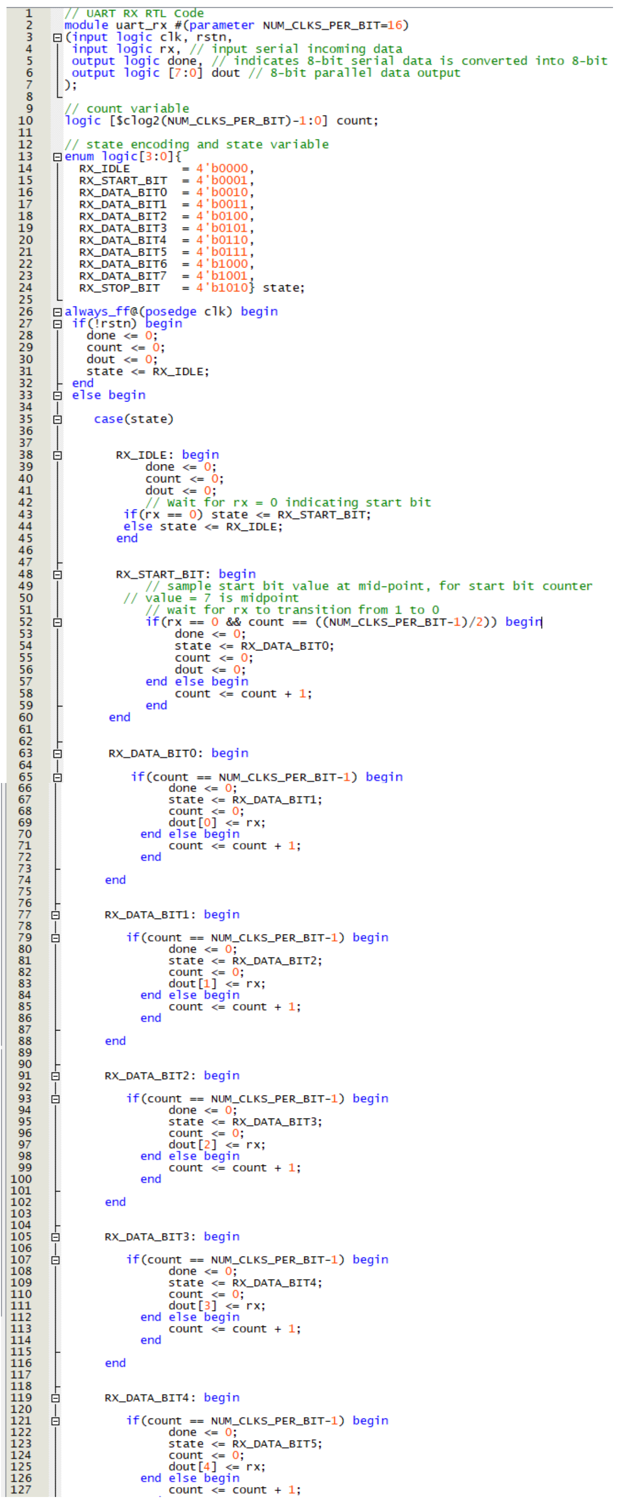
ECE 111 Winter 2022

HW7

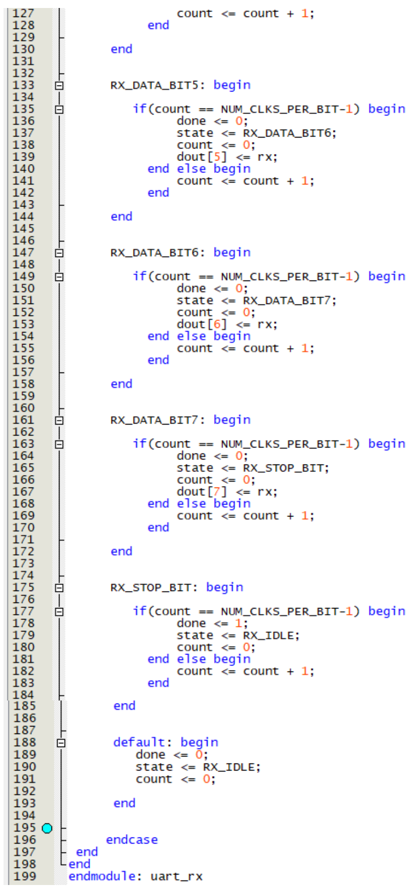
Hao Le A15547504

UART Receiver

Code

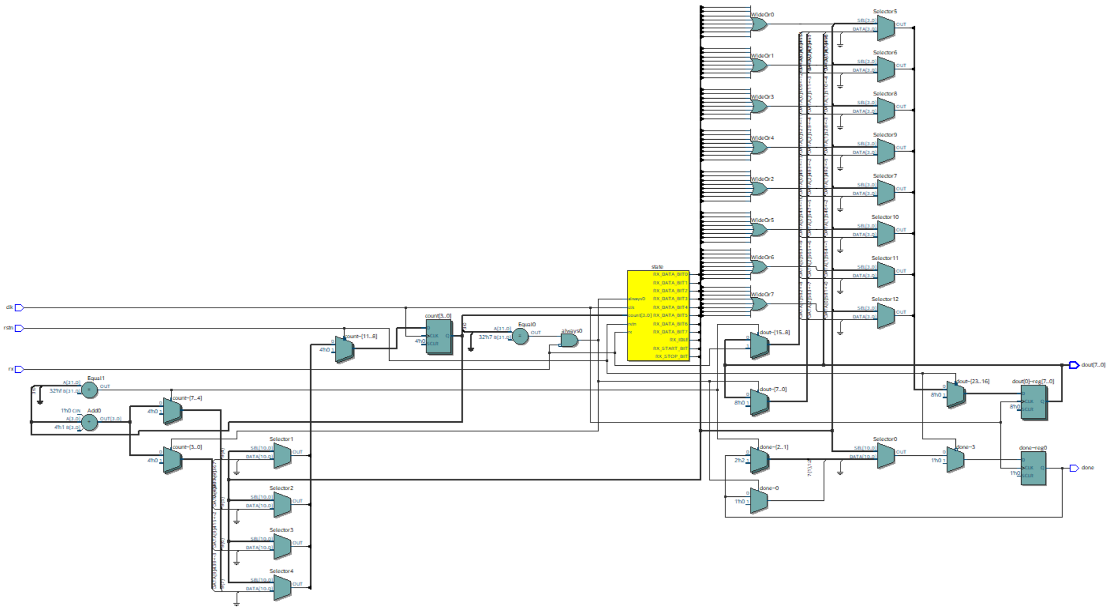


```
// UART RX RTL Code
module uart_rx #(parameter NUM_CLKS_PER_BIT=16)
(input logic clk, rstn,
 input logic rx, // input serial incoming data
 output logic done, // indicates 8-bit serial data is converted into 8-bit
 output logic [7:0] dout // 8-bit parallel data output
);

// count variable
logic [$clog2(NUM_CLKS_PER_BIT)-1:0] count;

// state encoding and state variable
enum logic[3:0]{
  RX_IDLE         = 4'b0000,
  RX_START_BIT    = 4'b0001,
  RX_DATA_BIT0    = 4'b0010,
  RX_DATA_BIT1    = 4'b0011,
  RX_DATA_BIT2    = 4'b0100,
  RX_DATA_BIT3    = 4'b0101,
  RX_DATA_BIT4    = 4'b0110,
  RX_DATA_BIT5    = 4'b0111,
  RX_DATA_BIT6    = 4'b1000,
  RX_DATA_BIT7    = 4'b1001,
  RX_STOP_BIT     = 4'b1010} state;

always_ff@(posedge clk) begin
 if(!rstn) begin
   done <= 0;
   count <= 0;
   dout <= 0;
   state <= RX_IDLE;
 end
 else begin

    case(state)


       RX_IDLE: begin
           done <= 0;
           count <= 0;
           dout <= 0;
           // Wait for rx = 0 indicating start bit
        if(rx == 0) state <= RX_START_BIT;
        else state <= RX_IDLE;
       end


       RX_START_BIT: begin
           // sample start bit value at mid-point, for start bit counter
        // value = 7 is midpoint
           // wait for rx to transition from 1 to 0
           if(rx == 0 && count == ((NUM_CLKS_PER_BIT-1)/2)) begin
               done <= 0;
               state <= RX_DATA_BIT0;
               count <= 0;
               dout <= 0;
           end else begin
               count <= count + 1;
           end
      end


      RX_DATA_BIT0: begin

         if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 0;
              state <= RX_DATA_BIT1;
              count <= 0;
              dout[0] <= rx;
          end else begin
              count <= count + 1;
          end

     end


     RX_DATA_BIT1: begin

        if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 0;
              state <= RX_DATA_BIT2;
              count <= 0;
              dout[1] <= rx;
          end else begin
              count <= count + 1;
          end

     end


     RX_DATA_BIT2: begin

        if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 0;
              state <= RX_DATA_BIT3;
              count <= 0;
              dout[2] <= rx;
          end else begin
              count <= count + 1;
          end

     end


     RX_DATA_BIT3: begin

        if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 0;
              state <= RX_DATA_BIT4;
              count <= 0;
              dout[3] <= rx;
          end else begin
              count <= count + 1;
          end

     end


     RX_DATA_BIT4: begin

        if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 0;
              state <= RX_DATA_BIT5;
              count <= 0;
              dout[4] <= rx;
          end else begin
              count <= count + 1;
```

