第 3 章

Verilog HDL 语法

视频讲解

3.1 Verilog 简介

在 FPGA 开发中使用最多的语言主要有两种：VHDL 和 Verilog。VHDL 语法格式严谨，入门比较难。VHDL 最初由美国国防部创建。Verilog 对软件爱好者来说有种似曾相识的感觉，不易被爱好者排斥，初看起来还以为是 C 语言编程，其实它起源于 C 语言，语法格式灵活，容易掌握，因此被大多数 FPGA 工程师所喜爱。本章只对 Verilog 语法内容进行介绍。

3.2 数据类型

3.2.1 常量

(1) 整数：整数可以用二进制 b 或 B、八进制 o 或 O、十进制 d 或 D、十六进 h 或 H 表示，例如，8'b00001111 表示 8 位位宽的二进制整数，4'ha 表示 4 位位宽的十六进制整数。

(2) x 和 z：x 代表不定值，z 代表高阻值，例如，5'b00x11，第三位为不定值，3'b00z 表示最低位为高阻值。

(3) 下画线：在位数过长时可以用来分割位数，提高程序可读性，如 8'b0000_1111。

(4) 参数 parameter：parameter 可以用标识符定义常量，运用时只使用标识符即可，以提高程序可读性及维护性，如 parameter width=8，寄存器 reg [width-1:0] a，即定义了 8 位宽度的寄存器。

(5) 参数的传递：在一个模块中如果有定义参数，那么在其他模块调用此模块时可以传递参数，并可以修改参数，如下面程序所示，在 module 后用 #() 表示。

例如，定义如下调用模块：

```
module rom
#(
parameter depth = 15,
```

```
module top();
wire[31:0] addr ;
wire[15:0] data ;
```

```
parameter width = 8
)(
input[depth-1:0] addr,
    input[width-1:0] data,
    output result
);

endmodule
```

```
wire result;
rom
#(
.depth(32),
.width(16)
)
r1
(
.addr(addr),
.data(data),
.result(result)
);
endmodule
```

parameter 可以用于模块间的参数传递，而 localparam 仅用于本模块内使用，不能用于参数传递，而 localparam 多用于状态机状态的定义。

3.2.2 变量

变量是指程序运行时可以改变其值的量。下面主要介绍几种常用的变量。

1. wire 类型

wire 类型变量也叫网络类型变量，用于结构实体之间的物理连接，如门与门之间，不能存储值，用连续赋值语句 assign 赋值，定义为 wire［n－1:0］a，其中 n 代表位宽，如定义“wire a，assign a＝b”，是将 b 节点连接到连线 a 上。如图 3-1 所示，两个实体之间的连线即是 wire 类型变量。

2. reg 类型

reg 类型变量也称为寄存器变量，可用来存储值，必须在 always 语句里使用。其定义为 reg［n－1:0］a ，表示 n 位位宽的寄存器，如 reg［7:0］a 表示定义 8 位位宽的寄存器 a。如下所示定义了寄存器 q，生成的电路为时序逻辑，如图 3-2 所示为 D 触发器。

图 3-1 wire 连线

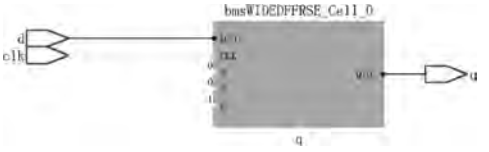

图 3-2 D 触发器

```
module top(d, clk, q);
input d;
input clk;
outputreg q;
always@(posedge clk)
begin
  q <= d;
```

```
end
endmodule
```

也可以生成组合逻辑，如数据选择器，敏感信号没有时钟，定义了 reg Mux，最终生成电路为组合逻辑，如图 3-3 所示。

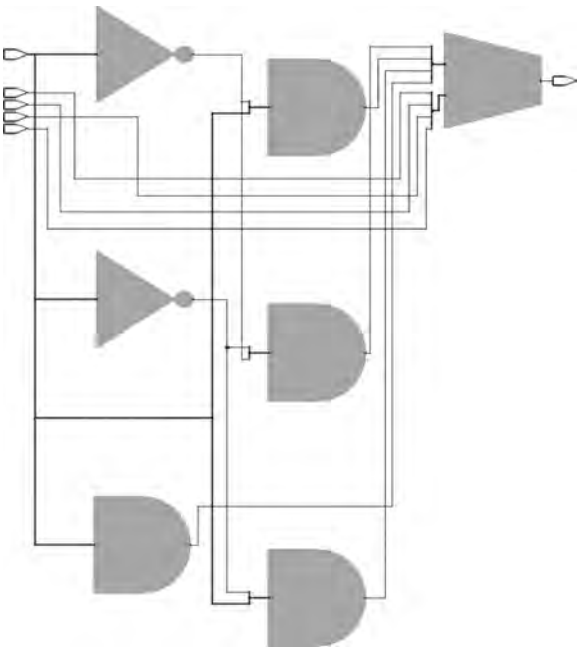

图 3-3 数据选择器

```
module top(a, b, c, d, sel, Mux);
input  a ;
input  b ;
input  c ;
input  d ;
input[1:0] sel ;
outputreg Mux ;
always@(sel or a or b or c or d)
begin
case(sel)
2'b00: Mux = a ;
2'b01: Mux = b ;
2'b10: Mux = c ;
2'b11: Mux = d ;
endcase
end
endmodule
```

3. memory 类型

可以用 memory 类型的变量来定义 RAM 和 ROM 等存储器，其结构为 reg [n－1:0] 存储器名[m－1:0]，含义为 m 个 n 位宽度的寄存器。例如，reg [7:0] ram [255:0]表示定义了 256 个 8 位寄存器，256 是存储器的深度，8 为数据宽度。

视频讲解

3.3 运算符

3.3.1 算术运算符

算术运算符包括“＋”(加法运算符)、“－”(减法运算符)、“＊”(乘法运算符)、“/”(除法运算符，如 7/3 ＝2)，“％”(取模运算符，也即求余数，如 7％3＝1，余数为 1)。

3.3.2 赋值运算符

赋值运算符分两种：“＝”阻塞赋值和“＜＝”非阻塞赋值。阻塞赋值为执行完一条赋值语句，再执行下一条，可理解为顺序执行，而且赋值是立即执行；非阻塞赋值可理解为并行执行，不考虑顺序，在 always 块语句执行完成后，变量才进行赋值。如下面的阻塞赋值：

激励文件代码如下：

```
module top(din,a,b,c,clk);
input din;
input clk;
outputreg a,b,c;
always@(posedge clk)
begin
        a = din;
        b = a;
        c = b;
end
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg din ;
reg clk ;
wire a,b,c ;
initial
begin
  din = 0;
  clk = 0;
forever
begin
#({ $ random} % 100)
    din = ~din ;
end
end
always #10 clk = ~clk ;
top t0(.din(din),.a(a),.b(b),.c(c),.clk(clk));
endmodule
```

