```
M2.M2_SRAM.memory[3] = 8'b0101_00_11;      // Read 131 to R3
M2.M2_SRAM.memory[4] = 131;
M2.M2_SRAM.memory[5] = 8'b0101_00_01;      // Read 128 to R1
M2.M2_SRAM.memory[6] = 128;
M2.M2_SRAM.memory[7] = 8'b0101_00_00;      // Read 129 to R0
M2.M2_SRAM.memory[8] = 129;

M2.M2_SRAM.memory[9] = 8'b0010_00_01;      // Sub R1-R0 to R1

M2.M2_SRAM.memory[10] = 8'b1000_00_00;     // BRZ
M2.M2_SRAM.memory[11] = 134;               // Holds address for BRZ

M2.M2_SRAM.memory[12] = 8'b0001_10_11;     // Add R2+R3 to R3
M2.M2_SRAM.memory[13] = 8'b0111_00_11;     // BR
M2.M2_SRAM.memory[14] = 140;
// Load data
M2.M2_SRAM.memory[128] = 6;
M2.M2_SRAM.memory[129] = 1;
M2.M2_SRAM.memory[130] = 2;
M2.M2_SRAM.memory[131] = 0;
M2.M2_SRAM.memory[134] = 139;
//M2.M2_SRAM.memory[135] = 0;
M2.M2_SRAM.memory[139] = 8'b1111_00_00;    // HALT
M2.M2_SRAM.memory[140] = 9;                //  Recycle
end
endmodule
```

7.4 Design Example: UART

Systems that exchange information and interact via serial data channels use modems as interfaces between the host machines/devices and the channel, as shown in Figure 7-14. For example, a modem allows a computer to connect to a telephone line and communicate with a receiving computer through its modem [2, 5]. The host machine stores information in a parallel word format, but transmits and receives data in a serial, single-bit, format. A modem is also called a UART, or *universal asynchronous receiver and*

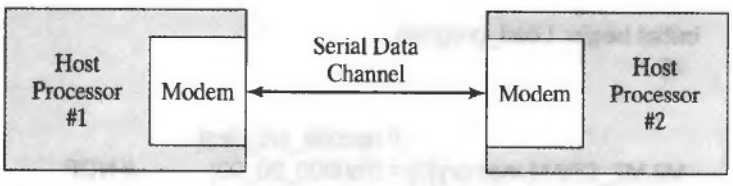


FIGURE 7-14 Processor/modem communication over a serial channel.

Stop Bit	Parity Bit	Data Bit 7	Data Bit 6	Data Bit 5	Data Bit 4	Data Bit 3	Data Bit 2	Data Bit 1	Data Bit 0	Start Bit

FIGURE 7-15 Data format for ASCII text transmitted by a UART.

transmitter, indicating that the device has the capability to both receive and transmit serial data. This design example will address the modeling and synthesis of a UART's transmitter and receiver.

For this discussion, a UART exchanges text data in an American Standard Code for Information Interchange (ASCII) format in which each alphabetical character is encoded by 7 bits and augmented by a parity bit that can be used for error detection. For transmission, the modem wraps this 8-bit subword with a *start-bit* in the least significant bit (LSB), and a *stop-bit* in the most significant bit (MSB), resulting in the 10-bit word format shown in Figure 7-15. The first 9 data bits are transmitted in sequence, beginning with the start-bit, with each bit being asserted at the serial line for one cycle of the modem clock. The stop-bit may assert for more than one clock.

7.4.1 UART Operation

The UART transmitter is always part of larger environment in which a host processor controls transmission by fetching a data word in parallel format and directing the UART to transmit it in a serial format. Likewise, the receiver must detect transmission, receive the data in serial format, strip off the start- and stop-bits, and store the data word in a parallel format. The receiver's job is more complex, because the clock used to send the inbound data is not available at the remote receiver. The receiver must regenerate the clock locally, using the receiving machine's clock rather than the clock of the transmitting machine.

The simplified architecture of a UART presented in Figure 7-16 shows the signals used by a host processor to control the UART and to move data to and from a data bus in the host machine. Details of the host machine are not shown.

7.4.2 UART Transmitter

The input–output signals of the transmitter are shown in the high-level block diagram in Figure 7-17. The input signals are provided by the host processor, and the output signals control the movement of data in the UART. The architecture of the transmitter will consist of a controller, a data register (*XMT_datareg*), a data shift register (*XMT_shftreg*), and a status register (*bit_count*) to count the bits that are transmitted. The status register will be included with the datapath unit.

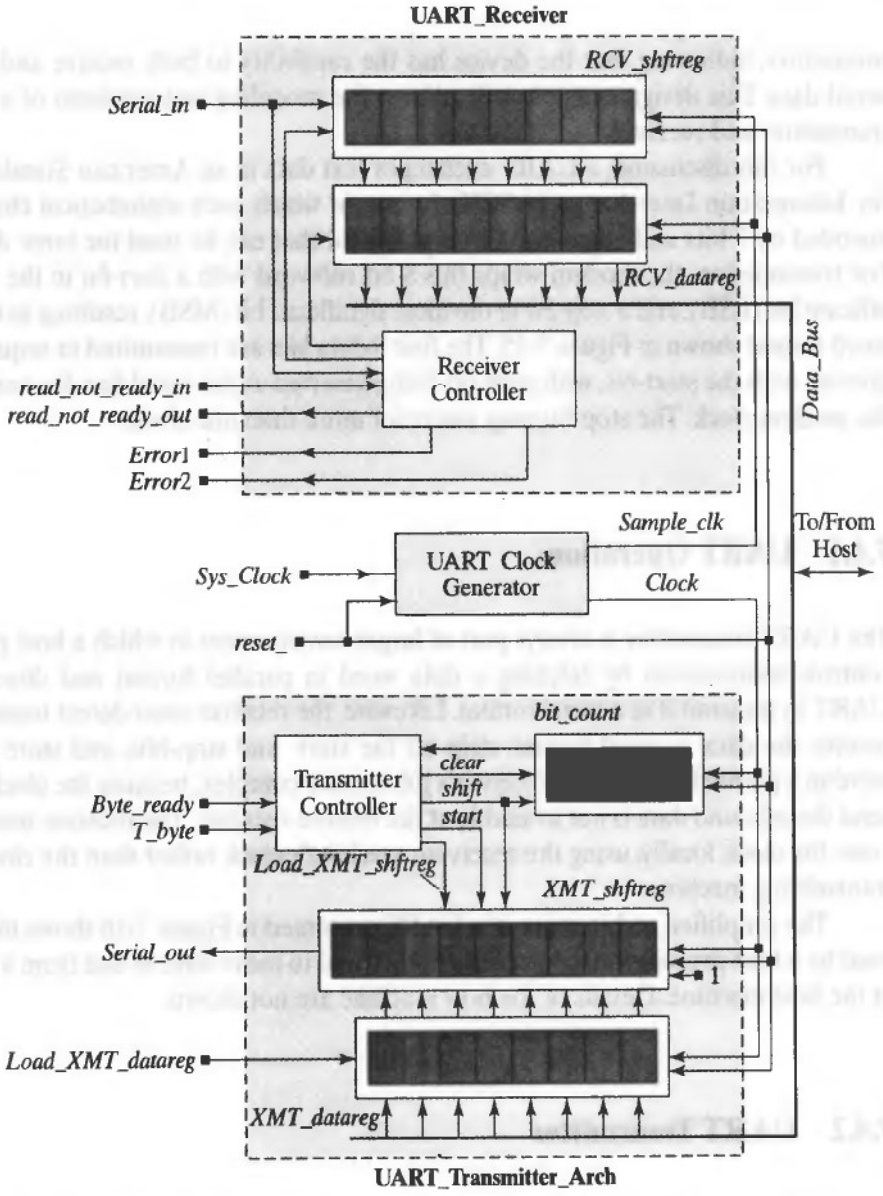


FIGURE 7-16 Block diagram of a UART.

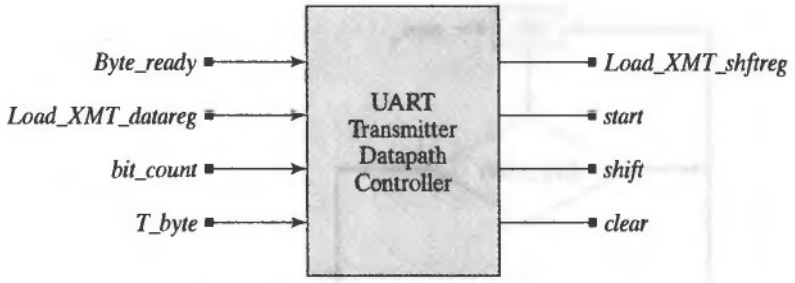


FIGURE 7-17 Interface signals of a state machine controller for a UART transmitter.

