第5章

如何写好状态机

随着时代的发展，现在可编程逻辑器件的规模越来越大，器件集成度也越来越高，以Altera最新的Stratix V和Cyclone V来说，其逻辑最小元素都是ALM，每个ALM之中已经集成了4个寄存器。前面几章介绍了几种基本电路设计的技巧，本章来讨论状态机电路的设计。

在进行FPGA开发或逻辑设计时，经常需要编写状态机，比如FPGA处理系统中随机发生的事件，事件处理完成后形成了一个128位的数据需要通过高速串行通道发送出去。而能够使用的高速串行接口只有一个通道，且并行数据位宽只有16位，所以，一个128位的数据需要经过8个并行周期或步骤才能发送完成，此时就非常适合使用状态机来进行控制。状态机是一个比较典型、应用也非常广泛的时序电路模块。设计状态机并不难，但是写好状态机并不容易。本章在简单介绍状态机的基础上，重点介绍如何写好状态机。

5.1 状态机的特定及常见问题

标准状态机分为摩尔(Moore)状态机和米勒(Mealy)状态机两种。摩尔状态机的输出只与当前状态值有关，且只在时钟边沿到来时才会有状态变化。米勒状态机的输出不但与当前状态值有关，而且与当前输入值有关，这一特点使得其控制和输出更加灵活，但同时也增加了设计复杂程度。这两种状态机的原理框图如图5-1所示。

从图5-1很容易理解状态机的结构。但是，为什么要使用状态机而不使用一般的时序电路呢？这是因为状态机具有一些一般时序电路所无法比拟的优点：

- HDL描述的状态机结构分明、易读、易懂、易排错。
- 相对其他时序电路而言，状态机更加稳定，运行模式类似CPU，易于控制。

与使用C语言设计软件一样，逻辑设计中使用状态机也会遇到意外状况，比如同样会经常遇到所谓的“跑飞”的问题。归纳来说，逻辑设计中状态机可能会遇到如下问题：

- 两个状态转换时，出现过渡状态。
- 在运行过程中，进入非法状态。
- 在一种器件上综合出理想结果，移植到另一器件时，不能得到与之相符的结果。

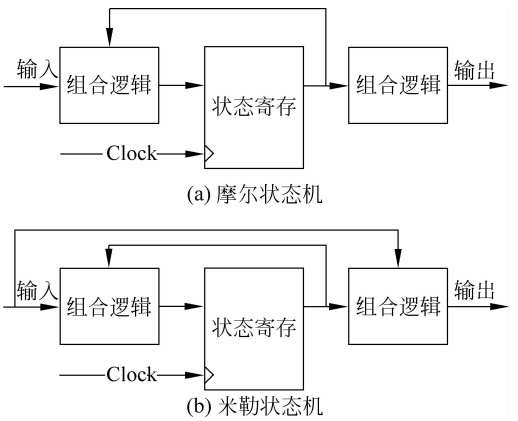

图 5-1 摩尔状态机和米勒状态机原理框图

• 状态机能够稳定工作,但是占用资源过多。

在针对 FPGA 器件综合时,状态机资源消耗过多会经常遇到,所以设计者必须慎重设计状态机,分析状态机内在的结构。在摩尔状态机中,输出信号是当前状态值的译码,当状态寄存器的状态值稳定时,输出也随之稳定了。经综合器综合后一般生成以触发器为核心的状态寄存器电路,其稳定性由此决定。如果时钟信号上升沿到达各个触发器的时间严格一致,状态值也会按照设计要求在规定的状态值之间转换。然而,这只是一种理想情况,实际 FPGA 器件一般无法满足这种苛刻的时序要求,特别是在布局布线后这些触发器相距较远时,时钟到达各个触发器的延时往往有一些差异(时钟歪斜)。这种差异将直接导致状态机在状态转换时产生过渡状态,当这种延时进一步加大时,将有可能导致状态机进入非法状态。这就是摩尔状态机的失效机理。对于米勒状态机而言,由于其与任何时刻的输出和输入有关,这种情况就更加常见了。因为米勒状态机的输出不与时钟同步,所以当状态译码比较复杂的时候,很容易在输出端产生大量的毛刺,这种情况是无法避免的。在一些特定的系统中,这些毛刺可能造成不可预料的结果。但是,由于输入变化可能出现在时钟周期内的任何时刻,这就使得米勒状态机对输入的响应可以比摩尔状态机对输入的响应要早一个时钟周期。摩尔状态机的输出与时钟同步,可以在一定程度上剔除抖动。从稳定性的角度来讲,建议使用摩尔状态机。

5.2 如何选择状态机的编码方式

状态编码,是指定义状态机的当前状态和下一状态(有时也使用上一状态和当前状态),一般有三种方式。

方式一 使用逻辑向量定义状态:

```
signal crrent_state: std_logic_vector(1 downto 0);
signal next_state: std_logic_vector(1 downto 0);
```

