第 5 章

时序逻辑电路实例

5.1 触发器

关于触发器之前在数字逻辑电路基础的章节里向读者介绍过。本节来看如何使用 Verilog 硬件描述语言来对触发器进行建模，将以 D 触发器和带低电平复位的 D 触发器为例向读者进行讲解。

5.1.1 Verilog 实现

11min

1. D 触发器

D 触发器的 Verilog 实现，代码如下：

```
//文件路径:5.1/src/dff.v
module dff(clk,din,dout);
  input clk;
  input din;
  output reg dout;

//上升沿触发(也可以改为下降沿触发,只要改为 negedge clk 即可),将输出端 dout 的值更新为输入
//端 din 的值
  always@(posedge clk)begin
    dout <= din;
  end

endmodule
```

2. 带低电平复位的 D 触发器

带低电平复位的 D 触发器的 Verilog 实现，代码如下：

```
//文件路径:5.1/src/dff_rst.v
module dff_rst(clk,rst_n,din,dout);
  input clk;
  input rst_n;
  input din;
```

```
  output reg dout;

//上升沿触发,将输出端 dout 的值更新为输入端 din 的值,并且当 rst_n 为低电平时进行复位,即将
//输出端置 0
  always@(posedge clk)begin
    if(!rst_n)
      dout <= 1'b0;
    else
      dout <= din;
  end

endmodule
```

5.1.2 测试平台

顶层测试模块的代码如下：

```
//文件路径:5.1/sim/testbench/demo_tb.sv
module top;
  logic clk;
  logic rst_n;
  logic dff_din;
  logic dff_dout;
  logic dff_rst_dout;

  dff DUT_dff(.clk(clk),
              .din(dff_din),
              .dout(dff_dout));              //例化 D 触发器

  dff_rst DUT_dff_rst(.clk(clk),
                      .rst_n(rst_n),
                      .din(dff_din),
                      .dout(dff_rst_dout));  //例化带低电平复位的 D 触发器

  initial begin                              //产生时钟翻转信号
    clk = 0;
    forever begin
      #10;
      clk = ~clk;
    end
  end

  initial begin                              //产生复位信号
    rst_n = 0;
    #5;
    rst_n = 1;
    #250;
    rst_n = 0;
  end
```

```
  initial begin
    int random_delay;
    $display("%t -> Start!!!", $time);
    repeat(10)begin  //产生随机延迟的数据输入端 dff_din,并将输入激励打印出来
      dff_din =  $urandom_range(1,0);
      random_delay =  $urandom_range(50,0);
      $display("%t -> random delay is %0d", $time,random_delay);
      #random_delay;
    end
    #100;
    $display("%t -> Finish!!!", $time);
    $finish;
  end

endmodule : top
```

5.1.3　仿真验证

仿真结果如下：

```
0 -> Start!!!
0 -> dff_din is 1,random delay is 17
17 -> dff_din is 0,random delay is 3
20 -> dff_din is 1,random delay is 11
31 -> dff_din is 1,random delay is 10
41 -> dff_din is 1,random delay is 4
45 -> dff_din is 1,random delay is 17
62 -> dff_din is 0,random delay is 7
69 -> dff_din is 0,random delay is 11
80 -> dff_din is 0,random delay is 1
81 -> dff_din is 1,random delay is 7
188 -> Finish!!!
```

仿真波形如图 5-1 所示。

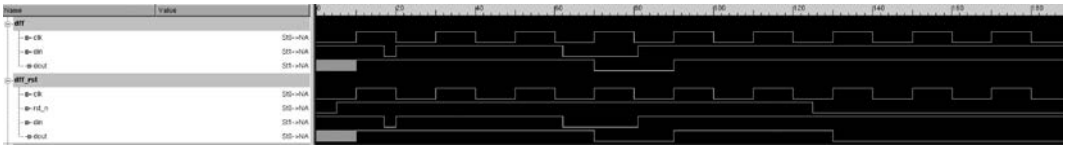

图 5-1　触发器仿真波形

可以看到，触发器在时钟上升沿时触发，将数据输入端 dff_din 的值更新到输出端 dout，同时可以看到两个触发器的区别，带低电平复位的 D 触发器在复位信号 rst_n 为低电平时，输出端 dout 的值被置为 0，因此触发器已经按照预期正确地运行了。

5.2 移位寄存器

9min

简单的单向移位寄存器，由低位向高位循环移动(循环左移)，可以加载设定移位寄存器的初始值。

移位寄存器的行为功能如下。

(1) 复位操作：当复位信号 rst_n 为低电平时，将移位寄存器输出端 dout 置为全 0。

(2) 加载操作：当复位信号 rst_n 为高电平且加载使能端 load_enable 为高电平时，会在时钟敏感边沿变化时将寄存器的初始值置为 load_data。

