CHAPTER 7

UART

7.1 INTRODUCTION

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 a RS-232 port with the 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 the 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, or 2.

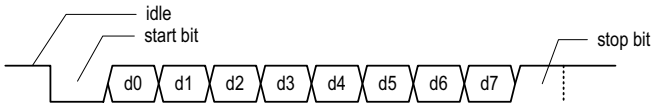


Figure 7.1 Transmission of a byte.

The transmission with 8 data bits, no parity, and 1 stop bit is shown in Figure 7.1. 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.

7.2 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.

7.2.1 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 only be a small fraction of the system clock rate, and thus this scheme is not appropriate for a high data rate.

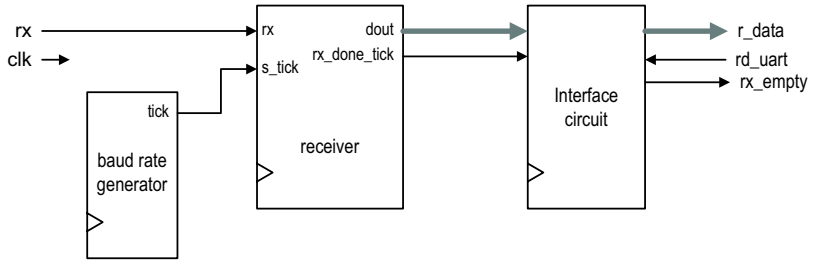


Figure 7.2 Conceptual block diagram of a UART receiving subsystem.

The conceptual block diagram of a UART receiving subsystem is shown in Figure 7.2. 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 buffer and status between the UART receiver and the system that uses the UART

7.2.2 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 the 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` generic to 163.

7.2.3 UART receiver

With an understanding of the oversampling procedure, we can derive the ASMD chart accordingly, as shown in Figure 7.3. 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 7.2.1 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`