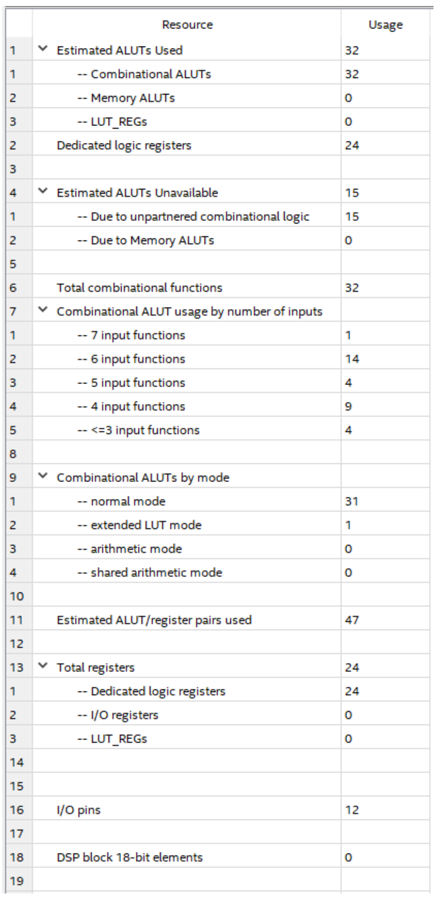


```
          end

     end


     RX_DATA_BIT5: begin

        if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 0;
              state <= RX_DATA_BIT6;
              count <= 0;
              dout[5] <= rx;
          end else begin
              count <= count + 1;
          end

     end


     RX_DATA_BIT6: begin

        if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 0;
              state <= RX_DATA_BIT7;
              count <= 0;
              dout[6] <= rx;
          end else begin
              count <= count + 1;
          end

     end


     RX_DATA_BIT7: begin

        if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 0;
              state <= RX_STOP_BIT;
              count <= 0;
              dout[7] <= rx;
          end else begin
              count <= count + 1;
          end

     end


     RX_STOP_BIT: begin

        if(count == NUM_CLKS_PER_BIT-1) begin
              done <= 1;
              state <= RX_IDLE;
              count <= 0;
          end else begin
              count <= count + 1;
          end

      end


      default: begin
         done <= 0;
         state <= RX_IDLE;
         count <= 0;

      end


     endcase
 end
end
endmodule: uart_rx
```

RTL netlist

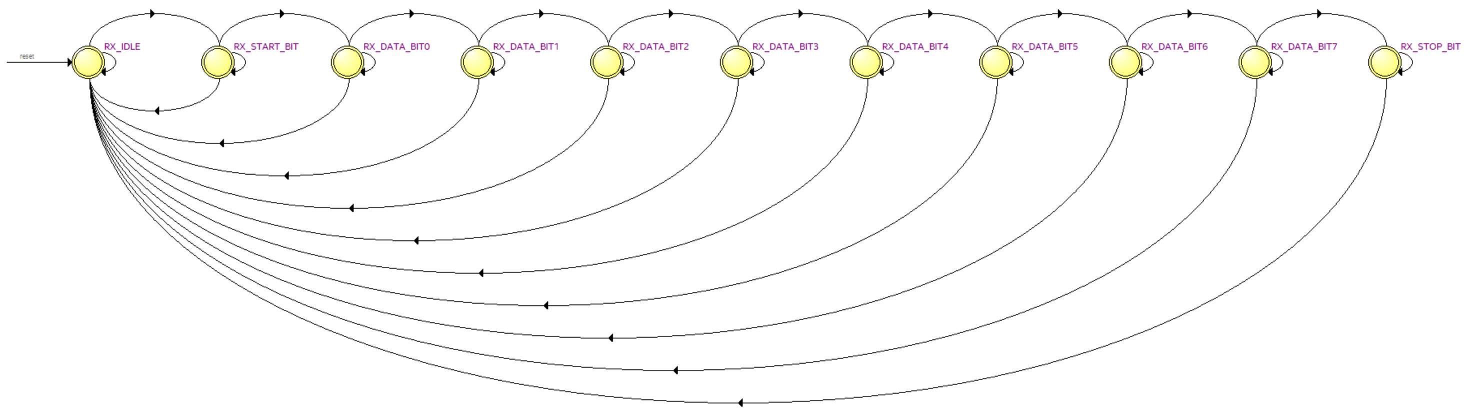


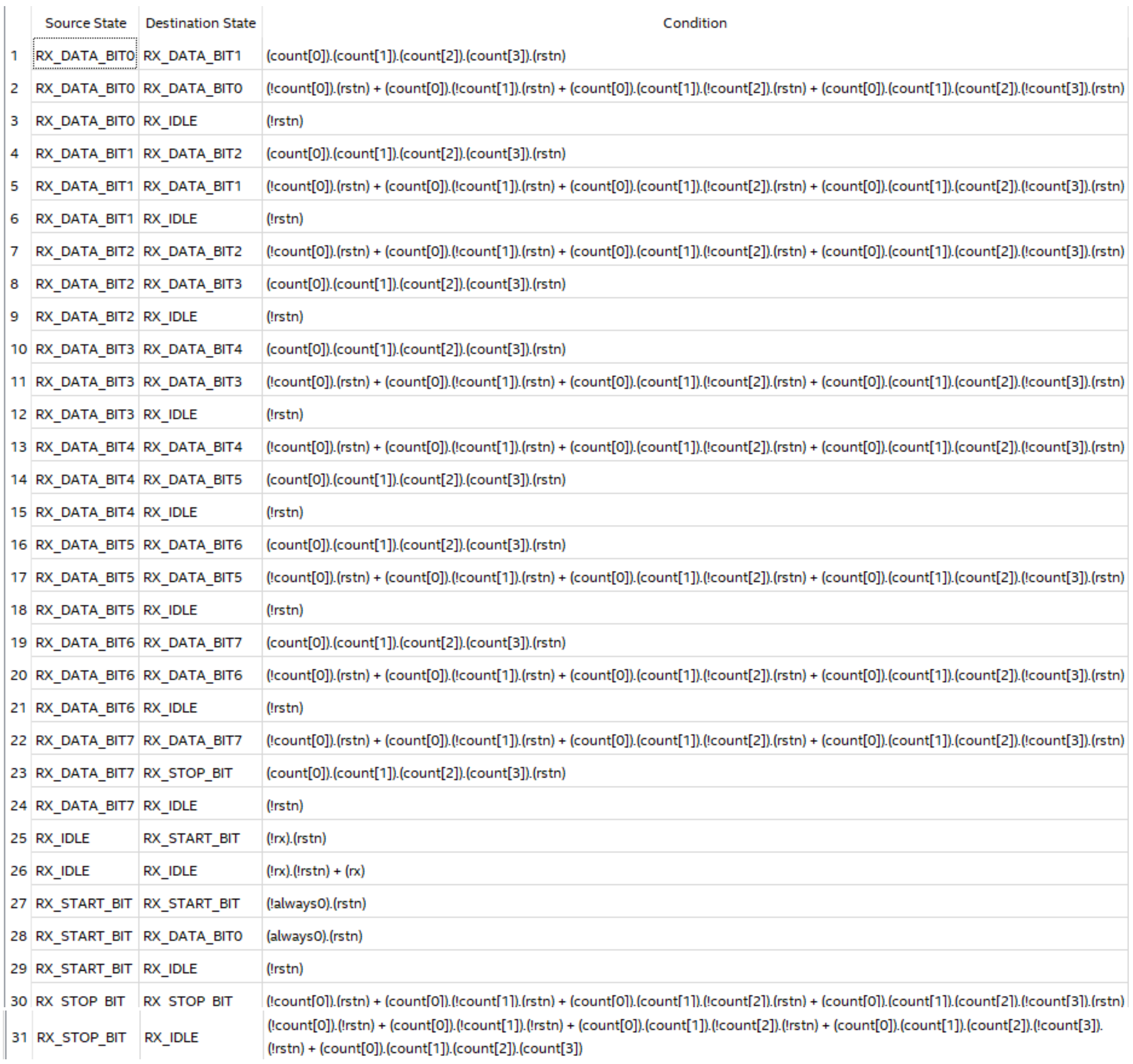
Resource usage



	Resource	Usage
1	Estimated ALUTs Used	32
1	-- Combinational ALUTs	32
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	24
3		
4	Estimated ALUTs Unavailable	15
1	-- Due to unpartnered combinational logic	15
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	32
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	1
2	-- 6 input functions	14
3	-- 5 input functions	4
4	-- 4 input functions	9
5	-- <=3 input functions	4
8		
9	Combinational ALUTs by mode	
1	-- normal mode	31
2	-- extended LUT mode	1
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	47
12		
13	Total registers	24
1	-- Dedicated logic registers	24
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	12
17		
18	DSP block 18-bit elements	0
19		

