第5章 Verilog代码规范和代码风格

强调 Verilog 代码编写规范，有助于提高书写代码的可读性、可修改性、可重用性，便于优化代码综合和仿真的结果。指导设计工程师使用 Verilog HDL 规范代码和优化电路，从而做到：① 逻辑功能正确；②可快速仿真；③ 综合结果最优（如果是 hardware model）；④可读性较好。

代码规范之后的一个境界是优良的代码风格，代码风格不同于代码规范，其重点强调逻辑上的风格，同样的功能，使用不同的代码风格，代码综合面积可能是几倍的关系；另外，人们不经意间的编程习惯可能会导致许多冗余代码，在 Verilog 综合之后，即使通过了最大化的优化计算，这些冗余未必能全部被优化掉，依旧会产生不必要的电路，比如多余的寄存器或者没用的组合逻辑，从而导致更多的流片成本或占用更多的 FPGA 硬件资源。

本章主要介绍编写 Verilog HDL 代码时的代码规范和代码风格，给出了基于数字系统设计的一些基本原则和设计技巧的代码风格，这些代码风格不仅适用于 ASIC 设计，同时也适用于 FPGA 设计。

5.1 Verilog 代码规范

遵循代码编写规范书写的代码，很容易阅读、理解、维护、修改、跟踪调试、整理文档。相反，编写风格随意的代码，通常晦涩、凌乱，会给开发者本人的调试、修改工作带来困难，也会给读者带来很大麻烦。Verilog 代码规范主要包括命名、格式、注释及语法规范，下面将详细论述各个规范的内容。

5.1.1 命名规范

1. 文件名

每个模块（module）一般应存在于单独的源文件中，便于模块的修改，通常源文件名与所包含模块名相同。模块、文件命名时采用功能命名，例如：

```
u1_mux.v          上行复用模块
```

模块层次尽可能不要超过 4 层，低层模块的命名要求包含上层模块名，例如：

```
u1_mux_reg.v      上行复用寄存器模块
```

这样便于理解 FPGA 的模块层次结构和功能。调用 CORE 模块除外。

此外还应注意，Verilog 程序必须存入某文件夹中（要求非中文文件夹名），不要保存在根目录内或桌面上。

2. 模块名

（1）在系统设计阶段应该为每个模块命名，最终的顶层模块应该以芯片的名称来命名。在顶层模块中，除 I/O 引脚和不需要综合的模块外，其余作为次级顶层模块，建议以“xx_core.v”命名。

（2）调用模块的命名与该模块名匹配。

同一模块内调用同一子模块时，调用名采用整数索引或采用整数多次索引，以增加模块的可读性，避免混淆。例如：

```
block  block_1(…);
block  block_2(…);
block  block_3(…);
```

3. 信号名

（1）采用有意义的，能反映对象特征、出处、功能和性质的单词命名，可以达到望文生义，以增强程序的可读性。例如，count8<=count8+8'h01 就显得含糊不清，而 addr_count<=addr_count+8'h01 就表明了意义。

（2）长的名字对书写和记忆会带来不便，甚至带来错误，所以要避免标识符过于冗长，对较长的单词应当采用适当的缩写形式，如用 rst 代替 reset，en 代替 enable，addr 代替 address，clk 代替 clock 等。

（3）在 RTL 源码的设计中任何元素包括端口、信号、变量、函数、任务、模块等的命名都不能与 Verilog 和 VHDL 的关键字同名。

（4）如果需要多个意义独立的字符串命名，字符串之间要用下画线“_”隔开，便于维护，有助于对设计的理解，如 ram_addr、max_delay、data_size 等。

（5）总线由高位到低位命名，如 bus[31:0]。

（6）对来自同一驱动源的信号在不同的子模块中采用相同的名字，这要求在芯片总体设计时就定义好顶层子模块间连线的名字，端口和连接端口的信号尽可能采用相同的名字。

5.1.2 格式规范

1. 空行和空格

（1）适当地在代码的不同部分插入空行，避免因程序拥挤不利阅读。例如，分节书写，各节之间加 1 行到多行空格，如每个 always、initial 语句都是一节。

（2）不同变量，以及变量与符号、变量与括号之间都应当保留一个空格。Verilog 关键字与其他任何字符串之间都应当保留一个空格，如 always @（…）。

（3）使用//进行的注释，在//后应当有一个空格。

(4) 在表达式中插入空格,避免代码拥挤,包括:

① 赋值符号两边要有空格。

② 双目运算符两边要有空格。

③ 单目运算符和操作数之间可没有空格。

例如:

```
a  <=  b;
c  =  a  +  b;
if (a  ==  b) then …
  e  <=  ~a  &  c;
```

2. 对齐和缩进

(1) 不要使用连续的空格来进行语句的对齐。

(2) 采用制表符 Tab 对语句对齐和缩进,Tab 键采用 4 个字符宽度,可在编辑器中设置。

(3) 各种嵌套语句尤其是 if…else 语句,必须严格地逐层缩进对齐。

(4) 同一个层次的所有语句左端对齐;initial、always 等语句块的 begin 关键词和相应的 end 关键词与 initial、always 对齐。

