Digital Design Group Project
EENG 28010

UART Transmitter and Receiver Modules

Dr. Faezeh Arab Hassani
Department of Electrical and Electronic Engineering

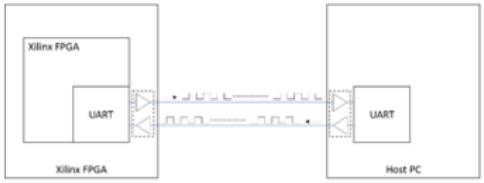
Outline

- UART
- UART Transmitter (Tx) Module
- UART Receiver (Rx) Module

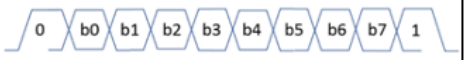
3

UART

- A **universal asynchronous receiver-transmitter** (**UART**) is a hardware device for asynchronous serial communication.
- It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits.
- We use the RS-232 serial communication protocol for the peak detector system. The RS-232 protocol is a popular protocol for asynchronous transmission, when two communicating modules do not share the same clock.



Peak detector system overview

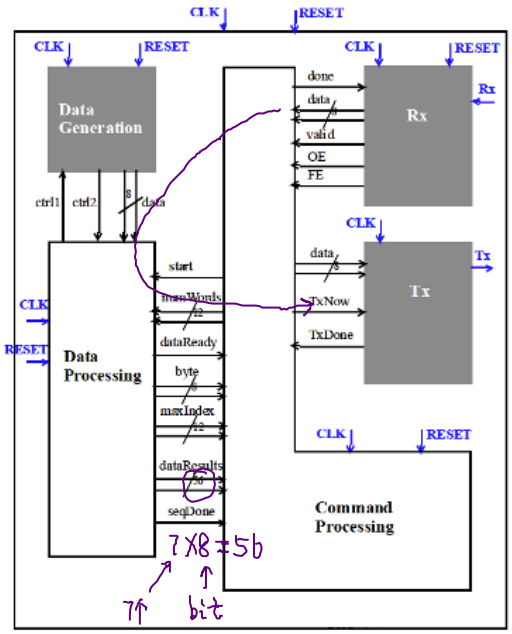


An RS-232 frame consists of at least 10 bits, a start bit, a byte of data, and a stop bit.

University of BRISTOL

We don't have parity bit in the RS-232 frame. Parity bit describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission.

