CHAPTER 8

UART

8.1 INTRODUCTION

A *universal asynchronous receiver and transmitter* (UART) is a circuit that sends parallel data through a serial line. UARTs are frequently used in conjunction with the EIA (Electronic Industries Alliance) RS-232 standard, which specifies the electrical, mechanical, functional, and procedural characteristics of two data communication equipment. Because the voltage level defined in RS-232 is different from that of FPGA I/O, a voltage converter chip is needed between a serial port and an FPGA's I/O pins.

The S3 board has an RS-232 port with a standard nine-pin connector. The board contains the necessary voltage converter chip and configures the various RS-232's control signals to automatically generate acknowledgment for the PC's serial port. A standard straight-through serial cable can be used to connect the S3 board and PC's serial port. The S3 board basically handles the RS-232 standard and we only need to concentrate on design of the UART circuit.

A UART includes a transmitter and a receiver. The transmitter is essentially a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate. The receiver, on the other hand, shifts in data bit by bit and then reassembles the data. The serial line is 1 when it is idle. The transmission starts with a *start bit*, which is 0, followed by *data bits* and an optional *parity bit*, and ends with *stop bits*, which are 1. The number of data bits can be 6, 7, or 8. The optional parity bit is used for error detection. For odd parity, it is set to 0 when the data bits have an odd number of 1's. For even parity, it is set to 0 when the data bits have an even number of 1's. The number of stop bits can be 1, 1.5,

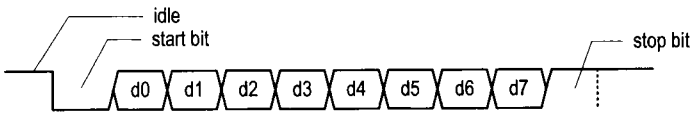


Figure 8.1 Transmission of a byte.

or 2. Transmission with 8 data bits, no parity, and 1 stop bit is shown in Figure 8.1. Note that the LSB of the data word is transmitted first.

No clock information is conveyed through the serial line. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud rate (i.e., number of bits per second), the number of data bits and stop bits, and use of the parity bit. The commonly used baud rates are 2400, 4800, 9600, and 19,200 bauds.

We illustrate the design of the receiving and transmitting subsystems in the following sections. The design is customized for a UART with a 19,200 baud rate, 8 data bits, 1 stop bit, and no parity bit.

8.2 UART RECEIVING SUBSYSTEM

Since no clock information is conveyed from the transmitted signal, the receiver can retrieve the data bits only by using the predetermined parameters. We use an *oversampling scheme* to estimate the middle points of transmitted bits and then retrieve them at these points accordingly.

8.2.1 Oversampling procedure

The most commonly used sampling rate is 16 times the baud rate, which means that each serial bit is sampled 16 times. Assume that the communication uses N data bits and M stop bits. The oversampling scheme works as follows:

1. Wait until the incoming signal becomes 0, the beginning of the start bit, and then start the sampling tick counter.
2. When the counter reaches 7, the incoming signal reaches the middle point of the start bit. Clear the counter to 0 and restart.
3. When the counter reaches 15, the incoming signal progresses for one bit and reaches the middle of the first data bit. Retrieve its value, shift it into a register, and restart the counter.
4. Repeat step 3 N–1 more times to retrieve the remaining data bits.
5. If the optional parity bit is used, repeat step 3 one time to obtain the parity bit.
6. Repeat step 3 M more times to obtain the stop bits.

The oversampling scheme basically performs the function of a clock signal. Instead of using the rising edge to indicate when the input signal is valid, it utilizes sampling ticks to estimate the middle point of each bit. While the receiver has no information about the exact onset time of the start bit, the estimation can be off by at most $\frac{1}{16}$. The subsequent data bit retrievals are off by at most $\frac{1}{16}$ from the middle point as well. Because of the oversampling, the baud rate can be only a small fraction of the system clock rate, and thus this scheme is not appropriate for a high data rate.