(3) 移位操作：当复位信号 rst_n 为高电平且加载使能端 load_enable 为低电平时，移位寄存器会在时钟敏感边沿变化时将输出端 dout 的值进行循环左移。

5.2.1 Verilog 实现

移位寄存器的 Verilog 实现，代码如下：

```
//文件路径:5.2/src/shifter.v
module shifter(clk,rst_n,load_enable,load_data,dout);
  input clk;
  input rst_n;
  input load_enable;       //加载使能端
  input[7:0] load_data;    //位宽为 8 的移位寄存器加载初始值输入端
  output[7:0] dout;        //位宽为 8 的移位寄存器的输出端

  reg[7:0] shift_data;     //内部移位寄存器变量

  always@(posedge clk)begin
    if(!rst_n)             //当复位信号为低电平时,在时钟敏感边沿将输出端置 0
      shift_data <= 'd0;
    else begin
      if(load_enable)      //加载使能端为高电平时,加载内部移位寄存器变量的初始值
        shift_data <= load_data;
      else                 //当加载使能端为低电平时,将内部移位寄存器变量的值进行循环左移
        shift_data <= {shift_data[6:0],shift_data[7]};
    end
  end

  assign dout = shift_data;          //将内部移位寄存器变量的值连续赋值给输出端进行输出

endmodule
```

5.2.2 测试平台

顶层测试模块的代码如下：

```
//文件路径:5.2/sim/testbench/demo_tb.sv
module top;
  logic clk;
  logic rst_n;
  logic load_enable;
  logic[7:0] load_data;
  logic[7:0] dout;

  shifter DUT(.clk(clk),
              .rst_n(rst_n),
              .load_enable(load_enable),
              .load_data(load_data),
              .dout(dout));     //例化连接 DUT

  initial begin                 //产生时钟翻转信号
    clk = 0;
    forever begin
      #10;
      clk = ~clk;
    end
  end

  initial begin                 //产生复位信号
    rst_n = 0;
    #50;
    rst_n = 1;
  end

  initial begin
    $display("%0t -> Start!!!", $time);
    load_enable = 0;
    load_data = 0;
    #100;

    load_enable = 1;            //将加载使能端置 1 以设置移位寄存器初始值
    std::randomize(load_data);  //对加载数据的初始值进行随机赋值
    $display("%0t -> load_data is %b", $time,load_data);
    #100;

    load_enable = 0;            //关闭加载使能

    #300; //延迟一段时间,这段时间内在每个时钟敏感边沿移位寄存器进行循环左移并输出
    $display("%0t -> Finish!!!", $time);
    $finish;
  end

endmodule : top
```

注意：这里使用了 SystemVerilog 中的随机方法 std::randomize()，用于获取变量的随机值，后面再向读者进行详细讲解。

5.2.3 仿真验证

仿真结果如下：

```
0 -> Start!!!
100 -> load_data is 01010001
500 -> Finish!!!
```

仿真波形如图 5-2 所示。

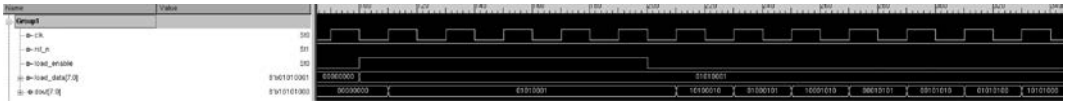

图 5-2 移位寄存器仿真波形

可以看到，当加载使能端 load_enable 为高电平时，会在时钟敏感边沿变化时将移位寄存器的初始值置为 load_data，即这里的随机值 8'b01010001，随后加载使能端 load_enable 被拉低，然后移位寄存器会在时钟敏感边沿变化时将输出端 dout 的值进行循环左移，波形中 8'b01010001 → 8'b10100010 → 8'b01000101 → 8'b10001010…这样一直循环左移下去，因此移位寄存器已经按照预期正确地运行了。

5.3 计数器

7min

顾名思义，计数器就是用来根据时钟边沿跳变来统计时钟周期数量的模块。

计数器的行为功能如下。

(1) 复位操作：当复位信号 rst_n 为低电平时，将计数器输出端 dout 置为全 0。

(2) 加载操作：当复位信号 rst_n 为高电平且加载使能端 load_enable 为高电平时，会在时钟敏感边沿变化时将计数器的初始值置为 load_counter。

(3) 计数操作：当复位信号 rst_n 为高电平且加载使能端 load_enable 为低电平时，计数器会在时钟敏感边沿变化时将输出端 dout 的值自增加 1。