(5) 每行只写一条语句可增加程序的可读性,便于用设计工具进行代码的语法分析。

3. 注释

必须加入详细、清晰的注释行以增强代码的可读性和可移植性,注释内容占代码篇幅不应少于 30%。使用//进行的注释行以分号结束;使用/* */进行的注释,/*和*/各占用一行,并且顶头。例如:

```
//edge detector used to synchronize the input signal;
```

代码行使用单行注释,不使用多行注释,代码行注释跟在注释代码之后,处于同一行。注释应简明扼要,避免使单行内容过长。如注释过长且难以简略,可以分行注释。注意放在下一行的注释应与前行注释左侧对齐。注意分行的注释内容要独占一行,该行不能有其他的代码。若代码本身较长,难以在同一行加以注释,可以在代码的前一行放置注释内容。注意这行注释要独占一行。

4. 模块调用格式

在 Verilog 中有两种模块调用的方法,一种是位置映射法,严格按照模块定义的端口顺序来连接,不用注明原模块定义时规定的端口名,其语法为:

```
模块名 (连接端口 1 信号名,连接端口 2 信号名,连接端口 3 信号名,…);
```

另一种为信号映射法,即利用“.”符号,表明原模块定义时的端口名,其语法为:

```
模块名 (.端口 1 信号名(连接端口 1 信号名),
        .端口 2 信号名(连接端口 2 信号名),
        .端口 3 信号名(连接端口 3 信号名),…);
```

显然，信号映射法同时将信号名和被引用端口名列出来，不必严格遵守端口顺序，不仅降低了代码易错性，还提高了程序的可读性和可移植性。因此，在良好的代码中，严禁使用位置映射法，全部采用信号映射法。

5. 大小写

如无特别需要，模块名和信号名一律采用小写字母；为醒目起见，自己定义的常数(`define 定义)/参数(parameter 定义)采用大写字母，如 parameter CYCLE=100。

6. 参数化设计格式

为了源代码的可读性和可移植性，不要在程序中直接写特定数值，尽可能采用`define 语句或 parameter 语句定义常数或参数。

5.1.3 RTL 可综合代码编写规范

用 HDL 实现电路，设计人员对可综合风格的 RTL 描述的掌握不仅会影响到仿真和综合的一致性，也是逻辑综合后电路可靠性和质量好坏最主要的因素，对此应当予以充分的重视。经常考虑以下几个方面。

(1) 每个模块尽可能只使用一个时钟，用一个时钟的上升沿或下降沿采样信号，不能一会儿用上升沿，一会儿用下降沿。如果既要用上升沿又要用下降沿则应分成两个模块设计。建议在顶层模块中对 Clock 做一非门，在层次模块中如果要用时钟下降沿就可以用非门产生的时钟，这样做的好处是在整个设计中采用同一时钟触发有利于综合。

(2) 代码描述应该尽量简单，如果在编码过程中无法预计其最终的综合结果，那综合工具可能会花很长的时间。

(3) 在内部逻辑中避免使用三态逻辑。

(4) 避免触发器在综合过程中生成锁存器。

(5) 尽量避免异步逻辑、带有反馈环的组合逻辑。

(6) 避免不必要的函数调用，重复的函数调用会增加综合次数，不仅会造成电路面积的浪费，还会使综合时间变长。

虽然不同的综合工具对 Verilog HDL 语法结构的支持不尽相同，但 Verilog HDL 中某些典型的结构是很明确地被所有综合工具支持或不支持的。

(1) 所有综合工具都支持的结构有 always、assign、begin、end、case、wire、tri、supply0、supply1、reg、integer、inout、input、instantitation、module、negedge、posedge、operators、output、parameter、default、for、function、and、nand、or、nor、xor、xnor、buf、not、bufif0、bufif1、notif0、notif1、if。

(2) 有些工具支持有些工具不支持的结构有 casex、casez、wand、triand、repeat、task、while、wor、trior、real、disable、forever、arrays、memories。

(3) 所有综合工具都不支持的结构有 time、defparam、$finish、fork、join、initial、delays、UDP、wait。

5.1.4 常见错误

对于初学者,在编写代码中经常出现以下错误。

(1) 语句的结尾缺少分号。

(2) 对包含多条语句的块语句,缺少 begin…end 语句或 begin…end 语句不匹配。

(3) 对连续赋值的左边不声明为线网型变量,对过程赋值的左边不声明为寄存器类型。

(4) 对二进制数缺少基('b)(也就是说,编译器将它们看作十进制数)。

(5) 在编译器伪指令中错误使用撇号(应当是后撇号或重音符号“`”)和数字基(应当是一般的单引号或倒转的逗号“’”)。

5.2 Verilog 代码风格

本节介绍基于数字系统设计原则及技巧的 Verilog 的代码风格。在进行数字系统设计时,可以使用 ASIC 设计方式,也可以使用 FPGA 设计方式。ASIC 设计和 FPGA 设计有以下几点不同:ASIC 设计的功耗比 FPGA 设计要低得多;ASIC 能完成高速设计,工作频率可在 10GHz 以上,而 FPGA 目前最快频率不过 500MHz,对于大规模器件,资源利用率达到 250MHz 都是非常困难的;ASIC 设计密度大,而 FPGA 底层硬件结构一致,在实现用户设计时会有大量单元不能充分利用,所以 FPGA 的设计效率并不高。与 ASIC 相比,FPGA 的等效系统门和 ASIC 门的设计效率比约为 1∶10。由于 ASIC 设计成本高,周期长等特点,设计人员也可以使用基于 FPGA 的设计作为 ASIC 设计流程中的验证手段。设计基于 FPGA 的硬件设计看似复杂,实则存在很多设计方法和内在规律,如果多做多练,必然可以掌握其中的技巧。

