第5章

Verilog 语法概述

本章导读

本章介绍学习 Verilog 语言的一些经验和 Verilog 语言可综合的基本语法，以及常见逻辑功能的代码风格与书写规范。

5.1 语法学习的经验之谈

FPGA 器件的设计输入有多种方式，如绘制原理图、编写代码或是调用 IP 核。早期的工程师对原理图的设计方式情有独钟，这种输入方式应付简单的逻辑电路还凑合，应该算得上简单实用，但随着逻辑规模的不断攀升，这种落后的设计方式已显得力不从心。取而代之的是代码输入的方式，今天的绝大多数设计都是采用代码来完成的。

FPGA 开发所使用的代码，通常称为硬件描述语言(Hardware Description Language)，目前最主流的是 VHDL 和 Verilog。VHDL 发展较早，语法严谨；Verilog 类似 C 语言，语法风格比较自由。IP 核调用通常也是基于代码设计输入的基础之上，现在很多 EDA 工具的供应商都在打 FPGA 的如意算盘，FPGA 的设计也在朝着软件化、平台化的方向发展。也许在不久的将来，越来越多的工程只需要设计者从一个类似苹果商店的 IP 核库中索取组件进行配置，最后像搭积木一样完成一个项目，或者整个设计都不需要见到一句代码。当然，未来什么情况都有可能发生，但是底层的代码逻辑编写方式无论如何还是有其生存空间的，毕竟一个个 IP 核组件都是从代码开始的，所以对于初入这个行业的新手而言，掌握基本代码设计的技能是必需的。

这里不过多谈论 VHDL 和 Verilog 语言孰优孰劣，总之这两种语言是当前业内绝大多数开发设计者所使用的语言，从二者对电路的描述和实现上看，有许多相通之处。无论是 VHDL 还是 Verilog，建议初学者先掌握其中一门，至于到底先下手哪一门，则需要读者根据自身的情况做考量。对于没有什么外部情况限制的朋友，若之前有一定的 C 语言基础，不妨先学 Verilog，这有助于加快对语法本身的理解。在将其中一门语言学精、用熟之后，最好也能够着手掌握另一门语言。虽然在单个项目中，很少需要“双语齐下”，但在实际工作中，还是很有可能需要去接触另一门语法所写的工程。网络上有很多很好的开源实例，若只会 Verilog，而参考实例却是 VHDL 的，那么就很尴尬了；忽然有一天 A 同事离职，老板把

他写了一半的 Verilog 工程扔给只会 VHDL 的你来维护，那可就被动难堪了……所以，对于 VHDL 和 Verilog 的取舍问题，建议先学精一门，也别忘了兼顾另一门，无论哪一种语言，至少需要具备看懂别人设计的基本能力。

HDL 虽然和软件语言有许多相似之处，但由于其实现对象是硬件电路，所以它们之间的设计思维存在较大差异。尤其是那些做过软件编程的朋友，很喜欢用软件的顺序思维来驾驭 HDL，岂不知 HDL 实现的硬件电路大都是并行处理的。也许就是这个大弯转不过来，所以很多朋友在研究 HDL 所实现的功能时常常百思不得其解。对于初学者，尤其是软件转行过来的初学者，笔者的建议是不要抛开实际电路而研究语法，在一段代码过后，多花些精力对比实际逻辑电路，必要时做一下仿真，最好能再找一些直观的外设在实验板上看看结果。长此以往，若能达到代码和电路都心中有数，那才证明是真真正正掌握 HDL 的精髓了。

HDL 的语法条目虽多，但并非所有的 HDL 语法都能够实现最终的硬件电路。由此进行划分，可实现为硬件电路的语法常称为可综合的语法；而不能够实现到硬件电路中，却常常可作为仿真验证的高层次语法则称为行为级语法。很多朋友在初学语法时，抱着一本语法书晕头转向地看，最后实战的时候却常常碰到这种语法不能用、那种语法不支持的报错信息，从而更加抱怨 HDL 不是好东西，学起来真困难。其实不然，可综合的语法是一个很小的子集，对于初学者，建议先重点掌握好这个子集，实际设计中或许靠着十来条基本语法就可以打天下了。怎么样？HDL 一下变简单了吧。这么说一点也不夸张，本书的重点就是要通过各种可实现到板级的例程让读者快速地掌握如何使用可综合的语法子集完成一个设计。5.2 节中会将常用的可综合语法子集逐一罗列并简单介绍。对于已入门的读者，也不是说掌握了可综合的语法子集就“万事大吉”了。

行为级语法也非一无是处，都说“存在即是合理”，行为级语法也大有用处。一个稍微复杂的设计，若是在板级调试前不经过几次三番的仿真测试，一次性成功的概率几乎为零。而仿真验证也有自己的一套高效便捷的语法，如果再像底层硬件电路一样搭仿真平台，恐怕就太浪费时间了。行为级语法最终的实现对象不是 FPGA 器件，而是手中的计算机，动辄上 G 甚至双核、四核的 CPU 可不愿做“老牛拉破车”的活，所以行为级语法帮助设计者在仿真过程中利用好手中的资源，能够快速、高效地完成设计的初期验证平台搭建。因此，掌握行为级语法，可以服务于设计的仿真验证阶段的工作。

对于 HDL 的学习，笔者根据自身的经验，提几点建议。

首先，手中需要准备一本比较完整的语法书籍。这类书市场上已经是满天飞了，内容相差无几，初学者最好能在开始 FPGA 的学习前花一些时间认真地看过一遍语法，尽可能地理解每条语法的基本功能和用法。当然，只需要认真看过、理解过，做到相关语法心中有数就行，这也不是为了应付考试，也没必要去“死记硬背”任何东西。语法的理论学习是必需的，能够为后面的实践打下坚实的基础。有些实在不好理解的语法，也不要强求，今后在实例中遇到类似语法的参考用法时再掌握也不迟。

其次，参考一些简单的例程，并且自己动手写代码实现相同或相近的电路功能。这个过程中，可能需要结合实际的 FPGA 开发工具和入门级学习套件。FPGA 的开发工具前面章节已经有所介绍，主要是掌握 Quartus Ⅱ（Altera 公司的器件使用）或 ISE（Xilinx 公司的器件使用）的使用，学会使用这些工具新建一个工程、编写代码、分配引脚、进行编译、下载配置文件到目标电路板中。入门级的学习套件，简单地说，就是一块板载 FPGA 器件的电路板。

