A-PDF Split DEMO : Purchase from www.A-PDF.com to remove the watermark

CHAPTER 9

PS2 MOUSE

9.1 INTRODUCTION

A computer mouse is designed mainly to detect two-dimensional motion on a surface. Its internal circuit measures the relative distance of movement and checks the status of the buttons. For a mouse with a PS2 interface, this information is packed in three packets and sent to the host through the PS2 port. In the *stream mode*, a PS2 mouse sends the packets continuously in a predesignated sampling rate.

Communication of the PS2 port is bidirectional and the host can send a command to the keyboard or mouse to set certain parameters. For our purposes, this functionality is hardly required for a keyboard, and thus the keyboard interface in Chapter 8 is limited to one direction, from the keyboard to the FPGA host. However, unlike the keyboard, a mouse is set to be in the non-steaming mode after power-up and does not send any data. The host must first send a command to the mouse to initialize the mouse and enable the stream mode. Thus, bidirectional communication of the PS2 port is needed for the PS2 mouse interface, and we must design a transmitting subsystem (i.e., from FPGA board to mouse) for the PS2 interface.

In this chapter, we provide a short overview of the PS2 mouse protocol, design a bidirectional PS interface, and derive a simple mouse interface.

FPGA Prototyping by VHDL Examples. By Pong P. Chu
Copyright © 2008 John Wiley & Sons, Inc.

Table 9.1 Mouse data packet format

byte 1	y_v	x_v	y_8	x_8	1	m	r	l
byte 2	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
byte 3	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

9.2 PS2 MOUSE PROTOCOL

9.2.1 Basic operation

A standard PS2 mouse reports the x-axis (right/left) and y-axis (up/down) movement and the status of the left button, middle button, and right button. The amount of each movement is recorded in a mouse's internal counter. When the data is transmitted to the host, the counter is cleared to zero and restarts the counting. The content of the counter represents a 9-bit signed integer in which a positive number indicates the right or up movement, and a negative number indicates the left or down movement.

The relationship between the physical distances is defined by the mouse's *resolution* parameter. The default value of resolution is four counts per millimeter. When a mouse moves continuously, the data is transmitted in a regular rate. The rate is defined by the mouse's *sampling rate* parameter. The default value of the sampling rate is 100 samples per second. If a mouse moves too fast, the amount of the movement during the sampling period may exceed the maximal range of the counter. The counter is set to the maximum magnitude in the appropriate direction. Two overflow bits are used to indicate the conditions.

The mouse reports the movement and button activities in 3 bytes, which are embedded in three PS2 packets. The detailed format of the 3-byte data is shown in Table 9.1. It contains the following information:

- $x_8, \ldots, x_0$: x-axis movement in 2's-complement format
- x_v: x-axis movement overflow
- $y_8, \ldots, y_0$: y-axis movement in 2's-complement format
- y_v: y-axis movement overflow
- l: left button status, which is '1' when the left button is pressed
- r: right button status, which is '1' when the right button is pressed
- m: optional middle button status, which is '1' when the middle button is pressed

During transmission, the byte 1 packet is sent first and the byte 3 packet is sent last.

9.2.2 Basic initialization procedure

The operation of a mouse is more complex than that of a keyboard. It has different operation modes. The most commonly used one is the *stream mode*, in which a mouse sends the movement data when it detects movement or button activity. If the movement is continuous, the data is generated at the designated sample rate.

During the operation, a host can send commands to a mouse to modify the default values of various parameters and set the operation mode, and a mouse may generate the status and send an acknowledgment. For our purposes, the default values are adequate, and the only task is to set the mouse to the stream mode.

The basic interaction sequence between a PS2 mouse and the FPGA host consists of the following:

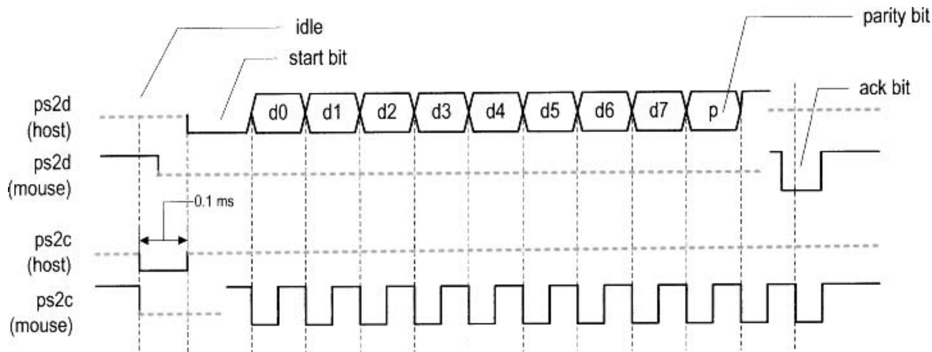

Figure 9.1 Host-to-device timing diagram of a PS2 port.

