

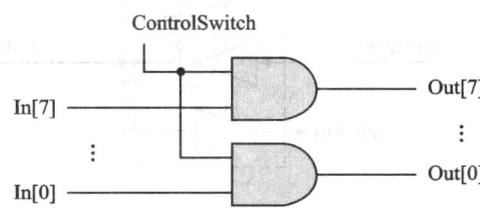
器、复杂的数据流动控制逻辑、运算控制逻辑、指令分析和操作控制逻辑。同步时序逻辑是设计复杂的数字逻辑系统的核心。时序逻辑借助于状态寄存器记住它目前所处的状态。在不同的状态下，即使所有的输入都相同，其输出也不一定相同。

【组合逻辑 1】一个八位数据通路控制器。

Verilog HDL 描述如下：

```
'define ON 1'b1
#define OFF 1'b0
wire ControlSwitch;
wire [7:0] Out, In;
assign Out = (ControlSwitch == 'ON) ? In : 8'h00;
```

数据通道开关的逻辑电路结构如图 11.1(a)所示。由数据通路所产生的波形如图 11.1(b)所示。



(a) 数据通道开关的逻辑图

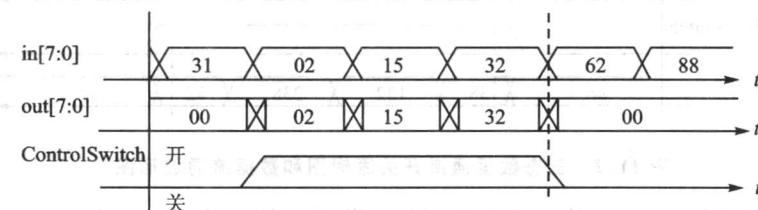


图 11.1 数据通道开关的逻辑图与波形图

【组合逻辑 2】一个八位三态数据通路控制器。

Verilog HDL 描述如下：

```
'define ON 1'b1
#define OFF 1'b0
wire LinkBusSwitch;
wire [7:0] outbuf;
reg [7:0] inbuf;
inout [7:0] bus;
assign bus = (LinkBusSwitch == 'ON) ? outbuf : 8'hzz;
always @ (posedge clk)
begin
    if (LinkBusSwitch)
        bus = outbuf;
    else
        bus = 8'hzz;
end
```

```

if (!LinkBusSwitch)
    inbuf <= bus;
:
end
:

```

八位三态数据通路控制器的逻辑电路结构和由此产生的波形如图 11.2 所示。

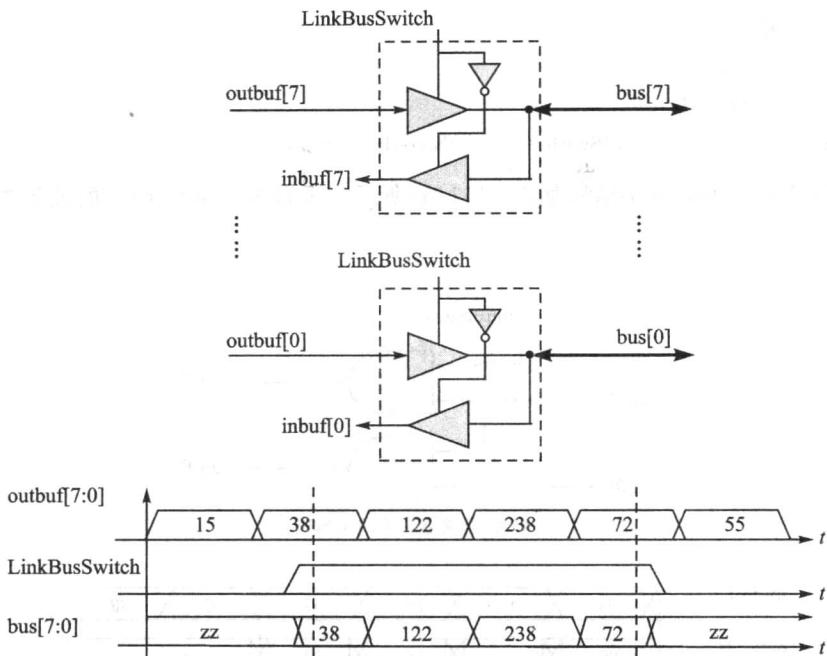


图 11.2 三态数据通道开关逻辑图和数据流通波形图

它与[组合逻辑例 1]的差别只是前者在开关断开时输出为零,而后者在开关断开时输出为高阻,即与总线脱离连接,不再有信号输出。从此时起,如果总线的另一端有驱动源的话,总线可以作为模块的输入信号线,把数据从外部送到模块内部的寄存器组(如图中 inbuf[7:0]引脚所示)。

11.2 数据在寄存器中的暂时保存

组合逻辑电路的输出与每个输入信号的电平直接相关。从严格的意义上讲,它的输出在每一个瞬间都有可能发生变化,而同步时序逻辑的输出只有在时钟跳变沿时刻才有可能变化。由于逻辑门和布线都有延迟,因此没有办法使实际电路的输出与理想的布尔方程计算完全一致。可以说,实际组合逻辑电路输出的瞬间不确定性是无法避免的。如果能使组合逻辑电路的输入稳定一段时间,即所有的输入信号在一段相对较长的时间段里不再发生变化,虽然在稳定时间片段的刚一开始由于冒险竞争现象会产生与理想情况不一致的毛刺或输出不确定的情况,但只要稳定时间片段大于最长的路径延迟,就可以取得组合逻辑电路的理想输出。如果能躲开输出不确定片段,在理想值稳定输出的片刻把该输出值存入寄存器组,则寄存器组中保留

的就是该组合逻辑电路的理想输出。如果不是有意地改变寄存器组的值,那么该值可以一直保留下去,直到改变寄存器组中保留的数值。可以把寄存器组中保留的数值作为下一级电路的输入,根据需要维持一定长度的时间片段,再作改变,以保证下一级组合电路有稳定的输入。

在上面一段中已经讲到用寄存器组把理想的输出保留下来,待改变的时候再用新的数值来替换它,这种电路在数字系统中得到了广泛应用,它是数字电路模块组成的重要部件之一。图 11.3 和图 11.4 分别为带使能端及复位端的时钟同步 8 位寄存器组逻辑和模块接口图,并配有 Verilog 模块的程序,以加深理解。

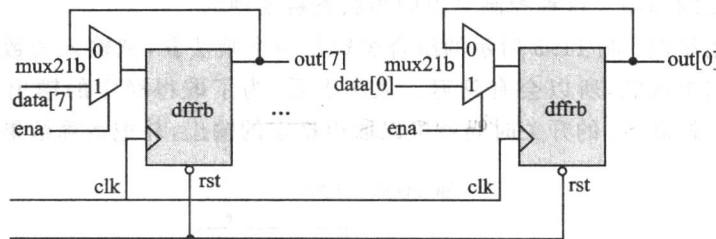


图 11.3 带使能端和复位端的时钟同步 8 位寄存器组逻辑

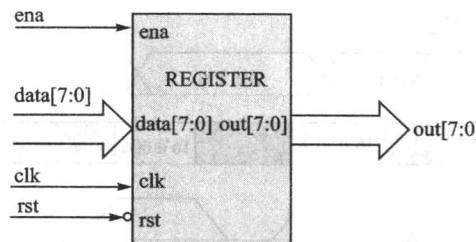


图 11.4 带使能端和复位端的时钟同步 8 位寄存器组模块接口图

模块 register8 是可以综合的,综合出来的电路逻辑如图 11.3 所示。只有在使能控制信号 ena 为高电平时才有可能在时钟正跳变沿时刻把新的值作为模块的输出;否则即使时钟正跳变不断发生,输入数据 data 也在不断变化,而 out 仍然保持原来的值。所以,可以通过控制 ena 信号的开/关(高电平/低电平)时刻,在数据通道上不断变化着的数据流中,选取有意义的数据保存在寄存器组中。以下是图 11.3 和图 11.4 的 Verilog 模块程序。

```
module register8(ena,clk,data,rst,out);
    input ena,clk,rst;
    input [7:0] data;
    output [7:0] out;
    reg [7:0] out;
    always @(posedge clk)
        if (!rst)
            out <= 0;
        else if (ena)
            out <= data;
    //虽然没有写 else 项,显然,如果 ena 为低电平,即使时钟变化,data 变化,但 out 仍保持不变
endmodule
```

11.3 数据流动的控制

我们知道,诸如加、减、乘、除、比较等运算都可以用组合逻辑来实现,但运算的输入必须有一段稳定的时间,才可能得到稳定的输出;而输出要被下一阶段的运算作为输入,也必须要有一段时间的稳定,因而输出结果必须保存在寄存器组中。在计算电路中设有许多寄存器组,它们是用来暂存运算的中间数据。对寄存器组之间数据流动进行精确的控制,在算法的实现过程中有着极其重要的作用。这种控制是由同步状态机实现的。

开关逻辑应用举例:图 11.5 所示的组合逻辑是一个乘法器,把输入的数乘 3,然后输出。因为乘法器是由门组成的,所以会有延迟。从图上看,为了取得稳定的输出需要 10 ns 的延迟。如果能有效地控制 S_n 的开关时间就可以取得稳定的输出,并把运算结果存入寄存器。

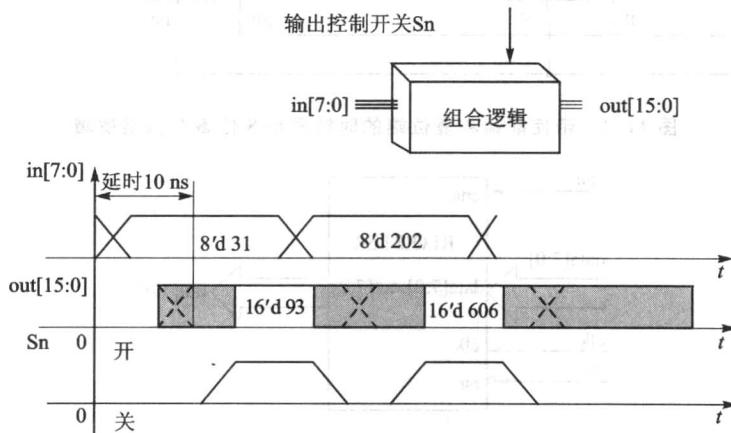


图 11.5 带输出控制开关的运算组合逻辑和数据流波形

如图 11.6 所示,由开关 S_1 和开关 S_2 控制的两个组合逻辑都是运算逻辑,例如乘法器或加法器等,而寄存器 A,B,C 是用来寄存运算的输入、中间和输出数据的。如果能与时钟配合来精确地控制开关的闭合和断开,在寄存器中暂存的中间或输出数据都会是上一步运算的稳定结果,而不会出现冒险和竞争的现象。

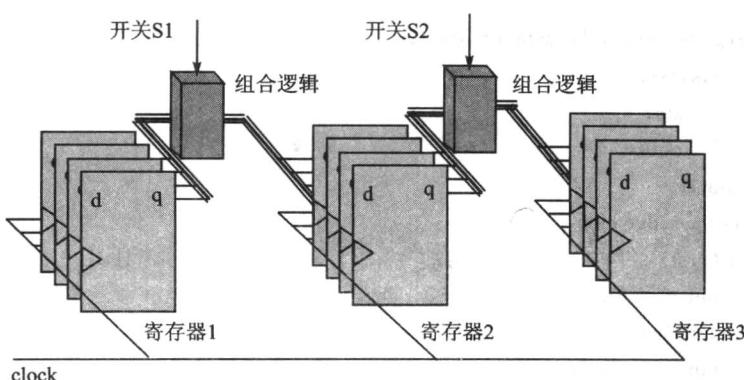


图 11.6 把组合逻辑运算的结果储存在寄存器中作为下一级的输入

图 11.7 中,由开关 S1,S3,S5 控制的三个组合逻辑都是运算逻辑,例如乘法器或加法器等,而寄存器组 A,B,C 是用来寄存运算的输入、中间和输出数据的。开关 S2,S4,S6 是三态门,能控制寄存器组 A,B,C 的输出到总线上还是与总线隔离。如果能与时钟配合来精确地控制 S1~S6 开关的闭合和断开,在寄存器中暂存的中间或输出数据都会是上一步运算的稳定结果,而不会出现冒险和竞争的现象。运算的过程可以在这几个寄存器组内反复地执行,直到通过开关的控制使其停止。下面通过简单的描述来说明一个极其重要的概念:生成与时钟精确配合的开关时序是计算逻辑的核心。

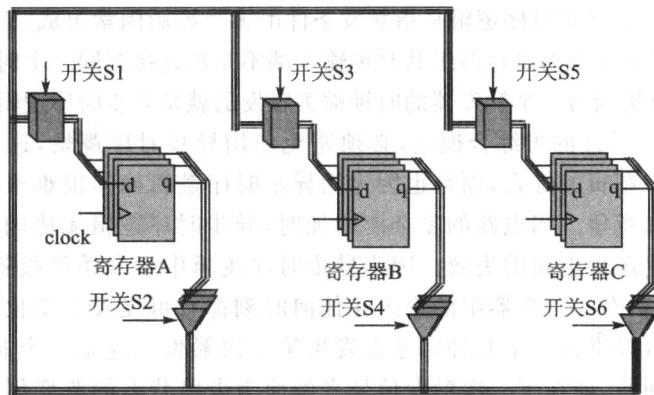


图 11.7 由开关逻辑控制的数据流动和计算逻辑结构示意图

在《数字电子技术基础》中已经知道:当时钟正跳变沿到来时,在 D 触发器数据端口的数据才能存入触发器中;当组合逻辑的输入变化时,输出必须经过一段时间后才能稳定,这是由于门级电路和布线的延迟造成的。只有稳定的输出对运算才是有意义的。如果想把寄存器组 C 中的数据经过组合逻辑的运算存入寄存器组 A 中,应该如何来控制这几个开关呢?从上面的描述知道,开关 S1,S3,S5 分别控制着 3 个组合运算逻辑的输出。如果把 S1,S3,S5 看成是 11.1.3 节中介绍的 3 个独立的寄存器组的使能控制信号 ena 都为低电平,则 A,B,C 3 个寄存器组的输出保持不变。当时钟正跳变沿到来时,这 3 个寄存器组中的值也不会改变。寄存器组 C 中保持的数据是上次 S5 为高电平时,时钟正跳变沿时存入的数据。如果在上一个正跳变沿到来前已经接通 S6,等到下一个时钟正跳变沿到来时接通 S1;在时钟正跳变沿到来时存入寄存器组 A 中的是上一个时钟寄存器组 C 中保存的数据和组合逻辑 A 的运算结果。这时开关 S3,S5,S2,S4 必须保持低电平,把数据通道断开。由原来保存在寄存器组 C 中的数据和组合逻辑 A 所生成的结果已稳定地存入寄存器组 A 中。同理断开所有的通道,只按时序先后接通 S2,S3,其他开关保持低电平,在下一运算时钟正跳变沿到来时就能稳定地把寄存器组 A 中的数据与组合逻辑 B 的运算结果存入寄存器组 B。这个简单的例子说明:如果能设计出一个状态机,在这个状态机的控制下生成一系列的开关信号,严格按时钟的节拍来开启或关闭数据通道,就能用硬件来构成复杂的计算逻辑。如果硬件的规模可以达到几十到几千万门,就可以设计出并行度很高的高速计算逻辑。在下一章里我们将详细地介绍怎样用 Verilog HDL 来编写可综合的复杂同步状态机。

11.4 在 Verilog HDL 设计中启用同步时序逻辑

同步时序逻辑是指表示状态的寄存器组的值只可能在唯一确定的触发条件发生时刻改变，只能由时钟的正跳沿或负跳沿触发的状态机就是一例。always @ (posedge clock)就是一个同步时序逻辑的触发条件，表示由该 always 控制的 begin end 块中寄存器变量重新赋值的情况只能在 clock 正跳沿才发生。而异步时序逻辑是指触发条件由多个控制因素组成，任何一个因素的跳变都可以引起触发。记录状态的寄存器组其时钟输入端不是都连接在同一个时钟信号上。例如用一个触发器的输出连接到另一个触发器的时钟端去触发的就是异步时序逻辑。

用 Verilog HDL 设计的可综合模块，必须避免使用异步时序逻辑，这不但是因为许多综合器不支持异步时序逻辑的综合，而且也因为用异步时序逻辑确实很难来控制由组合逻辑和延迟所产生的冒险和竞争。当电路的复杂度增加时，异步时序逻辑无法调试。工艺的细微变化也会造成异步时序逻辑电路的失效。因为异步时序逻辑中触发条件很随意，任何时刻都有可能发生，所以记录状态的寄存器组的输出在任何时刻都有可能发生变化。而同步时序逻辑中的触发输入至少可以维持一个时钟后才会发生第二次触发。这是一个非常重要的差别，因为可以利用这一时间段，即在下一次触发信号到来前为电路状态的改变创造一个稳定可靠的条件。由此可以得出结论：同步时序逻辑比异步时序逻辑具有更可靠、更简单的逻辑关系。如果我们强行作出规定，用 Verilog 来设计可综合的状态机必须使用同步时序逻辑，有了这个前提条件，实现自动生成电路结构的综合器就有了可能。因为这样做大大减少了综合工具的复杂度，为这种工具的成熟创造了条件，也为 Verilog 可综合代码在各种工艺和 FPGA 之间移植创造了条件。Verilog RTL 级的综合就是基于这个规定的。

在同步逻辑电路中，触发信号是时钟的正跳沿（或负跳沿），触发器的输入与输出是由两个时钟来完成的。第一个时钟的正跳沿（或负跳沿）为输入作准备，在第一个时钟正跳沿（或负跳沿）起直到第二个时钟正跳沿（或负跳沿）到来之前的这一段时间内，有足够的时间使输入稳定。当第二个时钟正跳沿（或负跳沿）到来时，由前一个时钟沿创造的条件已经稳定，所以能够使下一个状态正确地输出。

若在同一个时钟的正跳沿（或负跳沿）下对寄存器组既进行输入又进行输出，很有可能由于门的延迟使输入条件还未确定时，就输出了下一个状态，这种情况会导致逻辑的紊乱。而利用上一个时钟为下一个时钟创造触发条件的方式是安全可靠的。但这种工作方式需要有一个前提：确定下一个状态所使用的组合电路的延迟与时钟到各触发器的差值必须小于一个时钟周期的宽度。只有满足这一前提才可以避免逻辑紊乱。在实际电路的实现中，采取了许多有效的措施来确保这一条件的成立，其中主要有以下几点：

- (1) 全局时钟网络布线时尽量使各分支的时钟一致。
- (2) 采用平衡树结构，在每一级加入缓冲器，使到达每个触发器时钟端的时钟同步，如图 11.8 和 11.9 所示。

通过这些措施基本可以保证时钟的同步。在后仿真时，若逻辑与预期设计的不一样，可降低时钟频率，这就有可能消除由于时钟过快引起的触发器输入端由延迟和冒险竞争造成的不稳定，从而使逻辑正确。

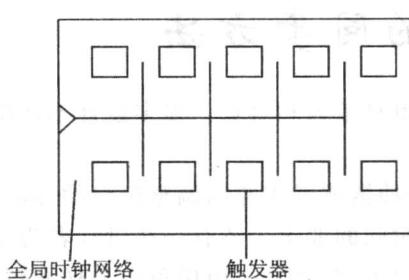


图 11.8 全局时钟网示意图

在组合逻辑电路中,多路信号的输入使各信号在同时变化时很容易产生竞争冒险,从而结果难以预料。图 11.10 是一个简单的组合逻辑的例子:
 $c = a \& b$ 。

a 和 b 变化不同步使 c 产生了一个脉冲。这个结果也许与当初设计时的想法并不一致,但如果能过一段时间,待 c 的值稳定后再来取用组合逻辑的运算结果,就可以避免竞争冒险。

同步时序逻辑由于用上一个时钟的跳变沿时刻(置寄存器作为组合逻辑的输入)来为下一个时钟的跳变沿时刻的置数(置下一级寄存器作为该组合逻辑的输出)做准备,只要时钟周期足够长,就可以在下一个时钟的跳变沿时刻得到稳定的置数条件,从而在寄存器组中存入可靠的数据。而这一点用异步电路是做不到的,因此在实际设计中应尽量避免使用异步时序逻辑。若用修补的方法来避免竞争冒险,所耗费的人力物力是很巨大的。也无法使所设计的 Verilog HDL 代码和已通过仿真测试的电路模块结构有知识产权的可能,因为工艺的细微改变就有可能使电路无法正常工作。显而易见,使用异步时序逻辑会带来设计的隐患,无法设计出能严格按同一时间节拍操作控制数据流动方向开关的状态机。而这种能按时钟节拍精确控制数据流动开关的状态机就是在下一章里将详细介绍的同步有限状态机。它是算法计算过程中数据流动控制的核心。计算结构的合理配置和运算效率的提高与算法状态机的设计有着非常密切的关系。只有通过阅读有关计算机体系结构的资料和通过大量的设计实践才能熟练地掌握复杂算法系统的设计。

在 Verilog 语法中有一种非阻塞赋值方式,用“ $<=$ ”符号表示。它的含义是:如果在 begin end 块中同时有许多个非阻塞赋值,则它们的赋值顺序是同时的,并不是按先后次序赋值。实际上它们表示的是同时赋入上一个时钟沿时刻送入寄存器的值。这与使用同一时钟沿触发的许多寄存器在同一个使能控制信号下赋值是完全一致的。所有被赋的值在上一个时钟沿前就已经保存在寄存器中,它们有足够的时问传送到被赋值的寄存器的数据端口。当时钟沿到来时被赋值都已经稳定,所以存入的寄存器的数值是可靠的。用这种方法可以避免由组合逻辑产生的冒险与竞争。

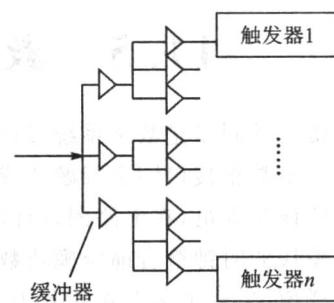


图 11.9 平衡树结构示意图

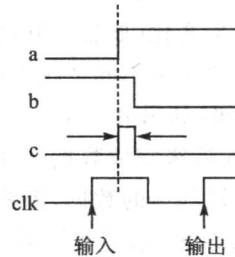


图 11.10 由于 a,b 变化不同步导致组合电路竞争冒险产生毛刺和防止办法

11.5 数据接口的同步方法

数据接口的同步是数字系统设计中常见的问题,也是重点和难点。很多设计不能稳定地工作往往是由数据接口同步问题造成的。

有一些设计人员,在电路图设计阶段,采用加入缓冲器或者用非门调整延迟的办法,从而保证本级模块的时钟符合前级模块数据的建立、保持时间的要求。还有一些设计者为了取得稳定的采样值,生成了多个相位差为 90° 的时钟信号,时而用正沿,时而用负沿采样取数据,用以调整数据的采样位置,这两种做法都是不可取的。一旦芯片更新换代,或者移植到其他系列的器件芯片上,采用这种方法设计的采样电路必须重新设计,因为电路不能稳定地工作,一旦外界条件发生变化(如温度升高),采样时序就有可能完全发生混乱,造成电路故障。

下面介绍3种不同情况下的数据接口的简单同步方法:

(1) 前级(如另外一个芯片、PCB布线、驱动接口元件)输出的延时是随机的,或者有可能变动,如何在后级完成数据的同步?

