CHAPTER 8

PS2 KEYBOARD

8.1 INTRODUCTION

PS2 port was introduced in IBM's Personal System/2 personnel computers. It is a widely supported interface for a keyboard and mouse to communicate with the host. The PS2 port contains two wires for communication purposes. One wire is for data, which is transmitted in a serial stream. The other wire is for the clock information, which specifies when the data is valid and can be retrieved. The information is transmitted as an 11-bit "packet" that contains a start bit, 8 data bits, an odd parity bit, and a stop bit. Whereas the basic format of the packet is identical for a keyboard and a mouse, the interpretation for the data bits is different. The FPGA prototyping board has a PS2 port and acts as a host. We discuss the keyboard interface in this chapter and cover the mouse interface in Chapter 9.

The 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, the bidirectional communication is hardly required for the PS2 keyboard, and thus our discussion is limited to one direction, from the keyboard to the prototyping board. Bidirectional design will be examined in the mouse interface in Chapter 9.

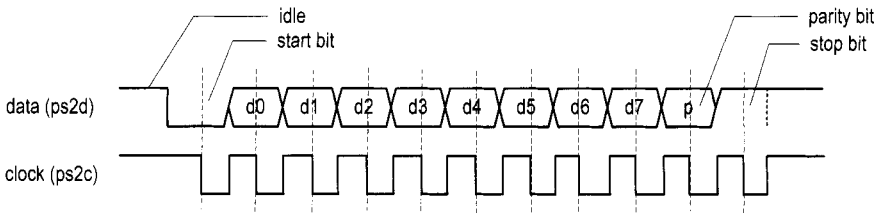

Figure 8.1 Timing diagram of a PS2 port.

8.2 PS2 RECEIVING SUBSYSTEM

8.2.1 Physical interface of a PS2 port

In addition to data and clock lines, the PS2 port includes connections for power (i.e., V_{cc}) and ground. The power is supplied by the host. In the original PS2 port, V_{cc} is 5 V and the outputs of the data and clock lines are open-collector. However, most current keyboards and mice can work well with 3.3 V. For an older keyboard and mouse, the 5-V supply can be obtained by switching the J2 jumper on the S3 board. The FPGA should still function properly since its I/O pins can tolerate 5-V input.

8.2.2 Device-to-host communication protocol

A PS2 device and its host communicate via packets. The basic timing diagram of transmitting a packet from a PS2 device to a host is shown in Figure 8.1, in which the data and clock signals are labeled `ps2d` and `ps2c`, respectively.

The data is transmitted in a serial stream, and its format is similar to that of a UART. Transmission begins with a start bit, followed by 8 data bits and an odd parity bit, and ends with a stop bit. Unlike a UART, the clock information is carried in a separate clock signal, `ps2c`. The falling edge of the `ps2c` signal indicates that the corresponding bit in the `ps2d` line is valid and can be retrieved. The clock period of the `ps2c` signal is between 60 and 100 μs (i.e., 10 kHz to 16.7 kHz), and the `ps2d` signal is stable at least 5 μs before and after the falling edge of the `ps2c` signal.

8.2.3 Design and code

The design of the PS2 port receiving subsystem is somewhat similar to that of a UART receiver. Instead of using the oversampling scheme, the falling-edge of the `ps2c` signal is used as the reference point to retrieve data. The subsystem includes a falling edge detection circuit, which generates a one-clock-cycle tick at the falling edge of the `ps2c` signal, and the receiver, which shifts in and assembles the serial bits.

The edge detection circuit discussed in Section 5.3.1 can be used to detect the falling edge and generate an enable tick. However, because of the potential noise and slow transition, a simple filtering circuit is added to eliminate glitches. Its code is

```
-- register
process (clk, reset)
   . . .
   filter_reg <= filter_next;
```

```
    . . .
end process;

-- 1-bit shifter
filter_next <= ps2c & filter_reg(7 downto 1);
-- "filter"
f_ps2c_next <= '1' when filter_reg="11111111" else
               '0' when filter_reg="00000000" else
               f_ps2c_reg;
```

The circuit is composed of an 8-bit shift register and returns a '1' or '0' when eight consecutive 1's or 0's are received. Any glitches shorter than eight clock cycles will be ignored (i.e., filtered out). The filtered output signal is then fed to the regular falling-edge detection circuit.