1. At power-on, a mouse performs a power-on test internally. The mouse sends 1-byte data AA, which indicates that the test is passed, and then 1-byte data 00, which is the id of a standard PS2 mouse.
2. The FPGA host sends the command, F4, to enable the stream mode. The mouse will respond with FE to acknowledge acceptance of the command.
3. The mouse now enters the stream mode and sends normal data packets.

If a mouse is plugged into the FPGA prototyping board in advance, it performs the power-on test when the power of the board is turned on and sends the AA 00 data immediately. The FPGA chip is not configured at this point and will not receive this data. Thus, we can usually ignore the power-on message in step 1. A minimal mouse interface circuit only needs to send the F4 command, check the FE acknowledge, and enter the normal operation mode to process the mouse's regular data packet.

We can force the mouse to return to the initial state by sending the reset command:

1. The FPGA host sends the command, FF, to reset the mouse. The mouse will respond with FE to acknowledge acceptance of the command.
2. The mouse performs a power-on test internally and then sends AA 00. The stream mode will be disabled during the process.

Newer mouses add more functionality, such as a scrolling wheel and additional buttons, and thus send more information. Additional bytes are appended to the original 3-byte data to accommodate these new features.

9.3 PS2 TRANSMITTING SUBSYSTEM

9.3.1 Host-to-PS2-device communication protocol

Host-to-PS2-device communication protocol involves bidirectional data exchange. The mouse's data and clock lines actually are *open-collector* circuits. For our design purposes, we treat them as tri-state lines. The basic timing diagram of transmitting a packet from a host to a PS2 device is shown in Figure 9.1, in which the data and clock signals are labeled ps2d and ps2c. For clarity, the diagram is split into two parts to show which activities are generated by the host (i.e., the FPGA chip) and which activities are generated by the device (i.e., mouse). The basic operation sequence is as follows:

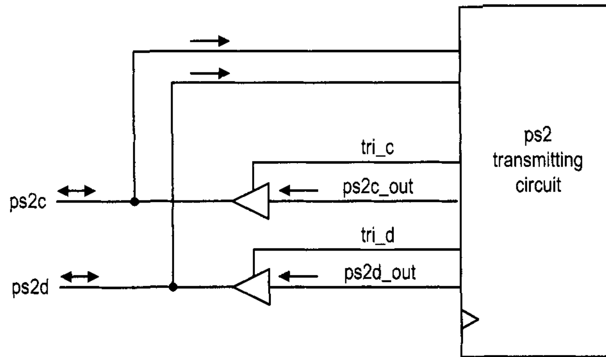

Figure 9.2 Tri-state buffers of the PS2 transmission subsystem.

1. The host forces the ps2c line to be '0' for at least 100 μs to inhibit any mouse activity. It can be considered that the host requests to send a packet.
2. The host forces the ps2d line to be '0' and disables the ps2c line (i.e., makes it high impedance). This step can be interpreted as the host sending a start bit.
3. The PS2 device now takes over the ps2c line and is responsible for future PS2 clock signal generation. After sensing the starting bit, the PS2 device generates a '1'-to-'0' transition.
4. Once detecting the transition, the host shifts out the least significant data bit over the ps2d line. It holds this value until the PS2 device generates a '1'-to-'0' transition in the ps2c line, which essentially acknowledges retrieval of the data bit.
5. Repeat step 4 for the remaining 7 data bits and 1 parity bit.
6. After sending the parity bit, the host disables the ps2d line (i.e., makes it high impedance). The PS2 device now takes over the ps2d line and acknowledges completion of the transmission by asserting the ps2d line to '0'. If desired, the host can check this value at the last '1'-to-'0' transition in the ps2c line to verify that the packet is transmitted successfully.

