2020-11-4

数字电路实验八　状态机及键盘输入

秦嘉余

191220088

1348288404@qq.com

目录

1.实验目的

本实验的目的是了解有限状态机的 verilog 写法，同时利用状态机的有关特性，根据 PS/2 键盘的编码与数据传递方式，完成 PS/2 状态机的设计

2.实验原理

1.状态机

有限状态机是数字电路系统中十分重要的电路模块，是一种输出取决于过 去输入和当前输

入的时序逻辑电路，它是组合逻辑电路和时序逻辑电路的组 合。其中组合逻辑分为两个部分，一个是用于产生有限状态机下一个状态的次 态逻辑，另一个是用于产生输出信号的输出逻辑，次态逻辑的功能是确定有限 状态机的下一个状态；输出逻辑的功能是确定有限状态机的输出。除了输入和 输出外，状态机还有一组具有“记忆”功能的寄存器，这些寄存器的功能是记忆有限状态机的内部状态，常被称作状态寄存器。在实际应用中，有限状态机被分为两种：Moore（摩尔）型有限状态机和 Mealy（米里）型有限状态机。

2.PS/2 键盘

PS/2 接口使用两根信号线，一根信号线传输时钟 PS2_CLK，另一根传输 数据 PS2_DAT。时钟信号主要用于指示数据线上的比特位在什么时候是有效 的。键盘和主机间可以进行数据双向传送，这里只讨论键盘向主机传送数据 的情况。当 PS2_DAT 和 PS2_CLK 信号线都为高电平（空闲）时，键盘才可以 给主机发送信号。如果主机将 PS2_CLK 信号置低，键盘将准备接受主机发来的命令。
当用户按键或松开时，键盘以每帧 11 位的格式串行传送数据给主机，同 时在 PS2_CLK 时钟信号上传输对应的时钟（一般为 10.0–16.7kHz）。第一位是 开始位（逻辑 0），后面跟 8 位数据位（低位在前），一个奇偶校验位（奇校 验）和一位停止位（逻辑 1）。每位都在时钟的下降沿有效。

3.实验环境/器材

本次实验的环境为 Quartus17.1 版本
本次实验的器材为 DE10 Standard 开发板

4. 状态机设计

1.设计思路

状态机的设计中，如果是摩尔型状态机，只需要根据原理分别在两个 always 语句块中完成状态的转化已经输出即可

2. 实验代码

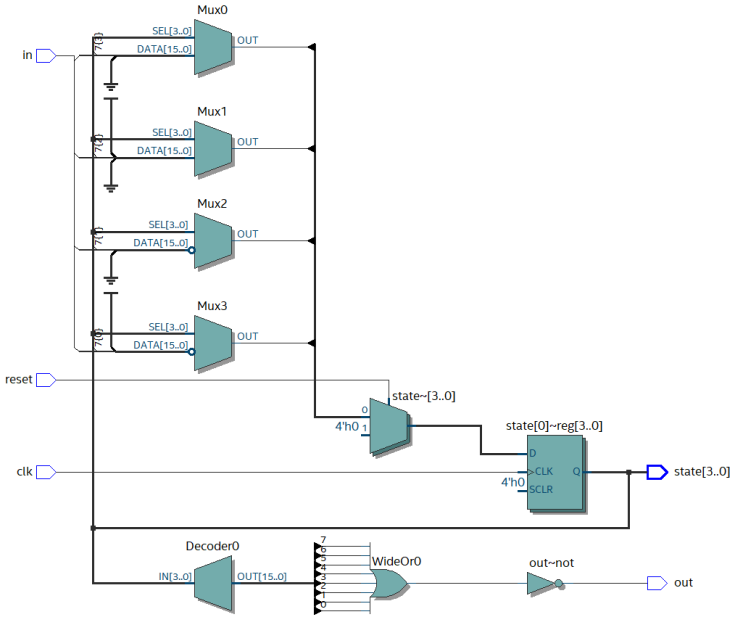
模块代码

```
module FSM_bin (
    input clk, input in, input reset,
    output reg out,output reg [3:0] state
);

    parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4, S5 = 5, S6 = 6, S7 = 7, S8 = 8;
// Output depends only on the state
    always @ (state)
    begin
       case (state)
          S0:   out=1'b0;
          S1:   out=1'b0;
          S2:   out=1'b0;
          S3:   out=1'b0;
          S4:   out=1'b0;
          S5:   out=1'b0;
          S6:   out=1'b0;
          S7:   out=1'b0;
          S8:   out=1'b1;
          default: out=1'bx;
       endcase
    end
// Determine the next state
    always @ (posedge clk)
    begin
       if (reset)
          state <= S0;
       else
          case (state)
          S0: if (in) state <= S5; else state <= S1;
          S1: if (in) state <= S5; else state <= S2;
          S2: if (in) state <= S5; else state <= S3;
          S3: if (in) state <= S5; else state <= S4;
          S4: if (in) state <= S5; else state <= S4;
          S5: if (in) state <= S6; else state <= S1;
          S6: if (in) state <= S7; else state <= S1;
          S7: if (in) state <= S8; else state <= S1;
          S8: if (in) state <= S8; else state <= S1;
          endcase

end endmodule
```