仿真结果如图 3-4 所示，在 clk 的上升沿，a 的值等于 din，并立即赋给 b，b 的值赋给 c。

如果改为非阻塞赋值，则仿真结果如图 3-5 所示，在 clk 上升沿，a 的值没有立即赋值给 b，b 为 a 原来的值，同样，c 为 b 原来的值。

可以从两者的 RTL 视图看出明显不同，如图 3-6 和图 3-7 所示。

注意：一般情况下，在时序逻辑电路中使用非阻塞赋值，可避免仿真时出现竞争冒险现

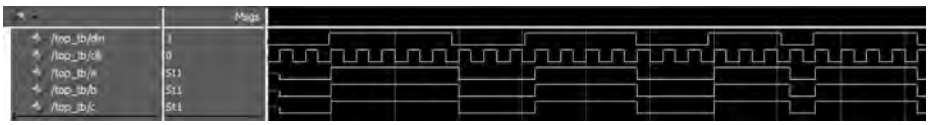

图 3-4 阻塞赋值仿真波形

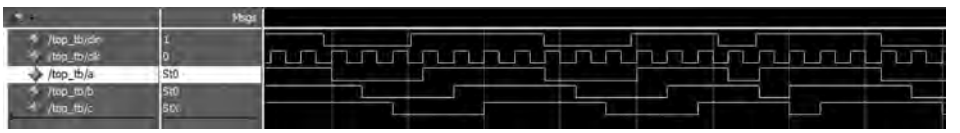

图 3-5 非阻塞赋值仿真波形

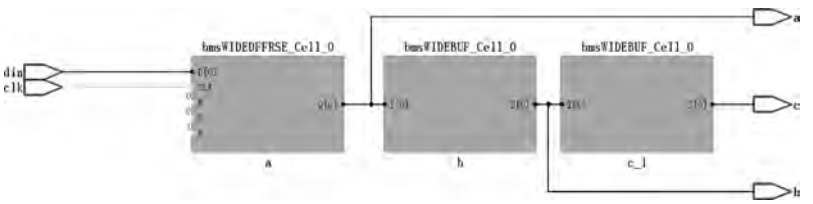

图 3-6 阻塞赋值 RTL 图

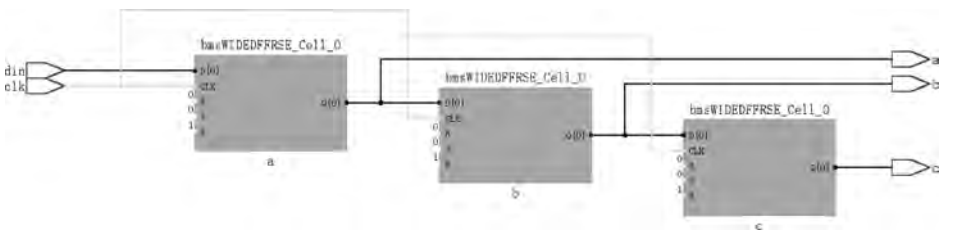

图 3-7 非阻塞赋值 RTL 图

象；在组合逻辑中使用阻塞赋值，执行赋值语句后立即改变；在 assign 语句中必须用阻塞赋值。

3.3.3 关系运算符

关系运算符用于表示两个操作数之间的关系，如 a>b、a<b，多用于判断条件，例如：

```
If (a >= b) q <= 1'b1 ;
else q <= 1'b0 ;
```

表示如果 a 的值大于或等于 b 的值，则 q 的值为 1，否则 q 的值为 0。

3.3.4 逻辑运算符

逻辑运算符包括“&&”(两个操作数逻辑与)、“‖”(两个操作数逻辑或)和“!”(单个操

作数逻辑非）。例如：if（a>b && c <d）表示条件为a>b并且c<d，if（! a）表示条件为a的值不为1，也就是0。

3.3.5 条件运算符

“?:”为条件判断，类似于if else，例如assign a = (i>8)? 1'b1:1'b0，判断i的值是否大于8，如果大于8则a的值为1，否则为0。

3.3.6 位运算符

位运算符包括“~”（按位取反）、“|”（按位或）、“^”（按位异或）、“&”（按位与）、“^~”（按位同或）。除了“~”只需要一个操作数外，其他几个都需要两个操作数，如a&b、a|b。具体应用3.4节中有讲解。

3.3.7 移位运算符

移位运算符包括“<<”左移位运算符和“>>”右移位运算符，如a<<1表示向左移1位，a>>2表示向右移2位。

3.3.8 拼接运算符

“{ }”为拼接运算符，用于将多个信号按位拼接起来，如{a[3:0],b[1:0]}，将a的低4位和b的低2位拼接成6位数据。另外，{n{a[3:0]}}表示将n个a[3:0]拼接，{n{1'b0}}表示n位的0拼接，{8{1'b0}}表示8'b0000_0000。

3.3.9 优先级

各种运算符的优先级如图3-8所示。

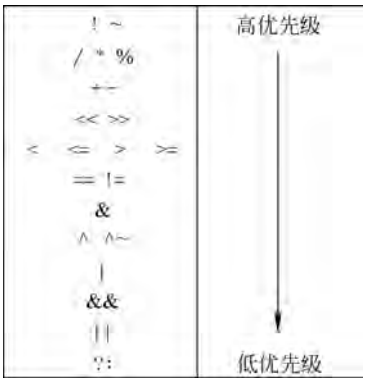

图3-8 优先级

视频讲解

3.4 组合逻辑

本节主要介绍组合逻辑，组合逻辑电路的特点是任意时刻的输出仅仅取决于输入信号，输入信号变化，输出立即变化，不依赖于时钟。

3.4.1 与门

在Verilog中以“&”表示按位与，如c=a&b，表示在a和b都等于1时结果才为1，其真值表和RTL视图如图3-9所示。

代码实现和激励文件如下：

输入		输出
a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

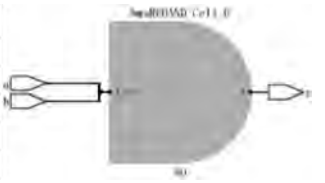

图 3-9 与门真值表及 RTL 视图

```
module top(a, b, c);
input a ;
input b ;
output c ;

assign c = a & b ;
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg a ;
reg b ;
wire c ;
initial
begin
  a = 0;
  b = 0;
forever
begin
#({$random}%100)
    a = ~a ;
#({$random}%100)
    b = ~b ;
end
end
top t0(.a(a),.b(b),.c(c));
endmodule
```

仿真结果如图 3-10 所示。

图 3-10 与门仿真波形

如果 a 和 b 的位宽大于 1,例如有定义“input [3:0] a,input [3:0]b”,那么 a&b 指 a 与 b 的对应位相与。如 a[0]&b[0]、a[1]&b[1]。

3.4.2 或门

在 Verilog 中以“|”表示按位或,如 c = a|b ,表示在 a 和 b 都为 0 时结果才为 0,其真值表和 RTL 视图如图 3-11 所示。

代码实现和激励文件如下:

```
module top(a, b, c);
input a ;
```

```
`timescale 1ns/1ns
module top_tb();
```