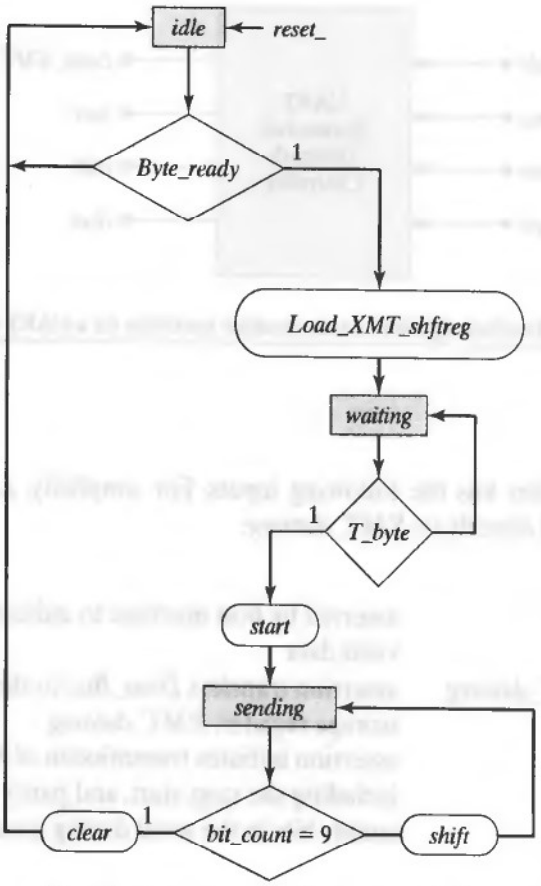
The controller has the following inputs. For simplicity, *Load_XMT_datareg* is shown connected directly to *XMT_datareg*:

Byte_ready	asserted by host machine to indicate that *Data_Bus* has valid data
Load_XMT_datareg	assertion transfers *Data_Bus* to the transmitter data storage register, *XMT_datareg*
T_byte	assertion initiates transmission of a byte of data, including the stop, start, and parity bits
bit_count	counts bits in the word during transmission

The state machine of the controller forms the following output signals that control the datapath of the transmitter:

Load_XMT_shftreg	assertion loads the contents of *XMT_data_reg* into *XMT_shftreg*
start	signals the start of transmission
shift	directs *XMT_shftreg* to shift by one bit towards the LSB and to backfill with a stop bit (1).
clear	clears *bit_count*

The ASM chart of the state machine controlling the transmitter is shown in Figure 7.18. The machine has three states: *idle*, *waiting*, and *sending*. When *reset_* is asserted, the machine asynchronously enters *idle*, *bit_count* is flushed, *XMT_shftreg* is loaded with 1s, and the control signals *clear*, *Load_XMT_shftreg*, *shift*, and *start* are driven to 0. In *idle*, if an active edge of *Clock* occurs while *Load_XMT_data_reg* is asserted by the external host, the contents of *Data_Bus* will transfer to *XMT_data_reg*. (This action is not part of the ASM chart because it occurs independently of the state of the machine.) The machine remains in *idle* until *start* is asserted.



Note: Only the branch corresponding to a true decision is annotated at a decision diamond; signals that are not shown explicitly asserted are de-asserted. Conditional assertions are indicated by the name of the asserted signal

FIGURE 7-18 ASM chart for the state machine controller for the UART transmitter.

When *Byte_ready* is asserted, *Load_XMT_shftreg* is asserted and *next_state* is driven to *waiting*. The assertion of *Load_XMT_shftreg* indicates that *XMT_datareg* now contains data that can be transferred to the internal shift register. At the next active edge of *Clock*, with *Load_XMT_shftreg* asserted, three activities occur: (1) *state* transfers from *idle* to *waiting*, (2) the contents of *XMT_datareg* are loaded into the leftmost bits of *XMT_shftreg*, a (*word_size* + 1)-bit shift register whose LSB signals the start and stop of transmission, and (3) the LSB of *XMT_shftreg* is reloaded with 1, the stop-bit. The machine remains in *waiting* until the external processor asserts *T_byte*.

At the next active edge of *Clock*, with *T_byte* asserted, *state* enters *sending*, and the LSB of *XMT_shftreg* is set to 0 to signal the start of transmission. At the same time,

shift is driven to 1, and *next_state* retains the state code corresponding to *sending*. At subsequent active edges of *Clock*, with *shift* asserted, *state* remains in *sending* and the contents of *XMT_shftreg* are shifted toward the LSB, which drives the external serial channel. As the data shifts occur, 1s are back-filled in *XMT_shftreg*, and *bit_count* is incremented. With *state* in *sending, shift* asserts while *bit_count* is less than 9. The machine increments *bit_count* after each movement of data, and when *bit_count* reaches 9 *clear* asserts, indicating that all of the bits of the augmented word have been shifted to the serial output. At the next active edge of *Clock*, the machine returns to *idle*.

The control signals produced by the state machine induce state-dependent register transfers in the data path. The activity of the primary inputs (*Byte_ready, Load_XMT_datareg*, and *T_byte*), and the signals from the controller (*Load_XMT_shftreg, start, shift, clear*) are shown in Figure 7-19, along with the movement of data in the datapath registers. The contents of the registers are shown at successive edges of clock, with a time axis going from the top of the figure toward the bottom. Transitions of the active edge of *clock* occur between the successive rows displaying contents of *XMT_datareg*. The bits of the transmitted signal are shown in the sequence in which they are transmitted, with the rightmost cell of *XMT_shftreg* holding the bit that is transmitted at the serial interface at each step. The state of the machine is shown, and the state transitions and datapath register transitions that occur on the rising edges of *clock* are shown in the register boxes. The values of the control signals that cause the register transitions are also shown. The displayed values of the control signals are those they held immediately before the active edge of *Clock*; and which cause the register transfers that are shown. The sequence of output bits is also shown, with 1s being pushed into the MSB of *XMT_shftreg* under the action of *shift*. The sequence of output bits of the transmitted signal are shown as a word at each time step, with the understanding that *the LSB of the word is the first bit that was transmitted*, and the MSB of the word is the most recent bit that was transmitted at the serial interface.

The Verilog description, *UART_Transmitter_Arch*, of the architecture for the transmitter has three cyclic behaviors—a level-sensitive behavior describing the combinational logic for next state and outputs of the controller, an edge-sensitive behavior to synchronize the state transitions of the controller, and another edge-sensitive behavior to synchronize the register transfers of the datapath registers. For simplicity, we include the entire description in a single Verilog module, rather than impose architectural boundaries around the datapath and the controller (see Figure 7-20). The module can be partitioned and synthesized into the individual functional units.

Some simulation results are shown in Figure 7-21 and Figure 7-22 for an 8-bit data word. The waveforms produced by the simulator have been annotated to indicate significant features of the transmitter's behavior. First, observe the values of the signals immediately after *reset_* is asserted. The state is *idle*. Note that *Data_Bus* initially contains the value $a7_h$ (1010_0111_2), a value specified by the testbench used for simulation. With *Byte_ready* not yet asserted, and with *Load_XMT_datareg* asserted, the *Data_Bus* is loaded into *XMT_datareg*. The machine remains in *idle* until *Byte_ready* is asserted. When *Byte_ready* asserts, then *Load_XMT_shftreg* asserts. This causes the