使用这种方式来定义状态会有比较多的毛病,比如缺乏具体状态的含义,程序的可读性

较差，更为重要的是，设计后期调试修改都比较麻烦。

方式二　定义如下：

```
type mystate is (st0,st1,st2,st3);
signal current_state,next_state: mystate;
```

使用这种方式定义的状态有具体状态的含义，可读性好，易于调试和修改。

方式三　通过定义常数的方式来定义状态：

```
constant st0: std_logic_vectro(1 downto 0): = "00";
constant st1: std_logic_vectro(1 downto 0): = "01";
constant st2: std_logic_vectro(1 downto 0): = "10";
constant st3: std_logic_vectro(1 downto 0): = "11";
signal current_state,next_state: std_logic_vectro(1 downto 0);
```

这种方式比方式一的可读性要好，只是其修改没有方式二方便。

上面介绍了状态编码的方式，而具体的编码形式又有三种。

- 顺序码：状态编码遵循传统的状态二进制序列。
- 格雷码：除了相邻状态编码之间只有一个位变化外，其他和顺序码类似。
- 独热码：这种方法是在状态机中为每一种状态分配一个触发器。只有一个触发器当前设置为有效，其余均设置为无效，故称为“独热”。

前面方式三中使用的就是顺序编码形式。可以将上述编码形式分别修改成格雷码和独热码编码形式。

格雷编码形式：

```
constant st0: std_logic_vectro(1 downto 0): = "00";
constant st1: std_logic_vectro(1 downto 0): = "01";
constant st2: std_logic_vectro(1 downto 0): = "11";
constant st3: std_logic_vectro(1 downto 0): = "10";
```

独热编码形式：

```
constant st0: std_logic_vectro(3 downto 0): = "0001";
constant st1: std_logic_vectro(3 downto 0): = "0010";
constant st2: std_logic_vectro(3 downto 0): = "0100";
constant st3: std_logic_vectro(3 downto 0): = "1000";
```

分析这 3 种编码形式，如果使用顺序编码，从状态“01”到“10”的转换过程中很可能会出现过渡状态“11”或“00”。这是因为中间信号 crrent_state 在状态转换过程中，状态寄存器的高位翻转和低位翻转时间有可能不一致，当高位翻转速度快时，会产生过渡状态“11”，当低位翻转速度快时会产生过渡状态“00”。所以，可想而知，如果状态机的状态值更多的话，则产生过渡状态的概率就更大。假如，状态机并未使用全部编码（比如只有 5 个状态，编码中有 3 个未用），由于这种过渡状态的反馈作用，将直接导致电路进入非法状态，若此时电路不具备自启动功能，那么电路就无法返回正常工作状态（彻底跑飞了）。

假如使用的是格雷编码，由于相邻两个数据之间只有一位不同，所以可在很大程度上消除由延时引起的过渡状态。使用格雷码虽然可以大大降低产生过渡状态的概率，但是如果当一个状态到下一个状态有多种转换路径时，就不能保证状态跳转时只有一个位变化，这样

将无法发挥格雷码的特点。所以,需要仔细分析状态机的结构,如果状态机中某个状态跳转方向多于一个,此时慎用格雷编码,可以采用独热编码。

5.3 合理选择及使用单进程或多进程来设计状态机

在状态机的具体逻辑描述时,分为单进程和多进程方式,其中多进程一般分为双进程和三进程两种,也有人将其分别称为单段、两段和三段式状态机。显而易见,所谓的单进程状态机,就是状态机所有的描述位于同一个进程之中,该进程既包括状态转移,又描述了状态的输入和输出。而所谓的多进程状态机,是将状态机中的时序逻辑和组合逻辑分别使用不同的进程进行描述。由于三进程和两进程状态机差别不大,本节主要介绍单进程和三进程状态机,最后通过简单的例子对单进程和多进程状态机进行比较。

5.3.1 多进程状态机

在状态机的描述中,多进程方式使用较多,双进程和三进程描述方式中,三进程描述方式仅仅比双进程多使用了一个进程对状态机的输出进行描述。而三进程状态机又分为两种,第一种是输出进程使用组合逻辑进行描述;第二种是使用时序逻辑对输出进行描述,其余两个进程完全相同。一般来说,使用寄存器输出可以改善输出的时序条件,还能避免组合电路的毛刺,所以是推荐的描述方式。首先来看使用组合进程来描述状态机输出的基本结构,模型描述如下:

```
SYNC_PROC: process(clock,reset)                    -- 同步进程
begin
    if (reset = '1') then                          -- 现态 <= 初始状态
      ...
    elsif (clock'event and clock = '1') then       -- 现态 <= 次态
      ...
    end if;
end process;
COMB_PROC1: process(现态,输入信号)                  -- 状态转换进程
begin
    case 现态 is
        when 初始状态 =>
              if (转换条件) then                    -- 次态赋值
                  ...
              end if;
         -- 其他所有状态转换描述
     end case;
end process;
COMB_PROC2: process(现态,输入信号)                  -- 输出描述进程
begin
    case 现态 is
        when 初始状态 =>
              if (输入端的变化) then
```

```
                    -- 输出赋值
                end if;
            -- 其他所有状态下输出赋值
        end case;
end process;
```