输入		输出
a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

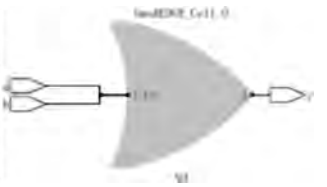

图 3-11　或门真值表与 RTL 视图

```
input b ;
output c ;

assign c = a | b ;
endmodule
```

```
reg a ;
reg b ;
wire c ;

initial
begin
  a =0;
  b =0;
forever
begin
#({ $ random} % 100)
    a = ~a ;
#({ $ random} % 100)
    b = ~b ;
end
end
top t0(.a(a),.b(b),.c(c));
endmodule
```

仿真结果如图 3-12 所示。

图 3-12　或门仿真波形

3.4.3　非门

在 Verilog 中以“~”表示按位取反，如 b＝~a，表示 b 等于 a 的相反数，其真值表和 RTL 视图如图 3-13 所示。

输入	输出
a	b
0	1
1	0

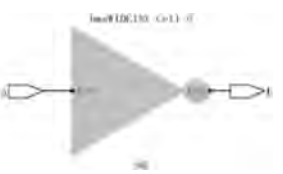

图 3-13　非门真值表及 RTL 视图

代码实现和激励文件如下：

```
module top(a, b);
input  a ;
output b ;

assign b = ~a ;
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg a ;
wire b ;

initial
begin
  a = 0;
forever
begin
#({ $ random} % 100)
    a = ~a ;
end
end
top t0(.a(a),.b(b));
endmodule
```

仿真结果如图 3-14 所示。

图 3-14 非门仿真波形

3.4.4 异或

在 Verilog 中以“^”表示异或，如 c= a^b ，表示当 a 和 b 相同时，输出为 0，其真值表和 RTL 视图如图 3-15 所示。

输入		输出
a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

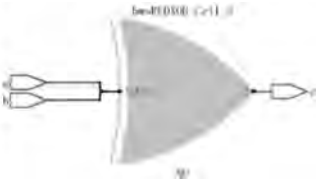

图 3-15 异或真值表及 RTL 视图

代码实现和激励文件如下：

```
module top(a, b, c);
input a ;
input b ;
output c ;

assign c = a ^ b ;
```

```
`timescale 1ns/1ns
module top_tb();
reg a ;
reg b ;
wire c ;
initial
```

```
endmodule
```

```
begin
  a = 0;
  b = 0;
forever
begin
#({$random}%100)
    a = ~a;
#({$random}%100)
    b = ~b;
end
end
top t0(.a(a),.b(b),.c(c));
endmodule
```

仿真结果如图 3-16 所示。

图 3-16　异或仿真波形

3.4.5　比较器

在 Verilog 中，比较器以大于“>”、等于“==”、小于“<”、大于或等于“>=”、小于或等于“<=”和不等于“!=”表示。以大于举例，如 c=a > b，表示如果 a 大于 b，那么 c 的值就为 1，否则为 0。其真值表和 RTL 视图如图 3-17 所示。

输入		输出
a	b	c
0	0	0
0	1	0
1	0	1
1	1	0

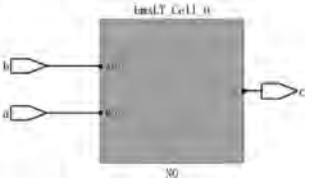

图 3-17　比较器真值表及 RTL 视图

代码实现和激励文件如下：

```
module top(a, b, c);
input a;
input b;
output c;

assign c = a > b;
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg a;
reg b;
wire c;
initial
begin
  a = 0;
  b = 0;
forever
```

```
begin
#({$random}%100)
    a = ~a ;
#({$random}%100)
    b = ~b ;
end
end
top t0(.a(a),.b(b),.c(c));
endmodule
```

仿真结果如图 3-18 所示。

图 3-18 比较器仿真波形

3.4.6 半加器

半加器和全加器是算术运算电路中的基本单元，由于半加器不考虑从低位来的进位，所以称之为半加器，sum 表示相加结果，count 表示进位。其真值表和 RTL 视图如图 3-19 所示。

输入		输出	
a	b	sum	count
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

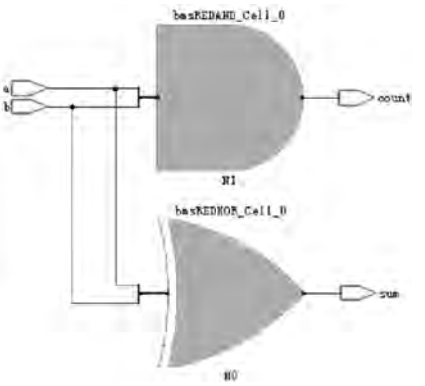

图 3-19 半加器真值表及 RTL 视图

代码实现和激励文件如下：

```
module top(a, b, sum, count);
input a ;
input b ;
output sum ;
output count ;

assign sum = a ^ b ;
assign count = a & b ;
```

```
`timescale 1ns/1ns
module top_tb();
reg a ;
reg b ;
wire sum ;
wire count ;

initial
begin
```

```
endmodule
```

```
    a = 0;
    b = 0;
forever
begin
#({$random}%100)
    a = ~a;
#({$random}%100)
    b = ~b;
end
end
top t0(.a(a),.b(b),
.sum(sum),.count(count));
endmodule
```

仿真结果如图 3-20 所示。

图 3-20　半加器仿真波形

3.4.7　全加器

全加器需要加上低位来的进位信号 cin，真值表和 RTL 视图如图 3-21 所示。

输入			输出	
cin	a	b	sum	count
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

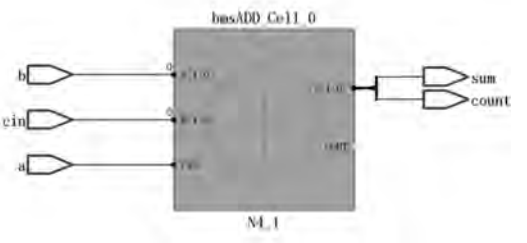

图 3-21　全加器真值表及 RTL 视图

代码实现和激励文件如下：

```
module top(cin, a, b, sum, count);
input cin ;
input a ;
input b ;
output sum ;
```

```
`timescale 1ns/1ns
module top_tb();
reg a ;
reg b ;
reg cin ;
```

```
output count ;

assign{count,sum} = a + b + cin ;

endmodule
```

```
wire sum ;
wire count ;
initial
begin
  a = 0;
  b = 0;
  cin = 0;
forever
begin
#({ $ random} % 100)
    a = ~a ;
#({ $ random} % 100)
    b = ~b ;
#({ $ random} % 100)
    cin = ~cin ;

end
end
top t0(.cin(cin),.a(a),.b(b),
.sum(sum),.count(count));
endmodule
```