这块电路板不需要有很多高级的外设，一些简单的常见外设即可（如蜂鸣器、流水灯、数码管、UART、IIC 等）。通过开发工具可以进行工程的建立和管理；而通过学习套件，就可以直观地验证工程是否实现了既定的功能。在实践的过程中，一定要注意自己的代码风格，当然，这在很大程度上取决于参考例程的代码风格。至于什么样的学习套件配套的参考例程是规范的，倒也没有定论，建议在选择口碑较好的学习套件的同时，推荐读者多去读读 FPGA 原厂 Altera（qts_qii5v1.pdf）或 Xilinx（xst.pdf）公司的官方文档，在它们的一些文档手册中有各种常见电路的实现代码风格和参考实例。在练习的过程中，也要学会使用开发工具生成的各种视图，尤其是 RTL 视图。RTL 视图是用户输入代码进行综合后的逻辑功能视图。这个视图很好地将用户的代码用逻辑门的方式诠释出来，初学者可以通过查看 RTL 视图的方式来看看自己编写的代码所能实现的逻辑电路，以加深对语法的理解；反之，也可以通过 RTL 视图来检验当前所写的代码是否实现了期望的功能。

总之，HDL 的学习，简单地归纳，就是需要初学者多看、多写、多思考、多对比。

本书主要实验和例程将以 Verilog 语言为主。本章后面的基础语法部分也不会进行太详细的讲解，只是蜻蜓点水般带过——简单给出基本的用法模板，但是也别担心，笔者会把重点放在后面的实例章节中，更深入地引领读者学以致用。当然，语法本身总是枯燥乏味的，故更建议读者在实例章节多回过头来细细品味语法。

5.2 可综合的语法子集

可综合的语法是指硬件能够实现的一些语法。这些语法能够被 EDA 工具所支持，能够通过编译最终生成用于烧录到 FPGA 器件中的配置数据流。无论是 Verilog 语言还是 VHDL，可综合的子集都很小。但是如何用好这些语法，什么样的代码风格更适合于硬件实现，是每一位初学者都需要下功夫好好掌握的。

下面是常用的 RTL 级的 Verilog 语法及其简单的用法描述。Verilog 和 C 语言在语法上确实有很多相似相通之处，学习语法时相互类比进行记忆也未尝不可。但是笔者担心一旦过多地混淆 C 和 Verilog 语言，会让初学者误入歧途，毕竟 Verilog 和 C 语言在本质上存在着很大的差异，尤其是它们的设计思想和实现载体存在着很大的差异，所以希望读者在语法的学习过程中，尽可能多地去了解和对比相关语法最终实现的硬件电路，从而尽快地从软件式的顺序思维中解脱出来，更好地理解硬件式的并行处理。

（1）模块声明类语法：module…endmodule。

在每个 Verilog 文件中都会出现该语法。它是一个固定的用法，所有的功能实现语法最终都应该包括在“…”中。module 的语法如下所示，module 后的 my_first_prj 为该 module 的命名，取名没有任何限制（默认数字、下画线和字母的组合均可），随后一个“()”内罗列出该模块所有的输入/输出端口信号名。

```
module my_first_prj(<端口信号列表> …);
    <逻辑代码> …
endmodule
```

(2) 端口声明：input,output,inout(inout 的用法比较特殊,需要注意)。

每个 module 都会有输入/输出的信号用于和外部器件或其他 module 进行连接。对于本地 module 而言,这些信号无非可以归为 3 类,即输入(input)信号、输出(output)信号和双向(inout)信号。通常,在 module 语法后紧接着就要声明该模块所有用于与外部接口的信号。从语法上来讲,这些信号名也都要在 module 名后的“()”内列出。

最常见的 3 种端口声明实例如下：

```
input clk;
input wire rst_n;
input [7:0] data_in;
```

第 1 个声明表示 1bit 的名称为 clk 的输入信号端口；第 2 个声明表示 wire 类型的 1bit 的名称为 rst_n 的输入信号；第 3 个声明则表示 8bit 的名称为 data_in 的输入信号。

(3) 参数定义：parameter。

parameter 用于声明一些常量,主要是便于模块的移植或升级时的修改。

通常,一个基本的 module 一定包括 module…endmodule 语法和任意两种端口声明(通常所设计的模块一定是有输入和输出的),而 parameter 则不一定,但是对于一个可读性强的代码来说也是不可少的。这样一个基本的 module 如下：

```
module <模块命名>(<端口命名 1>,<端口命名 2>,...);

    //输入端口声明
    input <端口命名 1>;
    input wire <端口命名 2>;
    input [<最高位>:<最低位>] <端口命名 3>;
    ...

    //输出端口声明
    output <端口命名 4>;
    output [<最高位>:<最低位>] <端口命名 5>;
    output reg [<最高位>:<最低位>] <端口命名 6>;
    ...

    //双向(输入/输出)端口声明
    inout <端口命名 7>;
    inout [<最高位>:<最低位>] <端口命名 8>;
    ...

    //参数定义
    parameter <参数命名 1> = <默认值 1>;
    parameter [<最高位>:<最低位>] <参数命名 2> = <默认值 2>;
    ...

    //具体功能逻辑代码
    ...

endmodule
```

注：“//”后的内容为注释。

(4) 信号类型：wire、reg 等。

在如图 5.1 所示的简单电路中，分别定义两个寄存器(reg)锁存当前的输入 din。每个时钟 clk 上升沿到来时，reg 都会锁存到最新的输入数据，而 wire 就是这两个 reg 之间直接的连线。

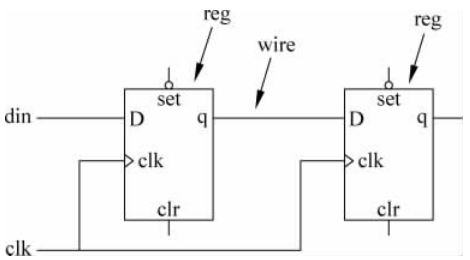

图 5.1 reg 和 wire 示例图

作为 input 或 inout 的信号端口只能是 wire 型，而 output 则可以是 wire 或 reg 类型。需要特别说明的是，虽然在代码中可以定义信号为 wire 或 reg 类型，但是实际的电路实现是否和预先的一致还要看综合工具的表现。例如，reg 定义的信号通常会被综合为一个寄存器(register)，但这有一个前提，就是这个 reg 信号必须是在某个由特定信号边沿敏感触发的 always 语句中被赋值。

wire 和 reg 的一些常见用法示例如下：

```
//定义一个 wire 信号
wire <wire 变量名>;

//给一个定义的 wire 信号直接连接赋值
//该定义等同于分别定义一个 wire 信号和使用 assign 语句进行赋值
wire <wire 变量名> = <常量或变量赋值>;

//定义一个多 bit 的 wire 信号
wire [<最高位>:<最低位>] <wire 变量名>;

//定义一个 reg 信号
reg <reg 变量名>;

//定义一个赋初值的 reg 信号
reg <reg 变量名> = <初始值>;

//定义一个多 bit 的 reg 信号
reg [<最高位>:<最低位>] <reg 变量名>;

//定义一个赋初值的多 bit 的 reg 信号
reg [<最高位>:<最低位>] <reg 变量名> = <初始值>;

//定义一个二维的多 bit 的 reg 信号
reg [<最高位>:<最低位>] <reg 变量名> [<最高位>:<最低位>];
```