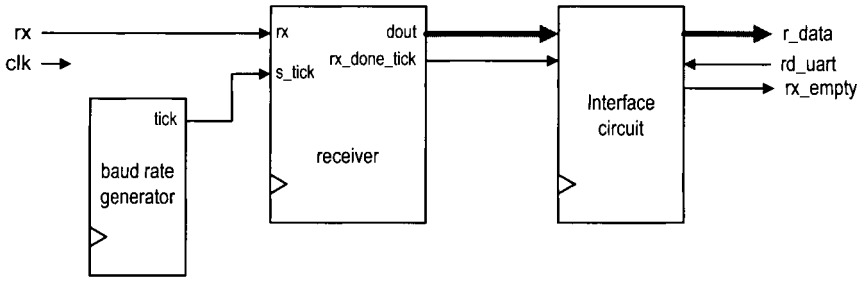


Figure 8.2 Conceptual block diagram of a UART receiving subsystem.

The conceptual block diagram of a UART receiving subsystem is shown in Figure 8.2. It consists of three major components:

- *UART receiver*: the circuit to obtain the data word via oversampling
- *Baud rate generator*: the circuit to generate the sampling ticks
- *Interface circuit*: the circuit that provides a buffer and status between the UART receiver and the system that uses the UART

8.2.2 Baud rate generator

The baud rate generator generates a sampling signal whose frequency is exactly 16 times the UART's designated baud rate. To avoid creating a new clock domain and violating the synchronous design principle, the sampling signal should function as enable ticks rather than the clock signal to the UART receiver, as discussed in Section 4.3.2.

For the 19,200 baud rate, the sampling rate has to be 307,200 (i.e., 19,200*16) ticks per second. Since the system clock rate is 50 MHz, the baud rate generator needs a mod-163 (i.e., $\frac{50*10^6}{307200}$) counter, in which a one-clock-cycle tick is asserted once every 163 clock cycles. The parameterized mod-m counter discussed in Section 4.3.2 can be used for this purpose by setting the `M` parameter to 163.

8.2.3 UART receiver

With an understanding of the oversampling procedure, we can derive the ASMD chart accordingly, as shown in Figure 8.3. To accommodate future modification, two constants are used in the description. The `D_BIT` constant indicates the number of data bits, and the `SB_TICK` constant indicates the number of ticks needed for the stop bits, which is 16, 24, and 32 for 1, 1.5, and 2 stop bits, respectively. `D_BIT` and `SB_TICK` are assigned to 8 and 16 in this design.

The chart follows the steps discussed in Section 8.2.1 and includes three major states, `start`, `data`, and `stop`, which represent the processing of the start bit, data bits, and stop bit. The `s_tick` signal is the enable tick from the baud rate generator and there are 16 ticks in a bit interval. Note that the FSMD stays in the same state unless the `s_tick` signal is asserted. There are two counters, represented by the `s` and `n` registers. The `s` register keeps track of the number of sampling ticks and counts to 7 in the `start` state, to 15 in the `data` state, and to `SB_TICK` in the `stop` state. The `n` register keeps track of the number of data bits received in the `data` state. The retrieved bits are shifted into and reassembled in the `b`