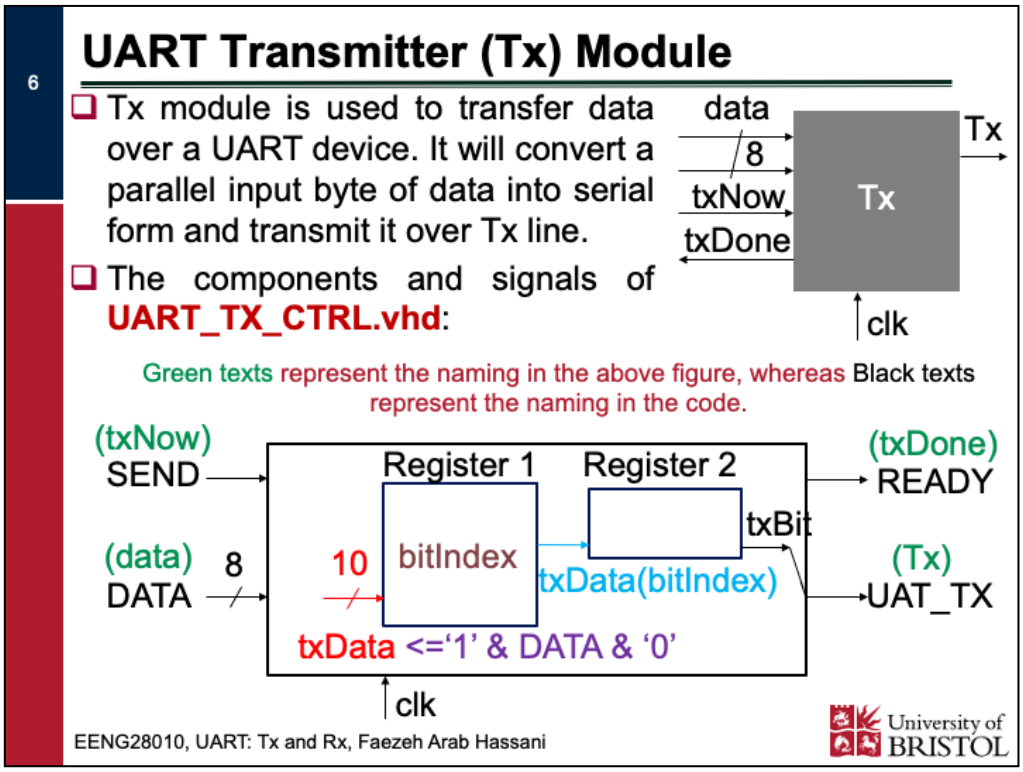
Global Architecture of Peak Detector



University of BRISTOL

Outline

- UART
- UART Transmitter (Tx) Module
- UART Receiver (Rx) Module



A separate register 2 ensures no changes in output

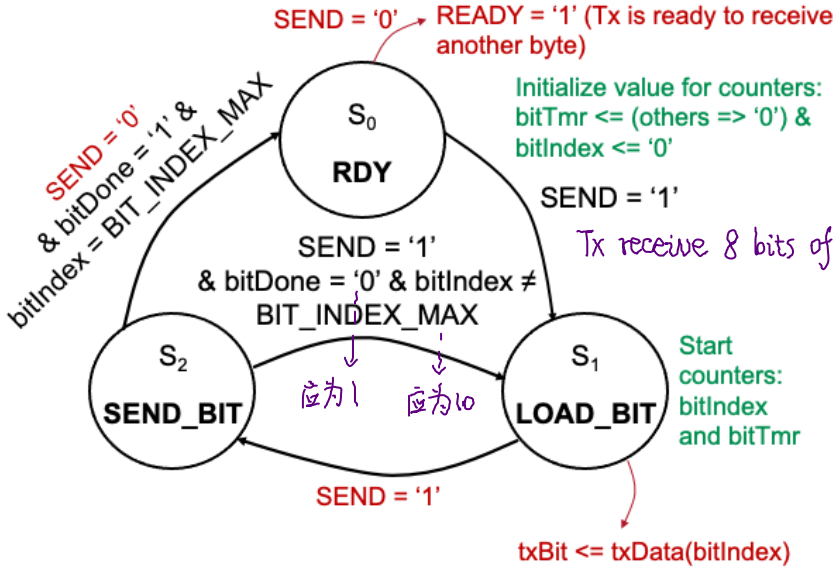
Baud Rate

- Baud is a common unit of rate measurement, which is one of the components that determine the speed of communication over a data channel.
- Baud can be expressed in bits per second.
- Baud rate for the UART device is 9600. If the clock frequency (f) is 100 MHz, [round(f/Baud)-1] will give you the number of clocks that are required to transfer a single bit. (e.g. 10416 that can be presented as '10100010110000'). => We need to have a counter named as bitTmr to count the number of clocks.
- BIT_TMR_MAX is the maximum number of 10416 or '10100010110000'.
- When bitTmr = BIT_TMR_MAX => bitDone = '1'

The Other Counter

- We need to have a counter named as bitIndex to count the number of bits received by register 1
- When all of 10 bits are received => BIT_INDEX_MAX = 10

State Diagram for Tx Module



Entity

```
38  library IEEE;
39  use IEEE.STD_LOGIC_1164.ALL;
40  use IEEE.std_logic_unsigned.all;
41
42  entity UART_TX_CTRL is
43      Port ( SEND : in  STD_LOGIC;
44             DATA : in  STD_LOGIC_VECTOR (7 downto 0);
45             CLK : in  STD_LOGIC;
46             READY : out  STD_LOGIC;
47             UART_TX : out  STD_LOGIC);
48  end UART_TX_CTRL;
49
```

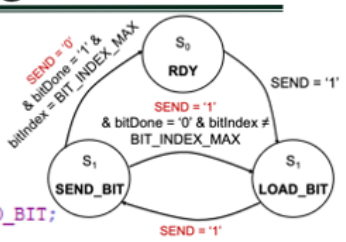
Architecture: Signals, Type, Constants

```
50  architecture Behavioral of UART_TX_CTRL is
51
52  type TX_STATE_TYPE is (RDY, LOAD_BIT, SEND_BIT);
53
54  constant BIT_TMR_MAX : std_logic_vector(13 downto 0) := "10100010110000"; --10416 = (round(100MHz / 9600)) - 1
55  constant BIT_INDEX_MAX : natural := 10;
56
57  --Counter that keeps track of the number of clock cycles the current bit has been held stable over the
58  --UART TX line. It is used to signal when the next bit should be transmitted
59  signal bitTmr : std_logic_vector(13 downto 0) := (others => '0');
60
61  --combinatorial logic that goes high when bitTmr has counted to the proper value to ensure
62  --a 9600 baud rate
63  signal bitDone : std_logic;
64
65  --Contains the index of the next bit in txData that needs to be transferred
66  signal bitIndex : natural;
67
68  --a register that holds the current data being sent over the UART TX line
69  signal txBit : std_logic := '1';
70
71  --A register that contains the whole data packet to be sent, including start and stop bits.
72  signal txData : std_logic_vector(9 downto 0);
73
74  signal txState : TX_STATE_TYPE := RDY;
75
```