(5) 多语句定义：begin…end。

通俗地说，begin…end 就是 C 语言里的“{ }”，用于单个语法的多个语句定义，其使用示例如下：

```
//含有命名的 begin 语句
begin : <块名>
```

```
    //可选声明部分
    //具体逻辑
end

//基本的 begin 语句
begin
    //可选声明部分
    //具体逻辑
end
```

(6) 比较判断：if…else、case…default…endcase。

判断语法 if…else 及 case 语句是最常用的功能语法，其基本的使用示例如下：

```
//if 判断语句
if(<判断条件>)
begin
    //具体逻辑
end

//if…else 判断语句
if(<判断条件>)
begin
    //具体逻辑 1
end
else
begin
    //具体逻辑 2
end

//if…else if…else 判断语句
if(<判断条件 1>)
begin
    //具体逻辑 1
end
else if(<判断条件 2>)
begin
    //具体逻辑 2
end
else
begin
    //具体逻辑 3
end

//case 语句
case(<判断变量>)
    <取值 1>: <具体逻辑 1>
    <取值 2>: <具体逻辑 2>
    <取值 3>: <具体逻辑 3>
    default: <具体逻辑 4>
endcase
```

(7) 循环语句：for。

for 循环语句用得也比较少，但也会在一些特定的设计中使用它。其示例如下：

```
//for 语句
for(<变量名> = <初值>; <判断表达式>; <变量名> = <新值>)
begin
    //具体逻辑
end
```

(8) 任务定义：task…endtask。

task 更像是 C 语言中的子函数，task 中可以有 input、output 和 inout 端口作为出入口参数，它可以用于实现一个时序控制。task 没有返回值，因此不可以用在表达式中。其基本用法如下：

```
task < task 命名>;
    //可选声明部分，如本地变量声明
    begin
        //具体逻辑
    end
endtask
```

(9) 连续赋值：assign 和问号表达式(?：)。

assign 用于直接互连不同的信号或直接给 wire 变量赋值。其基本用法如下：

```
assign < wire 变量名> = <变量或常量>;
```

“?:”表达式就是简单的 if…else 语句，但更多地用在组合逻辑中。其基本用法如下：

```
(判断条件) ? (判断条件为真时的逻辑处理) : (判断条件为假时的逻辑处理)
```

(10) always 模块：敏感表可以为电平、沿信号 posedge/negedge；通常和@连用。

always 有多种用法，在组合逻辑中，其用法如下：

```
always@( * )
begin
    //具体逻辑
end
```

always 后若有沿信号(上升沿 posedge，下降沿 negedge)声明，则多为时序逻辑，其基本用法如下：

```
//单个沿触发的时序逻辑
always@(<沿变化>)
begin
    //具体逻辑
end
```

```
//多个沿触发的时序逻辑
always@(<沿变化 1> or <沿变化 2>)
begin
    //具体逻辑
end
```

(11) 运算操作符：各种逻辑操作符、移位操作符、算术操作符大多是可综合的。

Verilog 中绝大多数运算操作符都是可综合的，其列表如下：

```
+          //加
-          //减
!          //逻辑非
~          //取反
&          //与
~&         //与非
|          //或
~|         //或非
^          //异或
^~         //同或
~^         //同或
*          //乘，是否可综合看综合工具
/          //除，是否可综合看综合工具
%          //取模
<<         //逻辑左移
>>         //逻辑右移
<          //小于
<=         //小于或等于
>          //大于
>=         //大于或等于
==         //逻辑相等
!          //逻辑不等于
&&         //逻辑与
||         //逻辑或
```

(12) 赋值符号：=和<=。

阻塞和非阻塞赋值，在具体设计中是很有讲究的，应该在具体实例中掌握它们的不同用法。

可综合的语法是 Verilog 语言中可用语法里很小的一个子集，硬件设计的精髓就是力求以最简单的语句描述最复杂的硬件，这也正是硬件描述语言的本质。对于 RTL 级设计来说，掌握好上面这些基本语法是很重要的。

5.3 代码风格与书写规范

不同的人可能对代码风格和代码书写规范这两个概念有不同的理解，很多人也会认为代码风格和代码书写规范说的是一码事。不管怎样，笔者在此为了说明和代码书写相关的

两个很重要的方面，进行如下区分界定：

- 代码书写规范，特指代码书写的基本格式，如不同语法之间的空格、换行、缩进以及大小写、命名等规则。强调代码书写规范，是为了更好地管理代码，便于阅读，以提高后续的代码调试、审查以及升级的效率。
- 代码风格，则是指一些常见的逻辑电路用代码实现的书写方式，它更多地是强调代码的设计。要想做好一个 FPGA 设计，好的代码风格能够起到事半功倍的效果。

下面将就这两个方面做一些深入的探讨，也许没有绝对意义上最优的代码书写规范和代码风格，但笔者会尽力结合自己多年的工程实践经验，给出一些具有较高参考价值的知识要点。

1）代码书写规范

虽然没有“国际标准”级别的 Verilog 或 VHDL 代码书写规范可供参考，但是相信每一个稍微规范点的做 FPGA 设计的公司都会为自己的团队制定一套供参考的代码书写规范。毕竟一个团队中，大家的代码书写格式达到基本一致的情况下，相互查阅、整合或移植起来才会“游刃有余”。因此，希望初学者从一开始就养成好的习惯，尽量遵从比较规范的书写方式。尽管不同的公司为自己的团队制定的 Verilog 或 VHDL 代码书写规范可能略有差异，但是真正好的书写规范应该都是大同小异的。这里也不刻意区分 Verilog 和 VHDL 书写规范上的不同，只是谈论一些基本的可供遵循的规范。

2）标识符

标识符包括语法保留的关键词、模块名称、端口名称、信号名称、各种变量或常量名称等。语法保留的关键词不可以作为后面几种名称使用。Verilog 和 VHDL 的主要关键字如下：

（1）Verilog 关键词。

```
always endmodule medium reg tranif0 and end primitive module release tranif1
assign endspecify nand repeat tri attribute endtable negedge rnmos tri0 begin
endtask nmos rpmos tri1 buf event nor rtran triand bufif0 for not rtranif0
trior bufif1 force notif0 rtranif1 trireg case forever notif1 scalared unsigned
casex fork or signed vectored casez function output small wait cmos highz0
parameter specify wand deassign highz1 pmos specparam weak0 default if posedge
strength weak1 defparam ifnone primitive strong0 while disable initial pull0
strong1 wire edge inout pull1 supply0 wor else input pulldown supply1 xnor
end integer pullup table xor endattribute join remos task endcase large real
time endfunction macromodule realtime tran
```

