SerDes VIP

Table Of Contents

Table Of Figures

1. Introduction to SerDes

1.1 Introduction

The **Serializer/Deserializer (SERDES)** is a **high-speed data interface** used to convert data between **parallel and serial formats** for transmission and reception. In modern SoCs and communication systems, SERDES plays a critical role in **reducing the number of physical I/O lines** by serializing wide parallel buses, transmitting them over fewer wires, and then deserializing the data at the receiver end. A typical SERDES system includes a **transmitter (serializer)** that **converts N-bit parallel data into a serial stream**, and **a receiver (deserializer) that reconstructs the original parallel data from the received serial bits.** This is often synchronized across two different clock domains: **a parallel clock for system-level data and a high-frequency serial clock** for transmission.

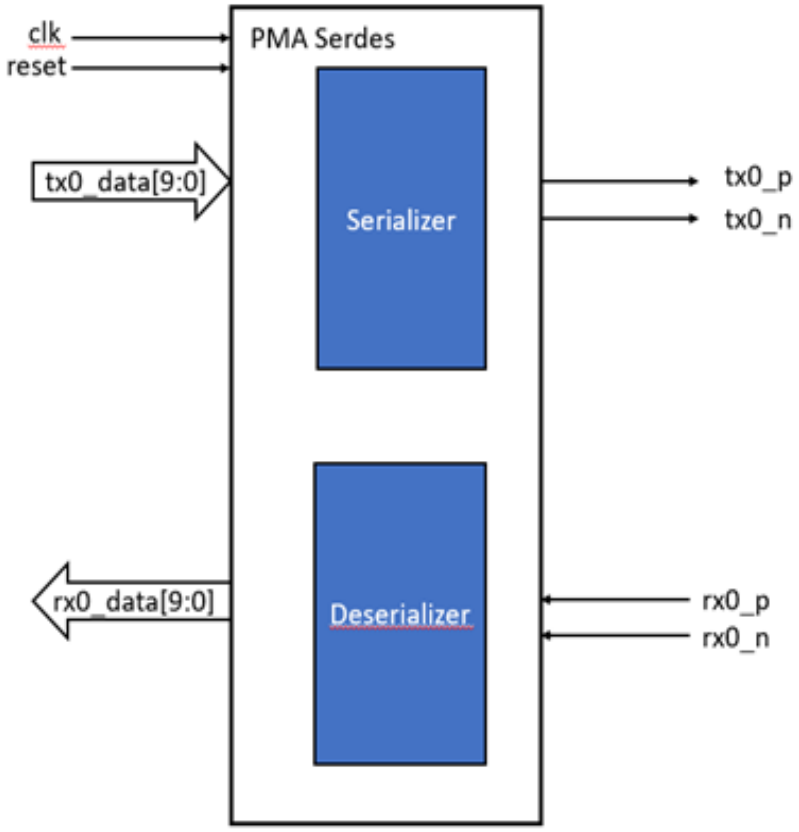

Figure 1 : SerDes Block Diagram

Serdes is a part of **PMA**. It has **2 blocks serializer and de-serializer**. **Serializer** will do the conversion of **parallel data to serial data** and **de-serializer** will do the conversion of **serial data into parallel data**. For Tx, the input data will be **10 bits parallel for single lane**, the **serializer will convert into serial data** and transmit on differential signal. **The baud rate for parallel and serial will remain same based on the speed selection**. Similarly for RX, the input data for PMA will be differential **RX signals** and **de-serializer will convert into the 10 bits parallel data**, here also the baud rate will remain same on both sides. **The clk and reset will provide as input to PMA.**

1.2 Configurations

Here the supported configuration is **speed selection from 1G to 10G**, the variable no. of lanes, variable parallel data selection, **loopback support** in which rx data will not be captured, **low power mode in which the data and clk will be shut down** so the lines will be on high impedance state. **Here the differential signal indicates the polarity inverse of each other** i.e. **on p and n signals on serial side the data will be compliment of each other.**

1.3 Features of SerDes

- High-speed serial operation
- Differential serial I/O
- Support for loopback
- Support for lane polarity inversion
- Low Power Mode
- Speed selection

2. Testbench Architecture Of SerDes

2.1 Testbench Architecture

The SERDES VIP testbench is designed with a layered and modular architecture to provide thorough verification of **both parallel and serial data paths of the DUT**. The testbench consists of **four primary agents**: two for the parallel interface (one active and one passive) and two for the serial interface (one active and one passive).