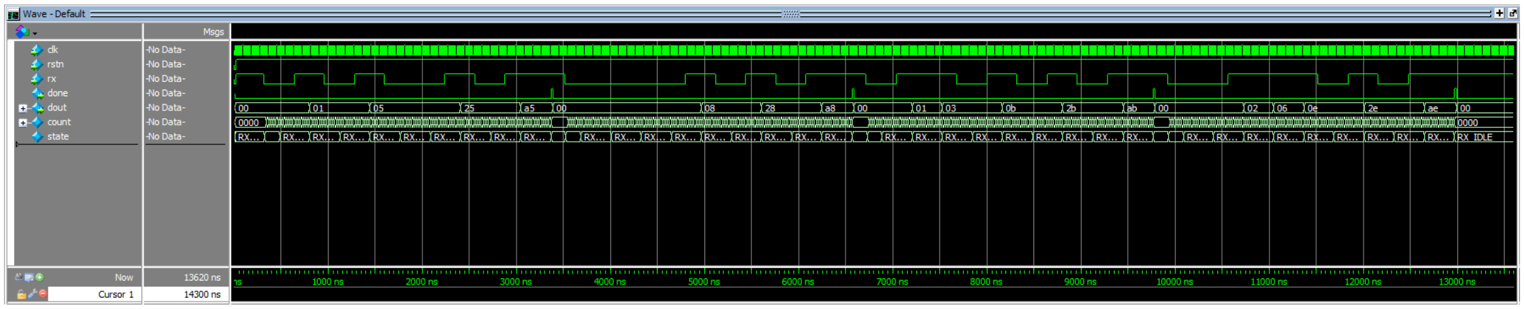
State diagram

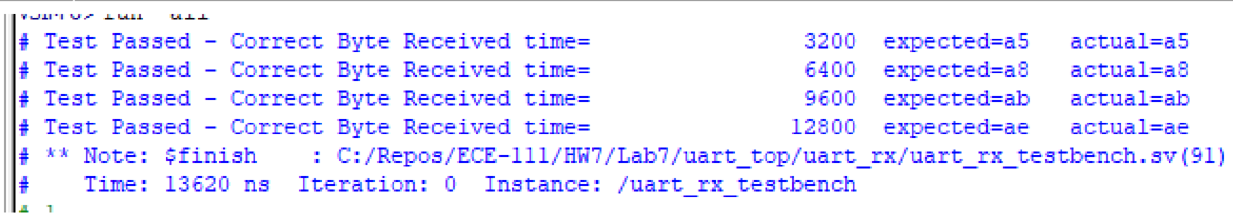




	Source State	Destination State	Condition
1	RX_DATA_BIT0	RX_DATA_BIT1	(count[0]).(count[1]).(count[2]).(count[3]).(rstn)
2	RX_DATA_BIT0	RX_DATA_BIT0	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
3	RX_DATA_BIT0	RX_IDLE	(!rstn)
4	RX_DATA_BIT1	RX_DATA_BIT2	(count[0]).(count[1]).(count[2]).(count[3]).(rstn)
5	RX_DATA_BIT1	RX_DATA_BIT1	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
6	RX_DATA_BIT1	RX_IDLE	(!rstn)
7	RX_DATA_BIT2	RX_DATA_BIT2	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
8	RX_DATA_BIT2	RX_DATA_BIT3	(count[0]).(count[1]).(count[2]).(count[3]).(rstn)
9	RX_DATA_BIT2	RX_IDLE	(!rstn)
10	RX_DATA_BIT3	RX_DATA_BIT4	(count[0]).(count[1]).(count[2]).(count[3]).(rstn)
11	RX_DATA_BIT3	RX_DATA_BIT3	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
12	RX_DATA_BIT3	RX_IDLE	(!rstn)
13	RX_DATA_BIT4	RX_DATA_BIT4	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
14	RX_DATA_BIT4	RX_DATA_BIT5	(count[0]).(count[1]).(count[2]).(count[3]).(rstn)
15	RX_DATA_BIT4	RX_IDLE	(!rstn)
16	RX_DATA_BIT5	RX_DATA_BIT6	(count[0]).(count[1]).(count[2]).(count[3]).(rstn)
17	RX_DATA_BIT5	RX_DATA_BIT5	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
18	RX_DATA_BIT5	RX_IDLE	(!rstn)
19	RX_DATA_BIT6	RX_DATA_BIT7	(count[0]).(count[1]).(count[2]).(count[3]).(rstn)
20	RX_DATA_BIT6	RX_DATA_BIT6	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
21	RX_DATA_BIT6	RX_IDLE	(!rstn)
22	RX_DATA_BIT7	RX_DATA_BIT7	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
23	RX_DATA_BIT7	RX_STOP_BIT	(count[0]).(count[1]).(count[2]).(count[3]).(rstn)
24	RX_DATA_BIT7	RX_IDLE	(!rstn)
25	RX_IDLE	RX_START_BIT	(!rx).(rstn)
26	RX_IDLE	RX_IDLE	(!rx).(!rstn) + (rx)
27	RX_START_BIT	RX_START_BIT	(!always0).(rstn)
28	RX_START_BIT	RX_DATA_BIT0	(always0).(rstn)
29	RX_START_BIT	RX_IDLE	(!rstn)
30	RX_STOP_BIT	RX_STOP_BIT	(!count[0]).(rstn) + (count[0]).(!count[1]).(rstn) + (count[0]).(count[1]).(!count[2]).(rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(rstn)
31	RX_STOP_BIT	RX_IDLE	(!count[0]).(!rstn) + (count[0]).(!count[1]).(!rstn) + (count[0]).(count[1]).(!count[2]).(!rstn) + (count[0]).(count[1]).(count[2]).(!count[3]).(!rstn) + (count[0]).(count[1]).(count[2]).(count[3])

Testbench simulation waveform



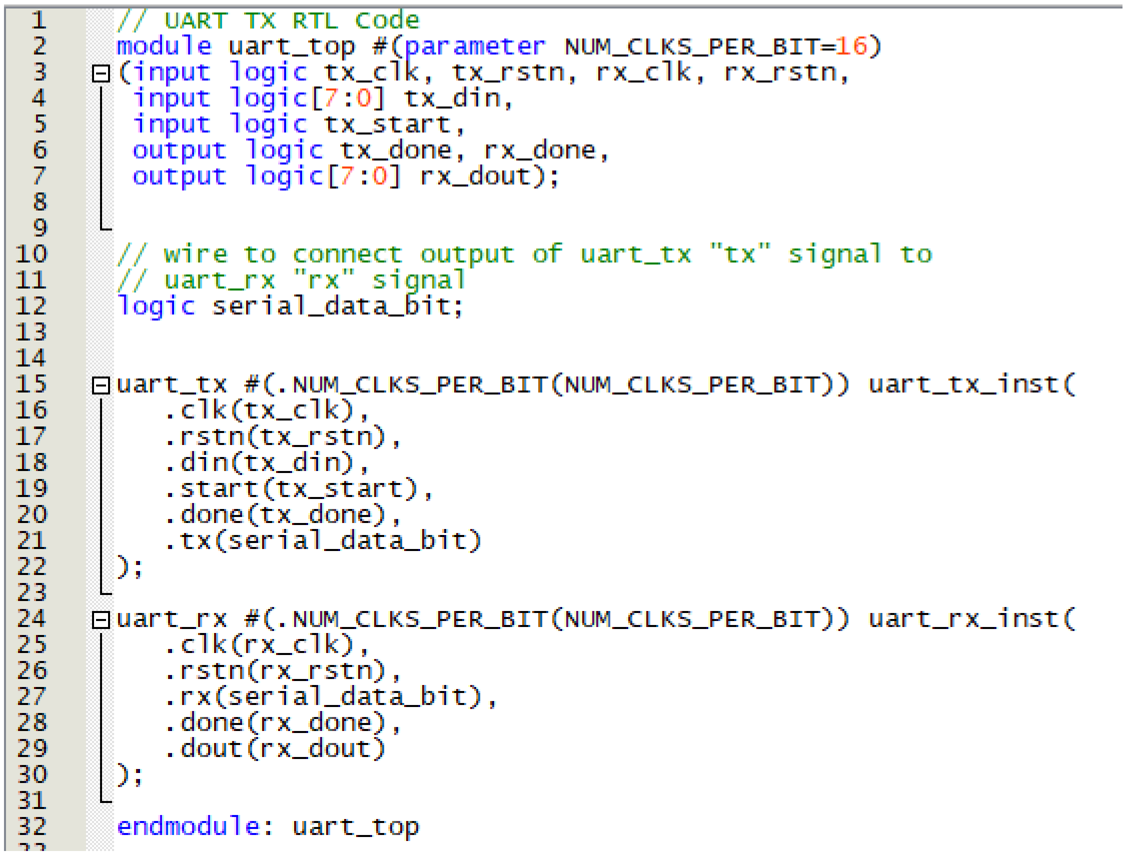


```
# Test Passed - Correct Byte Received time=              3200  expected=a5   actual=a5
# Test Passed - Correct Byte Received time=              6400  expected=a8   actual=a8
# Test Passed - Correct Byte Received time=              9600  expected=ab   actual=ab
# Test Passed - Correct Byte Received time=             12800  expected=ae   actual=ae
# ** Note: $finish    : C:/Repos/ECE-111/HW7/Lab7/uart_top/uart_rx/uart_rx_testbench.sv(91)
#    Time: 13620 ns  Iteration: 0  Instance: /uart_rx_testbench
```