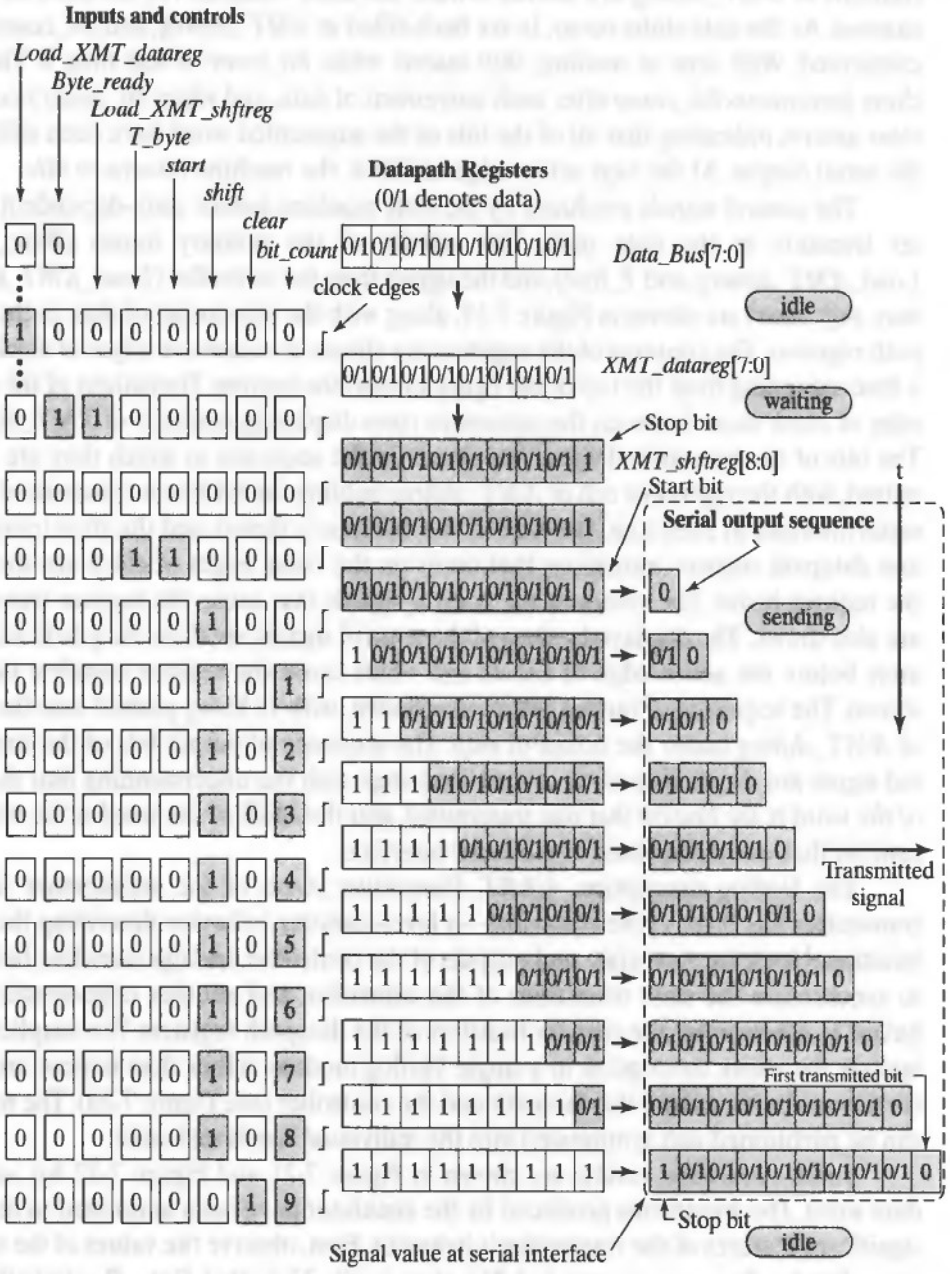


FIGURE 7-19 Control signals and dataflow in an 8-bit UART transmitter.

```
module UART_Transmitter_Arch
  (Serial_out, Data_Bus, Byte_ready, Load_XMT_datareg, T_byte, Clock, reset_);
  parameter       word_size = 8;                  // Size of data word, e.g., 8 bits
  parameter       one_hot_count = 3;              // Number of one-hot states
  parameter       state_count = one_hot_count;    // Number of bits in state register
  parameter       size_bit_count = 3;             // Size of the bit counter, e.g., 4
                                                  // Must count to word_size + 1
  parameter       idle = 3'b001;                  // one-hot state encoding
  parameter       waiting = 3'b010;
  parameter       sending = 3'b100;
  parameter       all_ones = 9'b1_1111_1111;      // Word + 1 extra bit

  output                      Serial_out          // Serial output to data channel
  input  [word_size - 1:0]    Data_Bus;           // Host data bus containing data word
  input                       Byte_ready;         // Used by host to signal ready
  input                       Load_XMT_datareg;   // Used by host to load the data register
  input                       T_byte;             // Used by host to signal the start of transmission
  input                       Clock;              // Bit clock of the transmitter
  input                       reset_;             // Resets internal registers, loads the
                                                  // XMT_shftreg with ones

  reg  [word_size - 1:0]      XMT_datareg;        // Transmit Data Register
  reg  [word_size:0]          XMT_shftreg;        // Transmit Shift Register: {data, start bit}
  reg                         Load_XMT_shftreg;   // Flag to load the XMT_shftreg
  reg  [state_count - 1:0]    state, next_state;  // State machine controller
  reg  [size_bit_count:0]     bit_count;          // Counts the bits that are transmitted
  reg                         clear;              // Clears bit_count after last bit is sent
  reg                         shift;              // Causes shift of data in XMT_shftreg
  reg                         start;              // Signals start of transmission

  assign Serial_out = XMT_shftreg[0];             // LSB of shift register

  always @ (state or Byte_ready or bit_count or T_byte) begin: Output_and_next_state
    Load_XMT_shftreg = 0;
    clear = 0;
    shift = 0;
    start = 0;
    next_state = state;
    case (state)
      idle:           if (Byte_ready == 1) begin
                        Load_XMT_shftreg = 1;
                        next_state = waiting;
                      end

      waiting:        if (T_byte == 1) begin
                        start = 1;
                        next_state = sending;
                      end

      sending:        if (bit_count != word_size + 1)
                        shift = 1;
                      else begin
                        clear = 1;
                        next_state = idle;
                      end

      default:        next_state = idle;
    endcase
  end
```

FIGURE 7-20 Verilog Description of the UART transmitter.

```
always @ (posedge Clock or negedge reset_) begin: State_Transitions
  if (reset_ == 0) state <= idle; else state <= next_state; end

always @ (posedge Clock or negedge reset_) begin: Register_Transfers
  if (reset_== 0) begin
    XMT_shftreg <= all_ones;
    bit_count <= 0;
  end
  else begin
    if (Load_XMT_datareg == 1)
      XMT_datareg <= Data_Bus;                    // Get the data bus

    if (Load_XMT_shftreg == 1)
      XMT_shftreg <= {XMT_datareg, 1'b1};         // Load shift reg,
                                                  // insert stop bit
    if (start == 1)
      XMT_shftreg[0] <= 0;                        // Signal start of transmission

    if (clear == 1) bit_count <= 0;
    else if (shift == 1) bit_count <= bit_count + 1;

    if (shift == 1)
      XMT_shftreg <= {1'b1, XMT_shftreg[word_size:1]};  // Shift right, fill with 1's
    end
  end
endmodule
```

FIGURE 7-20 Continued

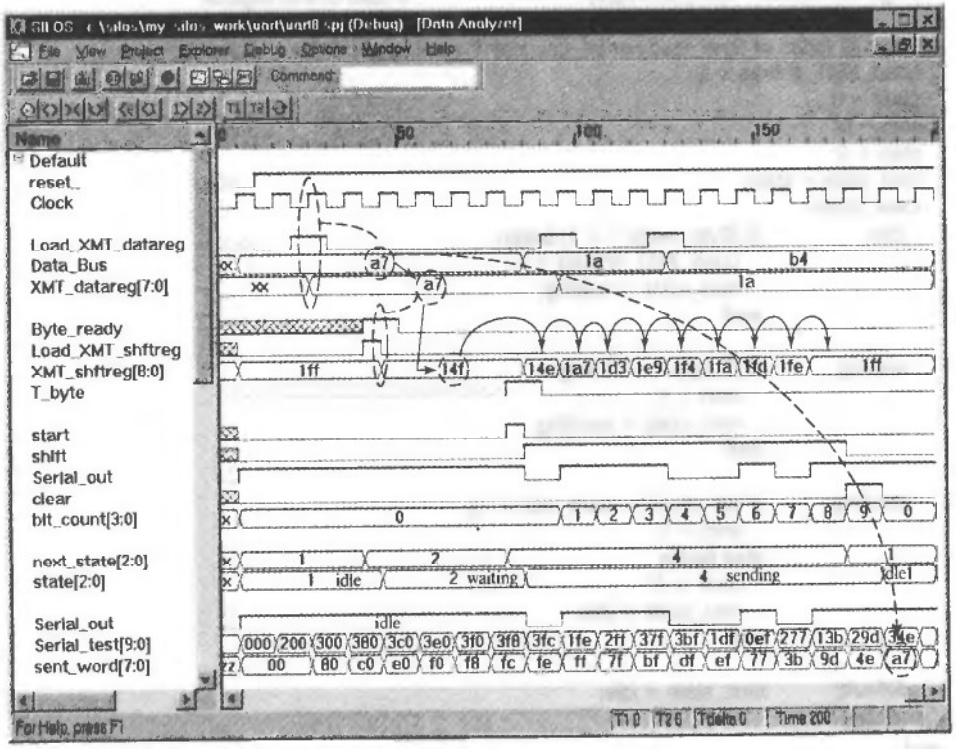


FIGURE 7-21 Annotated simulation results for the 8-bit UART transmitter.

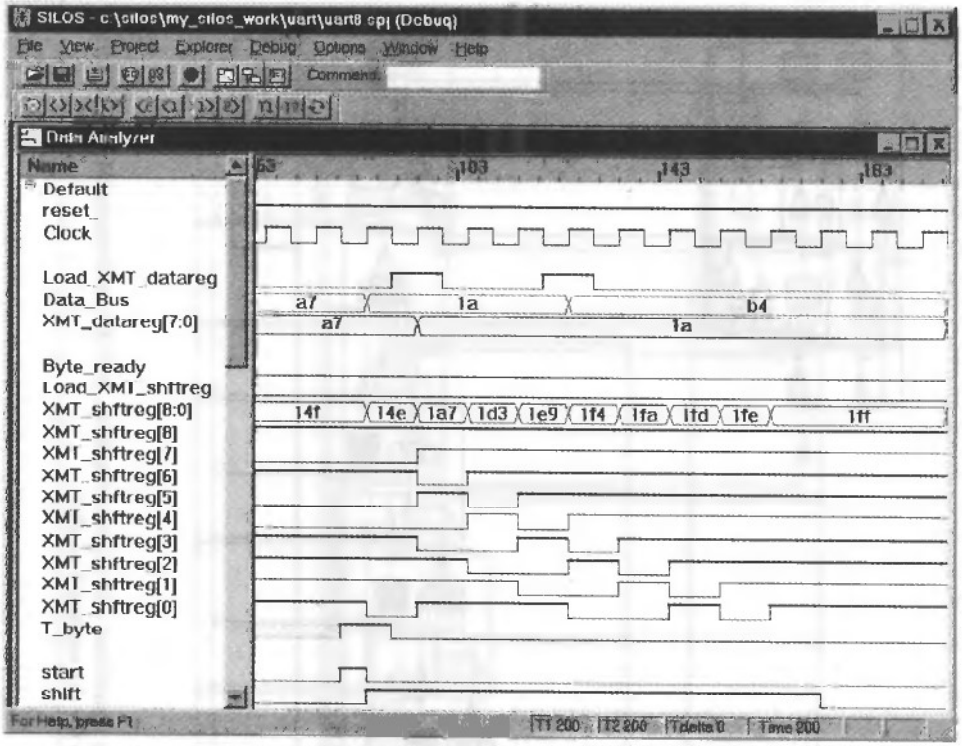


FIGURE 7-22 Data movement through *XMT_shftreg*.