5.2.1 基本原则

基于 FPGA 的数字系统设计通常遵照以下几点基本原则。

1. 资源共享

在 ASIC 设计中,硬件设计资源和面积是一个重要的技术指标。对于 FPGA/CPLD,其芯片面积(逻辑资源)是固定的,但有资源利用率问题,“面积”优化是一种习惯上的说法。

这里的“面积”常用所消耗的逻辑单元数和寄存器来衡量。在 Quartus Ⅱ 9.1 软件中,“面积”消耗可从 Compilation Report 中的 Flow Summary 中看到。面积优化的实现有多种方法,其中,资源共享(Resource Sharing)是一个较好的方法,尤其是将一些耗用资源较多的模块进行共享,能有效降低系统耗用的资源。

例如,要实现这样的功能,当 sel=0 时,mult= a * b;当 sel=1 时,mult= c * d。下面给出了两种实现方案,由于采用的代码风格不一样,综合后的 RTL 结构也不一样,如图 5-1 和图 5-2 所示。

```
//方案 1:用两个乘法器和一个 MUX 实现
module mult1 (mult, sel, a, b, c, d);
input[3:0] a,b,c,d;
input sel;
output[7:0] mult;
reg[7:0] mult;
always @ ( * )
begin
if(sel == 0) mult = a * b;
else mult = c * d;
end
endmodule
```

```
//方案 2:用两个 MUX 和一个乘法器实现
module mult2 (mult, sel, a, b, c, d);
input[3:0] a,b,c,d;
input sel;
output[7:0] mult;
wire[7:0] mult;
reg[3:0] temp1,temp2;
always @ ( * )
begin
if(sel == 0) begin temp1 = a;temp2 = b;end
else begin temp1 = c;temp2 = d;end
end
assign mult = temp1 * temp2;
endmodule
```

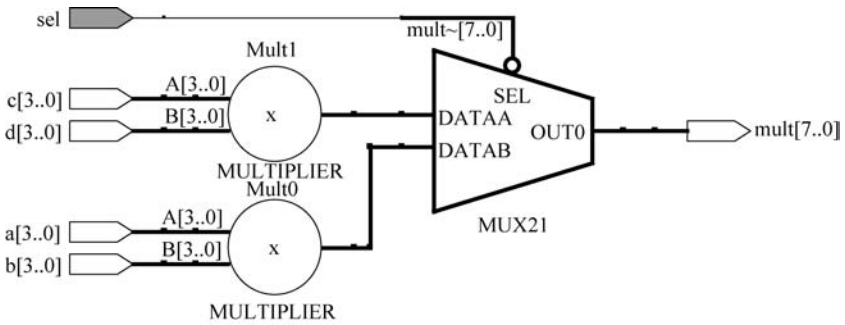

图 5-1 mult1 综合视图

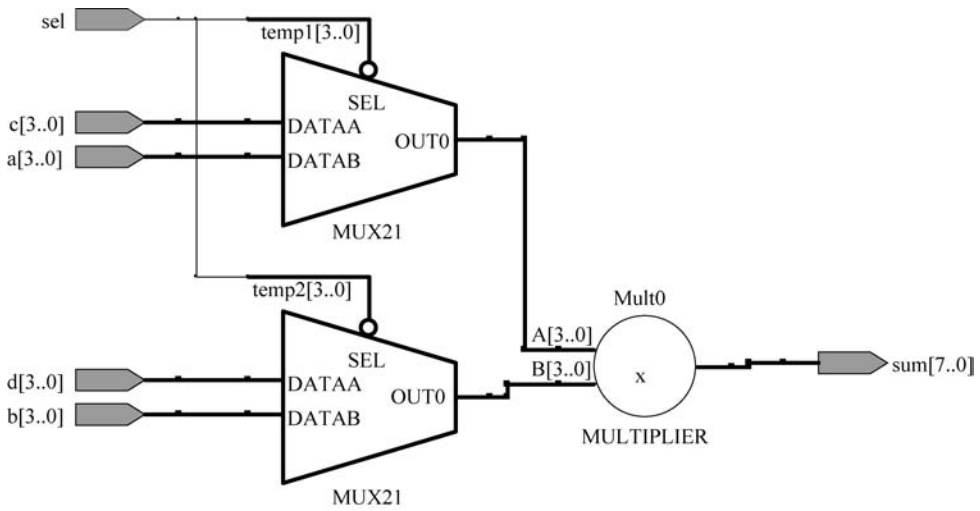

图 5-2 mult2 RTL 综合视图

从方案 1 对应的如图 5-1 所示的电路结构中可以看到,电路采用了两个乘法器,一个 MUX,乘法器在设计中面积占有率最大。而如图 5-2 所示的电路,增加了一个耗用资源比较小的 MUX,由于共享了乘法器,而减少了一个耗用资源多的乘法器,因此方案 2 更节省