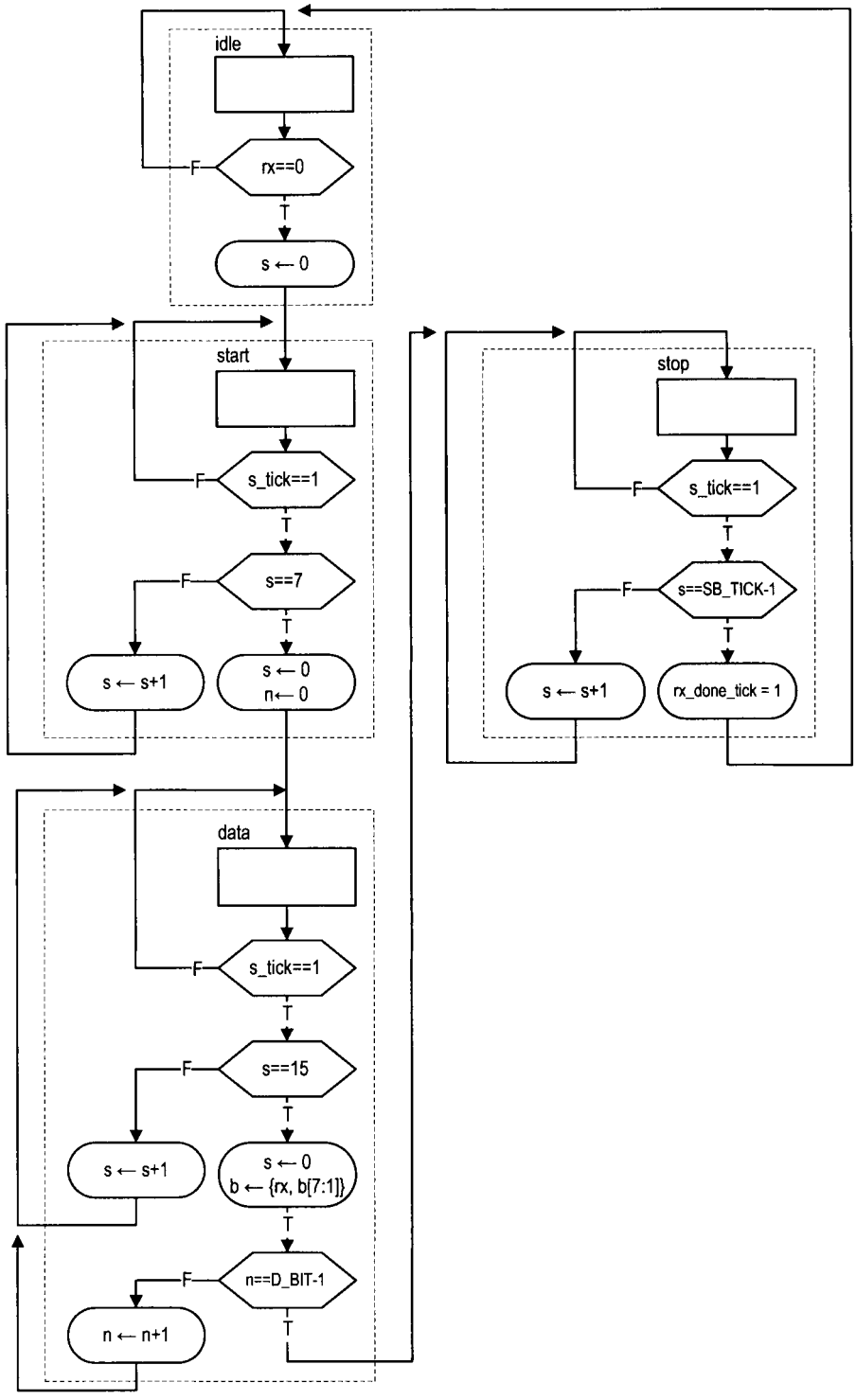


Figure 8.3 ASMD chart of a UART receiver.

register. A status signal, rx_done_tick, is included. It is asserted for one clock cycle after the receiving process is completed. The corresponding code is shown in Listing 8.1.