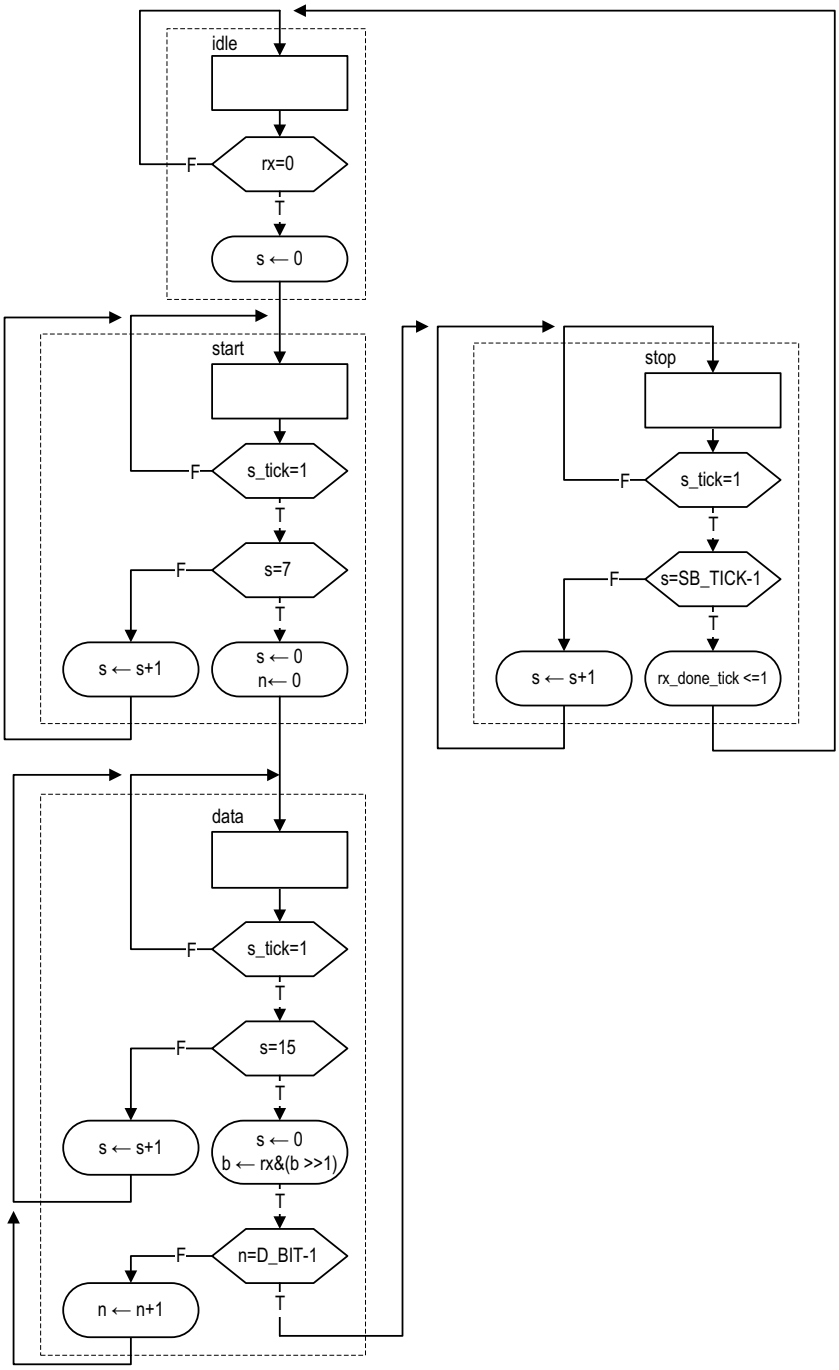


Figure 7.3 ASMD chart of a UART receiver.

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 7.1.

Listing 7.1 UART receiver

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_rx is
   generic(
      DBIT: integer:=8;     -- # data bits
      SB_TICK: integer:=16  -- # ticks for stop bits
   );
   port(
      clk, reset: in std_logic;
      rx: in std_logic;
      s_tick: in std_logic;
      rx_done_tick: out std_logic;
      dout: out std_logic_vector(7 downto 0)
   );
end uart_rx ;

architecture arch of uart_rx is
   type state_type is (idle, start, data, stop);
   signal state_reg, state_next: state_type;
   signal s_reg, s_next: unsigned(3 downto 0);
   signal n_reg, n_next: unsigned(2 downto 0);
   signal b_reg, b_next: std_logic_vector(7 downto 0);
begin
   -- FSMD state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         s_reg <= (others=>'0');
         n_reg <= (others=>'0');
         b_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         s_reg <= s_next;
         n_reg <= n_next;
         b_reg <= b_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,s_reg,n_reg,b_reg,s_tick,rx)
   begin
      state_next <= state_reg;
      s_next <= s_reg;
      n_next <= n_reg;
      b_next <= b_reg;
      rx_done_tick <='0';
      case state_reg is
         when idle =>
```