资源。所以,在电路设计中,应尽可能通过选择、复用的方式使硬件代价高的功能模块资源共享,以减少该模块的使用个数,达到减少资源使用、优化面积的目的。目前,Quartus Ⅱ和Synplify Pro等高级的HDL综合器通过设置就能自动识别设计中需要资源共享的逻辑结构,自动地进行资源共享。但这种设计风格值得读者关注。

在代码中,还可以用括号等方式控制综合的结果,尽量实现资源的共享,重用已计算过的结果。

例如,下面两段代码实现三个4位数相乘,对应的RTL综合视图分别如图5-3和图5-4所示。

```
//乘法方案1
module mult3 (m1,m2,a, b, c);
input[3:0] a,b,c;
output[7:0] m1;
output[11:0]m2;
reg[11:0] m2;
reg[7:0]m1;
always @ ( * )
begin
m1 = a * b;
m2 = c * a * b;
end
endm
```

```
//乘法方案2
module mult4 (m1,m2,a, b, c);
input[3:0] a,b,c;
output[7:0] m1;
output[11:0]m2;
reg[11:0] m2;
reg[7:0]m1;
always @ ( * )
begin
m1 = a * b;
m2 = c * (a * b);
end
```

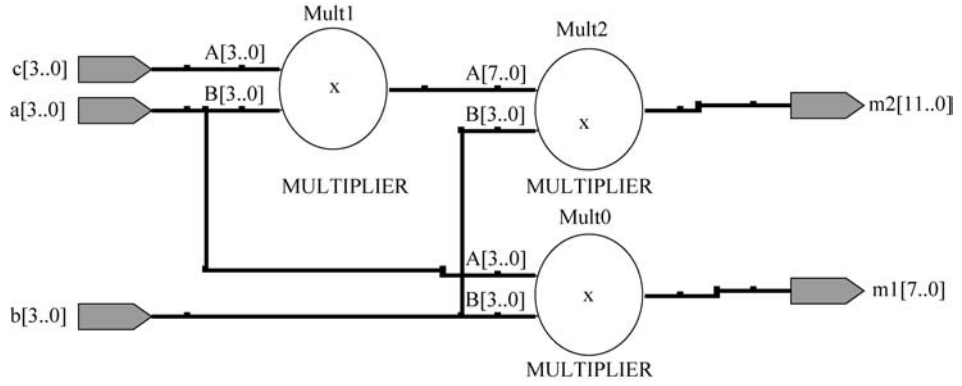

图5-3 mult3 RTL综合视图

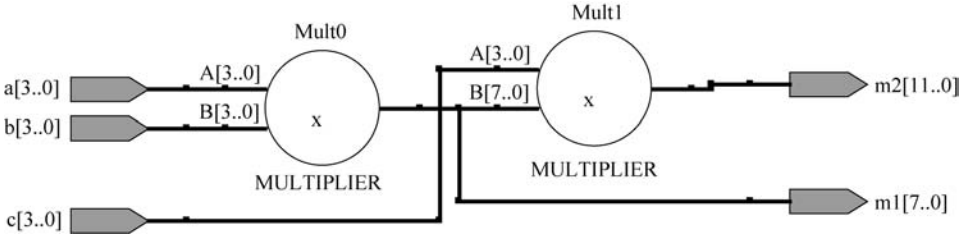

图5-4 mult4 RTL综合视图

上面两段代码实现的功能是完全相同的,但综合的结果却不同,方案1耗用了三个乘法器,而方案2只用了两个乘法器,因此方案2更优,这是因为方案2用括号控制了综合的结

果，重用了已计算过的值 m1，因此节省了资源。

2. 面积和速度的折中考虑

“速度”指设计在芯片上稳定运行时所能够达到的最高时钟频率，它不仅和 FPGA 内部各个寄存器的建立时间、保持时间以及 FPGA 与外部器件接口的各种时序要求有关，而且还和两个紧邻的寄存器间（有紧密逻辑关系的寄存器）的逻辑延时、走线延时有关。在 Quartus Ⅱ 9.1 软件中，“速度”情况可从 Compilation Report 中的 Timing Analyzer 中看到。由于 FPGA/CPLD 的逻辑资源、连接资源和 I/O 资源有限，器件的速度和性能也是有限的，用器件设计系统的过程相当于求最优解的过程。最优化目标有多种，设计中常见的最优化目标有：器件资源利用率最高；系统工作速度最快，即延时最小；布线最容易，即实现性最好。具体设计中，各个最优化目标间可能会产生冲突，这时应满足设计的主要要求。最常见的冲突就是面积（器件资源利用率）和速度的冲突，面积和速度这两个指标贯穿着 FPGA 设计的始终，是设计质量评价的终极目标。优秀的设计必然要求面积尽可能小，速度尽可能快。但面积和速度是一对对立的矛盾体，同时具有设计面积最小、运行频率最高并不现实。科学的办法是在满足设计时序要求的前提下，尽可能地占用最小的芯片面积；或者是在所规定的面积之下，使系统的运行频率尽可能高，设计的健壮性更强。当面积和速度有冲突时，它们产生的影响也并不一样。一般来说，满足设计时序要求更加重要，因此选择速度优先。

在 CPLD/FPGA 设计中，有一个重要的设计思想：面积和速度互换。面积和速度的互换一方面是指，如果一个设计的时序余量（slack）较大，运行速度远远超过所需要的速度要求，此时可以通过功能模块复用的方法减小芯片面积，代价是速度有所降低，也就是说，用速度的优势换取面积的节约；另一方面是指，若设计的时序要求很高，一般方法达不到所需要的时序要求时，可以通过数据流串并转换、操作模块的复制使设计时序达到要求，代价是面积有所增加，也就是说，用面积复制换取速度提高。

速度换面积的常用方法是功能模块时分复用。功能模块时分复用是指通过设计更高频率的功能模块，使得原来两个较低频率模块的逻辑功能由一个较高频率模块通过时分复用的方式完成，这样就实现了面积的缩小和频率的提高。功能模块时分复用原理如图 5-5 所示。

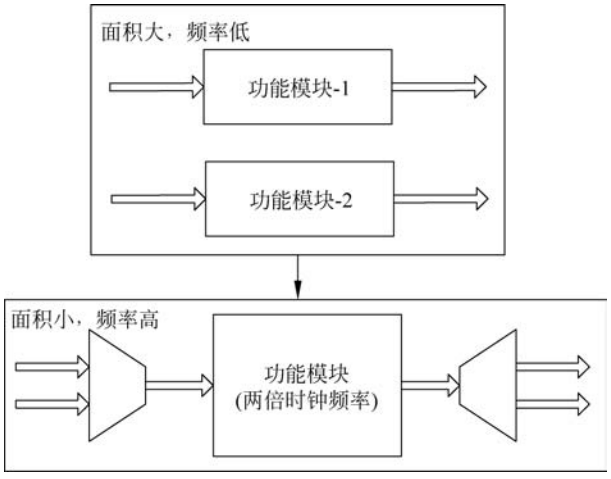

图 5-5 速度换面积之功能模块时分复用

面积换速度的常用方法是逻辑复制,其重要的应用是调整信号的扇出。在扇出很大的电路结构中,为了增强输出信号的驱动能力,一般方法是增加多级 buffer,这样就在一定程度上增加了信号的路径延时。如果在面积上不是要求特别苛刻时,可以通过复制输出信号的生成逻辑,来使多路同频同向信号驱动后续电路,从而减小每路输出的扇出,因此也不用增加 buffer 导致延时较大。逻辑复制原理如图 5-6 所示。

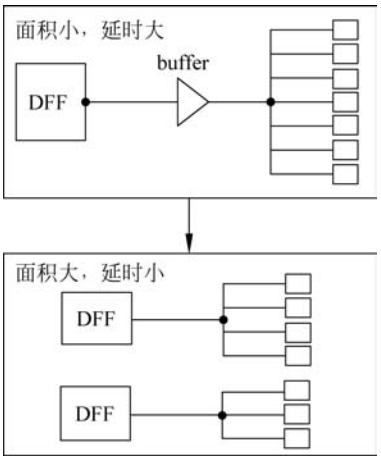

图 5-6 面积换速度之逻辑复制

3. 写代码时考虑硬件结构

在 FPGA 设计中常常使用 Verilog HDL 进行设计。但是务必要注意,虽然使用高级语言编写了代码,但是这绝不等同于软件设计,这是使用硬件描述语言进行的硬件电路设计。在编写代码进行硬件设计时一定要具备硬件设计思想,勾画出硬件情况,然后使用语言描述出来,这样综合工具才能快速有效地综合出最优结构。

评价 Verilog 代码的优劣不在于代码段的整洁简短,而在于代码是否能由综合工具流畅合理地转换成速度快和面积小的硬件形式。这是硬件语言编写代码和软件语言编写代码的最大不同之处,需要初学者不断练习才能体会。不同的代码逻辑功能相同,但是硬件结构大不相同,设计具有可综合风格的硬件具有重要意义。

4. 最好使用同步设计

电路设计可以是异步设计也可以是同步设计,但是异步设计的时序正确性完全取决于每个逻辑元件和布线的延迟情况,非常容易产生毛刺现象和亚稳态等,且难于处理,容易引起系统不稳定,而使用由时钟沿驱动的同步设计可以很好地避免毛刺情况,使系统稳定性和可靠性更好并且可以简化时序分析过程、减少工作环境对设计的影响。随着 FPGA/CPLD 设计规模的逐渐增加,片上时钟分布的质量变得非常重要,要充分有效地利用 FPGA/CPLD 专用的时钟分布资源和使用方法,产生高扇出低畸变的时钟信号。

在同步设计中,也需要满足一些原则才能保证系统的正确和稳定运行。

(1) 满足建立时间(setup time):建立时间是指触发器的时钟上升沿(如上升沿有效)到来以前,数据稳定不变的时间。即输入信号应提前于时钟上升沿 setup time 时间到达,如不满足 setup time,这个数据就不能被这一时钟打入触发器,只有等下一个时钟上升沿,数据才能被打入触发器。

(2) 满足保持时间(hold time):触发器的时钟信号上升沿到来以后,数据稳定不变的时间。如果 hold time 不够,数据同样不能被打入触发器。

对于系统而言,有时会有不同时钟域的设计,在尽量不用异步设计的同时,要重点考虑异步时钟域的数据转换问题。

5. 分模块设计方法

分模块设计方法也即结构层次化设计方法。目前大型设计中必须采用结构层次化设

计，以提高代码可读性，易于分工合作，易于仿真测试。一般来说，分模块至少两层，不超过5层。如图5-7所示是分模块设计示意图(图中模块层次为3层)。

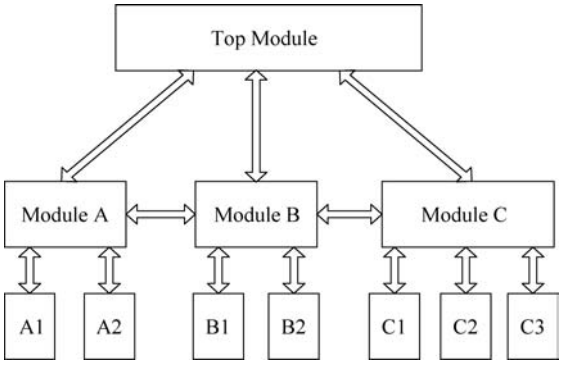

图5-7 分模块设计示意图

很多初学者都会有些不理解，明明一个设计可以在一个模块中描述清楚，为什么还要对其分模块设计，似乎发现分模块设计仅增加了内部接口描述的工作量，并没有体现出优势。实际上，对于较小的系统，使用分模块设计看不到什么优势，但是对于比较复杂的设计，如果能将其中的时序逻辑和组合逻辑分开，将不同优化目标分开，将不同功能的电路分开，不仅利于设计的分工合作，阅读和维护，而且决定了综合时的耗时和效率。因此，从初始设计开始，就要使用分模块方法，养成良好的设计习惯，为复杂设计打下基础。

在分模块设计时，有以下一些原则需要注意。

(1) 顶层模块主要完成对子模块的组织和调用，最好不要有复杂的逻辑功能。一般顶层模块包括：输入/输出引脚说明、模块调用、时钟与置位/复位、三态缓冲和简单组合逻辑。

(2) 子模块的划分一定要合理，要综合考虑功能、时序、复杂度等因素。有以下几点划分原则。

① 将相关逻辑或可复用逻辑划分在同一模块内，这样综合时会获得更好的综合优化效果。

② 对子模块的输出尽量使用寄存器。这也是从综合工具的角度考虑，便于综合工具权衡子模块中的组合部分和时序部分，从而达到更好的时序优化效果。

③ 将不同的优化目标分开设计。在设计之初，就已经初步规划了设计规模和时序关键路径。对于不同的模块，综合工具仅考虑一种优化目标和策略。例如，对于时序可能会紧张的部分要独立划分模块，这样综合时可单独设置优化目标是“speed”。对于资源消耗可能过大的部分也独立划分模块，综合时的优化目标为“area”。

④ 将时序要求不高的逻辑(如多周期路径)独立划分模块，综合时指定较松散的约束条件，则达到节省面积资源目的。

⑤ 将存储逻辑独立划分模块，不仅提高仿真时速度，也利于综合工具的综合。

(3) 为增加设计可读性和可维护性，尽量不要在深层次的模块间建立接口，也不要跨层次建立接口。如图5-7中，建立接口可以在第二层的Module A、Module B和Module C之

间,但是不要在第三层的 A1,A2,B1,B2,C1,C2,C3 之间建立,也不要在第二层和第三层之间建立。

5.2.2 设计技巧

在 FPGA 设计中,离不开一些基本设计。对于这些基本设计都有一些设计上的小技巧。使用这些小技巧可以使设计更加有效、快速和稳定。

1. 串并/并串转换技巧

串并/并串转换是处理数据流的常用技巧,是面积与速度互换的直接体现。对于串转并,是指串行的数据流转变为并行的数据流。进行串转并的目的在于,通过复制逻辑,实现数据吞吐率的提高,即通过面积的消耗来实现速度的提高。

对于小的设计来说,主要用寄存器实现串转并。具体实现代码如下。

```
input clk;
input serial_in;
reg [N-1:0] parallel_out;
always@(posedge clk)
parallel _out <= {parallel_out,serial_in};
```

其中,data_in 是指串行数据流,parallel_out 为并行输出的缓存寄存器。

对于排列顺序有特别规定的串并转换,可以使用 case 语句实现。对于更加复杂的串并转换,可以使用状态机实现。

2. 流水线设计技巧

流水线的运用是实现高速设计的一个常用技巧。流水线设计的代码风格是将组合逻辑系统地分割,并在各个部分(分级)之间添加寄存器,暂存中间数据的方法,目的是将一个大操作分解成若干小操作,各小操作能并行执行。若设计的数据流是单方向流动,即没有反馈或者迭代运算,前一个步骤的输出是下一个步骤的输入,则使用流水线设计可以提高系统的工作频率。流水线设计的代价是增加寄存器逻辑,增加了芯片资源的耗用,也是通过面积换取速度的典型体现。如图 5-8 所示为三步骤流水线设计结构图。