- When rx first goes low, this indicates a start bit, and we can observe the state variable going to that state
- Then, for 16 clock cycles from the halfway point of that start bit, we can observe the dout register getting values changed from LSB to MSB
- Moreover, the state also changes at the halfway point of every bit, indicating that the incoming bits are being sampled at the right times
- Once the eight bit has been sampled, the state changes to the stop state, where the stop bit is sampled as a high – subsequently the done bit is raised high as a pulse, dout represents the received byte, then the receiver goes back to idle state
- The receiver DUT passes all the tests
- Described above are signs that the receiver is behaving correctly

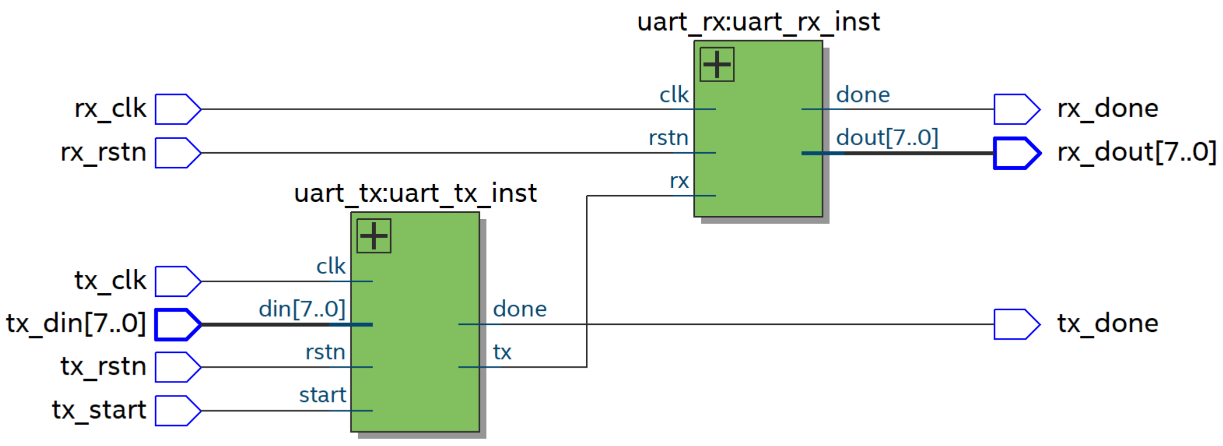
UART TX-RX Communication System

Code

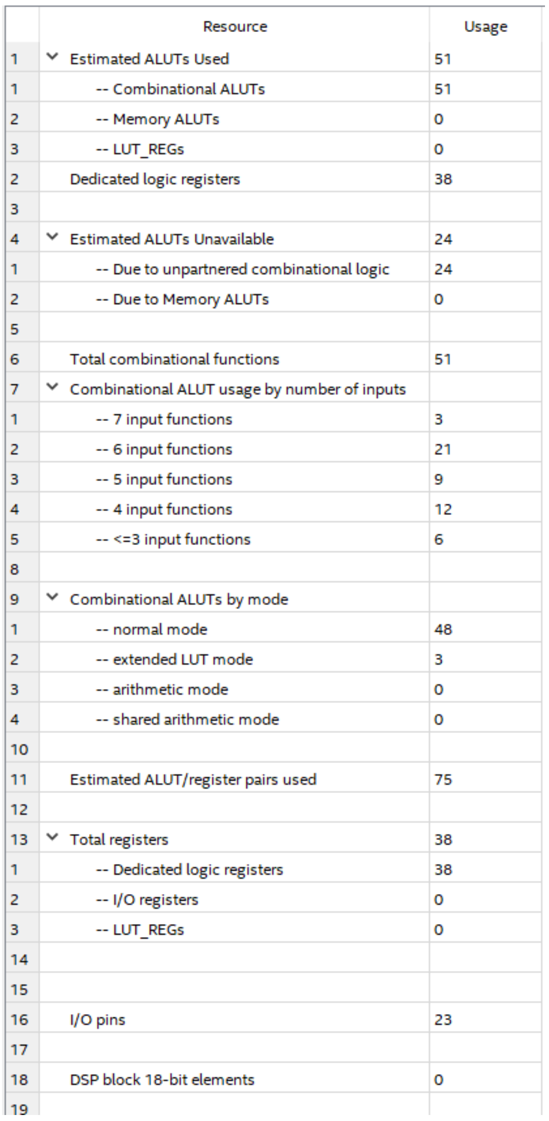


```
// UART TX RTL Code
module uart_top #(parameter NUM_CLKS_PER_BIT=16)
(input logic tx_clk, tx_rstn, rx_clk, rx_rstn,
 input logic[7:0] tx_din,
 input logic tx_start,
 output logic tx_done, rx_done,
 output logic[7:0] rx_dout);


// wire to connect output of uart_tx "tx" signal to
// uart_rx "rx" signal
logic serial_data_bit;


uart_tx #(.NUM_CLKS_PER_BIT(NUM_CLKS_PER_BIT)) uart_tx_inst(
    .clk(tx_clk),
    .rstn(tx_rstn),
    .din(tx_din),
    .start(tx_start),
    .done(tx_done),
    .tx(serial_data_bit)
);

uart_rx #(.NUM_CLKS_PER_BIT(NUM_CLKS_PER_BIT)) uart_rx_inst(
    .clk(rx_clk),
    .rstn(rx_rstn),
    .rx(serial_data_bit),
    .done(rx_done),
    .dout(rx_dout)
);

endmodule: uart_top
```

RTL netlist

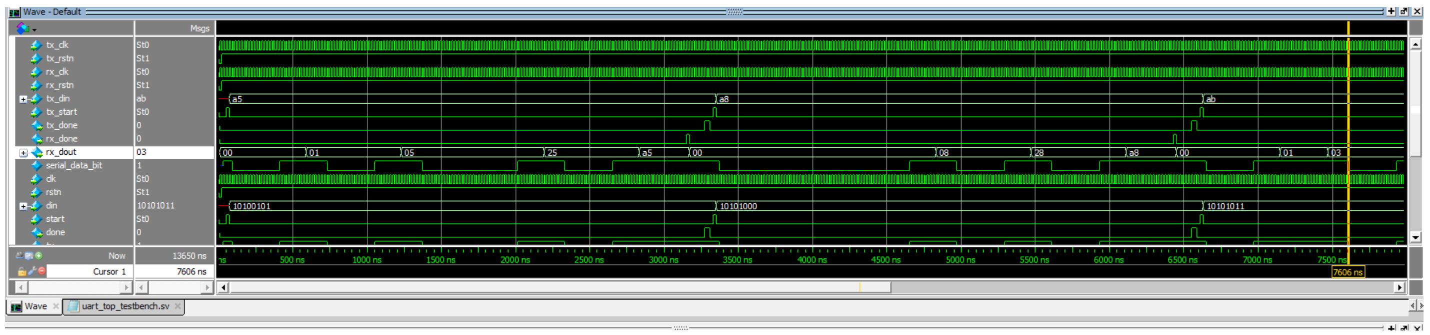


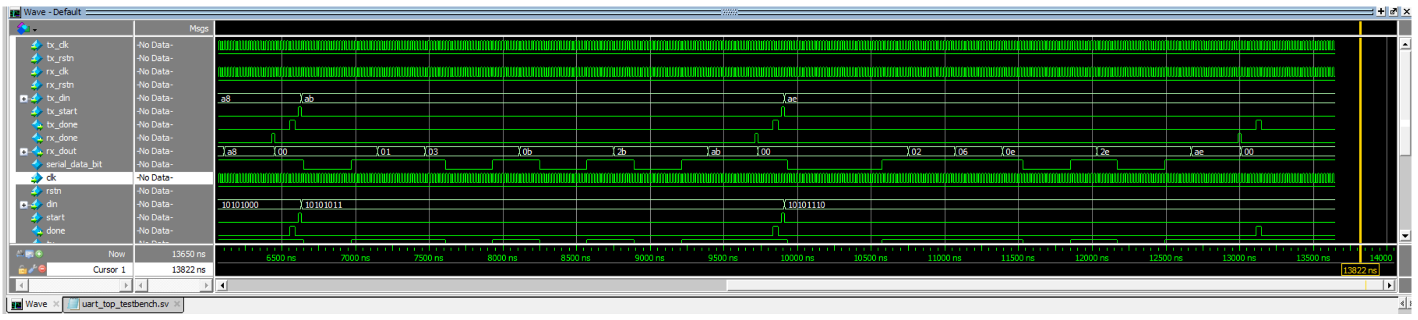
Resource usage

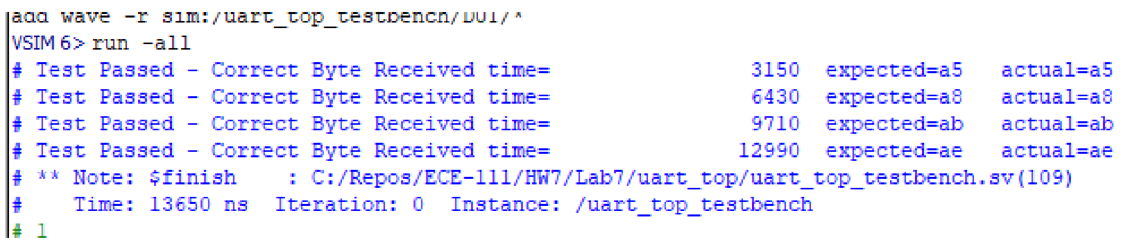


	Resource	Usage
1	Estimated ALUTs Used	51
1	-- Combinational ALUTs	51
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	38
3		
4	Estimated ALUTs Unavailable	24
1	-- Due to unpartnered combinational logic	24
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	51
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	3
2	-- 6 input functions	21
3	-- 5 input functions	9
4	-- 4 input functions	12
5	-- <=3 input functions	6
8		
9	Combinational ALUTs by mode	
1	-- normal mode	48
2	-- extended LUT mode	3
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	75
12		
13	Total registers	38
1	-- Dedicated logic registers	38
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	23
17		
18	DSP block 18-bit elements	0
19		