register 2 中的 signal

1 Process for Next State Logic

```
76  begin
77  --Next state logic
78  next_txState_process : process (CLK)
79  begin
80          if (rising_edge(CLK)) then
81                  case txState is
82                  when RDY =>
83                          if (SEND = '1') then
84                                  txState <= LOAD_BIT;
85                          end if;
86                  when LOAD_BIT =>
87                          txState <= SEND_BIT;
88                  when SEND_BIT =>
89                          if (bitDone = '1') then
90                                  if (bitIndex = BIT_INDEX_MAX) then
91                                          txState <= RDY;
92                                  else
93                                          txState <= LOAD_BIT;
94                                  end if;
95                          end if;
96                  when others=> --should never be reached
97                          txState <= RDY;
98                  end case;
99          end if;
100 end process;
101
```



University of BRISTOL

1 Process for bitTmr

```
102     bit_timing_process : process (CLK)
103     begin
104             if (rising_edge(CLK)) then
105                     if (txState = RDY) then
106                             bitTmr <= (others => '0');
107                     else
108                             if (bitDone = '1') then
109                                     bitTmr <= (others => '0');
110                             else
111                                     bitTmr <= bitTmr + 1;
112                             end if;
113                     end if;
114             end if;
115     end process;
116
117     bitDone <= '1' when (bitTmr = BIT_TMR_MAX) else
118                                     '0';
```

1 Process for bitIndex

```
119
120   bit_counting_process : process (CLK)
121   begin
122           if (rising_edge(CLK)) then
123                   if (txState = RDY) then
124                           bitIndex <= 0;
125                   elsif (txState = LOAD_BIT) then
126                           bitIndex <= bitIndex + 1;
127                   end if;
128           end if;
129   end process;
130
```

1 Process for Data Latch (Register 1)

```
131     tx_data_latch_process : process (CLK)
132     begin
133             if (rising_edge(CLK)) then
134                     if (SEND = '1') then
135                             txData <= '1' & DATA & '0';
136                     end if;
137             end if;
138     end process;
139
```

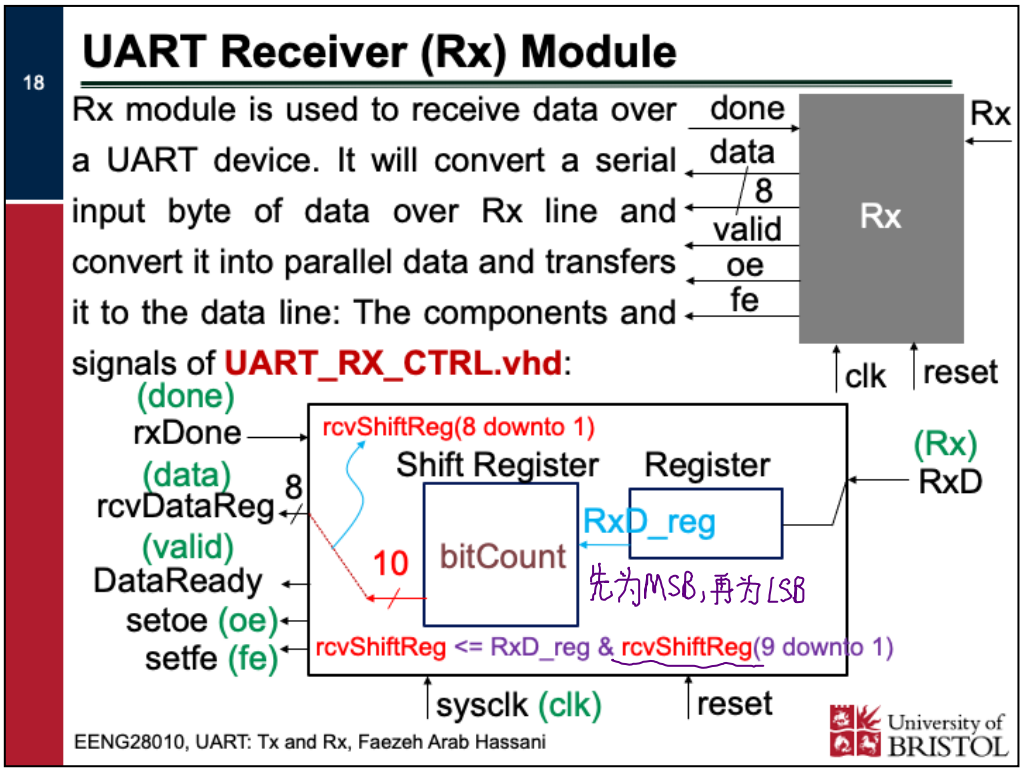
1 Process for Register 2

```
140   tx_bit_process : process (CLK)
141   begin
142           if (rising_edge(CLK)) then
143                   if (txState = RDY) then
144                           txBit <= '1';
145                   elsif (txState = LOAD_BIT) then
146                           txBit <= txData(bitIndex);
147                   end if;
148           end if;
149   end process;
150
151   UART_TX <= txBit;
152   READY <= '1' when (txState = RDY) else
153                           '0';
154
155   end Behavioral;
156
```

