ABSTRACT

The Universal Asynchronous Receiver-Transmitter (UART) protocol is a widely used serial communication protocol that enables asynchronous, full-duplex communication between digital devices. UART transmits data bit-by-bit over a communication channel, allowing reliable serial data transfer without the need for a separate clock signal. It consists of two main components: a transmitter, which converts parallel data into serial form, and a receiver, which reconverts serial data into parallel form. Data transmission through UART is characterized by its start, stop, and optional parity bits, which frame the data, ensure synchronization, and add a layer of error checking. Commonly found in embedded systems, UART is integral to communications between microcontrollers, sensors, and peripheral devices due to its simplicity, flexibility, and compatibility with a range of baud rates and data packet structures. Despite its relatively low data rate compared to other serial protocols, such as SPI and I2C, UART remains popular in low-speed, low-power applications. This abstract provides an overview of UART's functionality, data structure, and key advantages and limitations in modern communication systems.

TABLE OF CONTENTS

CHAPTER -1
INTRODUCTION

1.1 BACKGROUND

The Universal Asynchronous Receiver-Transmitter (UART) protocol is one of the oldest and simplest communication protocols, designed to enable serial data exchange between digital devices. Its origins can be traced back to the early development of telecommunication and computing systems, where there was a need for a standardized method to transmit and receive data over limited physical connections. As computing evolved, UART quickly became a popular choice in serial communications, providing a straightforward solution for data transfer without requiring synchronous clock signals.

In UART communication, data is transferred asynchronously, meaning that each byte of data is sent independently with its own framing, rather than relying on a shared clock to synchronize data bits. The fundamental structure of UART communication includes start and stop bits framing each byte, allowing devices to recognize the beginning and end of a data packet. The inclusion of optional parity bits also facilitates basic error checking, ensuring the data's integrity during transmission.

1.2 OBJECTIVE

The primary objective of the Universal Asynchronous Receiver-Transmitter (UART) protocol is to facilitate simple, reliable, and asynchronous serial communication between digital devices, such as microcontrollers, computers, and various peripherals. UART's main goals include:

1. Enable Asynchronous Communication:
UART allows data transmission without a shared clock signal. By using start, stop, and optional parity bits, UART synchronizes data at the byte level, making it suitable for applications where a synchronous clock is unnecessary or impractical.

2. Simplify Data Transfer:
UART converts parallel data from the transmitting device into a serial stream, then reconverts it back to parallel data at the receiving end. This simplicity reduces the number of wires needed for data transmission, using only transmit (TX) and receive (RX) lines for full-duplex communication.

CHAPTER -2
EXISTING METHOD

2.1 EXISTING METHOD

The existing method of the Universal Asynchronous Receiver-Transmitter (UART) protocol relies on a simple asynchronous serial communication technique to send and receive data between devices without the need for a shared clock signal. This method is well-suited for short-range, low-speed, point-to-point communication and is widely implemented in microcontrollers and various peripheral devices. Here’s how the current UART method operates:

UART communication is asynchronous, meaning it does not require a synchronized clock signal between the transmitter and receiver. Instead, it uses start and stop bits to indicate the beginning and end of each data packet (or frame). This framing allows devices to self-synchronize at the byte level, making UART suitable for applications where clock synchronization is challenging.

The optional parity bit enables simple error detection, where data integrity can be checked based on even or odd parity. If the parity bit does not match the expected parity, the receiver can flag the data as erroneous. However, UART does not provide built-in error correction, so it’s typically limited to scenarios with minimal interference.

UART supports full-duplex communication by using separate lines for transmitting (TX) and receiving (RX) data. This setup allows data to flow simultaneously in both directions, enhancing communication efficiency for bidirectional data exchange.

The existing Universal Asynchronous Receiver-Transmitter (UART) protocol has a long history in the realm of digital communication due to its simplicity, reliability, and adaptability to numerous embedded systems.

1. Detailed Byte Framing Mechanism
UART transmits data in discrete packets framed by start and stop bits, enabling the receiver to distinguish individual bytes without a synchronized clock. This framing includes:

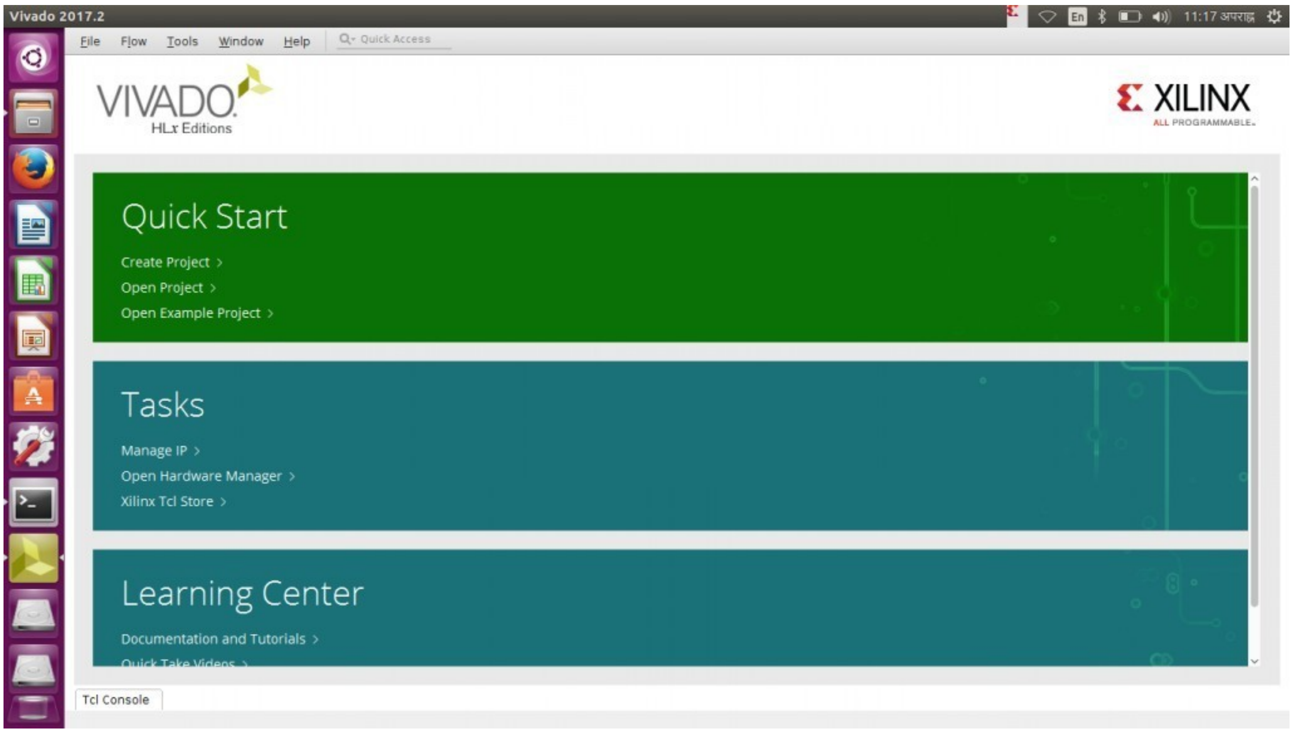
- Start Bit: UART initiates each transmission with a start bit, which pulls the line low and signals the start of a new data byte.
- Data Bits: The data is divided into bits, typically configurable from 5 to 9, though 8-bit data frames are most common in modern implementations.
- Parity Bit: If parity is enabled, UART adds an extra bit for error detection. Parity can be set to even, odd, or none, giving the system flexibility in its error-checking approach.
- Stop Bit(s): One or two stop bits are appended to indicate the end of a data frame. The line is held high during this bit, allowing the receiver time to process the byte and prepare for the next.

2. Baud Rate Synchronization and Compatibility
Baud rate defines the speed of communication and is typically standardized in multiples, such as 9600, 115200, etc. Both devices must be configured with the same baud rate to ensure proper communication. Although UART lacks a shared clock, setting a pre-defined baud rate allows both ends to independently maintain timing alignment within each frame. .

CHAPTER -3
SOFTWARE USED

3.1 XILINX VIVADO

Xilinx Vivado is an advanced FPGA (Field Programmable Gate Array) design and analysis suite developed by **Xilinx** (now a part of AMD). It provides a comprehensive toolset for designing, simulating, and programming FPGAs, SoCs (System on Chips), and other hardware programmable devices. Vivado Design Suite is specifically built to replace the older ISE (Integrated Software Environment) and to support the newer Xilinx FPGA architectures, especially the 7-series devices and beyond (e.g., UltraScale, UltraScale+, and Versal platforms).



Key Features of Xilinx Vivado:

1. **RTL Design and IP Integration**:
   - Vivado supports designing using **HDL** languages like **VHDL**, **Verilog**, and **SystemVerilog**.
   - It includes an **IP Integrator** that allows users to connect pre-built IP cores, including Xilinx's own IP, such as DSP cores, memory controllers, AXI bus interfaces, etc. The tool simplifies the integration of custom IP blocks with drag-and-drop interfaces.
2. **Block Design Tool**:
   - The **Block Design Tool** (BDT) enables users to visually design their hardware by connecting pre-verified IP blocks, similar to a schematic diagram. It's very useful for designing complex systems like processors or SoC designs with peripherals and interconnects.
3. **Vivado High-Level Synthesis (HLS)**:

- High-Level Synthesis allows you to write C, C++, or SystemC code and convert it into hardware description language (HDL) for implementation in an FPGA. HLS enables quicker design exploration, validation, and development by abstracting away some of the low-level design complexity.

4. **Comprehensive Simulation and Debugging**:
    - **Vivado Simulator** supports detailed behavioral, post-synthesis, and post-implementation simulation, allowing users to verify and debug the behavior of their design before programming the hardware.
    - The **Integrated Logic Analyzer (ILA)** allows for on-chip debugging by inserting probes in the design that can capture and analyze internal signals in real-time.
5. **Synthesis and Implementation**:
    - **Vivado Synthesis** translates the high-level RTL design into a netlist of logic gates and performs optimizations for the specific Xilinx FPGA target architecture.
    - The **Implementation** stage handles placement and routing, ensuring that the design is efficiently mapped to the FPGA's logic elements, DSP blocks, BRAM, and interconnects.
6. **Tcl-based Scripting and Automation**:
    - Vivado uses **Tcl (Tool Command Language)**for automation, enabling designers to run repetitive tasks through scripts, automate design flows, and customize processes like project creation, synthesis, and bitstream generation.
7. **Vivado Design Checkpoints (DCP)**:
    - A **Design Checkpoint** is a snapshot of a partially implemented design. It allows users to save, review, and resume work from a specific point in the design flow, especially useful in large designs for modular or team-based development.
8. **Support for Xilinx FPGAs**:
    - Vivado supports the latest Xilinx FPGA families, such as:
        - **7-Series**: Artix-7, Kintex-7, Virtex-7

- **Zynq-7000 SoCs**: Combining ARM processors with programmable logic
- **UltraScale**: High-performance, low-power FPGAs
- **UltraScale+**: Advanced FPGAs and SoCs, targeting AI and 5G applications
- **Versal ACAP**: Adaptive Compute Acceleration Platforms, which combine programmable logic, CPUs, and specialized acceleration engines.