```
            if rx='0' then
               state_next <= start;
               s_next <= (others=>'0');
            end if;
         when start =>
            if (s_tick = '1') then
               if s_reg=7 then
                  state_next <= data;
                  s_next <= (others=>'0');
                  n_next <= (others=>'0');
               else
                  s_next <= s_reg + 1;
               end if;
            end if;
         when data =>
            if (s_tick = '1') then
               if s_reg=15 then
                  s_next <= (others=>'0');
                  b_next <= rx & b_reg(7 downto 1) ;
                  if n_reg=(DBIT-1) then
                     state_next <= stop ;
                  else
                     n_next <= n_reg + 1;
                  end if;
               else
                  s_next <= s_reg + 1;
               end if;
            end if;
         when stop =>
            if (s_tick = '1') then
               if s_reg=(SB_TICK-1) then
                  state_next <= idle;
                  rx_done_tick <='1';
               else
                  s_next <= s_reg + 1;
               end if;
            end if;
      end case;
   end process;
   dout <= b_reg;
end arch;
```

7.2.4 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

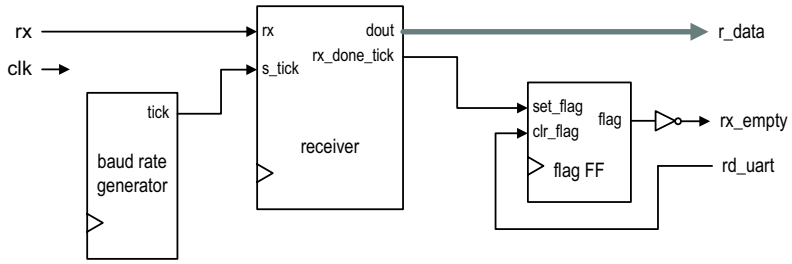
Note that the UART receiver asserts the `rx_ready_tick` signal one clock cycle after a data word is received.

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 7.4(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*.

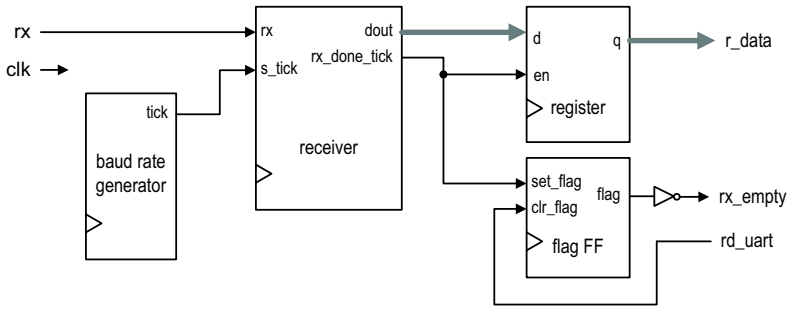
To provide some cushion, a one-word buffer can be added, as shown in Figure 7.4(b). When the `rx_ready_tick` signal is asserted, the received word is loaded to the buffer and the flag FF is set as well. The receiver can continue the operation without destroying the content of the last received word. 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 7.2.

Listing 7.2 Interface with a flag FF and buffer

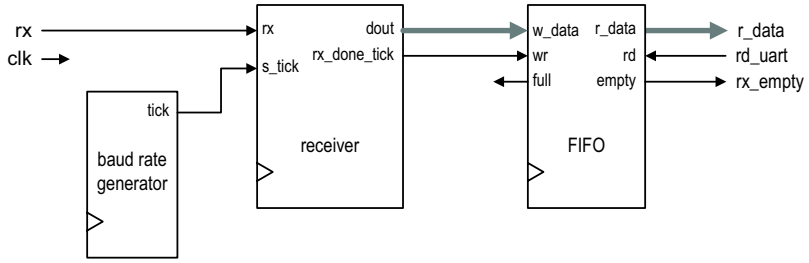
```
library ieee;
use ieee.std_logic_1164.all;
entity flag_buf is
   generic(W: integer:=8);
   port(
      clk, reset: in std_logic;
      clr_flag, set_flag: in std_logic;
      din: in std_logic_vector(W-1 downto 0);
      dout: out std_logic_vector(W-1 downto 0);
      flag: out std_logic
   );
end flag_buf;

architecture arch of flag_buf is
   signal buf_reg, buf_next: std_logic_vector(W-1 downto 0);
   signal flag_reg, flag_next: std_logic;
begin
   -- FF & register
   process(clk,reset)
   begin
      if reset='1' then
         buf_reg <= (others=>'0');
         flag_reg <= '0';
      elsif (clk'event and clk='1') then
         buf_reg <= buf_next;
         flag_reg <= flag_next;
      end if;
```



(a) Flag FF



(b) Flag FF and one-word buffer



(c) FIFO buffer

Figure 7.4 Interface circuit of a UART receiving subsystem.