对于随机到达的数据,需要建立同步机制。可以采用使数据通过RAM或者FIFO的缓存再读取的方法,达到数据同步的目的。将前级芯片提供的时钟作为基本时钟,将数据写入RAM或者FIFO,然后使用后级的基本时钟产生读信号,将数据读出来即可。这种做法的关键是必须要有堆栈满和空的指示信号来管理数据的写入和读取,以防止数据的丢失。在FPGA中一般都提供参数化宏模块,如FIFO、双口RAM,并有许多种配置可供选择,可以用来作为数据的同步之用,有关FIFO参数化模块的使用我们将在高级教程中讲解。

(2) 数据有固定的帧格式,数据的起始位置如何确定?

通信系统中,数据往往是按照帧组织的。由于系统对时钟的要求很高,常常设计专门时钟板产生高精度的时钟。数据帧是有起始位置的,在数据正确接收之前,必须先完成数据的同步,即确定数据的“头”是从什么地方开始的。数据同步采用的就是这种方法,即用同步头表示数据信号的起始,或者使用双口RAM、FIFO来缓存数据再传送到下一级。找到数据头的方法有两种:第一种,增加一条表示数据起始位置的信号线;第二种,对于异步系统,则常常在数据中插入一段有特殊码型的同步码(同步头),接收端通过相关运算检测到同步头。

(3) 级联的两个模块的基本时钟是异步时钟域的,如何把前级输出的数据准确地传送到下一级模块中?

如果输入数据的节拍和本级芯片的处理时钟同频,可以直接用本级芯片的主时钟对输入数据寄存器采样,完成输入数据的同步;如果输入数据和本级芯片的处理时钟是异步的,特别当两个时钟的频率不是由同一石英晶体分频产生的,则起码对输入数据做两次以上的采样寄存,才能初步完成输入数据的读入。需要说明的是对异步时钟域的数据进行两次以上采样,其作用只是防止了数据状态不稳定的传播,使后级电路处理的数据都是有效电平。但是这种做法并不能保证两级寄存器采样后的数据是完全正确的电平,这种方式处理一般都会读入一定数量的错误数据,所以只适用于允许发生少量错误的电路功能单元。

为了避免由异步时钟域产生的错误,经常使用双口RAM(DPRAM)、FIFO缓存的方法完成异步时钟域之间的数据传送。在输入端口使用前级时钟写数据,在输出端口使用本级时钟读数据,并有缓冲器空或满的控制信号来管理数据的读写,以避免数据的丢失,可以非常方便

准确地完成异步时钟域之间的数据交换。为了解决数据接口的同步时发生的问题通过实际例子说明如何设计这样的接口。

小 结

复杂数字系统是由组合逻辑部件和时序逻辑电路部件连接组成的，组合逻辑的输出必须暂时存在寄存器中，组合逻辑输出的数据存入寄存器的时机必须合适，必须注意防止把组合逻辑中由于冒险竞争产生的不稳定信号值存入寄存器。这会严重影响以后的逻辑操作和处理。为了保证逻辑设计非常可靠，并可以在不同工艺的器件中实现，所以必须采用可靠的同步设计方法，即芯片电路中必须有一个全局的时钟，并（通过控制电路芯片制造工艺实现）使得到达芯片中每个触发器的时钟端的时钟沿偏差（歪斜）非常小，这样设计者就可以通过控制使能端和时钟沿的配合，躲避冒险竞争现象，把理想的稳定的逻辑数据存入寄存器，供下一步逻辑操作或运算之用。产生这种可靠的以同步时钟为基准的产生多个使能控制信号的电路就是下面第12章要讲解的同步状态机。外来的异步信号若要高度可靠地引入芯片电路，必须符合一定的要求，并必须经过认真的同步处理，否则很容易产生电路设计隐患，系统设计工作者必须格外小心谨慎，严格测试，以避免出现此类情况。

思 考 题

- 利用数字电路的基本知识解释，为什么说即使组合逻辑的输入端的所有信号同时变化，其输出端的各个信号不可能同时达到新的值？各个信号变化的快慢由什么决定？
- 如果组合逻辑的输入端信号变化非常快，其输出端的逻辑关系能否正确？变化快到什么程度以后，就没有正确的输出？如果还有正确输出，但时间片段很小，有什么办法可以加长正确输出的时间片？
- 为使运算组合逻辑有一个确定的输出，为什么必须在复杂运算组合逻辑的输入端和输出端增加寄存器组来寄存数据？
- 对每一个寄存器组来说，上一个时钟的正跳沿是为置数做准备，下一个时钟正跳沿是把本寄存器组置数（并为下一级运算组合逻辑送去输入信号），则为下一级寄存器组的置数做准备的先决条件是什么？
- Verilog语法中使用了哪一种赋值符号可以表示与硬件寄存器组实现完全一致的赋值方式？
- 一个带使能端的寄存器组能被赋入一个正确的输入值需要哪3个条件？
- 为什么建议大家采用同步时序逻辑来设计数字逻辑电路，异步逻辑有什么不好？
- 简单叙述不同时钟域模块之间数据准确传送的方法。

第 12 章 同步状态机的原理、结构和设计

概 述

由于 Verilog HDL 和 VHDL 行为描述用于综合的历史还只有近 20 年的历史, 可综合风格的 Verilog HDL 和 VHDL 的语法只是它们各自语言的一个子集。又由于 HDL 的可综合性研究近年来非常活跃, 但可综合子集的国际标准目前尚未最后形成, 因此各厂商的综合器所支持的 HDL 子集也略有不同。本教材中有关可综合风格的 Verilog HDL 的内容, 只着重介绍 RTL 级、算法级和门级逻辑结构的描述; 而系统级(数据流级)的综合由于还不太成熟, 暂不作介绍。由于寄存器传输级(RTL)描述的是以时序逻辑抽象所得到的有限状态机为依据, 所以, 把一个时序逻辑抽象成一个同步有限状态机是设计可综合风格的 Verilog HDL 模块的关键。在本章中, 我们将在了解状态机结构的基础上通过各种实例, 由浅入深地介绍各种可综合风格的 Verilog HDL 模块, 并把重点放在时序逻辑的可综合有限状态机的 Verilog HDL 设计要点。至于组合逻辑, 因为比较简单, 只需阅读典型的用 Verilog HDL 描述的可综合的组合逻辑的例子就可以掌握。为了更好地掌握可综合风格, 还需要较深入地了解阻塞和非阻塞赋值的差别, 以及在不同的情况下正确使用这两种赋值的方法。只有深入地理解阻塞和非阻塞赋值语句的细微不同, 才有可能写出不仅可以仿真也可以综合的 Verilog HDL 模块。只要按照一定的原则来编写代码就可以保证 Verilog 模块综合前和综合后仿真的一致性。符合这样条件的可综合模块是用之设计的目标, 因为这种代码是可移植的, 可综合到不同的 FPGA 和不同工艺的 ASIC 中, 是一种具有知识产权价值的软核。

12.1 状态机的结构

图 12.1 表示的是数字电路设计中常用的时钟同步状态机的结构。其中状态寄存器是由一组触发器组成, 用来记忆状态机当前所处的状态。如果状态寄存器由 n 个触发器组成, 这个状态机最多可以记忆 2^n 个状态。所有的触发器的时钟端都连接在一个共同的时钟信号上, 所以状态的改变只可能发生在时钟的跳变沿上。可能发生的状态的改变由正跳变还是由负跳变触发, 取决于触发器的类型。状态是否改变、怎样改变还将取决于产生下一状态的组合逻辑 F 的输出, F 是当前状态和输入信号的函数。状态机的输出是由输出组合逻辑 G 提供的, G 也是当前状态和输入信号的函数。现代电路设计常用正跳变沿触发的 D 触发器, 特别是在可编程逻辑器件上实现的用综合工具自动生成的状态机, 其电路结构往往都是使用正跳变沿触发的 D 触发器。目前的 JK 触发器、其他类型的触发器和锁存器已经很少使用。图中的 F 和 G 两部分都是纯组合逻辑, 它们的逻辑函数表达式如下:

$$\text{下一个状态} = F(\text{当前状态}, \text{输入信号});$$

$$\text{输出信号} = G(\text{当前状态}, \text{输入信号});$$

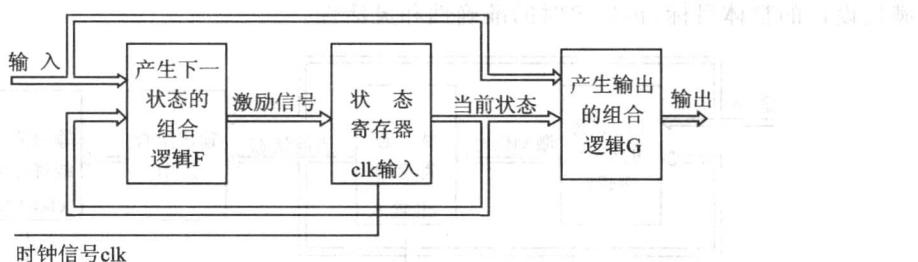


图 12.1 时钟同步的状态机结构(Mealy 状态机)

12.2 Mealy 状态机和 Moore 状态机的不同点

如果时序逻辑的输出不但取决于状态还取决于输入(见图 12.1),称为 Mealy 状态机。而有些时序逻辑电路的输出只取决于当前状态,即输出信号 = G(当前状态),这样的电路就称为 Moore 状态机,它的电路结构如图 12.2 所示。

很明显,这两种电路结构除了在输出电路部分有些不同外,其他地方都是相同的。在实际设计工作中,其实大部分状态机都属于 Mealy 状态机,因为状态机的输出中或多或少有几个属于 Mealy 类型的输出,输出不但与当前状态有关还与输入有关;还有几个输出属于 Moore 类型的,只与当前的状态有关。

在设计高速电路时,常常有必要使状态机的输出与时钟几乎完全同步。有一个办法是把状态变量直接用作输出,为此在指定状态编码时需要多费一些脑力,也可能会多用几个寄存器。这种设计思路,在高速状态机电路时常常使用,这称为输出编码的状态指定。这种状态机也属于图 12.2 所示的 Moore 状态机,但其输出组合逻辑部分只有连线,没有其他组合逻辑部件,详见[例 12.3]描述的状态机。

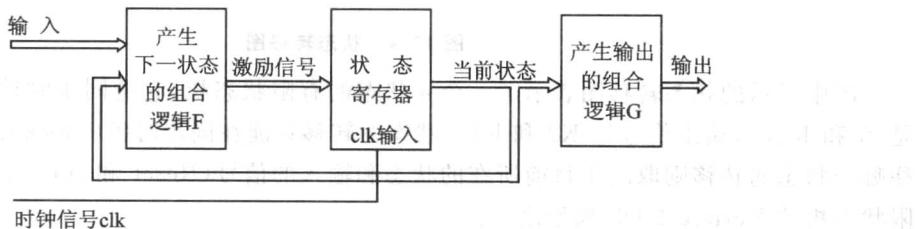


图 12.2 时钟同步的状态机结构(Moore 状态机)

设计高速状态机还有一种办法,即如图 12.3 所示,在输出逻辑 G 后面再加一组与时钟同步的寄存器输出流水线寄存器,让 G 所有的输出信号在下一个时钟跳变沿时同时存入寄存器组,即完全同步地输出,把这种输出称为流水线化的输出(pipelined outputs)。

其实这几种状态机之间,只要做一些改变,便可以从一种形式转变为另一种形式。例如将图 12.3 所示的状态机中产生流水输出的寄存器省去,把这些寄存器用在状态记忆上,就可以很容易地得到一个把状态变量用作输出信号的 Moore 状态机,详见[例 12.3]描述的状态机。

把状态机精确地分为这类或那类,其实并不重要,重要的是设计者如何把握输出的结构能

满足设计的整体目标,包括定时的准确性和灵活性。

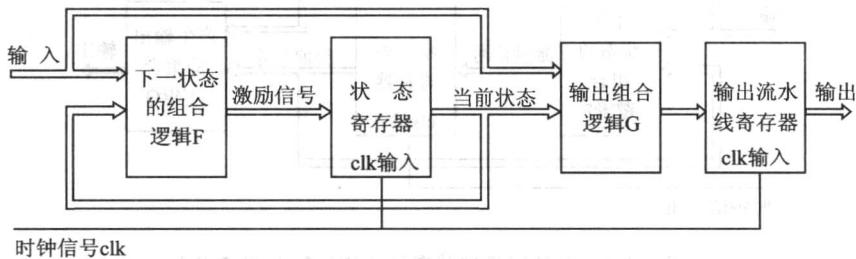


图 12.3 带流水线输出的 Mealy 状态机

12.3 如何用 Verilog 来描述可综合的状态机

在 Verilog HDL 中可以用许多种方法来描述有限状态机,最常用的方法是用 always 语句和 case 语句。图 12.4 所示的状态转移图表示了一个简单的有限状态机,[例 12.1]的程序就是该有限状态机的多种 Verilog HDL 模型之一。

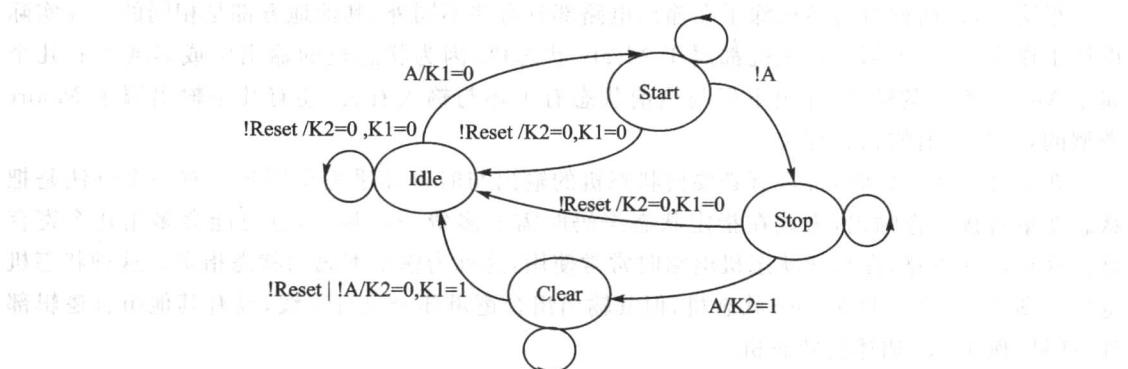


图 12.4 状态转移图

图中所示的状态转移图表示了一个 4 状态的有限状态机,它的同步时钟是 clk,输入信号是 A 和 Reset,输出信号是 K2 和 K1。状态的转移只能在同步时钟(Clock)的上升沿时发生,往哪个状态的转移则取决于目前所在的状态和输入的信号(Reset 和 A)。下面的例子是该有限状态机的 Verilog HDL 模型之一。

12.3.1 用可综合 Verilog 模块设计状态机的典型办法

【例 12.1】

```
module fsm (Clock, Reset, A, K2, K1, state);
    input Clock, Reset, A;
    output K2, K1;
    output [1:0]state;
    reg K2, K1;
    reg [1:0] state;
```

```
parameter Idle = 2'b00,
          Start = 2'b01;
          Stop = 2'b10;
          Clear = 2'b11;

always @(posedge Clock)
if (! Reset)
begin
    state <= Idle;
    K2<=0;
    K1<=0;
end
else
case (state)
    Idle: if (A) begin
        state<=Start;
        K1<=0;
    end
    else begin
        state<=Idle;
        K2<=0;
        K1<=0;
    end
    Start: if (! A) state<=Stop;
            else state<=Start;
    Stop: if (A) begin
        state<=Clear;
        K2<=1;
    end
    else begin
        state<=Stop;
        K2<=0;
        K1<=0;
    end
    Clear: if (! A) begin
        state <=Idle;
        K2<=0;
        K1<=1;
    end
    else begin
        state<=Clear;
    end
endcase
end
```

```

        K2<=0;
        K1<=1;
    end
default: state<=2'bxx,
endcase
endmodule

```

我们还可以用另一个 Verilog HDL 模型来表示同一个有限状态,见下例。

12.3.2 用可综合的 Verilog 模块设计、用独热码表示状态的状态机

【例 12.2】

```

module fsm (Clock, Reset, A, K2, K1);
    input Clock, Reset, A;
    output K2, K1;
    reg K2, K1;
    reg [3:0] state;

parameter Idle = 4'b1000,
          Start = 4'b0100,
          Stop = 4'b0010,
          Clear = 4'b0001;

always @ (posedge clock)
if (! Reset)
begin
    state <= Idle;
    K2<=0;
    K1<=0;
end
else
case (state)
    Idle: if (A) begin
            state <= Start;
            K1<=0;
        end
        else begin
            state<=Idle;
            K2<=0;
            K1<=1;
        end
    Start: if (! A) state<=Stop;
            else state<=Start;
    Stop: if (A) begin

```

```

state<=Clear;
K2<=1;
endcase
else begin
    state<=Stop;
    K2<=0;
    K1<=0;
end
Clear: if (! A) begin
    state<=Idle;
    K2<=0;
    K1<=1;
end
else begin
    state<=Clear;
    K2<=0;
    K1<=0;
end
default: state<=Idle;
endcase
endmodule

```

[例 12.2]与[例 12.1]的主要不同点是状态编码,[例 12.2]采用了独热编码,而[例 12.1]则采用 Gray 编码,究竟采用哪一种编码好要看具体情况而定。对于用 FPGA 实现的有限状态机建议采用独热码,因为虽然独热编码多用了两个触发器,但所用组合电路可省一些,因而使电路的速度和可靠性有显著提高,而总的单元数并无显著增加。采用独热编码后有了多余的状态,就有一些不可到达的状态。为此,在 case 语句的最后需要增加 default 分支项。这可以用默认项表示该项,也可以用确定项表示,以确保回到 Idle 状态。一般综合器都可以通过综合指令的控制来合理地处理默认项。

12.3.3 用可综合的 Verilog 模块设计、由输出指定的码表示状态的状态机

[例 12.3]采用了另一种方法,即在前面状态机结构部分讲过的,把输出直接指定为状态码。也就是把状态码的指定与状态机控制的输出联系起来,把状态的变化直接用作输出,这样做可以提高输出信号的开关速度并节省电路器件。但这种方法也有缺点,就是开关的维持时间必须与状态维持的时间一致,如果要完全实现上面两例子的开关输出波形,需要增加状态才能实现。这种设计方法常用在高速状态机中,建议大家在设计高速状态机时采用[例 12.3]的风格。例中 state[4] 和 state[0]分别表示前面两个例子中的输出 K2 和 K1。

【例 12.3】

```

module fsm (Clock, Reset, A, K2, K1, state);
input Clock, Reset, A;
output K2, K1;