The ASMD chart of the receiver is shown in Figure 8.2. The receiver is initially in the `idle` state. It includes an additional control signal, `rx_en`, which is used to enable or disable the receiving operation. The purpose of the signal is to coordinate the bidirectional operation. It can be set to '1' for the keyboard interface.

After the first falling-edge tick and the `rx_en` signal are asserted, the FSMD shifts in the start bit and moves to the `dps` state. Since the received data is in fixed format, we shift in the remaining 10 bits in a single state rather than using separate `data`, `parity`, and `stop` states. The FSMD then moves to the `load` state, in which one extra clock cycle is provided to complete the shifting of the stop bit, and the `psrx_done_tick` signal is asserted for one clock cycle. The HDL code consists of the filtering circuit and an FSMD, which follows the ASMD chart. It is shown in Listing 8.1.

Listing 8.1 PS2 port receiver

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ps2_rx is
   port (
      clk, reset: in  std_logic;
      ps2d, ps2c: in  std_logic;  -- key data, key clock
      rx_en: in std_logic;
      rx_done_tick: out  std_logic;
      dout: out std_logic_vector(7 downto 0)
   );
end ps2_rx;

architecture arch of ps2_rx is
   type statetype is (idle, dps, load);
   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;
   signal b_reg, b_next: std_logic_vector(10 downto 0);
   signal n_reg,n_next: unsigned(3 downto 0);
   signal fall_edge: std_logic;
begin
   --=================================================
   -- filter and falling edge tick generation for ps2c
```

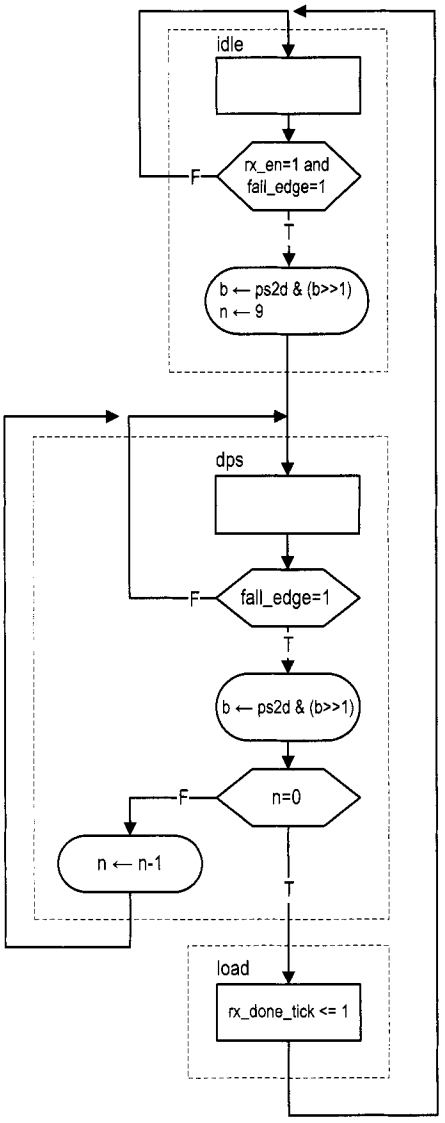

Figure 8.2 ASMD chart of the PS2 port receiver.

```
   --=================================================
   process (clk, reset)
   begin
      if reset='1' then
         filter_reg <= (others=>'0');
         f_ps2c_reg <= '0';
      elsif (clk'event and clk='1') then
         filter_reg <= filter_next;
         f_ps2c_reg <= f_ps2c_next;
      end if;
   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
                  f_ps2c_reg;
   fall_edge <= f_ps2c_reg and (not f_ps2c_next);

