributes (属性)。
//   MODTYPE = SYNC_RAM
//   BUS_WIDTH = 4
//   DEPTH = 16
//   STYLE = MAX_SPEED
//   USE_RPM = FALSE
//-----------------------------------------
ram_module u1
(.A(in_addr),
   .DO(out_data),
   .DI(in_data),
   .WR_EN(write_enable),
   .WR_CLK(clock) );
endmodule

module ram_module(A, DO, DI, WR_EN, WR_CLK);
input [3:0] A;
output [3:0] DO;
input [3:0] DI;
input WR_EN, WR_CLK;
endmodule
```

图 8-4 给出了 LogiBLOX 工具的主菜单。

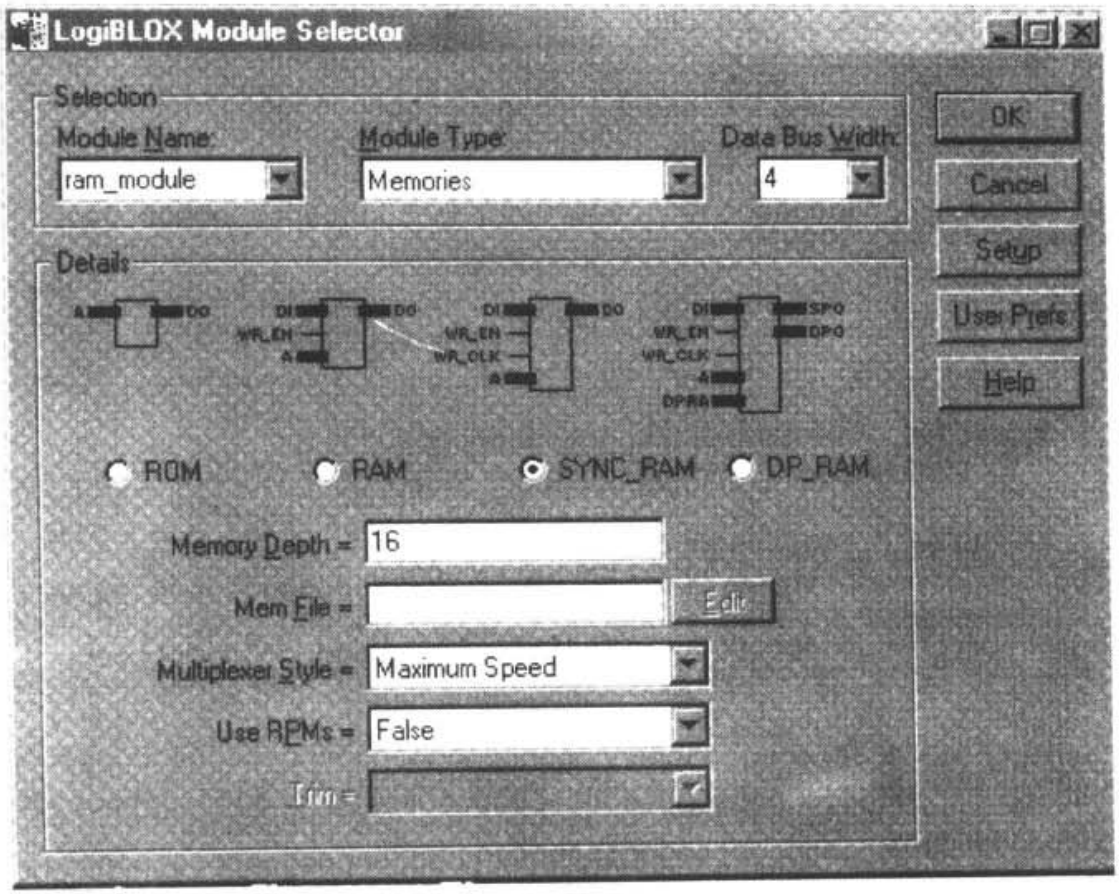

图 8-4　使用 LogiBLOX 工具生成一个 RAM 模块

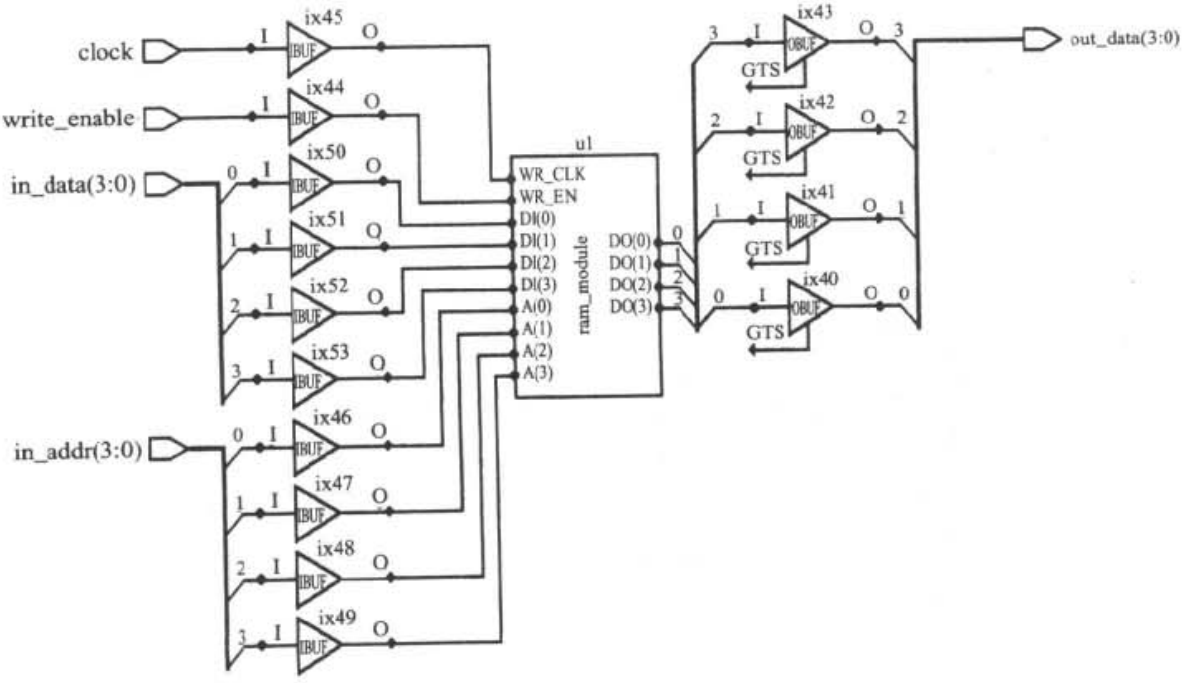

图 8-5　用 LogiBLOX 生成的 RAM 模块的结构化电路示意图

图 8-5 给出了对 LogiBLOX 生成模块的应用。

HDL 方法和原理图方法哪个更好呢？哪个生成速度更快、可移植性更好、更容易理解呢？哪个设计得更巧妙呢？哪个具有更好的可移植性呢？注意，编译器会根据设计的实现要求去判定缓冲器。设计者必须在电路图中标出缓存器。

8.5 使用 LogiBLOX 模块生成器

RAM 模块可以成为一种提高设计速度的手段。RAM 模块不是凭空创造出来的；它们是借助设计工具生成的。在此，我们使用 Xilinx 的 LogiBLOX 工具去创建 RAM 模块。其他的模块也可以被生成和参数化，这些模块包括：

Accumulators
Adders/Subtractors
Clock Dividers
Comparators
Constants
Counters
Data Registers
Decoders
Inputs/Outputs
Memories
Multiplexers
Pads
Shift Registers
Simple Gates
Tristate Buffers

对于 HDL 设计者来说，在其结构性设计中使用这些原理图型单元块将使得他们可以应用和特定硬件相关的硬件配置选项。例如，在三态缓存器逻辑块定义下，对上拉电阻有三个选项：无、上拉、双上拉。但对某些选项，如双上拉选项，Verilog 不提供直接的支持，但一些设计却有此要求。在某些场合要求使用结构性设计，在这种设计中，硬件描述语言(HDL)以结构化(原理图)形式被应用，或者 Verilog 模块由原理图拼凑而成。一句话，什么更有效就使用什么。

因此，提高设计效率的一个方法就是使用工具软件去自动生成特定形式的模块。

8.6　另一种模块生成器：CORE Generator 工具

Xilinx 公司(和 MEMEC 公司)提供了一种名为内核生成器(core generator)的工具，和 LogiBLOX 相比，这个工具提供了更复杂、更多种类的功能模块供设计者使用。包括：

• FPGA 开发工具(用于 DSP 和 FPGA 设计评估及基准的 DSP、FPGA 开发平台)。

• 处理器外围包括：C2910A 位片处理器内核、DRAM 控制器、M8237 DMA 控制器、M8254 可编程时序控制器、M8255 可编程外围接口、M8259 可编程中断控制器、XF8256 多功能微处理器支持控制器、XF8279 可编程键盘显示接口。

• 处理器产品包括：Intellicore™原型系统、RISC CPU 核心演示系统、可扩展的开发平台、TX400 系列 RISC CPU 内核、V8 uRISC 微处理器。

• UARTS 包括：M16450、M16550A 和 XF8250。

• 通信和网络核心包括：ATM 信元装配器、ATM 信元描述、ATM CRC10 生成器及校验器、ATM CRC32 生成器和校验器、ATM 从机(Utopia Slave，CC-141)、前向纠错 reed-Solomon 解码/编码器、Viterbi 解码器、HDLC 电信协议核心、MT1FT1 电信成帧器。

• XF9128 视频终端逻辑控制器。

• 标准的总线接口内核包括：IEEE 1394 火线连接层内核、火线 SuperLINK 内核试验电路板、双线串行接口、PCMCIA 内核、USB 内核。

• 其他。

这些内核是 Xilinx 支持模块(免费使用的)和第三方支持模块(须授权的)的混合体。这些内核的一个应用实例是一个称做 sincos8 的基于 8X8 Sin/Cos 查找表内核模式的设计。CORE Generator 工具生成一个名为 sincos8.vei 的 Verilog 接口文件。此文件定义了内核将要使用到的端口，并且为方便设计者的使用提供了一个模块实例。sincos8.vei 文件的形式见程序列表 8-3。

程序列表 8-3　sincos8.vei 文件

```
module sincos8 (
    ctrl,
    theta,
    c,
    dout);
```

```
input ctrl;
input [7:0] theta;
input c;
output [7:0] dout;
endmodule

// 下面是一个实例:
sincos8 YourInstanceName(
        .ctrl(ctrl),
        .theta(theta),
        .c(c),
        .dout(dout));
```

另外,CORE Generator 工具文件也将生成一个 Verilog 仿真文件,名为 sincos8.v。这个文件有 5287 行代码。如果我们一天写 100 行调试代码,则完成这个模块需用两个月的时间,而用 CORE Generator 工具仅需大约 5s。如果这样生成的模块可以满足设计要求的话,将不失为一种提高生产率的手段。但若它不能正常工作,则我们将得不到任何源代码,这个 CORE Generator 工具也就对设计工作起不到任何帮助作用。

链接到设计中的文件是经过编译后的 EDIF 网表文件,文件名为 sincos8.edf。为了便于理解,我们用一个 4000XL 器件来实现这个设计,并观察其结果如何。在设计工具的使用方面总是有一些小窍门的,在这种情况下,我们需要使用圆括号 B( )作为总线分隔符(XNF 网表格式则使用 B◇作为分隔符)以使 Xilinx 设计管理器可以正确识别和添加 EDIF 文件。我们可以通过不选择图 8-6 中"Verilog Instantiation Template"和"Verilog Behavioral Simulation Model"复选框来实现此目的(即使我们想生成这些文件),以得到我们想要的如图所示的网表总线模式。此外,在 Exemplar Leonardo 的 EDIF 输出标签中,需取消"Allow Writing Busses"复选框,因为 Xilinx 不能正确处理 EDIF 总线。

要注意的是,Verilog 编译器并不"认识"任何有关黑匣子的内容(在后端映射过程中被插入,如图 8-6 所示)。任何使用逻辑综合工具进行的有关速度和设计规模大小的评估都不包括黑匣子模块。

简单测试表明,此设计在最慢的 4005XL(-3)系列器件上的工作频率为 61MHz。程序列表 8-4 给出了此设计所用资源的报告。

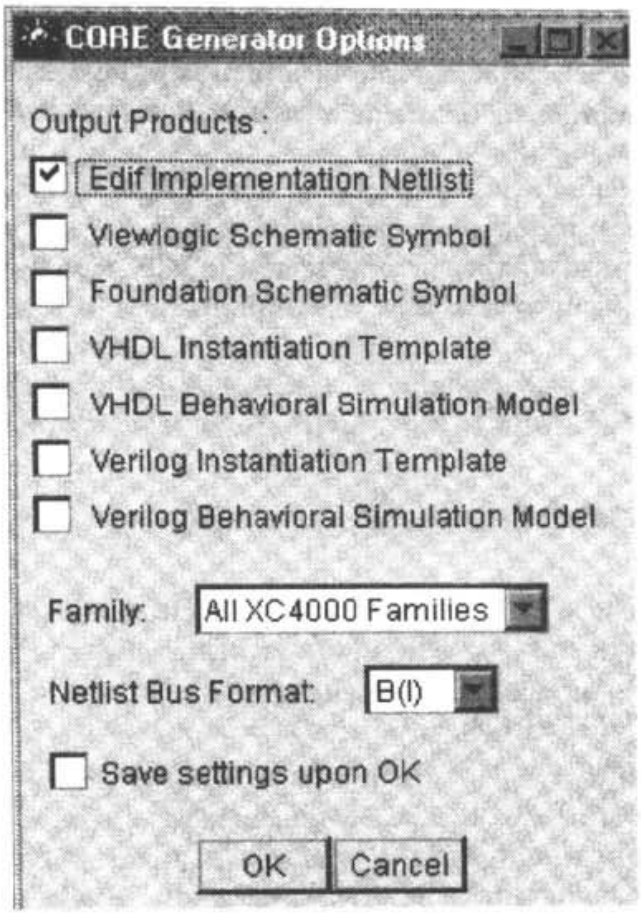

图 8-6　有关网表总线格式的 CORE Generator 选项

程序列表 8-4　sincos8 设计实例

```
Loading device database for application par from file" map.ncd".
sincos8 _ example is NCD, device xc4005xl, package pc84, speed - 3