9. **Power Optimization**:
   - Vivado provides power analysis tools to estimate and optimize the power consumption of FPGA designs. This is particularly useful for power-sensitive applications like mobile devices or data centers.
10. **Partial Reconfiguration**:
    - Vivado supports **partial reconfiguration**, allowing a portion of the FPGA to be reconfigured while the rest of the FPGA remains operational. This is useful for applications that require hardware changes without powering down the entire system.

Vivado Design Flow:

The typical Vivado design flow includes the following steps:

1. **Create a New Project**:
   - Define the target FPGA device and set up the working environment.
2. **Design Entry**:
   - Design using HDL (Verilog, VHDL, or SystemVerilog) or use the **Block Design** tool for IP-based integration.
   - Alternatively, use **Vivado HLS** for high-level C/C++-based design.
3. **Synthesis**:
   - Convert the HDL or Block Design into a netlist format, optimize it, and check for design rules.
4. **Implementation**:
   - Place and route the design on the FPGA to map the logic, routing resources, and I/O pins.

5. **Simulation and Debugging**:
    - Simulate the design using the built-in simulator or third-party simulators like ModelSim.
    - Debug using tools like the **Integrated Logic Analyzer (ILA)** or **Virtual I/O (VIO)** to capture and analyze signals in real-time.
6. **Generate Bitstream**:
    - Create a bitstream that can be used to program the FPGA hardware.
7. **Programming and Configuration**:
    - Load the generated bitstream onto the FPGA or SoC device and verify its functionality.
8. **Post-Implementation**:
    - Perform timing analysis, power estimation, and any further optimizations.

Vivado Editions:

- **Vivado WebPACK**: A free version with limited functionality, supporting lower-end devices like the Artix-7 and Spartan series.
- **Vivado Design Edition**: A paid version with additional features such as advanced synthesis and implementation tools.
- **Vivado System Edition**: The most advanced edition, supporting all high-end features, including System Generator, HLS, and complete IP support.

Applications of Vivado:

- **Embedded Systems**: Using Zynq SoCs to create custom embedded systems with hardware and software co-design.
- **AI and Machine Learning**: Implementing AI inference engines on FPGAs using high-performance features like DSP slices and HBM memory.
- **Telecommunications**: Designing custom FPGA-based systems for 5G, optical networks, and high-speed data communications.
- **DSP and Signal Processing**: Optimizing signal processing algorithms on FPGAs for applications like radar, medical imaging, and video processing.

H

CHAPTER - 4
WORKING PRINCIPLE

4.1 Working Principle of a UART Protocol :

Certainly! Here’s a deeper dive into the functionality and operation of UART, including data framing, flow control, and how UART handles data synchronization.

Detailed Working of UART

1. Data Framing:

UART frames data in a specific structure to ensure both ends can communicate effectively. The typical frame structure consists of:
Start Bit: Indicates the beginning of data transmission. It’s usually a low signal (0) that tells the receiver that data is incoming.
Data Bits: The actual data, which can be 5 to 9 bits in length (usually 8 bits). This data is sent bit by bit in serial form.
Parity Bit (optional): Used for basic error checking. It can be set to odd, even, or no parity, depending on the error-checking requirement.
Stop Bits: Marks the end of a data packet. It’s usually one or two bits and is set to high (1), indicating the end of transmission.
Each byte of data is packaged in this frame format before it is sent, ensuring proper start, data integrity, and termination.

2. Baud Rate and Synchronization:

UART is asynchronous, meaning it doesn’t have a shared clock signal. Therefore, both devices must agree on a common baud rate, which defines the speed at which data is sent and received. Baud rate (in bits per second, or bps) determines how long each bit is transmitted. Common baud rates are 9600, 19200, 115200 bps, etc.

3. Flow Control:

Flow control ensures data is transmitted and received without overloading the receiving device. UART supports:

4. Error Detection:

Basic error checking is done with the parity bit. In case the parity doesn’t match the expected value, the UART will mark the received data as erroneous.
However, UART doesn’t support advanced error correction, so it’s mainly suitable for relatively short distances or applications where high reliability isn’t a strict requirement.

5. UART Communication Types:

UART supports both full-duplex and half-duplex modes:
Full-Duplex: Data can be transmitted and received simultaneously, using separate lines for TX and RX.