（2）VHDL 关键词。

```
abs downto library postponed subtype access else linkage procedure then after
elsif literal process to alias end loop pure transport all entity map range type
and exit mod record unaffected architecture file nand register units array for
new reject until assert function next rem use attribute generate nor report
variable begin generic not return wait block group null rol when body guarded
of ror while buffer if on select with bus impure open severity xnor case in
or shared xor component inertial others signal configuration inout out sla
constant is package sra disconnect label port srl
```

除了以上这些保留的关键词不可以作为用户自定义的其他名称，还必须遵循以下一些用户自定义的命名规则：

- 命名中只能够包含字母、数字和下画线“_”(Verilog 的命名还可以包含符号“$”)。
- 命名的第一个字符必须是字母(Verilog 的命名首字符可以是下画线“_”，但一般不推荐这么命名)。
- 在一个模块中的命名必须是唯一的。
- VHDL 的命名中不允许连续出现多个下画线“_”，也不允许下画线“_”作为命名的最后一个字符。

关于模块名称、端口名称、信号名称、各种变量或常量名称等的命名，有很多推荐的规则可供参考：

- 尽可能使用能表达名称具体含义的英文单词命名，单词名称过长时可以采用易于识别的缩写形式替代，多个单词之间可以用下画线“_”进行分割。
- 对于出现频率较高的相同含义的单词，建议统一作为前缀或后缀使用。
- 对于低电平有效的信号，通常加后缀“_n”表示。
- 在同一个设计中，尽可能统一大小写的书写规范。很多规范里对命名的大小写书写格式有要求，但是笔者这里不做详细规定，可以根据自己的需要设定。

3) 格式

这里的格式主要是指每个代码功能块之间、关键词、名称或操作符之间的间距(行间距、字符间距)规范。得体的代码格式不仅看起来美观大方，而且便于阅读和调试。关于格式，可能不同的公司也都有相关的规范要求，笔者在此建议尽量遵循以下的一些原则：

- 每个功能块(如 Verilog 的 always 逻辑、VHDL 的 process 逻辑)之间尽量用一行或数行空格进行隔离。
- 一个语法语句一行，不要在同一行写多个语法语句。
- 单行代码不宜过长，所有代码行长度尽量控制在一个适当的便于查看的范围。
- 同层次的语法尽量对齐，使用 Tab 键(通常一个 Tab 对应 4 个字符宽度)进行缩进。
- 行尾不要有多余的空格。
- 关键词、各类名称或变量、操作符相互间都尽量保留一个空格以作隔离。

4) 注释

Verilog 的注释有“/* */”和“//”两种方式。“/*”右侧和“*/”左侧之间的部分为注释内容，此注释可以用在行前、行间、行末或多行中；“//”后面的内容为注释，该注释只可用在行末(当然，它也可以顶格，那么意味着整行都是注释)。

VHDL 的注释只有“--”一种。类似 Verilog 的“//”。“--”后面的内容为注释，该注释只可用在行末。

注释的位置和写法通常也有讲究，归纳出如下几个要点：

- 每个独立的功能模块都要有简单的功能描述，对输入/输出信号功能进行描述。
- 无论是习惯在代码末注释还是代码上面注释，在同一个模块或工程中尽量保持一致。
- 注释内容简明扼要，不要过于冗长或写废话(例如：add=add+1;//add 自增)。

5）代码风格

代码风格主要是指工程师用于实现具体逻辑电路的代码书写方式。换句话说，通常对于一样的逻辑电路，可以有多种不同的代码书写方式来实现，不同的工程师一般也会根据自己的喜好和习惯写出不同的代码，这就是所谓的代码风格。

对于一些复杂的 FPGA 开发，工程师的代码风格将会在很大程度上影响器件的时序性能、逻辑资源的利用率以及系统的可靠性。有人可能会说，今天的 EDA 综合工具已经做得非常强大了，能够在很大程度上保证 HDL 代码所实现逻辑电路的速度和面积的最优化。但是要提醒读者注意的是，人工智能永远无法完全识破人类的意图，当然，综合工具通常也无法知晓设计者真正的意图。要想让综合工具明白设计者的用心良苦，也只有一个办法，便是要求设计者写出的 HDL 代码尽可能最优化。那么，就又回到了老议题上来——设计者的代码风格。而到底如何书写 HDL 代码才算是最优化，什么样的代码才称得上是好的代码风格呢？对于琳琅满目的 FPGA 厂商和 FPGA 器件，既有都拍手叫好的设计原则和代码风格，也有需要根据具体器件和具体应用随机应变的漂亮的代码风格。一些基本的设计原则是所有器件都应该遵循的。当然，设计者若是能够对所使用器件的底层资源情况了如指掌，并在编写代码过程中结合器件结构，这样才有可能设计出最优化的代码风格。

这里将和读者一起探讨在绝大多数 FPGA 设计中必定会而且可能是非常频繁涉及的逻辑电路的设计原则、思想或代码书写方式。

6）寄存器电路的设计方式

5.2 节中已经介绍了寄存器的基本原型，在现代逻辑设计中，时序逻辑设计是核心，而寄存器又是时序逻辑的基础。因此，掌握时序逻辑的几种常见代码书写方式就是基础中的基础。下面以图文（代码）并茂地方式来讲解这些基本寄存器模型的代码书写。

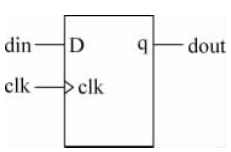

图 5.2　基本寄存器

（1）简单的寄存器输入/输出模型如图 5.2 所示。在每个时钟信号 clk 的有效沿（通常是上升沿），输入端数据 din 将被锁存到输出端 dout。

基本的代码书写方式如下：

```
//Verilog 例程
module dff(clk,din,dout);
input clk;
input din;
output dout;
reg dout;

always @ (posedge clk) begin
    dout <= din;
end

endmodule
```

(2) 带异步复位的寄存器输入/输出模型如图 5.3 所示。在每个时钟信号 clk 的有效沿(通常是上升沿),输入端数据 din 将被锁存到输出端 dout;而在异步复位信号 clr 的下降沿(低电平有效复位),将强制给输出数据 dout 赋值为 0(不论此时的输入数据 din 取何值),此输出状态将一直保持到 clr 拉高后的下一个 clk 有效触发沿。