5.3.1 Verilog 实现

计数器的 Verilog 实现，代码如下：

```
//文件路径:5.3/src/counter.v
module counter(clk,rst_n,load_enable,load_counter,dout);
```

```
  input clk;
  input rst_n;
  input load_enable;       //加载使能端
  input[7:0] load_counter;//位宽为 8 的计数器加载初始值输入端
  output[7:0] dout;        //位宽为 8 的计数器的输出端

  reg[7:0] counter;        //内部计数的寄存器变量

  always@(posedge clk)begin
    if(!rst_n)             //当复位信号为低电平时,在时钟敏感边沿将输出端置 0
      counter <= 'd0;
    else begin
      if(load_enable)      //当加载使能端为高电平时,加载内部计数的寄存器变量的初始值
        counter <= load_counter;
      else                 //当加载使能端为低电平时,将内部计数的寄存器变量的值自增加 1
        counter = counter + 1;
    end
  end

  assign dout = counter;  //将内部计数的寄存器变量的值连续赋值给输出端进行输出

endmodule
```

5.3.2　测试平台

顶层测试模块的代码如下：

```
//文件路径:5.3/sim/testbench/demo_tb.sv
module top;
  logic clk;
  logic rst_n;
  logic load_enable;
  logic[7:0] load_counter;
  logic[7:0] dout;

  counter DUT(.clk(clk),
              .rst_n(rst_n),
              .load_enable(load_enable),
              .load_counter(load_counter),
              .dout(dout));

  initial begin
    clk = 0;
    forever begin
      #10;
      clk = ~clk;
    end
  end

  initial begin
```

```
    rst_n = 0;
    #50;
    rst_n = 1;
  end

  initial begin
    $display("%0t -> Start!!!", $time);
    load_enable = 0;
    load_counter = 0;
    #100;

    load_enable = 1;
    std::randomize(load_counter);
    $display("%0t -> load_counter is %0d", $time,load_counter);
    #100;

    load_enable = 0;

    #300;
    $display("%0t -> Finish!!!", $time);
    $finish;
  end

endmodule : top
```

5.3.3 仿真验证

仿真结果如下：

```
0 -> Start!!!
100 -> load_counter is 81
500 -> Finish!!!
```

仿真波形如图 5-3 所示。

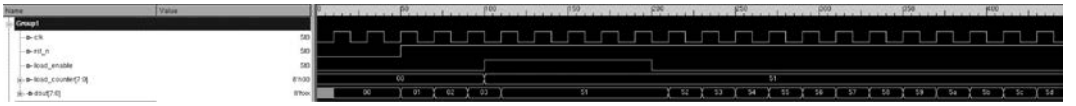

图 5-3 计数器仿真波形

可以看到，当加载使能端 load_enable 为高电平时，会在时钟敏感边沿变化时将寄存器的初始值置为 load_counter，即这里的随机值 8'd81（波形上是 8'h51），随后加载使能端 load_enable 被拉低，然后计数器会在时钟敏感边沿变化时将输出端 dout 的值进行自增加 1，波形中 8'h51 → 8'h52 → 8'h53 → 8'h54…这样一直递增下去，因此计数器已经按照预期正确地运行了。

5.4　状态机


16min

以一个经典的数字芯片设计笔试题——自动饮料售卖机为例，来向读者进行讲解。

5.4.1　过程分析


16min

1. 题目

要求设计一个自动饮料售卖机，饮料 10 分钱，硬币有 5 分和 10 分两种，并考虑找零。

(1) 分析并画出状态转移图、卡诺图，给出逻辑表达式。

(2) 使用 Verilog 实现。

(3) 搭建测试平台做简单验证。

下面带着读者一步步地分析和完成。


9min

2. 设计过程

第 1 步，确定输入输出。

• 输入部分：

A=1 表示投入 5 分钱，A=0 表示没有投入 5 分钱。

B=1 表示投入 10 分钱，B=0 表示没有投入 10 分钱。

• 输出部分：

Y=1 表示弹出饮料，Y=0 表示没有弹出饮料。

Z=1 表示找零，Z=0 表示不找零。


16min

第 2 步，确定电路状态。

S0 表示售卖机里还没有钱币，S1 表示已经投了 5 分钱。这里不存在其他情况，例如投了 10 分钱，已经足够完成本次交易，电路应该回归初始的 S0 状态，所以只会有 S0 和 S1 这两种状态。

第 3 步，画状态转移图。

• S0 状态时：

(1) 如果不投钱，则 AB=00，此时肯定不会弹出饮料，也不会找零，因此 YZ=00，此时状态也不会跳转，保持为 S0。

(2) 如果投入 5 分钱，则 AB=10，此时还不够 10 分钱，因此不会弹出饮料，也不会找零，因此 YZ=00，此时状态将跳转换为 S1。

(3) 如果投入 10 分钱，则 AB=01，此时售卖机中已经达到 10 分钱，因此会弹出饮料，但不会找零，因此 YZ=10，完成交易后回到初始状态 S0，等待进行下一次交易。