```

```

output[4:0]state;
reg[4:0]state;
assign K2= state[4];           //把状态变量的最高位用作输出 K2
assign K1= state[0];           //把状态变量的最低位用作输出 K1
parameter
//-----      K2_i_j_K1 ----- output coded state assignment -----
Idle      = 5'b0_0_0_0_0,
Start     = 5'b0_0_0_1_0,
Stop      = 5'b0_0_1_0_0,
StopToClear = 5'b1_1_0_0_0,
Clear     = 5'b0_1_0_1_0,
ClearToIdle = 5'b0_0_1_1_1;

always @ (posedge Clock)
if (! Reset)
begin
    state<=Zero;
end
else
case (state)
Idle: if (A)
        state<=Start;
    else state<=Idle;
Start: if(! A) state<=Stop;
    else state<=Start;

Stop:  if (A)
        state<=StopToClear;
    else state<=Stop;

StopToClear: state<=Clear;
Clear:  if(! A)
        state<=ClearToIdle;
    else state<=Clear;

ClearToIdle: state<=Idle;

default: state<=Idle;
endcase
endmodule

```

12.3.4 用可综合的Verilog模块设计复杂的多输出状态机时常用的方法

在比较复杂的状态机设计过程中,往往把状态的变化与输出开关的控制分成两部分来考虑,就像前面讲过的Mealy状态机输出部分的组合逻辑。为了调试方便,还常常把每一个输出开关写成一个个独立的always组合块。在调试多输出状态机时,这样做比较容易发现问题和改正模块编写中出现的问题。建议大家在设计复杂的多输出状态机时采用[例12.4]的风格举例,说明如下。

【例12.4】

```

module fsm (Clock, Reset, A, K2, K1);
    input Clock, Reset, A;
    output K2, K1;
    reg K2, K1;
    reg [1:0] state, nextstate;

    parameter
        Idle=2'b00,
        Start=2'b01,
        Stop=2'b10,
        Clear=2'b11;
    //-----每一个时钟沿产生一次可能的状态变化-----
    always @(posedge Clock)
        if (!Reset) state <= Idle;
        else state <= nextstate;
    //-----产生下一状态的组合逻辑-----
    always @(state or A)
        case (state)
            Idle: if (A) nextstate = Start;
                  else nextstate = Idle;
            Start: if (!A) nextstate = Stop;
                    else nextstate = Start;
            Stop: if (A) nextstate = Clear;
                  else nextstate = Stop;
            Clear: if (!A) nextstate = Idle;
                    else nextstate = Clear;
            default: nextstate = 2'bxx;
        endcase
    //-----产生输出 K1 的组合逻辑-----

```

```

always @ (state or Reset or A)
    if (!Reset) K1=0;
    else
        if (state == Clear && !A) //从 Clear 转向 Idle
            K1=1;
        else K1 = 0;

//-----产生输出 K2 的组合逻辑 -----
always @ (state or Reset or A )
    if (!Reset) K2 = 0;
    else
        if (state == Stop && A) // 从 Stop 转向 Clear
            K2 = 1;
        else K2 = 0;
//-----

endmodule

```

这种风格的描述比较适合大型的状态机,查错和修改比较容易。Synopsys 公司的综合器建议使用这种风格来描述状态机。

上面 4 个例子是同一个状态机的 4 种不同的 Verilog HDL 模型,它们都是可综合的,在设计复杂程度不同的状态机时有它们各自的优势。如用不同的综合器对这 4 个例子进行综合,综合出的逻辑电路可能会有些不同,但逻辑功能是相同的。下面列出测试不同风格状态机测试模块供大家参考。

测试模块:

```

`timescale 1ns/1ns
module t;
    reg a;
    reg clock, rst;
    wire k2, k1;
initial                               // initial 常用于仿真时信号的给出
begin
    a = 0;
    rst = 1;                         // 给复位信号变量赋初始值
    clock = 0;                        // 给时钟变量赋初始值
    #22 rst = 0;                     // 使复位信号有效
    #133 rst = 1;                    // 经过一个多周期后使复位信号无效
end

always #50 clock = ~clock;           // 产生周期性的时钟
always @ (posedge clock)           // 在每次时钟正跳变沿时刻产生不同的 a
begin

```

```

#30 a = { $random }%2; // 每次 a 是 0 还是 1 是随机的
#(3 * 50 + 12); // a 的值维持一段时间
end
initial
begin # 100000 $stop; end // 系统任务,暂停仿真以便观察仿真波形
//----- 调用被测试模块 t.m -----
fsm m(.Clock(clock), .Reset(rst), .A(a), .K2(k2), .K1(k1));
endmodule

```

下面总结了有限状态机设计的一般步骤,供大家参考。

小 结

有限状态机设计的一般步骤:

(1) 逻辑抽象,得出状态转换图 就是把给出的一个实际逻辑关系表示为时序逻辑函数,可以用状态转换表来描述,也可以用状态转换图来描述。这就需要:

① 分析给定的逻辑问题,确定输入变量、输出变量以及电路的状态数。通常是取原因(或条件)作为输入变量,取结果作为输出变量。

② 定义输入、输出逻辑状态的含意,并将电路状态顺序编号。

③ 按照要求列出电路的状态转换表或画出状态转换图。

这样,就把给定的逻辑问题抽象到一个时序逻辑函数了。

(2) 状态化简 如果在状态转换图中出现这样两个状态,它们在相同的输入下转换到同一状态去,并得到一样的输出,则称为等价状态。显然等价状态是重复的,可以合并为一个。电路的状态数越少,存储电路也就越简单。状态化简的目的就在于将等价状态尽可能地合并,以得到最简的状态转换图。

(3) 状态分配 状态分配又称状态编码。通常有很多编码方法,编码方案选择得当,设计的电路可以简单,反之,选得不好,则设计的电路就会复杂许多。在实际设计时,须综合考虑电路复杂度与电路性能之间的折衷。在触发器资源丰富的 FPGA 或 ASIC 设计中,采用独热编码(one-hot-coding)既可以使电路性能得到保证又可充分利用其触发器数量多的优势,也可以采取输出编码的状态指定来简化电路结构,并提高状态机的运行速度。

(4) 选定触发器的类型并求出状态方程、驱动方程和输出方程。

(5) 按照方程得出逻辑图 用 Verilog HDL 来描述有限状态机,可以充分发挥硬件描述语言的抽象建模能力,使用 always 块语句和 case(if)等条件语句及赋值语句即可方便实现。具体的逻辑化简、逻辑电路到触发器映射均可由计算机自动完成。上述设计步骤中的第(2)步及(4)、(5)步不再需要很多的人为干预,使电路设计工作得到简化,效率也有很大的提高。

思 考 题

1. 举例说明状态分配对状态机电路的复杂度和速度的影响。

2. 分别说明和解释[例 12.1]~[例 12.4]中两种不同赋值(即非阻塞赋值“ $<=$ ”和阻塞赋值“ $=$ ”)的用法, 和逻辑关系等号“ $=$ ”的含义。
3. 一般情况下状态机中的状态变量是用来干什么的? 是否可以把状态变量中的某些位指定为状态机的输出, 直接用来控制逻辑开关? 这样做有什么好处? 有什么缺点?
4. 分析[例 12.1]~[例 12.4]中用 Verilog 编写的状态机模块。经综合后产生的电路结构中, 哪个属于 Mealy 状态机? 哪个属于 Moore 状态机? 请在认真分析及综合出来的电路结构后, 给出正确的答案。
5. 如果需要设计带流水线输出的 Mealy 状态机, 其 Verilog 模块应该如何编写? 请您编写一下, 并通过综合器产生电路结构, 分析其电路结构和时序。
6. 在状态机的测试模块中, 最后面的 initial 块语句有什么作用, 若测试模块中没有最后的 initial 语句块能不能进行仿真? 如果能, 需要注意什么? 本测试模块还有什么地方没有测试到? 应该如何改进?

第 13 章 设计可综合的状态机的指导原则

概 述

同步有限状态机的 Verilog 模块其实并不难编写,在第 12 章中已经通过一个简单的例子说明了如何用 Verilog 语言来表达一个简单的状态机。这个例子虽然简单,但可以在这样一个模块的基础上,加上若干个状态和条件,就可以表示很复杂的状态机。第 12 章中所介绍的状态机模块设计举例就是用 Verilog 语言编写的显式状态机显示状态机代码的标准样板。任何一种综合工具都能把这种风格的 Verilog 模块转换为门级网表。有了综合工具,把一个已抽象为状态机的思路变为具体的由门级逻辑元件组成的电路变得非常简单。关键的问题是如何才能把一个电路系统抽象为一个或多个互相配合嵌套的状态机和组合逻辑模块?什么是标准的可以被任何综合器接受,并能转换为门级网表的 Verilog 状态机模块呢?编写时需要注意什么?在本章中将通过许多具体的例子着重讲解这些问题。

13.1 用 Verilog HDL 语言设计 可综合的状态机的指导原则

因为大多数 FPGA 内部的触发器数目相当多,又加上独热码状态机(one hot state machine)的译码逻辑最为简单,所以在设计采用 FPGA 实现的状态机时,往往采用独热码状态机(即每个状态只有一个寄存器置位的状态机)。

建议采用 case, casex 或 casez 语句来建立状态机的模型,因为这些语句表达清晰明了,可以方便地从当前状态分支转向下一个状态并设置输出。不要忘记写上 case 语句的最后一个分支 default,并将状态变量设为 'bx,这就等于告知综合器:case 语句已经指定了所有的状态。这样综合器就可以删除不需要的译码电路,使生成的电路简洁,并与设计要求一致。

如果将默认状态设置为某一确定的状态(例如:设置 default:state = statel)行不行呢?回答是:这样做有一个问题需要注意。因为尽管综合器产生的逻辑和设置 default:state='bx 时相同,但是状态机的 Verilog HDL 模型综合前和综合后的仿真结果会不一致。为什么会是这样呢?因为启动仿真器时,状态机所有的输入都不确定,因此立即进入 default 状态。这样的设置便会将状态变量设为 statel,但是实际硬件电路的状态机在通电之后,进入的状态是不确定的,很可能不是 statel 的状态,因此还是设置 default:state='bx 与实际情况更一致。但在有多余状态的情况下,可以通过综合指令(关于综合指令将在高级篇里讲解)将默认状态设置为某一确定的有效状态,因为这样做能使状态机在偶然进入多余状态后,仍能在下一时钟跳变时返回正常工作状态,否则会引起死锁。

状态机应该有一个异步或同步复位端,以便在通电时将硬件电路复位到有效状态,也可以在操作中将硬件电路复位(大多数 FPGA 结构都允许使用异步复位端)。

目前大多数综合器往往不支持在一个 always 块中由多个事件触发的状态机(即隐含状态机, implicit state machines),为了能综合出有效的电路,用 Verilog HDL 描述的状态机应明确地由唯一时钟触发。如果设计要求必须有不同的时钟触发的状态机,可以采用以下办法:编写另一个模块,在那个模块中使用另外一个时钟;然后用实例引用的方法在另外一个模块中把它们连接起来。为了使设计比较简单,调试比较容易,应该尽量使这两个状态机的时钟有一定关系。例如甲模块的时钟是乙模块时钟同步计数器的输出。

把异步触发的电路转换为同步时钟触发的电路并不困难,将在 15 章[例 15.3]中见到这样一个例子。目前大多数综合器不能综合,但可采用 Verilog HDL 描述的异步状态机转换为电路网表。异步状态机是没有确定时钟的状态机,它的状态转移不是由唯一的时钟跳变沿所触发。

之所以不能用异步状态机来综合的原因是因为异步状态机不容易规范触发的瞬间,因此不容易判别触发脉冲是正常的触发还是冒险竞争产生的毛刺。而同步状态机的时钟是由同一时钟产生的,通过特殊设计的全局时钟网络将唯一的时钟连接到每个触发器的时钟端,使得时钟沿到达每个触发器时钟端的时刻几乎完全相同。因此,只要知道组合逻辑的延迟,例如小于一个时钟周期或若干个时钟周期,就可以安排适当的时序让组合逻辑块有最快的稳定和正确的输出,并可靠地存在寄存器中,作为另一级电路的输入。若存在异步的触发时钟,则逻辑电路就很难用统一的方法来解决由于逻辑元件延迟产生的冒险和竞争现象出现的毛刺所造成的混乱。

千万不要使用综合工具来设计异步状态机。因为目前大多数综合工具在对异步状态机进行逻辑优化时会胡乱地简化逻辑,使综合后的异步状态机不能正常工作。如果一定要设计异步状态机,建议采用电路图输入的方法(或用实例引用的写法把几个引用的实例用异步时钟连接起来),而不要直接用 Verilog RTL 级别的描述方法通过综合来产生。

在 Verilog HDL 中,状态必须明确赋值。通常使用参数(parameters)或宏定义(define)语句加上赋值语句来实现。使用参数(parameters)语句赋状态值见下列程序:

```
parameter state1 = 2'h1, state2 = 2'h2;
:
current_state = state2;           //把 current state 设置成 2'h2
:
```

使用宏定义(define)语句赋状态值见下列程序:

```
'define state1 2'h1
#define state2 2'h2
:
current_state = `state2; //把 current state 设置成 2'h2
```

13.2 典型的状态机实例

以宇宙飞船控制器作为典型的状态机分析如下:

```
module statemachine( launch_shuttle, land_shuttle, start_countdown,
                     start_trip_meter, clk, all_systems_go,
```

```
        just_launched, is_landed, cnt, abort_mission);  
output  launch_shuttle, land_shuttle, start_countdown, start_trip_meter;  
input    clk, just_launched, is_landed, abort_mission, all_systems_go;  
input    [3:0]  cnt;  
reg      launch_shuttle, land_shuttle, start_countdown, start_trip_meter;  
          //设置独热码状态的参数  
parameter HOLD=5'b00001, SEQUENCE=5'b00010, LAUNCH=5'b00100;  
parameter ON_MISSION=5'b01000, LAND=5'b10000;  
reg      [4:0] state;  
  
always @(negedge clk or posedge abort_mission)  
begin  
    //检查异步 reset 的值,即 abort_mission 的值  
    if(abort_mission)  
        begin  
            {launch_shuttle, land_shuttle, start_trip_meter, start_countdown}<= 4'b0000;  
            state<=LAND;  
        end  
    else  
        begin  
            /* 主状态机,状态变量 state */  
            case(state)  
                HOLD: if(all_systems_go)  
                    begin  
                        state<=SEQUENCE;  
                        start_countdown<=1;  
                    end  
                else  
                    state<=HOLD;  
                SEQUENCE: if(cnt==0)  
                    state <= LAUNCH;  
                else  
                    state<=SEQUENCE;  
                LAUNCH:  
                    begin  
                        state<=ON_MISSION;  
                        launch_shuttle<=1;  
                    end  
                ON_MISSION:  
                    //取消使命前,一直留在使命状态  
                    if(just_launched)  
                        start_trip_meter<=1;  
                    else  
                        state<=ON_MISSION;
```

```

LAND:
    if(is_landed)
        state<=HOLD;
    else
        begin
            land_shuttle<=1;
            state<=LAND;
        end
    default: state<=5'bxxxxx;
endcase
end
// end of if-else
endmodule

```

13.3 综合的一般原则

通常,综合的一般原则为:

- (1) 综合之前一定要进行仿真,这是因为仿真会暴露逻辑错误,所以建议大家这样做。如果不做仿真,没有发现的逻辑错误会进入综合器,使综合的结果产生同样的逻辑错误。
- (2) 每一次布局布线之后都要进行仿真,在器件编程或流片之前要做最后的仿真。
- (3) 用 Verilog HDL 描述的异步状态机是不能综合的,因此应该避免用综合器来设计;如果一定要设计异步状态机,则可用电路图输入的方法来设计。
- (4) 如果要为电平敏感的锁存器建模,使用连续赋值语句是最简单的方法。

13.4 语言指导原则

1. always 块

- (1) 每个 always 块只能有一个事件控制“@(event-expression)”,而且要紧跟在 always 关键字的后面。
- (2) always 块可以表示时序逻辑或者组合逻辑,也可以用 always 块既表示电平敏感的透明锁存器又同时表示组合逻辑,但是不推荐使用这种描述方法,因为这容易产生错误和多余的电平敏感的透明锁存器。
- (3) 带有 posedge 或 negedge 关键字的事件表达式表示沿触发的时序逻辑,没有 posedge 或 negedge 关键字的表示组合逻辑或电平敏感的锁存器,或者两种都表示。在表示时序和组合逻辑的事件控制表达式中,如有多个沿和多个电平,其间必须用关键字“or”连接。
- (4) 每个表示时序 always 块只能由一个时钟跳变沿触发,置位或复位最好也由该时钟跳变沿触发。
- (5) 每个在 always 块中赋值的信号都必须定义成 reg 型或整型。整型变量默认为 32 位,使用 Verilog 操作符可对其进行二进制求补的算术运算。综合器还支持整型变量的范围说明,这样就允许产生不是 32 位的整型量。句法结构为:integer[<msb>:<lsb>]<identifier>。
- (6) always 块中应该避免组合反馈回路。每次执行 always 块时,在生成组合逻辑的

always块中赋值的所有信号必须有明确的值;否则,需要设计者在设计中加入电平敏感的锁存器来保持赋值前的最后一个值。只有这样,综合器才能正常生成电路;如果不这样做,综合器会发出警告,提示设计中插入了锁存器。如果在设计中存在综合器认为不是电平敏感锁存器的组合回路时,综合器会发出错误信息(例如设计中有异步状态机时)。

上述原则不太好理解,让我们再解释一下。这也就是说,用 always 块设计纯组合逻辑电路时,在生成组合逻辑的 always 块中参与赋值的所有信号都必须有明确的值,即在赋值表达式右端参与赋值的信号都必须在 always @ (敏感电平列表) 中列出。如果在赋值表达式右端引用了敏感电平列表中没有列出的信号,那么在综合时,将会为该没有列出的信号隐含地产生一个透明锁存器。这是因为该信号的变化不会立刻引起所赋值的变化,而必须等到敏感电平列表中某一个信号变化时,它的作用才显现出来,也就是相当于存在着一个透明锁存器,即把该信号的变化暂存起来,待敏感电平列表中某一个信号变化时再起作用,纯组合逻辑电路不可能做到这一点。这样,综合后所得的电路已经不是纯组合逻辑电路了,这时综合器会发出警告,提示设计中插入了锁存器。见下例。

```
例如: input a,b,c;
      reg e,d;
      always @(a or b or c)
      begin
          e = d & a & b;
          /* 因为 d 没有在敏感电平列表中,所以 d 变化时,e 不能立刻变化,要等到 a 或 b 或 c
             变化时才体现出来。这就是说,实际上相当于存在一个电平敏感的透明锁存器在
             起作用,把 d 信号的变化锁存其中 */
          d = e | c;
      end
```

2. 赋 值

(1) 对一个寄存器型(reg)和整型(integer)变量给定位的赋值,只允许在一个 always 块内进行,如在另一 always 块中也对其赋值,这是非法的。

(2) 把某一信号值赋为 'bx,综合器就把它解释成无关状态,因而综合器为其生成的硬件电路最简洁。

13.5 可综合风格的 Verilog HDL 模块实例

13.5.1 组合逻辑电路设计实例

【例 13.1】 8 位带进位端的加法器的设计实例(利用简单的算法描述)。

```
module adder_8(cout,sum,a,b,cin);
    output cout;
    output [7:0] sum;
    input cin;
    input[7:0] a,b;
    assign {cout,sum} = a+b+cin;
```

```
endmodule
```

【例 13.2】 指令译码电路的设计实例(利用电平敏感的 always 块来设计组合逻辑)。

```
//操作码的宏定义
`define plus      3'd0
`define minus     3'd1
`define band      3'd2
`define bor       3'd3
`define unegate   3'd4

module alu (out,opcode,a,b);
    output [7:0] out;
    input  [2:0] opcode;
    input  [7:0] a,b;
    reg      [7:0] out;

    always @ (opcode or a or b)
        //用电平敏感的 always 块描述组合逻辑
        begin
            case(opcode)
                //算术运算
                `plus: out=a+b;
                `minus: out=a-b;
                //位运算
                `band: out=a&b;
                `bor:  out=a|b;
                //单目运算
                `unegate: out=~a;
                default: out=8'hx;
            endcase
        end
endmodule
```

【例 13.3】 利用 task 和电平敏感的 always 块设计经比较后重组信号的组合逻辑。

```
module sort4(ra,rb,rc,rd,a,b,c,d);
    parameter t=3;
    output [t:0] ra, rb, rc, rd;
    input  [t:0] a, b, c, d;
    reg      [t:0] ra, rb, rc, rd;

    always @ (a or b or c or d)
        //用电平敏感的 always 块描述组合逻辑
        begin: local //此处 begin-end 块必须有一模块名,因为块中定义了局部变量
```

```

reg [t:0] va, vb, vc, vd;
{va,vb,vc,vd}={a,b,c,d};
sort2(va,vc);
sort2(vb,vd);
sort2(va,vb);
sort2(vc,vd);
sort2(vb,vc);
{ra,rb,rc,rd}={va,vb,vc,vd};

end

task sort2;
inout [t:0] x, y;
reg [t:0] tmp;
if( x > y )
begin
    tmp = x;
    x = y;
    y = tmp;
end
endtask

endmodule

```

【例 13.4】 比较器的设计实例(利用赋值语句设计组合逻辑)。

```

module compare(equal,a,b);
parameter size=1;
output equal;
input [size-1:0] a, b;
assign equal =(a==b)? 1 : 0;
endmodule

```

【例 13.5】 3-8 译码器设计实例(利用赋值语句设计组合逻辑)。

```

module decoder(out,in);
output [7:0] out;
input [2:0] in;
assign out = 1'b1<<in;
/* * * * 把最低位的 1 左移 in(根据从 in 口输入的值)位,并赋予 out * * * */
endmodule