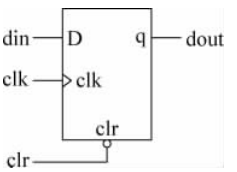

图 5.3 异步复位的寄存器

基本的代码书写方式如下:

```
//Verilog 例程
module dff(clk,rst_n,din,dout);
input clk;
input rst_n;
input din;
output dout;
reg dout;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) dout <= 1'b0;
    else dout <= din;
end

endmodule
```

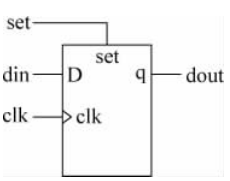

图 5.4 异步置位的寄存器

(3) 带异步置位的寄存器输入/输出模型如图 5.4 所示。在每个时钟信号 clk 的有效沿(通常是上升沿),输入端数据 din 将被锁存到输出端 dout;而在异步置位信号 set 的上升沿(高电平有效置位),将强制给输出数据 dout 赋值为 1(不论此时的输入数据 din 取何值),此输出状态将一直保持到 set 拉低后的下一个 clk 有效触发沿。

基本的代码书写方式如下:

```
//Verilog 例程
module dff(clk,set,din,dout);
input clk;
input din;
input set;
output dout;
reg dout;

always @ (posedge clk or posedge set) begin
    if(set) dout <= 1'b1;
    else dout <= din;
end

endmodule
```

(4) 既带异步复位又带异步置位的寄存器如图 5.5 所示。既带异步复位又带异步置位的寄存器其实是个很矛盾的模型，下面简单地分析一下：如果 set 和 clr 都处于无效状态(set＝0，clr＝1)，那么寄存器正常工作；如果 set 有效(set＝1)且 clr 无效(clr＝1)，那么 dout＝1 没有异议；同理，如果 set 无效(set＝0)且 clr 有效(clr＝0)，那么 dout＝0 也没有异议；但是如果 set 和 clr 同时有效(set＝1，clr＝0)，则输出 dout 到底是 1 还是 0？

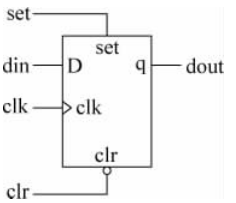

图 5.5 异步复位和异步置位的寄存器

其实这个问题也不难，设置一个优先级就可以了。当然，图 5.5 的理想寄存器模型通常只是作为电路的一部分来实现。如果期望这种既带异步复位又带异步置位的寄存器在复位和置位同时出现时，异步复位的优先级高一些，那么代码书写方式如下：

```
//Verilog 例程
module dff(clk,rst_n,set,din,dout);
input clk;
input din;
input rst_n;
input set;
output dout;
reg dout;

always @ (posedge clk or negedge rst_n posedge set) begin
    if(!rst_n) dout <= 1'b0;
    else if(set) dout <= 1'b1;
    else dout <= din;
end

endmodule
```

这样的代码，综合出来的寄存器视图则如图 5.6 所示。

(5) 图 5.7 所示是一种很常见的带同步使能功能的寄存器。在每个时钟 clk 的有效沿(通常是上升沿)，判断使能信号 ena 是否有效(此处取高电平为有效)，在 ena 信号有效的情况下，din 的值才会输出到 dout 信号上。

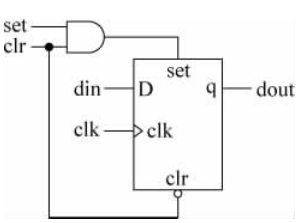

图 5.6 异步复位和置位的寄存器(复位优先级高)

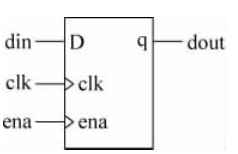

图 5.7 带同步使能的寄存器

基本的代码书写方式如下：

```
//Verilog 例程
```

```
module dff(clk,ena,din,dout);
input clk;
input din;
input ena;
output dout;
reg dout;

always @ (posedge clk) begin
    if(ena) dout <= din;
end

endmodule
```

7）同步以及时钟的设计原则

有了前面的铺垫，读者应该明白了寄存器的代码编写。接下来要更进一步从深层次来探讨基于寄存器的同步以及时钟的设计原则。

虽然在 3.2 节已经对组合逻辑和时序逻辑的基本概念做过描述，但在这里还是要再简单介绍一下组合逻辑和时序逻辑的历史渊源，以便读者更加了解为什么时序逻辑要明显优于组合逻辑的设计。早期的可编程逻辑设计，限于当时的工艺水平，无论是逻辑资源还是布线资源都比较匮乏，所以工程师们多是用可编程器件做一些简单的逻辑粘合。所谓的逻辑粘合，无非是由一些与、或、非等逻辑门电路简单拼凑的组合逻辑，没有时序逻辑，因此不需要引入时钟。而今天的 FPGA 器件的各种资源都非常丰富，已经很少有人只是用其来实现简单的组合逻辑功能，而更多地使用时序逻辑来实现各种复杂的功能，一旦大量地使用时序逻辑，时钟设计的各种攻略也就被不断地提上台面。时钟好比时序逻辑的“心脏”，它的好坏直接关系到整个系统的成败。那么，时钟设计到底有什么讲究？哪些基本原则是必须遵循的呢？在搞清楚这个问题之前，要先全面地了解时钟以及整个时序电路的工作原理。

在一个时序逻辑中，时钟信号掌控着所有输入和输出信号的进出。在每个时钟有效沿（通常是上升沿），寄存器的输入数据将会被采样并传送到输出端，此后输出信号可能会在长途跋涉般的“旅途”中经过各种组合逻辑电路并会随着信号的传播延时而处于各种“摇摆晃荡”之中，直到所有相关的信号都到达下一级寄存器的输入端。这个输入端的信号将会一直保持到下一个时钟有效沿来临。每一级寄存器都在不断重复着这样的数据流采集和传输。仅用这些枯燥的文字来描述时序逻辑和时钟之间的工作机理未免有些乏味，不妨举个轮船通行三峡大坝的例子做类比。

如图 5.8 所示，三峡大坝有 5 级船闸，船由上游驶往下游时，船位于上游。先关闭上游闸门和阀门；关闭第 1 级下游闸门和阀门，打开上游阀门，水由上游流进闸室，闸室水面与上游相平时，打开上游闸门，船由上游驶进闸室；关闭上游闸门和阀门，打开第 1 级下游阀门，当闸室水面降到与下游水面相平时，打开下游闸门，船驶出第 1 级闸室。如此操作 4 次，通过后面的 4 级船闸，开往下游。船闸的工作原理实际上是靠两个阀门开关，人为地先后造成两个连通器，使船闸内的水面先后与上、下游水面相平。