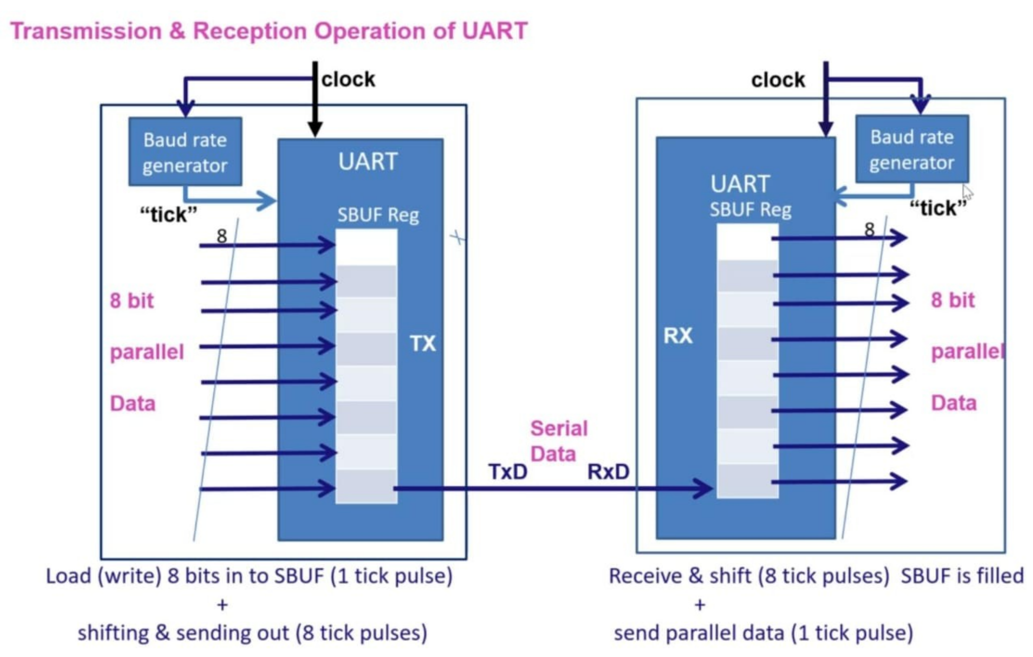
- Half-Duplex: Data can be transmitted or received one at a time, suitable for simple applications where simultaneous data transfer isn’t necessary.

Example UART Communication Flow

1. Initialization: Both UARTs (transmitter and receiver) are set up with the same baud rate, data length, parity, and stop bits configuration.

2. Start Transmission: The transmitter UART pulls the TX line low to signal the start bit, followed by sending each data bit at the defined baud rate.

3. Error Checking: If parity is enabled, the receiver checks the received data against the parity bit for errors. If an error is detected, it may request retransmission (if flow control is implemented) or log the error.


Transmission & Reception Operation of UART
clock
Baud rate generator
UART
"tick"
SBUF Reg
8
8 bit
parallel
Data
TX
Serial
Data
TxD
RxD
RX
Load (write) 8 bits in to SBUF (1 tick pulse)
+
shifting & sending out (8 tick pulses)
Receive & shift (8 tick pulses) SBUF is filled
+
send parallel data (1 tick pulse)

Block Diagram Components Explained :

1. Transmit and Receive Buffers: Hold parallel data before it's converted to serial (for TX) and after it's converted back to parallel (for RX). Buffers help smooth data flow between the microcontroller and the UART.

2. Baud Rate Generator: Maintains the timing for data transmission and reception.

3. Flow Control (RTS/CTS): Provides hardware flow control, ensuring data is sent only when both devices are ready. RTS/CTS lines enable this control.

4. Control Logic: Coordinates all the operations, including start/stop detection, data framing, and error checking.

Key Advantages of UART :

- Simple: UARTs are easy to use for short-range, low-speed communication between two devices.
- Error Checking: Basic error checking using parity bit helps detect simple errors.
- Flexible: UARTs are configurable with different baud rates and data formats (e.g., 7 or 8 data bits,   or 2 stop bits).

Limitations of UART :

- Limited Distance: UART is generally used for short-distance communication (a few meters) because longer distances are more susceptible to noise and errors.
- Limited Speed: UART's speed is limited by the baud rate, making it unsuitable for high-speed data transfer.
- Point-to-Point: UART is typically used for point-to-point communication, supporting only two devices per UART line (no multi-device communication).

UARTs are widely used in microcontroller-based systems and embedded devices where reliable, low-speed serial communication is required, such as interfacing sensors, GPS modules, and other peripheral devices.

CHAPTER -5
VERILOG IMPLEMENTATION

5.1 INTRODUCTION TO VERILOG

Verilog is a hardware description language (HDL) used to model electronic systems. It is widely employed in the design and verification of digital circuits at various abstraction levels, from high-level system design to low-level gate design.

Key Features of Verilog

1. **Hardware Modeling**: Verilog allows designers to describe the structure and behavior of digital systems. It can be used for combinational and sequential logic, as well as for describing complex systems.
2. **Simulation**: Verilog supports simulation of hardware designs, enabling verification of functionality before hardware implementation. It allows designers to test and debug their designs in a controlled environment.
3. **Synthesis**: Verilog can be synthesized into physical hardware (like FPGAs and ASICs). The code can be transformed into a netlist of gates and flip-flops that can be implemented on silicon.

Simulation and Synthesis Tools

- **Simulation Tools**: Tools like ModelSim and Vivado are used to simulate Verilog code, allowing for functional verification before synthesis.
- **Synthesis Tools**: Tools like Xilinx ISE and Synopsys Design Compiler convert Verilog code into gate-level implementations.