```

【例 13.6】 8-3 编码器的设计实例。

编码器设计方案之一：

```
module encoder1(none_on,out,in);
```

```

        output none_on;
        output [2:0] out;
        input [7:0] in;
        reg [2:0] out;
        reg none_on;
        always @(in)
        begin: local //此处 begin-end 块必须有一模块名,因为块中定义了局部变量
            integer i;
            out = 0;
            none_on = 1;
            /* 返回输入信号 in 的 8 位中,为 1 的最高位数 */
            for( i=0; i<8; i=i+1 )
                begin
                    if( in[i] )
                        begin
                            out = i;
                            none_on = 0;
                        end
                end
            end
        end
    endmodule

```

编码器设计方案之二：直接查表法（通过 Verilog 语句实现）

```

module encoder2 ( none_on, out2, out1, out0, h, g, f, e, d, c, b, a);
    input h, g, f, e, d, c, b, a;
    output none_on, out2, out1, out0;
    wire [3:0] outvec;

    assign outvec= h? 4'b0111 : g? 4'b0110 : f? 4'b0101:
    e? 4'b0100 : d? 4'b0011 :c? 4'b0010 : b? 4'b0001:
    a? 4'b0000 : 4'b1000;

    assign none_on = outvec[3];
    assign     out2 = outvec[2];
    assign     out1 = outvec[1];
    assign     out0 = outvec[0];

endmodule

```

编码器设计方案之三：

```

module encoder3 (none_on, out2, out1, out0, h, g,f, e, d, c, b, a);
    input h, g, f, e, d, c, b, a;

```

```

output out2, out1, out0;
output none_on;
reg [3:0] outvec;

assign {none_on,out2,out1,out0} = outvec;

always @ ( a or b or c or d or e or f or g or h)
begin
  if(h)      outvec=4'b0111;
  else if(g) outvec=4'b0110;
  else if(f) outvec=4'b0101;
  else if(e) outvec=4'b0100;
  else if(d) outvec=4'b0011;
  else if(c) outvec=4'b0010;
  else if(b) outvec=4'b0001;
  else if(a) outvec=4'b0000;
  else       outvec=4'b1000;
end
endmodule

```

【例13.7】多路器的设计实例。

使用连续赋值、case语句或if-else语句可以生成多路器电路。如果条件语句(case或if-else)中分支条件是互斥的话,综合器能自动地生成并行的多路器。

多路器设计方案之一:

```

module emux1(out, a, b, sel);
  output out;
  input a, b, sel;
  assign out = sel? a : b;
endmodule

```

多路器设计方案之二:

```

module mux2( out, a, b, sel);
  output out;
  input a, b, sel;
  reg out;
//用电平触发的always块来设计多路器的组合逻辑
  always @ ( a or b or sel )
    begin
      /* 检查输入信号sel的值,如为1,输出out为a,如为0,输出out为b. */
      case( sel )
        1'b1: out = a;
        1'b0: out = b;
        default: out = 'bx;
      endcase
    end
endmodule

```

```

    endcase
  end
endmodule

```

多路器设计方案之三：

```

module mux3( out, a, b, sel);
  output out;
  input a, b, sel;
  reg out;
  always @ ( a or b or sel )
  begin
    if( sel )
      out = a;
    else
      out = b;
  end
endmodule

```

【例 13.8】奇偶校验位生成器设计实例。

```

module parity( even_numbits,odd_numbits,input_bus);
  output even_numbits, odd_numbits;
  input [7:0] input_bus;
  assign odd_numbits = ~input_bus;
  assign even_numbits = ~odd_numbits;
endmodule

```

【例 13.9】三态输出驱动器设计实例(用连续赋值语句建立三态门模型)。

三态输出驱动器设计方案之一：

```

module trist1( out, in, enable);
  output out;
  input in, enable;
  assign out = enable? in: 'bz;
endmodule

```

三态输出驱动器设计方案之二：

```

module trist2( out, in, enable );
  output out;
  input in, enable;
  //bufif1 是一个 Verilog 门级原语(primitive)
  bufif1 mybuf1(out, in, enable);
endmodule

```

【例 13.10】三态双向驱动器设计实例。

```

module bidir(tri inout, out, in, en, b);

```

```

inout tri_inout;
output out;
input in, en, b;
assign tri_inout = en? In : 'bz;
assign out = tri_inout ^ b;
endmodule

```

13.5.2 时序逻辑电路设计实例

【例 13.11】 触发器设计实例。

```

module dff( q, data, clk );
    output q;
    input data, clk;
    reg q;
    always @ ( posedge clk )
        begin
            q <= data;
        end
endmodule

```

【例 13.12】 电平敏感型锁存器设计实例之一。

```

module latch1( q, data, clk );
    output q;
    input data, clk;
    assign q = clk? data : q;
endmodule

```

【例 13.13】 带置位和复位端的电平敏感型锁存器设计实例之二。

```

module latch2( q, data, clk, set, reset );
    output q;
    input data, clk, set, reset;
    assign q = reset? 0 : ( set? 1 : (clk? data : q) );
endmodule

```

【例 13.14】 电平敏感型锁存器设计实例之三。

```

module latch3( q, data, clk );
    output q;
    input data, clk;
    reg q;
    always @ (clk or data)
        begin
            if(clk)
                q = data;
        end
endmodule

```

```
    end
endmodule
```

注意:有的综合器会产生一警告信息,告诉你产生了一个电平敏感型锁存器。因为我们设计的就是一个电平敏感型锁存器,就不用管这个警告信息。

【例 13.15】 移位寄存器设计实例。

```
module shifter( din, clk, clr, dout);
    input din, clk, clr;
    output [7:0] dout;
    reg [7:0] dout;
    always @(posedge clk)
        begin
            if(clr)                      //清零
                dout <= 8'b0;
            else
                begin
                    dout <= dout<<1;   //左移一位
                    dout[0] <= din;      //把输入信号放入寄存器的最低位
                end
        end
endmodule
```

【例 13.16】 8 位计数器设计实例之一。

```
module counter1( out, cout, data, load, cin, clk);
    output [7:0] out;
    output cout;
    input [7:0] data;
    input load, cin, clk;
    reg [7:0] out;
    always @(posedge clk)
        begin
            if( load )
                out <= data;
            else
                out <= out + cin;
        end
    assign cout=(& out) & cin;
    //只有当 out[7:0]的所有各位都为 1,并且进位 cin 也为 1 时才能产生进位 cout
endmodule
```

【例 13.17】 8 位计数器设计实例之二。

```

module counter2( out, cout, data, load, cin, clk);
    output [7:0] out;
    output cout;
    input [7:0] data;
    input load, cin, clk;
    reg [7:0] out;
    reg cout;
    reg [7:0] preout;
    //创建 8 位寄存器
    always @(posedge clk)
    begin
        out <= preout;
    end
    / * * * * 计算计数器和进位的下一个状态。注意：为提高性能不希望加载影响进位 * * * */
    always @ ( out or data or load or cin )
    begin
        {cout, preout} = out + cin;
        if(load)
            preout = data;
    end
endmodule

```

13.6 状态机的置位与复位

13.6.1 状态机的异步置位与复位

异步置位与复位是与时钟无关的。当异步置位与复位到来时它们立即分别置触发器的输出为 1 或 0，不需要等到时钟沿到来才置位或复位。把它们列入 always 块的事件控制括号内就能触发 always 块的执行。因此，当它们到来时就能立即执行一次指定的操作。所以当触发条件（如时钟的跳变沿）反复出现时，就可以反复执行指定的操作。

状态机的异步置位与复位是用 always 块和事件控制实现的。先让我们来看一下事件控制的语法：

(1) 事件控制语法：

```

@( <沿关键词 时钟信号
      or 沿关键词 复位信号
      or 沿关键词 置位信号> )

```

沿关键词包括 posedge(用于高电平有效的 set、reset 或上升沿触发的时钟) 和 negedge(用于低电平有效的 set、reset 或下降沿触发的时钟)，信号可以按任意顺序列出。

(2) 事件控制实例：

① 异步、高电平有效的置位(时钟的上升沿):@ (posedge clk or posedge set)。

② 异步、低电平有效的复位(时钟的上升沿):@(posedge clk or negedge reset)。

③ 异步、低电平有效的置位和高电平有效的复位(时钟的上升沿):@(posedge clk or negedge set or posedge reset)。

④ 带异步、高电平有效的置位与复位的 always 块样板:

```
always @(posedge clk or posedge set or posedge reset)
begin
  if(reset)
    begin
      /* 置输出为 0 */
    end
  else
    if(set)
      begin
        /* 置输出为 1 */
      end
    else
      begin
        /* 与时钟同步的逻辑 */
      end
  end
end
```

⑤ 带异步高电平有效的置/复位端的 D 触发器实例:

```
module dff1( q, qb, d, clk, set, reset );
  input d, clk, set, reset;
  output q, qb;
  //声明 q 和 qb 为 reg 类型,因为它需要在 always 块内赋值
  reg q, qb;

  always @ (posedge clk or posedge set or posedge reset )
  begin
    if(reset)
      begin
        q<=0;
        qb<=1;
      end
    else
      if (set)
        begin
          q<=1;
          qb<=0;
        end
      else
        begin
          q<=d;
          qb<=1;
        end
    end
  end
endmodule
```

```

begin
    q<=d;
    qb<=~d;
end
end
endmodule

```

13.6.2 状态机的同步置位与复位

同步置位与复位是指只有在时钟的有效跳变沿时刻置位或复位,信号才能使触发器置位或复位(即,使触发器的输出分别转变为逻辑1或0)。因此不要把set和reset信号名列入always块的事件控制表达式,因为当它们有变化时不应触发always块的执行。相反,always块的执行应只由时钟有效跳变沿触发,是否置位或复位应在always块中首先检查set和reset信号的电平。所以, set或reset的电平维持时间必须大于时钟沿的间隔时间,否则set和reset不能每次都能有效地完成置位和复位的工作。为此在编写测试模块时和设计与其配合的电路时要注意这个问题。

(1) 事件控制语法:

@(<沿关键词 时钟信号>)

其中,沿关键词是指posedge(正沿触发的时钟)或negedge(负沿触发的时钟)。

(2) 事件控制实例:

① 正沿触发:

@(posedge clk)

② 负沿触发:

@(negedge clk)

③ 同步的具有高电平有效的置位与复位端的always块样板:

```
always @(posedge clk)
```

```
begin
```

```
if(reset)
```

```
begin
```

```
/* 置输出为0 */
```

```
end
```

```
else
```

```
if(set)
```

```
begin
```

```
/* 置输出为1 */
```

```
end
```

```
else
```

```
begin
```

```
/* 与时钟同步的逻辑 */
```

```
end
```

```
end
```

④ 同步的具有高电平有效的置位/复位端的 D 触发器：

```
module dff2( q, qb, d, clk, set, reset );
    input d, clk, set, reset;
    output q, qb;
    reg q, qb;
    always @(posedge clk)
        begin
            if(reset)
                begin
                    if(qb<=0);
                    q<=0;
                    qb<=1;
                end
            else
                if(set)
                    begin
                        begin
                            q<=1;
                            qb<=0;
                        end
                    end
                else
                    begin
                        begin
                            q<=d;
                            qb<=~d;
                        end
                    end
                end
            end
        endmodule
```

小 结

本章是 Verilog 设计方法学中最重要的一章。在这一章中阐述了什么样风格的 Verilog 模块是可以综合成电路结构的，以及综合的一般原则。其实对于一般的综合工具而言，可以借助于工具自动综合成电路结构的 Verilog 模块风格非常有限，即只有本章 13.1 节中组合逻辑和 13.6 节同步状态机的标准写法。因此可以通过阅读 13.5 节中的许多小例子和 13.6 节以下的模块样板来理解这些基本原则。在设计中围绕着这些基本原则来编写模块。

当系统比较复杂时，需要通过仔细的分析，把一个具体系统分解为数据流和控制流。构想哪些部分用组合逻辑，哪些部分的资源可以共享而不影响系统的性能，需要设置哪些开关逻辑来控制数据的流动，需要一个或几个同步有限状态机来正确有序地控制这些开关逻辑，以便有效地利用有限的硬件资源，才能编写出真正有价值的 RTL 级源代码，从而综合出有实用价值的高性能的数字逻辑电路系统。因此，可以说，认真地学习并掌握数字电路基础和计算机体系结构这两门学科的真谛是 Verilog 数字系统设计的基础。

思 考 题

思考题

1. 是不是只要符合 Verilog 语法规则行为正确的模块都可以综合成电路结构?
2. 为什么在用 Verilog 设计方法时不采用异步的状态机,采用异步状态机有什么问题不好解决?
3. 用 always 块语句如何编写纯组合逻辑电路? 在哪些情况下会生成不需要的锁存器?
4. 请用清晰的语句把标准的可综合的带同步复位端的同步状态机的样板模块表达出来。
5. 请用清晰的语句把标准的可综合的带异步复位端的同步状态机的样板模块表达出来。
6. 这两种不同的同步状态机有什么不同? 如果输入的复位脉冲很窄,哪种状态机不能可靠复位?
7. 为什么说,掌握数字电路基础和计算机体系结构这两门学科的真谛是 Verilog 数字系统设计的基础?
8. 如果一定要设计异步触发的计数电路,用 Verilog 描述有什么办法? 能否综合? 仿真时要注意什么问题?
9. 把本章中的例题编写完整。没有编写测试模块的编写相应的测试模块,并在 verilog 仿真环境下运行,以全面验证设计的正确。

思考题

本章主要介绍了设计可综合的同步状态机的基本方法。通过学习,读者应该能够掌握设计同步状态机的一般方法,并能根据具体的应用需求,灵活地设计出满足要求的同步状态机。同时,通过本章的学习,读者应该能够理解 Verilog 语言在设计同步状态机方面的优势,并能够熟练地使用 Verilog 语言来描述同步状态机。通过本章的学习,读者应该能够掌握设计同步状态机的一般方法,并能根据具体的应用需求,灵活地设计出满足要求的同步状态机。同时,通过本章的学习,读者应该能够理解 Verilog 语言在设计同步状态机方面的优势,并能够熟练地使用 Verilog 语言来描述同步状态机。

通过本章的学习,读者应该能够掌握设计可综合的同步状态机的基本方法。通过学习,读者应该能够掌握设计同步状态机的一般方法,并能根据具体的应用需求,灵活地设计出满足要求的同步状态机。同时,通过本章的学习,读者应该能够理解 Verilog 语言在设计同步状态机方面的优势,并能够熟练地使用 Verilog 语言来描述同步状态机。

通过本章的学习,读者应该能够掌握设计可综合的同步状态机的基本方法。通过学习,读者应该能够掌握设计同步状态机的一般方法,并能根据具体的应用需求,灵活地设计出满足要求的同步状态机。同时,通过本章的学习,读者应该能够理解 Verilog 语言在设计同步状态机方面的优势,并能够熟练地使用 Verilog 语言来描述同步状态机。

第 14 章 深入理解阻塞和非阻塞赋值的不同

概 述

在第 13 章中已经讲解了有限状态机的几种标准,即 Verilog 模块的样板。这些样板各有适用的对象,在状态变量的赋值或开关变量的赋值中,已明确建议大家使用非阻塞赋值。这不仅是因为综合工具要求这样做,最根本的原因是与非阻塞赋值语句对应的电路结构正是我们想要实现的。对于数字电路的初学者来说,由于电路结构知识和经验的不足,不太容易深入理解阻塞和非阻塞赋值语句在语意上的细微不同,其实这两种赋值语句对应着两种不同的电路结构。阻塞赋值对应的电路结构往往与触发沿没有关系,只与输入电平的变化有关系。而非阻塞赋值对应的电路结构往往与触发沿有关系,只有在触发沿时才有可能发生赋值的情况。本章是根据国外一篇论文,经过自己的理解编写而成的,希望能在更深入的层次上帮助大家理解这两种赋值的不同,使我们设计出更合乎要求的数字电路。由于论文的作者对于综合器和仿真器的内部原理有深入的了解,所以才能对这两种赋值语句的细微不同有深刻的认识。

14.1 阻塞和非阻塞赋值的异同

阻塞和非阻塞赋值的语言结构是 Verilog 语言中最难理解的概念之一。甚至有些很有经验的 Verilog 设计工程师也不能完全正确地理解:何时使用非阻塞赋值及何时使用阻塞赋值才能设计出符合要求的电路。他们也不完全明白在电路结构的设计中,即可综合风格的 Verilog 模块的设计中,究竟为什么还要用非阻塞赋值,以及符合 IEEE 标准的 Verilog 仿真器究竟如何来处理非阻塞赋值的仿真。本小节的目的是尽可能地把阻塞和非阻塞赋值的含义详细地解释清楚,并明确地提出可综合的 Verilog 模块编程在使用赋值操作时应注意的要点。按照这些要点来编写代码就可以避免在 Verilog 仿真时出现冒险和竞争的现象。我们在前面曾提到过下面两个要点:

- (1) 在描述组合逻辑的 always 块中用阻塞赋值,则综合成组合逻辑的电路结构;
- (2) 在描述时序逻辑的 always 块中用非阻塞赋值,则综合成时序逻辑的电路结构。

为什么一定要这样做呢?这是因为要使综合前仿真和综合后仿真一致的缘故。如果不按照上面两个要点来编写 Verilog 代码,也有可能综合出正确的逻辑,但前后仿真的结果就会不一致。

为了更好地理解上述要点,需要对 Verilog 语言中的阻塞赋值和非阻塞赋值的功能和执行时间上的差别有深入的了解。为了解释问题方便,下面定义两个缩写:

RHS——赋值等号右边的表达式或变量可分别缩写为 RHS 表达式或 RHS 变量;

LHS——赋值等号左边的表达式或变量可分别缩写为 LHS 表达式或 LHS 变量。

IEEE Verilog 标准定义了有些语句有确定的执行时间,有些语句没有确定的执行时间。

若有两条或两条以上语句准备在同一时刻执行,但由于语句的排列顺序不同(而这种排列顺序的不同是 IEEE Verilog 标准所允许的),却产生了不同的输出结果。这就是造成 Verilog 模块冒险和竞争现象的原因。为了避免产生竞争,理解阻塞和非阻塞赋值在执行时间上的差别是至关重要的。

14.1.1 阻塞赋值

阻塞赋值操作符用等号(即 =)表示。为什么称这种赋值为阻塞赋值呢?这是因为在赋值时先计算等号右手方向(RHS)部分的值,这时赋值语句不允许任何别的 Verilog 语句的干扰,直到现行的赋值完成时刻,即把 RHS 赋值给 LHS 的时刻,它才允许别的赋值语句的执行。一般可综合的阻塞赋值操作在 RHS 不能设定有延迟(即使是零延迟也不允许)。从理论上讲,它与后面的赋值语句只有概念上的先后,而无实质上的延迟。若在 RHS 上加延迟,则在延迟期间会阻止赋值语句的执行,延迟后才执行赋值,这种赋值语句是不可综合的,在需要综合的模块设计中不可使用这种风格的代码。

阻塞赋值的执行可以认为是只有一个步骤的操作,即计算 RHS 并更新 LHS,此时不能允许有来自任何其他 Verilog 语句的干扰。所谓阻塞的概念是指在同一个 always 块中,其后面的赋值语句从概念上(即使不设定延迟)是在前一句赋值语句结束后再开始赋值的。

如果在一个过程块中阻塞赋值的 RHS 变量正好是另一个过程块中阻塞赋值的 LHS 变量,这两个过程块又用同一个时钟沿触发,这时阻塞赋值操作会出现问题,即如果阻塞赋值的顺序安排不好,就会出现竞争。若这两个阻塞赋值操作用同一个时钟沿触发,则执行的顺序是无法确定的。[例 14.1]可以说明这个问题。

【例 14.1】 采用阻塞赋值的反馈振荡器。

```
module fbosc1 (y1, y2, clk, rst);
    output y1, y2;
    input  clk, rst;
    reg     y1, y2;

    always @ (posedge clk or posedge rst)
        if (rst) y1 = 0; // reset
        else     y1 = y2;

    always @ (posedge clk or posedge rst)
        if (rst) y2 = 1; // preset
        else     y2 = y1;
endmodule
```

按照 IEEE Verilog 的标准,[例 14.1]中两个 always 块是并行执行的,若复位信号已从 1 到 0,且例中的 always 块的有效时钟沿比下面的 always 块的时钟沿早几个皮秒(由时钟偏差造成)到达,则 y1 和 y2 都会取 1;而若下面的那个 always 块的有效时钟沿早几个皮秒到达,则 y1 和 y2 都会取 0。这清楚地说明这个 Verilog 模块是不稳定的,必定会产生冒险和竞争的情况。

14.1.2 非阻塞赋值

非阻塞赋值操作符用小于等于号(即 $<=$)表示。为什么称这种赋值为非阻塞赋值?这是因为赋值操作时刻开始时计算非阻塞赋值符的RHS表达式,赋值操作结束时刻才更新LHS。在计算非阻塞赋值的RHS表达式和更新LHS期间,其他的Verilog语句,包括其他的Verilog非阻塞赋值语句都能同时计算RHS表达式和更新LHS。非阻塞赋值允许其他的Verilog语句同时进行操作。非阻塞赋值的操作过程可以看作两个步骤:

- (1) 在赋值开始时刻,计算非阻塞赋值RHS表达式;
- (2) 在赋值结束时刻,更新非阻塞赋值LHS表达式。

非阻塞赋值操作只能用于对寄存器类型变量进行赋值,因此只能用在“initial”块和“always”块等过程块中,而非阻塞赋值不允许用于连续赋值。**[例14.2]**可以说明这个问题。

【例14.2】采用非阻塞赋值的反馈振荡器。

```
module fbosc2(y1, y2, clk, rst);
    output y1, y2;
    input clk, rst;
    reg y1, y2;

    always @(posedge clk or posedge rst)
        if (rst) y1 <= 0; //预置值
        else y1 <= y2;

    always @(posedge clk or posedge rst)
        if (rst) y2 <= 1; //预置值
        else y2 <= y1;
endmodule
```

同样,按照IEEE Verilog的标准,上例中两个always块是并行执行的,与前后顺序无关。复位信号回到0后,无论哪一个always块的有效沿先到,两个always块中的非阻塞赋值都在赋值开始时刻计算RHS表达式,而在结束时刻才更新LHS表达式。所以,复位信号从1回到0后,无论哪个always块的有效时钟沿早到几个皮秒,y1为1,而y2为0是确定的,因为实质上y1被赋的y2值是由rst正跳变沿确定的,而y2被赋的y1值也是由rst正跳变沿确定的(若以后rst继续保持为0,时钟信号不断重复,则每次被赋值的y1和y2都是由上一个周期的时钟有效沿确定的)。从用户的角度看这两个非阻塞赋值好像是并行执行的。

14.2 Verilog模块编程要点

下面将对阻塞和非阻塞赋值做进一步解释并将举更多的例子来说明这个问题。在此之前,掌握可综合风格的Verilog模块编程的以下8个原则对读者会有很大的帮助。在编写时Verilog代码时必须牢记这8个要点,才能在综合布局布线后的仿真中避免出现冒险竞争现象。

- (1) 时序电路建模时,用非阻塞赋值。
- (2) 锁存器电路建模时,用非阻塞赋值。
- (3) 用 always 块建立组合逻辑模型时,用阻塞赋值。
- (4) 在同一个 always 块中建立时序和组合逻辑电路时,用非阻塞赋值。
- (5) 在同一个 always 块中不要既用非阻塞赋值又用阻塞赋值。
- (6) 不要在一个以上的 always 块中为同一个变量赋值。
- (7) 用 \$strobe 系统任务来显示用非阻塞赋值的变量值。
- (8) 在赋值时不要使用 #0 延迟。

在后面的讲解中还要对为什么必须记住这些要点再做进一步的解释。Verilog 的新用户在彻底搞明白这两种赋值功能差别之前,一定要牢记这几条要点。按照要点来编写 Verilog 模块程序,就可省去很多麻烦。

14.3 Verilog 的层次化事件队列

详细地了解 Verilog 的层次化事件队列有助于理解 Verilog 的阻塞和非阻塞赋值的功能。所谓层次化事件队列指的是用于调度仿真事件的不同的 Verilog 事件队列。在 IEEE Verilog 标准中,层次化事件队列被看作是一个概念模型。设计仿真工具的厂商如何来实现事件队列,由于关系到仿真器的效率,被视为技术诀窍,不能公开发表,本节也不便作详细介绍。

在 IEEE 1364—1995 Verilog 标准的 5.3 节中定义了层次化事件队列在逻辑上分为用于当前仿真时间的 4 个不同的队列,和用于下一段仿真时间的若干个附加队列。

- (1) 动态事件队列(下列事件执行的顺序可以随意安排):

- ① 阻塞赋值;
- ② 计算非阻塞赋值语句右边的表达式;
- ③ 连续赋值;
- ④ 执行 \$display 命令;
- ⑤ 计算原语的输入和输出的变化。

- (2) 停止运行的事件队列: #0 延时阻塞赋值。

- (3) 非阻塞事件队列: 更新非阻塞赋值语句 LHS(左边变量)的值。

- (4) 监控事件队列:

- ① 执行 \$monitor 命令;
- ② 执行 \$strobe 命令。

- (5) 其他指定的 PLI 命令队列: 其他 PLI 命令。

以上 5 个队列就是 Verilog 的“层次化事件队列”。

大多数 Verilog 事件是由动态事件队列调度的。这些事件包括阻塞赋值、连续赋值、\$display 命令、实例和原语的输入变化以及它们的输出更新、非阻塞赋值语句 RHS 的计算等。而非阻塞赋值语句 LHS 的更新却不由动态事件队列调度。

在 IEEE 标准允许的范围内被加入到这些队列中的事件只能从动态事件队列中清除,而排列在其他队列中的事件要等到被“激活”后,即被排入动态事件队列中后,才能真正开始等待执行。IEEE 1364—1995 Verilog 标准的 5.4 节介绍了一个描述其他事件队列何时被“激活”

的算法。

在当前仿真时间中,另外两个比较常用的队列是非阻塞赋值更新事件队列和监控事件队列详细介绍见后。非阻塞赋值 LHS 变量的更新是安排在非阻塞赋值更新事件队列中,而 RHS 表达式的计算是在某个仿真时刻随机开始的,与上述其他动态事件是一样的。

`$strobe` 和 `$monitor` 显示命令是排列在监控事件队列中。在仿真的每一步结束时刻,当该仿真步骤内所有的赋值都完成以后, `$strobe` 和 `$monitor` 显示出所有要求显示的变量值的变化。

在 Verilog 标准 5.3 节中描述的第 4 个事件队列是停止运行事件队列,所有 `#0` 延时赋值都排列在该队列中。采用 `#0` 延时赋值是因为有些对 Verilog 理解不够深入的设计人员希望在两个不同的程序块中给同一个变量赋值,他们企图在同一个仿真时刻,通过稍加延时赋值来消除 Verilog 可能产生的竞争冒险。这样做实际上会产生问题。因为给 Verilog 模型附加完全不必要的 `#0` 延时赋值,使得定时事件的分析变得很复杂。我们认为采用 `#0` 延时赋值根本没有必要,完全可用其他的方式来代替,因此不推荐使用。

在 14.4 节的一些例子中,其 Verilog 代码的行为常常用上面介绍的层次化事件队列可以得到解释。时件队列的概念也常常用来说明为什么要坚持上面提到的 8 项原则。

14.4 自触发 always 块

一般而言,Verilog 的 always 块不能触发自己,如[例 14.3]中关于使用阻塞赋值的非自触发振荡器等。

【例 14.3】 使用阻塞赋值,不能自行触发的振荡器。

```
module osc1 (clk);
    output clk;
    reg     clk;

    initial #10 clk = 0;
    always @(clk) #10 clk = ~clk;      //该语句等价于:
                                         //always;
                                         //begin;
                                         //@(clk);
                                         // #10 clk=~clk;
                                         //end
endmodule
```

上例描述的时钟振荡器使用了阻塞赋值。在 initial 块中,经过 10 个单位时间的延迟,clk 被立即阻塞赋值为 0。当 clk 电平从不定态变为 0 的事件发生时,使 always 块的 `@(clk)` 条件触发,经过 10 个单位时间的延迟,计算 RHS 表达式(`~clk`)得到 1,并立即更新 LHS 的值,clk 立即被赋予 1。由于在此期间不允许其他语句的干扰,即使 always 循环回到判断触发条件 `@(clk)`,由于此时 clk 电平已经为 1,无法感知从 0 到 1 曾经发生过的变化,所以就阻塞在那里,

只有等待 clk 变为 0 才能进入下一句。因此,这是一个不能自触发的振荡器,不能产生时钟波形。更深入的解释由于涉及到仿真器运行原理细节,不再赘述。[例 14.4]中的振荡器使用的是非阻塞赋值,它是一个自触发振荡器。

【例 14.4】采用非阻塞赋值的自触发振荡器。

```
module osc2 (clk);
    output clk;
    reg     clk;

    initial #10 clk = 0;
    always @ (clk) #10 clk <= ~clk; //该语句等价于:
                                    //always;
                                    //begin;
                                    //@(clk);
                                    // #10 clk<=~clk;
                                    //end
endmodule
```

`@(clk)` 的第一次触发之后,非阻塞赋值的 RHS 表达式便计算出来,并把值赋给 LHS 的事件并安排在更新事件队列中。在非阻塞赋值更新事件队列被激活之前,又遇到了`@(clk)`触发语句,并且 always 块再次对 clk 的值变化产生反应。当非阻塞 LHS 的值在同一时刻被更新时,`@(clk)`再一次触发。该例是自触发式,虽例中的代码能产生周期时钟信号,但在编写仿真测试模块时不建议推荐使用这种写法的时钟信号源。

14.5 移位寄存器模型

图 14.1 表示一个简单的移位寄存器方框图。

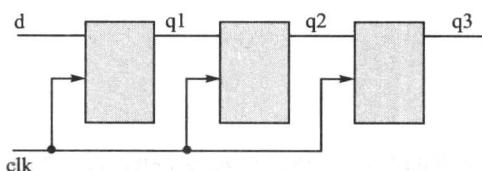


图 14.1 移位寄存器电路

从[例 14.5]~[例 14.8]介绍了 4 种用阻塞赋值实现图 14.1 移位寄存器电路的方式,其中有些是不正确的。

【例 14.5】不正确地使用阻塞赋值来描述移位寄存器(方式 #1)。

```
module pipeb1 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input        clk;
```

```

module pipeb1 (q3, q2, q1, d, clk);
    reg [7:0] q3, q2, q1;
    output [7:0] q3;
    input [7:0] d;
    input      clk;

    always @ (posedge clk)
        begin
            q1 = d;
            q2 = q1;
            q3 = q2;
        end
endmodule

```

在图 14.1 所示的模块中,按顺序进行的阻塞赋值将使得在下一个时钟上升沿时刻,所有的寄存器输出值都等于输入值 d。在每个时钟上升沿,输入值 d 将无延时地直接输出到 q3。

显然,上面的模块实际上被综合成只有一个寄存器的电路(见图 14.2),这并不是当初想要设计的移位寄存器电路。

【例 14.6】 用阻塞赋值来描述移位寄存器是可行的,但这种风格并不好(方式 #2)。

```

module pipeb2 (q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input      clk;
    reg      [7:0] q3, q2, q1;

    always @ (posedge clk)
        begin
            q3 = q2;
            q2 = q1;
            q1 = d;
        end
endmodule

```

在本例的模块中,阻塞赋值的顺序是经过仔细安排的,以使仿真的结果与移位寄存器相一致。虽然该模块可被综合成图 14.1 所示的移位寄存器,但不建议使用这种风格的模块来描述时序逻辑。

【例 14.7】 不用阻塞赋值来描述移位时序逻辑的风格(方式 #3)。

```

module pipeb3 (q3, d, clk);
    output [7:0] q3;
    input [7:0] d;
    input      clk;
    reg      [7:0] q3, q2, q1;

```

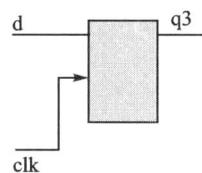


图 14.2 实际综合的结果

```

always @ (posedge clk) q1 = d;
always @ (posedge clk) q2 = q1;
always @ (posedge clk) q3 = q2;
endmodule

```

在[例 14.7]中,阻塞赋值分别被放在不同的 always 块里。仿真时,这些块的先后顺序是随机的,因此可能会出现错误的结果,这是 Verilog 中的竞争冒险。按不同的顺序执行这些块将导致不同的结果。但是,这些代码的综合结果却是正确的流水线寄存器。也就是说,前仿真和后仿真的结果可能会不一致。

【例 14.8】不用阻塞赋值来描述移位时序逻辑的风格(方式 #4)。

```

module pipeb4 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input        clk;
    reg      [7:0] q3, q2, q1;

    always @ (posedge clk) q2 = q1;
    always @ (posedge clk) q3 = q2;
    always @ (posedge clk) q1 = d;
endmodule

```

若在[例 14.8]中仅把 always 块的顺序稍作变动,也可以被综合成正确的移位寄存器逻辑,但仿真结果可能不正确。

如果用非阻塞赋值语句改写以上 4 个阻塞赋值的例子,每一个例子都可以正确仿真,并且综合为设计者期望的移位寄存器逻辑。

【例 14.9】正确使用非阻塞赋值来描述时序逻辑的设计风格(方式 #1)。

```

module pipen1 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input        clk;
    reg      [7:0] q3, q2, q1;

    always @ (posedge clk) begin
        q1 <= d;
        q2 <= q1;
        q3 <= q2;
    end
endmodule

```

【例 14.10】正确使用非阻塞赋值来描述时序逻辑的设计风格(方式 #2)。

```

module pipen2 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;

```

```

input      clk;
reg      [7:0] q3, q2, q1;

always @(posedge clk)
begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
end
endmodule

```

【例 14.11】 正确使用非阻塞赋值来描述时序逻辑的设计风格(方式 #3)。

```

module pipen3 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input      clk;
    reg      [7:0] q3, q2, q1;

    always @(posedge clk) q1 <= d;
    always @(posedge clk) q2 <= q1;
    always @(posedge clk) q3 <= q2;
endmodule

```

【例 14.12】 正确使用非阻塞赋值来描述时序逻辑的设计风格(方式 #4)。

```

module pipen4 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input      clk;
    reg      [7:0] q3, q2, q1;

    always @(posedge clk) q2 <= q1;
    always @(posedge clk) q3 <= q2;
    always @(posedge clk) q1 <= d;
endmodule

```

通过以上移位寄存器时序逻辑电路设计的例子表明：

- (1) 4 种阻塞赋值设计方式中有 1 种可以保证仿真正确；
- (2) 4 种阻塞赋值设计方式中有 3 种可以保证综合正确；
- (3) 4 种非阻塞赋值设计方式全部可以保证仿真正确；
- (4) 4 种非阻塞赋值设计方式全部可以保证综合正确。

虽然在一个 always 块中正确地安排赋值顺序，则用阻塞赋值可以实现移位寄存器时序流水线逻辑。但是，用非阻塞赋值实现同一时序逻辑要相对简单，而且，非阻塞赋值可以保证仿真和综合的结果都是一致和正确的。因此，建议大家在编写 Verilog 时序逻辑时必须要用非

阻塞赋值的方式。

14.6 阻塞赋值及一些简单的例子

许多关于 Verilog 和 Verilog 仿真的书籍都有一些使用阻塞赋值简单而且成功的例子, [例 14.13]就是一个在许多书上都出现过的关于触发器的例子。

【例 14.13】

```
module dffb (q, d, clk, rst);
    output q;
    input  d, clk, rst;
    reg    q;

    always @ (posedge clk)
        if (rst) q = 1'b0;
        else     q = d;
endmodule
```

该例虽然描述方式可行也很简单,但仍不建议用阻塞赋值来描述 D 触发器的编写代码的风格。确实采用阻塞赋值可以编写出时序逻辑的代码,如果考虑周到,也能建立正确的模型,通过仿真并综合成期望的逻辑。但是,这种做法将导致使用阻塞赋值的习惯,而在较为复杂的有多个 always 块的设计项目中,稍不注意就很可能导致竞争冒险,或者功能错误,使所设计的电路出现问题。

【例 14.14】 使用非阻塞赋值编写 D 触发器代码,这是提倡的代码风格。

```
module dffx (q, d, clk, rst);
    output q;
    input  d, clk, rst;
    reg    q;

    always @ (posedge clk)
        if (rst) q <= 1'b0;
        else     q <= d;
endmodule
```

在描述时序逻辑时,必须养成使用非阻塞赋值的习惯(无论用多个 always 块或单个 always 块描述时)。

现考察一个稍复杂一些的时序逻辑——线性反馈移位寄存器式 LFSR。

14.7 时序反馈移位寄存器建模

线性反馈移位寄存器(Linear Feedback Shift—Register, LFSR)是带反馈回路的时序逻辑。反馈回路给习惯于用顺序阻塞赋值描述时序逻辑的设计人员带来了麻烦,如[例 14.15]

所示。

【例 14.15】 用阻塞赋值实现的线性反馈移位寄存器, 实际上并不具有 LFSR 的功能。

```
module lfsrb1 (q3, clk, pre_n);
    output q3;
    input  clk, pre_n;
    reg    q3, q2, q1;
    wire   n1;

    assign n1 = q1 ^ q3;

    always @ (posedge clk or negedge pre_n)
        if (! pre_n)
            begin
                q3 = 1'b1;
                q2 = 1'b1;
                q1 = 1'b1;
            end
        else
            begin
                q3 = q2;
                q2 = n1;
                q1 = q3;
            end
    endmodule
```

除非使用中间暂存变量, 否则用[例 14.15]所示的赋值是不可能实现反馈逻辑的。

有的人可能会想到将这些赋值语句组成单行等式(见[例 14.16])来避免使用中间变量。如果逻辑再复杂一些, 单行等式是难以编写和调试的。这种方法一般不推荐使用。

【例 14.16】 用阻塞赋值描述的线性反馈移位寄存器, 其功能虽然正确, 但模型的含义较难理解。

```
module lfsrb2 (q3, clk, pre_n);
    output q3;
    input  clk, pre_n;
    reg    q3, q2, q1;

    always @ (posedge clk or negedge pre_n)
        if (! pre_n) {q3,q2,q1} = 3'b111;
        else      {q3,q2,q1} = {q2,(q1 ^ q3),q3};
    endmodule
```

如果将[例 14.15]和[例 14.16]中的阻塞赋值用非阻塞赋值代替, 如[例 14.17]和[例 14.18]所列程序, 则仿真结果都和 LFSR 的功能相一致。

【例 14.17】 用非阻塞语句描述的 LFSR, 不但功能正确, 也可综合成正确的电路。

```
module lfsrn1 (q3, clk, pre_n);
```

```
    output q3;
```

```
    input clk, pre_n;
```

```
    reg q3, q2, q1;
```

```
    wire n1;
```

```
    assign n1 = q1 ^ q3;
```

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

```
        if (! pre_n)
```

```
            begin
```

```
                q3 <= 1'b1;
```

```
                q2 <= 1'b1;
```

```
                q1 <= 1'b1;
```

```
            end
```

```
        else
```

```
            begin
```

```
                q3 <= q2;
```

```
                q2 <= n1;
```

```
                q1 <= q3;
```

```
            end
```

```
    endmodule
```

【例 14.18】 用非阻塞语句描述的 LFSR, 不但功能正确, 也可综合成正确的电路。

```
module lfsrn2 (q3, clk, pre_n);
```

```
    output q3;
```

```
    input clk, pre_n;
```

```
    reg q3, q2, q1;
```

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

```
        if (! pre_n) {q3, q2, q1} <= 3'b111;
```

```
        else {q3, q2, q1} <= {q2, (q1 ^ q3), q3};
```

```
    endmodule
```

从上面介绍的移位寄存器的例子以及 LFSR 的例子, 建议使用非阻塞赋值实现时序逻辑, 而用非阻塞赋值语句实现锁存器也是最为安全的。因此, 有以下原则:

原则 1 时序电路建模时, 用非阻塞赋值。

原则 2 锁存器电路建模时, 用非阻塞赋值。

14.8 组合逻辑建模时应使用阻塞赋值

在 Verilog 中可以用多种方法来描述组合逻辑, 但是当用 always 块来描述组合逻辑时, 应

该用阻塞赋值。

如果 always 块中只有一条赋值语句, 使用阻塞赋值或非阻塞赋值语句都可以。但是为了养成良好的编程习惯, 应该尽量使用阻塞赋值语句来描述组合逻辑。

有些设计人员认为非阻塞赋值语句不仅可以用于时序逻辑, 也可以用于组合逻辑的描述。对于简单的组合 always 块是可以这样的, 但是当 always 块中有多个赋值语句时, 如[例 14.19]所示的 4 输入与或门逻辑, 使用没有延时的非阻塞赋值可能导致仿真结果不正确。有时需要在 always 块的入口附加敏感事件参数, 才能使仿真正确, 因而从仿真的时间效率角度看也不合算。

【例 14.19】 使用非阻塞赋值语句来描述组合逻辑, 但不建议使用这种风格。

```
module ao4 (y, a, b, c, d);
    output y;
    input a, b, c, d;
    reg y, tmp1, tmp2;

    always @ (a or b or c or d)
        begin
            tmp1 <= a & b;
            tmp2 <= c & d;
            y <= tmp1 | tmp2;
        end
endmodule
```

在[例 14.19]中, 输出 y 的值由 3 个时序语句计算得到。由于非阻塞赋值语句在 LHS 更新前计算 RHS 的值, 因此 tmp1 和 tmp2 仍是应进入该 always 块时的值, 而不是在该步仿真结束时将更新的数值。输出 y 反映的是刚进入 always 块时的 tmp1 和 tmp2 的值, 而不是在 always 块中经计算后得到的值。

【例 14.20】 使用非阻塞赋值来描述多层组合逻辑, 虽可行, 但效率不高。

```
module ao5 (y, a, b, c, d);
    output y;
    input a, b, c, d;
    reg y, tmp1, tmp2;

    always @ (a or b or c or d or tmp1 or tmp2)
        begin
            tmp1 <= a & b;
            tmp2 <= c & d;
            y <= tmp1 | tmp2;
        end
endmodule
```