```
   end process;
   -- next-state logic
30 process(buf_reg,flag_reg,set_flag,clr_flag,din)
   begin
      buf_next <= buf_reg;
      flag_next <= flag_reg;
      if (set_flag='1') then
35       buf_next <= din;
         flag_next <= '1';
      elsif (clr_flag='1') then
         flag_next <= '0';
      end if;
40 end process;
   -- output logic
   dout <= buf_reg;
   flag <= flag_reg;
end arch;
```

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 7.4(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.

7.3 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 7.3.

Listing 7.3 UART transmitter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_tx is
   generic(
      DBIT: integer:=8;     -- # data bits
      SB_TICK: integer:=16  -- # ticks for stop bits
   );
   port(
      clk, reset: in std_logic;
      tx_start: in std_logic;
      s_tick: in std_logic;
      din: in std_logic_vector(7 downto 0);
      tx_done_tick: out std_logic;
      tx: out std_logic
   );
end uart_tx ;

architecture arch of uart_tx is
   type state_type is (idle, start, data, stop);
   signal state_reg, state_next: state_type;
   signal s_reg, s_next: unsigned(3 downto 0);
   signal n_reg, n_next: unsigned(2 downto 0);
   signal b_reg, b_next: std_logic_vector(7 downto 0);
   signal tx_reg, tx_next: std_logic;
begin
   -- FSMD state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         s_reg <= (others=>'0');
         n_reg <= (others=>'0');
         b_reg <= (others=>'0');
         tx_reg <= '1';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         s_reg <= s_next;
         n_reg <= n_next;
         b_reg <= b_next;
         tx_reg <= tx_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,s_reg,n_reg,b_reg,s_tick,
           tx_reg,tx_start,din)
   begin
      state_next <= state_reg;
      s_next <= s_reg;
      n_next <= n_reg;
      b_next <= b_reg;
      tx_next <= tx_reg ;
```

```
      tx_done_tick <= '0';
      case state_reg is
         when idle =>
            tx_next <= '1';
            if tx_start='1' then
               state_next <= start;
               s_next <= (others=>'0');
               b_next <= din;
            end if;
         when start =>
            tx_next <= '0';
            if (s_tick = '1') then
               if s_reg=15 then
                  state_next <= data;
                  s_next <= (others=>'0');
                  n_next <= (others=>'0');
               else
                  s_next <= s_reg + 1;
               end if;
            end if;
         when data =>
            tx_next <= b_reg(0);
            if (s_tick = '1') then
               if s_reg=15 then
                  s_next <= (others=>'0');
                  b_next <= '0' & b_reg(7 downto 1) ;
                  if n_reg=(DBIT-1) then
                     state_next <= stop ;
                  else
                     n_next <= n_reg + 1;
                  end if;
               else
                  s_next <= s_reg + 1;
               end if;
            end if;
         when stop =>
            tx_next <= '1';
            if (s_tick = '1') then
               if s_reg=(SB_TICK-1) then
                  state_next <= idle;
                  tx_done_tick <= '1';
               else
                  s_next <= s_reg + 1;
               end if;
            end if;
      end case;
   end process;
   tx <= tx_reg;
end arch;
```

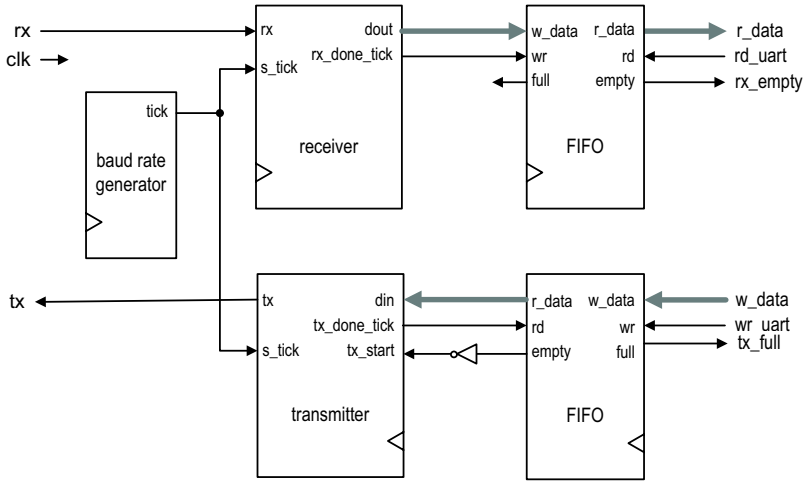


Figure 7.5 Block diagram of a complete UART.

