In The name of God
Project of Computer Architecture & Microprocessors

Professor: Dr.Hajsadeghi
Student :MohammadParsa Dini 400101204

June 29, 2023

Figure 1:

1 Introduction

1.1 basics

UART, which stands for Universal Asynchronous Receiver/Transmitter is a circuit for sending parallel data through a serial line. In this article we will look at how we can implement a simplified version of the receiver in verilog which we wish to implement in verilog.

Receiver & Transmitter Division of modules UART can be divided into two sub-modules: serial port sending module and serial port receiving module. The working diagram is shown in Figure 2 and Figure 3.

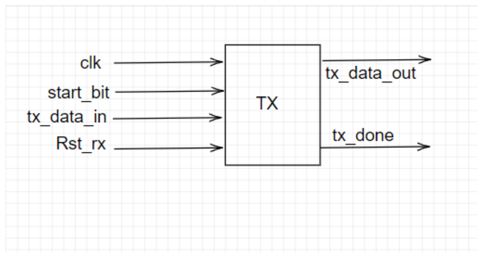


Figure 2: The Transmitter's inputs and outputs

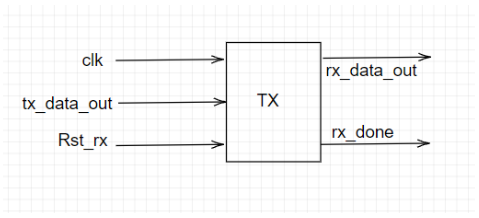


Figure 3: The Receiver's inputs and outputs

1.2 setting the baud rate

Before that we need to dig deeper into how the receiver actually performs. In UART there is no clock synchronization between the transmitter(Tx) and the receiver(Rx). This means they have to agree beforehand to a clock frequency. This is done by setting the baud rate for Rx and Tx.

In any asynchronous interface, the first thing we need to know is when in time you should sample (look at) the data. If you do not sample the data at the right time, you might see the wrong data. In order to receive your data correctly, the transmitter and receiver must agree on the baud rate. The baud rate is the rate at which the data is transmitted. For example, 8600 baud means 8600 bits per second. In this programme we will set the baud rate to either 8600 or 19200, which we will take care of it in the test bench file.

2 THe code implementation

Here is the test bench:

```
`timescale 1ns/10ps

`include "UART_TX.v"

module UART_TB ();

  // Testbench uses a 25 MHz clock
  // Want to interface to 115200 baud UART
  // 25000000 / 115200 = 217 Clocks Per Bit.
  parameter c_CLOCK_PERIOD_NS = 40;
  parameter c_CLKS_PER_BIT     = 217;
  parameter c_BIT_PERIOD       = 8600;

  reg r_Clock = 0;
  reg r_TX_DV = 0;
  wire w_TX_Active, w_UART_Line;
  wire w_TX_Serial;
  reg [7:0] r_TX_Byte = 0;
  wire [7:0] w_RX_Byte;

  UART_RX #(.CLKS_PER_BIT(c_CLKS_PER_BIT)) UART_RX_Inst
    (.i_Clock(r_Clock),
     .i_RX_Serial(w_UART_Line),
     .o_RX_DV(w_RX_DV),
     .o_RX_Byte(w_RX_Byte)
     );

  UART_TX #(.CLKS_PER_BIT(c_CLKS_PER_BIT)) UART_TX_Inst
    (.i_Clock(r_Clock),
      .i_TX_DV(r_TX_DV),
      .i_TX_Byte(r_TX_Byte),
      .o_TX_Active(w_TX_Active),
      .o_TX_Serial(w_TX_Serial),
      .o_TX_Done()
      );

  // Keeps the UART Receive input high (default) when
  // UART transmitter is not active
  assign w_UART_Line = w_TX_Active ? w_TX_Serial : 1'b1;

  always
    #(c_CLOCK_PERIOD_NS/2) r_Clock <= !r_Clock;

  // Main Testing:
  initial
    begin
      // Tell UART to send a command (exercise TX)
      @(posedge r_Clock);
      @(posedge r_Clock);
      r_TX_DV   <= 1'b1;
      r_TX_Byte <= 8'hF4;
      @(posedge r_Clock);
      r_TX_DV <= 1'b0;

      // Check that the correct command was received
      @(posedge w_RX_DV);
      if (w_RX_Byte == 8'h3F)
        $display("Test Passed - Correct Byte Received");
      else
        $display("Test Failed - Incorrect Byte Received");
      $finish();
    end

  initial
  begin
    // Required to dump signals to EPWave
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end
endmodule
```