对于单个数据的传输，就与这里轮船通过多级闸门的例子非常类似。轮船就是要传输的数据，闸门的开关就好比时钟的有效边沿变化，水位的升降过程也好比相关数据在两个寄

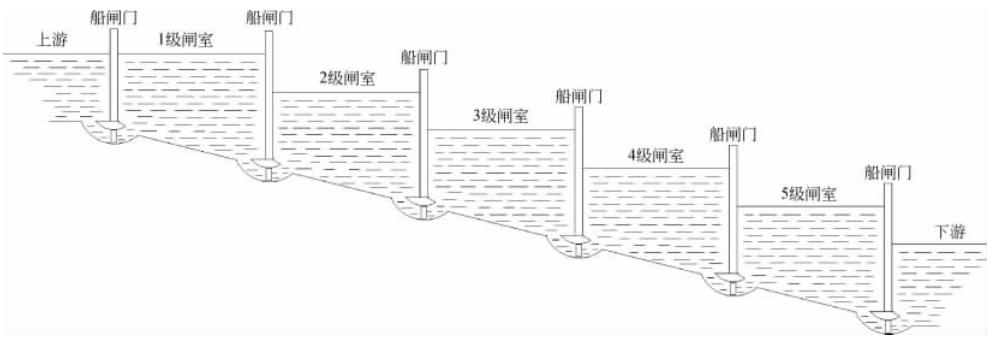

图 5.8 三峡大坝 5 级闸门示意图

存器间经过各种组合逻辑的传输过程。当轮船还处于上一级闸门准备进入下一级闸门时，要么当前闸门的水位要降低到下一级闸门的水平，要么下一级闸门的水位要升到上一级闸门的水平，只要不满足这个条件，最终都有可能造成轮船的颠簸甚至翻船。这多少也有点像寄存器锁存数据需要保证的建立时间和保持时间要求。关于建立时间和保持时间，有如下定义：

- 在时钟的有效沿之前，必须确保输入寄存器的数据在建立时间内是稳定的。
- 在时钟的有效沿之后，必须确保寄存器的输出数据至少在保持时间内是稳定的。

在理解了时钟和时序逻辑的工作机理后，也就能够理解为什么时钟信号对于时序逻辑而言是如此重要。关于时钟的设计要点，主要有以下 4 个方面：

(1) 避免使用门控时钟或系统内部逻辑产生的时钟，多用使能时钟去替代。

门控时钟或系统内部逻辑产生的时钟很容易导致功能或时序出现问题，尤其是内部逻辑(组合逻辑)产生的时钟容易出现毛刺，影响设计的功能实现；组合逻辑固有的延时也容易导致时序问题。

(2) 对于需要分频或倍频的时钟，用器件内部的专用时钟管理(如 PLL 或 DLL)单元去生成。

用 FPGA 内部的逻辑去做分频倒不是难事，倍频恐怕就不行了。但是无论是分频还是倍频，通常情况下都不建议用内部逻辑去实现，而应该采用器件内部的专用时钟管理单元(如 PLL 或 DLL)来产生，这类专用时钟管理单元的使用并不复杂，在 EDA 工具中打开配置页面进行简单的参数设置，然后在代码中对接口进行例化就可以很方便地使用引出的相应分频或倍频时钟了。

(3) 尽量对输入的异步信号用时钟进行锁存。

异步信号是指两个处于不同时钟频率或相位控制下的信号。这样的信号在相互接口时如果没有可靠的同步机制，则存在很大的隐患，甚至极有可能导致数据的误采集。笔者在工程实践中常常遇到这类异步信号误触发或误采集的问题，因此也需要引起初学者足够的重视。在笔者的《深入浅出玩转 FPGA》一书的笔记 6 中，列举了一些改进的复位设计方法，这就是非常典型的异步信号的同步机制。

(4) 避免使用异步信号进行复位或置位控制。

这一点和上文所强调的是同一类问题，异步信号不建议直接作为内部的复位或置位控制信号，最好能够用本地时钟锁存多拍后做同步处理，然后再使用。

上述几点对于初学者可能很难理解和体会，没有关系，当有了实践经历以后回头再品味一下或许就有“味道”多了。由于这几点多少也算是比较高级的技巧了，所以无法一一扩展开来深入剖析，更多相关的知识点可以参考笔者的《深入浅出玩转 FPGA》一书，那里有更多、更详细的介绍和说明。

8）双向引脚的控制代码

对于单向的引脚，输入信号或者输出信号，它们的控制比较简单，输入信号可以直接用在各类等式的右边用于作为赋值的一个因子，而输出信号则通常在等式的左边被赋值。那么，既可以作为输入信号又可以作为输出信号的双向信号又是如何进行控制的呢？如果直接地和单向控制一样既做输入又做输出，势必会使信号的赋值发生紊乱。列举一个简单的冲突，就是当输入为 0 而输出为 1 时，到底这个信号是什么值？而如何控制才能够避免这类不期望的赋值情况发生？可以先看看表 5.1 所列出的 I/O 驱动真值表。

表 5.1 I/O 驱动真值表

驱 动 源	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

从这个表里可以发现，当高阻态 Z 和 0 或 1 值同时出现时，总能保持 0 或 1 的原状态不变。在设计双向引脚的逻辑时要利用这个特性：引脚作为输入时，让输出值取 Z 状态，那么读取的输入值就完全取决于实际的输入引脚状态，而与输出值无关；引脚作为输出时，则只要保证与器件引脚连接的信号也是处于类似的 Z 状态，便可以正常输出信号值。当然，外部的状态是用对应芯片或外设的时序来保证的，在 FPGA 器件内部不直接可控，但我们还是可以把握好 FPGA 内部的输入、输出状态，保证不出现冲突情况。

用了不少篇幅，其实只要一幅图再加几段代码，读者可能就会明白其中的精髓。如图 5.9 所示，link 信号的高低用于控制双向信号的值是输出信号 yout 还是高阻态 Z。当 link 控制当前的输出状态为 Z 时，则输入信号 yin 的值由引脚信号 ytri 来决定。

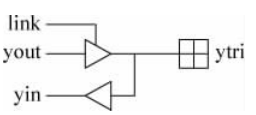

图 5.9 双向信号控制

实现代码如下：

```
//Verilog 例程
modulebidir(ytri,…);
inout ytri;
…

reg link;
wire yin;

…//link 的取值控制逻辑以及其他逻辑

assign ytri = link ? yout:1'bz;
```

```
assign yin = ytri;

…//yin 用于内部赋值

endmodule
```