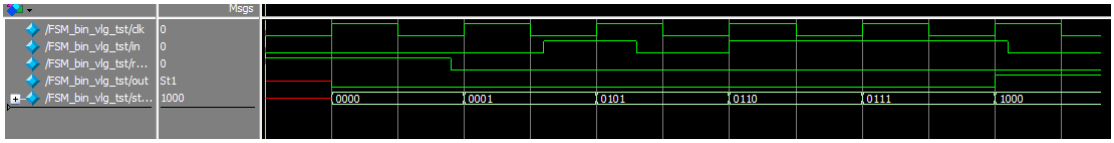
测试代码

```
// assign statements (if any)
FSM_bin i1 (
// port map - connection between master ports and sign
    .clk(clk),
    .in(in),
    .out(out),
    .reset(reset),
    .state(state)
);
initial
begin
// code that executes only once
// insert code here --> begin
    clk=0; reset=1;in=0;
    #7 reset=1;
    #7 reset=0;
    #7 in=1;
    #7 in=0;
    #7 in=1;
    #7 in=1;
    #7 in=1;
    #7 in=0;
    #7
$stop;
// --> end
$display("Running testbench");
end
always
// optional sensitivity list
// @(event1 or event2 or .... eventn)
begin
// code executes for every event on sensitivity list
// insert code here --> begin
    #5 clk=~clk;
//@eachvec;
// --> end
end
endmodule
```

3.RTL 视图



4.仿真结果



5.引脚分配

clk	Input	PIN_AB30	5B	B5B_N0	2.5 V (default)		12mA (default)
in	Input	PIN_Y27	5B	B5B_N0	2.5 V (default)		12mA (default)
out	Output	PIN_AC22	4A	B4A_N0	2.5 V (default)		12mA (default)
reset	Input	PIN_AB28	5B	B5B_N0	2.5 V (default)		12mA (default)
state[3]	Output	PIN_AD24	4A	B4A_N0	2.5 V (default)		12mA (default)
state[2]	Output	PIN_AC23	4A	B4A_N0	2.5 V (default)		12mA (default)
state[1]	Output	PIN_AB23	5A	B5A_N0	2.5 V (default)		12mA (default)
state[0]	Output	PIN_AA24	5A	B5A_N0	2.5 V (default)		12mA (default)