Testbench simulation waveform



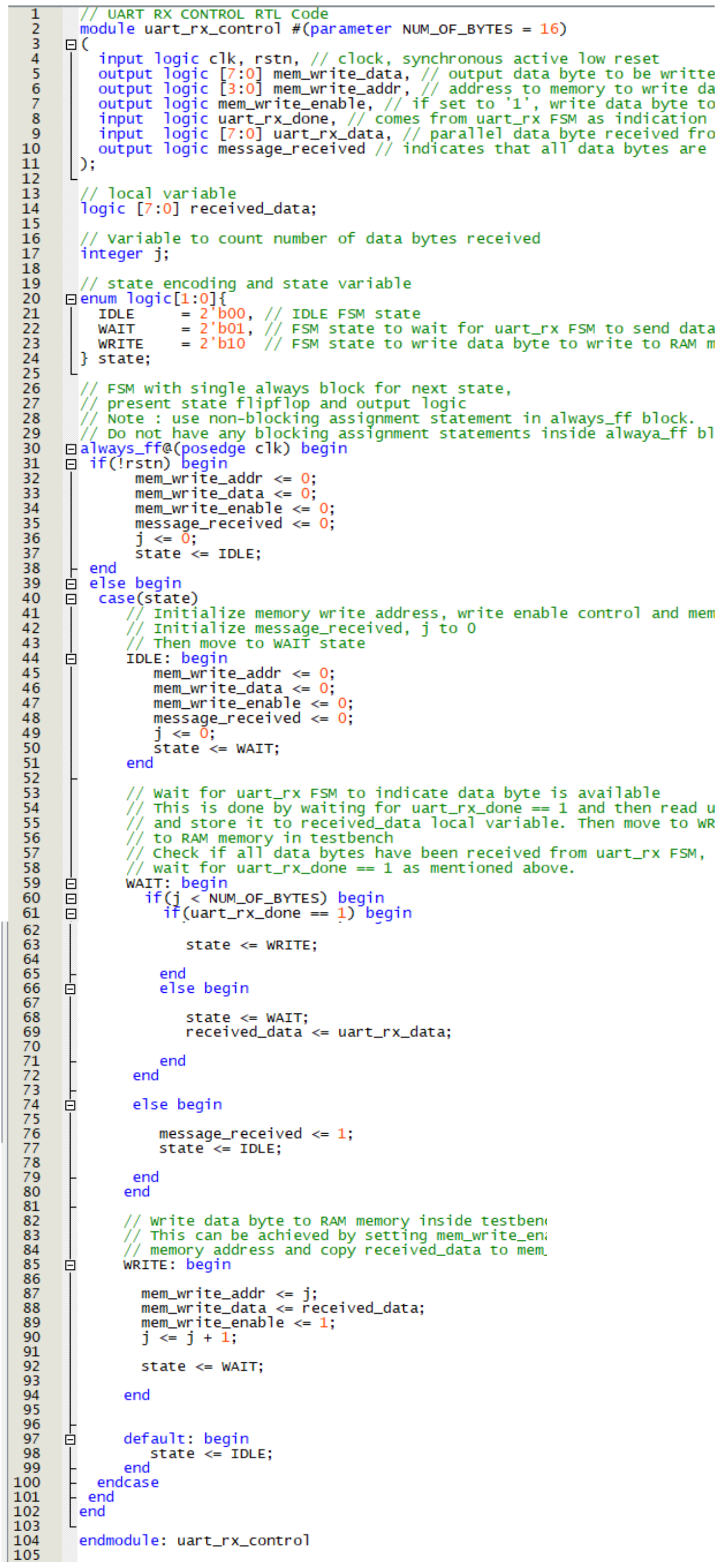



```
add wave -r sim:/uart_top_testbench/DUT/*
VSIM 6> run -all
# Test Passed - Correct Byte Received time=                  3150  expected=a5   actual=a5
# Test Passed - Correct Byte Received time=                  6430  expected=a8   actual=a8
# Test Passed - Correct Byte Received time=                  9710  expected=ab   actual=ab
# Test Passed - Correct Byte Received time=                 12990  expected=ae   actual=ae
# ** Note: $finish    : C:/Repos/ECE-111/HW7/Lab7/uart_top/uart_top_testbench.sv(109)
#    Time: 13650 ns  Iteration: 0  Instance: /uart_top_testbench
# 1
```

- Shown above is the detailed waveform of both tx and rx UART modules working together, where the tx module transmits 4 different bytes loaded in through tx_din, and the rx deserializes the byte and outputs that to rx_dot
- For example, the first byte to be transmitted is A5 in hexadecimal. After tx_start goes high, this signals tx to start serializing what is in tx_din which is A5
- After the starting of tx, we also see rx_dout start to change, indicating that rx has received that start bit, so it starts reading
- We can also observe the wire connecting the modules taking on the serialized data, on wire serial_data_bit
- Once rx has received all eight bits, it pulls rx_done high to indicate so. Similarly, tx also pulls tx_done high
- The end data is output to rx_dout. It matches with what was initially fed into tx_din
- Described above is the correct behavior between the two modules, for all four test cases