9.3.2 Design and code

Unlike the receiving subsystem, the ps2c and ps2d signals communicate in both directions. A tri-state buffer is needed for each signal. The tri-state interface is shown in Figure 9.2. The tri_c and tri_d signals are enable signals that control the tri-state buffers. When they are asserted, the corresponding ps2c_out and ps2d_out signals will be routed to the output ports.

To design the transmitting subsystem, we can follow the sequence of the preceding protocol to create an ASMD chart, as shown in Figure 9.3. The FSMD is initially in the idle state. To start the transmission, the host asserts the wr_ps2 signal and places the data on the din bus. The FSMD loads din, along with the parity bit, par to the shift_reg register, loads the "1···1" to c_reg, and moves to the rts (for "request to send") state. In this state, the ps2c_out is set to '0' and the corresponding tri_c is asserted to enable the corresponding tri-state buffer. The c_reg is used as a 13-bit counter to generate a 164-μs delay. The FSMD then moves to the start state, in which the PS2 clock line is disabled and the data line is set to '1'. The PS2 device (i.e., mouse) now takes over and generates

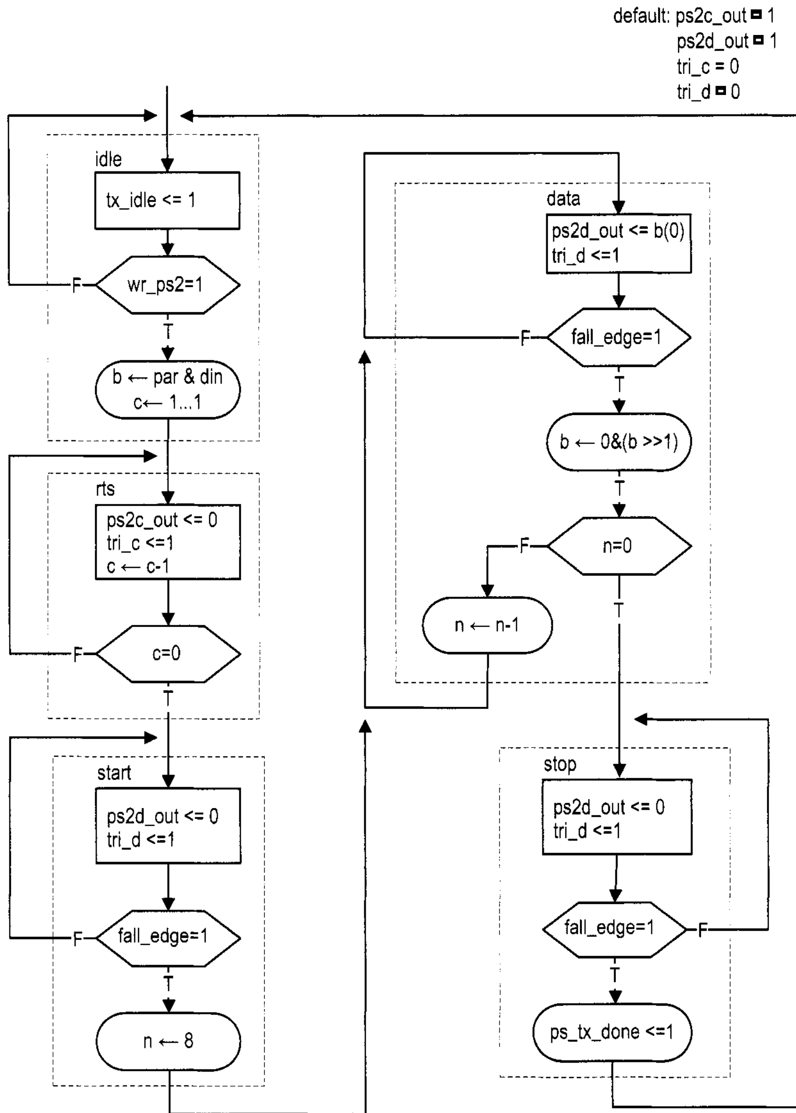

Figure 9.3 ASMD chart of the PS2 transmitting subsystem.

clock signal over the ps2c line. After detecting the falling edge of the ps2c signal through the fall_edge signal, the FSMD goes to the data state and shifts 8 data bits and 1 parity bit. The n register is used to keep track of the number of bits shifted. The FSMD then moves to the stop state, in which the data line is disabled. It returns to the idle state after sensing the last falling edge.