仿真结果如图 3-22 所示。

图 3-22 全加器仿真波形

3.4.8 乘法器

乘法的表示也很简单,利用“*”即可,如 a * b,乘法器的 RTL 视图如图 3-23 所示。

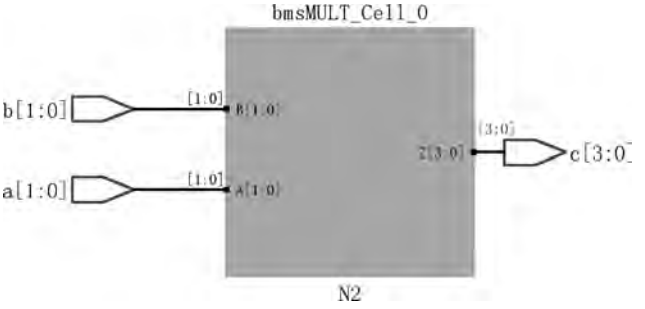

图 3-23 乘法器 RTL 视图

代码示例如下:

```
module top(a, b, c);
input[1:0] a ;
input[1:0] b ;
output[3:0] c ;

assign c = a * b ;
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg[1:0]a ;
reg[1:0]b ;
wire[3:0]c ;

initial
begin
  a = 0;
  b = 0;
forever
begin
#({$random}%100)
    a = ~a ;
#({$random}%100)
    b = ~b ;
end
end
top t0(.a(a),.b(b),.c(c));
endmodule
```

仿真结果如图 3-24 所示。

图 3-24　乘法器仿真波形

3.4.9　数据选择器

在 Verilog 中经常会用到数据选择器，通过选择信号，选择不同的输入信号输出到输出端，四选一数据选择器，sel[1:0]为选择信号，a、b、c、d 为输入信号，Mux 为输出信号。其真值表如表 3-1 所示，RTL 视图如图 3-25 所示。

表 3-1　数据选择器真值表

选择信号		输入信号				输出信号
sel[0]	sel[1]	a	b	c	d	Mux
0	0	x	x	x	x	a
0	1	x	x	x	x	b
1	0	x	x	x	x	c
1	1	x	x	x	x	d

代码实现和激励文件如下：

```
module top(a, b, c, d, sel, Mux);
```

```
`timescale 1ns/1ns
```

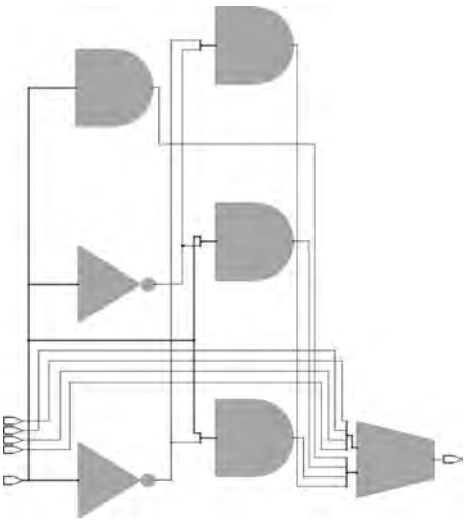

图 3-25 数据选择器 RTL 视图

```
input  a ;
input  b ;
input  c ;
input  d ;

input[1:0] sel ;

outputreg Mux ;

always@(sel or a or b or c or d)
begin
case(sel)
2'b00: Mux = a ;
2'b01: Mux = b ;
2'b10: Mux = c ;
2'b11: Mux = d ;
endcase
end

endmodule
```

```
module top_tb();
reg a ;
reg b ;
reg c ;
reg d ;
reg[1:0] sel ;
wire Mux ;
initial
begin
  a = 0;
  b = 0;
  c = 0;
  d = 0;
forever
begin
#({ $ random} % 100)
    a = { $ random} % 3;
#({ $ random} % 100)
    b = { $ random} % 3;
#({ $ random} % 100)
    c = { $ random} % 3;
#({ $ random} % 100)
    d = { $ random} % 3;
end
end
initial
begin
  sel = 2'b00;
```

```
#2000 sel = 2'b01;
#2000 sel = 2'b10;
#2000 sel = 2'b11;
end
top
t0(.a(a),.b(b),.c(c),.d(d),.sel(sel),
.Mux(Mux));
endmodule
```

仿真结果如图 3-26 所示。

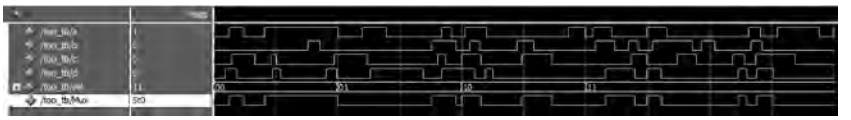

图 3-26　数据选择器仿真波形

3.4.10　3-8 译码器

3-8 译码器是一个很常用的器件，其真值表如表 3-2 所示，根据 A2、A1 和 A0 的值，得出不同的结果。其 RTL 视图如图 3-27 所示。

表 3-2　3-8 译码器真值表

输　入			输　出							
A2	A1	A0	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

代码实现和激励文件如下：

```
module top(addr, decoder);
input[2:0] addr ;
outputreg[7:0] decoder ;

always@(addr)
begin
case(addr)
3'b000: decoder = 8'b1111_1110;
3'b001: decoder = 8'b1111_1101;
3'b010: decoder = 8'b1111_1011;
```

```
`timescale 1ns/1ns
module top_tb();
reg[2:0] addr ;
wire[7:0] decoder ;

initial
begin
  addr = 3'b000;
#2000 addr = 3'b001;
#2000 addr = 3'b010;
```

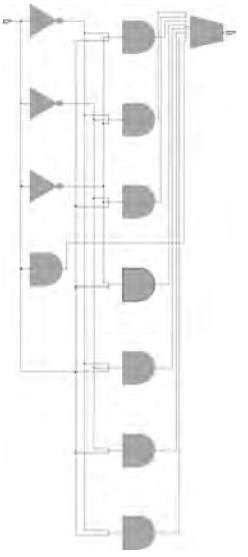

图 3-27 3-8 译码器 RTL 视图

```
3'b011: decoder = 8'b1111_0111;
3'b100: decoder = 8'b1110_1111;
3'b101: decoder = 8'b1101_1111;
3'b110: decoder = 8'b1011_1111;
3'b111: decoder = 8'b0111_1111;
endcase
end

endmodule
```

```
#2000 addr = 3'b011;
#2000 addr = 3'b100;
#2000 addr = 3'b101;
#2000 addr = 3'b110;
#2000 addr = 3'b111;
end
top
t0(.addr(addr),.decoder(decoder));
endmodule
```

仿真结果如图 3-28 所示。

图 3-28 3-8 译码器仿真结果

3.4.11 三态门

在 Verilog 中,经常会用到双向 I/O,需要用到三态输出电路,如 bio = en? din: 1'bz;,其中 en 为使能信号,用于打开关闭三态输出电路,双向 I/O 电路 RTL 视图如图 3-29 所示,如下程序介绍了怎样实现两个双向 I/O 的对接。

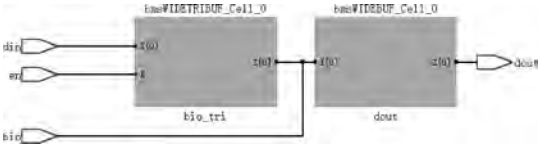

图 3-29 三态输出电路 RTL 视图

```
module top(en, din, dout, bio);
input din ;
input en ;
output dout ;
inout bio ;

assign bio = en? din :1'bz;
assign dout = bio ;

endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg en0 ;
reg din0 ;
wire dout0 ;
reg en1 ;
reg din1 ;
wire dout1 ;
wire bio ;
initial
begin
  din0 = 0;
  din1 = 0;
forever
begin
#({$random}%100)
    din0 = ~din0 ;
#({$random}%100)
    din1 = ~din1 ;
end
end
initial
begin
  en0 = 0;
  en1 = 1;
#100000
  en0 = 1;
  en1 = 0;
end
top
t0(.en(en0),.din(din0),.dout(dout0),.bi
o(bio));
top
t1(.en(en1),.din(din1),.dout(dout1),.bi
o(bio));
endmodule
```