或者：

```
COMB_PROC2: process(现态,输入信号)                    -- 输出描述进程
begin
    case 现态 is
        when 初始状态 =>                              -- 输出赋值
         -- 其他所有状态下输出赋值
     end case;
end process;
```

以上模型描述的三进程状态机中的各个进程分工相当明确，SYNC_PROC 进程完成状态的同步描述和状态机的初始化；COMB_PROC1 进程完成对状态转换的描述；COMB_PROC2 完成对输出的描述。三个进程分别负责一个工作，互不干扰。第一个进程的电路是时序逻辑，后面两个进程的电路是组合逻辑。COMB_PROC2 进程可能出现两种描述，第一种描述输出端是现态和输入的函数，这是米勒状态机；第二种描述输出端是仅为现态的函数，这是摩尔状态机。

当然，COMB_PROC1 和 COMB_PROC2 两个进程完全可以写到一个进程中，那么整个状态机的进程就分为时序进程和组合进程，这就是典型的双进程（或两段）状态机了。

不过，这里需要提醒大家注意的是，前面介绍的三进程状态机模型，并不是笔者推荐大家使用的模型。5.2 节在讨论状态机问题时提到，状态机的输出是组合逻辑，很容易产生毛刺。所以，推荐的三进程状态机描述方式，是将上述第二个组合逻辑进程改成时序电路进程，这样三个进程只有一个状态转移的组合进程。模型如下所示：

```
SYNC_PROC2: process(clock,reset)                      -- 输出同步进程
begin
    if (reset = '1') then                             -- 初始化输出状态
      ...
    elsif (clock'event and clock = '1') then          -- 输出赋值
      case 现态 is
          when 初始状态 =>                            -- 输出赋值
           -- 其他所有状态下输出赋值
      end case;
    end if;
end process;
```

经过上述对状态输出的寄存处理，那么可以将图 5-1 所示的状态机原理图进行改进，如图 5-2 所示。这种结构可以有效地抑制过渡状态的出现，这是因为输出寄存器只要求状态值在时钟边沿稳定。这种结构的状态机稳定性显然要优于一般结构的状态机，但是它占用的资源更多，状态机的输出增加了

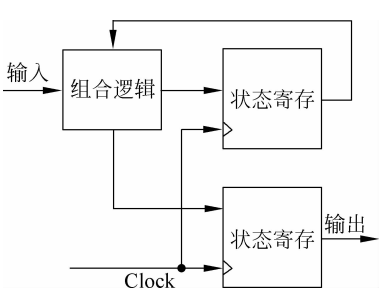

图 5-2　改进后的状态机原理图

一个时钟周期的延时,在设计时需要综合考虑。对于这两个缺点,笔者个人理解是大部分同步设计中都是可以忽略不计的,增加的逻辑资源消耗几乎可以忽略不计,因为现在逻辑器件的规模越来越大,不在意这一点点资源的消耗,至于输出的延时,对于同步设计来说,都是流水操作,一般不会影响到整个系统的性能(有死时间要求的除外)。

5.3.2 单进程状态机

在实际的设计中,在输入端引入噪声是难免的。如果使用的是米勒状态机,噪声很容易传递到输出端口,在输出端口出现毛刺,如果整个状态机的输出要被用于作为其他模块的同步信号,则系统设计很可能会失败。那么,在这种情况下最好采用摩尔状态机。在5.3.1节介绍的三进程状态机模型中,如果不对输出进行寄存,则输出进程的描述都是组合逻辑,而组合逻辑没有抑制毛刺的能力。所以,不带输出寄存的三进程摩尔状态机只是相对三进程米勒状态机有一定的抗干扰能力。

为了使系统更加稳定,可以在输出组合逻辑后面加一级寄存器,采用时钟边沿触发,这样可以在很大程度上剔除毛刺,抑制噪声干扰。一般单进程状态机都可以达到这样的设计目的。单进程状态机的基本结构可以由下面的模型来描述。

```
SYNC_PROC: process(clock,reset)                    -- 同步进程
begin
    if (reset = '1') then
      -- 现态 <= 初始状态
    elsif (clock'event and clock = '1') then
      case 现态 is
           when 初始状态 =>
                if ( -- 转换条件) then
                     -- 现态赋值
                end if;
                if ( -- 转换条件) then
                     输出赋值
                end if;
           -- 其他所有状态转换的描述
           -- 其他所有状态下输出的描述
      end case;
    end if;
end process;
```