9）提升系统性能的代码风格

下面要列举的代码示例是一些能够起到系统性能提升的代码风格。在逻辑电路的设计过程中，同样的功能可以由多种不同的逻辑电路来实现，那么就存在这些电路中孰优孰劣的问题。因此，带着这样的疑问，我们也一同来探讨一下几种常见的能够提升系统性能的编码技巧。请注意，本知识点所涉及的代码更多地是希望能够授人以“渔”，而非授人以“鱼”，读者应重点掌握前后不同代码所实现出来的逻辑结构，在不用的应用场合，可能会有不同的逻辑结构需求，那么就要学会灵活应变并写出适合需求的代码。

（1）减少关键路径的逻辑等级。

在时序设计过程中遇到一些无法收敛（即时序达不到要求）的情况，很多时候只是某一两条关键路径（这些路径在器件内部的走线或逻辑门延时太长）太糟糕。因此，设计者只要通过优化这些关键路径就可以改善时序性能。而这些关键路径所经过的逻辑门过多，往往是设计者在代码编写时误导综合工具所致。下面通过一个简单的例子，来看看两段不同的代码，其中的关键路径是如何明显得到改善的。

要实现如下的逻辑运算，它们的运算真值表如表 5.2 所示。

```
y = ((~a & b & c) | ~d) & ~e;
```

表 5.2 运算真值表

输入					输出
a	b	c	d	e	y
x	x	x	x	1	0
x	x	x	0	0	1
1	x	x	1	0	0
x	0	x	1	0	0
x	x	0	1	0	0
0	1	1	1	0	1

注：x 表示可以任意取 0 或 1。

按照常规的思路，可能会写出如下的代码：

```
//Verilog 例程
moduleexample(a,b,c,d,e,y);

input a,b,c,d,e;
output y;

wire m,n;
```

```
assign m = ~a & b & c;
assign n = m | ~d;
assign y = n & ~e;

endmodule
```

使用 Quartus Ⅱ 自带的综合工具，可以看到它的 RTL 视图如图 5.10 所示，与代码相吻合。

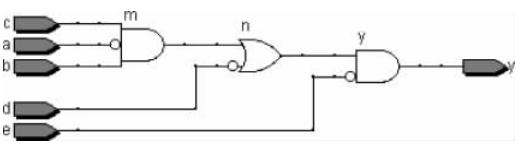

图 5.10 未优化前的综合结果

而现在假定输入 a 到输出 y 的路径是关键路径，其影响了整个逻辑的时序性能。那么，下面就要想办法从这条路径着手做一些优化的工作。很简单，目标是减少输入 a 到输出 y 之间的逻辑等级，目前是 3 级，可以想办法减少到 2 级甚至 1 级。

来分析一下公式“y=((~a & b & c) | ~d) & ~e;”，把“~a”从最里面的括号往外提取一级就等于减少了一级逻辑。当 a=0 时，y=((b & c) | ~d) & ~e；当 a=1 时，y=~d & ~e。由此不难得到“y=((~a | ~d) & ((b & c) | ~d)) & ~e;”与前式是等价的。可以修改前面的代码如下：

```
//Verilog 例程
module example(a,b,c,d,e,y);

input a,b,c,d,e;
output y;

wire m,n;

assign m = ~a | ~d;
assign n = (b & c) | ~d;
assign y = m & n & ~e;

endmodule
```

经过修改后代码的综合结果如图 5.11 所示，虽然 b 和 c 到 y 的逻辑等级还是 3，但是关键路径 a 到 y 的逻辑等级已经优化到了 2 级。与前面不同的是，优化后的 d 信号多了一级的负载，也多了一个逻辑门，这其实也是一种“面积换速度”思想的体现。正可谓“鱼和熊掌不可兼得”，在逻辑设计中往往需要在“鱼和熊掌”之间做抉择。

上面这个实例，只是一个未必非常恰当的“鱼”的例子。在前面章节里已经介绍过，在实际工程应用中，类似的逻辑关系可能在映射到最终器件结构时并非以逻辑门的方式来表现，通常是由 4 输入查找表来实现，那么它的优化可能与单纯简单逻辑等级的优化又有些不同。

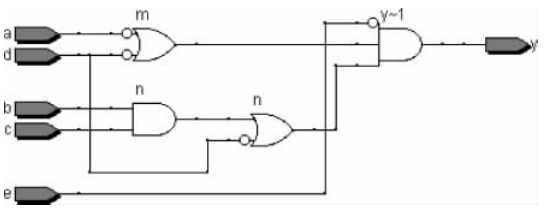

图 5.11 优化后的综合结果

不过，希望读者能在这个小例子中学到“渔”的技巧。

(2) 逻辑复制(减少重载信号的散出)与资源共享。

逻辑复制是一种通过增加面积来改善时序条件的优化手段。逻辑复制最主要的应用是调整信号的扇出。如果某个信号需要驱动的后级逻辑信号较多，也就是其扇出非常大，那么为了增加这个信号的驱动能力，就必须插入很多级的缓存(buffer)，这样就在一定程度上增加了这个信号的路径延时。这时可以复制生成这个信号的逻辑，用多路同频同相的信号驱动后续电路，使平均到每路的扇出变低，这样不需要插入 buffer 就能满足驱动能力增加的要求，从而减少该信号的路径延时。

资源共享和逻辑复制恰恰是逻辑复制的一个逆过程，它的好处在于节省面积，同时可能也要以速度的牺牲为代价。

看下面的一个实例：

```
//Verilog 例程
module example(sel,a,b,c,d,sum);

input sel,a,b,c,d;
output[1:0] sum;

wire[1:0] temp1 = {1'b0,a} + {1'b0,b};
wire[1:0] temp2 = {1'b0,c} + {1'b0,d};

assign sum = sel ? temp1:temp2;

endmodule
```

该代码综合后的视图如图 5.12 所示。和代码的表述一致，有两个加法器进行运算，结果通过 2 选 1 选择器后输出给 sum。

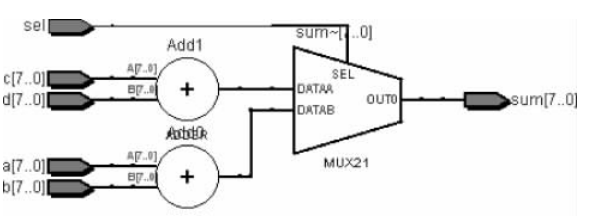

图 5.12 两个加法器的视图

同样实现这个功能，还可以这样编写代码：

```
//Verilog 例程
module example(sel,a,b,c,d,sum);

input sel;
input[7:0] a,b,c,d;
output[7:0] sum;

wire[7:0] temp1 = sel ? a:c;
wire[7:0] temp2 = sel ? b:d;

assign sum = temp1 + temp2;

endmodule
```

综合后的视图如图 5.13 所示，原先两个加法器现在用一个加法器同样可以实现其功能。而原先的一个 2 选 1 选择器则需要 4 选 2 选择器（可能是以两个 2 选 1 选择器来实现）来替代。如果在设计中加法器资源更宝贵一些，那么后面这段代码通过加法器的复用，相比前面一段代码更加节省资源。