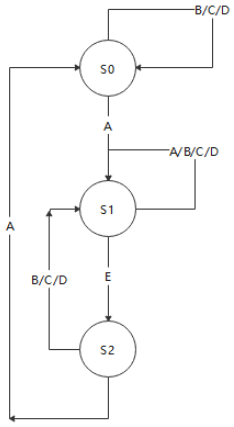
6.实验结果

已在开发板上观察结果

5. 键盘状态机设计

1.设计思路

在键盘中，由于键码的复杂特殊性，如果使用前面实验采用的传统设计思路，很难完成很多功能的同步，所以需要采用状态的思路将键盘所处的状态分类。但是由于此处，键盘的键码很多，接近无限状态的状态机，所以在设计状态机的时候，很多情况不能单独作为状态来转化，只能在状态转化的同时，利用标识位来标记，只把基础的功能对应的转化来作为状态。例如，按下，松开 A-Z 时状态转化，而 caps，shift 键等则只改变表示位，不进行转化。
我们把键码作为输入，将 ASCII 可以对应的 A-Z,0-9 键码作为一类，统称为输入 A; caps，shift，ctrl 分别作为输入 B,C,D，F0 对应的断码作为输入 E。
由于输出比较复杂,需要综合分析，所以无法表示在状态机的转化图中。
状态则分为 3 种，S0,S1,S2。其中 S0 为没有输入 ASCII 码对应键的状态，S1 为输入了 ASCII 码对应键的状态，S2 为输入断码 F0 后的状态
下面为设计的状态机的示意图。



2.实验代码

以下为主模块，调用键盘模块 ps2_keyboard,ascii 译码模块，以及 out 模块来确定 data 的输入

```
module Keyboard(clk,clrn,reset,ps2_clk,ps2_data,data,ready,overflow,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,caps,shift,ctrl,state);
    input clk,clrn,reset;            //SW[0],SW[1]
    input ps2_clk,ps2_data;
    wire nextdata_n;
    output [7:0] data;
    wire [7:0] kbdata;
    output ready;         //LEDR[6]
    output overflow;      //LEDR[7]
    output [6:0] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5;
    output caps,shift,ctrl;     //LEDR[3],LEDR[4],LEDR[5]
    output [1:0] state;         //LEDR[1:0]


    out o(kbdata,ps2_clk,ready,data,nextdata_n);


    ps2_keyboard i(clk,clrn,ps2_clk,ps2_data,kbdata, ready,nextdata_n,overflow);
    ascii2 display(reset,nextdata_n,data,ready,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,caps,shift,ctrl,state);

endmodule
```

以下为键盘 ps/2 模块

```
module ps2_keyboard(clk,clrn,ps2_clk,ps2_data,data, ready,nextdata_n,overflow);
   input clk,clrn,ps2_clk,ps2_data;
   input nextdata_n;
   output [7:0] data;
   output reg ready;
   output reg overflow; // fifo overflow
   // internal signal, for test
   reg [9:0] buffer; // ps2_data bits
   reg [7:0] fifo[7:0]; // data fifo
   reg [2:0] w_ptr,r_ptr;  // fifo write and read pointers
   reg [3:0] count; // count ps2_data bits
   // detect falling edge of ps2_clk
   reg [2:0] ps2_clk_sync;


   always @(posedge clk)
   begin
      ps2_clk_sync <= {ps2_clk_sync[1:0],ps2_clk};
   end


   wire sampling = ps2_clk_sync[2] & ~ps2_clk_sync[1];


   always @(posedge clk)
   begin
      if (clrn == 0) // reset
      begin
            count <= 0; w_ptr <= 0; r_ptr <= 0; overflow <= 0; ready<= 0;
      end
      else
      begin
         if ( ready ) // read to output next data
         begin
            if(nextdata_n == 1'b0) //read next data
            begin
               r_ptr <= r_ptr + 3'b1;
               if(w_ptr==(r_ptr+1'b1)) //empty
                  ready <= 1'b0;
            end
         end
         if (sampling)
         begin
            if (count == 4'd10)
```

以下为 out 模块

```
module out(kbdata,ps2_clk,ready,data,nextdata_n);
    input [7:0] kbdata;
    input ps2_clk,ready;
    output reg [7:0] data;
    output reg nextdata_n;


    initial
    begin
        nextdata_n<=1;
        data<=8'b00000000;
    end

    always @(ps2_clk)
    begin
        if(ready)
        begin
            nextdata_n<=0;
            data<=kbdata;
        end
        else
        begin
            nextdata_n<=1;
        end
    end

endmodule
```