The FSMD also includes a tx_idle signal to indicate whether a transmission is in progress. This signal can be used to coordinate operation between the receiving and transmitting subsystems. The code follows the ASMD chart and is shown in Listing 9.1. A filtering circuit similar to that of Section 8.2 is used to generate the fall_edge signal.

Listing 9.1 PS2 port transmitter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ps2_tx is
5   port (
      clk, reset: in  std_logic;
      din: in std_logic_vector(7 downto 0);
      wr_ps2: std_logic;
      ps2d, ps2c: inout std_logic;
10    tx_idle: out std_logic;
      tx_done_tick: out std_logic
   );
end ps2_tx;

15 architecture arch of ps2_tx is
   type statetype is (idle, rts, start, data, stop);
   signal state_reg, state_next: statetype;
   signal filter_reg, filter_next: std_logic_vector(7 downto 0);
   signal f_ps2c_reg,f_ps2c_next: std_logic;
20 signal fall_edge: std_logic;
   signal b_reg, b_next: std_logic_vector(8 downto 0);
   signal c_reg,c_next: unsigned(12 downto 0);
   signal n_reg,n_next: unsigned(3 downto 0);
   signal par: std_logic;
25 signal ps2c_out, ps2d_out: std_logic;
   signal tri_c, tri_d: std_logic;
begin
   --=================================================
   -- filter and falling-edge tick generation for ps2c
30 --=================================================
   process (clk, reset)
   begin
      if reset='1' then
         filter_reg <= (others=>'0');
35       f_ps2c_reg <= '0';
      elsif (clk'event and clk='1') then
         filter_reg <= filter_next;
         f_ps2c_reg <= f_ps2c_next;
      end if;
40 end process;
```

```
   filter_next <= ps2c & filter_reg(7 downto 1);
   f_ps2c_next <= '1' when filter_reg="11111111" else
                  '0' when filter_reg="00000000" else
45                f_ps2c_reg;
   fall_edge <= f_ps2c_reg and (not f_ps2c_next);

   --=================================================
   -- fsmd
50 --=================================================
   -- registers
   process (clk, reset)
   begin
      if reset='1' then
55       state_reg <= idle;
         c_reg <= (others=>'0');
         n_reg  <= (others=>'0');
         b_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
60       state_reg <= state_next;
         c_reg <= c_next;
         n_reg <= n_next;
         b_reg <= b_next;
      end if;
65 end process;
   -- odd parity bit
   par <= not (din(7) xor din(6) xor din(5) xor din(4) xor
               din(3) xor din(2) xor din(1) xor din(0));
   -- fsmd next-state logic and data path logic
70 process(state_reg,n_reg,b_reg,c_reg,wr_ps2,
           din,par,fall_edge)
   begin
      state_next <= state_reg;
      c_next <= c_reg;
75    n_next <= n_reg;
      b_next <= b_reg;
      tx_done_tick <='0';
      ps2c_out <= '1';
      ps2d_out <= '1';
80    tri_c <= '0';
      tri_d <= '0';
      tx_idle <='0';
      case state_reg is
         when idle =>
85          tx_idle <= '1';
            if wr_ps2='1' then
               b_next <= par & din;
               c_next <= (others=>'1'); -- 2^13-1
               state_next <= rts;
90          end if;
         when rts =>  -- request to send
            ps2c_out <= '0';
            tri_c <= '1';
            c_next <= c_reg - 1;
```