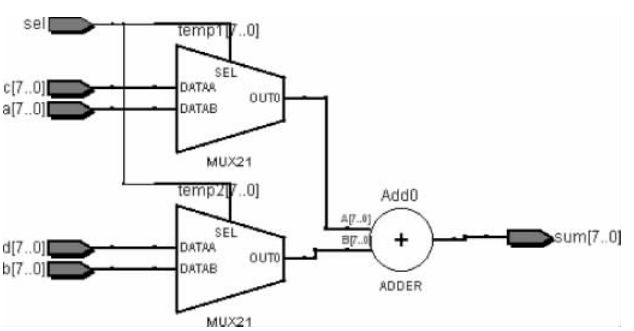

图 5.13 一个加法器的视图

（3）消除组合逻辑的毛刺。

在 3.2 节的最后部分，对于组合逻辑和时序逻辑的基本概念做了较详细的介绍，并且以一个实例说明了时序逻辑在大多数设计中更优于组合逻辑。组合逻辑在实际应用中，的确存在很多让设计者头疼的隐患，例如这里要说的毛刺。

任何信号在 FPGA 器件内部通过连线和逻辑单元时，都有一定的延时，正是通常所说的走线延时和门延时。延时的大小与连线的长短和逻辑单元的数目有关，同时还受器件本身的制造工艺、工作电压、温度等条件的影响。信号的高低电平转换也需要一定的上升或下降时间。由于存在这些因素的影响，多个信号的电平值发生变化时，在信号变化的瞬间，组合逻辑的输出并非同时，而是有先有后，因此往往会出现一些不正确的信号，例如一些很小的脉冲尖峰信号，称为“毛刺”。如果一个组合逻辑电路中有毛刺出现，就说明该电路存在“冒险”。

下面列举一个简单例子来看看毛刺现象是如何产生和消除的。如图 5.14 所示，这里在图 5.10 所示实例的基础上，对这个组合逻辑的各条走线延时和逻辑门延时做了标记。每个

门延时的时间是 2ns，而不同的走线延时略有不同。

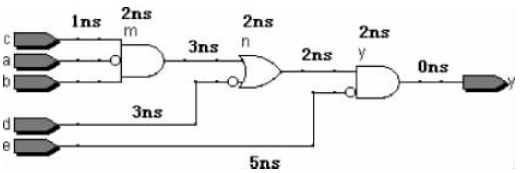

图 5.14　组合逻辑路径的延时标记

在这个实例模型中，不难计算出输入信号 a、b、c、d、e，从输入到输出（信号 y）所经过的延时。通过计算，可以得到 a、b、c 信号到达输出 y 的延时是 12ns，d 到达输出 y 的延时是 9ns，而 e 到达输出 y 的延时是 7ns。从这些传输延时中可以推断出，在第一个输入信号到达输出端 y 之前，输出 y 将保持原来的结果；而在最后一个输入信号到达输出端之后，输出 y 将获得所期望的新的结果。从本实例来看，7ns 之前输出 y 保持原结果，12ns 之后输出 y 获得新的结果。那么这里就存在一个问题，在 7ns 和 12ns 之间的这 5ns 时间内，输入 y 将会是什么状态呢？

如图 5.15 所示，这里列举一种出现毛刺的情况。假设在 0ns 以前，输入信号 a、b、c、d、e 取值均为 0，此时输出 y=1；在 0ns 时，b、c、d 由 0 变化为 1，输出 y=1。在理想情况下，输出 y 应该一直保持 1 不变。但从延时模型来看，实际上在 9～12ns 期间，输出 y 有短暂的低脉冲出现，这不是电路应有的状态，其实这就是这个组合逻辑的毛刺。

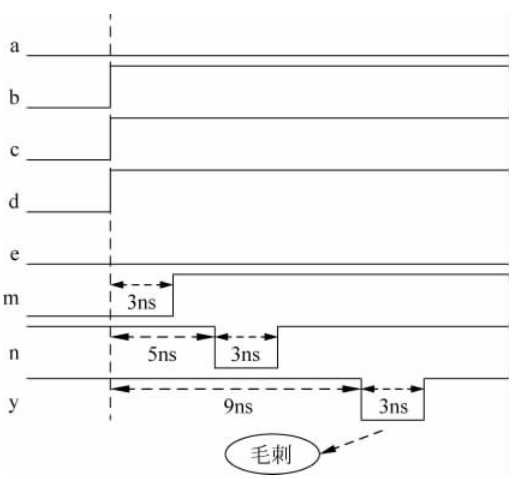

图 5.15　逻辑延时波形

既然多个输入信号的变化前后取值都保持高电平，那么这个低脉冲的毛刺其实不是所希望看到的，在后续电路中，这个毛刺很有可能导致后续的采集出现错误，甚至使得一些功能被误触发。

要消除这个毛刺，通常有两种办法。一种办法是“硬”办法，如果在 y 信号上并联一个电容，便可轻松地将这类脉冲宽度很小的干扰滤除。但是，现在是在 FPGA 器件内部，还真没有条件和可能这么处理，那么只能放弃这种方法。另一种办法其实就是引入时序逻辑，用寄存器使输出信号多打一拍，这也是时序逻辑明显优于组合逻辑的特性。

如图 5.16 所示，在原有组合逻辑的基础上，添加了一个寄存器用于锁存最终的输出信号 y。

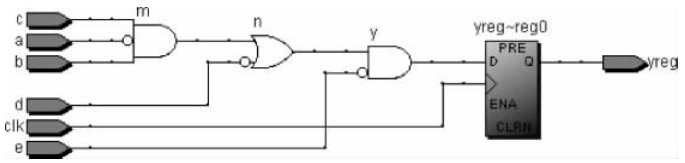

图 5.16 寄存器锁存组合逻辑输出

如图 5.17 所示，在引入了寄存器后，新的最终的输出 yreg 不再随意地改变，而是在每个时钟 clk 的上升沿锁存当前的输出值。

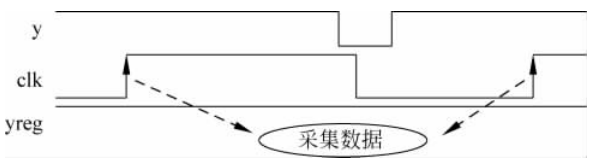

图 5.17 寄存器锁存波形

引入时序逻辑后，并不是说完全就不会产生错误的数据采集或锁存。在时序逻辑中，只要遵循一定的规则就可以避免很多问题，如保证时钟 clk 有效沿前后的数据建立时间和保持时间内待采集的数据是稳定的。