state to change to *waiting* at the next active edge of *Clock*. The 9-bit *XMT_shftreg* is now loaded with the value $\{a7_h, 1\} = 1\_0100\_1111_2 = 14f_h$. Note that the LSB of *XMT_shftreg* is loaded with a 1. The machine remains in *waiting* until *T_byte* is asserted. The assertion of *T_byte* asserts *start*. The machine enters *sending* at the active edge of *Clock* immediately after the host processor's assertion of *T_byte*, and the LSB of *XMT_shftreg* is loaded with a 0. The 9-bit word in *XMT_shftreg* becomes $1\_0100\_1110_2 = 14e_h$. The 0 in the LSB signals the start of transmission. Figure 7-21 has been annotated to show the movement of data through *XMT_shftreg*. Note that 1s are filled behind, as the word shifts to the right. At the active edge of the clock after *bit_count* reaches 9 (for an 8-bit word), *clear* asserts, *bit_count* is flushed, and the machine returns to *idle*.

For diagnostic purposes, the testbench includes a 10-bit shift register that receives *Serial_out* (by hierarchical dereferencing). The eight innermost bits of this register are displayed in Figure 7-21 as *sent_word[7: 0]* (skipping the start-bit and the stop-bit) to reveal the correct transmission of data, $a7_h$. The bit sequence of *Serial_out* likewise has this value. This is evident at the active edge of the clock after the assertion of *clear*. The movement of data through *XMT_shftreg* is shown in Figure 7-22.

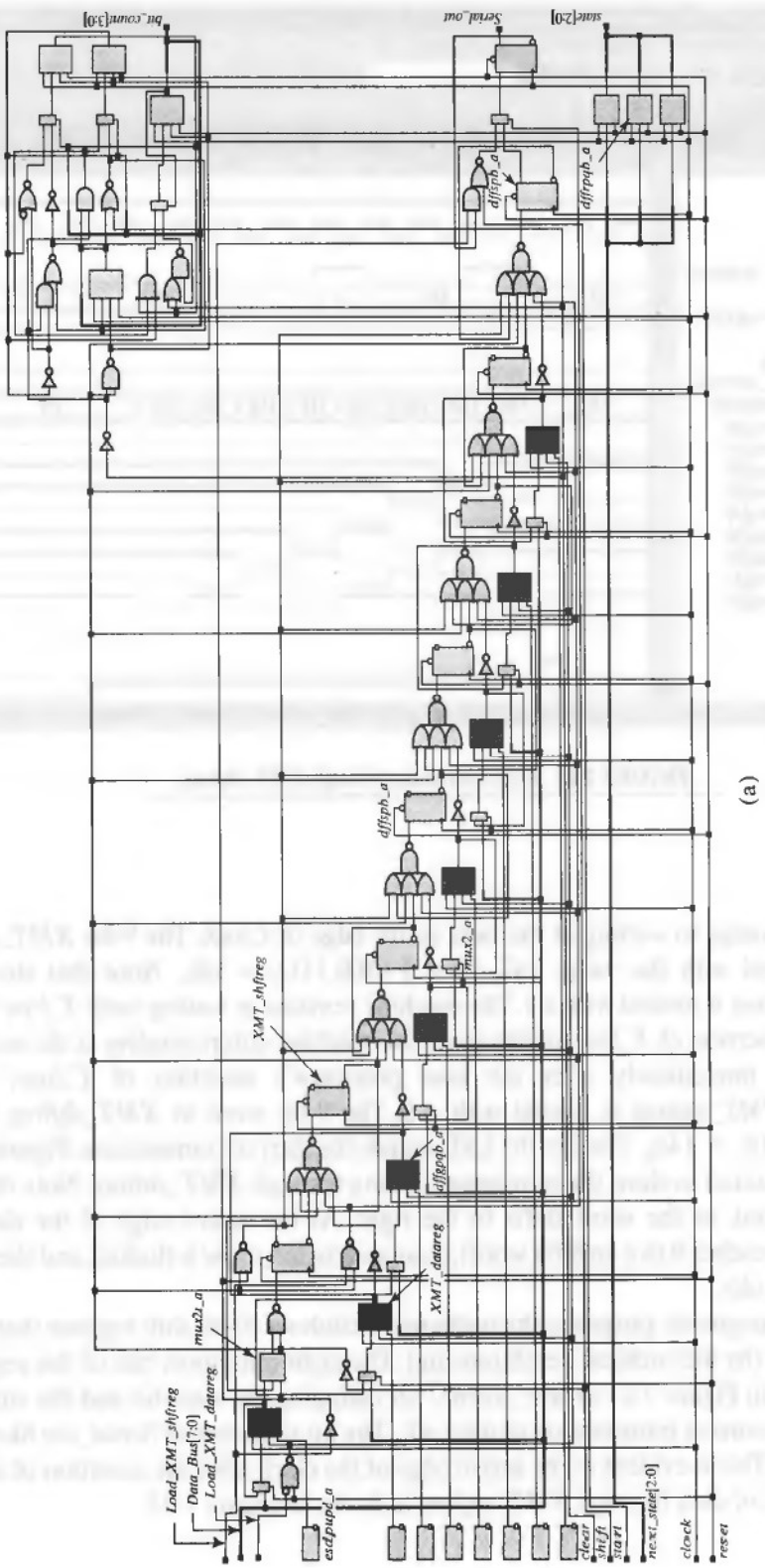
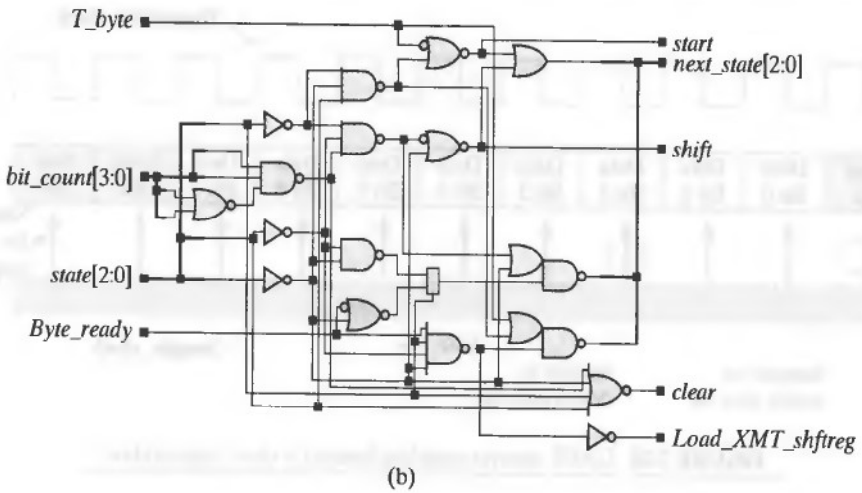


FIGURE 7-23 *UART_transmitter.* (a) logic synthesized to implement the state transitions and register transfers, and (b) combinational logic forming the next state and control signals for the register transfer.



(b)

FIGURE 7-23 Continued

The circuit synthesized from *UART_transmitter* is shown in Figure 7-23. The Verilog model synthesizes as a unit, but for illustration and discussion the description was partitioned and synthesized in two parts, one for the state transitions and register transfers, and another for the combinational logic forming the next state and the control signals for the register transfers. The part governing the state transitions and register transfers consists of an 8-bit register holding *XMT_datareg*, a 9-bit shift register holding *XMT_shftreg*, and a bit counter. The circuit uses *dffrgpqb_a*, a D-type flip-flop with a rising-edge clock, asynchronous active-low reset, and internal gated data of the external data or the output, and *dffspb_a*, a D-type flip-flop with rising-edge clock, and asynchronous active-low set. The shift registers have been highlighted in Figure 7-23.