   --=================================================
   -- fsmd to extract the 8-bit data
   --=================================================
   -- registers
   process (clk, reset)
   begin
      if reset='1' then
         state_reg <= idle;
         n_reg  <= (others=>'0');
         b_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         n_reg <= n_next;
         b_reg <= b_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,n_reg,b_reg,fall_edge,rx_en,ps2d)
   begin
      rx_done_tick <='0';
      state_next <= state_reg;
      n_next <= n_reg;
      b_next <= b_reg;
      case state_reg is
         when idle =>
            if fall_edge='1' and rx_en='1' then
               -- shift in start bit
               b_next <= ps2d & b_reg(10 downto 1);
               n_next <= "1001";
               state_next <= dps;
            end if;
         when dps =>  -- 8 data + 1 parity + 1 stop
            if fall_edge='1' then
            b_next <= ps2d & b_reg(10 downto 1);
               if n_reg = 0 then
```

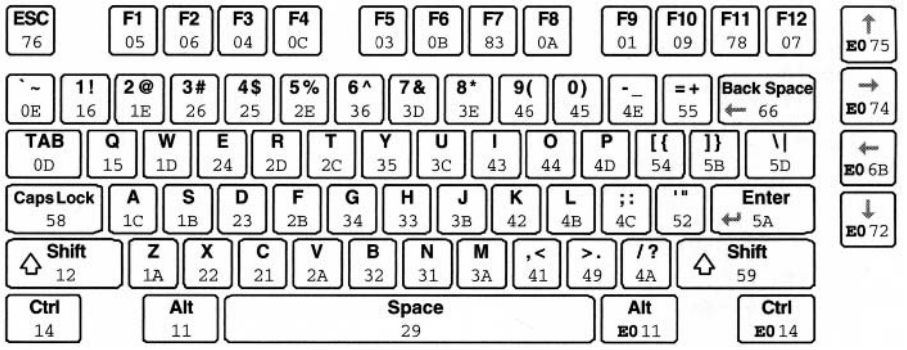

Figure 8.3 Scan code of the PS2 keyboard. (Courtesy of Xilinx, Inc. © Xilinx, Inc. 1994–2007. All rights reserved.)

```
                    state_next <=load;
80                else
                    n_next <= n_reg - 1;
                  end if;
              end if;
            when load =>
85              -- 1 extra clock to complete the last shift
                state_next <= idle;
                rx_done_tick <='1';
        end case;
    end process;
90  -- output
    dout <= b_reg(8 downto 1); -- data bits
end arch;
```

There is no error detection circuit in the description. A more robust design should check the correctness of the start, parity, and stop bits and include a watchdog timer to prevent the keyboard from being locked in an incorrect state. This is left as an experiment at the end of the chapter.

8.3 PS2 KEYBOARD SCAN CODE

8.3.1 Overview of the scan code

A keyboard consists of a matrix of keys and an embedded microcontroller that monitors (i.e., scans) the activities of the keys and sends *scan code* accordingly. Three types of key activities are observed:

- When a key is pressed, the *make code* of the key is transmitted.
- When a key is held down continuously, a condition known as *typematic*, the make code is transmitted repeatedly at a specific rate. By default, a PS2 keyboard transmits the make code about every 100 ms after a key has been held down for 0.5 second.
- When a key is released, the *break code* of the key is transmitted.

The make code of the main part of a PS2 keyboard is shown in Figure 8.3. It is normally 1 byte wide and represented by two hexadecimal numbers. For example, the make code

of the A key is 1C. This code can be conveyed by one packet when transmitted. The make codes of a handful of special-purpose keys, which are known as the *extended keys*, can have 2 to 4 bytes. A few of these keys are shown in Figure 8.3. For example, the make code of the upper arrow on the right is E0 75. Multiple packets are needed for the transmission. The break codes of the regular keys consist of F0 followed by the make code of the key. For example, the break code of the A key is F0 1C.

The PS2 keyboard transmits a sequence of codes according to the key activities. For example, when we press and release the A key, the keyboard first transmits its make code and then the break code:

```
1C F0 1C
```

If we hold the key down for awhile before releasing it, the make code will be transmitted multiple times:

```
1C 1C 1C ... 1C F0 1C
```

Multiple keys can be pressed at the same time. For example, we can first press the shift key (whose make code is 12) and then the A key, and release the A key and then release the shift key. The transmitted code sequence follows the make and break codes of the two keys:

```
12 1C F0 1C F0 12
```

The previous sequence is how we normally obtain an uppercase A. Note that there is no special code to distinguish the lower- and uppercase keys. It is the responsibility of the host device to keep track of whether the shift key is pressed and to determine the case accordingly.

8.3.2 Scan code monitor circuit

The scan code monitor circuit monitors the arrival of the received packets and displays the scan codes on a PC's HyperTerminal window. The basic design approach is to first split the received scan code into two 4-bit parts and treat them as two hexadecimal digits, and then convert the two digits to ASCII code words and send the words to a PC via the UART. The received scan codes should be displayed similar to the previous example sequences. The program is shown in Listing 8.2.

Listing 8.2 PS2 keyboard scan code monitor circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity kb_monitor is
    port (
        clk, reset: in  std_logic;
        ps2d, ps2c: in  std_logic;
        tx: out  std_logic
    );
end kb_monitor;

architecture arch of kb_monitor is
    constant SP: std_logic_vector(7 downto 0):="00100000";
    -- blank space in ASCII
```

```
   type statetype is (idle, send1, send0, sendb);
   signal state_reg, state_next: statetype;
   signal scan_data, w_data: std_logic_vector(7 downto 0);
   signal scan_done_tick, wr_uart: std_logic;
   signal ascii_code: std_logic_vector(7 downto 0);
   signal hex_in: std_logic_vector(3 downto 0);
begin
   --=====================================================
   -- instantiation
   --=====================================================
   -- instantiate PS2 receiver
   ps2_rx_unit: entity work.ps2_rx(arch)
      port map(clk=>clk, reset=>reset, rx_en=>'1',
               ps2d=>ps2d, ps2c=>ps2c,
               rx_done_tick=>scan_done_tick,
               dout=>scan_data);

   -- instantiate UART
   uart_unit: entity work.uart(str_arch)
      port map(clk=>clk, reset=>reset, rd_uart=>'0',
               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
   --=====================================================
   -- state registers
   process (clk, reset)
   begin
      if reset='1' then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg, scan_done_tick, ascii_code)
   begin
      wr_uart <= '0';
      w_data <= SP;
      state_next <= state_reg;
      case state_reg is
         when idle => -- start when a scan code received
            if scan_done_tick='1' then
               state_next <= send1;
            end if;
         when send1 => -- send higher hex char
            w_data <= ascii_code;
            wr_uart <= '1';
            state_next <= send0;
         when send0 => -- send lower hex char
            w_data <= ascii_code;
```

```
            wr_uart <= '1';
            state_next <= sendb;
         when sendb => -- send blank space char
            w_data <= SP;
            wr_uart <= '1';
            state_next <= idle;
      end case;
   end process;

   --=====================================================
   -- scan code to ASCII display
   --=====================================================
   -- split the scan code into two 4-bit hex
   hex_in <= scan_data(7 downto 4) when state_reg=send1 else
             scan_data(3 downto 0);
   -- hex digit to ASCII code
   with hex_in select
      ascii_code <=
         "00110000" when "0000",  -- 0
         "00110001" when "0001",  -- 1
         "00110010" when "0010",  -- 2
         "00110011" when "0011",  -- 3
         "00110100" when "0100",  -- 4
         "00110101" when "0101",  -- 5
         "00110110" when "0110",  -- 6
         "00110111" when "0111",  -- 7
         "00111000" when "1000",  -- 8
         "00111001" when "1001",  -- 9
         "01000001" when "1010",  -- A
         "01000010" when "1011",  -- B
         "01000011" when "1100",  -- C
         "01000100" when "1101",  -- D
         "01000101" when "1110",  -- E
         "01000110" when others;  -- F
end arch;
```

An FSM is used to control the overall operation. The UART operation is initiated when a new scan code is received (as indicated by the assertion of `scan_done_tick`). The FSM circulates through the `send1`, `send0`, and `sendb` states, in which the ASCII codes of the upper hexadecimal digit, lower hexadecimal digit, and blank space are written to the UART. Recall that the UART has a FIFO of four words, and thus no overflow will occur. Note that the UART receiver is not used and the corresponding ports are mapped to constants or **open**.

8.4 PS2 KEYBOARD INTERFACE CIRCUIT

As discussed in Section 8.3.1, a sequence of packets is transmitted even for simple keyboard activities. It will be quite involved if we want to cover all possible combinations. In this section, we assume that only one regular key is pressed and released at a time and design a circuit that returns the make code of this key. This design provides a simple way to send a character or digit to the prototyping board and should be satisfactory for our purposes.