(4) 同时投入 5 分钱和 10 分钱的情况假设不会发生(投币口只支持一次投一个币)，因此 AB=11 的情况，默认 YZ=00，并回到初始态 S0。

• S1 状态时：

(1) 如果不投钱，则 AB=00，此时肯定不会弹出饮料，也不会找零，因此 YZ=00，此时

状态也不会跳转，保持为 S1。

（2）如果投入 5 分钱，则 AB＝10，此时售卖机中已经达到 10 分钱，因此会弹出饮料，但不会找零，因此 YZ＝10，完成交易后回到初始状态 S0，等待进行下一次交易。

（3）如果投入 10 分钱，则 AB＝01，此时售卖机中已经达到 15 分钱，因此会弹出饮料，同时会找零，因此 YZ＝11，完成交易后回到初始状态 S0，等待进行下一次交易。

注意：一般来讲，如果顾客手中有两枚硬币，肯定直接投 10 分钱就好了，哪有先投 5 分钱再投 10 分钱等着把之前投进去的 5 分钱再找零回来的道理，但做电路设计需要考虑这种特殊的情况，那可能是这个顾客忘了自己有两枚硬币，也可能是上个顾客投了 5 分钱后，发现身上只有 5 分钱就走了，然后第 2 个顾客过来捡了便宜，所以只要有可能发生的情况，在设计时都必须考虑到，尤其是在实际项目中比这复杂的情况都要尽量考虑到，而这些地方恰恰是容易发生问题且易存在 Bug 的地方，需要尤其认真仔细。

（4）同时投入 5 分钱和 10 分钱的情况假设不会发生（投币口只支持一次投一个币），因此 AB＝11 的情况，默认 YZ＝00，并回到初始态 S0。

根据以上分析，可以画出状态转移图，如图 5-4 所示。

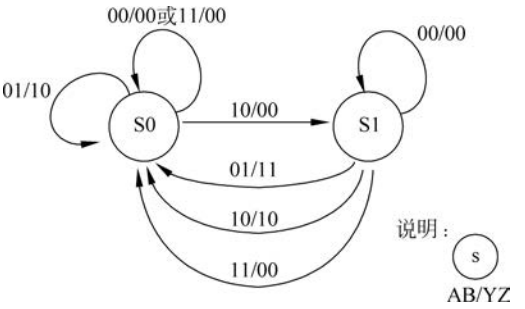

图 5-4　饮料售卖机状态转移图

第 4 步，真值表。

根据图 5-4 可以很容易地列出表 5-1 所示的真值表。

表 5-1　饮料售卖机真值表

A	*B*	S^n	S^{n+1}	*Y*	*Z*
0	0	0	0	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	1	0	0	0	0
0	0	1	1	0	0
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	0	0	0

第 5 步，卡诺图及逻辑表达式。

根据表 5-1 可以很容易地画出下一种状态 S^{n+1}、是否弹出饮料的输出端 Y、是否找零的输出端 Z 的卡诺图并对逻辑表达式进行化简，分别如图 5-5～图 5-7 所示。