7.4 OVERALL UART SYSTEM

7.4.1 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 7.5. The block diagram can be described by component instantiation, and the corresponding code is shown in Listing 7.4.

Listing 7.4 UART top-level description

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart is
   generic(
      -- Default setting:
      -- 19200 baud, 8 data bits, 1 stop bit, 2^2 FIFO
      DBIT: integer:=8;     -- # data bits
      SB_TICK: integer:=16; -- # ticks for stop bits, 16/24/32
                            --   for 1/1.5/2 stop bits
      DVSR: integer:= 163;  -- baud rate divisor
                            -- DVSR = 50M/(16*baud rate)
      DVSR_BIT: integer:=8; -- # bits of DVSR
      FIFO_W: integer:=2    -- # addr bits of FIFO
                            -- # words in FIFO=2^FIFO_W
   );
   port(
      clk, reset: in std_logic;
      rd_uart, wr_uart: in std_logic;
      rx: in std_logic;
      w_data: in std_logic_vector(7 downto 0);
      tx_full, rx_empty: out std_logic;
```

```
      r_data: out std_logic_vector(7 downto 0);
      tx: out std_logic
   );
end uart;

architecture str_arch of uart is
   signal tick: std_logic;
   signal rx_done_tick: std_logic;
   signal tx_fifo_out: std_logic_vector(7 downto 0);
   signal rx_data_out: std_logic_vector(7 downto 0);
   signal tx_empty, tx_fifo_not_empty: std_logic;
   signal tx_done_tick: std_logic;
begin
   baud_gen_unit: entity work.mod_m_counter(arch)
      generic map(M=>DVSR, N=>DVSR_BIT)
      port map(clk=>clk, reset=>reset,
               q=>open, max_tick=>tick);
   uart_rx_unit: entity work.uart_rx(arch)
      generic map(DBIT=>DBIT, SB_TICK=>SB_TICK)
      port map(clk=>clk, reset=>reset, rx=>rx,
               s_tick=>tick, rx_done_tick=>rx_done_tick,
               dout=>rx_data_out);
   fifo_rx_unit: entity work.fifo(arch)
      generic map(B=>DBIT, W=>FIFO_W)
      port map(clk=>clk, reset=>reset, rd=>rd_uart,
               wr=>rx_done_tick, w_data=>rx_data_out,
               empty=>rx_empty, full=>open, r_data=>r_data);
   fifo_tx_unit: entity work.fifo(arch)
      generic map(B=>DBIT, W=>FIFO_W)
      port map(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_unit: entity work.uart_tx(arch)
      generic map(DBIT=>DBIT, SB_TICK=>SB_TICK)
      port map(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);
   tx_fifo_not_empty <= not tx_empty;
end str_arch;
```

Xilinx specific

In the picoBlaze source file (discussed in Chapter 14), 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.

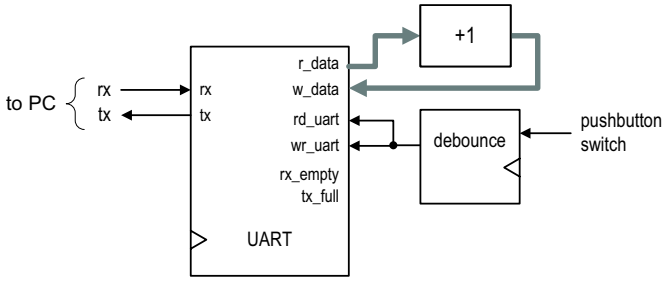


Figure 7.6 Block diagram of a UART verification circuit.

7.4.2 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 7.6. 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 then can 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 7.5.

Listing 7.5 UART verification circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_test is
   port(
      clk, reset: in std_logic;
      btn: std_logic_vector(2 downto 0);
      rx: in std_logic;
      tx: out std_logic;
      led: out std_logic_vector(7 downto 0);
      sseg: out std_logic_vector(7 downto 0);
      an: out std_logic_vector(3 downto 0)
   );
end uart_test;

architecture arch of uart_test is
   signal tx_full, rx_empty: std_logic;
   signal rec_data,rec_data1: std_logic_vector(7 downto 0);
```

```
    signal btn_tick: std_logic;
20 begin
    -- instantiate uart
    uart_unit: entity work.uart(str_arch)
       port map(clk=>clk, reset=>reset, rd_uart=>btn_tick,
                wr_uart=>btn_tick, rx=>rx, w_data=>rec_data1,
25              tx_full=>tx_full, rx_empty=>rx_empty,
                r_data=>rec_data, tx=>tx);
    -- instantiate debounce circuit
    btn_db_unit: entity work.debounce(fsmd_arch)
       port map(clk=>clk, reset=>reset, sw=>btn(0),
30              db_level=>open, db_tick=>btn_tick);
    -- incremented data loop back
    rec_data1 <= std_logic_vector(unsigned(rec_data)+1);
    -- led display
    led <= rec_data;
35  an <= "1110";
    sseg <= '1' & (not tx_full) & "11" & (not rx_empty) & "111";
end arch;
```