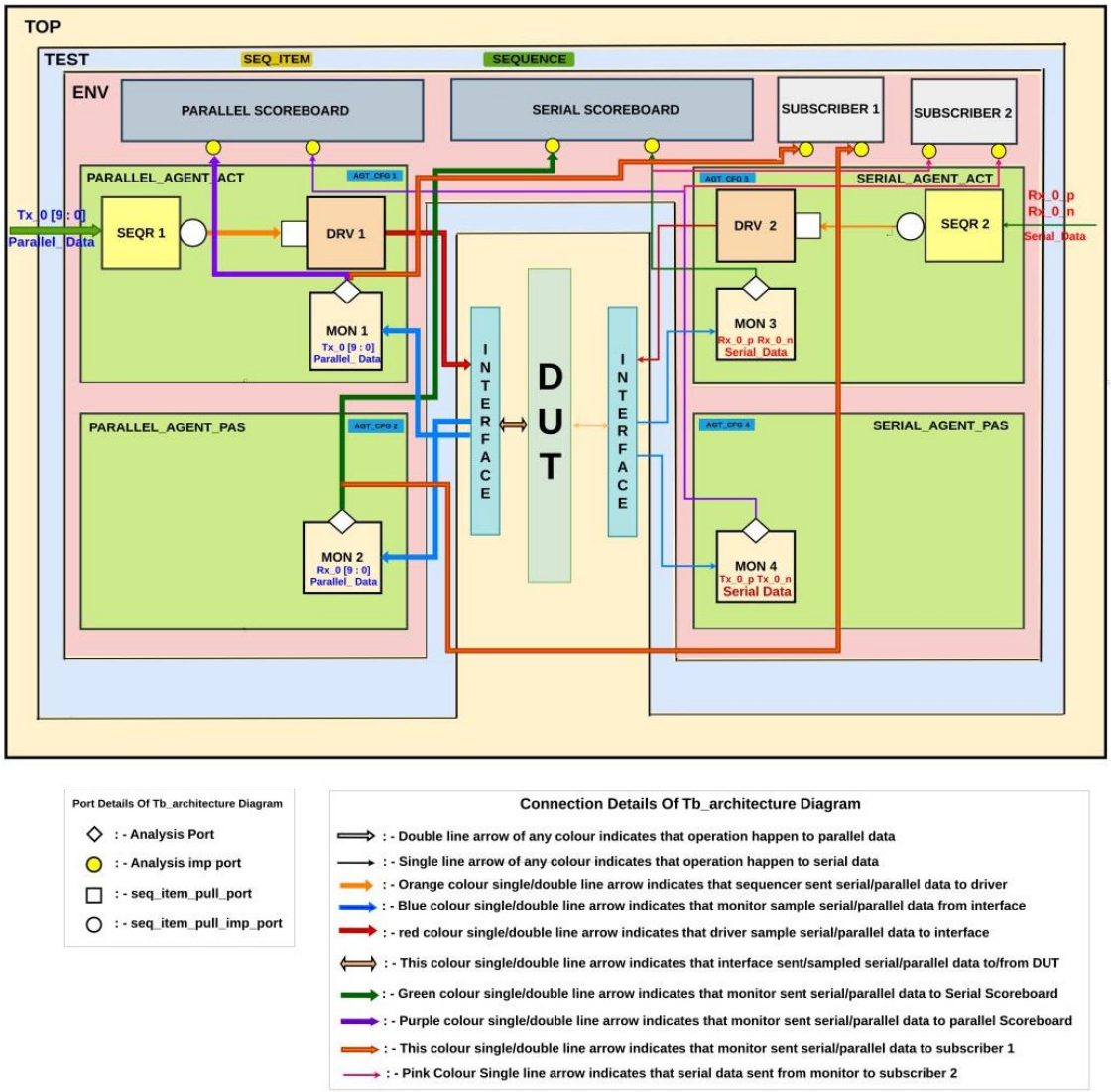

Figure 2 : Testbench Architecture Of SerDes

2.2 Components of Testbench

- Top : The **TOP** block is the highest level of the verification hierarchy and coordinates the entire verification process. Its responsibilities include:
    - Instantiating the **test**, which in turn creates the environment (ENV).
    - Connecting the DUT with the verification environment through properly configured interfaces.
    - Controlling the testbench clock and reset signals, including serial and parallel clocks.
    - Providing DUT-level configuration parameters such as data widths, clock frequencies, and protocol options.

- Test : The test class is the entry point for executing any scenario. It is responsible for:
    - Instantiating and configuring the environment.
    - Creating and starting test sequences.
    - Overriding factory-registered components if necessary.

- Environment :
    - The environment (env) is a container class derived from uvm_env.
    - It instantiates and connects:
        - Parallel agent (active/passive)
        - Serial agent (active/passive)
        - Scoreboards (for both domains)
        - Subscribers
        - uvm_analysis_port for monitors → scoreboard/subscriber

- Agents : There are 4 agents Instance in SerDes VIP Architecture where two instance are parallel(Active and Passive) and two instance are serial(Active and Passive)
    - Parallel Agent : There are two parallel agents
        - Active Parallel Agent : It drives 10 bit parallel data Tx0 to interface and monitor that 10 bit parallel data using active parallel monitor.
        - Passive Parallel Agent : It monitor only parallel data Rx0 that is converted by deserializer.

    - Serial Agent : There are two serial agents
        - Active Serial Agent : It drives two 1 bit serial signals Rx0_p and Rx0_n to interface and monitor that signals using active serial monitor.
        - Passive Serial Agent : It samples two serial data Tx0_p and Tx0_n which is converted by serializer from Tx0.
- Agent_config : There is one agent_config class that is responsible to configure 4 agents parallel or serial and active or passive.
- Sequencer : There is two instance of sequencer one is known as parallel sequencer which send 10 bit parallel data Tx0 to driver and another one is serial sequencer which is send serial data Rx0_p and Rx0_n to driver
- Driver : There are two instances of driver first is parallel driver and second is serial driver.
    - Parallel Driver : Parallel driver drives 10 bit parallel data Tx0 to interface
    - Serial Driver : Serial driver drives two serial data Rx0_p and Rx0_n to Interface
- Monitor : There are 4 Instances of monitor two is parallel and two is serial
    - Parallel Monitor : It basically monitor parallel data only two parallel monitor in the testbench architecture one is active which monitor Tx0 and one is passive monitor which monitor Rx0.
    - Serial Monitor : It basically monitor serial data only two serial monitor in the testbench architecture one is active and one is passive active only monitors Rx0_p and Rx0_n and passive only monitor Tx0_p and Tx0_n.
- Scoreboard : There are two instances of scoreboard one for Tx side and one for Rx side
    - TX Scoreboard : It takes packet from active parallel monitor and passive serial monitor and compares both data.
    - Rx Scoreboard : It takes packet from passive parallel monitor and active serial monitor and compares that data.

- Interaface : The SERDES interface encapsulates all the necessary DUT I/O signals related to both serial and parallel domains, including:
    - Clocking (parallel_clk and serial_clk)
    - Reset signal (serdes_reset)
    - Parallel data signals
    - Serial data line
    - Modports for agent-specific access control
- Test_config : The test_config class is used to access test properties indirectly for example scoreboard required data of reset signal and transaction count.
- Sequence_Item: The serdes_transaction class extends uvm_sequence_item and acts as a container for a single transaction between the testbench and DUT. It represents one unit of data sent to or received from the DUT.
- Sequence : The **serdes_sequence** class extends uvm_sequence and is used to generate multiple serdes_transaction objects and send them through the sequencer to the driver.

2.3 Tx Flow of Testbench Architecture

The **Transmit (TX) flow** in the SERDES VIP testbench architecture describes how data is generated, driven, and transmitted from the **parallel domain** to the **serial domain** through the serializer block of the DUT. The VIP components collaboratively enable functional verification of this data path.

Sequence Item Creation

- The sequence item (e.g., serdes_transaction) is extended from uvm_sequence_item.
- It contains the Tx0 field representing the 10-bit parallel data to be serialized.
- Randomization or directed values are generated in this transaction.

Sequence Execution

- The serdes_tx_sequence starts on the Parallel sequencer (seqr1).
- The sequence generates and sends one or more serdes_transaction objects using start_item() and finish_item() calls.

Sequencer-Driver Communication

- The parallel sequencer passes each transaction to the TX driver through the seq_item_port.
- The parallel driver receives the transaction using get_next_item() and processes it.

Driver Behaviour

- The parallel driver drives the transaction's parallel data (Tx0) to Interface.
- Using the interface (serdes_interface), it drives parallel 10 bit data to serializer dut
- The transmission of parallel driver is synchronized with parallel clock.

Interface Role

- The serdes_interface binds both the serial and parallel clocks (serial_clk, parallel_clk) and data lines.
- It provides a structured way to connect VIP drivers to the DUT using modports and clocking blocks.

DUT Serializer Function

- The DUT receives the 10-bit parallel input (Tx0) and serializes it into 10 serial bits.
- It transmits one bit per cycle on the serial_data line during each rising edge of the serial_clk.

Monitoring

- The parallel active and passive monitor observes the parallel input (Tx0) and parallel output (Rx0).
- It publishes transactions via an analysis port to the scoreboard and subscribers for checking and logging.

Scoreboarding

- The Tx Scoreboard takes data from Active Parallel Monitor and Passive Serial Monitor.
- Passive Serial Monitor convert that serial data into parallel data and send into the scoreboard.

2.4 Rx Flow of Testbench Architecture

The **Transmit (RX) flow** in the SERDES VIP testbench architecture describes how data is generated, driven, and transmitted from the **serial domain** to the **parallel domain** through the deserializer block of the DUT. The VIP components collaboratively enable functional verification of this data path.

Sequence Item Creation

- The sequence item (e.g., serdes_transaction) is extended from uvm_sequence_item.
- It contains the Rx0_p and Rx0_n field representing the 1 bit serial data.
- Randomization or directed values are generated in this transaction.

Sequence Execution

- The serdes_rx_sequence starts on the Serial sequencer (seqr2).
- The sequence generates and sends one or more serdes_transaction objects using start_item() and finish_item() calls.

Sequencer-Driver Communication

- The serial sequencer passes each transaction to the RX driver through the seq_item_port.
- The serial driver receives the transaction using get_next_item() and processes it.

Driver Behaviour

- The serial driver drives the transaction's serial data (Rx0_p and Rx0_n) to Interface.
- Using the interface (serdes_interface), it drives serial 1 bit data to deserializer dut.
- The transmission of serial driver is synchronized with serial clock.

Interface Role

- The serdes_interface binds both the serial and parallel clocks (serial_clk, parallel_clk) and data lines.
- It provides a structured way to connect VIP drivers to the DUT using modports and clocking blocks.

DUT Deserializer Function

- The DUT receives the 1 bit Serial input (Rx0_p and Rx0_n) and deserializes it into 10 parallel bits.
- It transmits 10 bit per cycle on the parallel_data line during each rising edge of the parallel_clk.

Monitoring

- The serial active and passive monitor observes the serial input (Rx0_p, Rx0_n) and serial output (Tx0_p, Tx0_n).
- It publishes transactions via an analysis port to the scoreboard and subscribers for checking and logging.

Scoreboarding

- The Rx Scoreboard takes data from Passive Parallel Monitor and Active Serial Monitor.
- Active Serial Monitor convert that serial data into parallel data and send into the scoreboard.

3. Testcase Development

3.1 Sanity Testcase Preview

In the **Sanity testcase parallel and serial transaction count transaction count is 1**. One 10 bit parallel data transmitted to dut from parallel driver and ten 1 bit serial data is transmitted from serial driver.

```
// ------------------------------------------------------------------------------------------ //
// This is test class
// This class basically create the environment component and it will create the env and sequence and start to sequence on the sequencer and it also check total number of transactions is equal to scoreboard actual transaction
// ------------------------------------------------------------------------------------------ //

class serdes_test extends uvm_test;

  // Factory registration of test class
  `uvm_component_utils(serdes_test);

  // Properties declaration of test class
  serdes_env env; // Serdes env class instance
  serdes_sequence seq[2]; // Two instance of sequence one for parallel sequencer and one for serial sequencer
  serdes_test_config test_cfg; // Test config class Instance
  int serial_transaction_count; // Serial transaction count
  int parallel_transaction_count; // Parallel transaction count
  real serial_clk_period;
  real drain_time;
  virtual serdes_interface vif; // Virtual Interface handle

 // Constructor of serdes_test class
 function new (string name, uvm_component parent);
  super.new(name, parent);
  test_cfg = serdes_test_config::type_id::create("test_cfg"); // Creation of test_cfg class instance
 endfunction : new

 // Build phase of serdes test class
 virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  // Get interface from tb_top using config db
  if(!uvm_config_db#(virtual serdes_interface)::get(this, "", "vif", vif))
    `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".vif"});

  // Get serial transaction count from tb_top using config db
  if(!uvm_config_db#(int)::get(this, "", "serial_transaction_count", serial_transaction_count))
    `uvm_fatal("NO_SERIAL_TRANSACTION_COUNT",{"Serial Transaction count must be set for: ",get_full_name()});

  // Get parallel transaction count from tb_top using config db
  if(!uvm_config_db#(int)::get(this, "", "parallel_transaction_count", parallel_transaction_count))
    `uvm_fatal("NO_PARALLEL_TRANSACTION_COUNT",{"Parallel transaction count must be set for: ",get_full_name()});

  // Get Serdes Speed from tb_top using config db
  if(!uvm_config_db#(real)::get(this, "", "serial_clk_period", serial_clk_period))
    `uvm_fatal("NO_SERIAL_CLK_PERIOD",{"serial_clk_period must be set for: ",get_full_name()});

  `uvm_info(get_type_name(), $sformatf("serial_clk_period = %f", serial_clk_period), UVM_LOW)
  drain_time = (serial_clk_period * 20) * 1000;
  `uvm_info(get_type_name(), $sformatf("Drain time = %0d ps", drain_time), UVM_LOW)

  parallel_transaction_count = 1;
  serial_transaction_count = 1;
```

```
    // Set serial transaction count for sequence using config db
    uvm_config_db #(int)::set(this, "*", "test_serial_transaction_count", serial_transaction_count);

    // Set serial transaction count for sequence using config db
    uvm_config_db #(int)::set(this, "*", "test_parallel_transaction_count", parallel_transaction_count);

    env = serdes_env::type_id::create("env", this); // Cretion of env class instance
    test_cfg.parallel_transaction_count = parallel_transaction_count;
    test_cfg.serial_transaction_count = serial_transaction_count;
  endfunction : build_phase

  // Connect phase of test class
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
  endfunction : connect_phase

  // End of elaboration phase of test class
  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    `uvm_info("Driver EOE Phase", $sformatf("Inside the EOE Phase of driver class"), UVM_LOW)
    uvm_top.print_topology();
  endfunction : end_of_elaboration_phase

  // Run phase of test class
  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    phase.raise_objection(this); // Objection is raised

    // Foreach loop of sequence instance here the sequence is created and if it is parallel sequence then is_parallel = 1 otherwise is_parallel = 0
    foreach(seq[i]) begin
      seq[i] = serdes_sequence::type_id::create($sformatf("seq[%0d]", i)); // Creation of sequence
      seq[i].is_parallel = env.agt[i*2].agt_cfg.is_parallel; // Is_parallel configuration
      `uvm_info(get_type_name(), $sformatf("seq[%0d].is_parallel = %b | env.agt[%0d].agt_cfg.is_parallel = %b", i,seq[i].is_parallel,i,env.agt[i*2].agt_cfg.is_parallel), UVM_LOW)
    end

    foreach(env.scb[i]) begin
      env.scb[i].test_cfg = test_cfg; // Provide the test config instance to scoreboard
    end

    // Inside the fork join the sequende is started
    fork
      seq[0].start(env.agt[0].seqr); // Sequence is started on sequencer 1
      seq[1].start(env.agt[2].seqr); // Sequence is started on sequencer 2
    join

    phase.phase_done.set_drain_time(this, drain_time); // Drian time
    phase.drop_objection(this); // Objection dropped

  endtask : run_phase

  // Report phase of test
```

```
103   // Report phase of test
104   // Inside the report phase the transaction count is checked and if it is match with actual count of respective scoreboard then testcase is passed otherwise it is display uvm error as testcase is failed
105   virtual function void report_phase(uvm_phase phase);
106     if((parallel_transaction_count+1 == env.scb[0].actual_count) && (serial_transaction_count+1 == env.scb[1].actual_count)) begin
107       `uvm_info("Report_Phase of test", $sformatf("Testcase Passed"), UVM_LOW)
108     end
109
110     else begin
111       `uvm_error("Report phase of test",$sformatf("Testcase Failed"))
112     end
113   endfunction : report_phase
114
115
116 endclass : serdes_test
```

Figure 3 : Sanity Testcase snippet

3.2 Sanity Testcase Explanation

- In the sanity testcase transaction count for serial and parallel is 1.
- Parallel_sequence generate one 10 bit parallel data and send to parallel_sequencer and on the other side Serial_sequence generate ten 1 bit serial data and send to serial_sequencer.
- Parallel_driver and serial driver get their respective data through the port connection between sequencer and driver.
- Parallel_driver drives parallel data on the rising edge of parallel clock if reset is off and serial driver drives serial data on the rising edge of serial clock if reset is off.
- Interface send data to dut for parallel to serial conversion and serial to parallel conversion .
- Serializer Dut convert 10 bit parallel data into 1 bit serial data and send to interface on the rising edge of the clock.
- Deserializer convert ten 1 bit serial data into 10 bit parallel data and gives data at rising edge of the parallel clock.
- After that respective monitor samples the data, parallel monitor samples parallel data and serial monitor samples serial data.
- Serial active and passive monitor convert that data into 10 bit packet and broadcast to analysis port
- Tx and Rx Scoreboard receive that packet and do comparison between expected and actual data and if they match then comparison pass otherwise comparison fail.