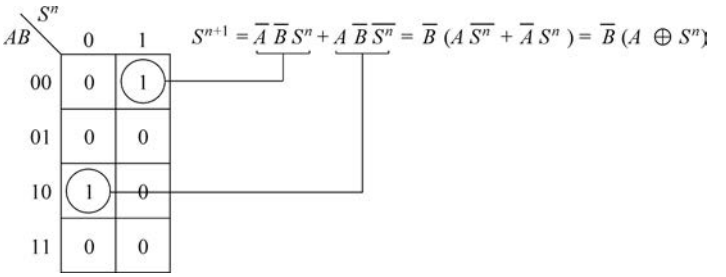

图 5-5　饮料售卖机下一种状态 S^{n+1} 卡诺图及逻辑表达式

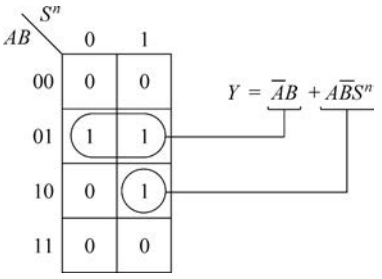

图 5-6　饮料售卖机输出端 Y 卡诺图及逻辑表达式

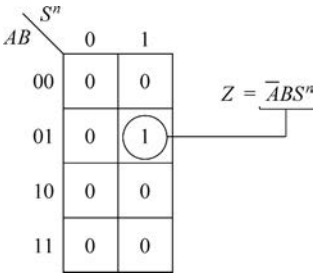

图 5-7　饮料售卖机输出端 Z 卡诺图及逻辑表达式

有了上面这些分析过程，下一步的 Verilog 实现就会变得很容易了。

5.4.2　Verilog 实现

1. 二段式 Verilog 描述——基于逻辑公式

二段式描述的状态机，可以理解为两个 always 程序块。

（1）第 1 个 always 程序块采用同步时序逻辑电路描述状态转移。

（2）第 2 个 always 程序块采用组合逻辑电路判断状态转移条件并描述状态转移规律，同时采用组合逻辑电路输出结果。

第 2 个 always 程序块中的组合逻辑电路会用到第 5.4.1 节内容中化简后的公式。

基于逻辑公式的二段式描述自动饮料售卖机的 Verilog 实现，代码如下：

```
//文件路径:5.4/src/sell.v
module sell1(clk,rst_n,a,b,y,z);
  parameter S0 = 1'b0;
  parameter S1 = 1'b1;
  input clk;
  input rst_n;
  input a,b;
```

```
    output reg y,z;

    reg current_state;
    reg next_state;

  //采用同步时序逻辑电路描述状态转移
    always@(posedge clk or negedge rst_n)begin
      if(!rst_n)
        current_state <= S0;
      else
        current_state <= next_state;
    end

  //采用组合逻辑电路判断状态转移条件并描述状态转移规律,同时采用组合逻辑电路输出结果,
  //这里使用前面的逻辑表达式可以很容易地实现组合逻辑
    always@(current_state or a or b)begin
      next_state = (~b) & (a^current_state);
      y = ((~a) & b) || (a & (~b) & current_state);
      z = (~a) & b & current_state;
    end

  endmodule
```

可以看到,在之前章节的内容中,给读者提到过的内容得到了应用:

(1) 当用 always 块来描述组合逻辑时,应当使用阻塞赋值。

(2) 当用 always 块来描述时序逻辑时,应当使用非阻塞赋值。

(3) 在同一个 always 模块中,最好不要混合使用阻塞赋值和非阻塞赋值,如果对同一变量既进行阻塞赋值,又进行非阻塞赋值,则在综合时会出错,所以 always 中要么全部使用非阻塞赋值,要么把阻塞赋值和非阻塞赋值分在不同的 always 中书写。

2. 二段式 Verilog 描述——基于行为级

这里依然采用二段式描述:

(1) 第 1 个 always 程序块采用同步时序逻辑电路描述状态转移。

(2) 第 2 个 always 程序块采用组合逻辑电路判断状态转移条件并描述状态转移规律,同时采用组合逻辑电路输出结果。

但是,这里采用行为级描述,不用像之前那样,又画状态转移图,又画卡诺图真值表,这次可以简单点。

行为级描述的第 2 个 always 程序块里的组合逻辑看起来更容易理解一点,写起来相对也更快,但是传统的采用卡诺图化简逻辑表达式的方式和这里的按照行为逻辑实现的方式都要掌握,两种方法结合才能解决在实际工作中遇到的更多问题。

基于行为级的二段式描述自动饮料售卖机的 Verilog 实现,代码如下:

```
//文件路径:5.4/src/sel1.v
module sel12(clk,rst_n,a,b,y,z);
```

```
  parameter S0 = 1'b0;
  parameter S1 = 1'b1;
  input clk;
  input rst_n;
  input a,b;
  output reg y,z;

  reg current_state;
  reg next_state;

//采用同步时序逻辑电路描述状态转移
  always@(posedge clk or negedge rst_n)begin
    if(!rst_n)
      current_state <= S0;
    else
      current_state <= next_state;
  end

//采用组合逻辑电路判断状态转移条件并描述状态转移规律,同时采用组合逻辑电路输出结果
//这里使用前面的状态转移图可以很容易地实现组合逻辑
  always@(current_state or a or b)begin
    y = 0;
    z = 0;
    case(current_state)
      S0: begin
        if((a == 1'b0) && (b == 1'b1))begin
          y = 1;
          next_state = S0;
        end
        else if((a == 1'b1) && (b == 1'b0))
          next_state = S1;
        else
          next_state = S0;
      end
      S1: begin
        if((a == 1'b0) && (b == 1'b0))
          next_state = S1;
        else if((a == 1'b0) && (b == 1'b1))begin
          y = 1;
          z = 1;
          next_state = S0;
        end
        else if((a == 1'b1) && (b == 1'b0))begin
          y = 1;
          next_state = S0;
        end
        else
          next_state = S0;
```

```
        end
      endcase
  end

endmodule
```