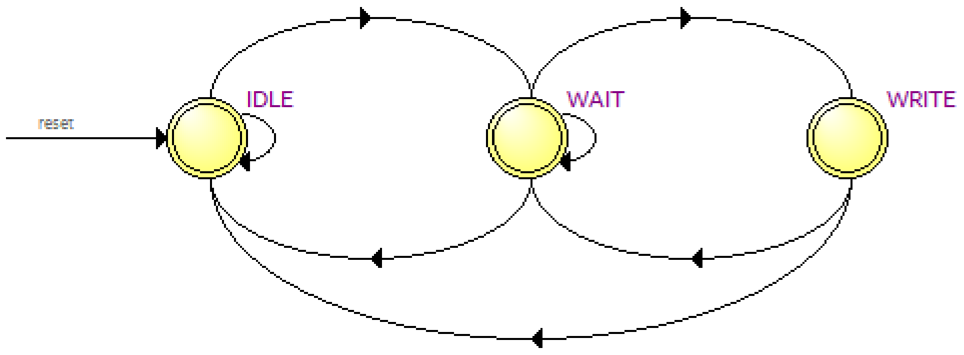
UART Control System

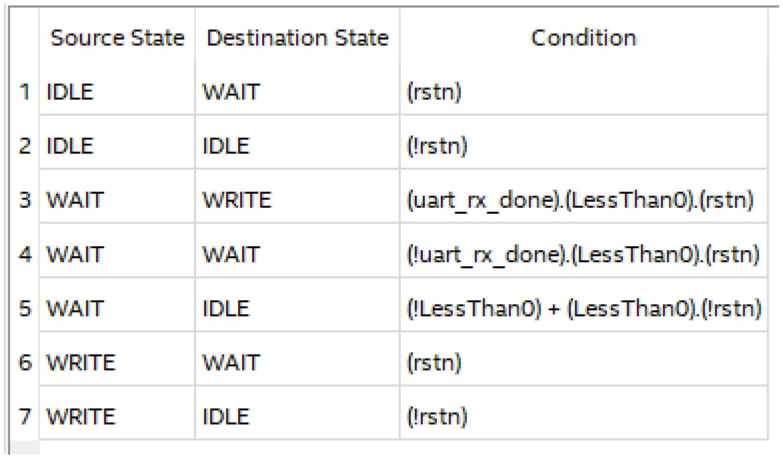
RX Control code



```
// UART RX CONTROL RTL Code
module uart_rx_control #(parameter NUM_OF_BYTES = 16)
(
  input logic clk, rstn, // clock, synchronous active low reset
  output logic [7:0] mem_write_data, // output data byte to be writte
  output logic [3:0] mem_write_addr, // address to memory to write da
  output logic mem_write_enable, // if set to '1', write data byte to
  input  logic uart_rx_done, // comes from uart_rx FSM as indication
  input  logic [7:0] uart_rx_data, // parallel data byte received fro
  output logic message_received // indicates that all data bytes are
);

// local variable
logic [7:0] received_data;

// Variable to count number of data bytes received
integer j;

// state encoding and state variable
enum logic[1:0]{
  IDLE     = 2'b00, // IDLE FSM state
  WAIT     = 2'b01, // FSM state to wait for uart_rx FSM to send data
  WRITE    = 2'b10  // FSM state to write data byte to write to RAM m
} state;

// FSM with single always block for next state,
// present state flipflop and output logic
// Note : use non-blocking assignment statement in always_ff block.
// Do not have any blocking assignment statements inside alwaya_ff bl
always_ff@(posedge clk) begin
 if(!rstn) begin
      mem_write_addr <= 0;
      mem_write_data <= 0;
      mem_write_enable <= 0;
      message_received <= 0;
      j <= 0;
      state <= IDLE;
 end
 else begin
  case(state)
     // Initialize memory write address, write enable control and mem
     // Initialize message_received, j to 0
     // Then move to WAIT state
     IDLE: begin
        mem_write_addr <= 0;
        mem_write_data <= 0;
        mem_write_enable <= 0;
        message_received <= 0;
        j <= 0;
        state <= WAIT;
     end

     // Wait for uart_rx FSM to indicate data byte is available
     // This is done by waiting for uart_rx_done == 1 and then read u
     // and store it to received_data local variable. Then move to WR
     // to RAM memory in testbench
     // Check if all data bytes have been received from uart_rx FSM,
     // wait for uart_rx_done == 1 as mentioned above.
     WAIT: begin
       if(j < NUM_OF_BYTES) begin
         if(uart_rx_done == 1) begin

            state <= WRITE;

         end
         else begin

            state <= WAIT;
            received_data <= uart_rx_data;

         end
      end

      else begin

         message_received <= 1;
         state <= IDLE;

      end
     end

     // Write data byte to RAM memory inside testben
     // This can be achieved by setting mem_write_en
     // memory address and copy received_data to mem_
     WRITE: begin

       mem_write_addr <= j;
       mem_write_data <= received_data;
       mem_write_enable <= 1;
       j <= j + 1;

       state <= WAIT;

     end


     default: begin
        state <= IDLE;
     end
  endcase
 end
end

endmodule: uart_rx_control
```

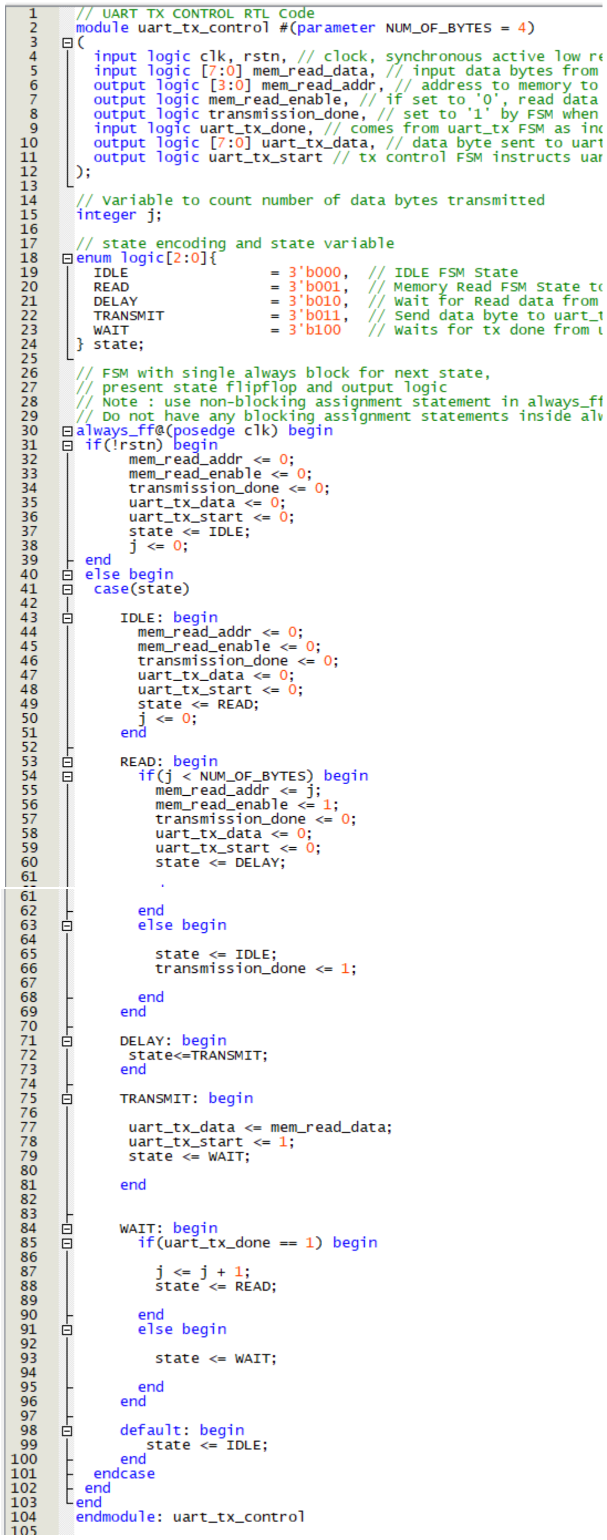
RX Control state diagram





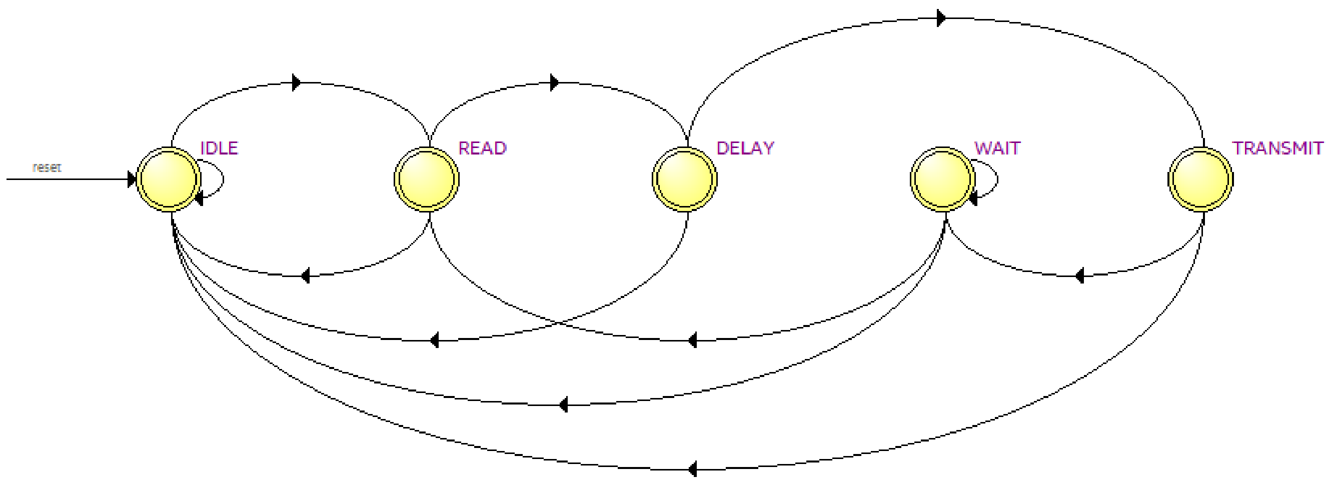
	Source State	Destination State	Condition
1	IDLE	WAIT	(rstn)
2	IDLE	IDLE	(!rstn)
3	WAIT	WRITE	(uart_rx_done).(LessThan0).(rstn)
4	WAIT	WAIT	(!uart_rx_done).(LessThan0).(rstn)
5	WAIT	IDLE	(!LessThan0) + (LessThan0).(!rstn)
6	WRITE	WAIT	(rstn)
7	WRITE	IDLE	(!rstn)