HyperTerminal of Windows On PC's 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 HyperTerminal window. The full and empty status of the respective FIFO buffers can be tested by consecutively receiving and transmitting more than four data words.

ASCII code In HyperTerminal, characters are sent in ASCII code, which is 7 bits and consists of 128 code words, including regular alphabets, digits, punctuation symbols, and

nonprintable control characters. The characters and their code words (in hexadecimal format) are shown in Table 7.1. The nonprintable characters are shown enclosed in parentheses, such as (del). Several nonprintable characters may introduce special action when received:

- (nul): null byte, which is the all-zero pattern
- (bel): generate a bell sound, if supported
- (bs): backspace
- (ht): horizontal tab
- (nl): new line
- (vt): vertical tab
- (np): new page
- (cr): carriage return
- (esc): escape
- (sp): space
- (del): delete, which is also the all-one pattern

Since we use the PC's serial port to communicate with the S3 board in many experiments and projects, the following observations help us to manipulate and process the ASCII code:

- When the first hex digit in a code word is 0_{16} or 1_{16}, the corresponding character is a control character.
- When the first hex digit in a code word is 2_{16} or 3_{16}, the corresponding character is a digit or punctuation.
- When the first hex digit in a code word is 4_{16} or 5_{16}, the corresponding character is generally an uppercase letter.
- When the first hex digit in a code word is 6_{16} or 7_{16}, the corresponding character is generally a lowercase letter.
- If the first hex digit in a code word is 3_{16}, the lower hex digit represents the corresponding decimal digit.
- The upper- and lowercase letters differ in a single bit and can be converted to each other by adding or subtracting 20_{16} or inverting the sixth bit.

Note that the ASCII code uses only 7 bits, but a data word is normally composed of 8 bits (i.e., a byte). The PC uses an extended set in which the MSB is 1 and the characters are special graphics symbols. This code, however, is not part of the ASCII standard.

7.5 CUSTOMIZING A UART

The UART discussed in previous sections is customized for a particular configuration. The design and code can easily be modified to accommodate other required features:

- *Baud rate*. The baud rate is controlled by the frequency of the sampling ticks of the baud rate generator. The frequency can be changed by revising the `M` generic of the mod-m counter, which is represented as the `DVSR` constant in code.
- *Number of data bits*. The number of data bits can be changed by modifying the upper limit of the `n_reg` register, which is specified as the `DBIT` constant in code.
- *Parity bit*. A parity bit can be included by introducing a new state between the `data` and `stop` states in the ASMD chart in Figure 7.3.
- *Number of stop bits*. The number of stop bits can be changed by modifying the upper limit of the `s_reg` register in the `stop` state of the ASMD chart. The `SB_TICK` constant is used for this purpose. It can be 16, 24, or 32, which is for 1, 1.5, or 2 stop bits, respectively.