仿真结果如图 3-30 所示，当 en0 为 0、en1 为 1 时，1 通道打开，双向 I/O bio 就等于 1 通道的 din1，1 通道向外发送数据，0 通道接收数据，dout0 等于 bio；当 en0 为 1、en1 为 0 时，0 通道打开，双向 I/O bio 就等于 0 通道的 din0，0 通道向外发送数据，1 通道接收数据，dout1 等于 bio。

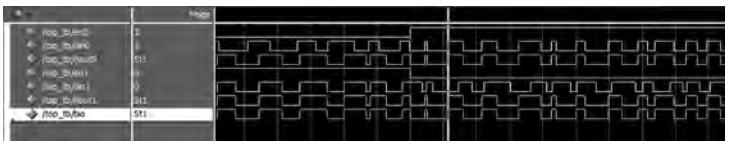

图 3-30　三态门仿真波形

3.5 时序逻辑

组合逻辑电路在逻辑功能上的特点是任意时刻的输出仅仅取决于当前时刻的输入，与电路原来的状态无关。而时序逻辑电路在逻辑功能上的特点是任意时刻的输出不仅取决于当前的输入信号，还取决于电路原来的状态。下面以一个典型的时序逻辑电路为例进行分析。

3.5.1 D触发器

D触发器在时钟的上升沿或下降沿存储数据，输出与时钟跳变之前输入信号的状态相同。

代码实现和激励文件如下：

```
module top(d, clk, q);
input d ;
input clk ;
outputreg q ;
always@(posedge clk)
begin
  q <= d ;
end
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg d ;
reg clk ;
wire q ;
initial
begin
  d = 0;
  clk = 0;
forever
begin
#({ $ random} % 100)
    d = ~d ;
end
end
always #10 clk = ~clk ;
top t0(.d(d),.clk(clk),.q(q));
endmodule
```

D触发器的RTL视图如图3-31所示。

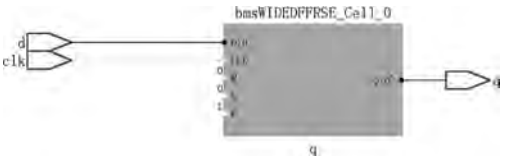

图3-31 D触发器的RTL视图

仿真结果如图3-32所示，可以看到在t0时刻时，d的值为0，则q的值也为0；在t1时刻d发生了变化，值为1，那么q相应也发生了变化，值变为1。可以看到，在t0～t1的一个

时钟周期内，无论输入信号 d 的值如何变化，q 的值是保持不变的，也就是有存储的功能，保存的值为在时钟的跳变沿时 d 的值。

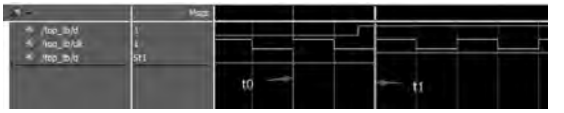

图 3-32　D 触发器仿真波形

3.5.2　两级 D 触发器

软件是按照两级 D 触发器的模型进行时序分析的，具体可以分析在同一时刻两个 D 触发器输出的数据有何不同，其 RTL 视图如图 3-33 所示。

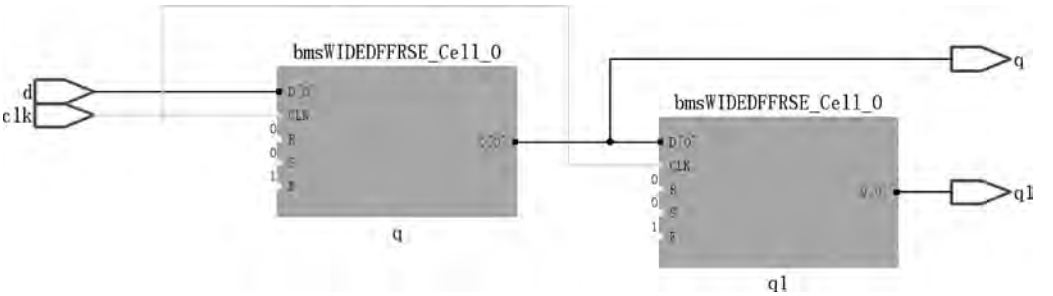

图 3-33　两级 D 触发器的 RTL 视图

代码实现和激励文件如下：

```
module top(d, clk, q, q1);
input d ;
input clk ;
outputreg q ;
outputreg q1 ;
always@(posedge clk)
begin
  q <= d ;
end
always@(posedge clk)
begin
  q1 <= q ;
end
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg d ;
reg clk ;
wire q ;
wire q1 ;
initial
begin
  d = 0;
  clk = 0;
forever
begin
#({ $random} % 100)
    d = ~d ;
end
end
always #10 clk = ~clk ;
top
t0(.d(d),.clk(clk),.q(q),.q1(q1));
endmodule
```

仿真结果如图 3-34 所示，可以看到，在 t0 时刻，d 为 0，q 输出为 0；在 t1 时刻，q 随着 d 的数据变化而变化，而此时钟跳变之前 q 的值仍为 0，那么 q1 的值仍为 0；在 t2 时刻，时钟跳变前 q 的值为 1，则 q1 的值相应为 1，q1 相对于 q 落后一个周期。

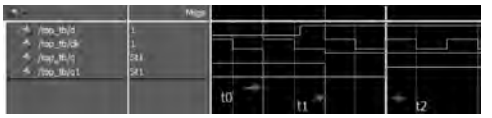

图 3-34 两级 D 触发器仿真波形

3.5.3 带异步复位 D 触发器

异步复位是指独立于时钟，一旦异步复位信号有效，就触发复位操作。这个功能在写代码时会经常用到，用于给信号复位和初始化，其 RTL 视图如图 3-35 所示。

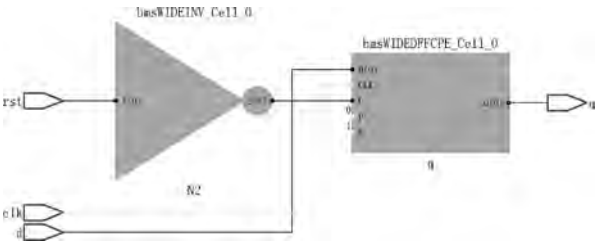

图 3-35 带异步复位 D 触发器的 RTL 视图

代码如下，注意要把异步复位信号放在敏感列表里。如果是低电平复位，即为 negedge；如果是高电平复位，则是 posedge。

```
module top(d, rst, clk, q);
input d ;
input rst ;
input clk ;
outputreg q ;
always@(posedge clk or negedge rst)
begin
if(rst == 1'b0)
    q <= 0;
else
    q <= d ;
end
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg d ;
reg rst ;
reg clk ;
wire q ;

initial
begin
  d = 0;
  clk = 0;
forever
begin
#({$random} % 100)
    d = ~d ;
end
end
initial
begin
```

```
    rst = 0;
  #200 rst = 1;
end
always #10 clk = ~clk ;
top
t0(.d(d),.rst(rst),.clk(clk),.q(q));
endmodule
```