```
            if (c_reg=0) then
               state_next <= start;
            end if;
         when start => -- assert start bit
            ps2d_out <= '0';
            tri_d <= '1';
            if fall_edge='1' then
               n_next <= "1000";
               state_next <= data;
            end if;
         when data => -- 8 data + 1 parity
            ps2d_out <= b_reg(0);
            tri_d <= '1';
            if fall_edge='1' then
               b_next <= '0' & b_reg(8 downto 1);
               if n_reg = 0 then
                   state_next <= stop;
               else
                   n_next <= n_reg - 1;
               end if;
            end if;
         when stop => -- assume floating high for ps2d
            if fall_edge='1' then
               state_next <= idle;
               tx_done_tick <='1';
            end if;
      end case;
   end process;
   -- tri-state buffers
   ps2c <= ps2c_out when tri_c ='1' else 'Z';
   ps2d <= ps2d_out when tri_d ='1' else 'Z';
end arch;
```

There is no error detection circuit in this code. A more robust design should check the correctness of the parity and acknowledgment bits and include a watchdog timer to prevent the mouse from being locked in an incorrect state.

9.4 BIDIRECTIONAL PS2 INTERFACE

9.4.1 Basic design and code

We can combine the receiving and transmitting subsystems to form a bidirectional PS2 interface. The top-level diagram is shown in Figure 9.4. We use the `tx_idle` and `rx_en` signals to coordinate the transmitting and receiving operations. Priority is given to the transmitting operation. When the transmitting subsystem is in operation, the `tx_idle` signal is deasserted, which, in turn, disables the receiving subsystem. The receiving subsystem can process input only when the transmitting subsystem is idle. The corresponding HDL code is shown in Listing 9.2.

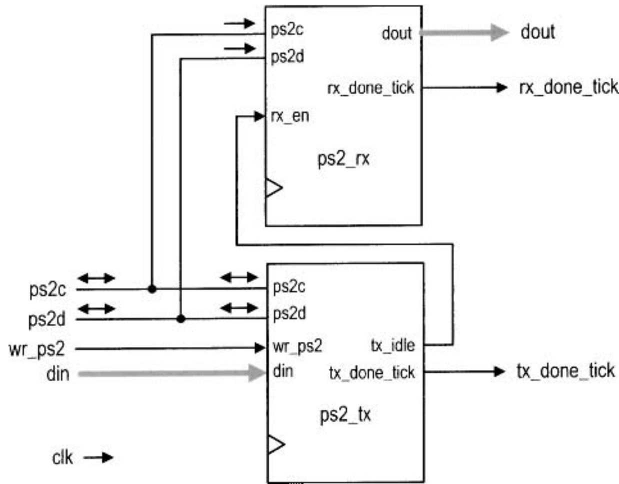

Figure 9.4 Top-level block diagram of a bidirectional PS2 interface.

Listing 9.2 Bidirectional PS2 interface

```
library ieee;
use ieee.std_logic_1164.all;
entity ps2_rxtx is
   port (
      clk, reset: in std_logic;
      wr_ps2: std_logic;
      din: in std_logic_vector(7 downto 0);
      dout: out std_logic_vector(7 downto 0);
      rx_done_tick: out  std_logic;
      tx_done_tick: out std_logic;
      ps2d, ps2c: inout std_logic
   );
end ps2_rxtx;

architecture arch of ps2_rxtx is
   signal tx_idle: std_logic;
begin
   ps2_tx_unit: entity work.ps2_tx(arch)
      port map(clk=>clk, reset=>reset, wr_ps2=>wr_ps2,
               din=>din, ps2d=>ps2d, ps2c=>ps2c,
               tx_idle=>tx_idle, tx_done_tick=>tx_done_tick);
   ps2_rx_unit: entity work.ps2_rx(arch)
      port map(clk=>clk, reset=>reset, rx_en=>tx_idle,
               ps2d=>ps2d, ps2c=>ps2c,
               rx_done_tick=>rx_done_tick, dout=>dout);
end arch;
```

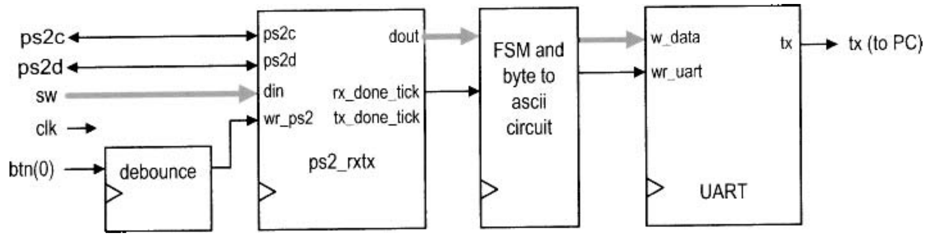

Figure 9.5 Block diagram of a mouse monitor circuit.

9.4.2 Verification circuit

We create a testing circuit to verify and monitor operation of the bidirectional interface. The block diagram is shown in Figure 9.5. A command is transmitted manually. We use the 8-bit switch to specify the data (i.e., the command from the host) and use a pushbutton to generate a one-clock-cycle tick to transmit the packet. The received packet data is first passed to the `byte-to-ascii` circuit, which converts the data into two ASCII characters plus a blank space. The characters are then transmitted via the UART and displayed in Windows HyperTerminal. The HDL code is shown in Listing 9.3.

Listing 9.3 Bidirectional PS2 interface monitor circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ps2_monitor is
   port (
      clk, reset: in  std_logic;
      sw: in std_logic_vector(7 downto 0);
      btn: in std_logic_vector(2 downto 0);
      ps2d, ps2c: inout  std_logic;
      tx: out  std_logic
   );
end ps2_monitor;

architecture arch of ps2_monitor is
   constant SP: std_logic_vector(7 downto 0):="00100000";
   -- blank space in ASCII
   type state_type is (idle, sendh, sendl, sendb);
   signal state_reg, state_next: state_type;
   signal rx_data, w_data: std_logic_vector(7 downto 0);
   signal psrx_done_tick: std_logic;
   signal wr_ps2, wr_uart: std_logic;
   signal ascii_code: std_logic_vector(7 downto 0);
   signal hex_in: std_logic_vector(3 downto 0);
begin
   --===============================================
   -- instantiation
   --===============================================
   btn_db_unit: entity work.debounce(fsmd_arch)
      port map(clk=>clk, reset=>reset, sw=>btn(0),
               db_level=>open, db_tick=>wr_ps2);
```

```
   ps2_rxtx_unit: entity work.ps2_rxtx(arch)
      port map(clk=>clk, reset=>reset, wr_ps2=>wr_ps2,
               din=>sw, dout=>rx_data, ps2d=>ps2d,
               ps2c=>ps2c, rx_done_tick=>psrx_done_tick,
35             tx_done_tick=>open);
   -- only use the UART transmitter
   uart_unit: entity work.uart(str_arch)
      generic map(FIFO_W=>4)
      port map(clk=>clk, reset=>reset, rd_uart=>'0',
40             wr_uart=>wr_uart, rx=>'1', w_data=>w_data,
               tx_full=>open, rx_empty=>open, r_data=>open,
               tx=>tx);
   --=============================================
   -- FSM to send 3 ASCII characters
45 --=============================================
   -- state registers
   process (clk, reset)
   begin
      if reset='1' then
50       state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
55 -- next-state logic
   process(state_reg,psrx_done_tick,ascii_code)
   begin
      wr_uart <= '0';
      w_data <= SP;
60    state_next <= state_reg;
      case state_reg is
         when idle =>
            if psrx_done_tick='1' then
               state_next <= sendh;
65          end if;
         when sendh => -- send higher hex char
            w_data <= ascii_code;
            wr_uart <= '1';
            state_next <= sendl;
70       when sendl => -- send lower hex char
            w_data <= ascii_code;
            wr_uart <= '1';
            state_next <= sendb;
         when sendb => -- send blank space char
75          w_data <= SP;
            wr_uart <= '1';
            state_next <= idle;
      end case;
   end process;
80 --=============================================
   -- scan code to ASCII display
   --=============================================
   -- split the scan code into two 4-bit hex
```

```
    hex_in <= rx_data(7 downto 4) when state_reg=sendh else
85             rx_data(3 downto 0);
    -- hex digit to ASCII code
    with hex_in select
       ascii_code <=
          "00110000" when "0000",   -- 0
90        "00110001" when "0001",   -- 1
          "00110010" when "0010",   -- 2
          "00110011" when "0011",   -- 3
          "00110100" when "0100",   -- 4
          "00110101" when "0101",   -- 5
95        "00110110" when "0110",   -- 6
          "00110111" when "0111",   -- 7
          "00111000" when "1000",   -- 8
          "00111001" when "1001",   -- 9
          "01000001" when "1010",   -- A
100       "01000010" when "1011",   -- B
          "01000011" when "1100",   -- C
          "01000100" when "1101",   -- D
          "01000101" when "1110",   -- E
          "01000110" when others;   -- F
105 end arch;
```

If a mouse is connected to the PS2 circuit, we can first issue the FF command to reset the mouse and then issue the F4 command to enable the stream mode. Windows HyperTerminal will show the mouse's acknowledge packets and subsequent mouse movement packets.