Loading device from file '4005xl.nph' in environment C:/Xilinx.
Device speed data version: xl _ 0.37 1.22 FINAL.

Device utilization summary:
Number of External IOBs          18 out of 61          29%
   Flops:     0
 Latches:     0

Number of CLBs:                  25 out of 196         12%
 Total Latches                   0 out of 392          0%
```

Total CLB Flops:	25 out of 392	6%
4 input LUTs:	43 out of 392	10%
3 input LUTs:	4 out of 196	2%
Number of TBUFs:	28 out of 448	6%

另一个例子是一个 8 位宽,16 位深的名为 fifo8x16 的 FIFO。Verilog 仿真文件有 293 行代码。程序列表 8-5 给出了接口文件。

程序列表 8-5 fifo8x16.vei 文件

```
module fifo8x16 (
        d,
        we,
        re,
        reset,
        c,
        full,
        empty,
        bufctr_ce,
        bufctr_updn,
        q);

input  [7:0] d;
input  we, re, reset, c;
output full;
output empty;
output bufctr_ce;
output bufctr_updn;
output [7:0] q;
endmodule

// 以下是一个实例:

fifo8x16 YourInstanceName (
        .d(d),
```

```
.we(we),
.re(re),
.reset(reset),
.c(c),
.full(full),
.empty(empty),
.bufctr_ce(bufctr_ce),
.bufctr_updn(bufctr_updn),
.q(q));
```

可以看出，使用 CORE Generator 是一个创建复杂模块的有效方法，并可提高设计的效率。这是一个碰运气的过程，因为如果在尝试过所有可用的编译器选项后，一个模块仍不能满足设计的需要的话，则由于没有源代码，无法对设计进行更改，不得不采用其他的方法（比如从头做起，花时间设计一个最优化的模块）。

8.7　设计的再用，重新使用你自己的代码

当你在从事设计工作时，你会发现你将重新使用到你原先的设计方法，甚至是特定的模块。如果这些都是你曾经设计过的，你肯定对它们的特点和局限性很熟悉，是利用已有设计还是重新进行设计，几乎可以凭直觉做出判断。

采用代码再用法进行设计时的注意事项。

- 对设计加入必要的注释。

在设计的顶部设置一个报头，用来对下面的设计进行描述。这个报头应当涉及设计的输入和输出要求及设计中将用到的所有技巧或特殊含义的词汇。应当加入大量的注释，而这些注释不应仅仅描述每条代码的含义，还应对解决问题的整体意图和设计路线进行解释。

- 使用具有版本控制功能的数据库产品，如 SourceSafe 或 VCS。

对原有设计创建一个完备的数据库，并遵循对修改的部分进行详细注释的原则，这将增强 Verilog 语言代码的可重复使用性。这些产品具备清除所曾做过的修改的功能，并确保以前的某个版本的设计可以被复原。这个特性可以大大降低设计风险。在和其他设计者协作进行一项设计工作时，这些工具的使用就显得更加重要了。

- 把逻辑单元划分为多个小模块。

专门用于某种特定功能的小模块比那些复杂而专业化的大模块具有更好的可重复使用性。

- 应用同步设计技术。

同步设计具有更高的可靠性和可移植性。对于那些必须工作于异步方式下的模块，应把它们孤立出来并做详细的记录；切勿将它们和其他运用同步方式的代码部分混杂在一起。

• 熟练键盘输入。

在键盘录入冗长的、描述性的标记时，你要么有很好的耐性，要么应当是一个打字高手。如果你的打字速度比较慢的话，不要使用诸如 video _ output _ enable _ active _ low 这样过于简单却含义复杂的标记。熟练运用英语会使你所编写的代码更具可读性。还要注意尽量减少使用缩略语。

• 正确使用时钟。

避免采用选通时钟电路或同时使用一个时钟脉冲的上下沿。

• 避免怪异数字。

怪异的数字通常是以常量的形式出现在代码中，例如：

(test _ pattern = = 4'he ;//(4'he)就是一个怪异数字的例子)

它们通常是参数，在高层或包含它的文件中可以被更改。

• 减少端口数量。

在进行模块分区时要尽量减少模块间的相互联接，特别是在时钟信号自始至终起作用的地方更应注意。应当使用自然的分界线来划分设计，就像剥橙子一样，将复杂的模块划分为许多小块是明智之举。

• 避免随意修改。

如果在一个工作模块中你发现有些是你不喜欢的，不要去管它。无意中的操作而造成损失的可能性是很大的，因此在做任何改动之前一定要弄清楚是不是真的有问题，是不是一定要修改。看似简洁巧妙的编码不见得是更好的编码。

• 不妨请教他人。

如果你遇到一个困难不知该如何去做，或者需要在两个相似的选项中做出选择时，你应当把你的问题告诉你的同事，或者登录网上的新闻组，或者请教现场应用工程师，甚至从你的邻居那里获取信息。即使不着边际也不要紧，从另外的思路去思考，说不准能激发你的灵感，找到一个更好的解决方案。

• 设计文件归档。

保留过去的设计底稿、数据库，以及所有与编译和实现设计有关的软件。

8.8 购买 IP 设计

IP 是什么样子呢？从用户角度来看，必须知道 IP 的接口定义，包括时钟信号(极性，包括使用上升沿还是下降沿或同时使用两个边沿、最大和最小频率、占空比、负载)，复位/预置信号(极性、同步或异步、信号保持和加载)，以及对其他端口

的要求。

在为 FPGA 设计而购买 IP(或称之为 Revenue IP,或称之为 Silicon IP)时,最大的问题不是在技术上,而是在如何协商取得使用许可方面。一次性支付应当定为多少钱?分期支付(专利权使用费)又应当定为多少呢?当产品产量高于或低于预期值时,应采取何种成本模式应对计划以外的花费呢?花费开销如何进行结算呢?如何能在有效保护 IP 供应商投资的同时,仍能从他们那里得到足够的数据以确保设计的成功实现呢?所有这些问题都应予以重视,以使 IP 应用于设计成为可能。

Revenue IP(RIP)将成为 FPGA 设计者生活中的一个重要组成部分吗?作为硬件设计人员,我们正在方便地使用着集成电路形式的硬 IP,其集成度往往超出了我们的实际需要。我们不是 ASIC 设计人员,因此不必使用昂贵的设计工具并且也没有机会直接接触实际的生产线。我们能够设计出功能相当的产品,但可能要用更大的电路板、更长的设计周期,乃至更多的资金。前两个问题可以通过采用 FPGA器件予以消除。设计周期可能长一些,不过想想办法节约开支,权衡之后成本同样可以降低。我们需要对当前的 IP 策略有个大致的了解。对于 FPGA 设计者,常用的两种 IP 类型是:硬/固 IP 和软 IP。

8.8.1　硬/固 IP

在 ASIC 行业,所谓硬 IP 就像标准单元一样,是一个经过预先设计并适合某个特定制作过程的内核。这种产品不是专为 FPGA 设计的。我们所能接触到的最常用的产品是硬/固 IP(一种已进行了预布线和预布局的模块,它可以和其他模块进行链接)。硬/固 IP 的使用很像是在设计中使用集成电路块。从用户的角度看,它是一个黑匣子。用户不能对其进行修改;它仅仅允许被嵌入一个设计之中,和所有其他电路以及环绕它的布线构成一个整体。这些模块提供行为模式,允许其被评估和测试。从供应商的角度看,这种 IP 可以被完整描述,其时序特性完全可预测,同时也是最安全的,因为要想对它进行反编译或想稍加修改就据为己有都是非常困难的。而从用户的角度看,硬 IP 表现的并不友好,它只具有单一的解决方案,除了使用内嵌的配置选项外不再具有其他的灵活性。除非经过 IP 提供商的重新编译,否则无法适应新的加工处理过程或工艺技术。