7.4.3 UART Receiver

The UART receiver has the task of receiving the serial bit stream of data, removing the start-bit, and transferring the data in a parallel format to a storage register connected to the host data bus. The data arrives at a standard bit rate, but it is not necessarily synchronized with the internal clock at the host of the receiver, and the transmitter's clock is not available to the receiver. This issue of synchronization is resolved by generating a *local* clock at a higher frequency and using it to sample the received data in a manner that preserves the integrity of the data. In the scheme used here, the data, assumed to be in a 10-bit format, will be sampled at a rate determined by *Sample_clock*, which is

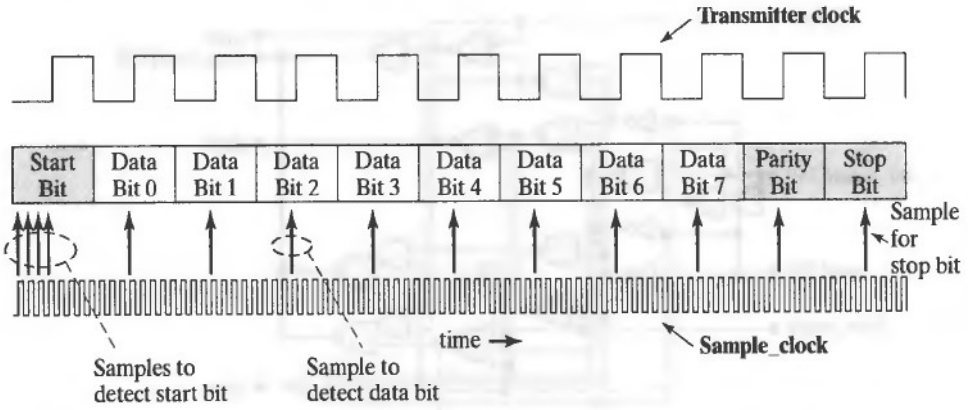


FIGURE 7-24 UART receiver sampling format for clock regeneration.

generated at the receiver's host. The cycles of *Sample_clock* will be counted to ensure that the data are sampled in the middle of a bit time, as shown in Figure 7-24. The sampling algorithm must (1) verify that a start bit has been received, (2) generate samples from 8 bits of the data, and (3) load the data onto the local bus.

Although a higher sampling frequency could be used, the frequency of *Sample_clock* in this example is 8 times the (known) frequency of the bit clock that transmitted the data. This ensures that a slight misalignment between the leading edge of a cycle of *Sample_clock* and the arrival of the start-bit will not compromise the sampling scheme, because the sample will still be taken within the interval of time corresponding to a transmitted bit. The arrival of a start-bit will be determined by successive samples of value 0 after the input data goes low. Then three additional samples will be taken to confirm that a valid start-bit has arrived. Thereafter, 8 successive bits will be sampled at approximately the center of their bit times. Under worst-case conditions of misalignment, the sample is taken a full cycle of *Sample_clock* ahead of the actual center of the bit time, which is a tolerable skew.

The high-level block diagram in Figure 7-25 shows the input–output signals of a state-machine controller that will interface with the host processor and direct the receiver's sampling scheme.

The state machine has the following inputs:

read_not_ready_in	signals that the host is not ready to receive data
Serial_in	serial bit stream received by the unit
*reset_*	active low reset
Sample_counter	counts the samples of a bit
Bit_counter	counts the bits that have been sampled

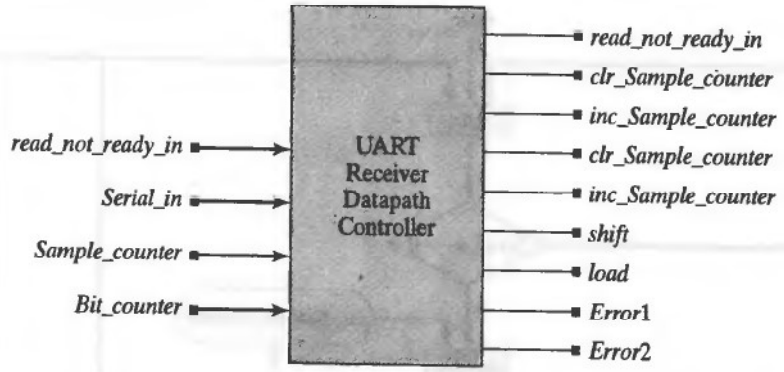


FIGURE 7-25 Interface signals of a state-machine controller for the UART receiver.

The state machine produces the following outputs:

read_not_ready_out	signals that the receiver has received 8 bits
inc_Sample_counter	increments *Sample_counter*
clr_Sample_counter	clears *Sample_counter*
inc_Bit_counter	increments *Bit_counter*
clr_Bit_counter	clears *Bit_counter*
shift	causes *RCV_shftreg* to shift towards the LSB
load	causes *RCV_shftreg* to transfer data to *RCV_datareg*
Error1	asserts if host is not ready to receive data after last bit has been sampled
Error2	asserts if the stop-bit is missing

The ASM chart of a state machine controller for the receiver is shown in Figure 7-26. The machine has three states: *idle, starting*, and *receiving*. Transitions between states are synchronized by *Sample_clk*. Assertion of an asynchronous active-low resct puts the machine in the *idle* state. It remains there until *Serial_in* is low, then makes a transition to *starting*. In *starting*, the machine samples *Serial_in* to determine whether the first bit is a valid start-bit (it must be 0). Depending on the sampled values, *inc_Sample_counter* and *clr_Sample_counter* may be asserted to increment or clear the counter at the next active edge of *Sample_clock*. If the next three samples of *Serial_in* are 0, the machine concludes that the start-bit is valid and goes to the state *receiving*. *Sample_counter* is cleared on the transition to *receiving*. In this state, eight successive samples are taken (one for each bit of the byte, at each active edge of *Sample_clk*), with *inc_Sample_counter* asserted. Then *Bit_counter* is incremented. If the sampled bit is not the last (parity) bit, *inc_Bit_counter* and *shift* are asserted. The assertion of *shift* will cause the sample value to be loaded into the MSB of *RCV_shftreg*, the receiver shift register, and will shift the 7 leftmost bits of the register toward the LSB.

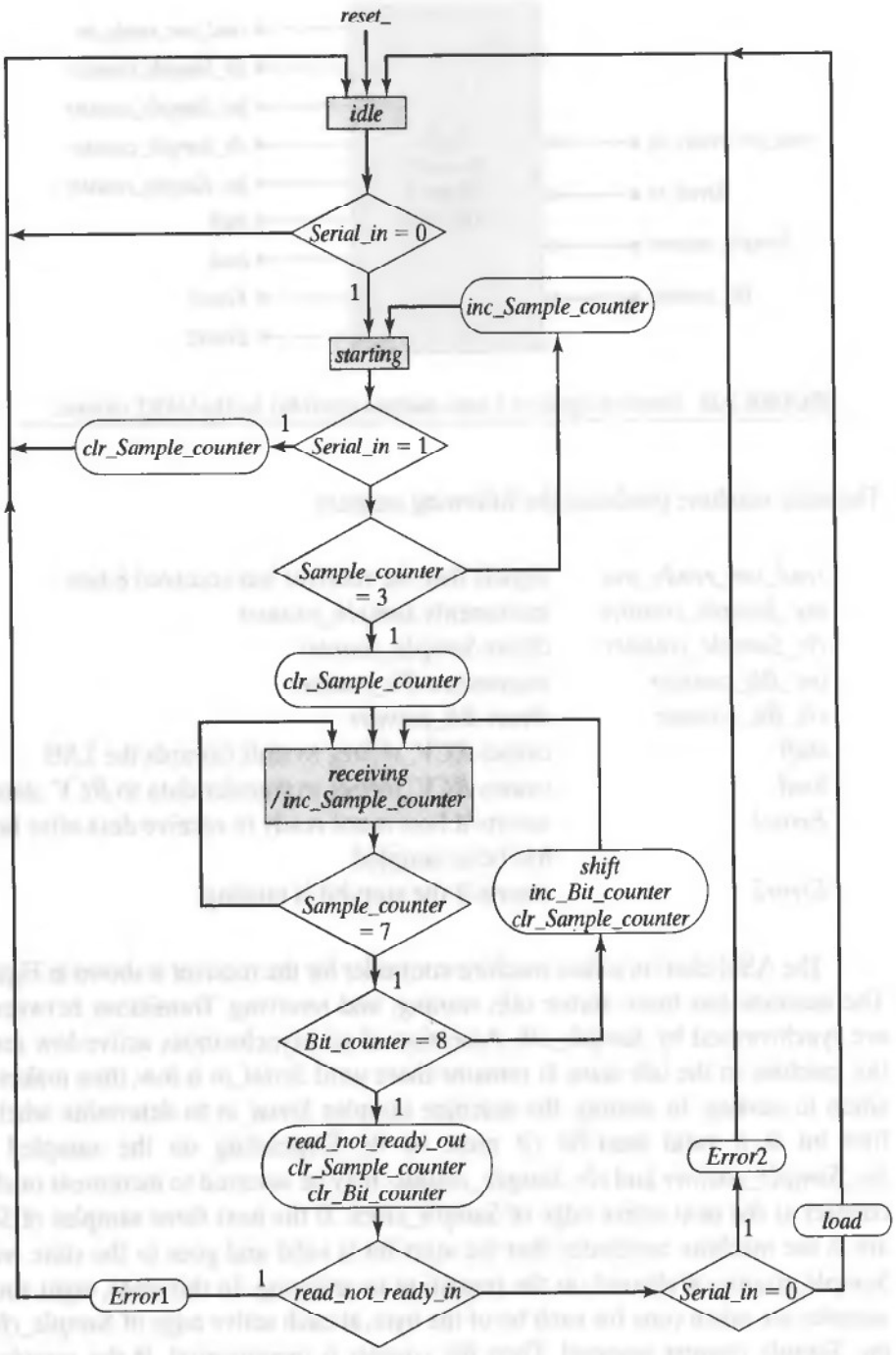


FIGURE 7-26 UART receiver ASM chart.