Applications of Verilog

1. **Digital Circuit Design**: Used in designing microcontrollers, FPGAs, and ASICs.
2. **System on Chip (SoC) Design**: Facilitates the integration of various subsystems on a single chip.
3. **Testbench Development**: Allows for the creation of testbenches to verify the functionality of designs.
4. **Educational Use**: Widely used in academia to teach digital design concepts.

4. **Concurrency**: Verilog is inherently concurrent, reflecting the parallel nature of hardware. Multiple processes can run simultaneously, making it suitable for modeling complex systems.
5. **Hierarchical Design**: Verilog supports hierarchical design, allowing designers to build complex systems from simpler modules, facilitating easier management and reuse of code.

Basic Syntax and Constructs

1. **Modules**: The basic building blocks in Verilog are modules. Each module can represent a piece of hardware, a system, or a testbench.

```
verilog
Copy code
module my_module (input a, input b, output c);
    assign c = a & b; // AND operation
endmodule
```

2. **Data Types**:
    - **reg**: Represents storage elements (used in procedural blocks).
    - **integer**: Used for integer variables.
    - **real**: Represents real numbers.
3. **Operators**: Verilog supports various operators for arithmetic, logical, relational, and bitwise operations.
4. **Procedural Blocks**: `always` and `initial` blocks define behavior.

```
verilog
Copy code
initial begin
    // Initialization code
end

always @(posedge clk) begin
    // Sequential logic code
end
```

5. **Control Statements**: Verilog supports if-else statements, case statements, and loops.

```
verilog
Copy code
always @(a) begin
    if (a ==    ) begin
        // Do something
    end else begin
        // Do something else
    end
end
```

Simulation and Synthesis Tools :

- **Simulation Tools**: Tools like ModelSim and Vivado are used to simulate Verilog code, allowing for functional verification before synthesis.
- **Synthesis Tools**: Tools like Xilinx ISE and Synopsys Design Compiler convert Verilog code into gate-level implementations.

Applications of Verilog :

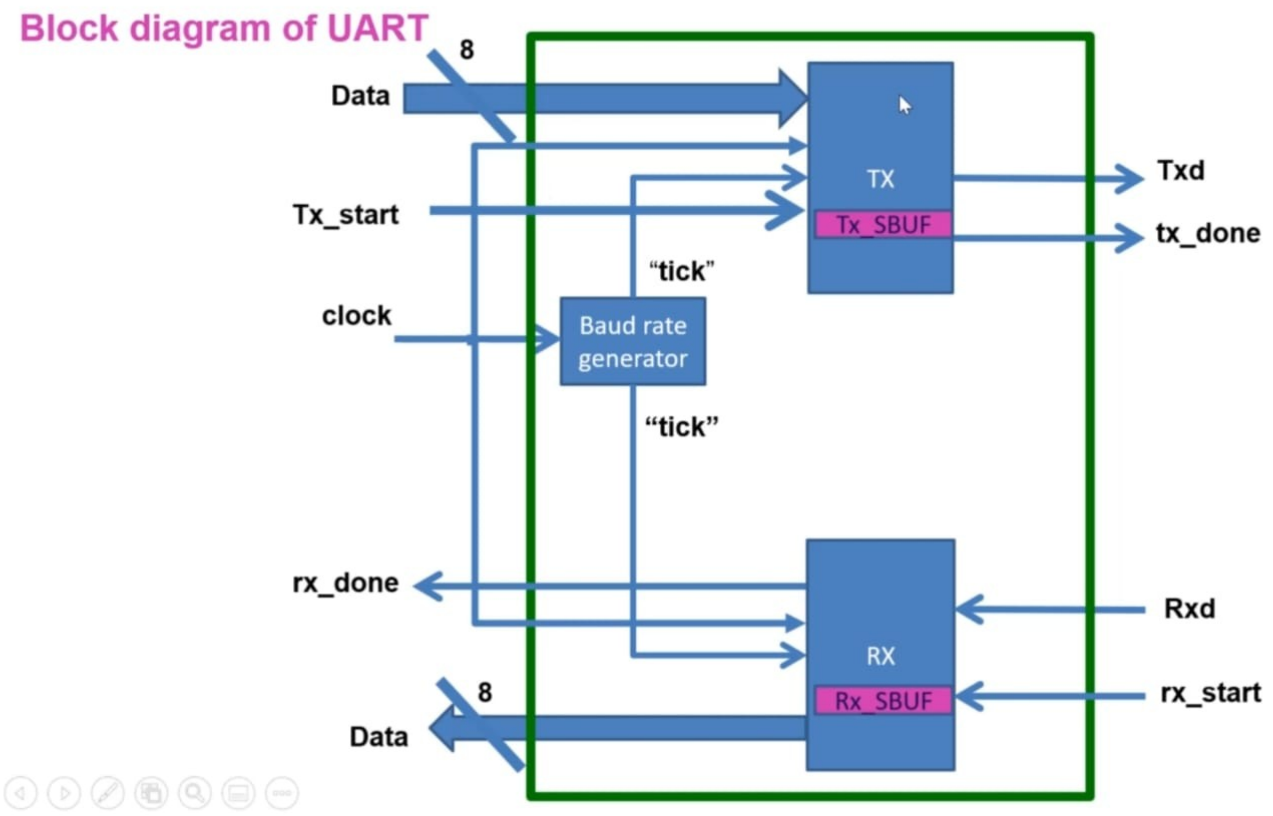
1. **Digital Circuit Design**: Used in designing microcontrollers, FPGAs, and ASICs.
2. **System on Chip (SoC) Design**: Facilitates the integration of various subsystems on a single chip.
3. **Testbench Development**: Allows for the creation of testbenches to verify the functionality of designs.
4. **Educational Use**: Widely used in academia to teach digital design concepts.