3. 三段式 Verilog 描述

三段式描述的状态机，可以理解为 3 个 always 程序块。

(1) 第 1 个 always 程序块采用同步时序逻辑电路描述状态转移。

(2) 第 2 个 always 程序块采用组合逻辑电路判断状态转移条件并描述状态转移规律。

(3) 第 3 个 always 程序块采用同步时序逻辑将结果寄存后输出。

两者的区别是三段式 Verilog 描述将原先第 2 个 always 程序块中对 y 和 z 的组合逻辑输出改为第 3 个 always 块的时序逻辑的寄存输出。三段式 Verilog 描述要将最后输出的结果进行时钟同步后通过寄存器输出。

下面用二段式改三段式描述的过程来向读者说明这两种描述方式的区别。

这是原本的二段式实现的代码：

```
//文件路径:5.4/src/sell.v
module sell2(clk,rst_n,a,b,y,z);
  parameter S0 = 1'b0;
  parameter S1 = 1'b1;
  input clk;
  input rst_n;
  input a,b;
  output reg y,z;

  reg current_state;
  reg next_state;

//采用同步时序逻辑电路描述状态转移
  always@(posedge clk or negedge rst_n)begin
    if(!rst_n)
      current_state <= S0;
    else
      current_state <= next_state;
  end

//采用组合逻辑电路判断状态转移条件并描述状态转移规律,同时采用组合逻辑电路输出结果
//这里使用前面的状态转移图可以很容易地实现组合逻辑
  always@(current_state or a or b)begin
    y = 0;
    z = 0;
    case(current_state)
      S0: begin
        if((a == 1'b0) && (b == 1'b1))begin
          y = 1;
          next_state = S0;
```

```
        end
        else if((a == 1'b1) && (b == 1'b0))
          next_state = S1;
        else
          next_state = S0;
      end
      S1: begin
        if((a == 1'b0) && (b == 1'b0))
          next_state = S1;
        else if((a == 1'b0) && (b == 1'b1))begin
          y = 1;
          z = 1;
          next_state = S0;
        end
        else if((a == 1'b1) && (b == 1'b0))begin
          y = 1;
          next_state = S0;
        end
        else
          next_state = S0;
      end
    endcase
  end

endmodule
```

下面对其一步步改造。

第 1 步，将第 2 个 always 程序块中的输出 y 和 z 部分挪到第 3 个 always 块中，代码如下：

```
module sell3(clk,rst_n,a,b,y,z);
  parameter S0 = 1'b0;
  parameter S1 = 1'b1;
  input clk;
  input rst_n;
  input a,b;
  output reg y,z;

  reg current_state;
  reg next_state;

  always@(posedge clk or negedge rst_n)begin  //第 1 个 always 块
    if(!rst_n)
      current_state <= S0;
    else
      current_state <= next_state;
  end

  always@(current_state or a or b)begin       //第 2 个 always 块
```

```
    y = 0;
    z = 0;
    case(current_state)
      S0: begin
        if((a == 1'b0) && (b == 1'b1))begin
          y = 1;
          next_state = S0;
        end
        else if((a == 1'b1) && (b == 1'b0))
          next_state = S1;
        else
          next_state = S0;
      end
      S1: begin
        if((a == 1'b0) && (b == 1'b0))
          next_state = S1;
        else if((a == 1'b0) && (b == 1'b1))begin
          y = 1;
          z = 1;
          next_state = S0;
        end
        else if((a == 1'b1) && (b == 1'b0))begin
          y = 1;
          next_state = S0;
        end
        else
          next_state = S0;
      end
    endcase
  end

  always@(posedge clk or negedge rst_n)begin  //第 3 个 always 块用于对结果进行寄存和输出
    if(!rst_n)begin
      y <= 0;
      z <= 0;
    end
    else begin
      case(current_state)
        S0: begin
          if((a == 1'b0) && (b == 1'b1))begin
            y <= 1;
          end
          else begin
            y <= 0;
            z <= 0;
          end
        end
        S1: begin
          if((a == 1'b0) && (b == 1'b1))begin
            y <= 1;
            z <= 1;
```

```
            end
            else if((a == 1'b1) && (b == 1'b0))begin
              y = 1;
              z = 0;
            end
            else begin
              y <= 0;
              z <= 0;
            end
          end
        endcase
      end
    end

endmodule
```