After the last bit has been sampled, the machine will assert *read_not_ready_out*, a handshake output signal to the processor, and clear the bit counter. At this time, the machine also checks the integrity of the data and the status of the host processor. If *read_not_ready_in* is asserted, the host processor is not ready to receive the data (*Error1*). If a stop-bit is not the next bit (detected by *Serial_in* = 0), there is an error in the format of the received data (*Error2*). Otherwise, *load* is asserted to cause the contents of the shift register to be transferred as a parallel word to *RCV_datareg*, a data register in the host machine, with a direct connection to *data_bus*.

The Verilog description of the 8-bit UART receiver is given in Figure 7-27. The description follows directly from the ASM chart in Figure 7-26.

The simulation results in Figure 7-28 are annotated to show functional features of the waveforms. The received data word is $b5_h$ = 1011_0101_2. The reception sequence is from LSB to MSB, and the data move through the inbound shift register from MSB to LSB. The data word is preceded by a start-bit and followed by a stop-bit. With *reset_* having a value of 0, the state is *idle* and the counters are cleared. At the first active edge of *Sample_clock* after the reset condition is de-asserted, with *Serial_in* having a value of 0, the controller's state enters *starting* to determine whether a start-bit is being received. Three more samples of *serial_in* are taken, and after a total of four samples have been found to be 0, the *Sample_counter* is cleared and the state enters *receiving*. After the eighth sample, *shift* is asserted. The sample at the next active edge of the clock is shifted into the MSB of *RCV_shftreg*. The value of *RCV_shftreg* becomes 80_h = 1000_0000_2. The sampling cycle repeats again, and a value of 0 is sampled and loaded into *RCV_shftreg*, changing the contents of the register to 0100_0000_2 = 40_h.

The end of the sampling cycle of the received word is shown in Figure 7-29. After the last data bit is sampled, the machine samples once more to detect the stop bit. In the absence of an error, the contents of *RCV_shftreg* will be loaded into *RCV_datareg*. In this example, the value $b5_h$ is finally loaded from *RCV_shftreg* into *RCV_datareg*. Other tests can be conducted to completely verify the functionality of the receiver.

The Verilog description of the partitioned receiver in Figure 7-30 synthesizes into the circuit shown in Figure 7-31. Although the entire description synthesizes as a single unit, the structure of the synthesized result is revealed more easily by partitioning the description into the asynchronous (combinational) and synchronous (state transition) parts shown below. Note that the ports of the parent modules of a partition must be sized properly to accommodate vector ports in the child modules. Otherwise, the ports will be treated as default scalars in the scope of the parent module.

The state-transition part of the synthesized circuit includes *RCV_shftreg*, the shift register receiving *Serial_in*, and *RCV_datareg* (the 8-bit register holding the received word), *Sample_counter, Bit_counter* (the 4-bit counter that determines when all of the bits have been received), and the state register for the machine. Two types of flip-flops are used: the four-input *dffrgpqb_a* and the three-input *dffrpqb_a*. The former is a D-type flip-flop with internal gated data between the external datapath and the output, a rising clock, and an asynchronous active-low reset; the latter is a D-type flip-flop with data, rising clock, and asynchronous active-low reset. Only the state register uses the *dffrpqb_a* flip-flop.

```
module UART8_Receiver
  (RCV_datareg, read_not_ready_out, Error1 ,Error2, Serial_in, read_not_ready_in, Sample_clk, reset_);
  // Sample_clk is 8x Bit_clk

  parameter   word_size          = 8;
  parameter   half_word          = word_size/2;
  parameter   Num_counter_bits   = 4;          // Must hold count of word_size
  parameter   Num_state_bits     = 2;          // Number of bits in state
  parameter   idle               = 2'b00;
  parameter   starting           = 2'b01;
  parameter   receiving          = 2'b10;

  output      [word_size - 1:0]  RCV_datareg;
  output                         read_not_ready_out,
                                 Error1, Error2;

  input       Serial_in,
              Sample_clk,
              reset_,
              read_not_ready_in,

  reg                            RCV_datareg;
  reg         [word_size - 1:0]  RCV_shftreg;
  reg         [Num_counter_bits - 1:0]  Sample_counter;
  reg         [Num_counter_bits:0]      Bit_counter;
  reg         [Num_state_bits - 1:0]    state, next_state;
  reg                            inc_Bit_counter, clr_Bit_counter;
  reg                            inc_Sample_counter, clr_Sample_counter;
  reg                            shift, load, read_not_ready_out;
  reg                            Error1, Error2;

//Combinational logic for next state and conditional outputs

always @ (state or Serial_in or read_not_ready_in or Sample_counter or Bit_counter) begin
  read_not_ready_out = 0;
  clr_Sample_counter = 0;
  clr_Bit_counter = 0;
  inc_Sample_counter = 0;
  inc_Bit_counter = 0;
  shift = 0;
  Error1 = 0;
  Error2 = 0;
  load = 0;
  next_state = state;

  case (state)
    idle:       if (Serial_in == 0) next_state = starting;

    starting:   if (Serial_in == 1) begin
                  next_state = idle;
                  clr_Sample_counter = 1;
                end else

                if (Sample_counter == half_word - 1) begin
                  next_state = receiving;
                  clr_Sample_counter = 1;
                end else inc_Sample_counter = 1;
```

FIGURE 7-27 Verilog description of UART8_Receiver, an 8-bit UART receiver.

```
receiving:      if (Sample_counter < word_ size − 1) inc_Sample_counter = 1;
                else begin
                  clr_Sample_counter = 1;
                  if (Bit_counter != word_size) begin
                    shift = 1;
                    inc_Bit_counter = 1;
                  end
                  else begin
                    next_state = idle;
                    read_not_ready_out = 1;
                    clr_Bit_counter = 1;
                    if (read_not_ready_in == 1) Error1 = 1;
                    else if (Serial_in == 0) Error2 = 1;
                    else load = 1;
                  end
                end
        default: next_state = idle;

  endcase
end

// state_transitions_and_register_transfers

  always @ (posedge Sample_clk) begin
    if (reset_ == 0) begin                    // synchronous reset_
      state <= idle;
      Sample_counter <= 0;
      Bit_counter <= 0;
      RCV_datareg <= 0;
      RCV_shftreg <= 0;
    end
    else begin
      state <= next_state;

      if (clr_Sample_counter == 1) Sample_counter <= 0;
      else if (inc_Sample_counter == 1) Sample_counter <= Sample_counter + 1;

      if (clr_Bit_counter == 1) Bit_counter <= 0;
      else if (inc_Bit_counter == 1) Bit_counter <= Bit_counter + 1;
      if (shift == 1 ) RCV_shftreg <= {Serial_in, RCV_shftreg[word_size − 1:1]};
      if (load == 1) RCV_datareg <= RCV_shftreg;
    end
  end
endmodule
```