9.5 PS2 MOUSE INTERFACE

9.5.1 Basic design

The basic PS2 mouse interface creates another layer over the bidirectional PS2 circuit. Its two basic functions are to enable the stream mode and to reassemble the 3 data bytes. The output of the circuit are `xm` and `ym`, which are two 9-bit x- and y-axis movement signals; `btm`, which is the 3-bit button status signal; and `m_done_tick`, which is a one-clock-cycle status signal and is asserted when the assembled data is available.

The HDL code is shown in Listing 9.4. It is implemented by an FSMD with seven states. The `init1`, `init2`, and `init3` states are executed once after the `reset` signal is asserted. In these states, the FSMD issues the F4 command, waits for completion of the transmission, and then waits for the acknowledgment packet. The mouse is in the stream mode now. The FSMD then obtains and assembles the next three packets in the `pack1`, `pack2`, and `pack3` states, and activates the `m_done_tick` signal in the `done` state. The FSMD circulates these four states afterward.

Listing 9.4 Basic mouse interface circuit

```
  library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  entity mouse is
5    port (
        clk, reset: in  std_logic;
```

```
      ps2d, ps2c: inout std_logic;
      xm, ym: out std_logic_vector(8 downto 0);
      btnm: out std_logic_vector(2 downto 0);
10    m_done_tick: out std_logic
   );
end mouse;

architecture arch of mouse is
15 constant STRM: std_logic_vector(7 downto 0):="11110100";
   -- stream command F4
   type state_type is (init1, init2, init3,
                        pack1, pack2, pack3, done);
   signal state_reg, state_next: state_type;
20 signal rx_data: std_logic_vector(7 downto 0);
   signal rx_done_tick, tx_done_tick: std_logic;
   signal wr_ps2: std_logic;
   signal x_reg, y_reg: std_logic_vector(8 downto 0);
   signal x_next, y_next: std_logic_vector(8 downto 0);
25 signal btn_reg, btn_next: std_logic_vector(2 downto 0);
begin
   -- instantiation
   ps2_rxtx_unit: entity work.ps2_rxtx(arch)
      port map(clk=>clk, reset=>reset, wr_ps2=>wr_ps2,
30             din=>STRM, dout=>rx_data,
               ps2d=>ps2d, ps2c=>ps2c,
               rx_done_tick=>rx_done_tick,
               tx_done_tick=>tx_done_tick);
   -- state and data registers
35 process (clk, reset)
   begin
      if reset='1' then
         state_reg <= init1;
         x_reg <= (others=>'0');
40       y_reg <= (others=>'0');
         btn_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         x_reg <= x_next;
45       y_reg <= y_next;
         btn_reg <= btn_next;
      end if;
   end process;
   -- next-state logic
50 process(state_reg,rx_done_tick,tx_done_tick,
           x_reg,y_reg,btn_reg,rx_data)
   begin
      wr_ps2 <= '0';
      m_done_tick <= '0';
55    x_next <= x_reg;
      y_next <= y_reg;
      btn_next <= btn_reg;
      state_next <= state_reg;
      case state_reg is
```

```
60        when init1=>
             wr_ps2 <= '1';
             state_next <= init2;
          when init2=> -- wait for send to complete
             if tx_done_tick='1' then
65              state_next <= init3;
             end if;
          when init3=> -- wait for acknowledge packet
             if rx_done_tick='1' then
                state_next <= pack1;
70           end if;
          when pack1=> -- wait for 1st data packet
             if rx_done_tick='1' then
                state_next <= pack2;
                y_next(8) <= rx_data(5);
75              x_next(8) <= rx_data(4);
                btn_next <=  rx_data(2 downto 0);
             end if;
          when pack2=> -- wait for 2nd data packet
             if rx_done_tick='1' then
80              state_next <= pack3;
                x_next(7 downto 0) <= rx_data;
             end if;
          when pack3=> -- wait for 3rd data packet
             if rx_done_tick='1' then
85              state_next <= done;
                y_next(7 downto 0) <= rx_data;
             end if;
          when done =>
             m_done_tick <= '1';
90           state_next <= pack1;
       end case;
    end process;
    xm <= x_reg;
    ym <= y_reg;
95  btnm <= btn_reg;
 end arch;
```

This design provides only minimal functionalities. A more sophisticated circuit should have a robust method to initiate the stream mode and add additional buffer, similar to that in Section 7.2.4, to interact better with the external system.