可以看到,在单进程描述的状态机中,由于状态机的所有工作都是在时钟上升沿触发下完成的,所以所有输出都经过了一级寄存器,从而抑制了毛刺的产生。

5.3.3 状态机的比较

本节给出一个简单的状态机的代码示例。该状态机分别使用一个进程、两个进程以及三个进程进行编写。来看看这三种不同写法的相同状态机的效果。

首先,来看单进程状态机的代码:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity statemach1 is

   port(
      clk     : in std_logic;
      reset   : in std_logic;
      in1     : in std_logic;
      in2     : in std_logic;
      out1    : out std_logic;
      out2    : out std_logic
   );

end entity;

architecture rtl of statemach1 is
type mystate is (st0, st1, st2, st3);
signal current_state,next_state : mystate;
begin
process (clk,reset)
begin
    if reset = '1' then
       current_state <= st0;
    elsif (rising_edge(clk)) then
       current_state <= next_state;
       case current_state is
            when st0 =>
                 if in1 = '1' then
                    next_state <= st1;
                 end if;
                 out1 <= '0';
                 out2 <= '0';
            when st1 =>
                 if in2 = '1' then
                    next_state <= st2;
                 end if;
                 out1 <= '1';
                 out2 <= '0';
            when st2 =>
                 if in1 = '0' and in2 = '1' then
                    next_state <= st3;
                 end if;
                 out1 <= '0';
                 out2 <= '1';
            when st3 =>
                 if in1 = '0' and in2 = '0' then
                    next_state <= st0;
                 end if;
```

```
                    out1 <= '1';
                    out2 <= '1';
                when others => next_state <= st0;
            end case;
        end if;
    end process;

    end rtl;
```

将上述代码在 Quartus Ⅱ 软件中进行编译。图 5-3 所示是代码编译后的 RTL 视图。图 5-4 则显示了单进程状态机的资源消耗情况。

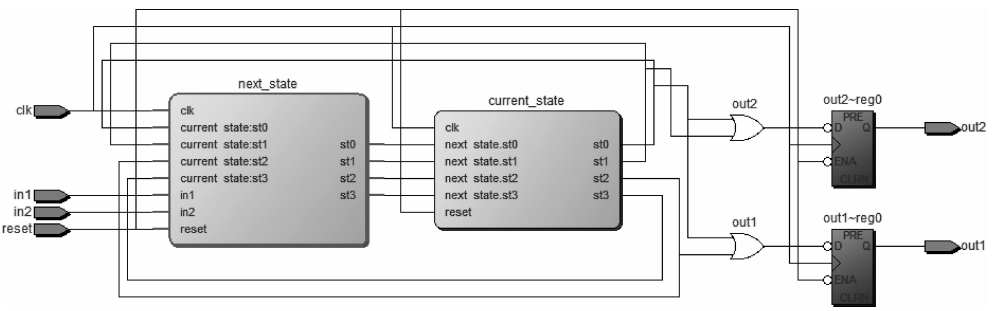

图 5-3 单进程状态机 RTL 视图

Device	EP2C5Q208C8
Timing Models	Final
Total logic elements	11 / 4,608 (< 1 %)
Total combinational functions	7 / 4,608 (< 1 %)
Dedicated logic registers	10 / 4,608 (< 1 %)
Total registers	10

图 5-4 单进程描述的状态机资源消耗报告

可以看到,图 5-3 中状态机的输出经过了一级寄存器,而单进程描述的状态机总共消耗了 11 个 LE。下面是双进程描述的同样的状态机的代码:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity statemach2 is

   port(
      clk     : in std_logic;
      reset   : in std_logic;
      in1     : in std_logic;
      in2     : in std_logic;
      out1    : out std_logic;
      out2    : out std_logic
   );
```

```
end entity;

architecture rtl of statemach2 is

type mystate is (st0, st1, st2, st3);

signal current_state,next_state : mystate;

begin
process (clk,reset)
begin
    if reset = '1' then
       current_state <= st0;
    elsif (rising_edge(clk)) then
       current_state <= next_state;
    end if;
end process;

process (in1,in2,current_state)
begin
    case current_state is
         when st0 =>
              if in1 = '1' then
                 next_state <= st1;
              end if;
              out1 <= '0';
              out2 <= '0';
         when st1 =>
              if in2 = '1' then
                 next_state <= st2;
              end if;
              out1 <= '1';
              out2 <= '0';
         when st2 =>
              if in1 = '0' and in2 = '1' then
                 next_state <= st3;
              end if;
              out1 <= '0';
              out2 <= '1';
         when st3 =>
              if in1 = '0' and in2 = '0' then
                 next_state <= st0;
              end if;
              out1 <= '1';
              out2 <= '1';
         when others => next_state <= st0;
    end case;
end process;

end rtl;
```