8.8.2　软 IP

从用户的观点来看,能够拥有可以被任意修改、编辑和综合的源代码实在是一件十分惬意的事情。但问题是,IP 提供商如何能够确保拿到用户所应支付的费用呢?预先收费?不太可能。如果一个设计的 47% 是使用 IP 产品,另外的 53% 是靠用户自己设计完成的,那又该如何收费呢?当设计(可能非常复杂)被修改时,如何使设计者避免因为改动而产生的时序困难呢?谁又应对此负责呢?

为了保护 IP 提供商的利益，软 IP 产品应运而生了。所谓软性 IP 就是其产品可以被加密或进行模糊化处理(去掉注释，将有含义的标签用截短的和无意义的符号代替，并且将代码压缩，以至于根本无法阅读)，以使得它可以被合成和植入设计的其他部分之中，同时又不易被反编译。

8.9 总　　结

对于 FPGA 设计者来说，设计重复使用最一般的形式是利用自己原有的设计模块。我们已经介绍了几种用于提高编码的可重复使用性的方法。另一个重复使用方法是采用你所在公司其他工程师所设计编写的模块，这可以避免额外的花费及合法性问题的产生。为提高生产效率所采用的最一般的工具是供应商提供的元件库及内核生成工具。在进行 FPGA 设计时，对于有经验的设计者，他知道如何借用已有的设计，外购 IP 只占他预算的一小部分。

第九章　面向 ASIC 转化的设计

把 FPGA 设计转换到 ASIC 有许多好处，包括集成多个 FPGA 变成一个 ASIC，产生一个低耗、高速器件；当然最主要的好处是减少成本。ASIC 的成本（即使包括不可修复的因素）将少于 FPGA 成本的 1/3。是什么驱使我们决定把 FPGA 设计转换成 ASIC 呢？

如果有以下的情况，请考虑转换：

- 年使用量大于 1000 片。
- 设计不大可能被要求修改。
- 期望额外的保护而不被反编译。
- 需要改进速度和降低功耗（与 FPGA 相比较）。

为了容易转换和降低前期成本，FPGA 转换成定制器件有三种选择：硬布线的 FPGA、使用激光编程的 FPGA 转换（或定制布线的器件）和全 ASIC 设计。因为所有 ASIC 公司都使用 Verilog 语言，对它感到方便，所以用 Verilog 作为设计和仿真工具使得完成 ASIC 的转换变得很容易。

FPGA 不是非常好的 ASIC 原型器件，但它每年都在改进，变得更像 ASIC。因为集成度增加和未来成本的减少，所以设计仍将用 FPGA 实现。我们大多数的设计仍将被转换成 ASIC。FPGA 正变得越来越便宜和高集成度，因此 ASIC 技术也改进了。

为什么 FPGA 是一个不好的 ASIC 原型呢？有以下几个方面的问题。

- 像设计“火车轮子”一样，FPGA 提供商的设计对那些囿于陈规的人来说，大大提高了成功的机会。特别地，时钟网络的延迟设计要求触发器具有零保持时间。FPGA 设计师把精力集中在如何满足信号建立时间的要求方面，而 ASIC 设计师则必须要努力使得信号建立和保持时间两者的窗口要求同时得到满足。
- FPGA 为时钟和复位/预置提供了低时偏的全局网线，这些网线必须在 ASIC 设计中产生。
- 对 ASIC 设计来说，FPGA 的实验性设计模式（一些事不能确定，试试，看它会发生什么）可能带来致命的错误。考虑到掩模费用和研制周期，ASIC 中的一个错误就意味着巨大的成本付出，因此要求仔细、小心谨慎，甚至保守的设计方法，以及全面的测试。
- 把逻辑写进 FPGA，然后让它快速运行是很困难的。ASIC 拥有惟一的设计所需要的资源（FPGA 内布线和逻辑资源是否存在取决于它们有没有被使用），那就

是更小的容积、更低的功耗、更快的运行，因此要花费多一点的时间对 FPGA 优化设计。

尽管存在这些困难，但 FPGA 到 ASIC 的成功转换每天都在发生，使用一些众所周知的设计手段将使这个过程变得顺利。下面，让我们来看一下 FPGA-ASIC 转换所涉及的技术。

首先是硬布线(HardWire)器件。在 Xilinx 的 FPGA 中，大约有一半的硅片是用来产生可编程布线网络的，Xilinx 提供了定制硬布线版本的 FPGA，除了布线被定制的金属布线替代外，它的 CLB 与常规的 FPGA 是一样的(捆绑在一起)。即使对设计做最不起眼的修改(器件使用与 FPGA 同样的布局和信号布线)，转换所要的时间也要一个月左右。最小的订单至少 1000 片。封装和引脚(包括电源和接地)与原始的 FPGA 完全一样，能使用配置信号仿真。例如，在电路板上配置 DONE 引脚用来控制处理器的复位信号。虽然硬布线器件不要求配置，但一旦需要有配置，引脚就可以用。硬布线器件和 FPGA 一样，在相同的生产线上制作，因此工艺技术(制版)、引脚的驱动性能、引脚的电压容限和 CLB 布局是相同的。

因为硬布线硅片与 FPGA 是相似的，硬布线设计可以从配置文件(文件后缀名为 .bit)中捕捉到，但转换仍要求源设计信息，因为它们将在转换过程中被采用。

在生产测试期间，可配置器件的测试方案可以被编程。但硬布线器件必须有特殊的设计测试支持，这是硬布线器件的一个不足之处。Xilinx 提供了 ATPG(automated test pattern generation，自动测试代码生成)和边界扫描测试能力。

Altera 公司(产品有 MPLD，即 Masked PLD)和 Lucent 公司(产品有 MACO，即 Masked Array Conversion for ORCA)都以它们的技术提供了相似的器件。

这里需提醒读者注意的是，转换为硬布线设计对 FPGA 设计者来说不费多少劲。Xilinx 能保证硬布线器件和 FPGA 器件的设计一样。FPGA 能够屏蔽产生尖峰脉冲干扰的环境，因为具有电容性负载的信号布线晶体管充当了低通滤波器(RC)的作用。而在硬布线器件中，这种效应将被大打折扣。由异步信号产生的在 FPGA 设计中被"过滤"掉的干扰环境仍会引起尖峰脉冲干扰。在转换过程中，Xilinx 公司将标记出异步信号，但采取措施避免隐患则需要由设计者自己来完成。

9.1 半定制器件

有多种技术可以实现按某种模式配置的逻辑阵列，从而使利用激光编程方式进行定制布线成为可能。Chip Express 公司提供了具有激光编程布线功能的快速编程器 (LPGA，即 Laser Personalized Gate Array，供用户操作的激光门阵列)，它可以被转换成带有一层或两层金属布线层的器件。Clear Logic 公司也提供了类似装置(LPLD，即 Laser Processed Logic Device，激光处理逻辑器)。他们采用了折中方法，其

设计同硬布线转换非常相似。

9.1.1　半定制 ASIC 转换

AMI(American Microsystem) 和 Orbit 这样的提供商,都提供了把 FPGA 转换为它们的门阵列设计的手段。这样处理使研制周期缩短(4 ~ 6 周),研制成本降低(5000 ~ 50000 美元)。这些公司对 FPGA 转换有许多经验,能使转换过程变得顺利、平滑。

9.1.2　全定制 ASIC 转换

在 FPGA 设计中,因为 FPGA 是通过预定义的结构来实现设计要求的,所以设计者能做的事情是有限的。而 ASIC 具有更大的自由度,不像"火车轮子"那样必须在轨道上跑。所有我们假定的特性,如可编程缓冲器、端电阻、内置振荡缓冲器和上电复位/预置等,除非在 ASIC 中对此有特殊要求,都将不再提及。因为布线是全定制的,并且仅仅那些实际被采用的门才获得配置,所以 ASIC 有其自身的一些优势。另外,ASIC 的集成度更高,因为原来需用多片 FPGA 来实现的设计,现在可以集成于一片全定制 ASIC 上。

9.1.3　转换需求列表

设计者必须提供有关转换过程的必要的信息。这些信息包含在 ASIC 提供商提供的清单中,其中应包含如下项目:

- 设计网表。
- 测试程序和仿真结果。

供应商愿意提供 Verilog 的测试程序,提供的越多,转换期间出现问题的风险就越低。

- 封装、引脚数目、引脚的形状、引脚的间距。
- 时钟和时钟频率列表。
- 所需门数。