以下为 ascii 模块

```
module ascii #(parameter Caps=8'h58,parameter Shift=8'h12,parameter Ctrl=8'h14,parameter Break=8'hf0,parameter None=8'hff)
              (reset,data,ready,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,caps,shift,ctrl,state);
   input reset;
   input [7:0] data;
   input ready;
   output [6:0] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5;
   reg [7:0] ascii;
   reg [7:0] count;
   //reg n_cnt;
   //reg is_over;
   output reg [1:0] state;
   output reg caps,shift,ctrl;
   reg is_last_f0;
   reg is_caps;

   wire en;
   (* ram_init_file = "ascii.mif" *)reg [7:0] myrom [255:0];

   parameter S0=0,S1=1,S2=2;
   initial begin
      count=0;
      state=S0;
      caps=0;
      shift=0;
      ctrl=0;
      is_last_f0=0;
      is_caps=0;
   end


   assign en=~(caps==0&&shift==0&&ctrl==0&&state==S0);


   decode_hex d_hex0(data[3:0],en,HEX0);
   decode_hex d_hex1(data[7:4],en,HEX1);



   always @(posedge ready or posedge reset)
   begin
   decode_ascii_hex d_hex2(ascii[3:0],en,is_caps,shift,HEX2);
   decode_ascii_hex d_hex3(ascii[7:4],en,is_caps,shift,HEX3);

   decode_hex d_hex4(count[3:0],1'b1,HEX4);
   decode_hex d_hex5(count[7:4],1'b1,HEX5);

endmodule
```

状态转移以及标志位的设置不在此赘述，可以查看压缩包内的相关文件

同时有显示输入的 decode_hex 模块

```
module decode_hex(in,en,HEX);
    input [3:0] in;
    input en;
    output reg [6:0] HEX;

    always @(in or en)
    begin
       if(en)
          case (in)
          0: HEX=7'b1000000;
          1: HEX=7'b1111001;
          2: HEX=7'b0100100;
          3: HEX=7'b0110000;
          4: HEX=7'b0011001;
          5: HEX=7'b0010010;
          6: HEX=7'b0000010;
          7: HEX=7'b1111000;
          8: HEX=7'b0000000;
          9: HEX=7'b0010000;
          10:HEX=7'b0001000;
          11:HEX=7'b0000011;
          12:HEX=7'b1000110;
          13:HEX=7'b0100001;
          14:HEX=7'b0000110;
          15:HEX=7'b0001110;
          default: HEX=7'b1111111;
          endcase
       else
          HEX=7'b1111111;
    end
endmodule
```