将上述代码在 Quartus Ⅱ 软件中进行编译。图 5-5 所示是代码编译后的 RTL 视图。图 5-6 则显示了双进程描述的状态机的资源消耗情况。

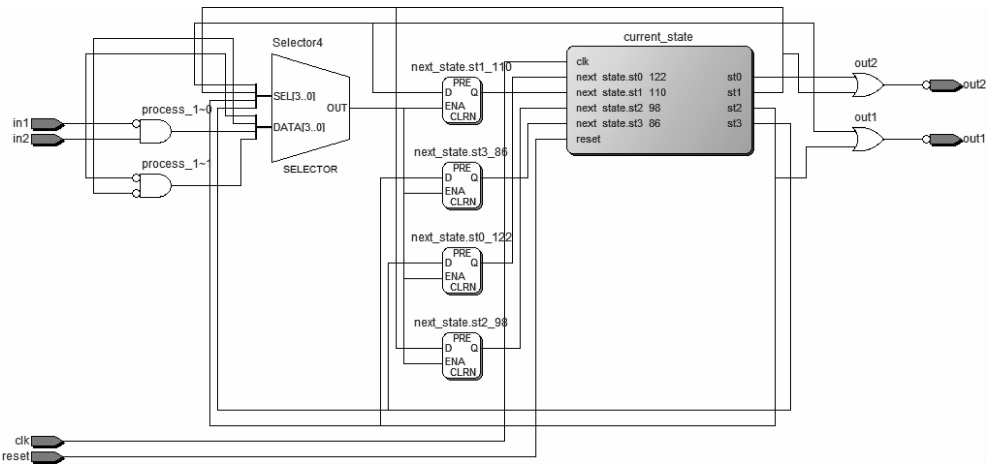

图 5-5 双进程描述的状态机 RTL 视图

Device	EP2C5Q208C8
Timing Models	Final
Total logic elements	10 / 4,608 (< 1 %)
Total combinational functions	10 / 4,608 (< 1 %)
Dedicated logic registers	4 / 4,608 (< 1 %)
Total registers	4

图 5-6 双进程描述的状态机资源消耗报告

正如代码描述那样,看到图 5-5 中,状态机的输出直接由组合逻辑驱动,但是该状态机只消耗了 10 个 LE 资源(10 个查找表和 4 个寄存器)。最后,来看看三进程描述的状态机:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity statemach3 is

   port(
      clk     : in std_logic;
      reset   : in std_logic;
      in1     : in std_logic;
      in2     : in std_logic;
      out1    : out std_logic;
      out2    : out std_logic
   );

end entity;
```

```
architecture rtl of statemach3 is

type mystate is (st0, st1, st2, st3);

signal current_state,next_state : mystate;

begin
process (clk,reset)
begin
    if reset = '1' then
       current_state <= st0;
    elsif (rising_edge(clk)) then
       current_state <= next_state;
    end if;
end process;

process (in1,in2,current_state)
begin
    case current_state is
         when st0 =>
              if in1 = '1' then
                 next_state <= st1;
              end if;
         when st1 =>
              if in2 = '1' then
                 next_state <= st2;
              end if;
         when st2 =>
              if in1 = '0' and in2 = '1' then
                 next_state <= st3;
              end if;
         when st3 =>
              if in1 = '0' and in2 = '0' then
                 next_state <= st0;
              end if;
         when others => next_state <= st0;
    end case;
end process;

process (clk,reset)
begin
    if reset = '1' then
       out1 <= '0';
       out2 <= '0';
    elsif (rising_edge(clk)) then
       case current_state is
            when st0 =>
```

```
                out1 <= '0';
                out2 <= '0';
            when st1 =>
                out1 <= '1';
                out2 <= '0';
            when st2 =>
                out1 <= '0';
                out2 <= '1';
            when st3 =>
                out1 <= '1';
                out2 <= '1';
            when others =>
                out1 <= '0';
                out2 <= '0';
        end case;
    end if;
end process;