Listing 8.1 UART receiver

```
module uart_rx
   #(
     parameter DBIT = 8,      // # data bits
               SB_TICK = 16   // # ticks for stop bits
   )
   (
    input wire clk, reset,
    input wire rx, s_tick,
    output reg rx_done_tick,
    output wire [7:0] dout
   );

   // symbolic state declaration
   localparam [1:0]
      idle  = 2'b00,
      start = 2'b01,
      data  = 2'b10,
      stop  = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [3:0] s_reg, s_next;
   reg [2:0] n_reg, n_next;
   reg [7:0] b_reg, b_next;

   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            s_reg <= 0;
            n_reg <= 0;
            b_reg <= 0;
         end
      else
         begin
            state_reg <= state_next;
            s_reg <= s_next;
            n_reg <= n_next;
            b_reg <= b_next;
         end

   // FSMD next-state logic
   always @*
   begin
      state_next = state_reg;
      rx_done_tick = 1'b0;
      s_next = s_reg;
```

```
50        n_next = n_reg;
          b_next = b_reg;
          case (state_reg)
             idle:
                if (~rx)
55                 begin
                      state_next = start;
                      s_next = 0;
                   end
             start:
                if (s_tick)
60                 if (s_reg==7)
                      begin
                         state_next = data;
                         s_next = 0;
65                       n_next = 0;
                      end
                   else
                      s_next = s_reg + 1;
             data:
                if (s_tick)
70                 if (s_reg==15)
                      begin
                         s_next = 0;
                         b_next = {rx, b_reg[7:1]};
75                       if (n_reg==(DBIT-1))
                            state_next = stop ;
                          else
                            n_next = n_reg + 1;
                       end
80                 else
                      s_next = s_reg + 1;
             stop:
                if (s_tick)
                   if (s_reg==(SB_TICK-1))
85                    begin
                         state_next = idle;
                         rx_done_tick =1'b1;
                      end
                   else
90                    s_next = s_reg + 1;
          endcase
       end
       // output
       assign dout = b_reg;
95
    endmodule
```

8.2.4 Interface circuit

In a large system, a UART is usually a peripheral circuit for serial data transfer. The main system checks its status periodically to retrieve and process the received word. The

receiver's interface circuit has two functions. First, it provides a mechanism to signal the availability of a *new* word and to prevent the received word from being retrieved multiple times. Second, it can provide buffer space between the receiver and the main system. There are three commonly used schemes:

- A flag FF
- A flag FF and a one-word buffer
- A FIFO buffer

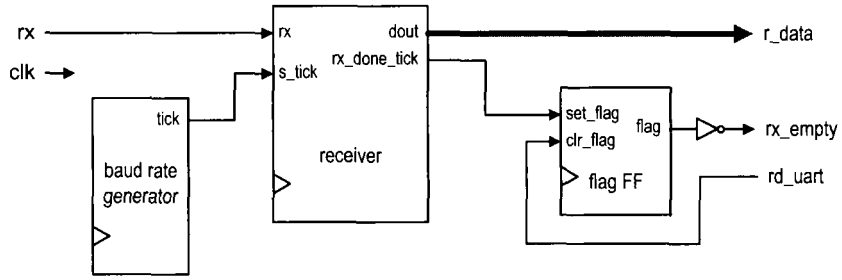
Note that the UART receiver asserts the `rx_ready_tick` signal one clock cycle after a data word is received.

The first scheme uses a *flag* FF to keep track of whether a new data word is available. The FF has two input signals. One is `set_flag`, which sets the flag FF to 1, and the other is `clr_flag`, which clears the flag FF to 0. The `rx_ready_tick` signal is connected to the `set_flag` signal and sets the flag when a new data word arrives. The main system checks the output of the flag FF to see whether a new data word is available. It asserts the `clr_flag` signal one clock cycle after retrieving the word. The top-level block diagram is shown in Figure 8.4(a). To be consistent with other schemes, the flag FF's output is inverted to generate the final `rx_empty` signal, which indicates that no new word is available. In this scheme, the main system retrieves the data word directly from the shift register of the UART receiver and does not provide any additional buffer space. If the remote system initiates a new transmission before the main system consumes the old data word (i.e., the flag FF is still asserted), the old word will be overwritten, an error known as *data overrun*.