already finish the 8 bit data

Outline

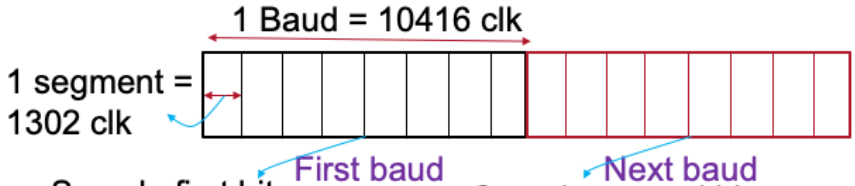
- UART
- UART Transmitter (Tx) Module
- UART Receiver (Rx) Module



- A separate register ensures no changes in output
- Frame error (fe) if start bit is not '0' and stop bit not '1'
- Overrun error (oe) if previous data has not been read yet (i.e. rxDone ≠ '1')

Bit Sampling at Middle of Baud Rate

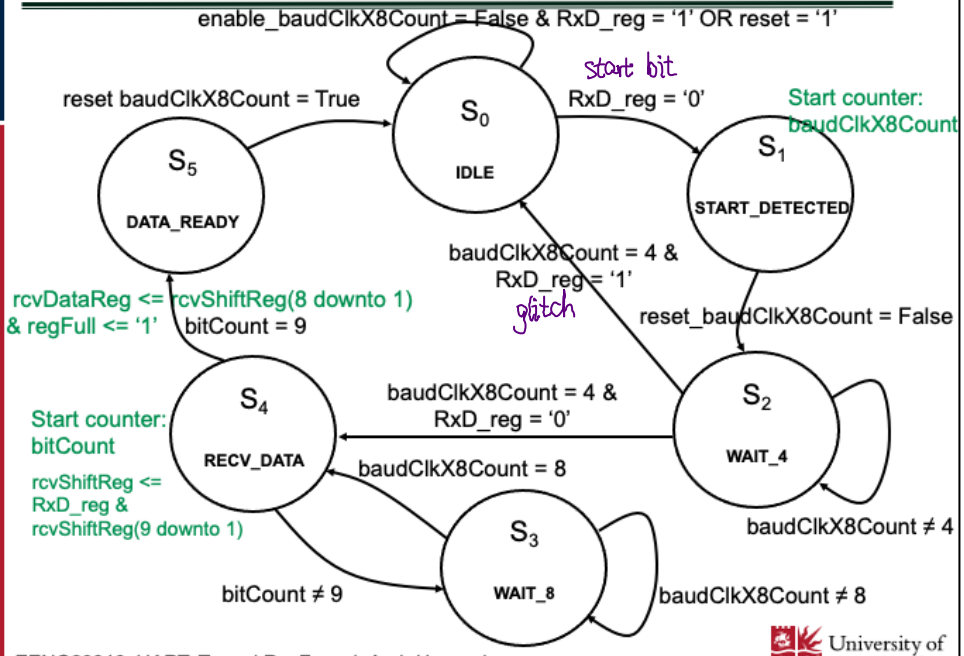
- For Rx, we would like to sample each of 10 bits at the middle of each baud.
- We divide each baud to 8 segments and each segment has 10416/8 = 1302 clk => We need to have a counter (baudClkX8Count) to count the number of segments as well as a counter (bitTmr) to count 1302 clk (BAUDX8_TMR_MAX = 1302)
- We use an enable_baudClkX8Segment to stop the counting when the serial input is quiet, and a reset baudClkX8Count to reset the counter