[例 14.20]和[例 14.19]的唯一区别在于, tmp1 和 tmp2 加入了敏感列表中。如前所描述, 当非阻塞赋值的 LHS 数值更新时, always 块将自动触发并用最新计算出的 tmp1 和 tmp2

值更新输出 y 的值。将 tmp1 和 tmp2 加入到敏感列表中后,现在输出 y 的值是正确的。但是,一个 always 块中有多次参数传递,由此降低了仿真器的性能,只有在没有其他合理方法的情况下才考虑这样做。

只需要在 always 块中使用阻塞赋值语句就可以实现组合逻辑,这样做既简单,且仿真效果具有又快又好的 Verilog 代码风格,建议大家使用。

【例 14.21】 使用阻塞赋值实现组合逻辑是推荐使用的编码风格。

```
module ao2 (y, a, b, c, d);
    output y;
    input  a, b, c, d;
    reg    y, tmp1, tmp2;

    always @ (a or b or c or d) begin
        tmp1 = a & b;
        tmp2 = c & d;
        y = tmp1 | tmp2;
    end
endmodule
```

[例 14.21]和[例 14.19]的唯一区别是,用阻塞赋值替代了非阻塞赋值。这样做,既保证了仿真时经一次数据传递输出 y 的值是正确的,又提高了仿真效率。因此有原则 3:

原则 3 用 always 块描述组合逻辑时,应采用阻塞赋值语句。

14.9 时序和组合的混合逻辑 ——使用非阻塞赋值

将简单的组合逻辑和时序逻辑写在一起很方便。当把组合逻辑和时序逻辑写入到一个 always 块中时,应遵从时序逻辑建模的原则,使用非阻塞赋值,如[例 14.22]所示。

【例 14.22】 在一个 always 块中同时实现组合逻辑和时序逻辑。

```
module nbex2 (q, a, b, clk, rst_n);
    output q;
    input  clk, rst_n;
    input  a, b;
    reg    q;

    always @ (posedge clk or negedge rst_n)
        if (!rst_n) q <= 1'b0; //时序逻辑
        else        q <= a ^ b; //异或,为组合逻辑
endmodule
```

用两个 always 块实现以上逻辑也是可以的,一个 always 块是采用阻塞赋值的纯组合部分,另一个是采用非阻塞赋值的纯时序部分,见[例 14.23]。

【例 14.23】 将组合和时序逻辑分别写在两个 always 块中。

```
module nbex1 (q, a, b, clk, rst_n);
    output q;
    input  clk, rst_n;
    input  a, b;
    reg    q, y;

    always @(a or b)
        y = a ^ b;

    always @ (posedge clk or negedge rst_n)
        if (!rst_n) q <= 1'b0;
        else         q <= y;
endmodule
```

而将组合和时序逻辑写在 alway 块中,应坚持原则 4:

原则 4 在同一个 always 块中描述时序和组合逻辑混合电路时,用非阻塞赋值。

14.10 其他阻塞和非阻塞混合使用的原则

Verilog 语法并没有禁止将阻塞和非阻塞赋值自由地组合在一个 always 块里。虽然 Verilog 语法是允许这种写法,但不建议在可综合模块的编写中采用这种风格。

【例 14.24】 在 always 块中同时使用阻塞和非阻塞赋值的例子(应尽量避免使用这种风格的代码,在可综合模块中应严禁使用)。

```
module ba_nba2 (q, a, b, clk, rst_n);
    output q;
    input  a, b, rst_n;
    input  clk;
    reg    q;

    always @ (posedge clk or negedge rst_n) begin; ff
        reg tmp;
        if (!rst_n) q <= 1'b0;
        else begin
            tmp = a & b;
            q <= tmp;
        end
    end
endmodule
```

[例 14.24]可以得到正确的仿真和综合结果,因为阻塞赋值和非阻塞赋值操作的不是同一个变量。虽然这种方法是可行的,但并不建议使用。

【例 14.25】 对同一变量既进行阻塞赋值,又进行非阻塞赋值会产生综合错误的结果。

```
module ba_nba6 (q, a, b, clk, rst_n);
```

```
    output q;
```

```
    input a, b, rst_n;
```

```
    input clk;
```

```
    reg q, tmp;
```

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

```
        if (!rst_n) q = 1'b0; //对 q 进行阻塞赋值
```

```
        else begin
```

```
            tmp = a & b;
```

```
            q <= tmp; //对 q 进行非阻塞赋值
```

```
        end
```

```
    endmodule
```

[例 14.25]在仿真时,其结果通常是正确的,但是综合时会出错,因为对同一变量既进行阻塞赋值,又进行了非阻塞赋值。因此,必须将其改写才能成为可综合模型。

为了养成良好的编程习惯,建议采用原则 5:

原则 5 不要在同一个 always 块中同时使用阻塞和非阻塞赋值。

14.11 对同一变量进行多次赋值

在一个以上 always 块中对同一个变量进行多次赋值可能会导致竞争冒险,即使使用非阻塞赋值也可能产生竞争冒险。在[例 14.26]中,两个 always 块都对输出 q 进行赋值。由于两个 always 块执行的顺序是随机的,所以仿真时会产生竞争冒险。

【例 14.26】 使用非阻塞赋值语句,由于两个 always 块对同一变量 q 赋值而产生竞争冒险的程序。

```
module badcode1 (q, d1, d2, clk, rst_n);
```

```
    output q;
```

```
    input d1, d2, clk, rst_n;
```

```
    reg q;
```

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

```
        if (!rst_n) q <= 1'b0;
```

```
        else q <= d1;
```

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

```
        if (!rst_n) q <= 1'b0;
```

```
        else q <= d2;
```

```
    endmodule
```

当综合工具(如 Synopsys)读到[例 14.25]的代码时,将产生以下警告信息:

Warning: In design'badcode1', there is 1 multiple-driver

net with unknown wired-logic type.

如果忽略这个警告,继续编译[例 14.26],则会产生两个触发器输出到一个两输入与门。其综合级前仿真与综合后仿真的结果不完全一致。因此,应建议采用原则 6:

原则 6 严禁在多个 always 块中对同一个变量赋值。

14.12 常见的对于非阻塞赋值的误解

1. 非阻塞赋值和 \$ display

误解 1:“使用 \$ display 命令不能用来显示非阻塞语句的赋值”。

事实是:非阻塞语句的赋值在所有的 \$ display 命令执行以后才更新数值。

【例 14.27】

```
module display_cmds;
    reg a;
    initial $ monitor("\$ monitor: a = %b", a);
    initial begin
        $ strobe ("\$ strobe : a = %b", a);
        a = 0;
        a <= 1;
        $ display ("\$ display: a = %b", a);
        #1 $ finish;
    end
endmodule
```

以下 3 条语句是非阻塞赋值和 \$ display 模块的仿真结果,这说明 \$ display 命令的执行是安排在活动事件队列中,但排在非阻塞赋值数据更新事件之前。

```
$ display: a = 0
$ monitor: a = 1
$ strobe : a = 1
```

2. #0 延时赋值

误解 2:“#0 延时把赋值强制到仿真时间步的末尾”。

事实是:#0 延时将赋值事件强制加入停止运行事件队列中。

【例 14.28】

```
module nb_schedule1;
    reg a, b;
    initial begin
```

```

a = 0;
b = 1;
a <= b;
b <= a;

$ monitor ("%0dns: \$ monitor: a=%b b=%b", $stime, a, b);
$ display ("%0dns: \$ display: a=%b b=%b", $stime, a, b);
$ strobe ("%0dns: \$ strobe : a=%b b=%b\n", $stime, a, b);
#0 $ display ("%0dns: #0 : a=%b b=%b", $stime, a, b);

#1 $ monitor ("%0dns: \$ monitor: a=%b b=%b", $stime, a, b);
$ display ("%0dns: \$ display: a=%b b=%b", $stime, a, b);
$ strobe ("%0dns: \$ strobe : a=%b b=%b\n", $stime, a, b);
$ display ("%0dns: #0 : a=%b b=%b", $stime, a, b);

#1 $ finish;
end
endmodule

```

以下8条语句是#0延时赋值模块的仿真结果,这说明#0延时命令在非阻塞赋值事件发生前,在停止运行事件队列中执行。

```

0ns: $ display : a=0 b=1
0ns: #0 : a=0 b=1
0ns: $ monitor: a=1 b=0
0ns: $ strobe : a=1 b=0

1ns: $ display : a=1 b=0
1ns: #0 : a=1 b=0
1ns: $ monitor: a=1 b=0
1ns: $ strobe : a=1 b=0

```

#0延时赋值建议遵循原则7:

原则7 用\$strobe系统任务来显示,应该用非阻塞赋值的变量值。

3. 对同一变量进行多次非阻塞赋值

误解3:“在Verilog语法标准中未定义,可在同一个always块中对某同一变量进行多次非阻塞赋值”。

事实是:Verilog标准定义了在同一个always块中,可对某同一变量进行多次非阻塞赋值,但在多次赋值中,只有最后一次赋值对该变量起作用。

参考IEEE 1364-1995 Verilog标准中关于决定论的内容如下:

“非阻塞赋值按照语句的顺序执行,请看下例:

```

initial begin
  a <= 0;

```

```
a <= 1;
end
```

执行该模块时,有两个非阻塞赋值更新事件加入到非阻塞赋值更新队列。以前的规则要求将非阻塞赋值更新事件按照它们在源文件的顺序加入队列,这便要求按照事件在源文件中的顺序,将事件从队列中取出并执行。因此,在仿真第一步结束的时刻,变量 a 被设置为 0,然后为 1。”

结论:最后一个非阻塞赋值决定了变量的值。

小结

本章中所有的原则归纳如下:

- (1) 原则 1:时序电路建模时,用非阻塞赋值。
- (2) 原则 2:锁存器电路建模时,用非阻塞赋值。
- (3) 原则 3:用 always 块写组合逻辑时,采用阻塞赋值。
- (4) 原则 4:在同一个 always 块中同时建立时序和组合逻辑电路时,用非阻塞赋值。
- (5) 原则 5:在同一个 always 块中不要同时使用非阻塞赋值和阻塞赋值。
- (6) 原则 6:不要在多个 always 块中为同一个变量赋值。
- (7) 原则 7:用 \$strobe 系统任务来显示用非阻塞赋值的变量值。
- (8) 原则 8:在赋值时不要使用 #0 延迟。

结论:遵循以上原则,有助于正确的编写可综合硬件,并且可以消除 90%~100% 在仿真时可能产生的竞争冒险现象。

思考题

1. 用带电平敏感列表触发条件的 always 块表示组合逻辑时,应该用哪一种赋值?
2. 用带时钟沿触发条件的 always 块表示时序电路时,应该用哪一种赋值?
3. 为什么不能在多个 always 块中为同一变量赋值?
4. 为什么不能用 \$display 系统任务来显示用非阻塞赋值的变量值?
5. \$strobe 和 \$display 这两个显示用系统任务有什么不同?各用于什么场合?
6. 仿真器在处理阻塞和非阻塞赋值操作队列过程中有什么不同?
7. 为什么在可综合 Verilog 模块的设计中,必须注意并遵守本章的 8 条原则?

第15章 较复杂时序逻辑电路设计实践

概 述

在前面14章中,已经学习了用Verilog设计数字系统的基本概念和方法。这些概念和方法只通过阅读不能完全明白,只有通过大量的上机实际操作才能逐渐掌握。在前面几章中,虽然也学习了不少例子,但因为它们都很简单,比较容易理解,不足以反映这种设计方法的优点。在本章中将通过两个稍微复杂一些的设计实例来说明如何用这种方法有序地完成具有一定难度的设计项目。

【例15.1】一个简单的状态机设计——序列检测器。

序列检测器是时序数字电路设计中经典的教学范例。下面我们将用Verilog HDL语言来描述、仿真并实现它。

序列检测器的逻辑功能:序列检测就是将一个指定的序列从数码流中识别出来。本例中,将设计一个“10010”序列的检测器。设X为数码流输入,Z为检出标记输出,高电平表示“发现指定序列”,低电平表示“没有发现指定序列”。考虑码流为“110010010000100101…”,则如表15.1所列。

表15.1 序列检测器的逻辑功能

时钟	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
X	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	1	...
Z	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	...

在时钟2~6,码流X中出现指定序列“10010”,对应输出Z在第6个时钟变为高电平“1”,表示“发现指定序列”。同样地,在时钟13~17,码流X中再次出现指定序列“10010”,Z输出“1”。注意,在时钟5~9还有一次检出,但它是与第一次检出的序列重叠的,即前者的前面两位同时也是后者的最后两位。

根据以上逻辑功能描述,可以分析得出如图15.1所示状态转换。

其中状态A~E表示5位序列“10010”按顺序正确地出现在码流中。考虑到序列重叠的可能,转换图中还有状态F、G。另外,电路的初始状态设为IDLE。

从上面的分析,可以编写出如下Verilog HDL程序如下。

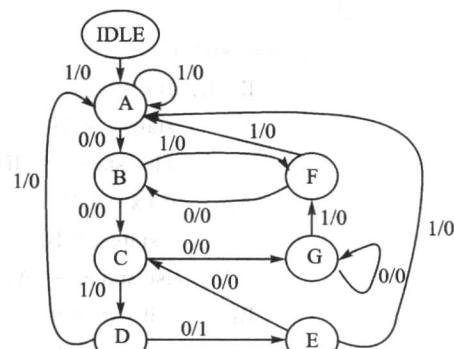


图15.1 状态转换

```

//----- 文件名:seqdet.v -----
/*
 * * * 模块功能:本模块能对串行输入的数据流进行检测,只要发现 10010 码型会立即输出一个
 * * * 高位的电平。
 * * * 本模块是 RTL 级可综合模块,已通过综合后门级仿真。
 */
module seqdet( x, z, clk, rst );
    input x,clk, rst;
    output z;

    reg [2:0] state; //状态寄存器
    wire z;

    parameter IDLE = 3'd0,
              A = 3'd1,
              B = 3'd2,
              C = 3'd3,
              D = 3'd4,
              E = 3'd5,
              F = 3'd6,
              G = 3'd7;

    assign z = (state == D && x == 0) ? 1 : 0; //状态为 D 时又收到了 0,表明 10010 收到应有输出 Z 为高

    always @(posedge clk or negedge rst)
        if(!rst)
            begin
                state <= IDLE;
            end
        else
            casex(state)
                IDLE: if(x == 1)
                    state <= A; //用状态变量记住高电平(x == 1)来过
                    else state <= IDLE; //输入的是低电平,不符合要求,所以状态保留不变
                A:   if(x == 0)
                    state <= B; //用状态变量记住第 2 位正确低电平(x == 0)来过
                    else state <= A; //输入的是高电平,不符合要求,所以状态保留不变
                B:   if(x == 0)
                    state <= C; //用状态变量记住第 3 位正确低电平(x == 0)来过
                    else state <= F; //输入的是高电平,不符合要求,记住只有 1 位对过
                C:   if(x == 1)
                    state <= D; //用状态变量记住第 4 位正确高电平(x == 1)来过
                    else state <= G; //输入的是低电平,不符合要求,记住没有 1 位

```

```

//曾经对过

D:      if(x==0)
        state<=E;           //用状态变量记住第5位正确低电平(x==0)来过
else     state<=A;           //输入的是高电平,不符合要求,记住只1位对过
        //回到状态 A

E:      if(x==0)
        state<=C;           //用状态变量记住100曾经来过,此状态为 C
else     state<=A;           //输入的是高电平,只有1位正确,该状态是 A

F:      if(x==1)
        state<=A;           //输入的是高电平,只有1位正确,该状态是 A
else
        state<=B;           //输入的是低电平,已有2位正确,该状态是 B

G:      if(x==1)
        state<=F;           //输入的又是高电平,只有1位正确,记该状态 F
else state<=B;           //输入的是低电平,已有2位正确,该状态是 B或 G

default: state<=IDLE;
endcase
endmodule
----- seqdet.v 文件的结束 -----

```

为了验证其正确性,接着编写测试用 Verilog 程序如下:

```

----- 测试文件名:t.v -----
`timescale 1ns/1ns
`define halfperiod 20

module t;
reg clk, rst;
reg [23:0] data;
wire z,x;
assign x=data[23];

initial
begin
    clk = 0;
    rst = 1;
    #2 rst = 0;
    #30 rst = 1;           //复位信号
    data= 20'b1100_1001_0000_1001_0100; //码流数据
    #(halfperiod * 1000) $stop;           //运行 500 个时钟周期后停止仿真
end

```

```

always #(halfperiod) clk=~clk; //时钟信号
always @ (posedge clk)          //移位输出码流
    #2 data={data[22:0],data[23]};
seqdet m (.x(x), .z(z), .clk(clk), .rst(rst)); //调用序列检测器模块
endmodule // 测试模块的结束

```

其中,X 码流的产生,采用了移位寄存器的方式,以方便更改测试数据。综合后门级 Verilog 网表仿真结果如图 15.2 所示。

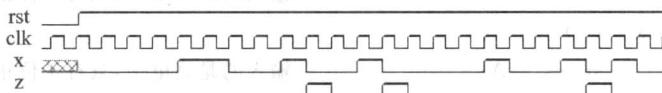


图 15.2 Verilog 网表仿真结果

从波形中可以看到,程序代码正确地完成了所要设计的逻辑功能。另外,seqdet.v 的编写,采用了可综合的 Verilog HDL 风格,它可以通过综合器的综合转换为 FPGA 或 ASIC 网表,再通过布局布线工具在 FPGA 或 ASIC 上实现。

【例 15.2】并行数据流转换为一种特殊串行数据流模块的设计。

设计两个可综合的电路模块:第一个模块(M1)能把 4 位的并行数据转换为符合以下协议的串行数据流,数据流用 scl 和 sda 两条线传输,sclk 为输入的时钟信号,data[3:0]为输入数据,ack 为 M1 请求 M0 发新数据信号。第二个模块(M2)能把串行数据流内的信息接收到,并转换为相应 16 条信号线的高电平,即若数据为 1,则第一条线路为高电平,数据为 n,则第 N 条线路为高电平。M0 为测试用信号模块。该模块接收 M1 发出的 ack 信号,并产生新的测试数据 data[3:0],如图 15.3 所示。

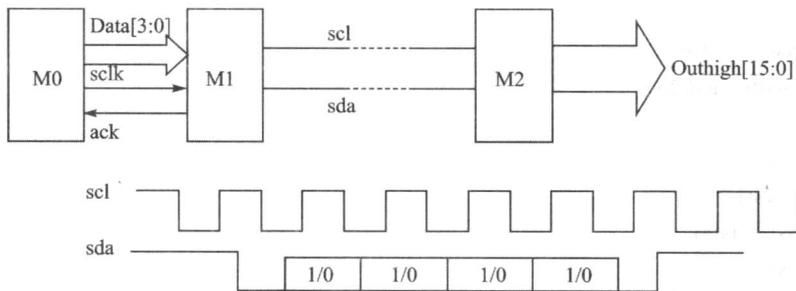


图 15.3 可综合模块结构及波形

通信协议:scl 为不断输出的时钟信号,如果 scl 为高电平时,sda 由高变低时刻,串行数据流开始;如果 scl 为高电平时,sda 由低变高时刻,串行数据结束。sda 信号的串行数据位必须在 scl 为低电平时变化,若变为高则为 1,否则为零。

描述 M1 模块的 Verilog 代码如下:

```

//-----ptosda.v 文件开始-----
// **** 模块功能:按照设计要求,把输入的 4 位并行数据转换为协议要求的串行数据流,并

```

```

*** 由 scl 和 sda 配合输出。
*** 本模块为 RTL 可综合模块,已通过综合后门级网表仿真。
***** module ptsosda (rst,sclk,ack,scl,sda,data);
input sclk, rst;
input [3:0] data;           //并行口数据输入
output ack;                //请求新的转换数据
output scl;                //串行时钟输出
output sda;                //定义 sda 为单向的串行输出
//inout sda;               //定义 sda 为双向的串行总线,注意思考题 5
reg scl, link_sda;
link_sda
  ack,           //ask_for_New_data
  sdabuf;        //sdabuf
reg [3:0] databuf;
reg [7:0] state;
assign sda = link_sda? sdabuf: 1'b0;           //link_sda 控制 sdabuf 输出到串行总线上
//assign sda=link_sda? sdabuf:1'bz;             //link_sda 控制 sdabuf 输出到双向串行总线上
//注意思考题 5
parameter ready = 8'b0000_0000,
start = 8'b0000_0001, bit0 = 8'b0000_0010, bit1 = 8'b0000_0100, bit2 = 8'b0000_1000, bit3 = 8'b0000_1000, bit4 = 8'b0001_0000, bit5 = 8'b0010_0000, stop = 8'b0100_0000, IDLE = 8'b1000_0000;
always @(posedge sclk or negedge rst)           //由输入的 sclk 时钟信号产生串行输出时钟 scl
begin
  if(!rst)
    scl <= 1;
  else
    scl <= ~scl;
end
always @(posedge ack)                          //请求新数据时存入并行总线上要转换的数据
  databuf <= data;
//-----主状态机:产生控制信号,根据 databuf 中保存的数据,按照协议产生 sda 串行信号
always @(negedge sclk or negedge rst)