仿真结果如图 3-36 所示，可以看到，在复位信号之前，虽然输入信号 d 数据有变化，但由于正处于复位状态，输入信号 q 始终为 0，在复位之后 q 的值就正常了。

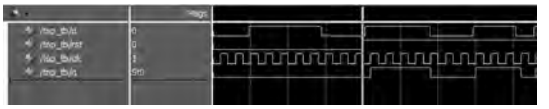

图 3-36　带异步复位 D 触发器仿真波形

3.5.4　带异步复位同步清零 D 触发器

前面讲到异步复位独立于时钟操作，而同步清零则是在同步时钟信号下操作的，当然也不仅限于同步清零，也可以是其他的同步操作，其 RTL 视图如图 3-37 所示。

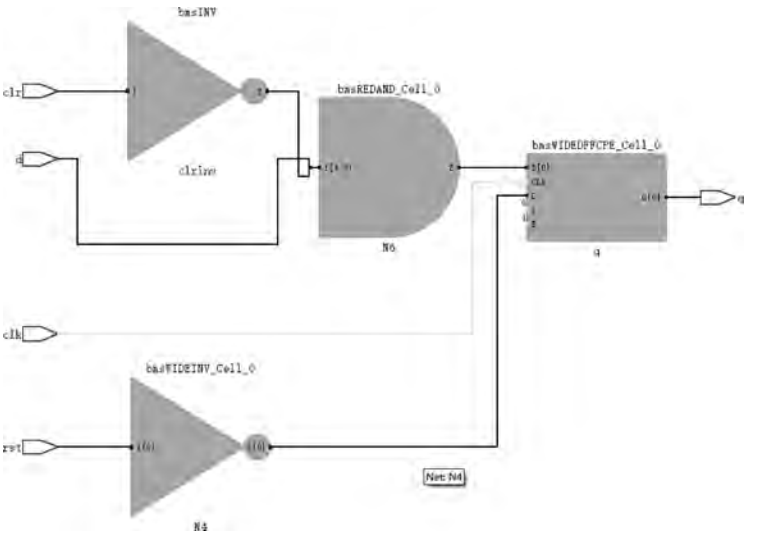

图 3-37　带异步复位同步清零 D 触发器的 RTL 视图

不同于异步复位，同步操作不能把信号放到敏感列表里。代码如下：

```
module top(d, rst, clr, clk, q);
input d ;
input rst ;
input clr ;
input clk ;
```

```
`timescale 1ns/1ns
module top_tb();
reg d ;
reg rst ;
reg clr ;
```

```
output reg q ;
always@(posedge clk or negedge rst)
begin
if(rst == 1'b0)
    q <= 0;
else if(clr == 1'b1)
    q <= 0;
else
    q <= d ;
end
endmodule
```

```
reg clk ;
wire q ;
initial
begin
  d = 0;
  clk = 0;
forever
begin
#({$random} % 100)
    d = ~d ;
end
end
initial
begin
  rst = 0;
  clr = 0;
#200 rst = 1;
#200 clr = 1;
#100 clr = 0;
end
always #10 clk = ~clk ;
top
t0(.d(d),.rst(rst),.clr(clr),.clk(clk),
.q(q));
endmodule
```

仿真结果如图 3-38 所示，可以看到 clr 信号拉高后，q 没有立即清零，而是在下一个 clk 上升沿之后执行清零操作，也就是 clr 同步于 clk。

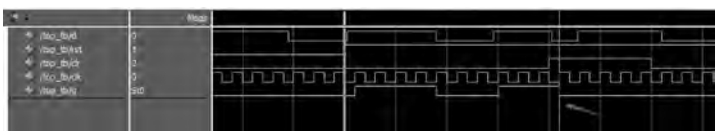

图 3-38 带异步复位同步清零 D 触发器仿真波形

3.5.5 移位寄存器

移位寄存器是指在每个时钟脉冲到来时，向左或向右移动一位，移位寄存器结构如图 3-39 所示，由于 D 触发器的特性，数据输出同步于时钟边沿，每个时钟来临，每个 D 触发器的输出 Q 等于前一个 D 触发器输出的值，从而实现移位的功能。

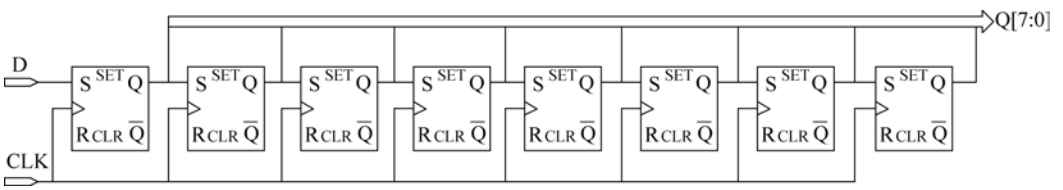

图 3-39 移位寄存器结构

代码实现和激励文件如下：

```
module top(d, rst, clk, q);
input d ;
input rst ;
input clk ;
outputreg[7:0] q ;
always@(posedge clk or negedge rst)
begin
if(rst == 1'b0)
    q <= 0;
else
    q <= {q[6:0], d};       //向左移位
//q <= {d, q[7:1]} ;        //向右移位
end
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg d ;
reg rst ;
reg clk ;
wire[7:0] q ;

initial
begin
  d = 0;
  clk = 0;
forever
begin
#({$random} % 100)
    d = ~d ;
end
end
initial
begin
  rst = 0;
#200 rst = 1;
end
always#10 clk = ~clk ;
top
t0(.d(d),.rst(rst),.clk(clk),.q(q));
endmodule
```

仿真结果如图 3-40 所示。可以看到，复位之后，数据随 clk 上升沿左移一位。

图 3-40　移位寄存器仿真波形

3.5.6　单口 RAM

单口 RAM 的写地址与读地址共用一个地址，代码如下，其中 reg [7:0] ram [63:0]定义了 64 个 8 位宽度的数据，addr_reg 可以将读地址延迟一周期之后将数据送出。

```
module top
(
input[7:0] data,
input[5:0] addr,
input wr,
input clk,
output[7:0] q
);
```

```
`timescale 1ns/1ns
module top_tb();
reg[7:0] data ;
reg[5:0] addr ;
reg wr ;
reg clk ;
wire[7:0] q ;
initial
```

```
reg[7:0] ram[63:0];
reg[5:0] addr_reg;          //地址寄存器

always@(posedge clk)
begin
if(wr)                      //写使能
    ram[addr]<= data;

  addr_reg <= addr;
end
assign q = ram[addr_reg];   //读数据
endmodule
```

```
begin
  data = 0;
  addr = 0;
  wr = 1;
  clk = 0;
end
always#10 clk = ~clk ;
always@(posedge clk)
begin
  data <= data + 1'b1;
  addr <= addr + 1'b1;
end
top t0(.data(data),
.addr(addr),
.clk(clk),
.wr(wr),
.q(q));
endmodule
```