- 温度范围和特殊环境要求(如军用规范等)。
- 引脚列表:引脚名和引脚位置,包括电源、接地、配置和未使用的引脚。
- 特殊的特性:如上拉或下拉电阻、关键时序路径、引脚驱动要求、RAM 和 ROM、FIFO 和其他专用逻辑模块。

9.2　ASIC 转换的设计准则

向 ASIC 转换的过程使设计者备感压力,可以说是"危机四伏"。一些可以看到

的隐患包括时延网络、竞争环境、组合反馈、脉冲发生器、浮动的内部总线、时钟偏移,以及选通或分路时钟。

大多数提供商提供了类似"恢复键"的转换过程,在此设计流程中,ASIC 提供商负责整个转换,并提供全部测试向量。和"共同设计"转换模式相比,这将花费更长的时间和更多的资金,而在"共同设计"中,FPGA 设计者须提供全部或部分测试向量,并且承担转换的责任。

AMI(American Microsystems, Inc.)公司提供了一种非向量转换模式,对不喜欢仿真的 FPGA 设计者来说这是一种最无痛苦的转换方式。然而,他们都必须遵从如下所述的近乎苛刻的规则:

- 只采用 Altera、Xilinx 和 Actel 的器件。
- 惟一的外部主时钟。
- 不允许有组合反馈环路。
- 不允许有延迟依赖或脉冲生成器。
- 惟一的外部主置位/复位信号。

9.3 同步设计规则

第一是进行同步设计。这并非一件易事,设计中每个增加的时钟都必须要细心地考虑。每个时钟区间、每个穿过时钟域边界的信号和每个异步信号,除非进行全面而缜密的设计,否则它们都将成为设计的隐患。如果设计的目的是从一个时钟区间转换到另一个时钟区间(如 FIFO 所完成的功能那样),那么显而易见,你将别无选择。如果你希望降低功耗,但设计的某些部分又需要高速运行,那么你也将没有选择。笔者的意见是使设计运行在一个尽可能低的速度上,这样将减少 RFI 辐射,并对一般的 HDL 设计所固有的低效性具有更强的适应能力。如果设计的某一部分必须工作于异步状态,则应对这部分进行单独处理,使其和设计中的同步部分隔离开来;并对此进行详细的记录,使得设计意图和隐患一目了然。

处理异步信号并不困难,但却容易被人所忽视。虽然设计结果能够工作,但并不一定可靠。

同步设计

同步设计不包含门控时钟和多选一时钟。时钟的个数甚至只用一个手指头就可以数清。

John McGibbon

Memec Design Services

如果设计是 100%同步的,提供 ASIC 转换服务的提供商有时会提供无向量转

换。这样将减少转换时间和成本。另外，由于测试的工作量和责任被转移了，因此 FPGA 设计者的压力也减小了。ASIC 提供商喜欢这种模式，因为相对来说，设计基本上不会产生什么问题，测试向量也能自动生成。对 FPGA 设计者来说，实现 100%的同步设计几乎是不可能的，但它却是一个值得追求的目标。

9.3.1　使用通用逻辑结构

做 FPGA 转换的 ASIC 提供商通常选用自己的单元库中参数化模块来代替 RAM 和其他的模块(比如加法器和计数器)。然而，每一个置换都会给设计添加一个新的设计块，同时每一个改动也会带来运行出错的风险。LogiBLOX 或者内核在被用于 FPGA 设计前，应该进行审核，其目的是降低向 ASIC 进行转换或者替换的难度。在转换的过程中要尽可能地不涉及设计网表。如果设计中只包含与非门，那么转换是毫不费劲的，因为并不存在模块替换问题。

在门阵列中，将会用寄存器来代替 RAM 和 ROM 模块，因而导致门数量的爆炸式增长。对一个 512×8 的 RAM，要占用 4096 个触发器。逻辑译码单元就添加这个数目(每添加一个地址线，解码逻辑复杂度将成倍地增加)。

特别烦人的一点是在 ASIC 转换过程中 RAM 的初始化(或者 ROM 的初始化)。FPGA 可以在加电的过程中将数据写入 RAM。但相应的过程在 ASIC 中却没有。所有的 RAM 单元都必须通过 RAM 数据总线来写入数据。

9.3.2　上电条件

FPGA 设计中的一部分工作是使用 GSR(全局置位/复位)和器件配置程序对所有 I/O 引脚进行上电初始化。对于 ASIC 设计，除非设计者专门进行规定，否则 ASIC 将不具有这些特性。ASIC 提供商倾向于不使用太多的全局网络(比如复位/预置网络)，因为这些特点都要由用户定制，而且这些特点都占用芯片布线通道。

9.3.3　内部总线

Xilinx 和其他的 FPGA 提供商的器件都可以使用器件内部的三态总线及总线保持功能以防止由于浮点运算导致缓冲器溢出带来的问题。一些 ASIC 提供商并没有生产这类器件的能力或者因为这项技术使得测试过于复杂，而不愿在器件中使用它。而 Exemplar Leonardo 公司的产品却具有这样特点：内部总线能自动地转换成不支持内部三态总线技术的 MUX。

9.3.4　引脚配置

通常 FPGA 引脚配置功能被应用于器件的外部逻辑设计上(如使用配置项的 DONE 配置引脚的上电复位逻辑)。专门的逻辑功能被嵌入 ASIC 器件中以提供对

于引脚配置仿真的支持。FPGA 的设计者必须给出要用的信号和相应的引脚行为描述。对于一般的 FPGA 信号和结构，ASIC 提供商都有其相关的设计经验，并也能为其用户提供设计上的帮助。

9.3.5 I/O 引脚缓冲器

输入信号的阈值必须要由设计人员来定义。输入阈值选项包括 TTL(阈值电压大约是供电电压 30%)、CMOS(阈值电压大约是供电电压的 50%)，或者是用户定义。

输出引脚的驱动也必须由 FPGA 设计人员来定义。设计中要使低阻(大电流)缓冲器阻值最小化以减少功耗和由该设计产生的 RFI 噪声。ASIC 在驱动能力方面可能比 FPGA 有更多的选择。通常选择采用能在本设计中正常运行的速度最低、能量损耗最小的引脚 I/O 缓冲器。

9.3.6 振荡器

振荡器一般是模拟电路，但有时候振荡器也可以用 FPGA 技术来实现。这种设计用到了具有低增益线性放大特性的倒相器。倒相器产生 180°的相位滞后，可以用 RC 振荡器(体积最大，但最便宜)，也可以用陶瓷振荡器(中等尺寸，但价位适中)或者晶体振荡器(体积最小，性能最好，但更昂贵)产生 180°的相移以产生振荡(整个环路产生 360°的相移，全部增益为 1)。一种典型的由门电路构成的振荡器如图 9-1 所示。

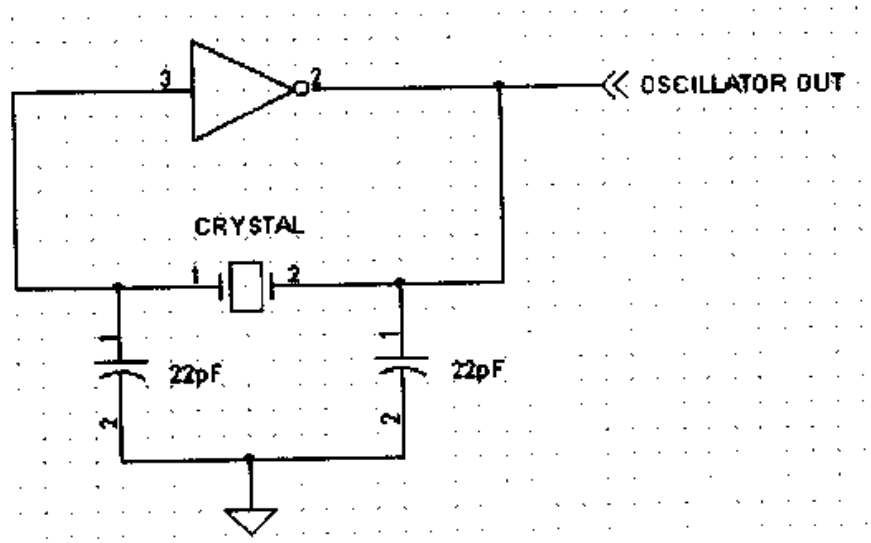

图 9-1 典型门振荡器电路

这类电路要由 ASIC 提供商提供与之相兼容的转换。振荡器可能采用门选通电路或者多选一电路(这和具有门时钟的通用运行模式有很大的不同)技术来进行设计,从而使得测试设备可以利用已知的频率和相位来驱动时钟的输出。可以把这类电路加入 ASIC 设计中,使之成为 ASIC 的一部分,也可以不把它作为 FPGA 的一部分。

不要使振荡器的使能引脚固定在高或低电平状态,在电路中可以增加一个电阻以使外部的电源能控制振荡器的开关,如图 9-2 所示。

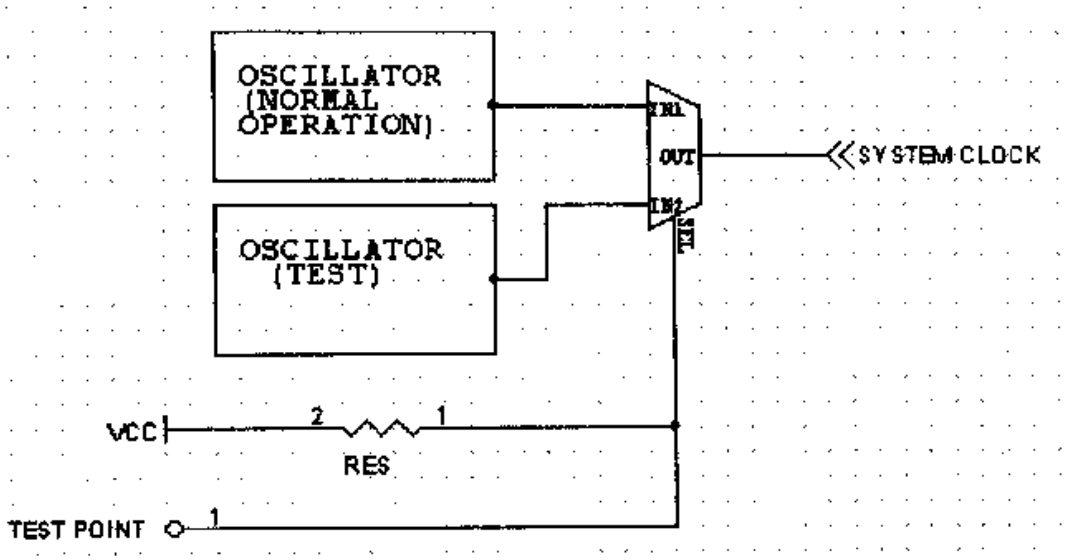

图 9-2　ASIC 振荡器禁用电路

为了获得最佳的性能,时钟电路应在物理空间上远离噪声源或者用防护环把它和其他的噪声源隔离开,并且绕线也应紧匝,这样可减少环路面积以尽可能降低 EMI。注意:振荡倒相器运行在线性模式下,并且输出应当近似为正弦波,目的也是尽可能降低 EMI。

9.4　延　迟　线

FPGA 设计人员有时使用一根延迟线来产生时间延迟信号,特别是在和外部 SRAM 或者 DRAM 组件进行接口的时候。注意,这类延迟是另一类模拟电路,延迟线可能是一串缓冲器。我们不推荐使用这种产生延迟的方法,因为这种方法依赖于典型缓冲器延迟,而这种延迟是不可控的,并且随着环境温度和过程/工艺的改变而改变。

在 ASIC 的转换中,延迟线缓冲器将被具有不同传输时延的缓冲器所替代(通常更短,因为 ASIC 缓冲器通常比 FPGA 缓冲器速度更快)或者完全被去掉。因为

从数字电路设计的观点来看，这些延迟线缓冲器是冗余逻辑。这些时延必须要写成文件形式并进行校验，以确保它们能在程序运行中被正确的实现。

可以选择使用外部电路来创建时延，如图 9-3 所示。这个电路可能是一个 RC 时延电路，它能够在 ASIC 转换过程中，把由输入驱动的改变和输出驱动负载的改变造成的对电路的影响减至最小。这种时延并不精确，而与引脚驱动的传输时延、缓冲器的传输时延、缓冲器的阈值电压、RC 元器件的容差，以及工作温度有关，夸张地说，甚至与月球重力场产生的以太流有关。