TX Control code



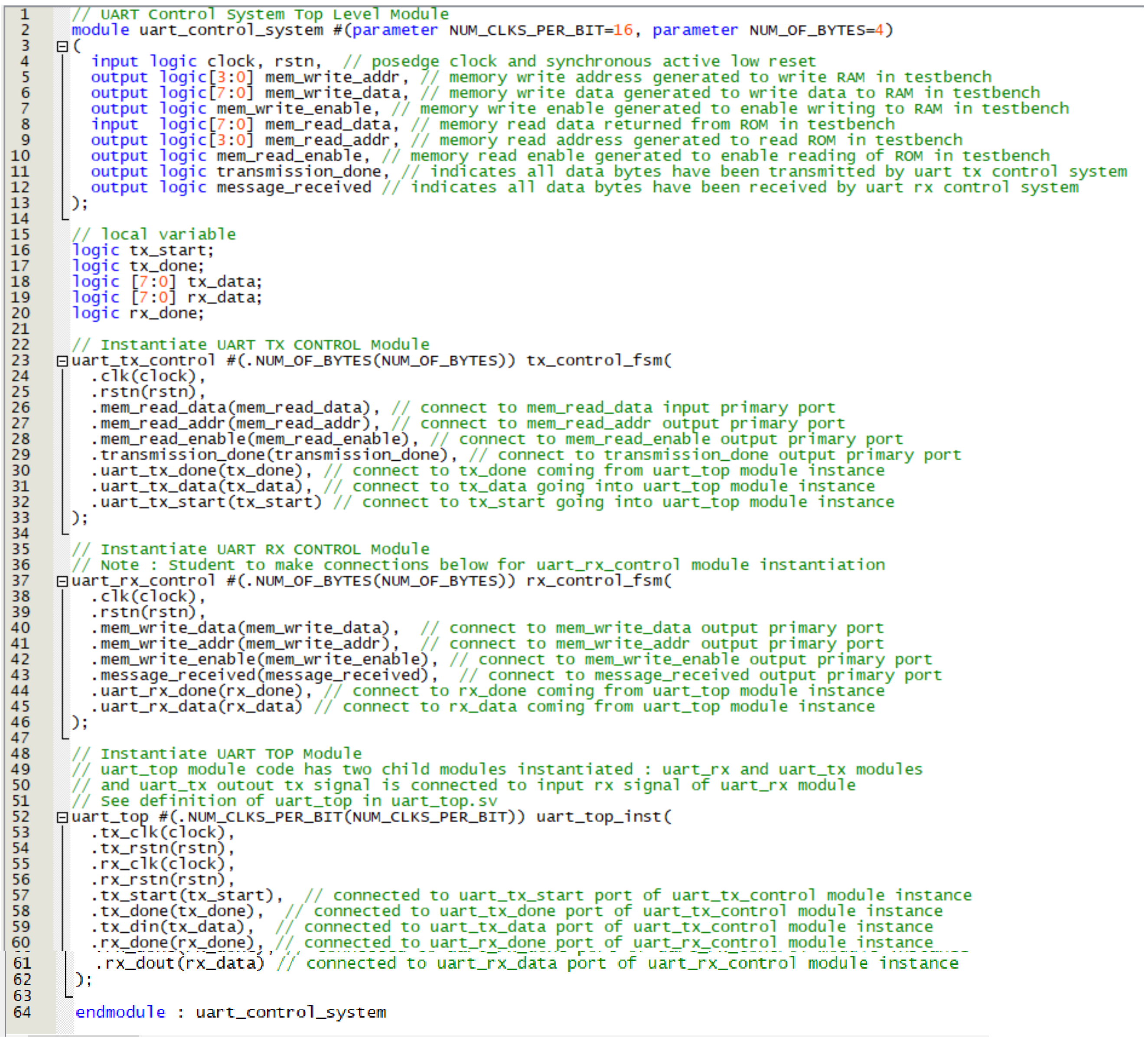
```
// UART TX CONTROL RTL Code
module uart_tx_control #(parameter NUM_OF_BYTES = 4)
(
  input logic clk, rstn, // clock, synchronous active low re
  input logic [7:0] mem_read_data, // input data bytes from
  output logic [3:0] mem_read_addr, // address to memory to
  output logic mem_read_enable, // if set to '0', read data
  output logic transmission_done, // set to '1' by FSM when
  input logic uart_tx_done, // comes from uart_tx FSM as inc
  output logic [7:0] uart_tx_data, // data byte sent to uart
  output logic uart_tx_start // tx control FSM instructs uar
);

// Variable to count number of data bytes transmitted
integer j;

// state encoding and state variable
enum logic[2:0]{
  IDLE                  = 3'b000,  // IDLE FSM State
  READ                  = 3'b001,  // Memory Read FSM State to
  DELAY                 = 3'b010,  // Wait for Read data from
  TRANSMIT              = 3'b011,  // Send data byte to uart_t
  WAIT                  = 3'b100   // Waits for tx done from u
} state;

// FSM with single always block for next state,
// present state flipflop and output logic
// Note : use non-blocking assignment statement in always_ff
// Do not have any blocking assignment statements inside alw
always_ff@(posedge clk) begin
 if(!rstn) begin
      mem_read_addr <= 0;
      mem_read_enable <= 0;
      transmission_done <= 0;
      uart_tx_data <= 0;
      uart_tx_start <= 0;
      state <= IDLE;
      j <= 0;
 end
 else begin
  case(state)

     IDLE: begin
       mem_read_addr <= 0;
       mem_read_enable <= 0;
       transmission_done <= 0;
       uart_tx_data <= 0;
       uart_tx_start <= 0;
       state <= READ;
       j <= 0;
     end

     READ: begin
       if(j < NUM_OF_BYTES) begin
         mem_read_addr <= j;
         mem_read_enable <= 1;
         transmission_done <= 0;
         uart_tx_data <= 0;
         uart_tx_start <= 0;
         state <= DELAY;



       end
       else begin

         state <= IDLE;
         transmission_done <= 1;

       end
     end

     DELAY: begin
      state<=TRANSMIT;
     end

     TRANSMIT: begin

      uart_tx_data <= mem_read_data;
      uart_tx_start <= 1;
      state <= WAIT;

     end


     WAIT: begin
       if(uart_tx_done == 1) begin

         j <= j + 1;
         state <= READ;

       end
       else begin

         state <= WAIT;

       end
     end

     default: begin
        state <= IDLE;
     end
  endcase
 end
end
endmodule: uart_tx_control
```

TX Control state diagram



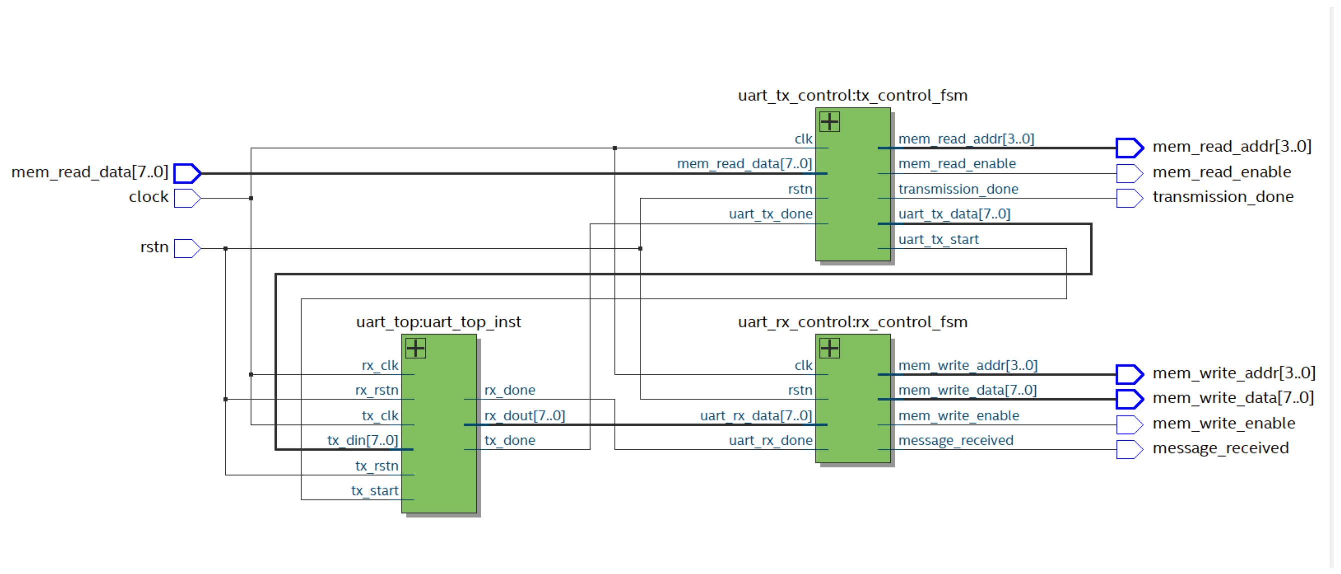
	Source State	Destination State	Condition
1	DELAY	IDLE	(!rstn)
2	DELAY	TRANSMIT	(rstn)
3	IDLE	IDLE	(!rstn)
4	IDLE	READ	(rstn)
5	READ	DELAY	(LessThan0).(rstn)
6	READ	IDLE	(!LessThan0) + (LessThan0).(!rstn)
7	TRANSMIT	IDLE	(!rstn)
8	TRANSMIT	WAIT	(rstn)
9	WAIT	IDLE	(!rstn)
10	WAIT	WAIT	(!uart_tx_done).(rstn)
11	WAIT	READ	(uart_tx_done).(rstn)