```

```

if (!rst)
begin
    link_sda<=0;           //把 sdabuf 与 sda 串行总线断开
    state <= ready;
    sdabuf<= 1;
    ack<=0;                //请求新数据置 0
end
else begin
    case(state)
        ready: if (ack)           //并行数据已经到达
            begin
                link_sda<=1;      //把 sdabuf 与 sda 串行总线连接
                state <= start;
            end
        else                      //并行数据尚未到达
            begin
                link_sda<=0;      //把 sda 总线让出,若前面把 sda 定义成双向串行总线,
                //则此时 sda 可作为输入
                state <= ready;
                ack<=1;              //请求新数据信号置 1
            end
        start: if (scl && ack)     //产生 sda 的开始信号
            begin
                sdabuf<=0;      //在 sda 连接的前提下,输出开始信号
                state <= bit1;
            end
            else state <= start;
        bit1: if (!scl)             //在 scl 为低电平时送出最高位 databuf[3]
            begin
                sdabuf<=databuf[3];
                state <= bit2;
                ack<=0;
            end
            else state <= bit1;
        bit2: if (!scl)             //在 scl 为低电平时送出次高位 databuf[2]
            begin
                sdabuf<=databuf[2];
                state <= bit3;
            end
            else state <= bit2;
        bit3: if (!scl)             //在 scl 为低电平时送出次低位 databuf[1]
            begin
                sdabuf<=databuf[1];
                state <= bit4;
            end
            else state <= bit3;
    endcase
end

```

```

bit4: if (! scl)           //在 scl 为低电平时送出最低位 databuf[0]
begin
    sdabuf<=databuf[0];
    state <= bit5;
end
else state <= bit4;

bit5: if (! scl)           //为产生结束信号做准备,先把 sda 变为低
begin
    sdabuf<=0;
    state <= stop;
end
else state <= bit5;

stop: if (scl)             //在 scl 为高时,把 sda 由低变高产生结束信号
begin
    sdabuf<=1;
    state <= IDLE;
end
else state <= stop;

IDLE: begin
    link_sda <= 0; // 把 sdabuf 与 sda 串行总线脱开
    state <= ready;
end

default: begin
    link_sda <= 0;
    sdabuf<=1;
    state <= ready;
end
endcase
end
endmodule
//-----ptosda.v 文件结束-----

```

描述 M2 模块的 Verilog 代码如下：

```

//-----out16hi.v 文件开始-----
// **** 模块功能:按照协议接收串行数据,进行处理并按照数据值在相应位输出高电平
// **** 本模块为 RTL 可综合模块,已通过综合后门级网表仿真
// ****
module out16hi (scl,sda,outhigh);
input scl,sda;           //串行数据输入
output [15:0]outhigh;    //根据输入的串行数据设置高电平位

```

```

reg [5:0] mstate /* synthesis preserve */;
//本模块的主状态
// /* synthesis preserve */为综合指令,使综合器布局布线后仍能保留状态信号,以便于观察,
//不同的综合器综合指令并不相同,必须参考综合器的技术资料才能掌握
reg [3:0] pdata,
          pdatabuf;           //记录串行数据位时,用寄存器和最终数据寄存器
reg [15:0]outhigh;           //输出位寄存器
reg StartFlag,
     EndFlag;            //数据开始和结束标志

always @ (negedge sda)
begin
  if (scl) begin
    begin
      StartFlag <= 1;           //串行数据开始标志
      end
    end
    else if (EndFlag)
      StartFlag<=0;
  end
end

always @ (posedge sda)
if (scl)
begin
  EndFlag <= 1;           //串行数据结束标志
  pdatabuf <= pdata;       //把收到的4位数据存入寄存器
end
else
  EndFlag <= 0;           //数据接收还没有结束

parameter ready=6'b000_0000,
         sbit0=6'b000_0001,
         sbit1=6'b000_0010,
         sbit2=6'b000_0100,
         sbit3=6'b000_1000,
         sbit4=6'b010_0000;

always@ (pdatabuf)           //把收到的数据变为相应位的高电平
begin
  case(pdatabuf)
    4'b0001: outhigh =16'b0000_0000_0000_0001;
    4'b0010: outhigh =16'b0000_0000_0000_0010;
    4'b0011: outhigh =16'b0000_0000_0000_0100;
    4'b0100: outhigh =16'b0000_0000_0000_1000;
  endcase
end

```

```

4'b0101: outhigh = 16'b0000_0000_0001_0000;
4'b0110: outhigh = 16'b0000_0000_0010_0000;
4'b0111: outhigh = 16'b0000_0000_0100_0000;
4'b1000: outhigh = 16'b0000_0000_1000_0000;
4'b1001: outhigh = 16'b0000_0001_0000_0000;
4'b1010: outhigh = 16'b0000_0010_0000_0000;
4'b1011: outhigh = 16'b0000_0100_0000_0000;
4'b1100: outhigh = 16'b0000_1000_0000_0000;
4'b1101: outhigh = 16'b0001_0000_0000_0000;
4'b1110: outhigh = 16'b0010_0000_0000_0000;
4'b1111: outhigh = 16'b0100_0000_0000_0000;
4'b0000: outhigh = 16'b1000_0000_0000_0000;

      endcase
    end
  end

always @ (posedge scl) //在检测到开始标志后,每次 scl 正跳变沿时接收数据,共 4 位
if(StartFlag)
  case (mstate)
    sbit0: begin
      if(mstate <= sbit1);
      pdata[3] <= sda;
      $display("I am in sdabit0");
    end

    sbit1: begin
      mstate <= sbit2;
      pdata[2] <= sda;
      $display("I am in sdabit1");
    end

    sbit2: begin
      mstate <= sbit3;
      pdata[1] <= sda;
      $display("I am in sdabit2");
    end

    sbit3: begin
      mstate <= sbit4;
      pdata[0] <= sda;
      $display("I am in sdabit3");
    end

    sbit4: begin
  end

```

```

        mstate <= sbit0;
        $display("I am in sdastop");
    end
    default:   mstate <= sbit0;

endcase
else  mstate <= sbit0;
endmodule
//-----out16hi.v 文件结束-----

```

描述 M0 模块的 Verilog 代码如下：

```

//----- sigdata.v 文件的开始 -----
/*
 * * * 模块功能：本模块产生测试信号对设计中的模块进行测试。
 * * * 本模块只用于测试，不能通过综合转换为电路。
 */
`timescale 1ns/1ns
`define halfperiod 50
module sigdata (rst,sclk,data,ask_for_data);
    output rst;                                //复位信号
    output [3:0] data;                          //输出的数据信号
    output sclk;                               //输出的时钟信号
    input ask_for_data;                        //从并串转换器来的请求数据信号
    reg rst,sclk;
    reg [3:0] data;

initial
begin
    rst=1;
    # 10rst=0;
    #( `halfperiod * 2 +3)  rst=1;
end

initial                                //寄存器变量初始化
begin
    sclk = 0;
    data = 0;
    #( `halfperiod * 1000) $ stop;
end

always #(`halfperiod)  sclk = ~sclk;      //产生第一个模块需要的输入时钟
//每次请求新数据信号的正跳变沿，等一段时间后将输出数据增加 1
always @(posedge ask_for_data)

```

```

begin
# ('halfperiod/2 + 3) data = data + 1;
end
endmodule
//----- sigdata.v 结束 -----

```

描述顶层模块的 Verilog 代码如下：

```

//----- 文件名 top.v -----
/*
 * 模块功能：对所设计的两个可综合模块 ptosda 和 out16hi 进行联合测试。
 * 观察 ptosda 模块能否正确地把并行数据转换成符合协议要求
 * 的串行码流；串行码流能否通过 out16hi 模块的处理输出符合
 * 设计要求的信号。本模块是为教学需要专门设计的。为了使
 * 大家容易理解接口功能，作了许多简化，因此无实用价值。
 */

/*
 * 模块说明：本模块还可以用于综合或布局布线后的电路网表模块的测试。
 * 做后仿真时，把包括的文件改为布局布线后的门级电路网表文件，即由
 * 综合工具生成的 ptosda.vm 和 out16hi.vm 文件，或布局布线工具产生
 * 的 ptosda.vo 和 out16hi.vo 文件。为了能使门级电路网表模块进行仿真，
 * 有时还需要包括一个布线所用的 FPGA 或 ASIC 基本元件仿真库模块。
 */

'timescale 1ns/1ns
`include "sigdata.v"
`include "ptosda.v"
`include "out16hi.v"

/*
 * 可用综合后产生的门级 Verilog 网表(netlist)文件和布局布线后产生的带延迟参数
 * 的 Verilog 网表(netlist)文件来代替上面两个文件。分别进行门级后仿真和布线后
 * 仿真。这两种文件的扩展名不同，但都是 Verilog 门级模块。 */
module top;
    wire [3:0] data;
    wire sclk;
    wire scl;
    wire sda;
    wire rst;
    wire [15:0] outhigh;

    sigdata m0 (.rst(rst),.sclk(sclk),.data(data),.ask_for_data(ack));
    ptosda m1 (.rst(rst),.sclk(sclk),.ack(ack),.scl(scl),
               .sda(sda),.data(data));
    out16hi m2 (.scl(scl),.sda(sda),.outhigh(outhigh));

```

```
endmodule
```

```
//-----全面测试文件 top.v 结束-----
```

说明:以上两个程序的编程、仿真、综合和后仿真在 PC WINDOWS(98,NT,2000,XP)操作系统及 ModelSim、Synplify、MaxplusII 和 QuartusII 等环境下通过测试。

注意:以上程序中,out16hi 模块布局布线后仿真如果要输出正确结果,必须使 ptosda 模块的输出 sda 的高阻状态变为一个固定的输出后才有可能。

图 15.4 为综合后的门级网表的仿真波形图。图中数字用 Hex 表示,sda 信号中短处为高阻状态,outhigh 启始一段为不定态。

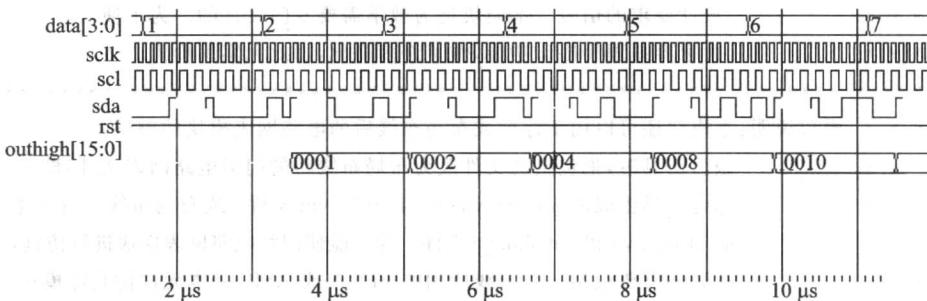


图 15.4 综合后的门级网表的仿真波形

小结

通过以上两个例题,可以看到有限状态机在数字逻辑电路中的作用。用状态变量来记住曾经发生过的事情,这些曾发生过的事情对于电路下一时钟的操作有非常重要的作用。在[例 15.1]中必须记住 10010 是否按照时钟节拍来到过,或者已经有几位正确地输入。只有这样才能正确地控制 Z 的输出。在[例 15.2]中,无论 M1 和 M2 模块的设计都必须用状态变量记住目前所处的状态,才能正确地控制输入和输出。可综合模块的设计必须在电路总体结构明确的情况下,用状态机写出控制的节拍和步骤后才能进行。状态机的编写不可能一下子就完善,存在一些问题是必然的。我们可以通过仿真调试逐步改正状态机控制中的不完善,直到正确无误。

思考题

1. 在计算机上对[例 15.1]进行 RTL 级仿真和综合后的门级 Verilog 网表仿真。观测两种层次的仿真输出波形有什么不同,试着说明为什么出现不同。分析实际电路的波形应该如何?
2. 把[例 15.1]改写成能对 111000101 序列进行检测。在接下去的检测中不再考虑已经检出序列中的任何位。编写完整的 Verilog 程序,并进行综合后的门级 Verilog 网表仿真。当出现以上序列时,让 Z 的高电平维持两个节拍。

3. 把[例15.2]改写成能对两位并行数据处理的电路。把两位并行数据转换为符合协议的串行数据,来控制4条输出线的电平。编写完整的Verilog程序,并进行综合后的门级Verilog网表仿真,验证设计的正确性。

4. 把[例15.2]改写成能对三位并行数据处理的电路。把三位并行数据转换为符合协议的串行数据,来控制8条线的电平。但要求并串转换模块能在rst信号有效后,发出ack信号到信号源模块,然后由信号源模块发出数据(data[2:0]),编写完整的Verilog程序,并进行综合后的门级Verilog网表仿真,验证设计的正确性,并编写出实验总结报告。

5. [例15.2]中的out16hi模块布局布线后仿真如果要输出正确结果,为什么必须使从上一个模块ptosda输出的sda高阻状态变为一个固定的输出后才有可能得到结果,否则没有正确的电平输出;而RTL级仿真和综合后的Verilog网表(即out16hi.v)仿真却能得到正确的答案?请以该例仿真运行中发现的问题写出进行布线后仿真必须注意的几个重要因素。

第 16 章 复杂时序逻辑电路设计实践

概 述

在前面各章的基础上,我们对比较复杂的数字电路设计过程有了一定程度的了解。经过上机操作练习,掌握了如何通过仿真调试来改进源代码,使其完全符合设计要求。在本章中将对一个简化的实际工程设计过程进行认真细致的分析,以更深入地学习数字逻辑电路系统设计的核心技术。通过一个相对简单的工程实例可以对复杂的数字电路的设计方法和过程有更深入的认识。

下面我们将介绍一个经过实际运行、验证并可综合到各种 FPGA 和 ASIC 工艺的串行 EEPROM 读写器件的设计过程,列出了所有有关的 Verilog HDL 程序。这个器件能把并行数据和地址信号转变为串行 EEPROM 能识别的串行码,并把数据写入相应的地址,或根据并行的地址信号从 EEPROM 相应的地址读取数据并把相应的串行码转换成并行的数据放到并行地址总线上。当然,还需要有相应的读信号或写信号和应答信号配合才能完成以上的操作。

16.1 二线制 I²C CMOS 串行 EEPROM 的简单介绍

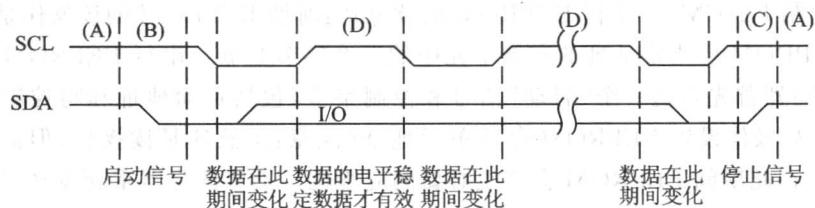
二线制 I²C CMOS 串行 EEPROM AT24C02/4/8/16 是一种采用 CMOS 工艺制成的串行可用电擦除可编程随机读写存储器。串行 EEPROM 一般具有两种写入方式,一种是字节写入方式,还有一种是页写入方式,允许在一个写周期内同时对一个字节到一页的若干字节进行编程写入。一页的大小取决于芯片内页寄存器的大小,不同公司的同一种型号存储器内的页寄存器可能是不一样的。为了程序的简单起见,在这里只编写串行 EEPROM 的一个字节的写入和读出方式的 Verilog HDL 的行为模型代码,串行 EEPROM 读写器的 Verilog HDL 模型也只是字节读写方式的可综合模型,对于页写入和读出方式,建议读者可以参考有关书籍,自己改写串行 EEPROM 的行为模型和串行 EEPROM 读写器的可综合模型,以检查是否真正掌握了本章的内容。

16.2 I²C 总线特征介绍

I²C(Inter Integrated Circuit)双向二线制串行总线协议定义为:只有在总线处于“非忙”状态时,数据传输才能开始。在数据传输期间,只要时钟线为高电平,数据线都必须保持稳定,否则数据线上的任何变化都被当作“启动”或“停止”信号。图 16.1 是被定义的总线状态。

以下介绍 A,B,C,D 段的工作状态。

(1) 总线非忙状态(A 段):该段内的数据线(SDA)和时钟线(SCL)都保持高电平。

图 16.1 I²C 双向二线制串行总线

(2) 启动数据传输(B段):当时钟线(SCL)为高电平状态时,数据线(SDA)由高电平变为低电平的下降沿被认为是“启动”信号。只有出现“启动”信号后,其他的命令才有效。

(3) 停止数据传输(C段):当时钟线(SCL)为高电平状态时,数据线(SDA)由低电平变为高电平的上升沿被认为是“停止”信号。随着“停止”信号的出现,所有的外部操作都结束。

(4) 数据有效(D段):在出现“启动”信号以后,在时钟线(SCL)为高电平状态时,数据线是稳定的,这时数据线的状态就是要传送的数据。数据线(SDA)上数据的改变必须在时钟线为低电平期间完成,每位数据占用一个时钟脉冲。每个数据传输都是由“启动”信号开始,结束于“停止”信号。

(5) 应答信号:每个正在接收数据的 EEPROM 在接到一个字节的数据后,通常需要发出一个应答信号。而每个正在发送数据的 EEPROM 在发出一个字节的数据后,通常需要接收一个应答信号。EEPROM 读写控制器必须产生一个与这个应答位相联系的额外的时钟脉冲。在 EEPROM 的读操作中,EEPROM 读写控制器对 EEPROM 完成的最后一个字节不产生应答位,但是应该给 EEPROM 一个结束信号。

16.3 二线制 I²C CMOS 串行 EEPROM 的读写操作

(1) EEPROM 的写操作(字节编程方式):所谓 EEPROM 的写操作(字节编程方式)就是通过读写控制器把一个字节数据发送到 EEPROM 中指定地址的存储单元。其过程如下:EEPROM 读写控制器发出“启动”信号后,紧跟着送 4 位 I²C 总线器件特征编码 1010 和 3 位 EEPROM 芯片地址/页地址 XXX,以及写状态的 R/W 位(=0)到总线上。这一字节表示在接收到被寻址的 EEPROM 产生的一个应答位后,读写控制器将跟着发送 1 个字节的 EEPROM 存储单元地址和要写入的 1 个字节数据。EEPROM 在接收到存储单元地址后,又一次产生应答位,使读写控制器才发送数据字节,并把数据写入被寻址的存储单元。EEPROM 再一次发出应答信号,读写控制器收到此应答信号后,便产生“停止”信号。AT24C02/4/8/16 字节写入帧格式如图 16.2 所示。



图 16.2 AT24C02/4/8/16 字节写入帧格式

(2) 二线制 I²C CMOS 串行 EEPROM 的读操作: 所谓 EEPROM 的读操作是通过读写控制器读取 EEPROM 中指定地址的存储单元中的一个字节数据。串行 EEPROM 的读操作分两步进行: 读写器首先发送一个“启动”信号和控制字节(包括页面地址和写控制位)到 EEPROM, 再通过写操作设置 EEPROM 存储单元地址(注意: 虽然这是读操作, 但需要先写入地址指针的值), 在此期间 EEPROM 会产生必要的应答位。接着读写器重新发送另一个“启动”信号和控制字节(包括页面地址和读控制位 R/W = 1), EEPROM 收到后发出应答信号, 然后, 要寻址存储单元的数据就从 SDA 线上输出。读操作有 3 种: 读当前地址存储单元的数据, 读指定地址存储单元的数据, 读连续存储单元的数据。在这里只介绍读指定地址存储单元数据的操作。读指定地址存储单元数据的帧格式如图 16.3 所示。

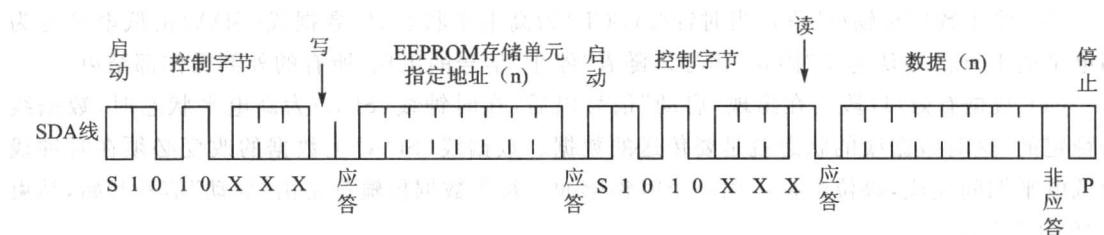


图 16.3 AT24C02/4/8/16 读指定地址存储单元的数据帧格式

16.4 EEPROM 的 Verilog HDL 程序

要设计一个串行 EEPROM 读写器件, 不仅要编写 EEPROM 读写器件的可综合 Verilog HDL 的代码, 而且要编写相应的测试代码以及 EEPROM 的行为模型。EEPROM 的读写电路及其测试电路如图 16.4 所示。

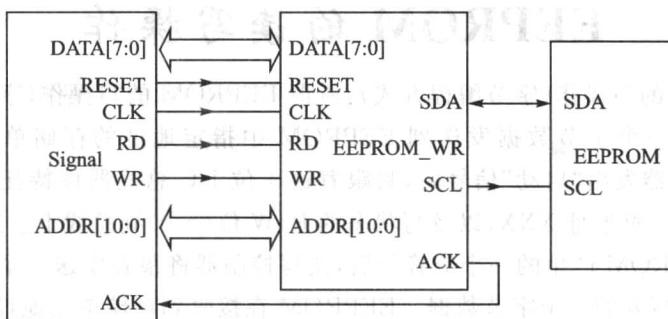


图 16.4 EEPROM 读写电路及其测试电路

(1) EEPROM 的行为模型: 为了设计一个电路, 首先要设计一个 EEPROM 的 Verilog HDL 模型。而设计这样一个模型需要仔细地阅读和分析 EEPROM 器件的说明书, 因为 EEPROM 不是要设计的对象, 而是验证设计对象所需要的器件。所以, 只需设计一个 EEPROM 的行为模型, 而不需要可综合风格的模型, 这就大大简化了设计过程。下面的 Verilog HDL 程序就是这个 EEPROM(AT24C02/4/8/16) 能完成一个字节数据读写的部分行为模型, 请读者查阅 AT24C02/4/8/16 说明书, 并对照 Verilog HDL 程序理解设计的要点。

这里只对在操作中用到的信号线进行模拟, 对于没有用到的信号线就略去了。对 EEPROM

ROM 用于基本总线操作的引脚 SCL 和 SDA 说明如下: SCL 为串行时钟端, 这个信号用于对输入和输出数据的同步, 而写入串行 EEPROM 的数据用其上升沿同步, 输出数据用其下降沿同步; SDA 为串行数据(/地址)输入/输出总线端。EEPROM 的行为模型如下:

```

/*
***** 模块名称:EEPROM      文件名:eprom.v *****
模块功能:用于模拟真实的 EEPROM(AT 24C02/4/8/16) 的随机读写功能。对于符合 AT 24C02/4/8/16 要求的 scl 和 sda 随机读/写信号能根据 I2C 协议, 分析其含义并进行相应的读/写操作。
模块说明:本模块为行为模块,不可综合为门级网表。而且本模块为教学需求做了许多简化,由于功能不完整,不能用做商业目的。
***** */

`timescale 1ns/1ns
`define timeslice 100
module EEPROM(scl, sda);
    input scl; //串行时钟线
    inout sda; //串行数据线
    reg out_flag; //SDA 数据输出的控制信号
    reg[7:0] memory[2047:0];
    reg[10:0] address;
    reg[7:0] memory_buf;
    reg[7:0] sda_buf; //SDA 数据输出寄存器
    reg[7:0] shift; //SDA 数据输入寄存器
    reg[7:0] addr_byte; //EEPROM 存储单元地址寄存器
    reg[7:0] ctrl_byte; //控制字寄存器
    reg[1:0] State; //状态寄存器
    integer i;

//-----参数部分-----
parameter r7 = 8'b10101111, w7 = 8'b10101110, //main7
r6 = 8'b10101101, w6 = 8'b10101100, //main6
r5 = 8'b10101011, w5 = 8'b10101010, //main5
r4 = 8'b10101001, w4 = 8'b10101000, //main4
r3 = 8'b10100111, w3 = 8'b10100110, //main3
r2 = 8'b10100101, w2 = 8'b10100100, //main2
r1 = 8'b10100011, w1 = 8'b10100010, //main1
r0 = 8'b10100001, w0 = 8'b10100000; //main0
//-----赋值部分-----
assign sda = (out_flag == 1) ? sda_buf[7] : 1'bz;
//-----寄存器和存储器初始化-----
initial
begin

```

```

//----- 初始化 -----
addr_byte    = 0;
ctrl_byte    = 0;
out_flag     = 0;
sda_buf      = 0;
State         = 2'b00;
memory_buf   = 0;
address       = 0;
shift         = 0;
for(i=0;i<=2047; i=i+1)
    memory[i]=0;
end

//----- 启动信号 -----
always @ (negedge sda)
    if(scl == 1 )
        begin
            State = State + 1; //注意：ModelSim6.1以上版本，认为从高阻态到1是负跳变沿
            if(State == 2'b11)
                disable write_to_eeprom;
        end

//----- 主状态机 -----
always @(posedge sda)
    if (scl == 1 )           //停止操作
        stop_W_R;
    else
        begin
            casex(State)
                2'b01
/* **** 注意：老版书上为 2'b01，因为 ModelSim 6.0 以下版本不认为从高阻态到1是跳变沿，而 **
** ModelSim 6.1 以上版本，在 RTL 仿真时，认为从高阻态到1是负跳变沿，所以写 EEPROM **
** 操作从 2'b10 状态开始。而做布线后仿真时，ModelSim 6.1 以上版本，并不认为高阻态 **
** 到1是跳变沿，所以应该将进入状态 2'b10，改为与老版书一致，即 2'b01。 **
** 不同的仿真工具在处理高阻和不定态时有所不同，必须引起设计者的注意。 */
                begin
                    read_in;
                    if(ctrl_byte==w7 || ctrl_byte==w6 || ctrl_byte==w5 ||
                       || ctrl_byte==w4 || ctrl_byte==w3 || ctrl_byte==w2 ||
                       || ctrl_byte==w1 || ctrl_byte==w0)
                        begin
                            State = 2'b10;

```

```
        write_to_eeprom; //写操作
    end
else
    State = 2'b00;
end

2'b11:
    read_from_eeprom; //读操作

default:
    State=2'b00;

endcase
end                                //主状态机结束

//----- 操作停止 -----
task stop_W_R;
begin
    State = 2'b00;                  //状态返回为初始状态
    addr_byte = 0;
    ctrl_byte = 0;
    out_flag = 0;
    sda_buf = 0;
end
endtask

//----- 读进控制字和存储单元地址 -----
task read_in;
begin
    shift_in(ctrl_byte);
    shift_in(addr_byte);
end
endtask

//----- EEPROM 的写操作-----
task write_to_eeprom;
begin
    shift_in(memory_buf);
    address = {ctrl_byte[3:1],addr_byte};
    memory[address] = memory_buf;
    $display("eeprom-----memory[%0h]=%0h",address,memory[address]);
    State = 2'b00;                  //回到 0 状态
end
```

```

endtask

//----- EEPROM 的读操作-----
task read_from_eeprom;
begin
    shift_in(ctrl_byte);
    if(ctrl_byte==r7||ctrl_byte==r6||ctrl_byte==r5||ctrl_byte==r4
    ||ctrl_byte==r3||ctrl_byte==r2||ctrl_byte==r1||ctrl_byte==r0)
        begin
            address = {ctrl_byte[3:1],addr_byte};
            sda_buf = memory[address];
            shift_out;
            State= 2'b00;
        end
    end
endtask

//-----SDA 数据线上的数据存入寄存器,数据在 SCL 的高电平有效-----
task shift_in;
output [7:0] shift;
begin
    @ (posedge scl) shift[7] = sda;
    @ (posedge scl) shift[6] = sda;
    @ (posedge scl) shift[5] = sda;
    @ (posedge scl) shift[4] = sda;
    @ (posedge scl) shift[3] = sda;
    @ (posedge scl) shift[2] = sda;
    @ (posedge scl) shift[1] = sda;
    @ (posedge scl) shift[0] = sda;
    @ (negedge scl)
        begin
            # 'timeslice ;
            out_flag = 1;                                //应答信号输出
            sda_buf = 0;
        end
    @ (negedge scl)
        # 'timeslice out_flag = 0;
end
endtask

//-----EEPROM 存储器中的数据通过 SDA 数据线输出,数据在 SCL 低电平时变化
task shift_out;
begin

```

```

out_flag = 1; //表示应答信号输出
for(i=6;i>=0; i=i-1)
begin
  @(negedge scl); //时钟下降沿
  #`timeslice;
  sda_buf = sda_buf<<1;
end
@(negedge scl) #`timeslice sda_buf[7] = 1; //非应答信号输出
@(negedge scl) #`timeslice out_flag = 0;
end
endtask
endmodule
//-----eprom.v 文件结束-----

```

(2) EEPROM 读写器的可综合的 Verilog HDL 模型:下面的 Verilog 程序是一个可综合的 EEPROM 读写器模型,它接收来自信号源模型产生的读信号、写信号、并行地址信号和并行数据信号,并把它们转换为相应的串行信号发送到串行 EEPROM(AT24C02/4/8/16)的行为模型中去。它还发送应答信号(ACK)到信号源模型,以便让信号源来调节发送或接收数据的速度,以配合 EEPROM 读写器模型从 EEPROM 行为模型发送(写)和接收(读)数据。因为它是我们的设计对象,所以它的仿真不但要正确无误,还需要能综合成门级网表。

这个程序基本上由两部分组成:开关组合电路和控制时序电路,如图 16.5 所示。开关电路在控制时序电路的控制下,按照设计的要求有节奏地打开或闭合,这样 SDA 可以按 I²C 数据总线的格式输出或输入,使 SDA 和 SCL 一起完成 EEPROM 的读写操作。

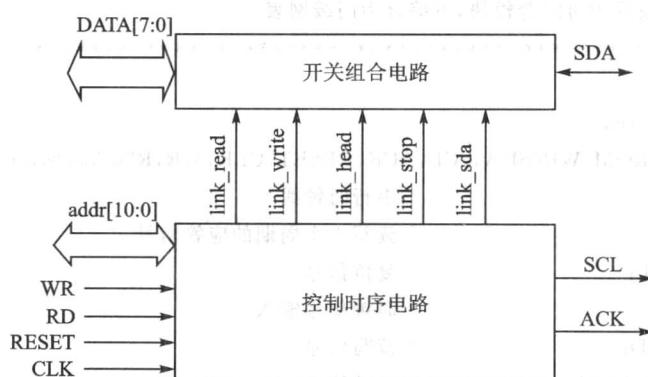


图 16.5 EEPROM 读写器的结构

电路最终用同步有限状态机(FSM)的设计方法实现。程序实则上是一个嵌套的状态机,由主状态机和从状态机通过由控制线启动的总线在不同的输入信号情况下构成不同功能的较复杂的有限状态机,这个有限状态机只有唯一的驱动时钟 CLK。根据串行 EEPROM 的读写操作时序可知,用 5 个状态时钟可以完成写操作,用 7 个状态时钟可以完成读操作。由于读写操作的状态中有几个状态是一致的,用一个嵌套的状态机即可。状态转移如图 16.6 所示。程序由一个读写大任务和若干个较小的任务所组成,其状态机采用独热编码,若需改变状态编码,只需改变程序中的 parameter 定义即可。读者可以通过模仿这一程序来编写较复杂的可

综合 Verilog HDL 模块程序。这个设计已通过后仿真，并可在 FPGA 上实现布局布线。

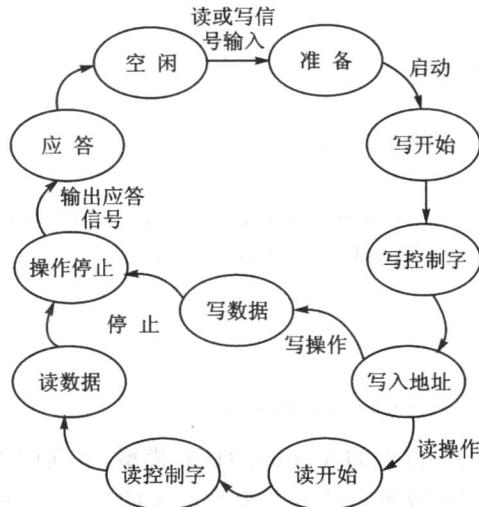


图 16.6 读写操作状态转移

```
/*
模块名称:EEPROM_WR 文件名:eprom_wr.v
模块功能:EEPROM 读写器根据 MCU 的并行数据、地址线和读/写控制线对 EEPROM (AT24
C02/4/8/16) 的行为模块进行随机读写操作,而且本模块为教学要求做了许多简化。
但本模块只能做随机的读写操作,只能做随机的读写操作,功能不完整,不能用做商业目的。
```

模块说明:本模块为可综合模块,可综合为门级网表。

```
'timescale 1ns/1ns
module EEPROM_WR(SDA,SCL,ACK,RESET,CLK,WR,RD,ADDR,DATA);
output SCL; //串行时钟线
output ACK; //读写一个周期的应答信号
input RESET; //复位信号
input CLK; //时钟信号输入
input WR,RD; //读写信号
input [10:0] ADDR; //地址线
inout SDA; //串行数据线
inout [7:0] DATA; //并行数据线
reg ACK;
reg SCL;
reg WF,RF; //读写操作标志
reg FF; //标志寄存器
reg [1:0] head_buf; //启动信号寄存器
reg [1:0] stop_buf; //停止信号寄存器
reg [7:0] sh8out_buf; //EEPROM 写寄存器
```

```

reg [8:0] sh8out_state;           //EEPROM 写状态寄存器
reg [9:0] sh8in_state;           //EEPROM 读状态寄存器
reg [2:0] head_state;            //启动状态寄存器
reg [2:0] stop_state;            //停止状态寄存器
reg [10:0] main_state;           //主状态寄存器
reg [7:0] data_from_rm;          //EEPROM 读寄存器
reg link_sda;                   //SDA 数据输入 EEPROM 开关
reg link_read;                  //EEPROM 读操作开关
reg link_head;                  //启动信号开关
reg link_write;                 //EEPROM 写操作开关
reg link_stop;                  //停止信号开关
wire sda1,sda2,sda3,sda4;

//-----串行数据在开关控制下有序的输出或输入-----
assign sda1 = (link_head)? head_buf[1] : 1'b0;
assign sda2 = (link_write)? sh8out_buf[7] : 1'b0;
assign sda3 = (link_stop)? stop_buf[1] : 1'b0;
assign sda4 = (sda1 | sda2 | sda3);
assign SDA = (link_sda)? sda4 : 1'bz;
assign DATA = (link_read)? data_from_rm : 8'hzz;

//-----主状态机状态定义-----
parameter
    Idle      = 11'b000000000001,
    Ready     = 11'b000000000010,
    Write_start = 11'b000000000100,
    Ctrl_write = 11'b000000010000,
    Addr_write = 11'b000000100000,
    Data_write = 11'b000001000000,
    Read_start = 11'b000010000000,
    Ctrl_read = 11'b000100000000,
    Data_read = 11'b001000000000,
    Stop      = 11'b010000000000,
    Ackn     = 11'b100000000000,
    SDA       = 11'b000000000000,
    DATA      = 8'hzz;

//-----并行数据串行输出状态-----
sh8out_bit7 = 9'b000000001,
sh8out_bit6 = 9'b000000010,
sh8out_bit5 = 9'b000000100,
sh8out_bit4 = 9'b000001000,
sh8out_bit3 = 9'b000010000,
sh8out_bit2 = 9'b000100000,
sh8out_bit1 = 9'b001000000,

```

```

        sh8out_bit0 = 9'b010000000,
        sh8out_end  = 9'b100000000;
//-----串行数据并行输出状态-----
parameter    sh8in_begin  = 10'b0000000001,
              sh8in_bit7   = 10'b0000000010,
              sh8in_bit6   = 10'b0000000100,
              sh8in_bit5   = 10'b0000001000,
              sh8in_bit4   = 10'b0000010000,
              sh8in_bit3   = 10'b0000100000,
              sh8in_bit2   = 10'b0001000000,
              sh8in_bit1   = 10'b0010000000,
              sh8in_bit0   = 10'b0100000000,
              sh8in_end    = 10'b1000000000,
//-----启动状态-----
head_begin  = 3'b001,
head_bit    = 3'b010,
head_end    = 3'b100,
//-----停止状态-----
stop_begin  = 3'b001,
stop_bit    = 3'b010,
stop_end    = 3'b100,
//-----主状态机程序-----
parameter YES      = 1,
          NO       = 0,
//-----产生串行时钟 SCL,为输入时钟的 2 分频-----
always @ (negedge CLK)
  if(RESET)
    SCL <= 0;
  else
    SCL <= ~SCL;
//-----主状态机程序-----
always @ (posedge CLK)
  if(RESET)
    begin
      link_read  <= NO;
      link_write <= NO;
      link_head  <= NO;
      link_stop  <= NO;
      link_sda   <= NO;
      ACK        <= 0;
      RF         <= 0;
    end
  else
    begin
      if(link_head)
        begin
          if(link_stop)
            link_stop <= NO;
          else
            link_head <= NO;
        end
      end
    end
end

```

```

WF      <= 0;
FF      <= 0;
main_state <= Idle;
end
else
begin
  casex(main_state)
    when(Idle)
      begin
        link_read  <= NO;
        link_write <= NO;
        link_head   <= NO;
        link_stop   <= NO;
        link_sda    <= NO;
        if(WR)
          begin
            WF <= 1;
            main_state <= Ready ;
          end
        else if(RD)
          begin
            RF <= 1;
            main_state <= Ready ;
          end
        else
          begin
            WF <= 0;
            RF <= 0;
            main_state <= Idle;
          end
      end
    Ready:
      begin
        link_read  <= NO;
        link_write <= NO;
        link_stop   <= NO;
        link_head   <= YES;
        link_sda    <= YES;
        head_buf[1:0] <= 2'b10;
        stop_buf[1:0] <= 2'b01;
        head_state  <= head_begin;
        FF         <= 0;
        ACK        <= 0;
      end
  endcase
end

```

```

    main_state      <= Write_start;
end

Write_start:
if(FF == 0)
    shift_head;
else
begin
    sh8out_buf[7:0] <= {1'b1,1'b0,1'b1,1'b0,ADDR[10:8],1'b0};
    link_head      <= NO;
    link_write     <= YES;
    FF            <= 0;
    sh8out_state   <= sh8out_bit6;
    main_state     <= Ctrl_write;
end

Ctrl_write:
if(FF == 0)
    shift8_out;
else
begin
    sh8out_state <= sh8out_bit7;
    sh8out_buf[7:0] <= ADDR[7:0];
    FF            <= 0;
    main_state     <= Addr_write;
end

Addr_write:
if(FF == 0)
    shift8_out;
else
begin
    FF <= 0;
    if(WF)
        begin
            sh8out_state <= sh8out_bit7;
            sh8out_buf[7:0] <= DATA;
            main_state     <= Data_write;
        end
    if(RF)
        begin
            head_buf       <= 2'b10;
            head_state    <= head_begin;
            main_state    <= Read_start;
        end
end

```

```

Data_write:
    if(FF == 0)
        shift8_out;
    else
        begin
            stop_state      <= stop_begin;
            main_state      <= Stop;
            link_write      <= NO;
            FF              <= 0;
        end

Read_start:
    if(FF == 0)
        shift_head;
    else
        begin
            sh8out_buf     <= {1'b1,1'b0,1'b1,1'b0,ADDR[10:8],1'b1};
            link_head      <= NO;
            link_sda       <= YES;
            link_write      <= YES;
            FF              <= 0;
            sh8out_state   <= sh8out_bit6;
            main_state      <= Ctrl_read;
        end

Ctrl_read:
    if(FF == 0)
        shift8_out;
    else
        begin
            link_sda      <= NO;
            link_write      <= NO;
            FF              <= 0;
            sh8in_state   <= sh8in_begin;
            main_state      <= Data_read;
        end

Data_read:
    if(FF == 0)
        shift8in;
    else
        begin
            link_stop      <= YES;
            link_sda       <= YES;
            stop_state      <= stop_bit;
        end

```

```

          FF           <= 0;
          main_state   <= Stop;
      end

Stop:
if(FF == 0)
    shift_stop;
else
begin
    ACK           <= 1;
    FF            <= 0;
    main_state    <= Ackn;
end

Ackn:
begin
    ACK           <= 0;
    WF            <= 0;
    RF            <= 0;
    main_state    <= Idle;
end

default:   main_state <= Idle;
endcase
end
//-----串行数据转换为并行数据任务-----
task shift8in;
begin
casex(sh8in_state)
sh8in_begin:
    sh8in_state           <= sh8in_bit7;
sh8in_bit7: if(SCL)
begin
    data_from_rm[7]     <= SDA;
    sh8in_state         <= sh8in_bit6;
end
else
    sh8in_state           <= sh8in_bit7;
sh8in_bit6: if(SCL)
begin
    data_from_rm[6]     <= SDA;
    sh8in_state         <= sh8in_bit5;
end
else
    sh8in_state           <= sh8in_bit6;
end

```

```

sh8in_bit5: if(SCL)
begin
    data_from_rm[5] <= SDA;
    sh8in_state <= sh8in_bit4;
end
else
    sh8in_state <= sh8in_bit5;

sh8in_bit4: if(SCL)
begin
    data_from_rm[4] <= SDA;
    sh8in_state <= sh8in_bit3;
end
else
    sh8in_state <= sh8in_bit4;

sh8in_bit3: if(SCL)
begin
    data_from_rm[3] <= SDA;
    sh8in_state <= sh8in_bit2;
end
else
    sh8in_state <= sh8in_bit3;

sh8in_bit2: if(SCL)
begin
    data_from_rm[2] <= SDA;
    sh8in_state <= sh8in_bit1;
end
else
    sh8in_state <= sh8in_bit2;

sh8in_bit1: if(SCL)
begin
    data_from_rm[1] <= SDA;
    sh8in_state <= sh8in_bit0;
end
else
    sh8in_state <= sh8in_bit1;

sh8in_bit0: if(SCL)
begin
    data_from_rm[0] <= SDA;
    sh8in_state <= sh8in_end;
end
else
    sh8in_state <= sh8in_bit0;

sh8in_end: if(SCL)

```