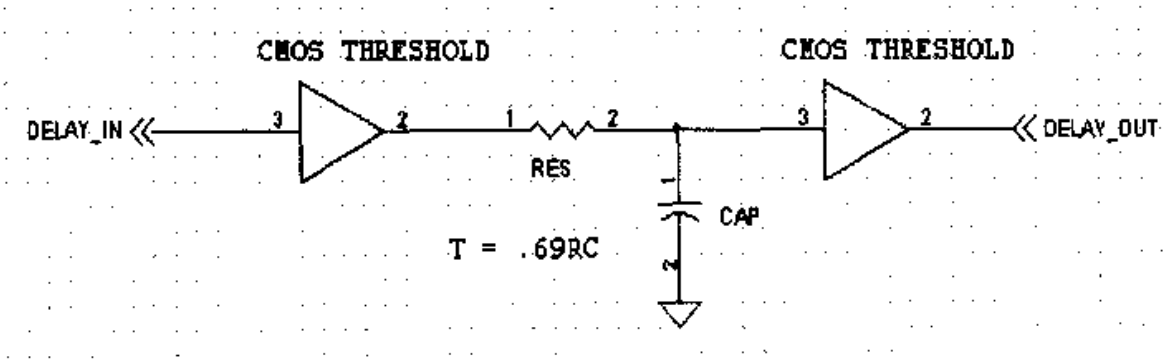

图 9-3 典型的外部缓冲 RC 延迟电路

一块十分复杂的电路板上，每一英寸的布线都会产生 175ps 的时延。这里谈到的时延应包含缓冲时延，焊点走线时延和电路板布线时延。如图 9-4 所示。当然时延是有上述许多假设的。读者在设计此类电路之前应学习 Johnson 和 Graham 合著的《高速数字电路设计》一书及《黑色魔法手册》(见本书参考文献)。

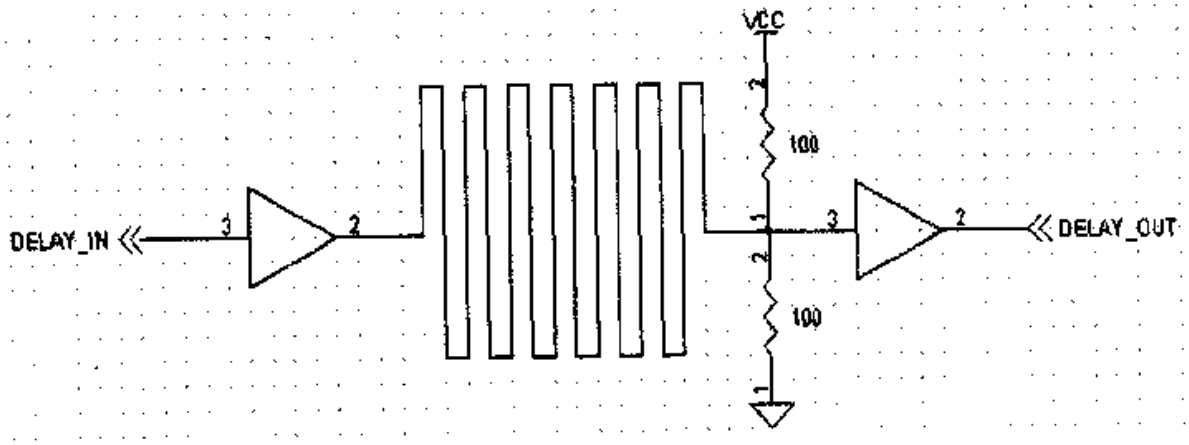

图 9-4 典型外部延迟线电路

假定使用 FR-4 的电路板材，20mil(密耳)[①] 的金属线宽，每段 0.6in 长，50mil 的间距，可以从 Murphy 定律得到所要的答案。

要得到更好的设计性能，就要考虑花更多钱并使用从 Dallas Semiductor 公司和其他半导体公司生产的数字时延电路芯片来做设计。

9.5 测试用语

我们并不打算涵盖有关测试的深层次的内容，但一些常用的术语还是应该知道的。

At-speed testing(在线测试)　以设计的实际运算速度进行测试。多数测试是在较低的时钟速度下进行的，这有利于测试设备性能的发挥。频率范围大约在 1～5MHz。

ATPG(自动测试向量生成)　这些测试向量包括串行向量(如果有的话，和 BST 扫描链同步)和并行向量(以和器件输入端并行的方式出现)。

Boundary scan(边界扫描)　一种测试方法。通过加入多路转换器件(MUX)及锁存器来支持数据的串行移位输入和串行移位输出。这可以实现加入测试向量及内部逻辑状态的输出。

BIST(内嵌式自我测试)　设计中加入相应的硬件设备用以实现自我测试。

Fault grading(故障等级)　对设计硬件测试效果的评估标准。它等于测试向量数目与故障范围之比。

Functional test(功能测试)　采用用户自定义输入并检测输出的方法对器件进行的测试。一般来说，这类测试无法做得非常全面。对于交流状态时的性能没有提供相应的参数测试手段。

$\mathbf{I_{DDQ}}$　内部所有节点处于静态时供电电源电流的测试。测试时，为防止产生振荡并抑制门电路进入线性状态，惟一的输入要断开。这是一种剔除不合格器件的快速的检测方法。

JTAG(联合测试执行组)　该执行组制定了 IEEE1149.1 边界扫描寄存器和测试访问端口(TAP)的技术标准。

Observability(可观察性)　测试设备访问内部节点的能力。所有的输出引脚都是可观察的。

Parametric testing(参数测试)　测试门电路的输入阈值和输出端的驱动能力。在校验 ASIC 的工艺过程时也进行类似的测试。

Partial scan(局部扫描)　仅对设计的特定选择部分进行扫描测试。

---

① 1mil = 2.54×10^{-5}m。

Test coverage(测试覆盖率)　对一套测试程序所达指标的一种描述。它等于被测到的全部故障数与可能出现的总的故障数目之比。

Stuck-at faults("固定"故障)　当节点本应从一种态变到另外一种态时,它却停留在"0"或"1"态,由此所产生的故障。

现重点对上述某些用语做一些解释。

边界扫描

因为基于 SRAM 的 FPGA 器件可以进行重复编程,因此 FPGA 提供商可以通过加载一个测试配置程序,进行完整的产品测试。对于定制器件(比如 ASIC 器件),在设计中应当考虑加入对测试需求的支持。通常采用的提供测试支持的方法是插入一个边界扫描测试(BST)电路,这个扫描测试电路在被测试器件的外围附近形成一条串行链。这条串行链可以包含其他的器件,它可以由 4 或 5 个信号构成(TDI——测试数据输入,TDO——测试数据输出,TCLK——测试时钟,TMS——测试模式选择,以及可选的测试复位 TRSTn 信号)。在 ASIC 内部,采用 MUX 器件将选中的信号连接至一个长移位寄存器上,从而实现信号从被测器件移入和移出的功能。

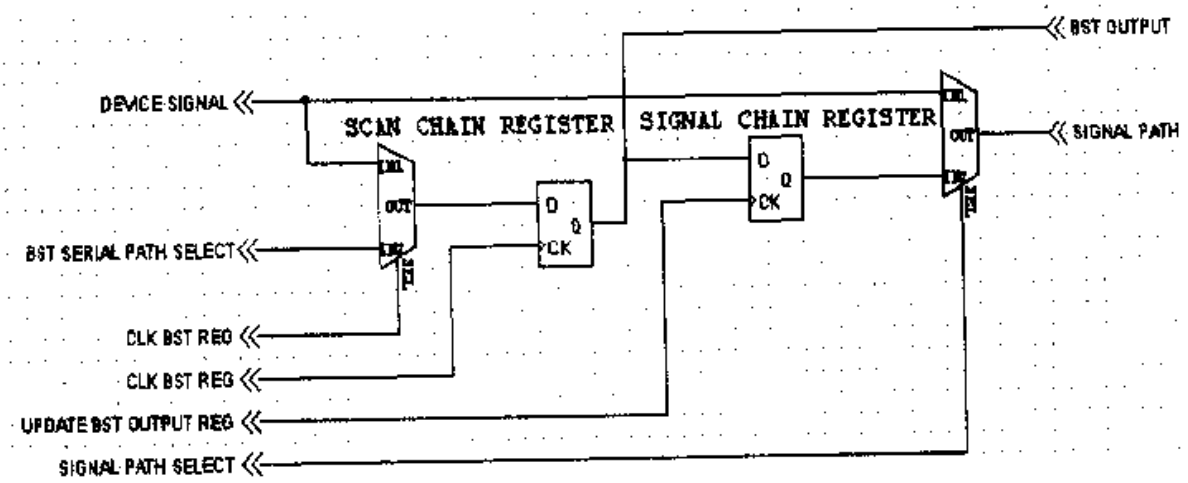

图 9-5　边界扫描电路

图 9-5 给出了 ASIC 中加入 BST 硬件的示意图。这些硬件的加入增加了 15% ~ 25%的 ASIC 设计量。由于 BST 电路的引入,将给每个 MUX 结构带来大约 1 ~ 2ns的信号延迟。BST 硬件的接入及扫描向量的产生过程都是自动的。值得注意的是器件信号总是要通过一个 MUX 的。这种结构使得串行链可以读取器件信号,或者为其他测试信号的通过提供通路,另外也可以使得测试信号进入信号链。

还有其他的测试方法。有关这些方法完整的讨论已经超出了本书的范围,但

我们至少可以把它们罗列下来并加以简单的介绍。测试可以分为三类：产品测试（设计得到验证并对过程中出现的疑问进行测试）、设计变更测试（用以确保设计变更的正确性，这些工作通常主要是通过设计者提供的功能测试矢量来完成的），静态时序测试（用以确保 ASIC 中不同门电路产生的延迟及时钟的畸变不会引起其他的问题）。

IDDQ 测试

这是一种针对产品的快速测试方法。如果我们发现器件的电流消耗远远大于我们的期望值，则说明该器件本身存在缺陷，很快就能检出。

功能测试

这类测试采用了由设计者提供的测试向量，该向量对典型的运算模式进行仿真，并查询可预见的输出。这类测试通常不够全面，因为设计者无法知道输入模式及逻辑序列的所有可能的组合。

9.6　POC 测试向量

ASIC 的提供商将需要 POC（Print-on-Change）测试向量。这是一张有关输入序列及测试预期输出的以 ASCII 码格式打印的程序列表。在 Verilog 测试工具中使用 $display 和 $monitor 指令不难提取出这些向量。

程序列表 9-1　简单的 POC 向量实例（或门）

	I I	O
	N N	U
	1 2	T
TIME		
0	00	0
50	01	0
53	01	1
100	00	1
103	00	0
150	10	0
153	10	1

从程序列表 9-1 中，我们可以看到门电路的延迟是 3ns（在 50 ~ 53ns 这段时间内输出发生变化）。

参 考 文 献

Bhasker J. 1997. *A Verilog HDL Primer*. Star Galaxy Press, Allentown, PA

Bhasker J. 1998. *Verilog HDL Synthesis, A Practical Primer*. Star Galaxy Publishing, Allentown, PA

Ciletti Michael D. 1999. *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL*. Prentice Hall, Upper Saddle River, NJ

Johnson Howard W, Graham Martin. 1992. *High-Speed Digital Design: A Handbook of Black Magic*. Prentice Hall, Upper Saddle River, NJ

Keating Michael, Bricaud Pierre. 1998. *Reuse Methodology Manual for System-on-a-Chip Designs*. Kluwer Academic Publishers, Norwell, MA

Kurup Pran, Abbasi Taher and Bedi Ricky. 1998. *It's the Methodology, Stupid*. Bytek Designs, Palo Alto, CA

Lee James M. 1997. *Verilog Quickstart*. Kluwer Academic Publishers, Norwell, MA

Malvino Albert Paul, Leach Donald P. 1975. *Digital Principles and Applications*. 2nd ed. McGraw-Hill Book Company. New York, NY

Maxfield, Clive "Max". 1998. *Designus Maximus Unleashed!*. Butterworth-Heinemann, Woburn, MA

Palnitkar Samir. 1996. *Verilog HDL, A Guide to Digital Design and synthesis*. Prentice Hall, Upper Saddle River, NJ

Sagdeo Vivek. 1998. *The Complete Verilog Book*. Kluwer Academic Publishers, Norwell, MA

Smith Douglas J. 1997. *HDL Chip Design*. Doone Publications, Madison AL

Smith Michael J S. 1997. *Application-Specific Intergrated Circuits*. Addison-Wesley, Reading, MA

Sternheim Eli, Singh Rajvir, Madhaven Rajeev and Trivedi Yatin. 1993. *Digital Design and Synthesis with Verilog HDL*. Automata, San Jose, CA

Zeidman Bob. 1999. *Verilog Designer's Library*. Prentice Hall, Upper Saddle River, NJ

光盘使用说明

本书配送的 CD-ROM 包括了书中所有的 Verilog 源代码，所有的工程文件和一些有用的实用程序，一个 Verilog 仿真器 Simucad's Silos Ⅲ 的评价版，一个 David Murray's Prism 编译器的示范版本，一个 Windows 环境下的 Bytech Service's Emath-Pro 评价版本。

关于 Silos Ⅲ

Silos Ⅲ是以 Verilog HDL 为基础的逻辑仿真环境，是目前惟一包括了完整的图形调试系统的逻辑仿真工具。它能处理大型的复杂的 ASIC 和 FPGA 设计，且速度快、使用方便简单。

Silos Ⅲ的安装通过 setup.exe 完成，该文件存在于 CD-ROM 的 Silos Ⅲ文件夹中。双击 setup 图标或在 Windows 下运行 setup.exe 文件即可。在 www.simucad.com 网站上提供了相关技术支持。

关于 Prism 编译器

Prism 编译器是一个界面友好的编译器，适用于 Windows NT/95/98。无论你编写的文件是哪种类型，该编译器都能将它们合并为一个高效的系统。它具有一个完整的文本功能编译器的全部特征，但它的功能还要强大的多。许多人把它作为一个文本窗口而不仅仅是一个编译器来使用。一个普通的文本报告可在瞬间被组合并具有进一步分析所需要的自己的宏。

Prism 的安装通过 setup.exe 完成，该文件存在于 CD-ROM 的 PrismEditor 文件夹中。双击 setup 图标或在 Windows 下运行 setup.exe 文件即可。注册程序、更新和技术支持都可以通过编译器实现。Prism 编译器试用版在 30 天内具有全部的功能，30 天以后一些功能会无法使用。本书提供一个书籍注册码：130998516，持有此码的购买者在购买 Prism 编译器时可享受优惠价。

关于 Emath-Pro

Emath-Pro 是一个用于电工学和电子工程学应用的先进的公式计算器。它涵盖了从基础电子学到高级电子学的基本公式，如传输线公式和磁学设计公式。大多数公式都能通过输入已知变量来求解未知变量。Emath-Pro 是工程师、技术员和学术的必备公式求解软件工具。

Emath-Pro 的安装通过 setup.exe 完成，该文件存在于 CD-ROM 的 Emath 文件夹中。这个示范版本的功能是完全的，使用者可在 www.bytechservice.com 网站上注册。

技术支持

Prentice Hall 并不提供磁软件的技术支持。如果此光盘存在什么问题，读者可以将所遇到的问题的描述通过电子邮件发送到 discexchange@prenhall.com，并要求替换该光盘。

获取本书内容和软件的勘误表、校正及其他信息请登录 www.bytechsercices.com/verilog/。

术 语 表

AHDL Altera Hardware Description Language, Altera 公司的硬件描述语言

Algorithm 算法

Antifuse 反熔丝

ASIC Application-Specific Integrated Circuit, 专用集成电路, 为完成一项特定功能而设计的集成电路, 即使这个功能是通用功能(如微处理器就是一个 ASIC)

ATPG Automatic Test Pattern Generator, 自动测试向量发生器

asynchronous 异步逻辑, 即不使用同一个参考时钟的逻辑电路。设计者所面临的 90% 的问题都与异步信号计时有关

autorouting 自动布线

behaviorial 行为模式, 是一种程序化的代码编写风格, 此风格下的逻辑描述与被综合的硬件没有直接联系。与结构化门电路和网线赋值语句相比, 这是一种更抽象的逻辑定义方式

bidirectional 双向端口, 既可作为输出端口又可作为输入端口。此端口只有一个输入端, 并与多个驱动相连, 但设计者必须保证在同一时间内只有一个输出驱动有效

binary 二进制, 只有两种状态, 即"0"或"1"

BIST Built-In Self Test, 内建自测试

bit 比特

bitstream 串行通信中 FPGA/CPLD 的格式化配置信息流

bitwise 位逻辑运算

blocking 阻塞性(赋值), 其后的语句必须在阻塞性赋值完成后执行。在数学结构中, 阻塞性赋值的顺序是很重要的

Boolean 布尔体系

Buskeeper 在总线三态时用以保持节点逻辑状态的低电流驱动电路

buffer 缓冲器 用来对信号进行隔离或提供功率增益以驱动低阻抗负载的信号驱动器

capacitance 电容量

case 多输入判决语句, case 语句是按照优先级顺序进行测试的, 与输入相匹配的第一个 case 语句将被执行。case 语句的结果是"真"或"假", 其输入值将与 0、1、X 或 Z 相比较

casex 将 Z 和 X 视为无关项(X)的 case 状态

casez 将 Z 视为无关项(X)的 case 状态

checksum 校验和

CLB Configurable Logic Block,可配置逻辑模块,Xilinx FPGA 的基本单元,包括一个 3～5 输入的 LUT

CMOS 互补型金属氧化物半导体(即同时使用 P 型和 N 型晶体管)

combinational 组合,即对输出直接和立即赋值的异步操作

concatenation 级联,在 Verilog 语言中,包含在{}中的项被链接在一起并被当做一个单一输入项

configuration 配置,即将用户设计的程序下载到 FPGA

constraints 约束条件,为性能优化而加入设计中的条件和要求,约束条件包括:信号路径的时序要求,器件引脚的分配,逻辑块的相对位置等

core 内核(知识产权元素,是预先设计的功能模块)

CPLD Complex Programmable Logic Devices,复杂可编程逻辑器件,与 FPGA 相比,CPLD 具有更复杂的逻辑单元和更大块的布线资源

CPU Central Processing Unit,中央处理器

CRC Cyclic Redundancy Checksum,循环冗余校验和,由一个数据流生成的伪随机数

DeMorgan's Theorems 德·摩根定律,是对或、与运算进行转换的两个布尔逻辑定律