UART Control System code

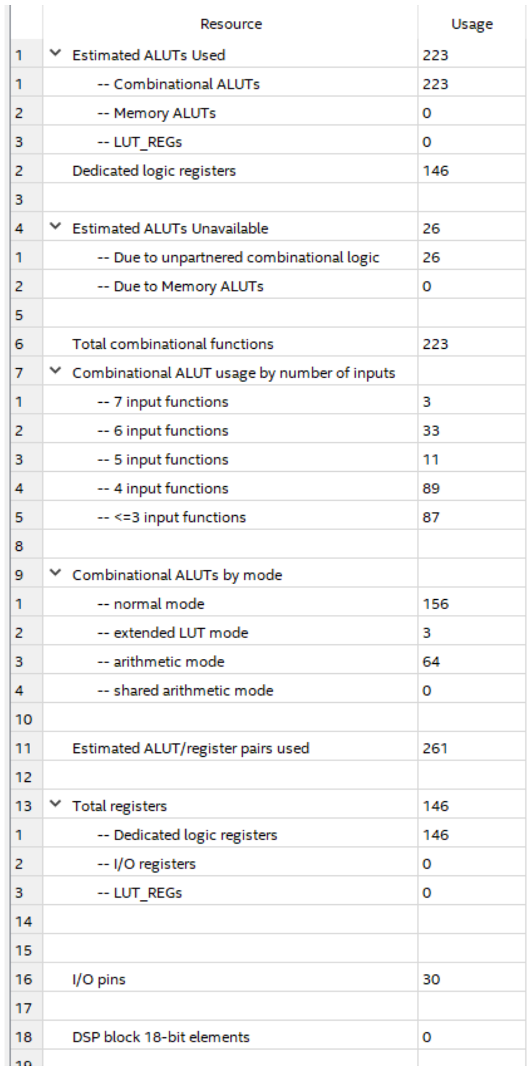


```
// UART Control System Top Level Module
module uart_control_system #(parameter NUM_CLKS_PER_BIT=16, parameter NUM_OF_BYTES=4)
(
  input logic clock, rstn,   // posedge clock and synchronous active low reset
  output logic[3:0] mem_write_addr, // memory write address generated to write RAM in testbench
  output logic[7:0] mem_write_data, // memory write data generated to write data to RAM in testbench
  output logic mem_write_enable, // memory write enable generated to enable writing to RAM in testbench
  input  logic[7:0] mem_read_data, // memory read data returned from ROM in testbench
  output logic[3:0] mem_read_addr, // memory read address generated to read ROM in testbench
  output logic mem_read_enable, // memory read enable generated to enable reading of ROM in testbench
  output logic transmission_done, // indicates all data bytes have been transmitted by uart tx control system
  output logic message_received // indicates all data bytes have been received by uart rx control system
);

// local variable
logic tx_start;
logic tx_done;
logic [7:0] tx_data;
logic [7:0] rx_data;
logic rx_done;

// Instantiate UART TX CONTROL Module
uart_tx_control #(.NUM_OF_BYTES(NUM_OF_BYTES)) tx_control_fsm(
  .clk(clock),
  .rstn(rstn),
  .mem_read_data(mem_read_data), // connect to mem_read_data input primary port
  .mem_read_addr(mem_read_addr), // connect to mem_read_addr output primary port
  .mem_read_enable(mem_read_enable), // connect to mem_read_enable output primary port
  .transmission_done(transmission_done), // connect to transmission_done output primary port
  .uart_tx_done(tx_done), // connect to tx_done coming from uart_top module instance
  .uart_tx_data(tx_data), // connect to tx_data going into uart_top module instance
  .uart_tx_start(tx_start) // connect to tx_start going into uart_top module instance
);

// Instantiate UART RX CONTROL Module
// Note : Student to make connections below for uart_rx_control module instantiation
uart_rx_control #(.NUM_OF_BYTES(NUM_OF_BYTES)) rx_control_fsm(
  .clk(clock),
  .rstn(rstn),
  .mem_write_data(mem_write_data),  // connect to mem_write_data output primary port
  .mem_write_addr(mem_write_addr),  // connect to mem_write_addr output primary port
  .mem_write_enable(mem_write_enable), // connect to mem_write_enable output primary port
  .message_received(message_received),  // connect to message_received output primary port
  .uart_rx_done(rx_done), // connect to rx_done coming from uart_top module instance
  .uart_rx_data(rx_data) // connect to rx_data coming from uart_top module instance
);

// Instantiate UART TOP Module
// uart_top module code has two child modules instantiated : uart_rx and uart_tx modules
// and uart_tx outout tx signal is connected to input rx signal of uart_rx module
// See definition of uart_top in uart_top.sv
uart_top #(.NUM_CLKS_PER_BIT(NUM_CLKS_PER_BIT)) uart_top_inst(
  .tx_clk(clock),
  .tx_rstn(rstn),
  .rx_clk(clock),
  .rx_rstn(rstn),
  .tx_start(tx_start),  // connected to uart_tx_start port of uart_tx_control module instance
  .tx_done(tx_done),  // connected to uart_tx_done port of uart_tx_control module instance
  .tx_din(tx_data),  // connected to uart_tx_data port of uart_tx_control module instance
  .rx_done(rx_done), // connected to uart_rx_done port of uart_rx_control module instance
  .rx_dout(rx_data) // connected to uart_rx_data port of uart_rx_control module instance
);

endmodule : uart_control_system
```

RTL netlist

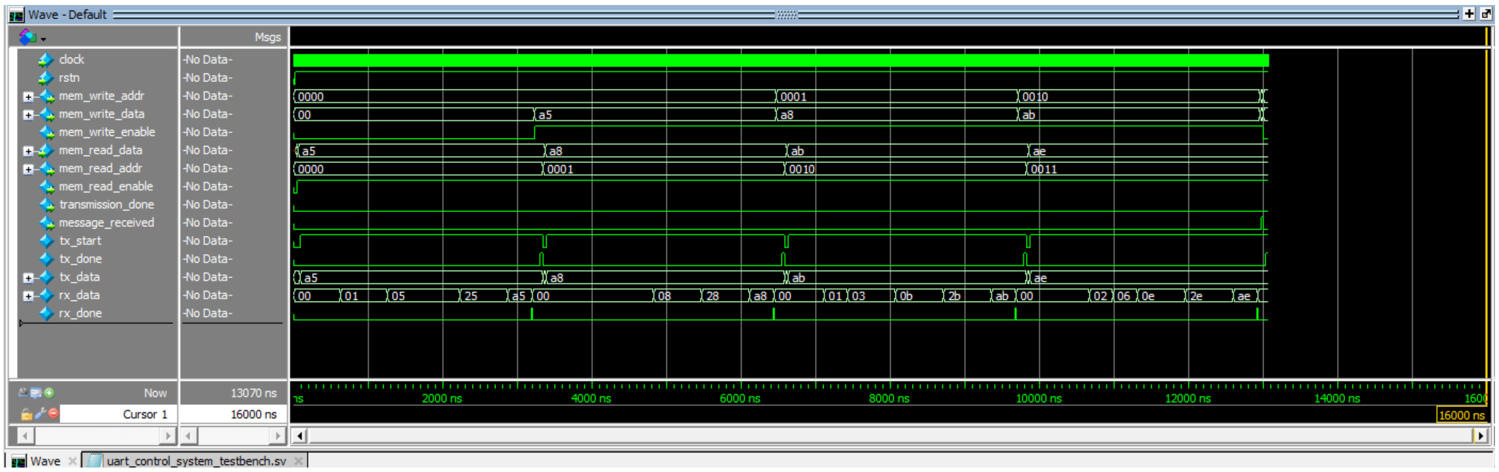


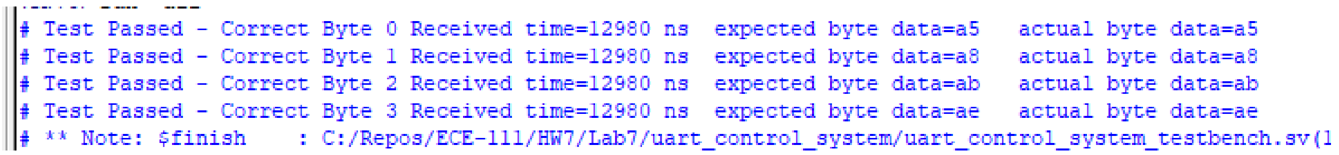
Resource usage



	Resource	Usage
1	Estimated ALUTs Used	223
1	-- Combinational ALUTs	223
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	146
3		
4	Estimated ALUTs Unavailable	26
1	-- Due to unpartnered combinational logic	26
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	223
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	3
2	-- 6 input functions	33
3	-- 5 input functions	11
4	-- 4 input functions	89
5	-- <=3 input functions	87
8		
9	Combinational ALUTs by mode	
1	-- normal mode	156
2	-- extended LUT mode	3
3	-- arithmetic mode	64
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	261
12		
13	Total registers	146
1	-- Dedicated logic registers	146
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	30
17		
18	DSP block 18-bit elements	0

Testbench simulation waveform





```
# Test Passed - Correct Byte 0 Received time=12980 ns  expected byte data=a5   actual byte data=a5
# Test Passed - Correct Byte 1 Received time=12980 ns  expected byte data=a8   actual byte data=a8
# Test Passed - Correct Byte 2 Received time=12980 ns  expected byte data=ab   actual byte data=ab
# Test Passed - Correct Byte 3 Received time=12980 ns  expected byte data=ae   actual byte data=ae
# ** Note: $finish    : C:/Repos/ECE-111/HW7/Lab7/uart_control_system/uart_control_system_testbench.sv(1
```

- The data being read from memory by the tx controller is matched with the data written to memory by the rx controller – we can observe this by seeing mem_read_data being matched by mem_write_data after rx_done signal goes high to indicate all 8 bits have been received by the UART top module
- When all four bytes have been transferred, message_received goes high indicating that all 4 bytes have been written to memory
- Described above is the correct behavior for the UART control system