The Other Counter

- We need to have a counter named as bitCount to count the number of bits received by shift register

State Diagram for Rx Module



Entity

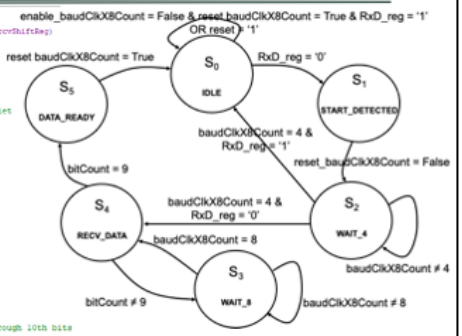
```
66  library ieee;
67  use ieee.std_logic_1164.all;
68  use ieee.std_logic_unsigned.all;
69
70  entity UART_RX_CTRL is
71    port(
72      RxD: in std_logic;                    -- serial data in
73           sysclk: in std_logic;                 -- system clock
74           reset: in std_logic;             --      synchronous reset
75           rxDone: in std_logic;            -- data succesfully read (active high)
76           rcvDataReg: out std_logic_vector(7 downto 0); -- received data
77           dataReady: out std_logic;        -- data ready to be read
78           setOE: out std_logic;            -- overrun error (active high)
79           setFE: out std_logic             -- frame error (active high)
80    );
81  end UART_RX_CTRL;
82
83
```

Architecture: Signals, Constant, Types

```
architecture RCVR of UART_RX_CTRL is

        -- constant to generate 8x oversampling clock
        -- baudClk_max =(round(100MHz / 9600)) - 1 = 10416
        -- baudClk_x8_max =10416/8=1302
        constant BAUDX8_TMR_MAX: integer := 1302;

        type state_type is (IDLE, START_DETECTED, WAIT_4, WAIT_8, RECV_DATA, DATA_READY);
        type logical_type is (TRUE, FALSE);

        signal currentState, nextState: state_type;
        signal RxD_reg : std_logic;
        signal rcvShiftReg: std_logic_vector (9 downto 0); -- receive shift register to receive all bits, including stop and start
        signal baudClkX8Count : integer range 0 to 8; -- indicates when to read the RxD input
        signal bitCount : integer range 0 to 10; -- counts number of bits read
        signal bitTmr : integer range 0 to BAUDX8_TMR_MAX; -- counts system clock cycles to assert baudClkX8Count
        signal enable_baudClkX8Count : logical_type; -- no need to count when serial input is quiet
        signal enable_bitCount, reset_baudClkX8Count, shiftDataIn : logical_type;
        signal regFull : std_logic;

```

1 Process for Next State

```
begin
  receiver_nextState: process(currentState, baudClkX8Count, bitCount, RxD_reg, regFull, rxDone, rcvShiftReg)
  begin
    -- assign defaults at the beginning to avoid assigning in every branch
    setFE <= '0'; setOE <= '1';
        enable_baudClkX8Count <= TRUE;
        reset_baudClkX8Count <= FALSE;
    case currentState is
      when IDLE =>
        enable_baudClkX8Count <= FALSE; -- disable counting to save power when serial line is quiet
        reset_baudClkX8Count <= TRUE;
        if RxD_reg = '0' then -- start bit is a '0'
          nextState <= START_DETECTED;
        else
          nextState <= IDLE;
        end if;
      when START_DETECTED =>
        reset_baudClkX8Count <= FALSE;
        nextState <= WAIT_4;
      when WAIT_4 => -- wait for 4 baud "segments" to sample in the middle of the first bit
        reset_baudClkX8Count <= FALSE;
        if baudClkX8Count = 4 then -- in the middle of the bit cycle
          if (RxD_reg = '0') then -- no glitch
            nextState <= RECV_DATA;
          else
            nextState <= IDLE;
          end if;
        else
                                   nextState <= WAIT_4;
        end if;
      when WAIT_8 => -- wait for 8 baud "segments" to sample in the middle of the 2nd through 10th bits
        reset_baudClkX8Count <= FALSE;
        if baudClkX8Count = 8 then -- in the middle of the bit cycle
          nextState <= RECV_DATA;
        else
          nextState <= WAIT_8;
        end if;
      when RECV_DATA =>
        reset_baudClkX8Count <= TRUE;
        if bitCount = 9 then -- all 10 bits have been read (1 start + 8 data + 1 stop)
          nextState <= DATA_READY;
        else -- some more bits to be read yet
          nextState <= WAIT_8;
        end if;
      when DATA_READY => -- all 10 bits have been read, and data is valid
        reset_baudClkX8Count <= TRUE;
        nextState <= IDLE;
        --  frame error if stop bit not 1 or start bit not 0
        if rcvShiftReg(9) /= '1' or rcvShiftReg(0) /= '0' then
          setFE <= '1';
        end if;
        --  overrun error if previous data has not been read yet
        if  rxDone /= '1' then
          setOE <= '1';
        end if;
      when OTHERS => -- should never be reached
        nextState <= IDLE;
    end case;
  end process;
```