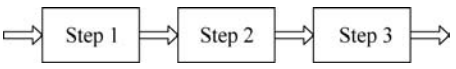

图 5-8 三步骤流水线设计结构图

三步骤流水线设计的时序图如图 5-9 所示。在流水线设计中,重点在于每个步骤中设计时序的合理安排,要保证前后级接口数据流的匹配。尽量使前级操作时间大致等于后级操作时间,若前级操作时间小于后级操作时间,则要对前级输出进行缓存,否则会出现后级数据的溢出。若前级操作时间大于后级操作时间,则需要对前级进行逻辑复制或串并转换等手段来进行数据分流,否则会造成前后级的处理节拍不匹配。

下面以 8 位全加器的设计为例,对比非流水线设计和流水线设计的性能。

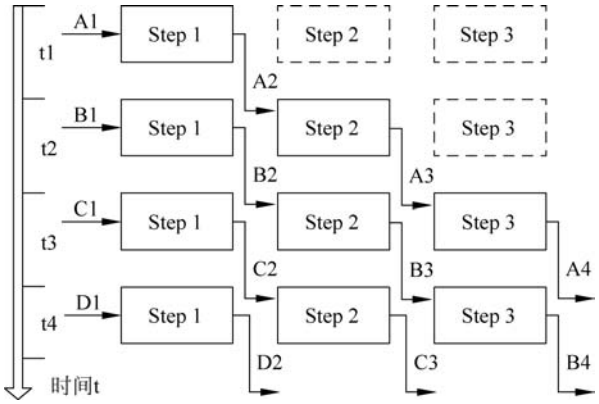

图 5-9 三步骤流水线设计时序图

1. 普通 8 位全加器

```
module add8(sum,cout, clk,cin,ina,inb);
input           clk, cin;
input [7:0]     ina,inb;
output[7:0]     sum;
output          cout;
reg             cout;
reg[7:0]        sum;
always @ (posedge clk)
{cout, sum} = ina + inb + cin;
endmodule
```

2. 两级流水线方式实现 8 位全加器

图 5-10 为两级流水线加法器实现框图，从图中可以看出，该加法器采用了两级锁存、两级加法，第一级低 4 位加法，第二级高 4 位加法，首次延迟需要两个周期，但执行重复操作时，只要一个时钟周期来获得最后的计算结果。代码如下。

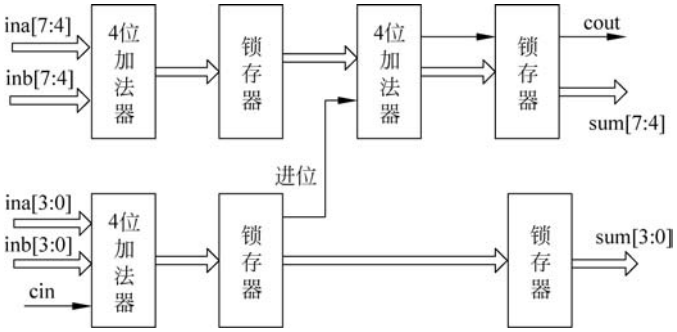

图 5-10 8 位全加器两级流水线实现框图

```
module add8_pip(sum,cout, clk,cin,ina,inb);
input          clk, cin;
input [7:0]    ina,inb;
output[7:0]    sum;
output         cout;
reg            cout;
reg            cout1;              //插入的寄存器
reg [3:0]      sum1 ;              //插入的寄存器
reg [7:0]      sum;
reg [3:0]      tempa, tempb;       //插入的寄存器
always @(posedge clk)              //第一级流水
begin
    {cout1 , sum1} = ina[3:0] + inb [3:0] + cin ;
    tempa = ina[7:4]; tempb = inb[7:4];
end
always @(posedge clk)              //第二级流水
begin
    {cout ,sum[7:0]} = {{1'b0, tempa} + {1'b0, tempb} + cout1 , sum1[3:0]} ;
end
endmodule
```

流水线就是插入寄存器,以面积换取速度。而 FPGA 的寄存器资源非常丰富,所以对 FPGA 设计而言,流水线是一种先进的而又不耗费过多器件资源的结构。但是采用流水线后,数据通道将会变成多时钟周期,所以要特别考虑设计的其余部分,解决增加通路带来的延迟。

3. 乒乓操作技巧

乒乓操作是一个常常应用于数据流控制的处理技巧,可以给数据处理模块赢得更多的处理时间,可避免由于数据处理时无法接收数据而导致的数据丢失。原理图如图 5-11 所示。