为了方便大小写 ascii 码的输出，对上文件更改完成一个改进的 decode_ascii_hex 模块

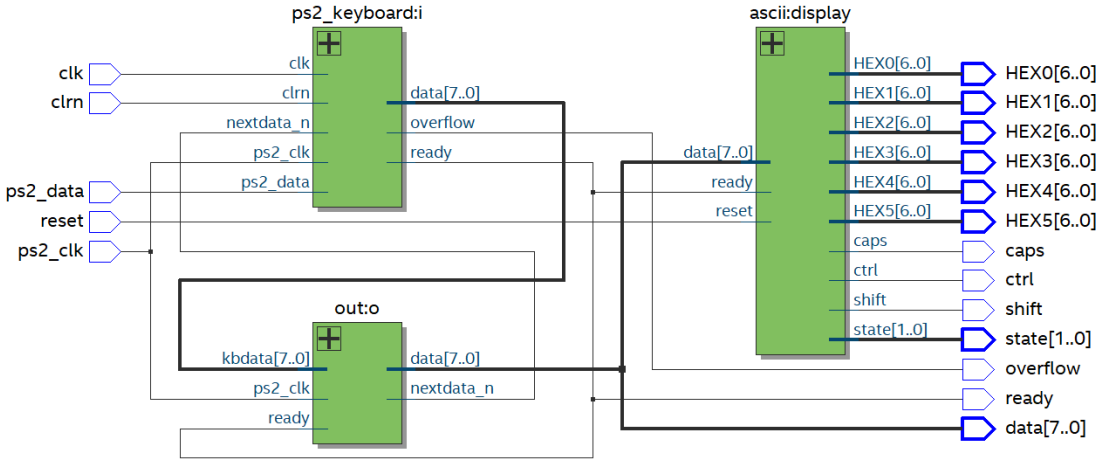
```
module decode_ascii_hex(in,en,is_caps,shift,HEX);
   input [3:0] in;
   input en;
   input shift,is_caps;
   output reg [6:0] HEX;

   wire is_upper;
   wire [3:0] _in;


   assign is_upper=shift^is_caps;

   assign _in=(in<=8'h7a&&in>=8'h61)?(is_upper?(in-32):in):in;


   always @(_in or en)
   begin
      if(en)
         case (_in)
         0: HEX=7'b1000000;
         1: HEX=7'b1111001;
         2: HEX=7'b0100100;
         3: HEX=7'b0110000;
         4: HEX=7'b0011001;
         5: HEX=7'b0010010;
         6: HEX=7'b0000010;
         7: HEX=7'b1111000;
         8: HEX=7'b0000000;
         9: HEX=7'b0010000;
         10:HEX=7'b0001000;
         11:HEX=7'b0000011;
         12:HEX=7'b1000110;
         13:HEX=7'b0100001;
         14:HEX=7'b0000110;
         15:HEX=7'b0001110;
         default: HEX=7'b1111111;
         endcase
      else
         HEX=7'b1111111;
   end
endmodule
```

以下为测试代码的主要内容

```
`timescale 1ns / 1ps
module ps2_keyboard_model(output reg ps2_clk,output reg ps2_data);
   parameter [31:0] kbd_clk_period = 60;
   initial ps2_clk = 1'b1;

task kbd_sendcode;
   input [7:0] code; // key to be sent
   integer i;
   reg[10:0] send_buffer;
   begin
      send_buffer[0] = 1'b0; // start bit
      send_buffer[8:1] = code; // code
      send_buffer[9] = ~(^code); //old parity bit
      send_buffer[10] = 1'b1; // stop bit
      i = 0;
      while( i < 11) begin // set kbd_data
         ps2_data = send_buffer[i];
         #(kbd_clk_period/2) ps2_clk = 1'b0;
         #(kbd_clk_period/2) ps2_clk = 1'b1;
         i = i + 1;
      end
   end
endtask
endmodule
```