Table 7.1 ASCII codes

Code	Char	Code	Char	Code	Char	Code	Char
00	(nul)	20	(sp)	40	@	60	‘
01	(soh)	21	!	41	A	61	a
02	(stx)	22	"	42	B	62	b
03	(etx)	23	#	43	C	63	c
04	(eot)	24	$	44	D	64	d
05	(enq)	25	%	45	E	65	e
06	(ack)	26	&	46	F	66	f
07	(bel)	27	’	47	G	67	g
08	(bs)	28	(	48	H	68	h
09	(ht)	29	)	49	I	69	i
0a	(nl)	2a	*	4a	J	6a	j
0b	(vt)	2b	+	4b	K	6b	k
0c	(np)	2c	,	4c	L	6c	l
0d	(cr)	2d	-	4d	M	6d	m
0e	(so)	2e	.	4e	N	6e	n
0f	(si)	2f	/	4f	O	6f	o
10	(dle)	30	0	50	P	70	p
11	(dc1)	31	1	51	Q	71	q
12	(dc2)	32	2	52	R	72	r
13	(dc3)	33	3	53	S	73	s
14	(dc4)	34	4	54	T	74	t
15	(nak)	35	5	55	U	75	u
16	(syn)	36	6	56	V	76	v
17	(etb)	37	7	57	W	77	w
18	(can)	38	8	58	X	78	x
19	(em)	39	9	59	Y	79	y
1a	(sub)	3a	:	5a	Z	7a	z
1b	(esc)	3b	;	5b	[	7b	{
1c	(fs)	3c	<	5c	\	7c	|
1d	(gs)	3d	=	5d	]	7d	}
1e	(rs)	3e	>	5e	^	7e	~
1f	(us)	3f	?	5f	_	7f	(del)

- *Error checking*. Three types of errors can be detected in the UART receiving subsystem:
  - *Parity error*. If the parity bit is included, the receiver can check the correctness of the received parity bit.
  - *Frame error*. The receiver can check the received value in the `stop` state. If the value is not '1', the frame error occurs.
  - *Buffer overrun error*. This happens when the main system does not retrieve the received words in a timely manner. The UART receiver can check the value of the buffer's `flag_reg` signal or FIFO's `full` signal when the received word is ready to be stored (i.e., when the `rx_done_tick` signal is generated). Data overrun occurs if the `flag_reg` or `full` signal is still asserted.

7.6 BIBLIOGRAPHIC NOTES

Although the RS-232 standard is very old, it still provides a simple and reliable low-speed communication link between two devices. The *Wikipedia* Web site has a good overview article and several useful links on the subject (search with the keyword `RS232`). *Serial Port Complete* by Jan Axelson provides information on interfacing hardware devices to PC's serial port.

7.7 SUGGESTED EXPERIMENTS

7.7.1 Full-featured UART

The alternative to the customized UART is to include all features in design and to dynamically configure the UART as needed. Consider a full-featured UART that uses additional input signals to specify the baud rate, type of parity bit, and the numbers of data bits and stop bits. The UART also includes an error signal. In addition to the I/O signals of the `uart_top` design in Listing 7.4, the following signals are required:

- `bd_rate`: 2-bit input signal specifying the baud rate, which can be 1200, 2400, 4800, or 9600 baud
- `d_num`: 1-bit input signal specifying the number of data bits, which can be 7 or 8
- `s_num`: 1-bit input signal specifying the number of stop bits, which can be 1 or 2
- `par`: 2-bit input signal specifying the desired parity scheme, which can be no parity, even parity, or odd parity
- `err`: 3-bit output signal in which the bits indicate the existence of the parity error, frame error, and data overrun error

Derive this circuit as follows:

1. Modify the ASMD chart in Figure 7.3 to accommodate the required extensions.
2. Revise the UART receiver code according to the ASMD chart.
3. Revise the UART transmitter code to accommodate the required extensions.
4. Revise the top-level UART code and the verification circuit. Use the onboard switches for the additional input signals and three LEDs for the error signals. Synthesize the verification circuit.
5. Create different configurations in HyperTerminal and verify operation of the UART circuit.