FIGURE 7-27 Continued

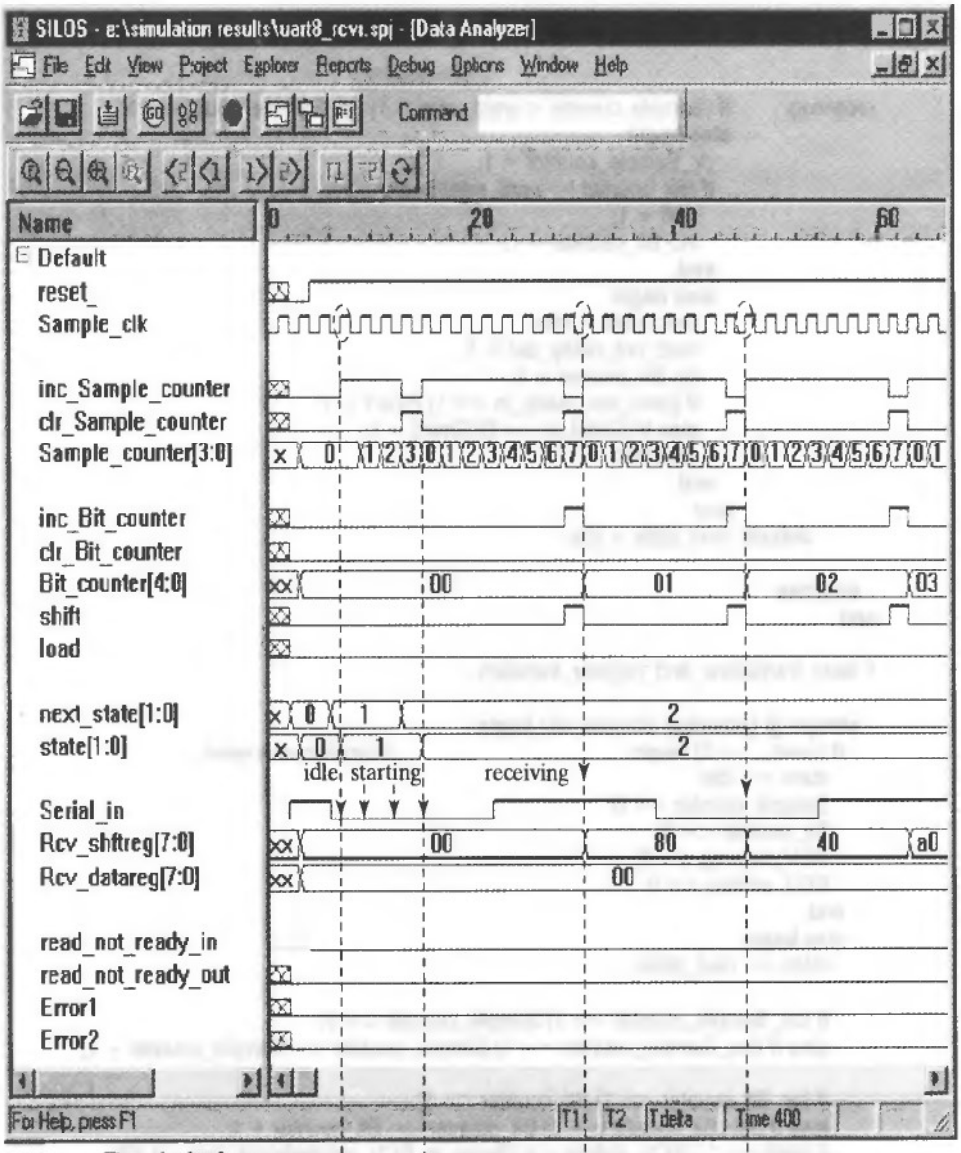


FIGURE 7-28 Annotated simulation results for the UART receiver.

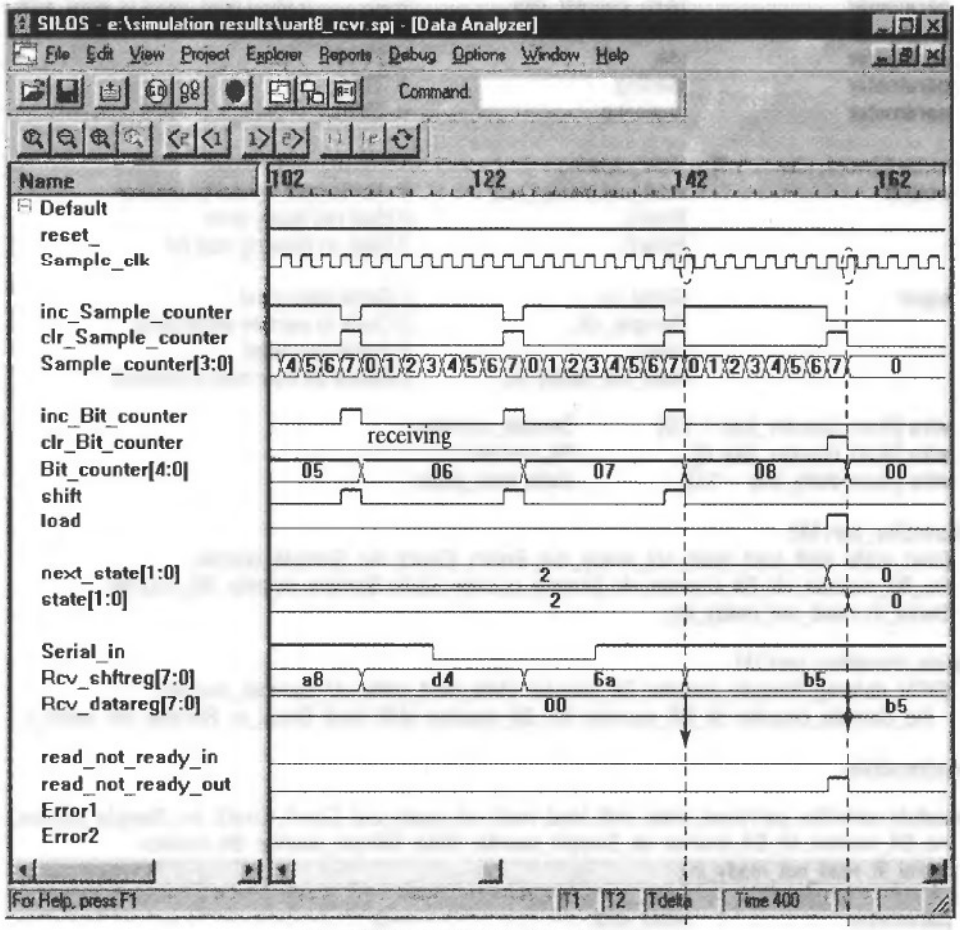


FIGURE 7-29 Transfer of data word into *RCV_datareg* at the end of sampling.

```
module UART8_rcvr_partition (RCV_datareg, read_not_ready_out, Error1, Error2, Serial_in,
  read_not_ready_in, Sample_clk, reset_);

  // partitioned UART receiver                    // Sample_clk is 8x Bit_clk

  parameter              word_size               = 8;
  parameter              half_word               = word_size / 2;
  parameter              Num_counter_bits        = 4;    // Must hold count of word_size
  parameter              Num_state_bits          = 2;    // Number of bits in state
  parameter              idle                    = 2'b00;
  parameter              starting                = 2'b01;
  parameter              receiving               = 2'b10;

  output [word_size - 1:0]  RCV_datareg;
  output                 read_not_ready_out,     // Handshake to host processor
                         Error1,                 // Host not ready error
                         Error2;                 // Data_in missing stop bit

  input                  Serial_in,              // Serial data input
                         Sample_clk,             // Clock to sample serial data
                         reset_,                 // Active-low reset
                         read_not_ready_in;      // Status bit from host processor

  wire [Num_counter_bits - 1:0]     Sample_counter;
  wire [Num_counter_bits: 0]        Bit_counter;
  wire [Num_state_bits - 1:0]       state, next_state;

controller_part M2
  (next_state, shift, load, read_not_ready_out, Error1, Error2, inc_Sample_counter,
  inc_Bit_counter, clr_Bit_counter, clr_Sample_counter, state, Sample_counter, Bit_counter,
  Serial_in, read_not_ready_in);

state_transition_part M1
  (RCV_datareg, Sample_counter, Bit_counter, state, next_state, clr_Sample_counter,
    inc_Sample_counter, clr_Bit_counter, inc_Bit_counter, shift, load, Serial_in, Sample_clk, reset_);

endmodule

module controller_part (next_state, shift, load, read_not_ready_out, Error1, Error2, inc_Sample_counter,
  inc_Bit_counter, clr_Bit_counter, clr_Sample_counter, state, Sample_counter, Bit_counter,
  Serial_in, read_not_ready_in);

  parameter              word_size               = 8;
  parameter              half_word               = word_size / 2;
  parameter              Num_counter_bits        = 4;    // Must hold count of word_size
  parameter              Num_state_bits          = 2;    // Number of bits in state
  parameter              idle                    = 2'b00;
  parameter              starting                = 2'b01;
  parameter              receiving               = 2'b10;
```

FIGURE 7-30 Verilog description of *UART8_rcvr_partition*, an 8-bit UART receiver partitioned into a controller and a datapath.

```
output [Num_state_bits - 1:0]   next_state;
output                          shift, load, inc_Sample_counter;
output                          inc_Bit_counter, clr_Bit_counter, clr_Sample_counter;
output                          read_not_ready_out, Error1, Error2;

input [Num_state_bits - 1:0]    state;
input [Num_counter_bits - 1:0]  Sample_counter;
input [Num_counter_bits: 0]     Bit_counter;
input                           Serial_in, read_not_ready_in;

reg next_state;
reg inc_Sample_counter, inc_Bit_counter, clr_Bit_counter, clr_Sample_counter;
reg shift, load,  read_not_ready_out, Error1, Error2;

always @ (state or Serial_in or read_not_ready_in or Sample_counter or Bit_counter) begin
   read_not_ready_out = 0;        //Combinational logic for next state and conditional outputs
   clr_Sample_counter = 0;
   clr_Bit_counter = 0;
   inc_Sample_counter = 0;
   inc_Bit_counter = 0;
   shift = 0;
   Error1 = 0;
   Error2 = 0;
   load = 0;
   next_state = state;

   case (state)
    idle:          if (Serial_in == 0) next_state = starting;

    starting:      if (Serial_in == 1) begin
                     next_state = idle;
                     clr_Sample_counter = 1;
                   end else

                   if (Sample_counter == half_word - 1) begin
                     next_state = receiving;
                     clr_Sample_counter = 1;
                   end else inc_Sample_counter = 1;

    receiving:     if (Sample_counter < word_size - 1) inc_Sample_counter = 1;
                   else begin
                     clr_Sample_counter = 1;
                     if (Bit_counter != word_size) begin
                       shift = 1;
                       inc_Bit_counter = 1;
                     end
                     else begin
                       next_state = idle;
                       read_not_ready_out = 1;
                       clr_Bit_counter = 1;
                       if (read_not_ready_in == 1) Error1 = 1;
                       else if (Serial_in == 0) Error2 = 1;
                       else load = 1;
                     end
                   end
    default:       next_state = idle;

   endcase
 end
endmodule
```