注意：第 3 个 always 块为时序逻辑电路，采用非阻塞赋值。

第 2 步，删除第 2 个 always 块中的 y 和 z 部分，代码如下：

```
module sell3(clk,rst_n,a,b,y,z);
  parameter S0 = 1'b0;
  parameter S1 = 1'b1;
  input clk;
  input rst_n;
  input a,b;
  output reg y,z;

  reg current_state;
  reg next_state;

  always@(posedge clk or negedge rst_n)begin     //第 1 个 always 块
    if(!rst_n)
      current_state <= S0;
    else
      current_state <= next_state;
  end

  always@(current_state or a or b)begin          //第 2 个 always 块
    case(current_state)
      S0: begin
        if((a == 1'b0) && (b == 1'b1))begin
          next_state = S0;
        end
        else if((a == 1'b1) && (b == 1'b0))
          next_state = S1;
        else
          next_state = S0;
      end
      S1: begin
```

```
        if((a == 1'b0) && (b == 1'b0))
          next_state = S1;
        else if((a == 1'b0) && (b == 1'b1))begin
          next_state = S0;
        end
        else if((a == 1'b1) && (b == 1'b0))begin
          next_state = S0;
        end
        else
          next_state = S0;
      end
    endcase
  end

  always@(posedge clk or negedge rst_n)begin //第 3 个 always 块
    if(!rst_n)begin
      y <= 0;
      z <= 0;
    end
    else begin
      case(current_state)
        S0: begin
          if((a == 1'b0) && (b == 1'b1))begin
            y <= 1;
          end
          else begin
            y <= 0;
            z <= 0;
          end
        end
        S1: begin
          if((a == 1'b0) && (b == 1'b1))begin
            y <= 1;
            z <= 1;
          end
          else if((a == 1'b1) && (b == 1'b0))begin
            y = 1;
            z = 0;
          end
          else begin
            y <= 0;
            z <= 0;
          end
        end
      endcase
    end
  end

endmodule
```

第 3 步，化简合并第 2 个 always 块中的逻辑即可得到最终的三段式状态机描述，代码如下：

```
//文件路径:5.4/src/sell.v
module sell3(clk,rst_n,a,b,y,z);
  parameter S0 = 1'b0;
  parameter S1 = 1'b1;
  input clk;
  input rst_n;
  input a,b;
  output reg y,z;

  reg current_state;
  reg next_state;

  always@(posedge clk or negedge rst_n)begin //第 1 个 always 块
    if(!rst_n)
      current_state <= S0;
    else
      current_state <= next_state;
  end

  always@(current_state or a or b)begin //第 2 个 always 块
    case(current_state)
      S0: begin
        if((a == 1'b1) && (b == 1'b0))
          next_state = S1;
        else
          next_state = S0;
      end
      S1: begin
        if((a == 1'b0) && (b == 1'b0))
          next_state = S1;
        else
          next_state = S0;
      end
    endcase
  end

  always@(posedge clk or negedge rst_n)begin //第 3 个 always 块
    if(!rst_n)begin
      y <= 0;
      z <= 0;
    end
    else begin
      case(current_state)
        S0: begin
          if((a == 1'b0) && (b == 1'b1))begin
            y <= 1;
          end
```

```
            else begin
              y <= 0;
              z <= 0;
            end
          end
          S1: begin
            if((a == 1'b0) && (b == 1'b1))begin
              y <= 1;
              z <= 1;
            end
            else if((a == 1'b1) && (b == 1'b0))begin
              y = 1;
              z = 0;
            end
            else begin
              y <= 0;
              z <= 0;
            end
          end
        endcase
      end
    end

endmodule
```

注意这里用二段式改三段式描述的过程只是为了向读者说明这两种描述方式的区别，并不是让读者先实现二段式，再实现三段式。实际上，读者完全可以参考状态转移图直接编写并实现三段式描述的 Verilog 代码。