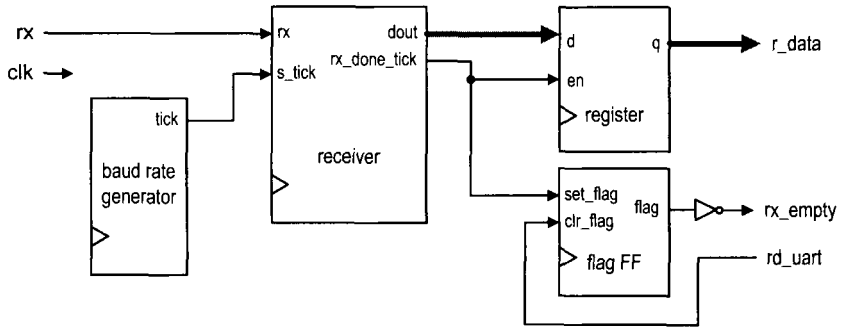
To provide some cushion, a one-word buffer can be added, as shown in Figure 8.4(b). When the `rx_ready_tick` signal is asserted, the received word is loaded to the buffer and the flag FF is set as well. The receiver can continue the operation without destroying the content of the last received word. Data overrun will not occur as long as the main system retrieves the word before a new word arrives. The code for this scheme is shown in Listing 8.2.

Listing 8.2 Interface with a flag FF and buffer

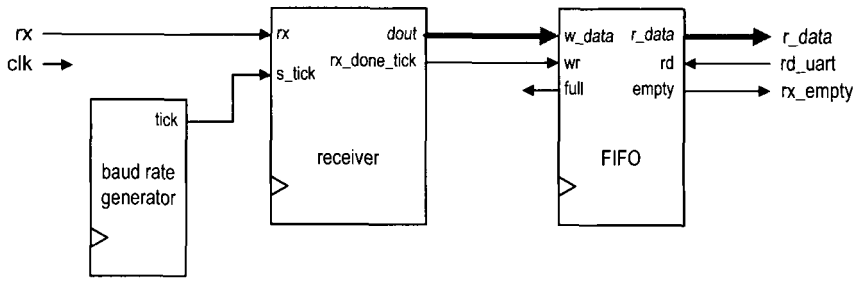
```
module flag_buf
   #(parameter W = 8) // # buffer bits
   (
    input wire clk, reset,
    input wire clr_flag, set_flag,
    input wire [W-1:0] din,
    output wire flag,
    output wire [W-1:0] dout
   );

   // signal declaration
   reg [W-1:0] buf_reg, buf_next;
   reg flag_reg, flag_next;


   // body
   // FF & register
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            buf_reg <= 0;
```



(a) Flag FF



(b) Flag FF and one-word buffer



(c) FIFO buffer

Figure 8.4 Interface circuit of a UART receiving subsystem.

```
            flag_reg <= 1'b0;
         end
      else
         begin
            buf_reg <= buf_next;
            flag_reg <= flag_next;
         end

   // next-state logic
   always @*
   begin
      buf_next = buf_reg;
      flag_next = flag_reg;
      if (set_flag)
         begin
            buf_next = din;
            flag_next = 1'b1;
         end
      else if (clr_flag)
         flag_next = 1'b0;
   end
   // output logic
   assign dout = buf_reg;
   assign flag = flag_reg;

endmodule
```

The third scheme uses a FIFO buffer discussed in Section 4.5.3. The FIFO buffer provides more buffering space and further reduces the chance of data overrun. We can adjust the desired number of words in FIFO to accommodate the processing need of the main system. The detailed block diagram is shown in Figure 8.4(c).

The `rx_ready_tick` signal is connected to the `wr` signal of the FIFO. When a new data word is received, the `wr` signal is asserted one clock cycle and the corresponding data is written to the FIFO. The main system obtains the data from FIFO's read port. After retrieving a word, it asserts the `rd` signal of the FIFO one clock cycle to remove the corresponding item. The `empty` signal of the FIFO can be used to indicate whether any received data word is available. A data-overrun error occurs when a new data word arrives and the FIFO is full.

8.3 UART TRANSMITTING SUBSYSTEM

The organization of a UART transmitting subsystem is similar to that of the receiving subsystem. It consists of a UART transmitter, baud rate generator, and interface circuit. The interface circuit is similar to that of the receiving subsystem except that the main system sets the flag FF or writes the FIFO buffer, and the UART transmitter clears the flag FF or reads the FIFO buffer.