FIGURE 7-30 Continued

```
module state_transition_part (RCV_datareg, Sample_counter, Bit_counter, state, next_state,
clr_Sample_counter, inc_Sample_counter, clr_Bit_counter, inc_Bit_counter, shift, load, Serial_in,
Sample_clk, reset_);
  parameter                word_size          = 8;
  parameter                half_word          = word_size / 2;
  parameter                Num_counter_bits   = 4;      // Must hold count of word_size
  parameter                Num_state_bits     = 2;      // Number of bits in state
  parameter                idle               = 2'b00;
  parameter                starting = 2'b01;
  parameter                receiving          = 2'b10;

  output [word_size - 1:0]          RCV_datareg;
  output [Num_counter_bits - 1:0]   Sample_counter;
  output [Num_counter_bits: 0]      Bit_counter;
  output [Num_state_bits - 1:0]     state;

  Input [Num_state_bits - 1:0]      next_stage;
  input                             Serial_in;
  input                             inc_Sample_counter, inc_Bit_counter;
  input                             clr_Bit_counter, clr_Sample_counter, shift, load;
  input                             Sample_clk, reset_;

  reg                               Sample_counter, Bit_counter;
  reg [word_size - 1:0]             RCV_shftreg, RCV_datareg;
  reg                               state;

// state_transitions_and_datapath_register_transfers

  always @ (posedge Sample_clk) begin
    if (reset_ == 0) begin                  // synchronous reset_
      state <= idle;
      Sample_counter <= 0;
      Bit_counter <= 0;
      RCV_datareg <= 0;
      RCV_shftreg <= 0;
    end
    else begin
      state <= next_state;

      if (clr_Sample_counter == 1) Sample_counter <= 0;
      else if (inc_Sample_counter == 1) Sample_counter <= Sample_counter + 1;

      if (clr_Bit_counter == 1) Bit_counter <= 0;
      else if (inc_Bit_counter == 1) Bit_counter <= Bit_counter + 1;
      if (shift == 1) RCV_shftreg <= {Serial_in, RCV_shftreg[word_size - 1:1]};
      if (load == 1) RCV_datareg <= RCV_shftreg;
    end
  end
endmodule
```

FIGURE 7-30 Continued

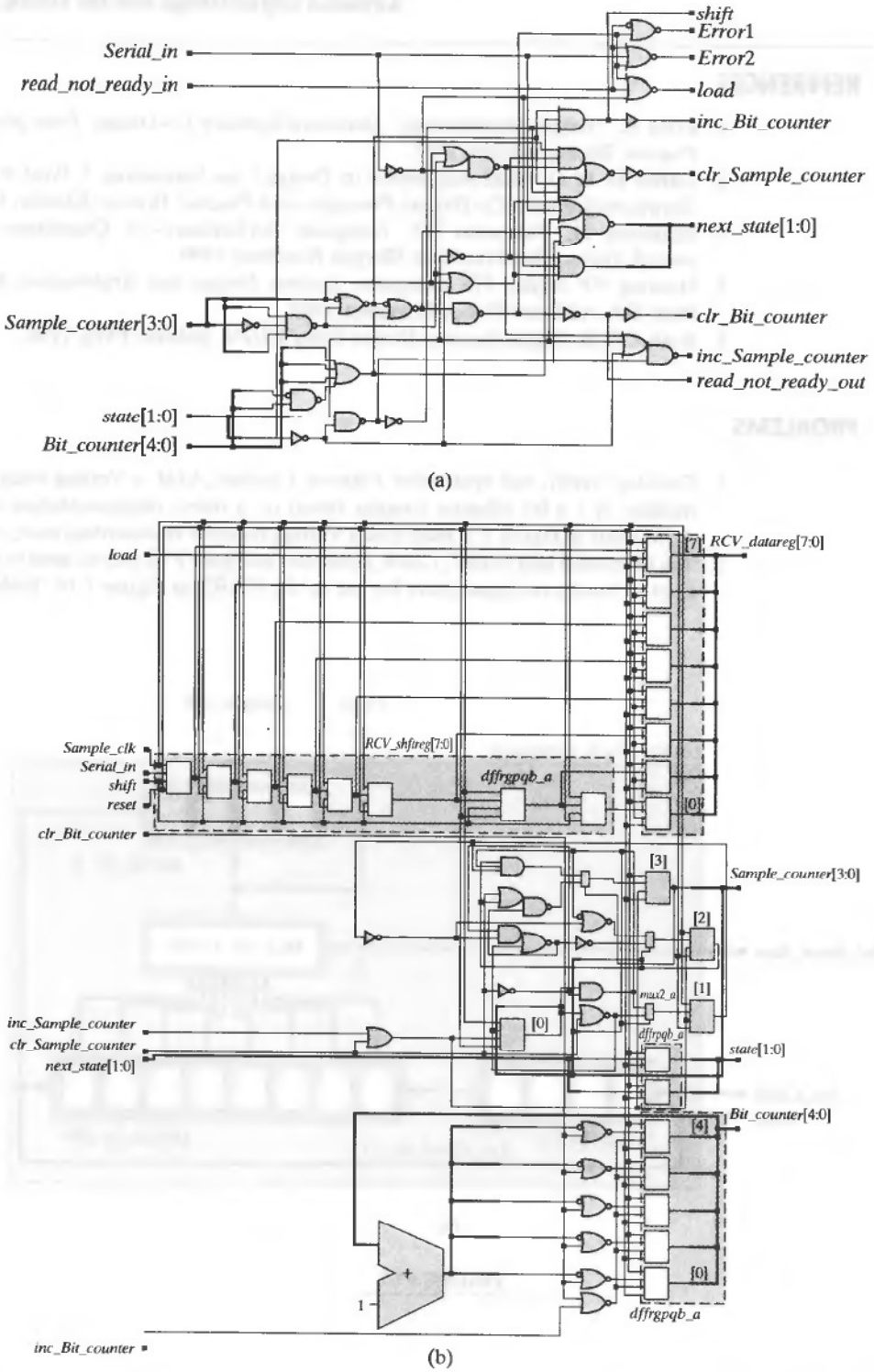


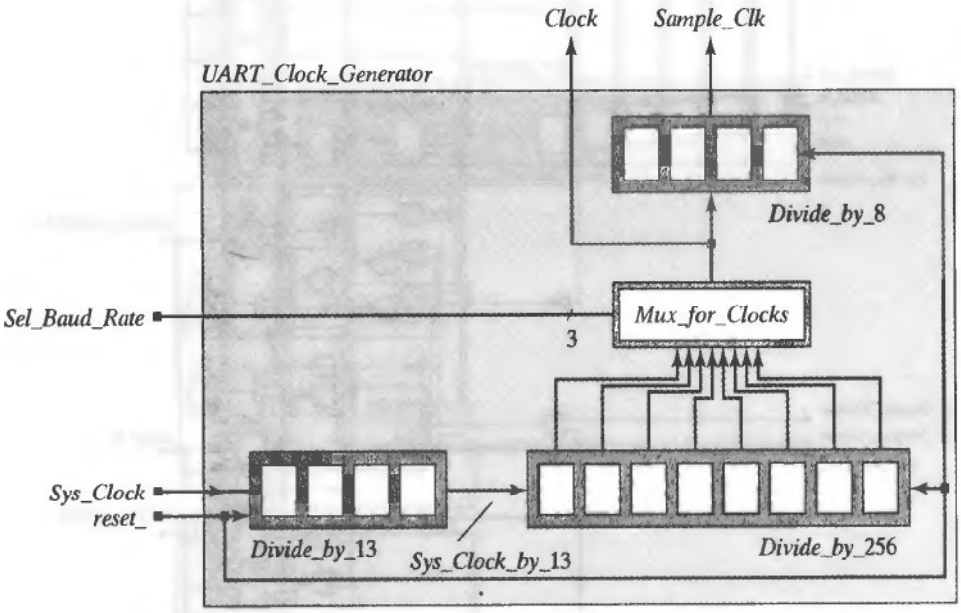
FIGURE 7-31 Circuits synthesized from *UART8_receiver*. (a) state transition and register transfer logic, and (b) combinational logic (forming the next state), output register, and control signals for register transfers.

REFERENCES

1. Ernst R. "Target Architectures." *Hardware/Software Co-Design: Principles and Practice*. Boston: Kluwer, 1997.
2. Gajski D, et al. "Essential Issues in Design," In: Staunstrup J, Wolf W, eds. *Hardware/Software Co-Design: Principles and Practice*. Boston: Kluwer, 1997.
3. Hennessy JL, Patterson DA. *Computer Architecture—A Quantitative Approach*. 2nd ed. San Francisco: Morgan Kaufman, 1996.
4. Heuring VP, Jordan HF. *Computer Systems Design and Architecture*. Menlo Park, CA: Addison-Wesley Longman, 1997.
5. Roth CW, Jr. *Digital Systems Design Using VHDL*. Boston: PWS, 1998.

PROBLEMS

1. Develop, verify, and synthesize *Johnson_Counter_ASM*, a Verilog behavioral module of a 4-bit Johnson counter based on a direct implementation of the ASM chart in Figure 7.5. *Hint*: Use a Verilog function to described *next_count*.
2. The functional unit *UART_Clock_Generator* in Figure P7-2 can be used to create a set of baud rate signal pairs for use in the UART in Figure 7-16. Table P7.2



(a)

FIGURE P7-2