end rtl;
```

将上述代码在 Quartus Ⅱ 软件中进行编译。图 5-7 所示是代码编译后的 RTL 视图。图 5-8 则显示了三进程描述的状态机的资源消耗情况。

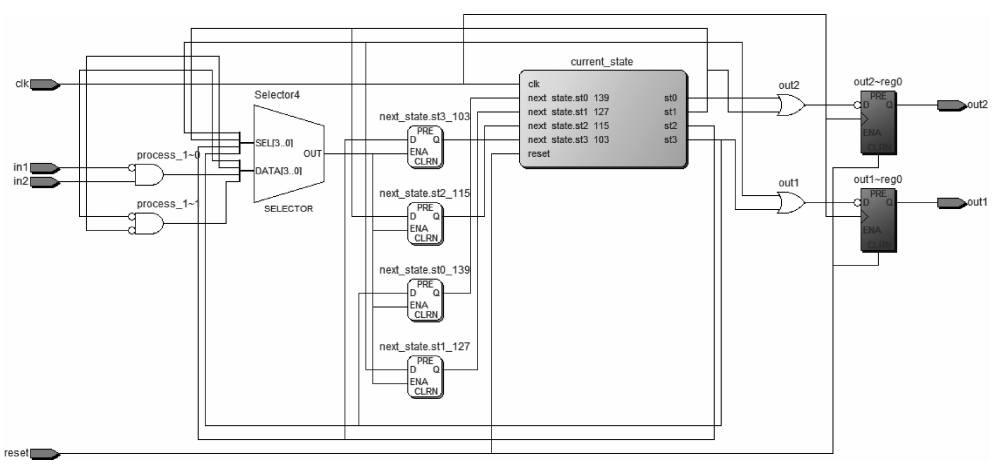

图 5-7　三进程描述的状态机 RTL 视图

Device	EP2C5Q208C8
Timing Models	Final
Total logic elements	10 / 4,608 (< 1 %)
Total combinational functions	10 / 4,608 (< 1 %)
Dedicated logic registers	6 / 4,608 (< 1 %)
Total registers	6

图 5-8　三进程描述的状态机资源消耗报告

看到图 5-7 中，状态机的输出同样经过了一级寄存，但是该状态机只消耗了 10 个 LE 资源(10 个查找表和 6 个寄存器)，可以看到三进程描述的状态机比双进程描述的状态机只是多消耗了两个寄存器，即状态机输出寄存所需要的寄存器。

5.4 设计综合工具能够识别的状态机

根据前面几节介绍的内容，应该可以设计安全的状态机。为了使设计出来的状态机能够被综合工具所识别，确保工具能够理解设计出来的状态机的属性并对其进行优化，本节再来讨论一些设计状态机的其他注意事项。

前面几节介绍的状态机模型以及代码示例都是基于 VHDL 的，本节将首先介绍一些 VHDL 和 Verilog 通用的在设计状态机方面的指导原则。只有当综合工具能够识别状态机时，工具才会去减小设计的面积和提升设计的性能。为了达到此目的，笔者建议大家不管是使用 VHDL 还是 Verilog，在设计状态机时应尽量遵循以下原则：

- 给状态机的输出分配默认值，防止综合器产生不必要的锁存器。
- 将状态机逻辑和所有的算术逻辑功能以及数据路径分离，包括与状态机输出值的分配分离，这也是为何推荐大家尽量使用多进程来描述状态机的原因。
- 如果设计中包含一个在多个状态都要使用的运算，那么在状态机外面定义这个运算，然后让状态机的输出逻辑来使用该运算结果。
- 使用简单的同步或异步复位来确保状态机定义了一个上电初始状态。如果状态机设计中，复位逻辑比较复杂，比如同时使用了一个异步复位和异步加载信号来定义初始状态，那么综合工具只会将状态机综合成普通逻辑。

如果由于器件或系统原因使得状态机进入非法状态，那么设计在下一次复位之前都不可能再正常工作。对于这种情况，综合工具默认是无法解决这种情况的。同样，设计中其他寄存器遇到类似故障，默认情况下也是无法寻求综合工具帮助的。假如设计并不会或并不需要刻意进入这种非法状态，那么给设计增加一个 default 或 when others 语句并不会影响设计的正常操作。如果状态机根本不可能会出现这种状态，那么综合工具会删除由一个默认状态产生的任何逻辑。

许多综合工具(包括 Quartus Ⅱ 软件中的综合工具)，都包含有一个实现安全状态机的选项。软件的这个选项会给设计插入额外的逻辑来探测非法状态并迫使状态机在遇到这类状态时转换到复位状态。这个选项在状态机可能会进入非法状态的情况下常常会用到。一般这种状况多发生在状态机的输入信号中有来自其他时钟域的控制信号，比如来自异步 FIFO 的控制信号。

这个选项仅仅是通过强迫状态机进入复位状态来保护状态机的，设计中的其他寄存器并不能通过这种方式来保护。如果设计中有异步输入，笔者建议大家参考并使用前面时钟域管理一章(第 2 章)介绍的同步器方法。

5.4.1 采用 Verilog 编写

为了更好地让综合工具识别 Verilog 编写的状态机,除了注意前面介绍的指导原则外,下面再单独讨论一些只针对 Verilog 的注意事项。

如果状态机无法被综合工具所识别,那么状态机将会被综合成普通的逻辑门和寄存器,而且在相关的综合编译报告中的状态机区域也将会看不到该状态机的任何信息。在这种情况下,软件将无法执行针对状态机的任何优化。下面具体列出针对 Verilog 编写状态机的相关指导原则:

- 如果使用的是 System Verilog,一定要使用枚举类型来描述状态机。
- 使用参数(Parameters)来对状态机进行分配,因为参数会使得状态机易读以及减少编码过程中的错误。
- 尽管 Quartus Ⅱ能够识别出整数,笔者不建议大家直接使用整数来定义状态值。
- 如果在状态转换逻辑中使用了下例所示的算术运算,那么 Quartus Ⅱ将不会识别状态机。

```
case (state)
0: begin
if (ena) next_state <= state + 2;
else next_state <= state + 1;
end
1: begin
...
endcase
```

- 如果将状态变量作为输出,那么 Quartus Ⅱ将无法识别状态机。
- 状态机中使用有符号变量,那么 Quartus Ⅱ将无法识别状态机。

只要按照以上原则来设计状态机,那么就能设计出安全且能被综合工具识别的状态机,最后给出一个由 Verilog 编写的状态机代码示例。这个状态机有 5 个状态。异步复位将状态变量 state 设置到 state_0,在状态 state_1 和 state_2 中,输入 in_1 和 in_2 的和将作为状态机的输出。临时变量 tmp_out_0 和 tmp_out_1 分别用于保存输入 in_1 和 in_2 的求和结果与它们的差。这里在状态机的不同状态中使用这些临时变量,可以确保设计在那些互斥的状态之间正确地共享运算结果,由此节省了设计资源。

```
module verilog_fsm (clk, reset, in_1, in_2, out);
input clk, reset;
input [3:0] in_1, in_2;
output [4:0] out;
parameter state_0 = 3'b000;
parameter state_1 = 3'b001;
parameter state_2 = 3'b010;
parameter state_3 = 3'b011;
parameter state_4 = 3'b100;
reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
reg [2:0] state, next_state;
```

```
always @ (posedge clk or posedge reset)
begin
if (reset)
    state <= state_0;
else
    state <= next_state;
end
always @ ( * )
begin
    tmp_out_0 = in_1 + in_2;
    tmp_out_1 = in_1 - in_2;
    case (state)
        state_0: begin
            tmp_out_2 = in_1 + 5'b00001;
            next_state = state_1;
        end
        state_1: begin
            if (in_1 < in_2) begin
                next_state = state_2;
                tmp_out_2 = tmp_out_0;
            end
            else begin
                next_state = state_3;
                tmp_out_2 = tmp_out_1;
            end
        end
        state_2: begin
            tmp_out_2 = tmp_out_0 - 5'b00001;
            next_state = state_3;
        end
        state_3: begin
            tmp_out_2 = tmp_out_1 + 5'b00001;
            next_state = state_0;
        end
        state_4:begin
            tmp_out_2 = in_2 + 5'b00001;
            next_state = state_0;
        end
        default:begin
            tmp_out_2 = 5'b00000;
            next_state = state_0;
        end
    endcase
end
assign out = tmp_out_2;
endmodule
```

5.4.2 采用 VHDL 编写

同样为了确保综合工具能够正确识别状态机，建议大家在使用 VHDL 编写状态机的时

候使用枚举类型来定义状态值。和 Verilog 一样,使用枚举可以增加代码易读性,并减少编码时出错的概率。如果未使用枚举类型来分配状态值,软件工具将不会正确识别状态机,此时状态机将会被综合成普通的逻辑,这样也就不会执行专门针对状态机的优化。

用 VHDL 编写状态机可以参考前面给出的实例代码。只是这里需要再次强调一定要处理好未定义的状态,噪声以及硬件上的虚假事件都可能使状态机进入不确定的状态。设计者如果没有正确对待那些未被定义的状态,那么很可能会导致硬件出现神秘的"死锁"状况。对于综合工具来说,需要设计者使用明显的代码语句来明示这些未定义状态,对于 VHDL 来说,就是使用前面提到的 when others 语句。

现在有很多编译工具都提供状态机安全选项,Quartus Ⅱ软件的集成综合工具也不例外,但是为了不把所有的工作都交给工具,通过一个例子来说明如何设计安全状态机。如图 5-9 所示,只是一个简单的包含 5 个转移状态的状态机,使用二进制对状态进行编码。

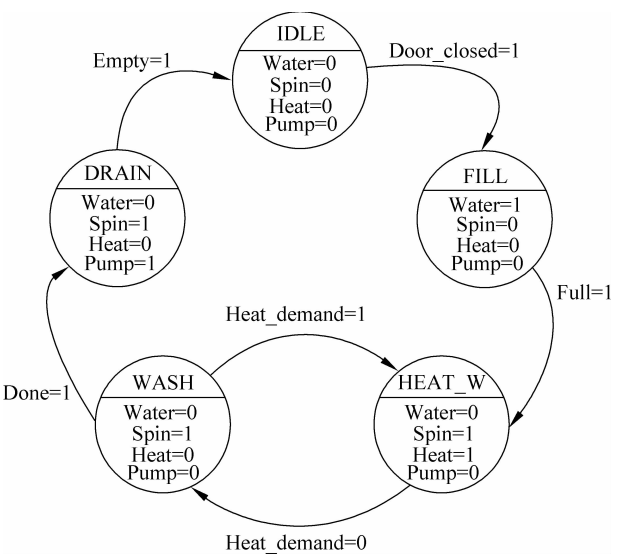

图 5-9 包含 5 个状态的状态机举例

图 5-9 所示的状态机其通常的代码实现如下:

```
TYPE state_type IS
        (idle, fill, heat_w, wash, drain);
SIGNAL state, next_state : state_type;
PROCESS (state, door_closed, full, heat_demand, done, empty)
BEGIN
CASE state is
    WHEN idle = >
            next_state < = fill WHEN door_closed = '1' ELSE state;
    WHEN fill = >
            next_state < = heat_w WHEN full = '1' ELSE state;
    WHEN heat_w = >
            next_state < = wash WHEN heat_demand = '0'
                ELSE state;
    WHEN wash = >
```

```
            next_state <= drain WHEN done = '1' ELSE state;
    WHEN drain =>
            next_state <= idle WHEN empty ELSE state;
    WHEN others =>
            next_state <= idle;
END CASE;
END PROCESS;
```

大家考虑一下，该状态机是否依然安全呢？

其实，上述代码并未考虑到未定义的状态，WHEN others 语句仅仅考虑了其他未出现在 case 语句入口的枚举状态，而状态 101、110、111 并未考虑。安全的做法是将上述代码中的状态定义修改为：