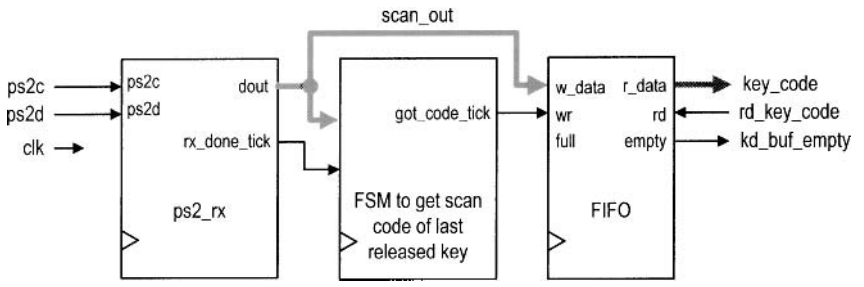

Figure 8.4 Block diagram of a last-released key circuit.

8.4.1 Basic design and HDL code

The keyboard circuit, as a UART, is a peripheral circuit of a large system and needs a mechanism to communicate with the main system. The flagging and buffering schemes discussed in Section 7.2.4 can be applied for the keyboard circuit as well. We use a four-word FIFO buffer as the interface in this design.

The top-level conceptual diagram is shown in Figure 8.4. It consists of the PS2 receiver, a FIFO buffer, and a control FSM. The basic idea is to use the FSM to keep track of the F0 packet of the break code. After it is received, the next packet should be the make code of this key and is written into the FIFO buffer. Note that this scheme cannot be applied to the extended keys since their make codes involve multiple packets. The corresponding HDL code is shown in Listing 8.3.

Listing 8.3 PS2 keyboard last-released key circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity kb_code is
   generic(W_SIZE: integer:=2);  -- 2^W_SIZE words in FIFO
   port (
      clk, reset: in  std_logic;
      ps2d, ps2c: in  std_logic;
      rd_key_code: in std_logic;
      key_code: out std_logic_vector(7 downto 0);
      kb_buf_empty: out std_logic
   );
end kb_code;

architecture arch of kb_code is
   constant BRK: std_logic_vector(7 downto 0):="11110000";
   -- F0 (break code)
   type statetype is (wait_brk, get_code);
   signal state_reg, state_next: statetype;
   signal scan_out, w_data: std_logic_vector(7 downto 0);
   signal scan_done_tick, got_code_tick: std_logic;

begin
   --=============================================================
   -- instantiation
```

```
   --=========================================================
   ps2_rx_unit: entity work.ps2_rx(arch)
      port map(clk=>clk, reset=>reset, rx_en=>'1',
               ps2d=>ps2d, ps2c=>ps2c,
               rx_done_tick=>scan_done_tick,
               dout=>scan_out);

   fifo_key_unit: entity work.fifo(arch)
      generic map(B=>8, W=>W_SIZE)
      port map(clk=>clk, reset=>reset, rd=>rd_key_code,
               wr=>got_code_tick, w_data=>scan_out,
               empty=>kb_buf_empty, full=>open,
               r_data=>key_code);

   --=========================================================
   -- FSM to get the scan code after F0 received
   --=========================================================
   process (clk, reset)
   begin
      if reset='1' then
         state_reg <= wait_brk;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;