5.2 UART ARCHITECTURE

The architecture of the Universal Asynchronous Receiver-Transmitter (UART) protocol is designed for simple, efficient serial communication, facilitating data exchange between digital devices with minimal hardware requirements. UART's architecture includes two primary components: the transmitter and the receiver, which handle the conversion of data from parallel to serial formats and vice versa.

Control registers are another critical part of the UART architecture. These registers allow configuration of baud rate, data format (number of data bits, stop bits, and parity), and other operational parameters. They also provide status flags for error conditions such as framing errors (when the expected stop bit is not detected), parity errors, and buffer overflow errors.

4.2 Block Diagram of an UART Protocol :



Key Advantages of the Architecture :

- **Speed**: The parallel computation of partial products allows for faster multiplication compared to traditional methods.
- **Efficiency**: The use of Vedic mathematics reduces the number of required operations and optimizes the design for hardware implementation.
- **Scalability**: The architecture can be easily scaled for larger bit-width multiplications by applying the same principles.

CHAPTER -6
SOURCE CODE

6.1 **TRANSMITTER CODE** :

```
`define idle 3'b000
`define start 3'b001
`define trans 3'b010
`define stop 3'b011
module uart_tx(clk,reset,tx_start,data_in,txd,tx_done);

    input clk,reset,tx_start;
    input [7:0]data_in;
    output txd,tx_done;

    wire clk,reset,tx_start;
    wire[7:0]data_in;
    wire txd;
    reg tx_done;
    reg[2:0]ps,ns;
    reg[7:0]sbuf_reg,sbuf_next;
    reg[0:2]count_reg,count_next;
    reg tx_reg,tx_next;

    wire tick;
    reg[9:0]q;
    wire [9:0]q_next;

    //memory for FSM
   always @(posedge clk)
    begin
     if(reset)
      begin
       ps=`idle;
       sbuf_reg=0;
       count_reg=0;
       tx_reg=0;
      end
     else
      begin
        ps=ns;
        sbuf_reg=sbuf_next;
        count_reg=count_next;
        tx_reg=tx_next;
      end
    end
```

```
//next state block and output block of FSM
 always@(*)
  begin
    ns=ps;
    sbuf_next=sbuf_reg;
    count_next=count_reg;
    tx_next=tx_reg;
   case(ps)
      `idle:
       begin
        $display($time,"------idle state------");
        tx_next=  ;
        tx_done=0;
        if(tx_start ==   )
         begin
           ns=`start;
           //sbuf_reg=data_in;
             $display($time,"------idle to start------");
         end
        end
       `start:
         begin
          tx_next=0;//start bit
          $display($time,"------start state------");
          if(tick)
           begin
            sbuf_next= data_in;
            count_next=0;
            ns=`trans;
            $display($time,"------start to trans------");
           end
        End
     `trans:
       begin
        tx_next=sbuf_reg[0];
        $display($time,"----- trans state------");
        if(tick)
         begin
          sbuf_next=sbuf_reg>>  ;
          if(count_reg==7)
           begin
            ns=`stop;
            $display($time,"------trans to stop------");
           end
        else
         count_next =count_reg+  ;
        end
       end
     `stop:
       begin
        tx_next=  ;//stop bit
        $display($time,"------stop state------");
        if(tick)
         begin
          tx_done=  ;
          ns=`idle;
         end
       end
      default:ns=`idle;
    endcase
   end
  assign txd=tx_reg;
  //------------------------
  //---------baud rate generator----------
  always@(posedge clk)
        begin
         if(reset)
          q=0;
         else
          q=q_next;
        end
    assign q_next=(q==500)?0:q+  ;
    assign tick=(q==500)?  :0;

  //--------------------
endmodule
```

TEST BENCH CODE:

```
 module uart_tx_tb;
  reg clk,reset,tx_start;
  reg [7:0]data_in;
  wire txd,tx_done;
  uart_tx UUT(clk,reset,tx_start,data_in,txd,tx_done);

  //instantation
   initial begin
    clk=0;
    reset=  ;//if
    #20 reset=0;//else
    #40 tx_start =  ;
    #20 $display($time,"---------start process");
     data_in = 8'haa; //1010_1010
     #5000 tx_start =0;
     #200210 $stop;
   end
   always #10 clk=~clk;//T=20ns,f=  /T=50Mhz
   initial begin
     $dumpfile("dump.vcd");//waveform generator
     $dumpvars(2,uart_tx_tb);
  end
endmodule
```

6.2 **RECEIVER CODE :**

```
`define idle 3'b000
`define start 3'b001
`define trans 3'b010
`define stop 3'b011

module uart_rx(clk,reset,data_out,rxd,rx_done);
    input clk,reset,rxd;
    output rx_done;
    output [7:0]data_out;

    reg[2:0]ps,ns;
    reg[7:0]sbuf_reg,sbuf_next;
    reg[2:0]count_reg,count_next;

     wire tick;
     reg[7:0]q;
    wire [7:0]q_next;

    //memory block FSM
    always@(posedge clk)
     Begin
     if(reset)
      begin
       ps=`idle;
      sbuf_reg=0;
      count_reg=0;
      End
```