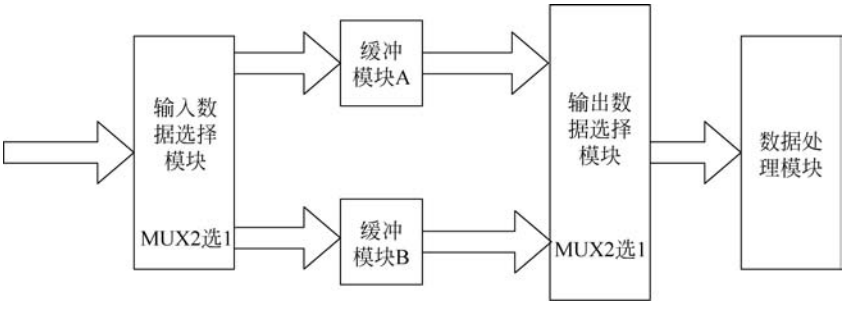

图 5-11 乒乓操作原理图

输入数据流通过“输入数据选择模块”后,轮流等时地被分配到两个数据缓冲区模块 A 和 B。缓冲模块可以是任何存储模块,如双口 RAM(DPRAM)、单口 RAM(SPRAM)、FIFO 等。在第 1 个缓冲周期,输入的数据流经过“输入数据选择模块”的切换缓存到“缓冲模块 A”,在第 2 个缓冲周期,通过“输入数据选择模块”的切换,将输入的数据流缓存到“缓

冲模块 B”，同时将“缓冲模块 A”缓存的第 1 周期数据通过“输出数据选择模块”的选择，送到“数据处理模块”进行运算处理；在第 3 个缓冲周期通过“输入数据选择模块”的再次切换，将输入的数据缓存到“缓冲模块 A”，同时将“缓冲模块 B”缓存的第 2 个周期的数据通过“输出数据选择模块”切换，送到“数据处理模块”进行运算处理，如此循环。

通过输入数据选择模块和输出数据选择模块的相互配合，乒乓操作实现了缓冲数据流流畅无停顿地被送入数据处理模块进行处理。从整体来看，输入数据流和输出数据流都是连续不断的，因此适合对数据流进行流水线式处理，可实现数据的无缝缓冲和处理。

由于设定了不同的缓存模块，分节拍工作，也实现了缓存区空间的节约。主要应用在处理以帧为单位的数据处理中，而且每帧的处理时间小于帧周期的情况。

另外，通过乒乓操作，可实现使用低速处理模块处理高速数据流的效果，如图 5-12 所示。假设输入数据的速率为 100Mb/s，则可使用两个处理模块来进行数据处理，每个处理模块的处理速率为 50Mb/s。实际上，通过乒乓操作实现低速模块处理高速数据的本质是，缓存模块的使用实现了数据流的串并转换，使用两个处理模块来并行处理数据，最后再经过输出数据选择模块实现并串转换。这其实也是通过面积来换取速度的典型方法。

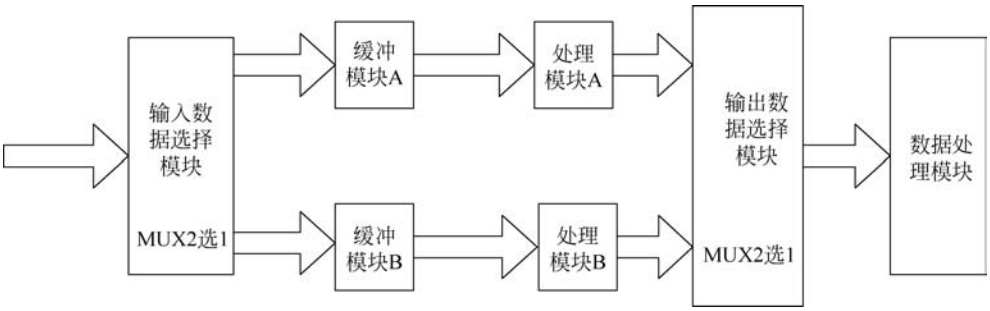

图 5-12 乒乓操作实现低速模块处理高速数据

5.3 小结

本章列出的代码编写规范，无法覆盖代码编写的方方面面，还有很多细节问题，需要在实际编写过程中加以考虑。并且有些规定也不是绝对的，需要灵活处理，并不是律条。但是在一个项目组内部、一个项目的进程中，应该有一套类似的代码编写规范来作为约束。总的方向是，努力写整洁、可读性好的代码。最后给出了基于 FPGA 的数字系统设计原则和设计技巧的代码风格。这些知识是进行优秀设计必须具备的基础，可以从宏观上指导初学者写出更好的设计。当然，所有的设计原则和方法，只是方法论的内容。读者必须从一个个小的设计开始做起，不断总结思考，不断实践，才能真正写出优秀的设计代码。

习题 5

1. 如何避免在综合过程中产生异步逻辑和带有反馈环的组合逻辑？
2. 举例说明触发器在什么情况下会在综合过程中生成锁存器。

3. 利用资源共享的方法对下面的程序进行面积优化。

```
module add4(sum, sel, a, b, c, d);
input[3:0] a,b,c,d;
input sel;
output[7:0] sum;
reg[7:0] sum;
always @ ( * )
if(sel == 0) sum = a + b;
else su = c + d;
endmodule
```

4. 列出几种基于 FPGA 的数字系统设计的基本原则和设计技巧。

5. 用 4 级流水线方式实现 8 位全加器。