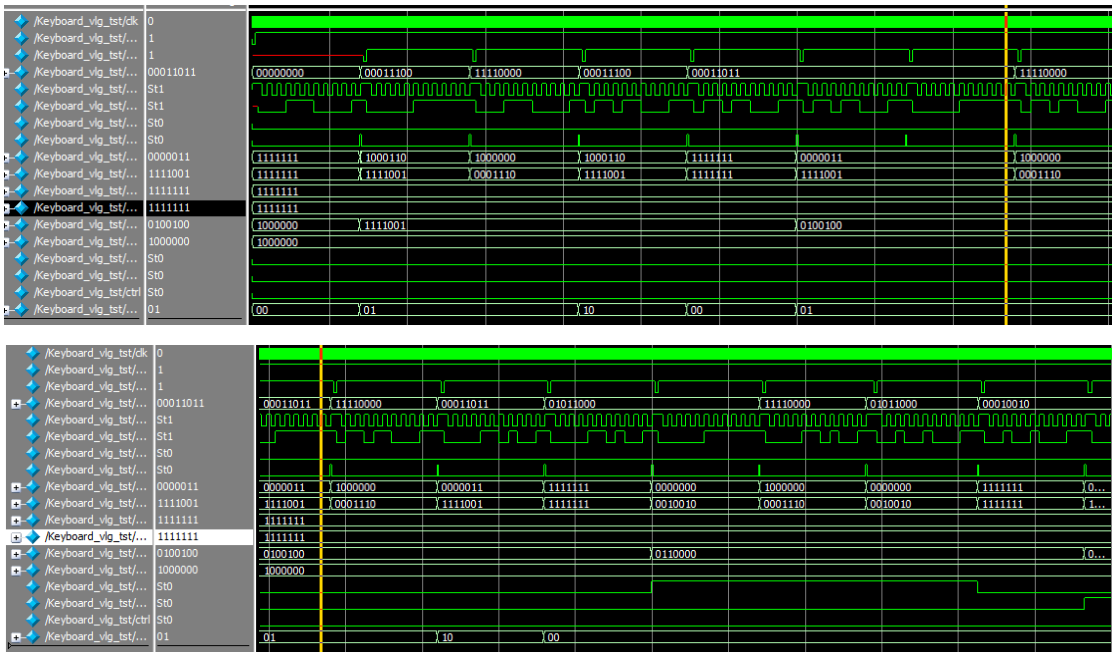
```
clrn = 1'b0; #20; clrn = 1'b1; #20;
model.kbd_sendcode(8'h1C); // press 'A'
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1;//read data
model.kbd_sendcode(8'hF0); // break code
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h1C); // release 'A'
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h1B); // press 'S'
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h1B); // press 'S'
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h1B); // press 'S'
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'hF0); // break code
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h1B); // release 'S'
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h58);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h58);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'hF0);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h58);
#50 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h12);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h12);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'hF0);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h12);
#50 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h14);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h14);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'hF0);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h14);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h58);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h58);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'hF0);
#20 nextdata_n =1'b0; #20 nextdata_n =1'b1; //read data
model.kbd_sendcode(8'h58);
```

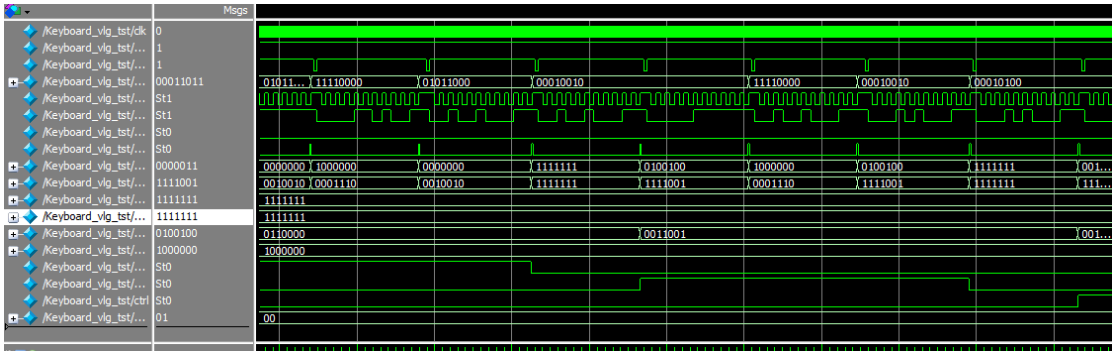
3.RTL 视图



每个模块内单独的 RTL 视图不易看清，则不再展示

4.仿真结果





5.引脚分配

out caps	Output	PIN_AD24	4A	B4A_N0	2.5 V (default)
in clk	Input	PIN_AA16	4A	B4A_N0	2.5 V (default)
in clrn	Input	PIN_AB30	5B	B5B_N0	2.5 V (default)
out ctrl	Output	PIN_AF25	4A	B4A_N0	2.5 V (default)