9.5.2 Testing circuit

We use a simple testing circuit to demonstrate the use of the PS2 interface. The circuit uses a mouse to control the eight discrete LEDs of the prototyping board. Only one of the eight LEDs is lit and the position of that LED follows the x-axis movement of the mouse. The pressing of the left or right button places the lit LED to the leftmost or rightmost position.

The HDL code is shown in Listing 9.5. It uses a 10-bit counter to keep track of the current x-axis position. The counter is updated when a new data item is available (i.e., when the m_done_tick signal is asserted). The counter is set to 0 or maximum when the left or right mouse button is pressed. Otherwise, it adds the amount of the signed-extended

x-axis movement. A decoding circuit uses the three MSBs of the counter to activate one of the LEDs.

Listing 9.5 Mouse-controlled LED circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mouse_led is
   port (
      clk, reset: in  std_logic;
      ps2d, ps2c: inout std_logic;
      led: out std_logic_vector(7 downto 0)
   );
end mouse_led;

architecture arch of mouse_led is
   signal p_reg, p_next: unsigned(9 downto 0);
   signal xm: std_logic_vector(8 downto 0);
   signal btnm: std_logic_vector(2 downto 0);
   signal m_done_tick: std_logic;

begin
   -- instantiation
   mouse_unit: entity work.mouse(arch)
      port map(clk=>clk, reset=>reset,
               ps2d=>ps2d, ps2c=>ps2c,
               xm=>xm, ym=>open, btnm=>btnm,
               m_done_tick=>m_done_tick);
   -- register
   process (clk, reset)
   begin
      if reset='1' then
         p_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         p_reg <= p_next;
      end if;
   end process;
   -- counter
   p_next <= p_reg when m_done_tick='0' else
             "0000000000" when btnm(0)='1' else --left button
             "1111111111" when btnm(1)='1' else --right button
             p_reg + unsigned(xm(8) & xm);

   with p_reg(9 downto 7) select
      led <= "10000000" when "000",
             "01000000" when "001",
             "00100000" when "010",
             "00010000" when "011",
             "00001000" when "100",
             "00000100" when "101",
             "00000010" when "110",
             "00000001" when others;
end arch;
```

9.6 BIBLIOGRAPHIC NOTES

The bibliographic information for this Chapter is similar to that for Chapter 8.

9.7 SUGGESTED EXPERIMENTS

The mouse is used mainly with a graphic video interface, which is discussed in Chapters 12 and 13. Many additional mouse-related experiments can be found in these chapters.

9.7.1 Keyboard control circuit

A host can issue a command to set certain parameters for a PS2 keyboard as well. For example, we can control the three LEDs of the keyboard by sending `ED 0X`. The `X` is a hexadecimal number with a format of "*0snc*", where s, n, and c are 1-bit values that control the Scroll, Num, and `Caps Lock` LEDs, respectively. We can incorporate this feature into the keyboard interface circuit of Section 8.4.1 and use a 3-bit switch to control the three keyboard LEDs. Design the expanded interface circuit, resynthesize the circuit, and verify its operation.

9.7.2 Enhanced mouse interface

For the mouse interface discussed in Section 9.5, we can alter the design to manually enable or disable the steam mode. This can be done by using two pushbuttons of the FPGA prototyping board. One button issues the reset command, `FF`, which disables the stream mode during operation, and the other button issues the `F4` command to enable the steam mode. Modify the original interface to incorporate this feature, and resynthesize the LED testing circuit to verify its operation.

9.7.3 Mouse-controlled seven-segment LED display

We can use the mouse to enter four decimal digits on the four-digit seven-segment LED display. The circuit functions as follows:

- Only one of the four decimal points of the LED display is lit. The lit decimal point indicates the location of the selected digit.
- The location of the selected digit follows the x-axis movement of the mouse.
- The content of the select seven-segment LED display is a decimal digit (i.e., 0, . . ., 9) and changes with the y-axis movement of the mouse.

Design and synthesize this circuit and verify its operation.