5.4.3 测试平台

可以设计以下 3 种实际投币购买饮料的场景来进行测试。

1. 场景 1

先投 5 分钱，然后投 5 分钱。

此时期望的结果是弹出饮料，但不会找零，即 Y 和 Z 输出分别为 1 和 0，并且至少 Y 会有一段高电平的状态。

2. 场景 2

先投 5 分钱，然后投 10 分钱。

此时期望的结果是弹出饮料，并找零，即 Y 和 Z 输出都为 1，并且 Y 和 Z 都会有一段高电平的状态。

3. 场景 3

直接投 10 分钱。

此时期望的结果是弹出饮料，但不会找零，即 Y 和 Z 输出分别为 1 和 0，并且至少 Y 会有一段高电平的状态。

二段式和三段式描述的 Verilog 测试平台的代码是一样的，都对上述 3 种场景进行测试，只要例化不同的 DUT 模块即可。

（1）sell1：二段式 Verilog 描述——基于逻辑公式。

（2）sell2：二段式 Verilog 描述——基于行为级。

（3）sell3：三段式 Verilog 描述。

顶层测试模块的参考代码如下：

```
//文件路径:5.4/sim/testbench/demo_tb.sv
module top;
  logic clk;
  logic rst_n;
  logic a,b;
  logic y1,z1;
  logic y2,z2;
  logic y3,z3;

  sell1 DUT1(.clk(clk),.rst_n(rst_n),.a(a),.b(b),.y(y1),.z(z1));
  sell2 DUT2(.clk(clk),.rst_n(rst_n),.a(a),.b(b),.y(y2),.z(z2));
  sell3 DUT3(.clk(clk),.rst_n(rst_n),.a(a),.b(b),.y(y3),.z(z3));

  initial begin
    clk = 0;
    forever begin
      #10;
      clk = ~clk;
    end
  end

  initial begin
    rst_n = 0;
    #50;
    rst_n = 1;
  end

  initial begin
     $display("%10t -> Start!!!", $time);
    a = 0;
    b = 0;
    #80;
    $display("-----------------------------------------");
    $display("%10t -> Scenario 1", $time);
    $display("%10t -> insert 5 cents", $time);
    a = 1;
    b = 0;
    #20;
    $display("%10t -> insert 5 cents", $time);
    a = 1;
    b = 0;
    #20;
```

```
        a = 0;
        b = 0;

        #100;
        $display("-----------------------------------------");
        $display("%10t -> Scenario 2", $time);
        $display("%10t -> insert 5 cents", $time);
        a = 1;
        b = 0;
        #20;
        $display("%10t -> insert 10 cents", $time);
        a = 0;
        b = 1;
        #20;
        a = 0;
        b = 0;

        #100;
        $display("-----------------------------------------");
        $display("%10t -> Scenario 3", $time);
        $display("%10t -> insert 10 cents", $time);
        a = 0;
        b = 1;
        #20;
        a = 0;
        b = 0;

        #100;
        $display("%10t -> Finish!!!", $time);
        $finish;
    end

endmodule : top
```

5.4.4 仿真验证

仿真结果如下：

```
         0 -> Start!!!
-----------------------------------------
        80 -> Scenario 1
        80 -> insert 5 cents
       100 -> insert 5 cents
-----------------------------------------
       220 -> Scenario 2
       220 -> insert 5 cents
       240 -> insert 10 cents
-----------------------------------------
       360 -> Scenario 3
       360 -> insert 10 cents
       480 -> Finish!!!
```

从仿真结果的日志报告上来看，一共对 3 种场景进行了测试，和之前描述一致。

仿真波形如图 5-8 所示。

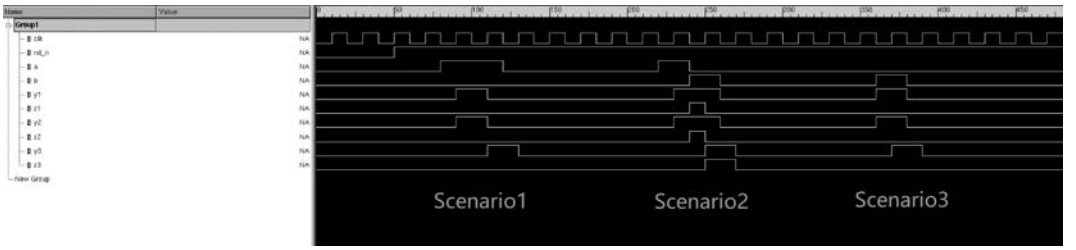

图 5-8　饮料售卖机仿真波形

为了看得更清楚，标注了上述 3 种场景，可以看到，输出的波形结果是符合期望的。

其中二段式状态机描述的输出结果 y1 和 y2，z1 和 z2 的波形是一致的，而三段式状态机描述通过将结果 y3 和 z3 寄存后输出，使其与时钟进行同步，输出电平以时钟周期为单位进行了整型。

简单来说，二段式状态机描述采用组合逻辑输出，因此输出结果会立刻变化，而三段式状态机描述则采用了与时钟同步的寄存器对结果进行寄存后再输出，因此输出的 y3 和 z3 波形是以时钟周期为电平单位进行变化的，只有在时钟的跳变沿才会产生变化，即不像组合逻辑那样立刻产生变化。

这样的好处主要是改善了时序条件，便于后期满足电路的时序要求，消除了组合逻辑带来的毛刺，但是三段式要相对复杂一点，多写了一个 always 块，即将结果寄存输出的时序逻辑，从而使综合后的电路面积可能会相对更多一些，但为了提高设计的稳定性，推荐采用三段式描述进行状态机的设计。

5.5　本章小结

本章通过几个典型的时序逻辑电路的实例，结合讲过的 Verilog 语法基础，引导读者简单、迅速地完成设计和简单验证的流程，从而让读者加深理解，提升对数字电路及 Verilog 基础内容的学习效果。

另外给读者留两个作业，也是数字芯片设计经常会看到的笔试题，感兴趣可以进行练习，以提升学习效果。

(1) 用有限状态机(FSM)实现 101101 的序列检测模块。

(2) 画状态机，接收 1、2、5 分钱的卖报机，每份报纸卖 5 分钱。