$$\sim (A \mid B) = (\sim A \,\&\, \sim B)$$

$$\sim (A \,\&\, B) = (\sim A \mid \sim B)$$

DFF D-Type FlipFlop, D 型触发器

dissipation 耗散 在正常工作中附带产生的损耗。对于 FPGA 来说,是指在信号转换时所产生的功率损耗。这会造成 FPGA 器件温度升高,耗散的大小(加热效果)和信号负载和转换频率成正比

DLL Delay-Locked Loop,延时锁相环,即通过延迟时钟路径以使所以的时钟边沿大致同步,是控制时钟偏移的一种方法

DRAM Dynamic Random Access Memory,动态随机存取存储器

EAB Embedded Array Block,嵌入式阵列块(Altera 公司 CPLD 中的基本 RAM 模块)

edge-triggered 边沿触发,仅在参考时钟的上升沿和(或)下降沿有效的信号

EDIF Electronic Design Interchange Format,电子设计交换格式,是电子工业联合会(EIA)采纳的标准

EEPROM Electrically Erasable Programmable Read-Only Memory,电可擦除可

编程只读存储器

EMI Electro Magnetic Interference,电磁干扰,在信号改变的过程中,一些能量被发射到空中浪费掉了,这些能量若不加以处理,就可能对其他电路产生不良影响

EPROM Electrically Programmable Read-Only Memory,可擦除可编程只读存储器

fanout 扇出(输出系数)

feedback 反馈

FET Field Effect Transistor,场效应晶体管

FG Function Generator,函数发生器,一个3输入、4输入或5输入的查找表,是Xilinx的基本逻辑单元

FIFO First-In First-Out,先入先出(寄存器组)

flatten 将模块和库单元合并为一个单一网表的过程

flipflop 触发器,一个双稳(态)多谐振荡器,其输出是"真"或是"假",输出由输入和输入的历史状态(记忆状态)决定

floorplan 器件内部逻辑单元物理结构布置

footprint 器件的封装和引脚的物理性布置

FPGA Field Programmable Gate Array,现场可编程门阵列,同CPLD相比,FPGA具有更多分段的布线资源和更简单的逻辑单元。这带来了更大的灵活性,但同时造成了电路性能的不可预测性

FSM Finite State Machine,有限状态机

GAL Generic Array Logic,通用阵列逻辑

GIGO Garbage-In, Garbage-Out, 无用输入导致无用输出,即输入信号的质量决定输出信号的质量

glitch 瞬态干扰信号

GSR Global Set/Reset,全局置位/复位

GTL Gunning Transcevier Logic,射电收发逻辑电路

GTS Global TriState,全局三态

GUI Graphical User Interface,图形用户界面

hazard 由于输入信号的交迭或丢失而引起的瞬态干扰信号

HDL Hardware Description Language,硬件描述语言

hex 十六进制的缩写形式

hierarchy 分层

hold time 为保证触发器或锁存器输出结果的正确性,输入信号必须在时钟沿之后保持稳定的最小时间

hysteresis 滞后，和摩擦类似的一种效应，反馈被用于减缓输出对输入的反应是所产生的一种效应，常用于减少脉冲干扰

impedance 阻抗，对信号方向和强度变化的一种阻碍。阻抗大小为电阻和电抗之和

inout 双向模式端口

input 输入端口

instance 事件(一个信号、库或模块的出现)

instantiate 创建一个事件

integer 整数(不含分数和小数部分)，Verilog 语言定义一个整数至少是 32 位

IP Intellectual Property，知识产权

LAB Logic Array Block，逻辑阵列块(Altera 公司 CPLD 中的基本模块)

latch 锁存器，该电路通过设置反馈，使它可以“记忆”电路原有的输入情况，并维持相应的电路状态

latency 级联时间，即从处理输入到产生输出的一段时间。在同步系统中，这段时间可由完成操作所需的时钟数来计算

LE Logic Element，逻辑元件，Altera 公司的 LAB 就是由这些查找表结构组构成的

Lint 一种计算机语言的语法检查程序

LSB Least Significant Bit，最低有效位

LUT Look-Up Table，查找表

management 经理人

metastability 亚稳定性，当触发器的建立或保持时间被破坏后，输出将是不确定的，触发器的这种性质称为亚稳定性

MSB Most Significant Bit，最高有效位

MUX A multiplexer，多路转换器，该电路使用一个或一组控制条件来对其输出进行转换或选择

NAND Not-AND 与非门

net 网线

netlist 设计网表

newbie 新手

nonblocking 非阻塞性，这种赋值方法不会阻塞程序流。非阻塞性赋值可以同时进行且互不干扰，它们在一个程序块中的顺序并不重要

nsec Nanosecond 十亿分之一(10^{-9})秒

oscillator 振荡器，产生交替变化或脉动的输出的器件，常用于创建同步电

路的参考时钟。对于振荡器有两个基本要求：360°的反馈、总环路增益为1。有一个古老的谚语：如果你试图设计一个振荡器的话，得到的将是一个放大器，而如果你打算设计一个放大器时，得到的却往往是一个振荡器

output 输出端口

pad 焊点(本书指的是 FPGA 的引脚焊接点)

parameter 参数，这个值通常可以在编译过程中变化

PCB Printed Circuit Board，印刷电路板

pipeline 流水线，流水线结构以牺牲级联时间为代价来换取运行速度的提高

PIP Programmable Interconnect Point，可编程互连点，Altera 的信号连接方法

PLD Programmable Logic Device，可编程逻辑器件

PLL Phase-Locked Loop，锁相环路，一种用来同步参考频率的方法

portability 可移植性

POST Power-On Self Test，上电自检

primitive 原语，是设计中最基本的单元。Verilog 原语包括与、与非、或非、或、异或、同或。原语可以描述 FPGA/CPLD 的单元结构(引脚缓冲器、时钟驱动器、查找表等)

propagation 传播，当用电荷来描述信号时，电荷流动穿过电路所花费的时间称作传播时间

pull -down 下拉电阻，当线路没有其他驱动作用时，下拉电阻使该节点处于逻辑低电平

pull -up 上拉电阻，当线路没有其他驱动作用时，上拉电阻使该节点处于逻辑高电平

PWB Printed Wiring Board，印刷线路板

RAM Random Access Memory，随机存储器

reg 寄存器，一个数据存储器，可以是锁存器、触发器或是一个记忆单元或单元，**Verilog** 寄存器的默认状态是 **X**

RFI Radio Frequency Interference，射频干扰

route 路由，信号通过的物理路径

RTL Register Transfer Level，寄存器传输级，RTL 假设 FPGA 硬件和库单元定义了一组硬件结构，HDL 代码将与这些结构相映射。RTL 结构包括诸如触发器、锁存器、MUX 等与 FPGA 布线资源相连接的电路

schematic 电路示意图

SDF Standard Delay Format，标准延迟格式

sensitivity list 灵敏度表，又称为事件表或事件灵敏度表，是一个模块中所