```
TYPE state_type IS
        (idle, fill, heat_w, wash, drain, unused1, unused2, unused3);
```

5 个状态需要 3 比特来定义状态变量，上述修改后的状态定义补齐了本状态机未使用到的状态，这样所有状态都已经被考虑到了。

上述代码未展示状态机的输出，前面已经讨论到，为了避免组合逻辑产生的毛刺，可以对状态机的输出进行寄存。图 5-10 则是对典型状态机的输出添加寄存器之后的框图。

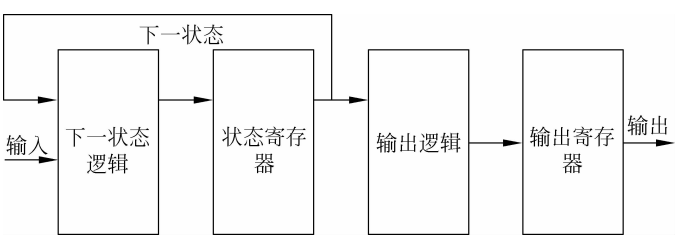

图 5-10　输出寄存后的状态机框图

按照图 5-10 来继续描述图 5-9 所示状态机的输出逻辑，分别由两个进程组成，一个用于描述输出逻辑（组合逻辑代码）；另一个用于寄存器状态机的输出（时序逻辑），其代码分别为：

组合逻辑描述的状态机输出逻辑：

```
PROCESS (state)
BEGIN
    water_i <= '0';
    spin_i <= '0';
    heat_i <= '0';
    pump_i <= '0';
    CASE state IS
        WHEN idle =>
        WHEN fill =>
            water_i <= '1';
        WHEN heat_w =>
            spin_i <= '1';
            heat_i <= '1';
        WHEN wash =>
```

```
            spin_i <= '1';
        WHEN drain =>
            spin_i <= '1';
            pump_i <= '1';
    END CASE;
END PROCESS;
```

时序逻辑完成的状态机输出寄存器:

```
PROCESS (clk)
BEGIN
IF rising_edge (clk) THEN
    water <= water_i;
    spin <= spin_I;
    heat <= heat_I;
    pump <= pump_I;
END IF
END PROCESS
```

另外,在进行状态机设计时,不管是使用 Verilog 还是 VHDL,都要注意尽量将那些计数、时间统计以及算术运算等功能从状态机内部移除,并在状态机外部进行明确的描述。这样做的好处是可以大大减少状态机逻辑的消耗并提升其性能,如图 5-11 所示,状态机中过多的运算导致其过长的等待时间。

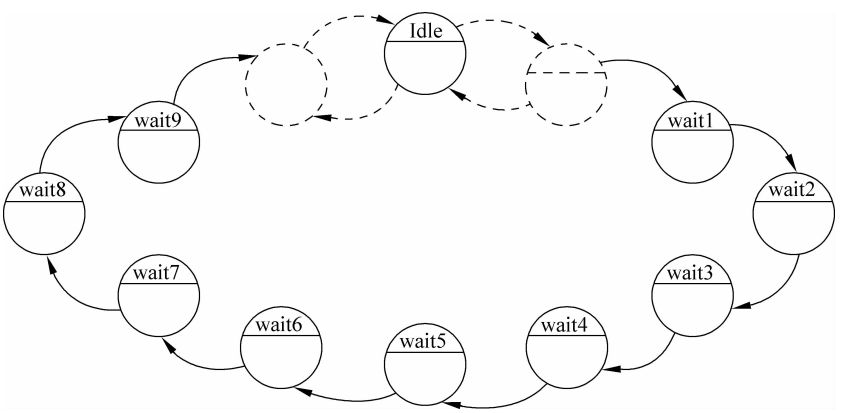

图 5-11 状态机中过多的运算功能带来更多的等待时间

合理的做法是将这些功能放到状态机外部进行描述,如图 5-12 所示,这样只需要状态机的输出对这些功能进行控制即可,同时外部功能可以产生一个控制信号来作为状态机的输入。

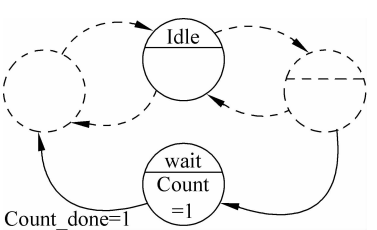

图 5-12 计数器位于状态机外部

5.5　小结

状态机在实际逻辑电路设计中经常会使用到，而且状态编码方式以及状态机的编写方式多样，大家一定要根据上面介绍的原则来采用合适的方式、方法来编写安全且可以识别的状态机。建议使用单进程来描述简单且状态值少于 5 个的状态机，尽管单进程描述的状态机会消耗更多的资源，且有可读性较差的缺点，但是它能确保状态机的安全，另一方面，现在器件规模越来越大，有的时候并不在意这点资源。当然，在养成良好代码习惯的基础上，笔者强烈建议大家还是采用多进程来描述状态机，尤其是采样三进程及寄存状态机输出的方式来描述。