7.7.2 UART with an automatic baud rate detection circuit

The most commonly used number of data bits of a serial connection is eight, which corresponds to a byte. When a regular ASCII code is used in communication (as we type in the HyperTerminal window), only seven LSBs are used and the MSB is '0'. If the UART is configured as 8 data bits, 1 stop bit, and no parity, the received word is in the form of `0_dddd_ddd0_1`, in which `d` is a data bit and can be '0' or '1'. Assume that there is sufficient time between the first word and subsequent transmissions. We can determine the baud rate by measuring the time interval between the first '0' and last '0'. Based on this observation, we can derive a UART with an automatic baud rate detection circuit. In this scheme, the transmitting system first sends an ASCII code for rate detection and then resumes normal operation afterward. The receiving subsystem uses the first word to determine a baud rate and then uses this rate for the baud rate generator for the remaining transmission.

Assume that UART configuration is 8 data bits, 1 stop bit, and no parity bit, and the baud rate can be 4800, 9600, or 19,200 baud. The revised UART receiver should have two operation modes. It is initially in the "detection mode" and waits for the first word. After the word is received and the baud rate is determined, the receiver enters "normal mode" and the UART operates in a regular fashion. Derive the UART as follows:

1. Draw the ASMD chart for the automatic baud rate detector circuit.
2. Derive the VHDL code for the ASMD chart. Use three LEDs on the S3 board to indicate the baud rate of the incoming signal.
3. Modify the UART to include three different baud rates: 4800, 9600, and 19,200. This can be achieved by using a register for the divisor of the baud rate generator and loading the value according to the desired baud rate.
4. Create a top-level FSMD to keep track of the mode and to control and coordinate operation of the baud rate detection circuit and the regular UART receiver. Use a pushbutton switch on the S3 board to force the UART into the detection mode.
5. Revise the top-level UART code and the verification circuit. Synthesize the verification circuit.
6. Create different configurations in HyperTerminal and verify operation of the UART.

7.7.3 UART with an automatic baud rate and parity detection circuit

In addition to the baud rate, we assume that the parity scheme also needs to be determined automatically, which can be no parity, even parity, or odd parity. Expand the previous automatic baud rate detection circuit to detect the parity configuration and repeat Experiment 7.7.2.

7.7.4 UART-controlled stopwatch

Consider the enhanced stopwatch in Experiment 4.7.6. Operation of the stopwatch is controlled by three switches on the S3 board. With the UART, we can use PC's HyperTerminal to send commands to and retrieve time from the stopwatch:

- When a `c` or `C` (for "clear") ASCII code is received, the stopwatch aborts current counting, is cleared to zero, and sets the counting direction to "up."
- When a `g` or `G` (for "go") ASCII code is received, the stopwatch starts to count.
- When a `p` or `P` (for "pause") ASCII code is received, counting pauses.
- When a `u` or `U` (for "up-down") ASCII code is received, the stopwatch reverses the direction of counting.

- When a `r` or `R` (for "receive") ASCII code is received, the stopwatch transmits the current time to the PC. The time should be displayed as "`DD.D`", where `D` is a decimal digit.
- All other codes will be ignored.

Design the new stopwatch, synthesize the circuit, connect it to a PC, and use HyperTerminal to verify its operation.

7.7.5 UART-controlled rotating LED banner

Consider the rotating LED banner circuit in Experiment 4.7.5. With the UART, we can use PC's HyperTerminal to control its operation and dynamically modify the digits in the banner:

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

Design the new rotating LED banner, synthesize the circuit, connect it to a PC, and use HyperTerminal to verify its operation.