   process(state_reg, scan_done_tick, scan_out)
   begin
      got_code_tick <='0';
      state_next <= state_reg;
      case state_reg is
         when wait_brk => -- wait for F0 of break code
            if scan_done_tick='1' and scan_out=BRK then
               state_next <= get_code;
            end if;
         when get_code => -- get the following scan code
            if scan_done_tick='1' then
               got_code_tick <='1';
               state_next <= wait_brk;
            end if;
      end case;
   end process;
end arch;
```

The main part of the code is the FSM, which screens for the break code and coordinates the operation of two other modules. It checks the received packets in the `wait_brk` state continuously. When the F0 packet is detected, it moves to the `get_code` state and waits for the next packet, which is the make code of the key. The FSM then asserts the `code_done_tick` signal for one clock cycle and returns to the `wait_brk` state.

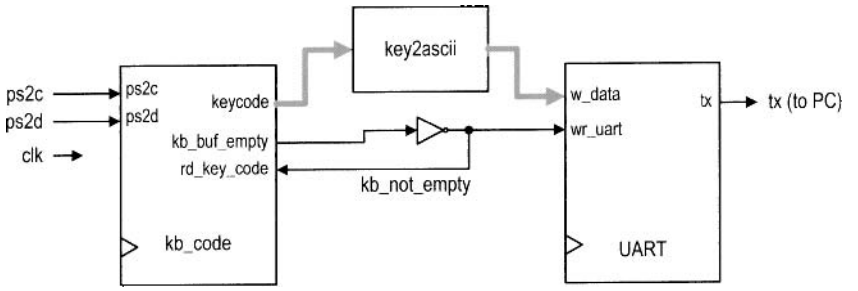

Figure 8.5 Block diagram of a keyboard verification circuit.

8.4.2 Verification circuit

We design a simple serial interface and decoding circuit to verify operation of the PS2 keyboard interface. The top-level block diagram is shown in Figure 8.5. The circuit converts a key's make code to the corresponding ASCII code and then sends the ASCII code to the UART. The corresponding character or digits can be displayed in the HyperTerminal window. The HDL code for the conversion circuit is shown in Listing 8.4.

Listing 8.4 Keyboard make code to ASCII code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity key2ascii is
   port (
      key_code: in std_logic_vector(7 downto 0);
      ascii_code: out std_logic_vector(7 downto 0)
   );
end key2ascii;

architecture arch of key2ascii is
begin
   with key_code select
      ascii_code <=
         "00110000" when "01000101",  -- 0
         "00110001" when "00010110",  -- 1
         "00110010" when "00011110",  -- 2
         "00110011" when "00100110",  -- 3
         "00110100" when "00100101",  -- 4
         "00110101" when "00101110",  -- 5
         "00110110" when "00110110",  -- 6
         "00110111" when "00111101",  -- 7
         "00111000" when "00111110",  -- 8
         "00111001" when "01000110",  -- 9

         "01000001" when "00011100",  -- A
         "01000010" when "00110010",  -- B
         "01000011" when "00100001",  -- C
         "01000100" when "00100011",  -- D
         "01000101" when "00100100",  -- E
```

```
            "01000110" when "00101011",  -- F
            "01000111" when "00110100",  -- G
            "01001000" when "00110011",  -- H
            "01001001" when "01000011",  -- I
            "01001010" when "00111011",  -- J
            "01001011" when "01000010",  -- K
            "01001100" when "01001011",  -- L
            "01001101" when "00111010",  -- M
            "01001110" when "00110001",  -- N
            "01001111" when "01000100",  -- O
            "01010000" when "01001101",  -- P
            "01010001" when "00010101",  -- Q
            "01010010" when "00101101",  -- R
            "01010011" when "00011011",  -- S
            "01010100" when "00101100",  -- T
            "01010101" when "00111100",  -- U
            "01010110" when "00101010",  -- V
            "01010111" when "00011101",  -- W
            "01011000" when "00100010",  -- X
            "01011001" when "00110101",  -- Y
            "01011010" when "00011010",  -- Z

            "01100000" when "00001110",  -- `
            "00101101" when "01001110",  -- -
            "00111101" when "01010101",  -- =
            "01011011" when "01010100",  -- [
            "01011101" when "01011011",  -- ]
            "01011100" when "01011101",  -- \
            "00111011" when "01001100",  -- ;
            "00100111" when "01010010",  -- '
            "00101100" when "01000001",  -- ,
            "00101110" when "01001001",  -- .
            "00101111" when "01001010",  -- /

            "00100000" when "00101001",  -- (space)
            "00001101" when "01011010",  -- (enter, cr)
            "00001000" when "01100110",  -- (backspace)
            "00101010" when others;      -- *
end arch;
```

The complete code for the verification circuit follows the block diagram and is shown in Listing 8.5.

Listing 8.5 Keyboard verification circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity kb_test is
    port (
        clk, reset: in  std_logic;
        ps2d, ps2c: in  std_logic;
        tx: out  std_logic
    );
```