University of BRISTOL

1 Process for State Register

```
166     stateRegister:          process (sysclk)
167     begin
168                     if rising_edge (sysclk) then
169                             if (reset = '1') then
170                                     currentState <= IDLE;
171                             else
172                                     currentState <= nextState;
173                             end if;
174                     end if;
175             end process;
176
```

1 Process for bitTmr and baudClkX8Count

```
177     -- divide a baud (bit) cycle into 8 segments
178     bit_timing_process : process (sysclk)
179     begin
180      if rising_edge(sysclk) then
181                  if reset ='1' then
182                          bitTmr <= 0;
183                          baudClkX8Count <= 0;
184                  else
185                          if enable_baudClkX8Count = TRUE then -- detecting serial data
186                                  if bitTmr = BAUDX8_TMR_MAX then -- completed 1 baud "segment"
187                                          bitTmr <= 0;
188                                          baudClkX8Count <= baudClkX8Count + 1;
189                                  else -- not completed a baud "segment"
190                                          bitTmr <= bitTmr + 1;
191                                  end if;
192                                  if reset_baudClkX8Count = TRUE or baudClkX8Count = 9 then -- reset both
193                                          bitTmr <= 0;
194                                          baudClkX8Count <= 0;
195                                  end if;
196                          end if;
197                  end if;
198          end if;
199     end process;
200
```

数 number of clock

1 Process for bitCount

```
200
201     -- count number of bits that have been sampled
202     bit_counting_process : process (sysclk)
203     begin
204           if rising_edge(sysclk) then
205                   if reset ='1' then
206                           bitCount <= 0;
207                   else
208                           if currentState = RECV_DATA then
209                                   bitCount <= bitCount + 1;
210                           elsif currentState = IDLE then
211                                   bitCount <= 0;
212                           end if;
213                   end if;
214           end if;
215     end process;
216
```

1 Process for Data Shift (Shift Register)

```
217     -- shift data in serially when ready to be sampled
218     dataShift:    process (sysclk)
219     begin
220      if rising_edge(sysclk) then
221                   if reset ='1' then
222                           rcvShiftReg <= (others => '1');
223                   else
224                           if currentState = RECV_DATA then
225                                   rcvShiftReg <= RxD_reg & rcvShiftReg(9 downto 1);
226                           end if;
227                   end if;
228           end if;
229     end process;
230
```

1 Process for Data Latch

```
231   -- latch the data when valid
232   -- A separate set of registers ensures no changes in output
233   -- bus until all the bits are valid
234   dataLatch:    process (sysclk)
235   begin
236     if rising_edge(sysclk) then
237                 if reset ='1' then
238                         rcvDataReg <= (others => '1');
239                         regFull <= '0';
240                 else
241                         if currentState = DATA_READY then -- update output lines, signal data is valid
242                                 rcvDataReg <= rcvShiftReg(8 downto 1);
243                                 regFull <= '1';
244                         elsif rxDone = '1' then  -- ok to clear register
245                                 regFull <= '0';
246                         end if;
247       end if;
248     end if;
249   end process;
250
```

1 Process for Register

```
251     -- shift data in serially when ready to be sampled
252     RxD_register:    process (sysclk)
253     begin
254      if rising_edge(sysclk) then
255           if reset ='1' then
256               RxD_reg <= '1';
257           else
258               RxD_reg <= RxD;
259           end if;
260       end if;
261     end process;
262
263     dataReady <= regFull;
264
265   end RCVR;
266
```