The UART transmitter is essentially a shift register that shifts out data bits at a specific rate. The rate can be controlled by one-clock-cycle enable ticks generated by the baud rate generator. Because no oversampling is involved, the frequency of the ticks is 16 times slower than that of the UART receiver. Instead of introducing a new counter, the UART transmitter usually shares the baud rate generator of the UART receiver and uses an internal

counter to keep track of the number of enable ticks. A bit is shifted out every 16 enable ticks.

The ASMD chart of the UART transmitter is similar to that of the UART receiver. After assertion of the `tx_start` signal, the FSMD loads the data word and then gradually progresses through the `start`, `data`, and `stop` states to shift out the corresponding bits. It signals completion by asserting the `tx_done_tick` signal for one clock cycle. A 1-bit buffer, `tx_reg`, is used to filter out any potential glitch. The corresponding code is shown in Listing 8.3.

Listing 8.3 UART transmitter

```
module uart_tx
   #(
     parameter DBIT = 8,     // # data bits
               SB_TICK = 16  // # ticks for stop bits
   )
   (
    input wire clk, reset,
    input wire tx_start, s_tick,
    input wire [7:0] din,
    output reg tx_done_tick,
    output wire tx
   );

   // symbolic state declaration
   localparam [1:0]
      idle  = 2'b00,
      start = 2'b01,
      data  = 2'b10,
      stop  = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [3:0] s_reg, s_next;
   reg [2:0] n_reg, n_next;
   reg [7:0] b_reg, b_next;
   reg tx_reg, tx_next;

   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            s_reg <= 0;
            n_reg <= 0;
            b_reg <= 0;
            tx_reg <= 1'b1;
         end
      else
         begin
            state_reg <= state_next;
            s_reg <= s_next;
```

```
            n_reg <= n_next;
            b_reg <= b_next;
            tx_reg <= tx_next;
        end

    // FSMD next-state logic & functional units
    always @*
    begin
        state_next = state_reg;
        tx_done_tick = 1'b0;
        s_next = s_reg;
        n_next = n_reg;
        b_next = b_reg;
        tx_next = tx_reg ;
        case (state_reg)
            idle:
                begin
                    tx_next = 1'b1;
                    if (tx_start)
                        begin
                            state_next = start;
                            s_next = 0;
                            b_next = din;
                        end
                end
            start:
                begin
                    tx_next = 1'b0;
                    if (s_tick)
                        if (s_reg==15)
                            begin
                                state_next = data;
                                s_next = 0;
                                n_next = 0;
                            end
                        else
                            s_next = s_reg + 1;
                end
            data:
                begin
                    tx_next = b_reg[0];
                    if (s_tick)
                        if (s_reg==15)
                            begin
                                s_next = 0;
                                b_next = b_reg >> 1;
                                if (n_reg==(DBIT-1))
                                    state_next = stop ;
                                else
                                    n_next = n_reg + 1;
                            end
                        else
                            s_next = s_reg + 1;
```

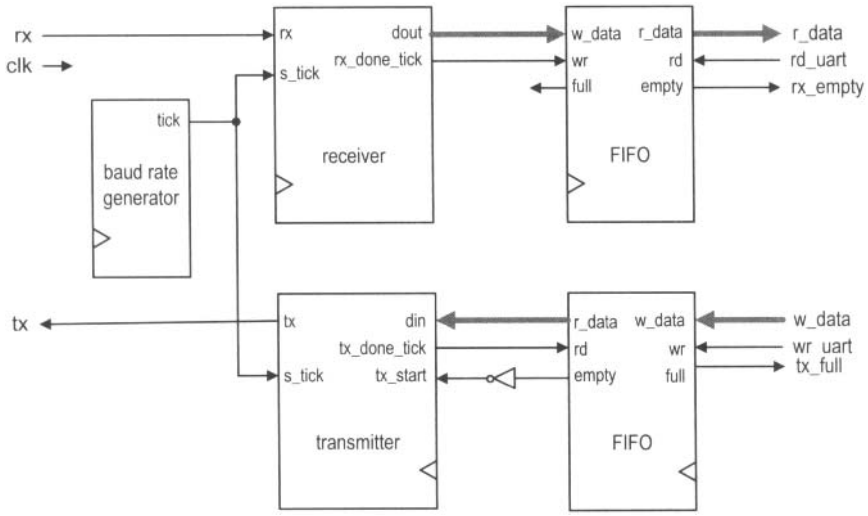


Figure 8.5 Block diagram of a complete UART.

```
            end
         stop:
            begin
               tx_next = 1'b1;
100            if (s_tick)
                  if (s_reg==(SB_TICK-1))
                     begin
                        state_next = idle;
                        tx_done_tick = 1'b1;
105                  end
                  else
                     s_next = s_reg + 1;
            end
      endcase
110 end
   // output
   assign tx = tx_reg;

endmodule
```

8.4 OVERALL UART SYSTEM

8.4.1 Complete UART core

By combining the receiving and transmitting subsystems, we can construct the complete UART core. The top-level diagram is shown in Figure 8.5. The block diagram can be described by component instantiation, and the corresponding code is shown in Listing 8.4.

Listing 8.4 UART top-level description

```
module uart
   #( // Default setting:
      // 19,200 baud, 8 data bits, 1 stop bit, 2^2 FIFO
      parameter DBIT = 8,      // # data bits
                SB_TICK = 16,  // # ticks for stop bits,
                               // 16/24/32 for 1/1.5/2 bits
                DVSR = 163,    // baud rate divisor
                               // DVSR = 50M/(16*baud rate)
                DVSR_BIT = 8,  // # bits of DVSR
                FIFO_W = 2     // # addr bits of FIFO
                               // # words in FIFO=2^FIFO_W
   )
   (
    input wire clk, reset,
    input wire rd_uart, wr_uart, rx,
    input wire [7:0] w_data,
    output wire tx_full, rx_empty, tx,
    output wire [7:0] r_data
   );

   // signal declaration
   wire tick, rx_done_tick, tx_done_tick;
   wire tx_empty, tx_fifo_not_empty;
   wire [7:0] tx_fifo_out, rx_data_out;

   //body
   mod_m_counter #(.M(DVSR), .N(DVSR_BIT)) baud_gen_unit
      (.clk(clk), .reset(reset), .q(), .max_tick(tick));

   uart_rx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_rx_unit
      (.clk(clk), .reset(reset), .rx(rx), .s_tick(tick),
       .rx_done_tick(rx_done_tick), .dout(rx_data_out));

   fifo #(.B(DBIT), .W(FIFO_W)) fifo_rx_unit
      (.clk(clk), .reset(reset), .rd(rd_uart),
       .wr(rx_done_tick), .w_data(rx_data_out),
       .empty(rx_empty), .full(), .r_data(r_data));

   fifo #(.B(DBIT), .W(FIFO_W)) fifo_tx_unit
      (.clk(clk), .reset(reset), .rd(tx_done_tick),
       .wr(wr_uart), .w_data(w_data), .empty(tx_empty),
       .full(tx_full), .r_data(tx_fifo_out));

   uart_tx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_tx_unit
      (.clk(clk), .reset(reset), .tx_start(tx_fifo_not_empty),
       .s_tick(tick), .din(tx_fifo_out),
       .tx_done_tick(tx_done_tick), .tx(tx));

   assign tx_fifo_not_empty = ~tx_empty;

endmodule
```

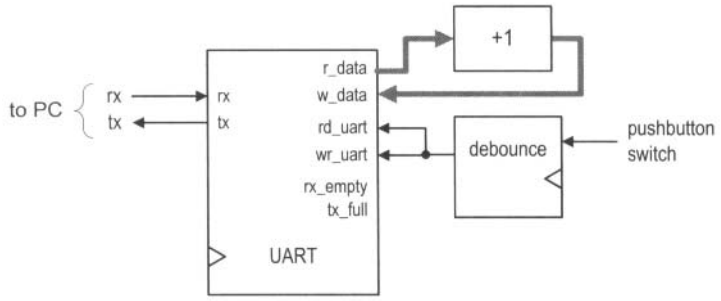


Figure 8.6 Block diagram of a UART verification circuit.

Xilinx specific

In the picoBlaze source file (discussed in Chapter 15), Xilinx supplies a customized UART module with similar functionality. Unlike our implementation, the module is described using low-level Xilinx primitives. It can be considered as a gate-level description that utilizes Xilinx-specific components. Since the designer has the expert knowledge of Xilinx devices and takes advantage of its architecture, its implementation is more efficient than the generic RT-level device-independent description of this chapter. It is instructive to compare the code complexity and the circuit size of the two descriptions.

8.4.2 UART verification configuration

Verification circuit We use a loop-back circuit and a PC to verify the UART's operation. The block diagram is shown in Figure 8.6. In the circuit, the serial port of the S3 board is connected to the serial port of a PC. When we send a character from the PC, the received data word is stored in the UART receiver's four-word FIFO buffer. When retrieved (via the `r_data` port), the data word is incremented by 1 and then sent back to the transmitter (via the `w_data` port). The debounced pushbutton switch produces a single one-clock-cycle tick when pressed and it is connected to the `rd_uart` and `wr_uart` signals. When the tick is generated, it removes one word from the receiver's FIFO and writes the incremented word to the transmitter's FIFO for transmission. For example, we can first type `HAL` in the PC and the three data words are stored in the FIFO buffer of the UART receiver. We can then push the button on the S3 board three times. The three successive characters, `IBM`, will be transmitted back and displayed. The UART's `r_data` port is also connected to the eight LEDs of the S3 board, and its `tx_full` and `rx_empty` signals are connected to the two horizontal bars of the rightmost digit of the seven-segment display. The code is shown in Listing 8.5.

Listing 8.5 UART verification circuit

```
module uart_test
   (
    input wire clk, reset,
    input wire rx,
    input wire [2:0] btn,
    output wire tx,
    output wire [3:0] an,
    output wire [7:0] sseg, led
   );
```

```
10
    // signal declaration
    wire tx_full, rx_empty, btn_tick;
    wire [7:0] rec_data, rec_data1;

15  // body
    // instantiate uart
    uart uart_unit
       (.clk(clk), .reset(reset), .rd_uart(btn_tick),
        .wr_uart(btn_tick), .rx(rx), .w_data(rec_data1),
20      .tx_full(tx_full), .rx_empty(rx_empty),
        .r_data(rec_data), .tx(tx));
    // instantiate debounce circuit
    debounce btn_db_unit
       (.clk(clk), .reset(reset), .sw(btn[0]),
25      .db_level(), .db_tick(btn_tick));
    // incremented data loops back
    assign rec_data1 = rec_data + 1;
    // LED display
    assign led = rec_data;
30  assign an = 4'b1110;
    assign sseg = {1'b1, ~tx_full, 2'b11, ~rx_empty, 3'b111};

endmodule
```

HyperTerminal of Windows On the PC side, Windows' HyperTerminal program can be used as a virtual terminal to interact with the S3 board. To be compatible with our customized UART, it has to be configured as 19,200 baud, 8 data bits, 1 stop bit, and no parity bit. The basic procedure is:

1. Select Start ≻ Programs ≻ Accessories ≻ Communications ≻ HyperTerminal. The HyperTerminal dialog appears.
2. Type a name for this connection, say `fpga_192`. Click OK. This connection can be saved and invoked later.
3. A Connect_to dialog appears. Press the Connecting Using field and select the desired serial port (e.g., COM1). Click OK.
4. The Port Setting dialog appears. Configure the port as follows:
   - Bits per second: 19200
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
   - Flow control: None

   Click OK.
5. Select File ≻ Properties ≻ Setting. Click ASCII Setup and check the Echo typed characters locally box. Click OK twice. This will allow the typed characters to be shown on the screen.

The HyperTerminal program is set up now and ready to communicate with the S3 board. We can type a few keys and observe the LEDs of the S3 board. Note that the received words are stored in the FIFO buffer and only the first received data word is displayed. After we press the pushbutton, the first data word will be removed from the FIFO and the incremented word will be looped back to the PC's serial port and displayed in the