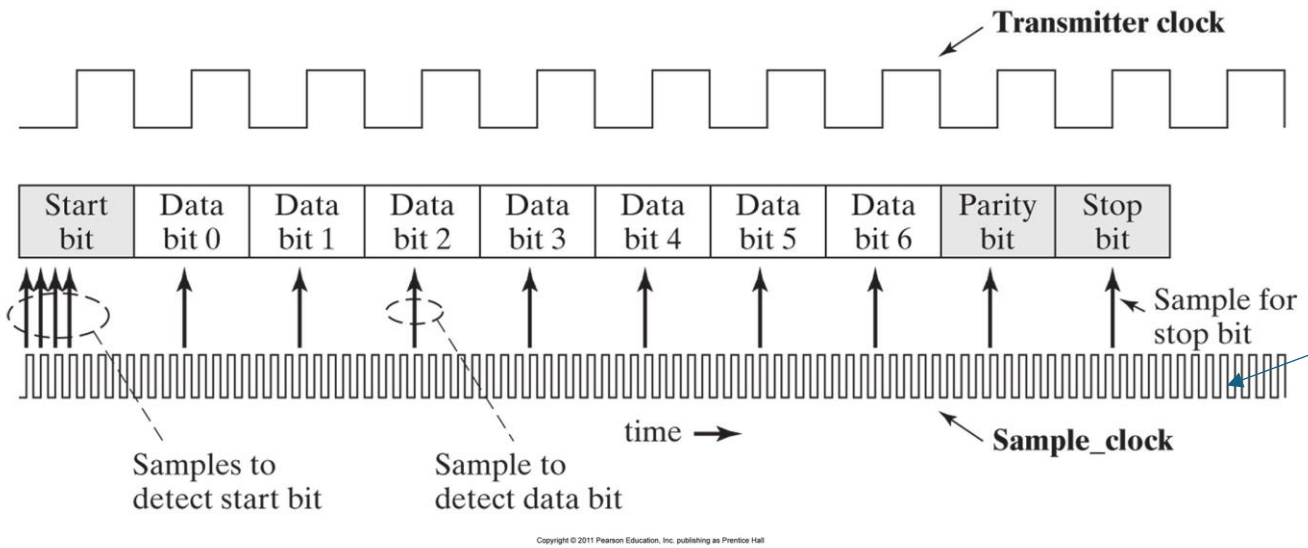
UART Receiver:



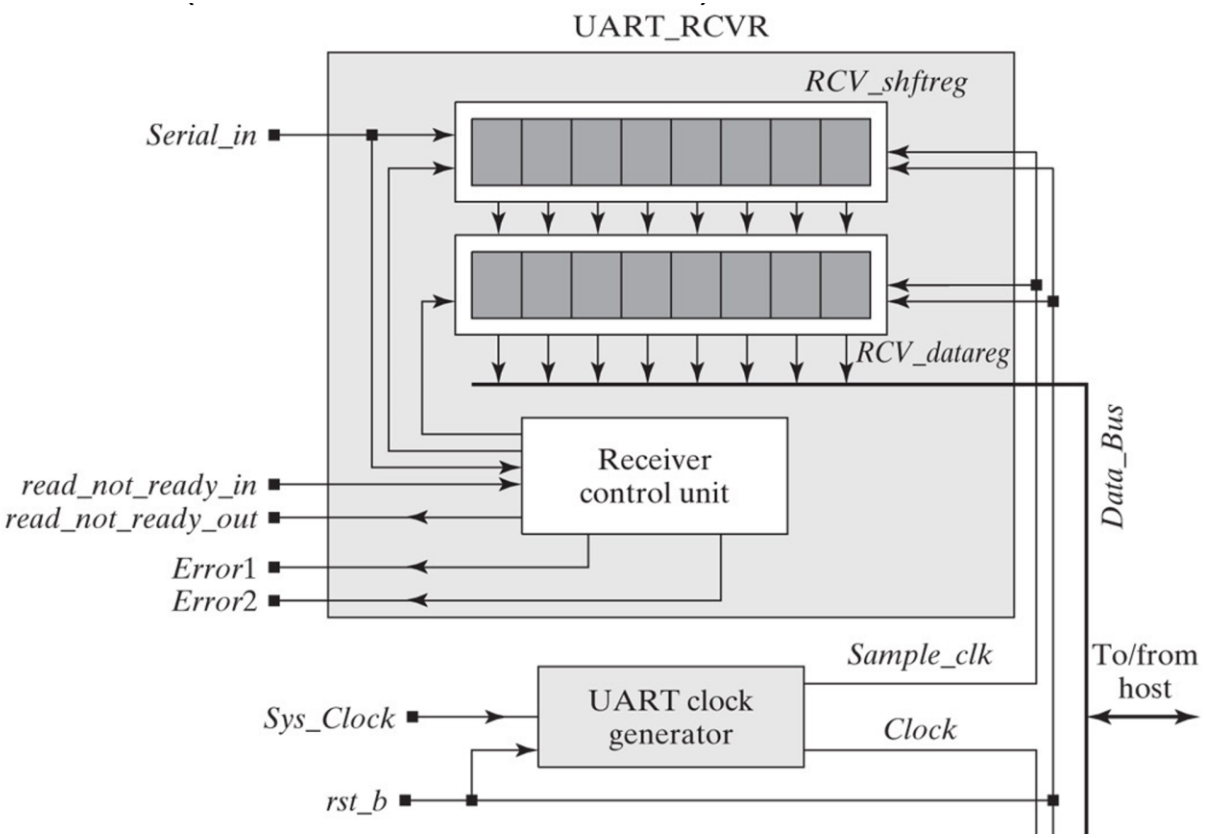
Sample_clock is 8 times higher than the bit rate in this example.

The UART receiver receives the serial bit stream of data, removes the start-bit, and transfers the data in a parallel format to a storage register. The Tx clock is not available to the receiver, so, although the data arrives at a standard bit rate, the data is not necessarily synchronized with the internal clock of the receiver host. The synchronization issue is resolved by generating a local sample clock at a higher frequency and using it to sample the received data in a manner that preserves the integrity of the data. The cycles of the sample clock will be counted to ensure that the data are sampled in the middle of a bit-time.

The sampling algorithm must: (1) Verify that the start bit has been received. (2) Generate samples from 8 bits of data. (3) Load the sampled data onto the local bus.

The arrival of a start bit will be determined by the detection of successive samples of value 0 after the serial 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 DataPath holds the sample counter which implement this scheme, and its status is reported to the control unit.

Explaining the block diagram of the UART receiver:

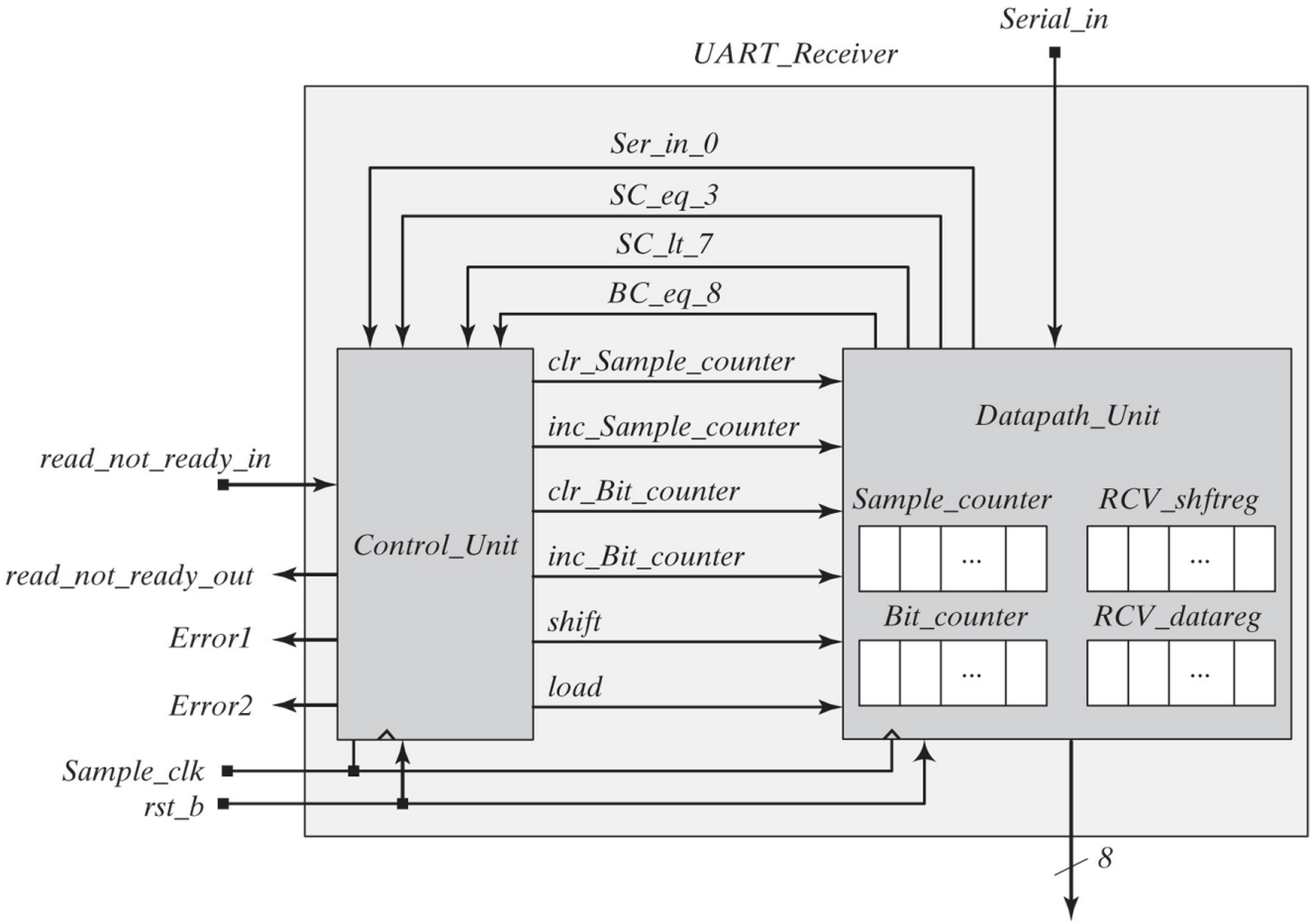


State Machine inputs:

read_not_ready_in:
signals that the host is not ready to receive data.
Ser_in_0:
asserts while *Serial_in* is 0.
SC_eq_3:
asserts while *Sample_counter* = 3.
SC_lt_7:
asserts while *Sample_counter* < 7
BC_eq_8:
asserts while Bit_counter = 8
Sample_counter:
counts the samples of a bit.
Bit_counter :
counts the bits that have been sampled

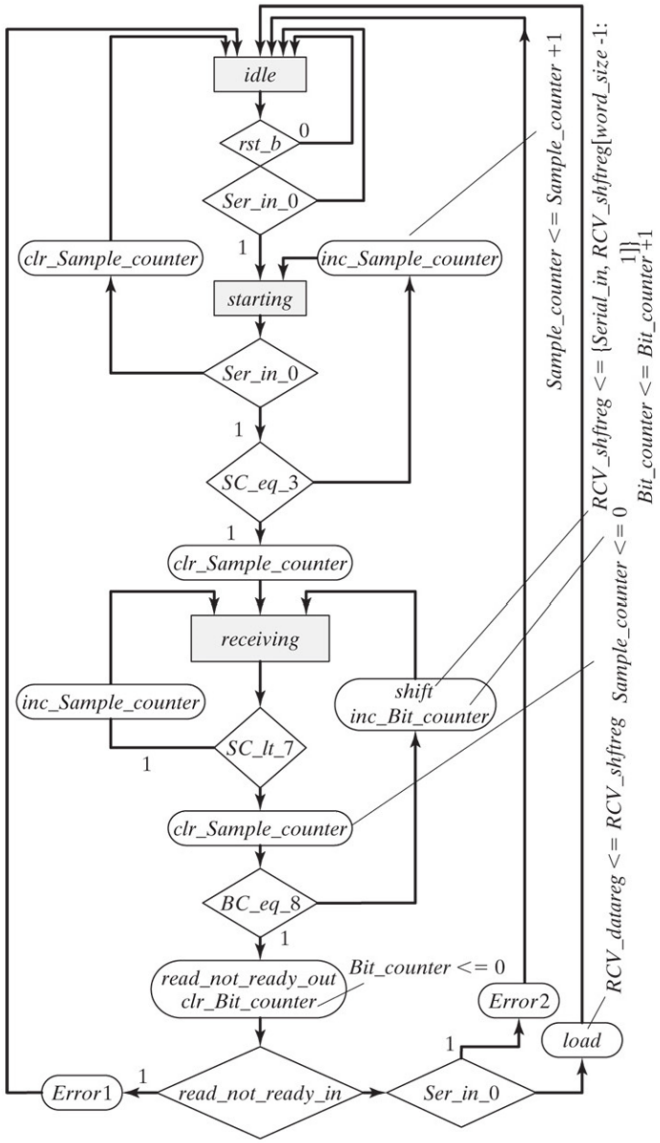
Read_not_ready_out:	signals that the receiver has received 8 bits
clr_Samples_counter:	clears Sample_counter
inc_Sample_counter:	increments Sample_counter
clr_Bit_counter:	*clears Bit_counter*
Inc_Bit_counte:	increments Bit_counter
shift	causes RCV_shftreg to shift towards the LSB
load:	causes RCV_shftreg 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

Block diagram of *UART_receiver*, including the interface signals between the control unit and the datapath unit:



```
module UART_RCVR #(parameter word_size = 8,
                            half_word = word_size/2           )
                  (output [word_size-1: 0] RCV_datareg,
                   output                  read_not_ready_out,
                                           Error1, Error2,
                   input                   Serial_in,
                                           read_not_ready_in,
                                           Sample_clk,
                                           rst_b              );


   Control_Unit M0 ( read_not_ready_out,      // output from the controller to the host
                     Error1,                  // output to host (host is not ready to receive data)
                     Error2,                  // output to host (stop_bit is missing)
                     clr_Sample_counter,      // output to DataPath (clear Sample count)
                     inc_Sample_counter,      // output to DataPath (increment Sample count)
                     clr_Bit_counter,         // output to DataPath (clear Bit counter)
                     inc_Bit_counter,         // output to DataPath (increment Bit counter)
                     shift,                   // output to DataPath
                     load,                    // output to DataPath
                     read_not_ready_in,       // input from the host
                     Ser_in_0,                // input from the DataPath (flagging zero input data)
                     SC_eq_3,                 // input from the DataPath (flagging center bit)
                     SC_lt_7,                 // input from the DataPath (Sample counter less than 7)
                     BC_eq_8,                 // input from the DataPath (Bit counter reached 8)
                     Sample_clk,              // input high rate sample clock
                     rst_b        );          // input reset

  DataPath_Unit M1 ( RCV_datareg,             // output parallel data word to the host
                     Ser_in_0,                // output flag to the controller
                     SC_eq_3,                 // output flag to the controller
                     SC_lt_7,                 // output flag to the controller
                     BC_eq_8,                 // output flag to the controller
                     Serial_in,               // input serial bit from transmission link
                     clr_Sample_counter,      // input from controller
                     inc_Sample_counter,      // input from controller
                     clr_Bit_counter,         // input from controller
                     inc_Bit_counter,         // input from controller
                     shift,                   // input from controller
                     load,                    // input from controller
                     Sample_clk,              // overall input sample clock
                     rst_b        );          // overall input reset

  endmodule
```

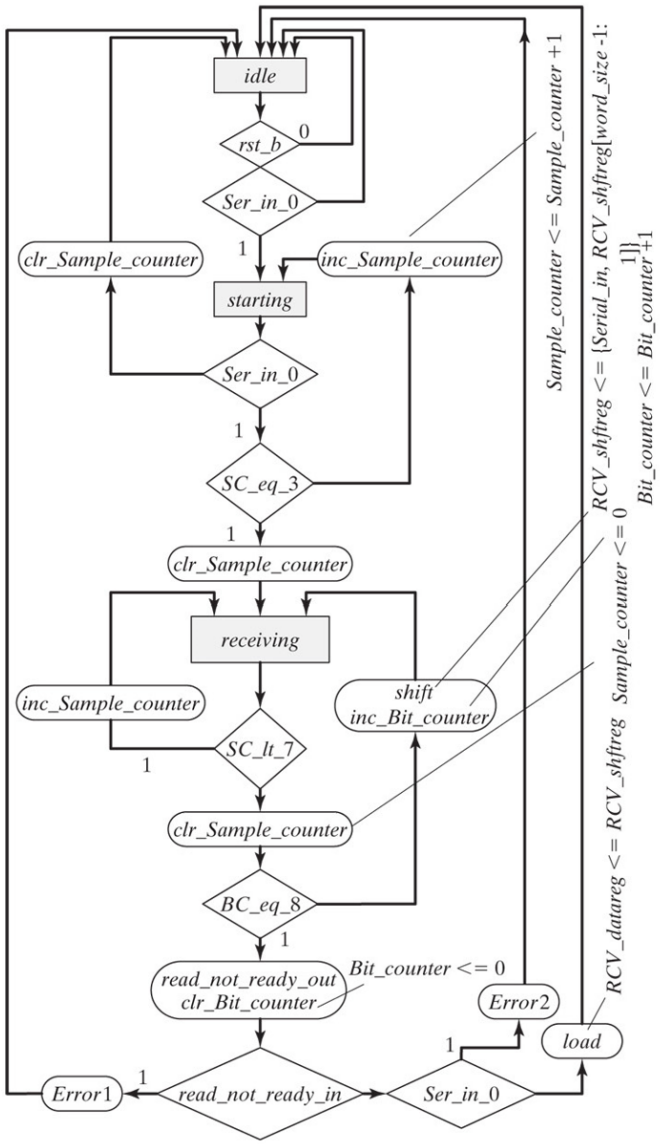
ASMD chart for *UART_receiver:*



Note: Ser_in_0 asserts if *Serial_in is* 0
SC_eq_3 asserts if *Sample_counter* = 3
SC_lt_7 asserts while *Sample_counter* < 7
BC_eq_8 asserts if *Bit_counter* = 8

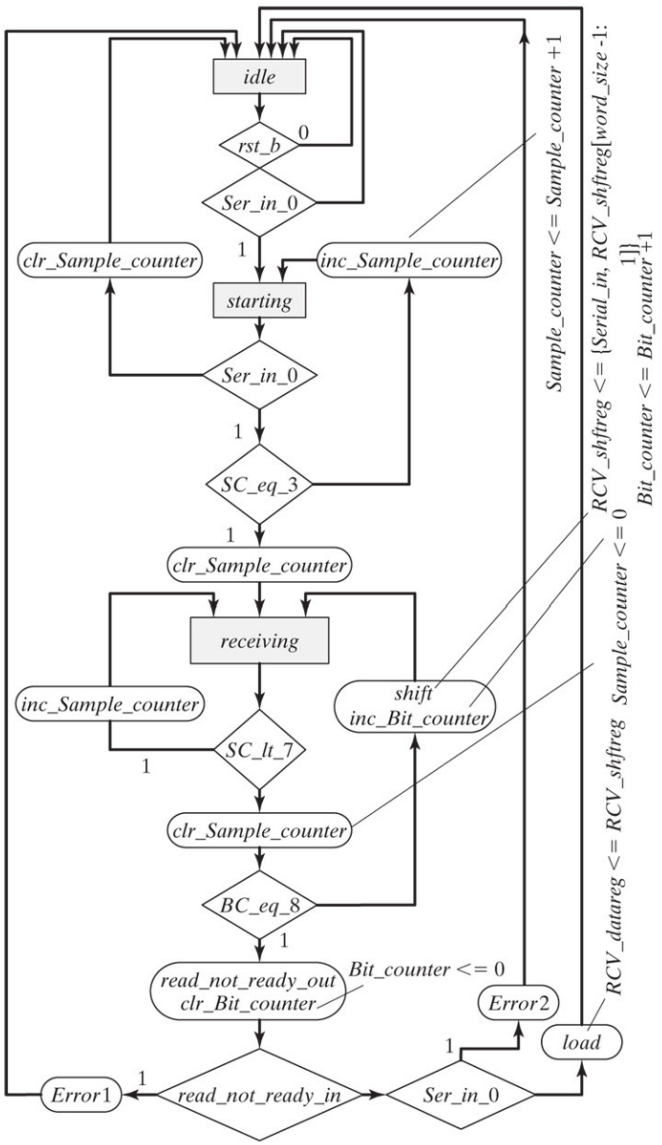
The machine has 3 states: idle, starting, and receiving. Transitions between states are synchronized by Sample_clk. Assertion of an active-low reset puts the machine in the idle state. It remains there until the Ser_in_0 is low and then makes a transition to starting.

In starting, the machine samples Serial_in repeatedly 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 3 next samples of Serial_in are 0, the machine treats the bit as a valid start-bit and goes to the state receiving. Sample_counter is clearedon the transition to receiving. In this state 8 successive samples are taken (One for each bit of the byte, at eah 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 loade into the MSB of RCV_shftreg, the receiver sift register, and will shift the 7 lefmost bits of the register toward the LSB. After the last bit has been sampled, the machine will asser 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 the 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 Ser_in_0 = 1), there is an erroring 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.



Note: Ser_in_0 asserts if *Serial_in is* 0
SC_eq_3 asserts if *Sample_counter* = 3
SC_lt_7 asserts while *Sample_counter* < 7
BC_eq_8 asserts if *Bit_counter* = 8

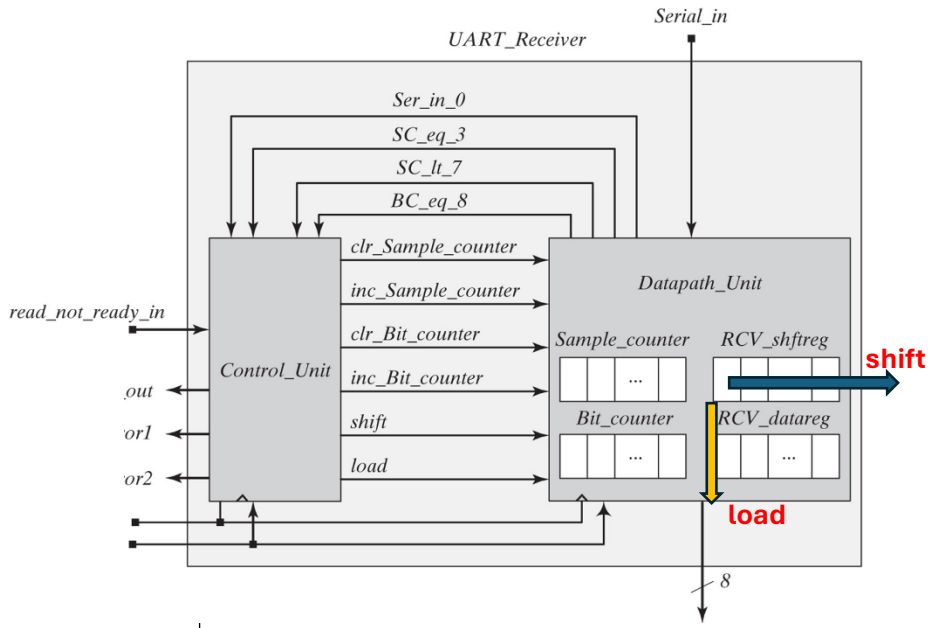
```
module Control_Unit #(parameter      word_size = 8,
                                     half_word_size = word_size/2,
                                     Num_state_bits = 2,
                                     idle       = 2'b00,
                                     starting   = 2'b01,
                                     receiving = 2'b10  )
                     ( output reg    read_not_ready_out,
                                     Error1, Error2,
                                     clr_Sample_counter,
                                     inc_Sample_counter,
                                     clr_Bit_counter,
                                     inc_Bit_counter,
                                     shift,
                                     load,
                       input         read_not_ready_in,
                                     Ser_in_0,
                                     SC_eq_3,
                                     SC_lt_7,
                                     BC_eq_8,
                                     Sample_clk,
                                     rst_b               );

  reg    [word_size-1      :0]      RCV_shftreg;
  reg    [Num_state_bits-1 :0]      state, next_state;


  always @ (posedge Sample_clk)
     if (~rst_b)   state <= idle;
        else       state <= next_state;

  always @ (state, Ser_in_0, SC_eq_3, SC_lt_7, read_not_ready_in)
     begin
     // default values
     read_not_ready_out = 0;
     Error1             = 0;
     Error2             = 0;
     clr_Sample_counter = 0;
     inc_Sample_counter = 0;
     clr_Bit_counter    = 0;
     inc_Bit_counter    = 0;
     shift              = 0;
     load               = 0;
     next_state         = idle;
```



Note: Ser_in_0 asserts if *Serial_in is* 0
SC_eq_3 asserts if *Sample_counter* = 3
SC_lt_7 asserts while *Sample_counter* < 7
BC_eq_8 asserts if *Bit_counter* = 8

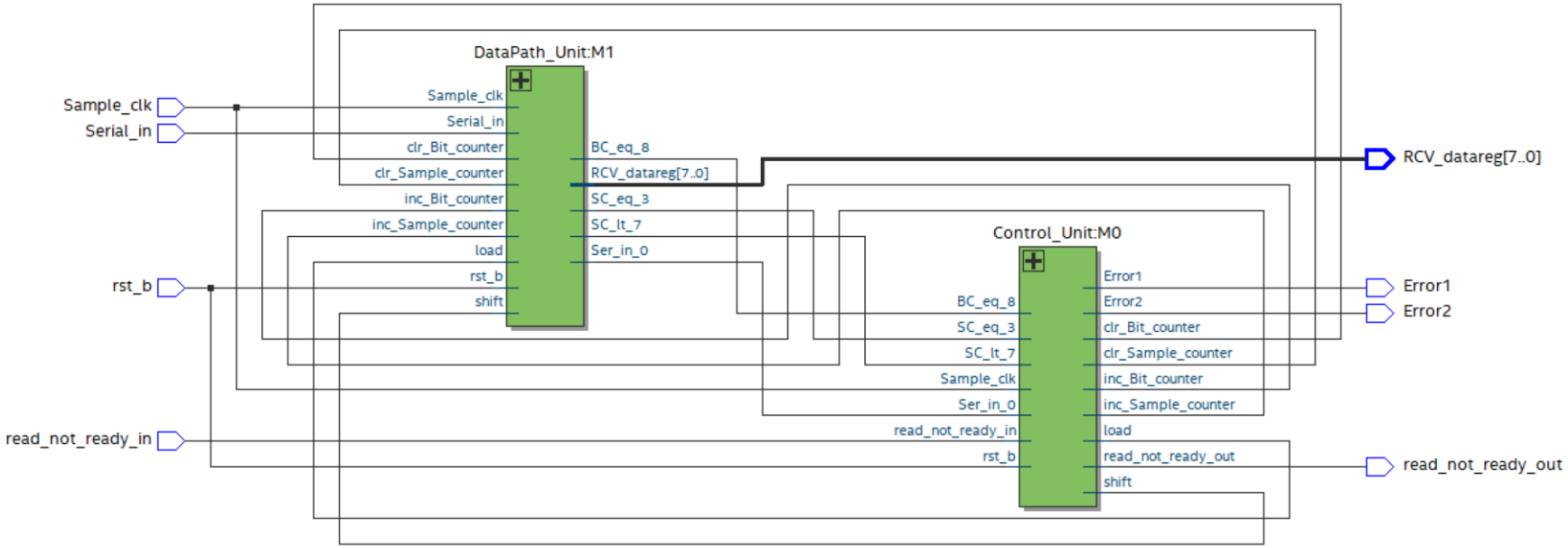
```
case (state)

idle       :    if (Ser_in_0)    next_state = starting;
                else             next_state = idle;

starting   :    if (Ser_in_0 == 1'b0)
             // not enough samples of zero for start-bit
                begin
                next_state = idle;
                clr_Sample_counter = 1;
                end
              else if (SC_eq_3 == 1'b1)
             // enough samples confirming a 0 start bit
                begin
                next_state = receiving;
                clr_Sample_counter = 1;
                end
               else
             // still checking the validity of start-bit
                 begin
                 inc_Sample_counter = 1;
                 next_state = starting;
                 end


receiving  :    if (SC_lt_7 == 1'b1)
                   begin
                   inc_Sample_counter = 1;
                   next_state = receiving;
                   end
                 else
                   begin
                   clr_Sample_counter = 1;
                   if (!BC_eq_8)
                       begin
                       shift = 1;
                       inc_Bit_counter = 1;
                       next_state = receiving;
                       end
                    else begin
                       next_state = idle;
                       read_not_ready_out = 1;
                       clr_Bit_counter = 1;
                       if (read_not_ready_in) Error1 = 1;
                       else if (Ser_in_0)     Error2 = 1;
                       else                   load   = 1;
                          end
                    end

default    : next_state = idle;
endcase
end
```

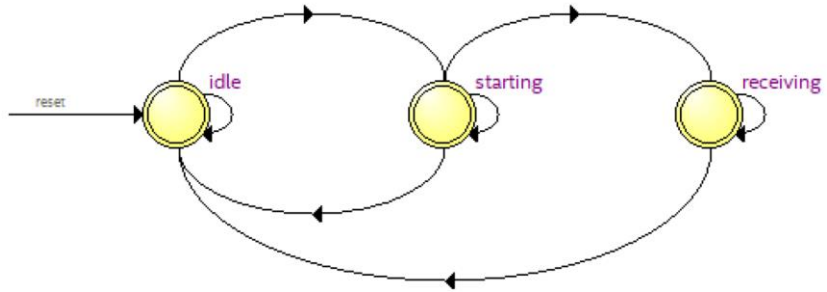
```
module DataPath_Unit #( parameter word_size = 8,
                                  half_word = word_size /2,
                                  Num_counter_bits = 4  )
( output reg [word_size -1 : 0]   RCV_datareg,
  output                          Ser_in_0,
                                  SC_eq_3,
                                  SC_lt_7,
                                  BC_eq_8,
  input                           Serial_in,
                                  clr_Sample_counter,
                                  inc_Sample_counter,
                                  clr_Bit_counter,
                                  inc_Bit_counter,
                                  shift,
                                  load,
                                  Sample_clk,
                                  rst_b                    );
reg [word_size-1: 0]          RCV_shftreg;
reg [Num_counter_bits-1: 0]   Sample_counter;     // 0-7 count
reg [Num_counter_bits  : 0]   Bit_counter;        // 0-8 count

assign Ser_in_0 = (Serial_in == 1'b0);
assign BC_eq_8  = (Bit_counter == word_size);
assign SC_lt_7  = (Sample_counter < word_size-1);
assign SC_eq_3  = (Sample_counter == half_word-1);


  always @ (posedge Sample_clk)
  if (!rst_b)
      begin
      Sample_counter <= 0;
      Bit_counter    <= 0;
      RCV_datareg    <= 0;
      RCV_shftreg    <= 0;
      end
  else begin
           if (clr_Sample_counter)
           Sample_counter <= 0;
      else if (inc_Sample_counter)
           Sample_counter <= (Sample_counter + 1);
           if (clr_Bit_counter)
           Bit_counter <= 0;
      else if (inc_Bit_counter)
           Bit_counter <= (Bit_counter + 1);
           if (shift)
           RCV_shftreg <= {Serial_in, RCV_shftreg [word_size-1: 1]};
           if (load)
           RCV_datareg <= RCV_shftreg;
       end

endmodule
```



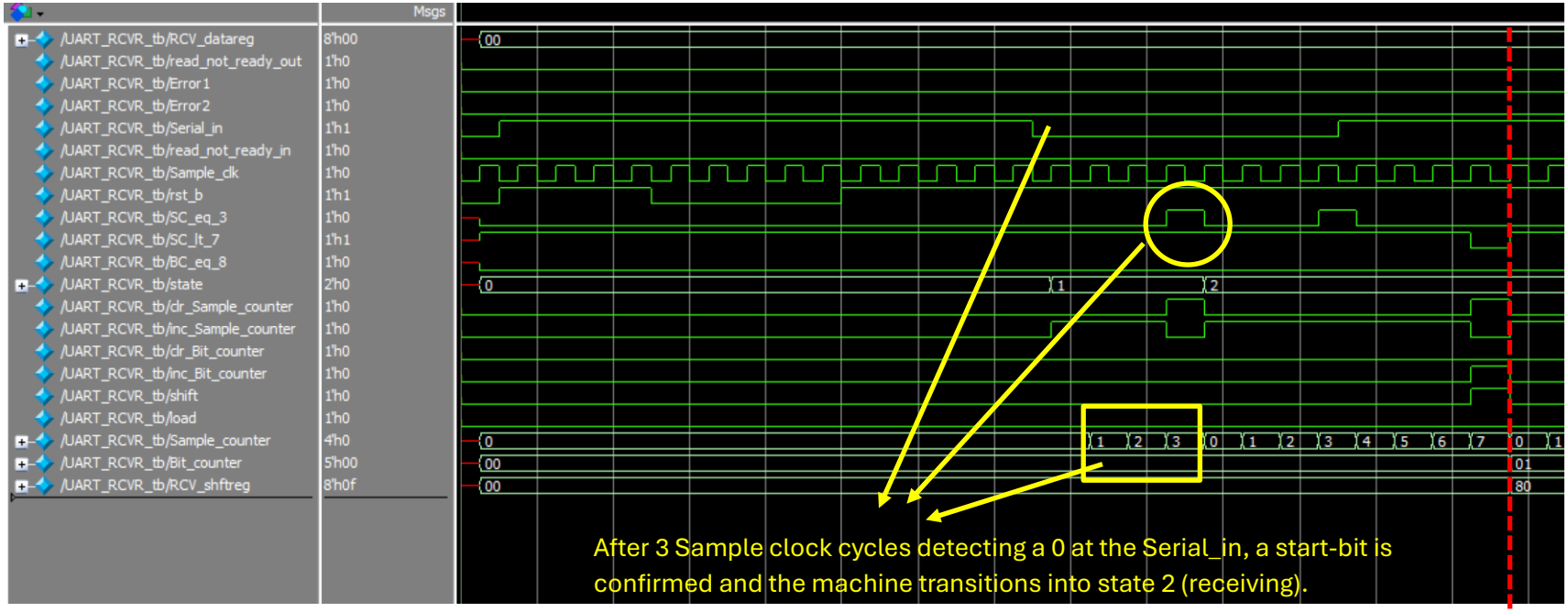
```
module UART_RCVR_tb ();

parameter word_size = 8, half_word = word_size/2;
parameter Num_counter_bits = 4;
parameter Num_state_bits = 2;

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

reg                    Serial_in;
reg                    read_not_ready_in;
reg                    Sample_clk;
reg                    rst_b;

 UART_RCVR  UUT  (RCV_datareg,
                  read_not_ready_out,
                  Error1, Error2,
                  Serial_in,
                  read_not_ready_in,
                  Sample_clk,
                  rst_b               );

// internal probes:
wire                          SC_eq_3;
wire                          SC_lt_7;
wire                          BC_eq_8;
wire [Num_state_bits-1    : 0] state;
wire                          clr_Sample_counter;
wire                          inc_Sample_counter;
wire                          clr_Bit_counter;
wire                          inc_Bit_counter;
wire                          shift, load;
wire [Num_counter_bits-1 : 0] Sample_counter;
wire [Num_counter_bits   : 0] Bit_counter;
wire [word_size-1        : 0] RCV_shftreg;

assign SC_eq_3            = UUT.SC_eq_3;
assign SC_lt_7            = UUT.SC_lt_7;
assign BC_eq_8            = UUT.BC_eq_8;
assign state              = UUT.M0.state;
assign clr_Sample_counter = UUT.clr_Sample_counter;
assign inc_Sample_counter = UUT.inc_Sample_counter;
assign clr_Bit_counter    = UUT.clr_Bit_counter;
assign inc_Bit_counter    = UUT.inc_Bit_counter;
assign shift              = UUT.shift;
assign load               = UUT.load;
assign Sample_counter     = UUT.M1.Sample_counter;
assign Bit_counter        = UUT.M1.Bit_counter;
assign RCV_shftreg        = UUT.M1.RCV_shftreg;
```

```
initial    // b5 = 1011_0101 starting with LSB
  begin       Serial_in = 0;
        # 10  Serial_in = 1;
        #140  Serial_in = 0;    // start-bit
        # 80  Serial_in = 1;    // LSB
        # 80  Serial_in = 0;
        # 80  Serial_in = 1;
        # 80  Serial_in = 0;
        # 80  Serial_in = 1;
        # 80  Serial_in = 1;
        # 80  Serial_in = 0;
        # 80  Serial_in = 1;
        # 80  Serial_in = 1;    // stop_bit (This design does
        # 80                    // not include the parity bit)
            Serial_in = 0;    // start-bit
        # 80  Serial_in = 1;    // LSB
        # 80  Serial_in = 1;
        # 80  Serial_in = 1;
        # 80  Serial_in = 0;
        # 80  Serial_in = 1;
        # 80  Serial_in = 1;
        # 80  Serial_in = 0;
        # 80  Serial_in = 0;
        # 80  Serial_in = 1;    // stop_bit (This design does
        # 80                    // not include the parity bit)


            Serial_in = 0;    // start-bit
        # 80  Serial_in = 1;    // LSB
        # 80  Serial_in = 1;
        # 80  Serial_in = 1;
        # 80  Serial_in = 1;
        # 80  Serial_in = 0;
        # 80  Serial_in = 0;
        # 80  Serial_in = 0;
        # 80  Serial_in = 1;
        # 80  Serial_in = 1;    // stop_bit (This design does
        # 80                    // not include the parity bit)

            Serial_in = 0;
        # 720 Serial_in = 1;
        #  80

            Serial_in = 0;
        #  80 Serial_in = 1;
        # 320 Serial_in = 0;   // stop_bit is not a one error
        # 400 Serial_in = 1;
  end

initial
  begin     read_not_ready_in = 0;
     #1500  read_not_ready_in = 1;
     # 700  read_not_ready_in = 0;
  end

endmodule
```

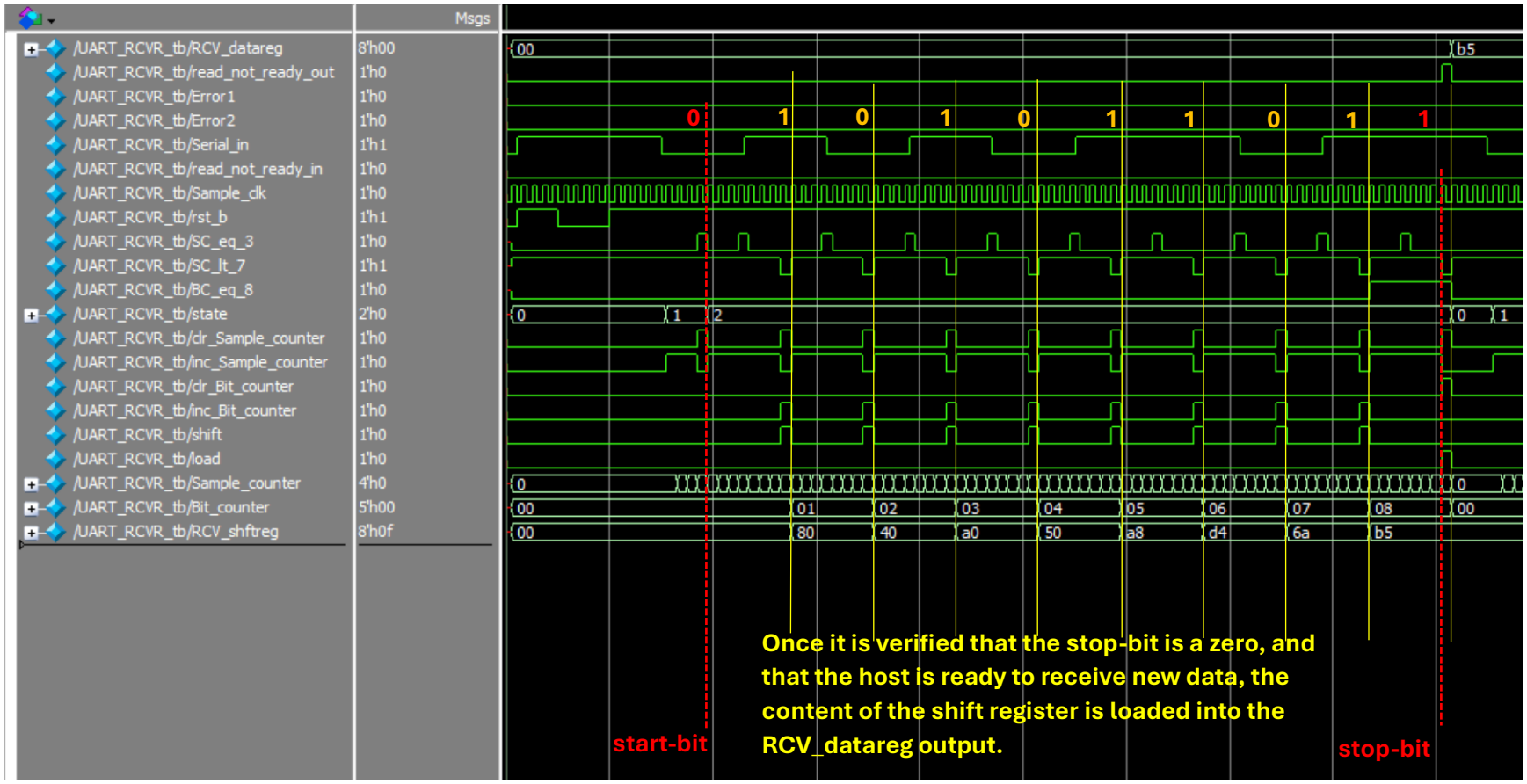




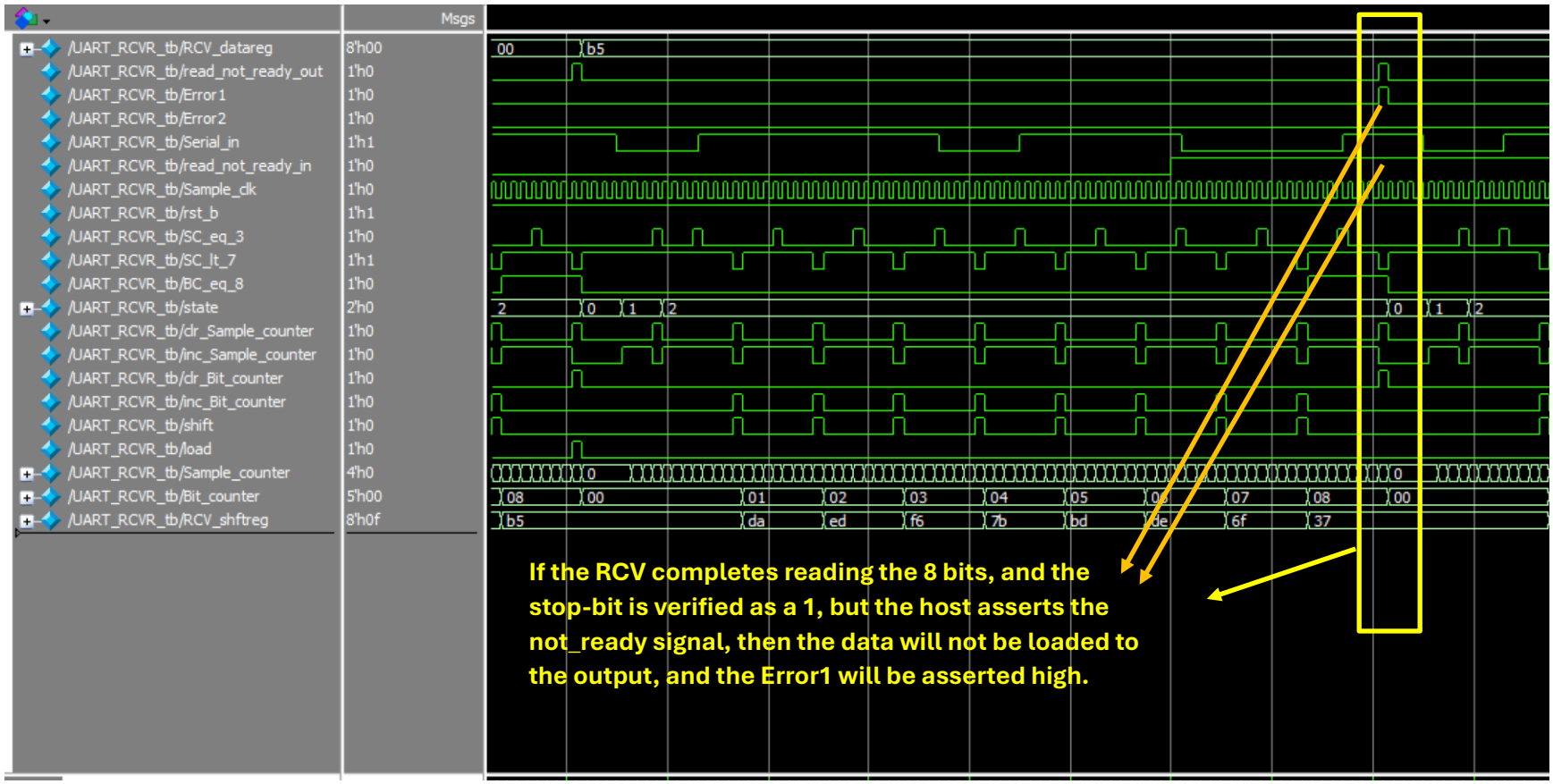
	Source State	Destination State	Condition
1	idle	idle	(!Ser_in_0) + (Ser_in_0).(!rst_b)
2	idle	starting	(Ser_in_0).(rst_b)
3	receiving	idle	(!BC_eq_8).(!rst_b) + (BC_eq_8).(!SC_lt_7) + (BC_eq_8).(SC_lt_7).(!rst_b)
4	receiving	receiving	(!BC_eq_8).(rst_b) + (BC_eq_8).(SC_lt_7).(rst_b)
5	starting	idle	(!Ser_in_0) + (Ser_in_0).(!rst_b)
6	starting	receiving	(SC_eq_3).(Ser_in_0).(rst_b)
7	starting	starting	(inc_Sample_counter).(rst_b)



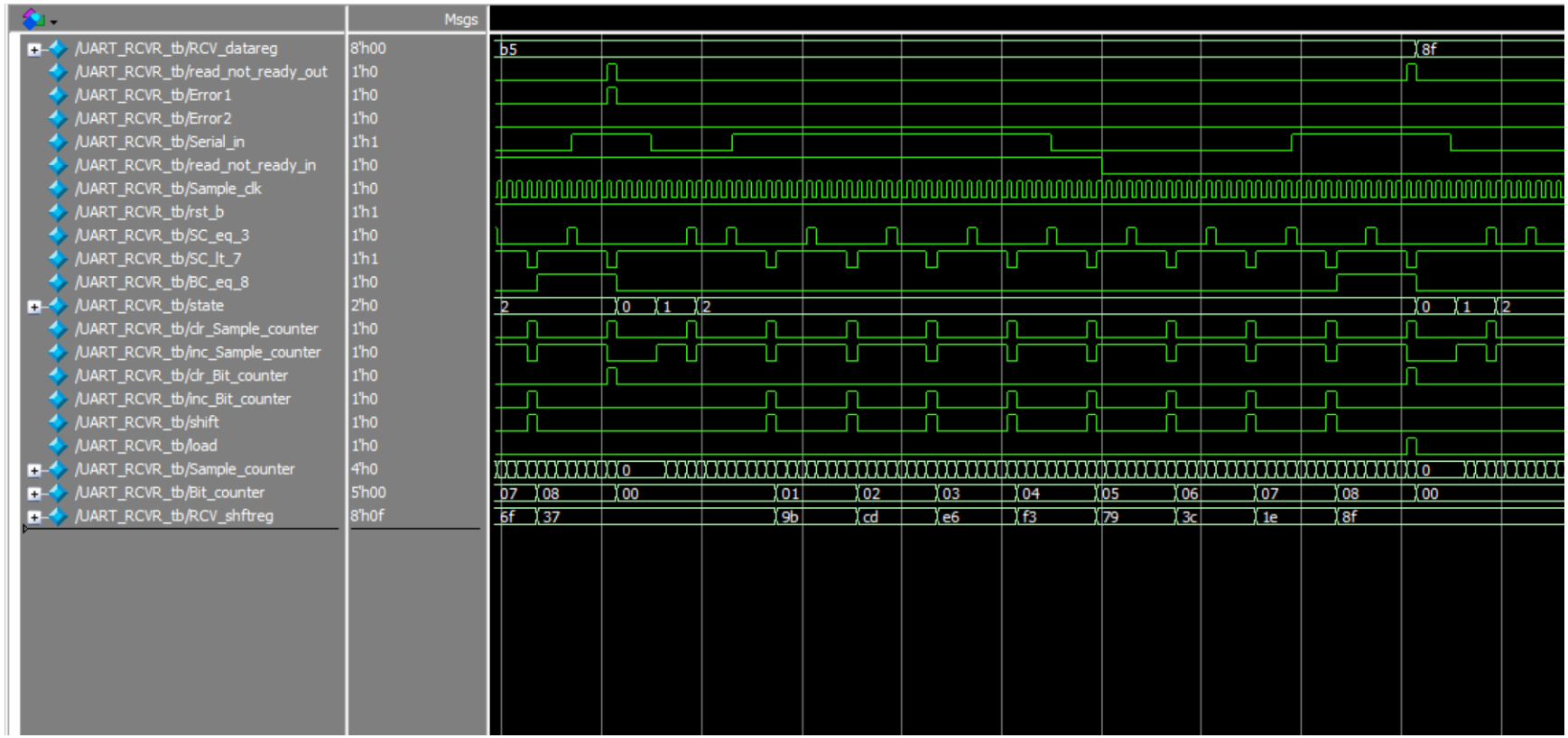
Simulation figure 1: *The detection of the first start bit after an active reset. After confirming the start bit using 3 consecutive samples, the Sample-clk count starts. After 8 Sample_clk cycles are counted, the Serial_in value is sampled and pushed into the shift register. The first bit is a 1 in this example and the RCV_shftreg was all zeros. When the sampled value is shifted into the register, 8'h 80 shows in the updated register value.*



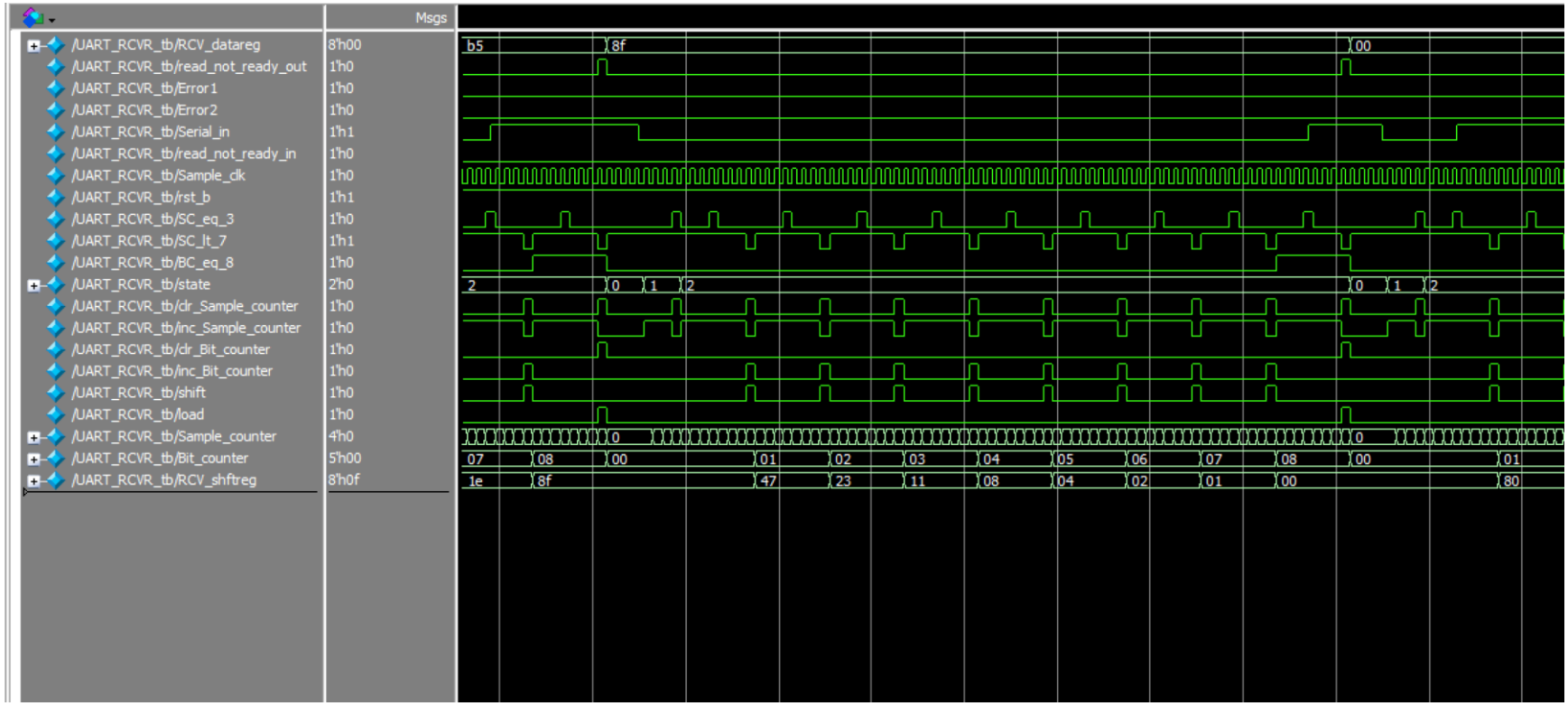
Simulation figure 2: *Here a full byte has been received, the host was ready to receive the data, and it was successfully loaded to the data register. We also see that the FSM of the controller will not load the data to the data register until it guarantees that the correct stop-bit has been received.*



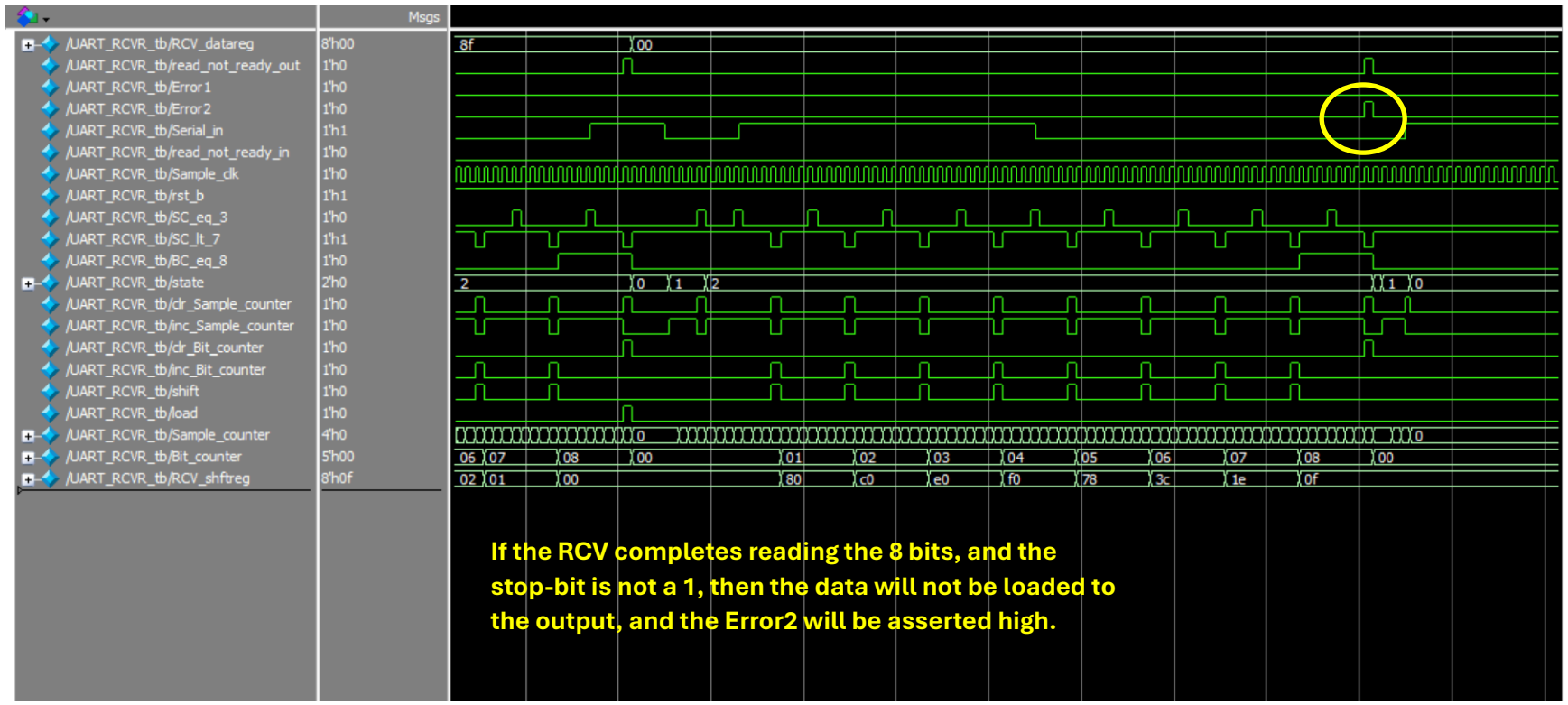
Simulation figure 3: *This is showing the case of having the host not ready to receive the data, and in this case the FSM of the controller raises the error1 flag.*



Simulation figure 4: *This is here again a perfect scenario receiving a data of 8F. Notice how the input serial bits are shifted within the shift register without resetting it. These intermediate values are completely replaced by the new incoming byte. Only the final value is loaded to the output. The start-bit can follow the stop-bit of the previous packet, and the FSM will be ready to handle the new packet. The read_not_ready_in has only an effect when the byte is fully received. Its status does not make a difference during the receiving operation.*



Simulation figure 5: *This is here again a perfect scenario receiving a data of 00 after 8F. Notice how the input serial bits are shifted within the shift register without resetting it. These intermediate values are completely replaced by the new incoming byte. Only the final value is loaded to the output. The start-bit can follow the stop-bit of the previous packet, and the FSM will be ready to handle the new packet.*



Simulation figure 6: *Here error2 is raised. A byte of 0F was sent, but the stop bit was not a 1.*