仿真结果如图 3-41 所示。可以看到，q 的输出与写入的数据一致。

图 3-41 单口 RAM 仿真波形

3.5.7 伪双口 RAM

伪双口 RAM 的读写地址是独立的，可以随机选择写或读地址，同时进行读写操作，代码如下：

```
module top
(
input[7:0] data,
input[5:0] write_addr,
input[5:0] read_addr,
input wr,
input rd,
input clk,
outputreg[7:0] q
);

reg[7:0] ram[63:0];
reg[5:0] addr_reg;          //地址寄存器

always@(posedge clk)
begin
if(wr)                      //写使能
```

```
`timescale 1ns/1ns
module top_tb();
reg[7:0] data ;
reg[5:0] write_addr ;
reg[5:0] read_addr ;
reg wr ;
reg clk ;
reg rd ;
wire[7:0] q ;
initial
begin
  data = 0;
  write_addr = 0;
  read_addr = 0;
  wr = 0;
  rd = 0;
  clk = 0;
```

```
    ram[write_addr]<= data;
if(rd)                          //读使能
     q <= ram[read_addr];
end

endmodule
```

```
#100 wr = 1;
#20 rd = 1;
end
always#10 clk = ~clk ;
always@(posedge clk)
begin
if(wr)
begin
     data <= data + 1'b1;
     write_addr <= write_addr + 1'b1;
if(rd)
       read_addr <= read_addr + 1'b1;
end
end
top t0(.data(data),
.write_addr(write_addr),
.read_addr(read_addr),
.clk(clk),
.wr(wr),
.rd(rd),
.q(q));
endmodule
```

仿真结果如图 3-42 所示。可以看到，在 rd 有效时，对读地址进行操作，读出数据。

图 3-42　伪双口 RAM 仿真波形

3.5.8　真双口 RAM

真双口 RAM 有两套控制线和数据线，允许两个系统对其进行读写操作，代码如下：

```
module top
(
input[7:0] data_a, data_b,
input[5:0] addr_a, addr_b,
input wr_a, wr_b,
input rd_a, rd_b,
input clk,
outputreg[7:0] q_a, q_b
);

reg[7:0] ram[63:0];          //声明 ram

//端口 A
```

```
`timescale 1ns/1ns
module top_tb();
reg[7:0] data_a, data_b ;
reg[5:0] addr_a, addr_b ;
reg wr_a, wr_b ;
reg rd_a, rd_b ;
reg clk ;
wire[7:0] q_a, q_b ;
initial
begin
  data_a = 0;
  data_b = 0;
  addr_a = 0;
```

```
always@(posedge clk)
begin
if(wr_a)                    //写
begin
     ram[addr_a]<= data_a;
     q_a <= data_a ;
end
    if(rd_a)
//读
     q_a <= ram[addr_a];
end


//端口 B
always@(posedge clk)
begin
if(wr_b)                    //写
begin
     ram[addr_b]<= data_b;
     q_b <= data_b ;
end
if(rd_b)
//读
     q_b <= ram[addr_b];
end


endmodule
```

```
  addr_b = 0;
  wr_a = 0;
  wr_b = 0;
  rd_a = 0;
  rd_b = 0;
  clk = 0;
#100 wr_a = 1;
#100 rd_b = 1;
end
always#10 clk = ~clk ;
always@(posedge clk)
begin
if(wr_a)
begin
    data_a <= data_a + 1'b1;
    addr_a <= addr_a + 1'b1;
end
else
begin
     data_a <= 0;
     addr_a <= 0;
end
end
always@(posedge clk)
begin
if(rd_b)
begin
     addr_b <= addr_b + 1'b1;
end
else addr_b <= 0;
end
top
t0(.data_a(data_a),.data_b(data_b),
.addr_a(addr_a),.addr_b(addr_b
),
.wr_a(wr_a),.wr_b(wr_b),
.rd_a(rd_a),.rd_b(rd_b),
.clk(clk),
.q_a(q_a),.q_b(q_b));
endmodule
```

仿真结果如图 3-43 所示。

图 3-43 真双口 RAM 仿真波形

3.5.9 单口 ROM

ROM 是用来存储数据的，可以按照下列代码形式初始化 ROM，但用这种方法处理大容量的 ROM 时就比较麻烦，建议用 FPGA 自带的 ROM IP 核实现，并添加初始化文件。

代码实现和激励文件如下：

```
module top
(
input[3:0] addr,
input clk,
outputreg[7:0] q
);
reg[7:0] rom [15:0];          //申明 rom
always@(addr)
begin
case(addr)
4'd0: rom[addr] = 8'd15;
4'd1: rom[addr] = 8'd24;
4'd2: rom[addr] = 8'd100;
4'd3: rom[addr] = 8'd78;
4'd4: rom[addr] = 8'd98;
4'd5: rom[addr] = 8'd105;
4'd6: rom[addr] = 8'd86;
4'd7: rom[addr] = 8'd254;
4'd8: rom[addr] = 8'd76;
4'd9: rom[addr] = 8'd35;
4'd10: rom[addr] = 8'd120;
4'd11: rom[addr] = 8'd85;
4'd12: rom[addr] = 8'd37;
4'd13: rom[addr] = 8'd19;
4'd14: rom[addr] = 8'd22;
4'd15: rom[addr] = 8'd67;
endcase
end
always@(posedge clk)
begin
  q <= rom[addr];
end
endmodule
```

```
`timescale 1ns/1ns
module top_tb();
reg[3:0] addr ;
reg clk ;
wire[7:0] q ;
initial
begin
  addr = 0;
  clk = 0;
end
always#10 clk = ~clk ;
always@(posedge clk)
begin
     addr <= addr + 1'b1;
end
top t0(.addr(addr),
.clk(clk),
.q(q));
endmodule
```

仿真结果如图 3-44 所示。

图 3-44 单口 ROM 仿真波形

3.5.10 有限状态机

在 Verilog 中经常会用到有限状态机来处理相对复杂的逻辑，设定好不同的状态，根据触发条件跳转到对应的状态，在不同的状态下做相应的处理。有限状态机主要用到 always 及 case 语句。下面以一个四状态的有限状态机举例说明，如图 3-45 所示。

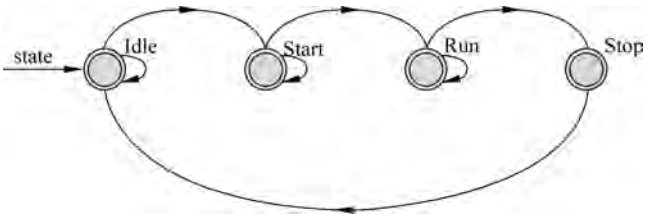

图 3-45 状态机跳转

在程序中设计了 8 位的移位寄存器。在 Idle 状态下，判断 shift_start 信号是否为高，如果为高，进入 Start 状态，在 Start 状态延迟 100 个周期，进入 Run 状态，进行移位处理，如果 shift_stop 信号有效，则进入 Stop 状态，在 Stop 状态，将 q 的值清零，再跳转到 Idle 状态。