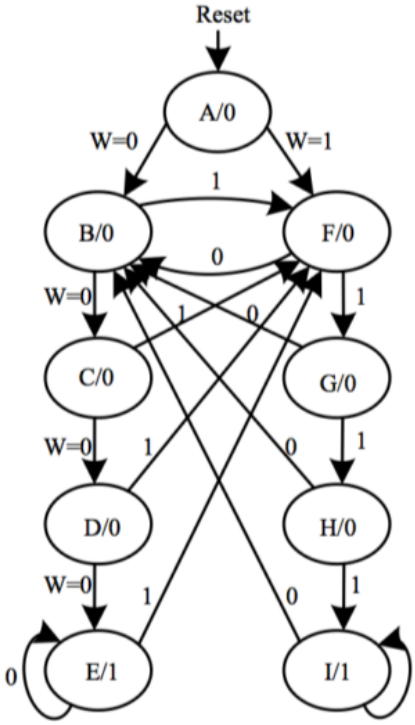
out overflow	Output	PIN_AF24	4A	B4A_N0	2.5 V (de
in ps2_clk	Input	PIN_AB25	5A	B5A_N0	2.5 V (de
in ps2_data	Input	PIN_AA25	5A	B5A_N0	2.5 V (de
out ready	Output	PIN_AE24	4A	B4A_N0	2.5 V (de
in reset	Input	PIN_Y27	5B	B5B_N0	2.5 V (de
out shift	Output	PIN_AG25	4A	B4A_N0	2.5 V (de
out state[1]	Output	PIN_AB23	5A	B5A_N0	2.5 V (de
out state[0]	Output	PIN_AA24	5A	B5A_N0	2.5 V (de

HEX 引脚过多，不在此展示

6.实验结果

本实验已在开发板上验收

6.思考题



上述为一个摩尔型的状态机设计实例，请查阅相关资料，研究米里型状态 的设计与此有何不同?

米里型状态机与摩尔型状态机最大区别为摩尔型状态机的输出至于当前的状态有关，而米里型状态机的输入不光与当前的状态有关，也与当前的输入有关，所以在设计米里型状态机时，在判断输出的 always 语句块中，敏感信号需要同时含有输入与状态，根据两者综合判断输出。当然，在这种情况下，状态的转化与输出在同一个 always 语句块内进行有时也更为简便。总之，需要根据实验的具体要求判断采用那种写法。

7.实验实验中遇到的问题及解决办法

问题 1

本次实验的内容比以往的实验都要复杂很多，如果不进行单独模块的仿真测试，很容易出现遇到问题无法下手，没办法找出问题出现的模块的情况，所以需要尽量对单独模块进行仿真测试，与此同时，还可以将一些信号引入顶层模块输出在指示灯上，判断相关的信号是否正确。

问题 2

在实验伊始，采用了传统顺序执行的思路，不断判读状态，在写了一段时间后发现这样过于复杂，而难以修改。所以重新设计了状态机，好好思考了各种情况，画出了相关的状态转移图，再重新开始写程序。此后在思路上才顺畅了很多。所以，在写程序时，不要盲目开始写，架构的设计往往更为重要，不妨在好好思考后再写，磨刀不费砍柴工。

问题 3

对于状态机的 always 块，一开始采用了错误的敏感信号，导致 count 计数多计，状态多跳，后来思考后，改为了合适的敏感信号，解决了问题

问题 4

对于 data 与 kbdata，一开始没有正确区分，直接将 kbdata 引入了 ascii 模块，后来重新读了 ps/2 键盘模块的代码与 pdf 文档了解了 kbdata 的形式，写了 out 模块根据 kbdata 获取 data，才得到了正确的键码输入

问题 5

除此之外，在这个实验中，初始化的方式也尤为重要。在仿真测试中，只需要在模块中添加 initial 块既可以完成对一些变量初始值的赋值。而 initial 块只在仿真测试中有效，在开发板上并不能很好的初始化。一开始没有注意到这个问题的时候，开发板上的初始状态总是不对，导致后面的转化一直出错，在查阅资料后，发现是初始化的问题。所以之后设置 reset 端，对变量可以随时初始化，即有效又方便。