使用的信号的目录。此列表驱动仿真器；仿真器能够判断信号的改变并确定信号是否在模块中被使用

setup time 建立时间，为保证触发器或锁存器输出结果的正确性，输入信号必须在时钟沿之前保持稳定的最小时间

skew 时偏，即一个信号在 FPGA 某个部分产生的时间和该信号到达 FPGA 另一部分的时间之间的差异

slack time 弛豫时间，正弛豫有益，负弛豫则有弊

SMT Surface Mount，表面贴装

SRAM Static Random Access Memory，静态随机存储器

structural 结构性描述，HDL 的一种代码风格。这种方式下，电路单元以类似电路原理图一样的形式相互连接在一起

stuck 一种逻辑错误形式，当一个信号应当随输入的变化而变化时，却被阻塞在一个特定的值上（比如固定在“1”值上）

synchronous 同步的

synthesis 综合，将 HDL 映射到可用的硬件的过程

ternary 三重的

threshold 阈值，信号被判断为“1”电平或“0”电平时所需满足的电压值。对于 TTL 电路，这个电压值近似为 1.4V，对于 CMOS 电路，则近似等于 1/2 的供电电压

tick 符号“'”在 Verilog 中用以标识编译指令（如'define），不要和用于定义数的符号‘相混淆（如 1'b0）

timescale 时标，在仿真时所使用的基本时间单元。在 Verilog 中默认的单位是纳秒（ns）

TLA Three Letter Acronym，三字母缩略词

toggle 状态切换

tri 多源驱动的 Verilog 网线

tristate 三态，输出驱动存在三种状态，0、1 或 Z（高阻）

μA 微安（10^{-6}安培）

UART Universal Asynchronous Receiver-Transmitter，通用异步收发器

vector 向量，多比特结点或寄存器变量。Verilog 只支持一维向量。此名词也是测试向量（一组用于测试的输入和输出值）的简称

vendor 提供商

VHDL Very high-speed integrated circuit Hardware Description Language，高速集成电路硬件描述语言 此语言根植于 Ada 编程语言，是 Verilog 语言的主要竞争对手

Verilog 一种硬件描述语言，最早是由 Phil Moorby 在 1983 - 1984 年为 Automated Integrated Design System 公司(后改为 Gateway Design Automation 公司)的仿真器而开发的专用语言。1990 年 Verilog 被 OVI 推广。1995 年，Verilog 语言成为 IEEE 标准，称为 IEEE Std 1364—1995。其完整的标准在 Verilog 硬件描述语言参考手册中有详细说明

wire 线网，可由单个信号源驱动的 Verilog 网线

X 未知值

XNOR Exclusive NOR，同或，是和异或逻辑相反的逻辑

XOR Exclusive OR，异或，仅当输入变量不同时，输出才为真

Z 高阻抗

资 料 索 引

有关《基于 Verilog 语言的实用 FPGA 设计》的更新和勘误表信息，请浏览以下网站：

www.bytechservices.com

如发现本书有什么错误或有什么意见的话，请发邮件到：

kcoffman@sos.net

互联网是用于研究的一个极好的工具。网络新闻组是一个可以自由发布信息、观点和交谈的极好的资源集散地。

网络新闻组

comp.lang.verilog
comp.lang.vhdl
comp.arch.fpga
comp.cad.synthesis

Verilog FAQ (Frequently Asked Questions with answers!)

http://www.siliconlogic.com/verilog/verilog/verilog-faq.html

FPGA 和 CPLD 制造商

www.actel.com
www.altera.com
www.latticesemi.com
www.lucent.com/micro/fpga
www.manuflex.com
www.xilinx.com

软件提供商

www.buluepc.com
www.cadence.com
www.exemplar.com
www.model.com

www. simucad. com

www. synopsys. com

www. synplicity. com

www. veritools-web. com

后　记

展望未来：天文数字般的逻辑门

新闻发布：Xilinx 公司，2005 年 7 月 7 日，San Jose，California。

Xilinx 公司宣布其 XZ-200 系列半导体器件的最新成员 XZ202XXL 问世。XZ202 支持工作在 10GHz 的 16 相位锁定和时延锁定时钟，而供电电压仅为 0.75V。集成的模拟特性包括 8 个象限功率控制（器件的电源可以在 8 个象限区域被接通或断开）、具有上端电压和下端电压可编程的 10 位 A/D 和 D/A 转换、采样/跟踪-保持、SVGA 监视器输出、Delta 调制转换、电压-频率转换、脉宽调制（PWM）、高速数据捕捉、开关电容供电（用于支持 RS－232 和其他高电压的 I/O 设备，以及 CCD/CMOS 成像器件）、运算放大器。以 8 个象限控制为基础，支持系统的自我和实时可重复配置模式。XZ202 具有对单端和差分 I/O 口阈值大小的可编程控制功能，并且具有多达 1500 根 I/O 引脚。在其核心芯片上配置有 64MB 的高速 DRAM（由 8 个 8MB 的存储模块组成）。32768 个 CLB 中的每一个都可被单独配置成单端口或双端口 RAM 来使用，每个 CLB 都具有同步 FIFO 模式。

每个器件在出厂时都被分配了一个 128 位的 IPv6 地址。配套的测试设备中包括一个集成的上网用调制解调器，以使得 Xilinx 公司的技术支持人员可以对网上用户的配置进行远程的读出和测试操作。在硅谷和印度的 Xilinx 公司的技术人员可以提供 24 小时的全球范围的技术支持。随同每个器件的销售，都可免费获得一个内嵌的示波器和逻辑分析仪软件的使用授权。另外，还包括一个 JTAG 接口仿真器和环境背景调试器；一套免费的 Verilog 设计系统（具有面向对象扩展功能）也将附带提供。

支付不多的授权使用费就可以得到种类繁多的 IP 核产品，其中有：

- 工作频率 10MHz～1GHz 的 4～64 位微处理器，具有 RISC 和 DSP 核。
- 对 2.4GHz，GSM，PCS 和其他射频信号及格式的直接数字变换。
- JPEG，MJPEG 和 MPEG 编码和解码。
- 有线接收或无线接收的电视信号和 AM/FM 无线信号调谐器。
- PGP 和其他加密/解密内核。
- 离散余弦变换（DCT），快速傅里叶变换算法（FFT），小波变换和其他变换。
- 游戏和教育类软件，包括 Flight Simulator，Riven Ⅳ 和 Quake 2005。

价格和上市时间：一次购买 10000 件，则每件 25 美元，2005 年 3 季度可出样

品，2006 年 1 季度正式产品上市。

Xilinx 可以为每个员工提供公平竞争的机会。另外，所有的 Xilinx 器件都预先考虑了 Y3K 兼容性问题。

作 者 介 绍

K.科夫曼从事过许多种不同的工作：草莓采摘工、洗碗工、猫粮工厂的工人、空军中士、摇滚乐队的贝司手、大学讲师、电子技师、模型浇铸机操作工、电气产品设计，还做过工程经理和小商品经营。对他来说，如果在电子工程设计领域毫无建树的话，他至少还可以回去干洗碗工。

1995年，Ken和Mark Bothum合作出版了一部小说：鳄鱼谷（*Alligator Alley*），并且根据这部小说编写了一个电影剧本。

Ken是IEEE 1364.1工作组语义部分，以及Verilog语言RTL逻辑综合规范方面项目负责人。他拥有北Cogswell学院（即现在的Henry Cogswell学院）电气工程学士学位。