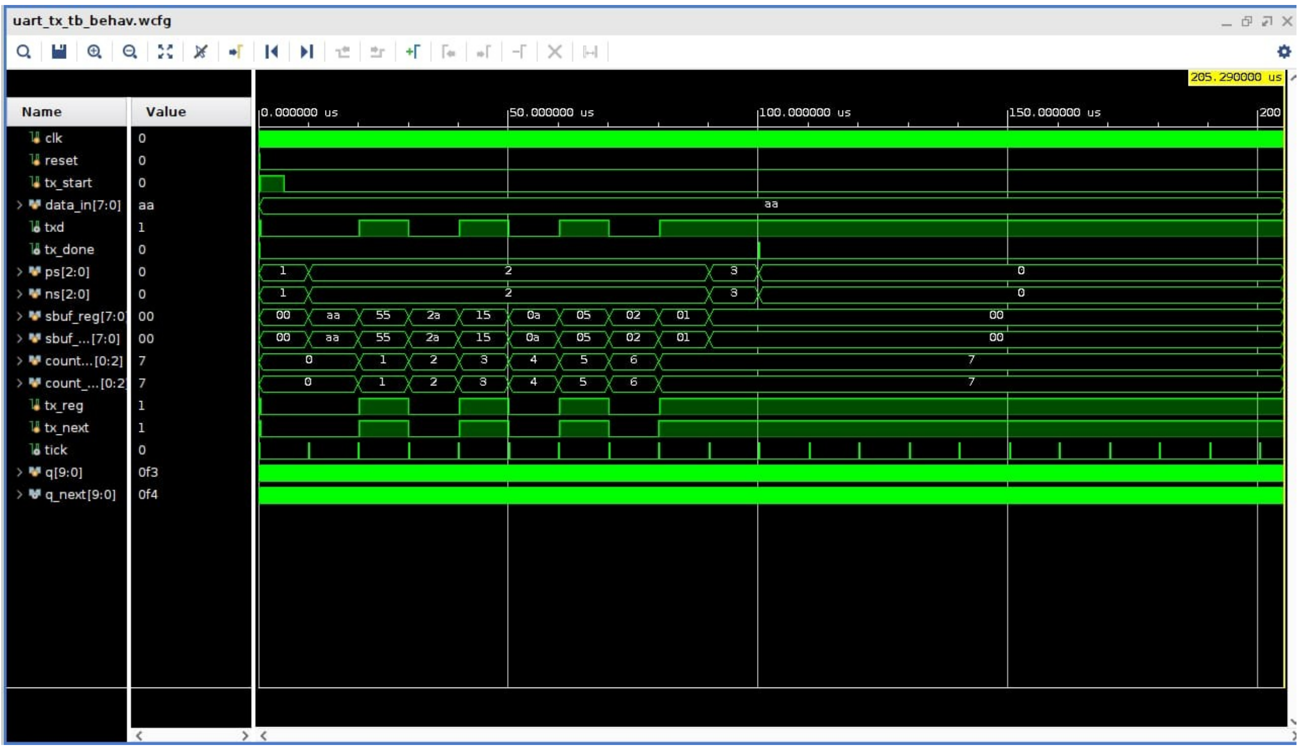
```
  else
   begin
     ps=ns;
    sbuf_reg=sbuf_next;
    count_reg=count_next;
   end
    End
//next state block & block  of FSM
always@(*)
 begin
  ns=ps;
  sbuf_next=sbuf_reg;
  count_next=count_reg;

 case(ps)
  `idle:
    begin
      $display($time,"------idle state------");
      if(rxd==0)
        begin
         ns=`start;
         $display($time,"------idle to start------");
        end
     end
      `start:
        begin
          $display($time,"------start state------");
          if(tick)
          begin
           ns=`trans;
             $display($time,"------start to trans------");
             count_next=0;
          End
        end
       `trans:
        begin
          $display($time,"------trans state------");

         if(tick)
          begin
           sbuf_next={rxd,sbuf_reg[7:  ]};
           if(count_reg==7)
          begin
           ns=`stop;
             $display($time,"------trans to stop------");
          End
        else
          count_next=count_reg+  ;
         end
     End
   `stop:
      begin
        $display($time,"------stop state------");
        if(tick)
          begin
            ns=`idle;
            $display($time,"------stop to idle------"
        end
      end
      default:ns=`idle;
     endcase
 end
 assign data_out =sbuf_reg;
 //------end Rx
 //------Baud generator-------
  always@(posedge clk)
     begin
      if(reset)
       q=0;
      else
       q=q_next;
     end
  assign q_next=(q==200)?0:q+  ;
  assign tick=(q==200)?  :0;
endmodule
```

TEST BENCH CODE:

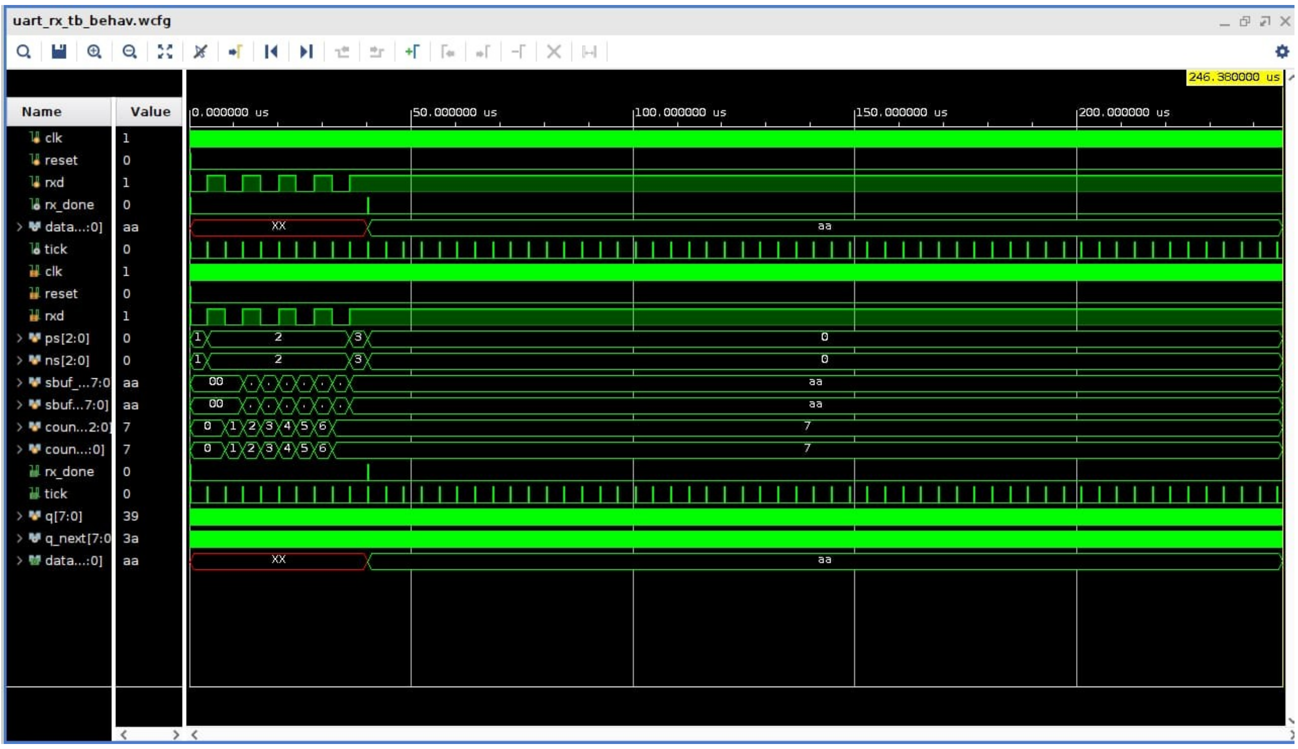
```
module uart_rx_tb;
 reg clk,reset,rxd;
 wire rx_done;
 wire[7:0]data_out;
 wire tick;
uart_rx UUT(clk,reset,data_out,rxd,rx_done);
 //instantation
 always #10 clk=~clk;//T=20ns,f=50Mhz
 initial begin
  clk=0;
  reset=  ;//if
  #11 reset=0;//else
  #20 rxd =0;//start
   $display($time,"-----start bit process-------");
   rdata(  );
   $display($time,"------rdata   -----");
   rdata(0);
   $display($time,"------rdata 2-----");
   rdata(  );
   $display($time,"------rdata 3-----");
   rdata(0);
   $display($time,"------rdata 4-----");
   rdata(  );
   $display($time,"------rdata 5-----");
   rdata(0);
   $display($time,"------rdata 6-----");
   rdata(  );
   $display($time,"------rdata 7-----");
   rdata(0);
   $display($time,"------rdata 8-----");
  rdata(  );
   $display($time,"------stop bit-----");
   #10000  $display($time,"------dataout=%h-----",data_out);
   #200210 $stop;
  end
  task rdata;
   input inp;
    begin
     @(posedge tick)
      begin
       rxd=inp;
          $display($time,"------supply data------");
      end
 end
 endtask
 initial begin
 $dumpfile("dump.vcd");
 $dumpvars(2,uart_rx_tb);
 end
endmodule
```

CHAPTER -7
RESULT AND ANALYSIS

7.1 TRANSMITTER OUTPUT :



7.2 RECEIVER OUTPUT :



7.2 CONCLUSION

The UART (Universal Asynchronous Receiver/Transmitter) protocol is a widely used communication method for serial data transmission, especially in embedded systems and computer peripherals. Operating asynchronously, UART does not require a shared clock between devices. Instead, the sender and receiver agree on parameters like baud rate and data format, allowing data interpretation without constant synchronization. This simplicity makes UART cost-effective and easy to implement, relying on just two primary wires: TX (transmit) and RX (receive), with additional optional lines for flow control, such as RTS and CTS, to enhance reliability. Data is sent in packets containing start bits, data bits, and stop bits, which help organize the data and indicate where transmission begins and ends. However, UART has limitations, including relatively low data transfer speeds compared to other protocols like SPI or I2C, and it only supports point-to-point (one-to-one) communication, making it less suitable for high-speed or long-range applications. Despite its basic error detection (such as optional parity bits), UART remains popular due to its simplicity and effectiveness for short-range, moderate-speed communication in embedded systems.