Here is the Transmitter module:

```
module UART_RX
  #(parameter CLKS_PER_BIT = 217)
  (
   input         i_Clock,
   input         i_RX_Serial,
   output        o_RX_DV,
   output  [7:0] o_RX_Byte
   );

  parameter IDLE          = 3'b000;
  parameter RX_START_BIT  = 3'b001;
  parameter RX_DATA_BITS  = 3'b010;
  parameter RX_STOP_BIT   = 3'b011;
  parameter CLEANUP       = 3'b100;

  reg [7:0] r_Clock_Count = 0;
  reg [2:0] r_Bit_Index   = 0; //8 bits total
  reg [7:0] r_RX_Byte     = 0;
  reg       r_RX_DV       = 0;
  reg [2:0] r_SM_Main     = 0;


  // Purpose: Control RX state machine
  always @(posedge i_Clock)
  begin

    case (r_SM_Main)
      IDLE :
        begin
          r_RX_DV       <= 1'b0;
          r_Clock_Count <= 0;
          r_Bit_Index   <= 0;

          if (i_RX_Serial == 1'b0)          // Start bit detected
            r_SM_Main <= RX_START_BIT;
          else
            r_SM_Main <= IDLE;
        end

      // Check middle of start bit to make sure it's still low
      RX_START_BIT :
        begin
          if (r_Clock_Count == (CLKS_PER_BIT-1)/2)
          begin
            if (i_RX_Serial == 1'b0)
            begin
              r_Clock_Count <= 0;  // reset counter, found the middle
              r_SM_Main     <= RX_DATA_BITS;
            end
            else
              r_SM_Main <= IDLE;
          end
          else
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main     <= RX_START_BIT;
          end
        end // case: RX_START_BIT


      // Wait CLKS_PER_BIT-1 clock cycles to sample serial data
      RX_DATA_BITS :
        begin
          if (r_Clock_Count < CLKS_PER_BIT-1)
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main     <= RX_DATA_BITS;
          end
          else
```

```
          begin
            r_Clock_Count          <= 0;
            r_RX_Byte[r_Bit_Index] <= i_RX_Serial;

            // Check if we have received all bits
            if (r_Bit_Index < 7)
            begin
              r_Bit_Index <= r_Bit_Index + 1;
              r_SM_Main   <= RX_DATA_BITS;
            end
            else
            begin
              r_Bit_Index <= 0;
              r_SM_Main   <= RX_STOP_BIT;
            end
          end
        end // case: RX_DATA_BITS


      // Receive Stop bit.  Stop bit = 1
      RX_STOP_BIT :
        begin
          // Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
          if (r_Clock_Count < CLKS_PER_BIT-1)
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main     <= RX_STOP_BIT;
          end
          else
          begin
            r_RX_DV       <= 1'b1;
            r_Clock_Count <= 0;
            r_SM_Main     <= CLEANUP;
          end
        end // case: RX_STOP_BIT


      // Stay here 1 clock
      CLEANUP :
        begin
          r_SM_Main <= IDLE;
          r_RX_DV   <= 1'b0;
        end


      default :
        r_SM_Main <= IDLE;

    endcase
  end

  assign o_RX_DV   = r_RX_DV;
  assign o_RX_Byte = r_RX_Byte;

endmodule // UART_RX
```

Here is the Receiver module:

```
module UART_RX
  #(parameter CLKS_PER_BIT = 217)
  (
   input        i_Clock,
   input        i_RX_Serial,
   output       o_RX_DV,
   output [7:0] o_RX_Byte
   );

  parameter IDLE         = 3'b000;
  parameter RX_START_BIT = 3'b001;
  parameter RX_DATA_BITS = 3'b010;
  parameter RX_STOP_BIT  = 3'b011;
  parameter CLEANUP      = 3'b100;

  reg [7:0] r_Clock_Count = 0;
  reg [2:0] r_Bit_Index   = 0; //8 bits total
  reg [7:0] r_RX_Byte     = 0;
  reg       r_RX_DV       = 0;
  reg [2:0] r_SM_Main     = 0;


  // Purpose: Control RX state machine
  always @(posedge i_Clock)
  begin

    case (r_SM_Main)
      IDLE :
        begin
          r_RX_DV       <= 1'b0;
          r_Clock_Count <= 0;
          r_Bit_Index   <= 0;

          if (i_RX_Serial == 1'b0)          // Start bit detected
            r_SM_Main <= RX_START_BIT;
          else
            r_SM_Main <= IDLE;
        end

      // Check middle of start bit to make sure it's still low
      RX_START_BIT :
        begin
          if (r_Clock_Count == (CLKS_PER_BIT-1)/2)
          begin
            if (i_RX_Serial == 1'b0)
            begin
              r_Clock_Count <= 0;  // reset counter, found the middle
              r_SM_Main     <= RX_DATA_BITS;
            end
            else
              r_SM_Main <= IDLE;
          end
          else
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main       <= RX_START_BIT;
          end
        end // case: RX_START_BIT


      // Wait CLKS_PER_BIT-1 clock cycles to sample serial data
      RX_DATA_BITS :
        begin
          if (r_Clock_Count < CLKS_PER_BIT-1)
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main     <= RX_DATA_BITS;
          end
          else
          begin
            r_Clock_Count          <= 0;
            r_RX_Byte[r_Bit_Index] <= i_RX_Serial;

            // Check if we have received all bits
            if (r_Bit_Index < 7)
            begin
              r_Bit_Index <= r_Bit_Index + 1;
              r_SM_Main   <= RX_DATA_BITS;
            end
            else
            begin
              r_Bit_Index <= 0;
              r_SM_Main   <= RX_STOP_BIT;
            end
          end
        end // case: RX_DATA_BITS


      // Receive Stop bit.  Stop bit = 1
      RX_STOP_BIT :
        begin
          // Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
          if (r_Clock_Count < CLKS_PER_BIT-1)
          begin
            r_Clock_Count <= r_Clock_Count + 1;
            r_SM_Main     <= RX_STOP_BIT;
          end
          else
          begin
            r_RX_DV       <= 1'b1;
            r_Clock_Count <= 0;
            r_SM_Main     <= CLEANUP;
          end
        end // case: RX_STOP_BIT


      // Stay here 1 clock
      CLEANUP :
        begin
          r_SM_Main <= IDLE;
          r_RX_DV   <= 1'b0;
```

```
          end

        default :
          r_SM_Main <= IDLE;

      endcase
    end

    assign o_RX_DV   = r_RX_DV;
    assign o_RX_Byte = r_RX_Byte;

endmodule // UART_RX
```