```
end kb_test;

architecture arch of kb_test is
   signal scan_data, w_data: std_logic_vector(7 downto 0);
   signal kb_not_empty, kb_buf_empty: std_logic;
   signal key_code, ascii_code: std_logic_vector(7 downto 0);
begin
   kb_code_unit: entity work.kb_code(arch)
      port map(clk=>clk, reset=>reset, ps2d=>ps2d, ps2c=>ps2c,
               rd_key_code=>kb_not_empty, key_code=>key_code,
               kb_buf_empty=>kb_buf_empty);
   uart_unit: entity work.uart(str_arch)
      port map(clk=>clk, reset=>reset, rd_uart=>'0',
               wr_uart=>kb_not_empty, rx=>'1',
               w_data=>ascii_code, tx_full=>open,
               rx_empty=>open, r_data=>open, tx=>tx);
   key2a_unit: entity work.key2ascii(arch)
      port map(key_code=>key_code, ascii_code=>ascii_code);

   kb_not_empty <= not kb_buf_empty;
end arch;
```

8.5 BIBLIOGRAPHIC NOTES

Three articles, "PS/2 Mouse/Keyboard Protocol," "PS/2 Keyboard Interface," and "PS/2 Mouse Interface," by Adam Chapweske, provide detailed information on the PS2 keyboard and mouse interface. They can be found at the http://www.computer-engineering.org site. *Rapid Prototyping of Digital Systems: Quartus® II Edition* by James O. Hamblen et al. also contains a chapter on the PS2 port and the keyboard and mouse protocols.

8.6 SUGGESTED EXPERIMENTS

8.6.1 Alternative keyboard interface I

The interface circuit in Section 8.4 returns the make code of the last released key and thus ignores the typematic condition. An alternative approach is to consider the typematic condition. The keyboard interface circuit should return a key's make code repeatedly when it is held down and ignore the final break code. For simplicity, we assume that the extended keys are not used. Design the new interface circuit, resynthesize the verification circuit, and verify operation of the new interface circuit.

8.6.2 Alternative keyboard interface II

We can expand the interface circuit to distinguish whether the shift key is pressed so that both lower- and uppercase characters can be entered. The expanded circuit can be modified as follows:

- The keycode output should be extended from 8 bits to 9 bits. The extra bit indicates whether the shift key is held down.

- The FSM should add a special branch to process the make and break codes of the shift key and set the value of the corresponding bit accordingly.
- The width of the FIFO buffer should be extended to 9 bits.

Design the expanded interface circuit, modify the `key2ascii` circuit to handle both lower- and uppercase characters, resynthesize the verification circuit, and verify operation of the expanded interface circuit.

8.6.3 PS2 receiving subsystem with watchdog timer

There is no error-handling capability in the PS2 receiving subsystem in Section 8.2. The potential noise and glitches in the `ps2c` signal may cause the FSMD to be stuck in an incorrect state. One way to deal with this problem is to add a watchdog timer. The timer is initiated every time the `fall_edge_tick` signal is asserted in the `get_bit` state. The `time_out` signal is asserted if no subsequently falling edge arrives in the next 20 μs, and the FSMD returns to the `idle` state. Design the modified receiving subsystem, derive a testbench, and use simulation to verify its operation.

8.6.4 Keyboard-controlled stopwatch

Consider the enhanced stopwatch in Experiment 4.7.6. Operation of the stopwatch is controlled by three switches on the prototyping board. We can use the keyboard to send commands to the stopwatch:

- When the `C` (for "clear") key is pressed, the stopwatch aborts the current counting, is cleared to zero, and sets the counting direction to "up."
- When the `G` (for "go") key is pressed, the stopwatch starts to count.
- When the `P` (for "pause") key is pressed, the counting pauses.
- When the `U` (for "up-down") key is pressed, the stopwatch reverses the direction of counting.
- All other keys will be ignored.

Design the new stopwatch, synthesize the circuit, and verify its operation.

8.6.5 Keyboard-controlled rotating LED banner

Consider the rotating LED banner circuit in Experiment 4.7.5. We can use a keyboard to control its operation and dynamically modify the digits in the banner:

- When the `G` (for "go") key is pressed, the LED banner rotates.
- When the `P` (for "pause") key is pressed, the LED banner pauses.
- When the `D` (for "direction") key is pressed, the LED banner reverses the direction of rotation.
- When a decimal digit (i.e., 0, 1, ..., 9) key is pressed, the banner will be modified. The banner can be treated as a 10-word FIFO buffer. The new digit will be inserted at the beginning (i.e., the leftmost position) of the banner, and the rightmost digit will be shifted out and discarded.
- All other keys will be ignored.

Design the new rotating LED banner, synthesize the circuit, and verify its operation.