Mealy 有限状态机的输出不仅与当前状态有关，也与输入信号有关，在 RTL 视图中会与输入信号有连接。Mealy 有限状态机的 RTL 视图如图 3-46 所示，其实现代码如下：

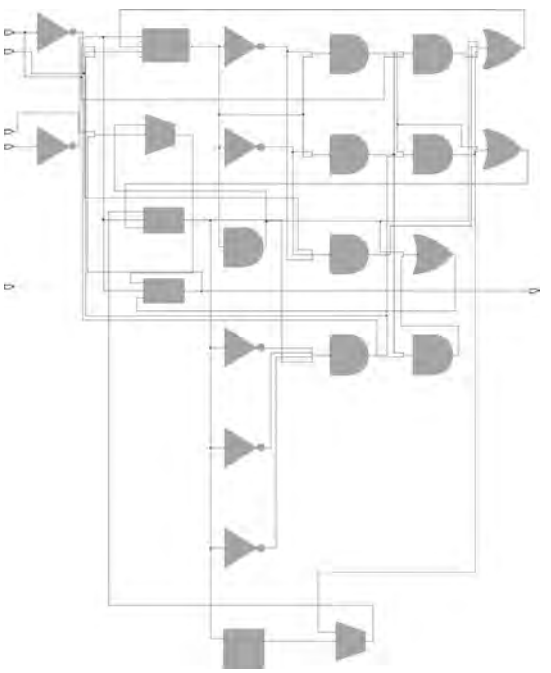

图 3-46 Mealy 有限状态机的 RTL 视图

```
module top
(
input shift_start,
input shift_stop,
input rst,
input clk,
input d,
outputreg[7:0] q
);
parameter Idle = 2'd0;                    //初始状态
parameter Start = 2'd1;                   //开始状态
parameter Run  = 2'd2;                    //运行状态
parameter Stop = 2'd3;                    //停止状态
reg[1:0] state ;
reg[4:0] delay_cnt ;                      //延迟计数
always@(posedge clk or negedge rst)
begin
if(!rst)begin
   state <= Idle ;
   delay_cnt <= 0;
   q <= 0; end
else
case(state)
    Idle :begin
if(shift_start)
state <= Start ; end
    Start :begin
if(delay_cnt == 5'd99)begin
delay_cnt <= 0;
state <= Run ; end
else
delay_cnt <= delay_cnt + 1'b1; end
    Run  :begin
if(shift_stop)
state <= Stop ;
else
q <= {q[6:0], d}; end
    Stop :begin
              q <= 0;
state <= Idle ; end
default: state <= Idle ;
endcase
end
endmodule
```

Moore 有限状态机的输出只与当前状态有关，与输入信号无关，输入信号只影响状态的改变，不影响输出，比如对 delay_cnt 和 q 的处理，只与当前状态有关。Moore 有限状态机

的 RTL 视图如图 3-47 所示，其实现代码如下：

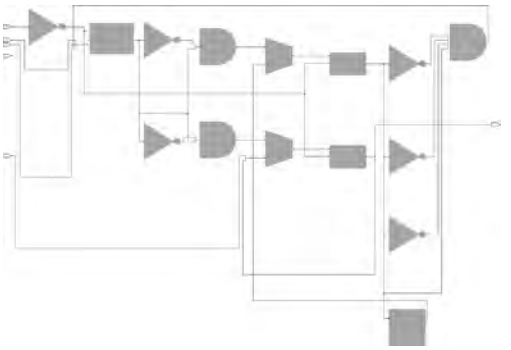

图 3-47 Moore 有限状态机的 RTL 视图

```
module top
(
input shift_start,
input shift_stop,
input rst,
input clk,
input d,
outputreg[7:0] q
);
parameter Idle = 2'd0;                    //初始状态
parameter Start = 2'd1;                   //开始状态
parameter Run  = 2'd2;                    //运行状态
parameter Stop = 2'd3;                    //停止状态
reg[1:0] current_state ;
reg[1:0] next_state ;
reg[4:0] delay_cnt ;                      //延迟计数
//第一部分:状态转换
always@(posedge clk or negedge rst)begin
if(!rst)
   current_state <= Idle ;
else
   current_state <= next_state ; end
//第二部分:组合逻辑,判断语句转换条件
always@( * )begin
case(current_state)
    Idle :begin
if(shift_start)
next_state <= Start ;
else
next_state <= Idle ; end
    Start :begin
if(delay_cnt == 5'd99)
next_state <= Run ;
else
```

```
            next_state <= Start ;
end
    Run   :begin
if(shift_stop)
next_state <= Stop ;
else
                next_state <= Run ;
end
    Stop : next_state <= Idle ;
default:next_state <= Idle ;
endcase
end
//第三部分:输出数据
always@(posedge clk or negedge rst)
begin
if(!rst)
    delay_cnt <= 0;
else if(current_state == Start)
    delay_cnt <= delay_cnt + 1'b1;
else
    delay_cnt <= 0;
end

always@(posedge clk or negedge rst)
begin
if(!rst)
    q <= 0;
else if(current_state == Run)
    q <= {q[6:0], d};
else
    q <= 0;
end
endmodule
```

上面两个程序中用到了两种方式的写法。第一种是 Mealy 状态机，采用了一段式的写法，只用了一个 always 语句，所有的状态转移，判断状态转移条件，数据输出都在一个 always 语句里，缺点是如果状态太多，则会使整段程序显得冗长。第二种是 Moore 状态机，采用了三段式的写法，状态转移用了一个 always 语句，判断状态转移条件是组合逻辑，采用了一个 always 语句，数据输出也是单独的 always 语句，这样写起来比较直观清晰，状态很多时也不会显得烦琐。

激励文件如下：

```
`timescale 1ns/1ns
module top_tb();
reg shift_start ;
reg shift_stop ;
reg rst ;
reg clk ;
```

```
reg d ;
wire[7:0] q ;
initial
begin
  rst = 0;
  clk = 0;
  d = 0;
#200 rst = 1;
forever
begin
#({ $random} % 100)
    d = ~d ;
end
end

initial
begin
  shift_start = 0;
  shift_stop = 0;
#300 shift_start = 1;
#1000 shift_start = 0;
        shift_stop = 1;
#50 shift_stop = 0;
end
always#10 clk = ~clk ;
top t0
(
.shift_start(shift_start),
.shift_stop(shift_stop),
.rst(rst),
.clk(clk),
.d(d),
.q(q)
);
Endmodule
```

仿真结果如图 3-48 所示。

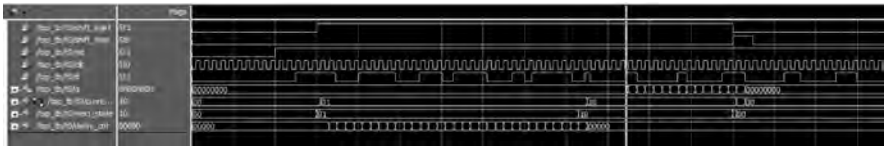

图 3-48 状态机仿真波形

3.6 总结

本章介绍了组合逻辑以及时序逻辑中常用的模块，其中有限状态机较为复杂，但经常用到，希望大家能够深入理解，在编写代码时多运用、多思考，有利于快速提升水平。