3 results

In this section we will run the code via Modelsim and show the results in here. As you can see here is the wave forms.

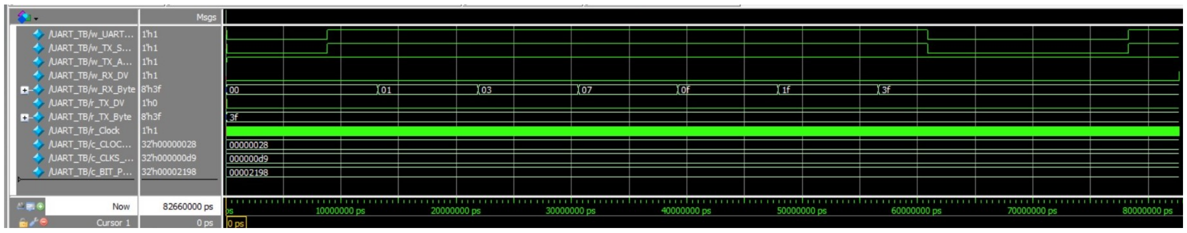
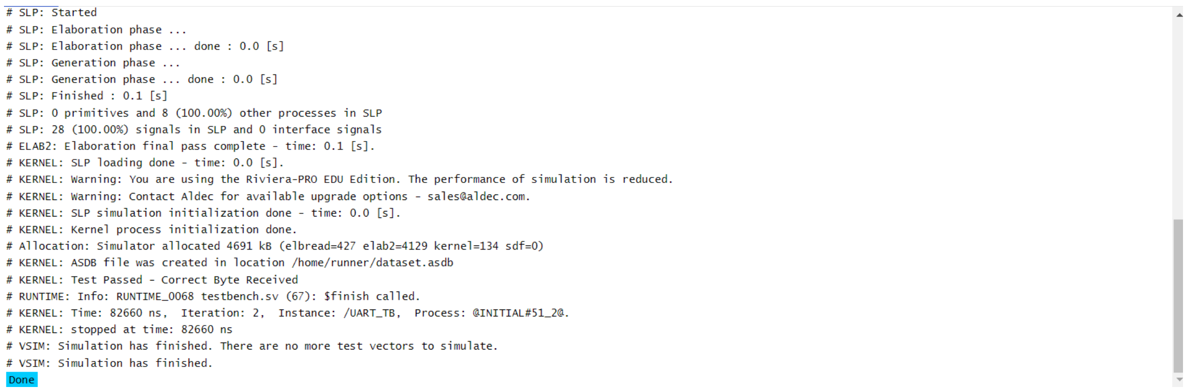


Figure 4:

And this is the results that approves that we runned the code and it worked.



```
# SLP: Started
# SLP: Elaboration phase ...
# SLP: Elaboration phase ... done : 0.0 [s]
# SLP: Generation phase ...
# SLP: Generation phase ... done : 0.0 [s]
# SLP: Finished : 0.1 [s]
# SLP: 0 primitives and 8 (100.00%) other processes in SLP
# SLP: 28 (100.00%) signals in SLP and 0 interface signals
# ELAB2: Elaboration final pass complete - time: 0.1 [s].
# KERNEL: SLP loading done - time: 0.0 [s].
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 4691 kB (elbread=427 elab2=4129 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: Test Passed - Correct Byte Received
# RUNTIME: Info: RUNTIME_0068 testbench.sv (67): $finish called.
# KERNEL: Time: 82660 ns,  Iteration: 2,  Instance: /UART_TB,  Process: @INITIAL#51_2@.
# KERNEL: stopped at time: 82660 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Figure 5: