ETHERNET
TX - VERIFICATION PLAN

Submitted By
H.NARESH KUMAR
U.SREEJA
MD.ARBAAZ
B.RAJESH

CONTENTS

List of Figures:

CHAPTER-1

INTRODUCTION

Ethernet protocol definition: The most popular and oldest LAN technology is Ethernet Protocol, so it is more frequently used in LAN environments which is used in almost all networks like offices, homes, public places, enterprises, and universities. Ethernet has gained huge popularity because of its maximum rates over longer distances using optical media.

The Ethernet protocol uses a star topology or linear bus which is the foundation of the IEEE 802.3 standard. The main reason to use Ethernet widely is, it is simple to understand, maintain, implement, provides flexibility, and permits less cost network implementation.

1.1.Ethernet Protocol Architecture

In the OSI network model, Ethernet protocol operates at the first two layers like the Physical & the Data Link layers but, Ethernet separates the Data Link layer into two different layers called the Logical Link Control layer & the Medium Access Control layer.

The physical layer in the network mainly focuses on the elements of hardware like repeaters, cables & network interface cards (NIC). For instance, an Ethernet network like 100BaseTX or 10BaseT indicates the cable type that can be used, the length of cables, and the optimal topology.

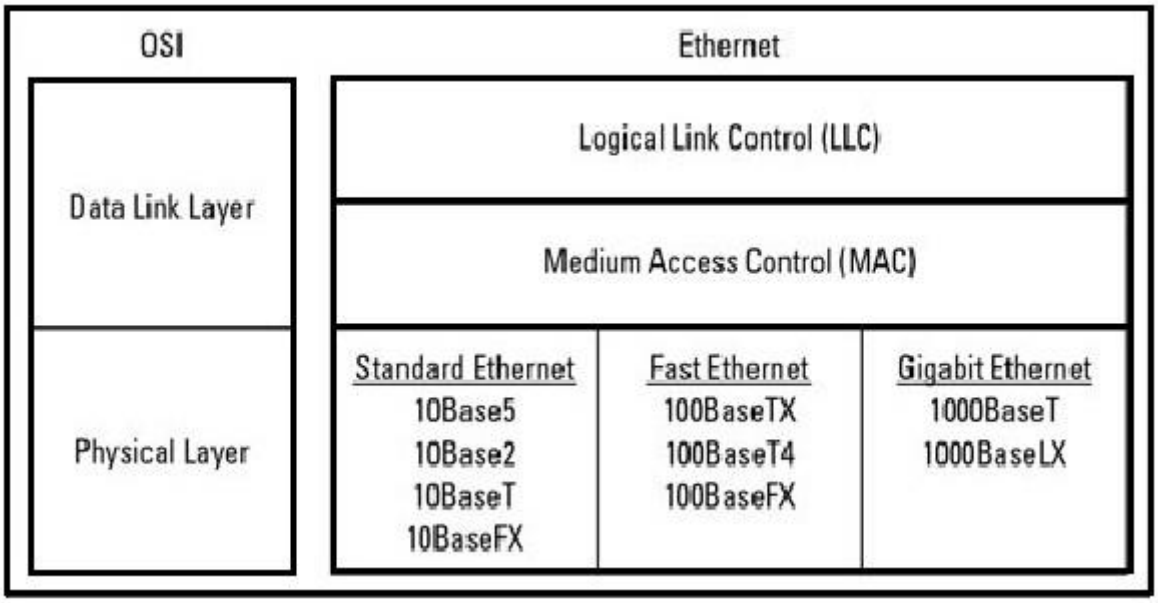


Fig 1.1. Ethernet Protocol Architecture

The data link layer in the network system mainly addresses the way that data packets are transmitted from one type of node to another. Ethernet uses an access method called CSMA/CD (Carrier Sense Multiple Access/Collision Detection). This is a system where every computer listens to the cable before transmitting anything throughout the network.

The two layers in the above ethernet protocol block diagram deal with the physical network structure where the network devices can transmit data from one device to another on a network. Certainly, the most popular set of protocols used for both the Physical & Data Link layers is known as Ethernet. Ethernet is available in different forms where the current Ethernet can be defined through the IEEE 802.3 standard.

Ethernet protocols are available in different flavors and operate at various speeds by using different types of media. But, all the Ethernet versions are well-matched through each other. These versions can mix & match on a similar network with the help of different network devices like hubs, switches, bridges to connect the segments of the network that utilize different types of media.

The Ethernet protocol's actual transmission speed can be measured in Mbps (millions of bits for each second), The speed versions of Ethernet are available in three different types 10 Mbps, called Standard Ethernet; 100 Mbps called Fast Ethernet & 1,000 Mbps, called as Gigabit Ethernet. The transmission speed of the network is the maximum speed that can be attained over the network in ideal conditions. The output of the Ethernet network rarely achieves this highest speed.

1.2. ETHERNET FRAME FORMAT:

Generally, the structure of the Ethernet frame is defined within the IEEE 802.3 standard. But numerous optional frame formats are being employed for Ethernet to expand the capacity of the protocol. The Early frame versions were very slow but the latest Ethernet versions operate at 10 Gigabits/sec. So this is the very fastest Ethernet version.

The frame structure at the data link layer in the OSI model is almost the same for all Ethernet speeds. The structure of the frame simply adds headers & trailers in the region of the Layer 3 PDU (Protocol Data Unit) to summarize the message. The frame structure of Ethernet is shown below and it begins with the Preamble that functions at the physical layer

Ethernet header includes both Source & Destination MAC address, after which the frame's payload is present. The end field is Cyclical Redundancy Checking, used to notice the error. The following diagram shows the structure of the frame & fields.

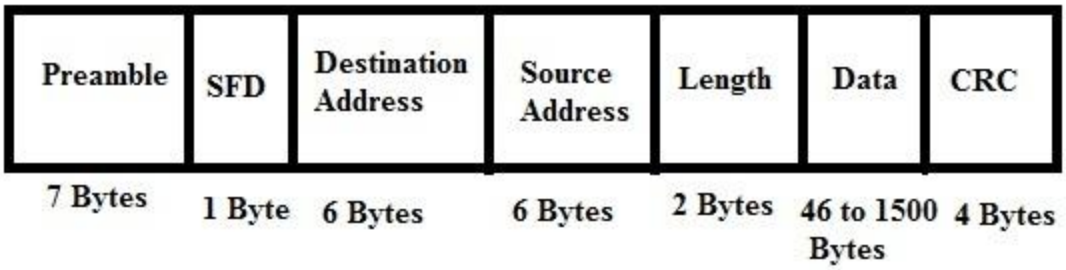


FIG 1.2 : Ethernet Frame Structure

Preamble

The first pattern of the Ethernet Protocol frame is 7-Bytes of Preamble where alternative 0's and 1's in this frame indicate the beginning of the frame & permit the sender & receiver to set up bit-level synchronization. At first, a Preamble in the above frame was introduced to permit for the few bits loss because of signal delays.

However, the present high-speed-based Ethernet doesn't require a Preamble for protecting the frame bits. The preamble specifies to the receiver that the frame is coming & lets the receiver lock on the data stream before the genuine frame starts.

Start of Frame Delimiter

The start of frame delimiter (SFD) is a 1-byte field with 10101011 values that indicates that upcoming bits are the beginning of the frame, which is the address of the destination. The start of the frame delimiter is mainly designed to split the pattern of the bit to the preamble & signal the beginning of the frame.

Sometimes, the start of the frame delimiter is considered the main part of the preamble, so this is the main reason that the Preamble is expressed as 8 Bytes in several places. The SFD gives a warning to stations that this is the final opportunity for synchronization.

Destination Address

The Destination Address Field is a 6-byte field in the above Ethernet frame. The address within the frame & the device MAC address is compared. If both addresses are matched, then the device simply allows the frame. This MAC address is a unicast, multicast, or broadcast.

Source Address

The source address is a 6-byte field, including the source machine's MAC address. Once the address of the source is an individual address or Unicast always, then the LSB of an initial byte will be always.

Length

This field size is 2-byte long which specifies the entire Ethernet frame length. The length value held by the 16-bit field ranges from 0 to 65534, however, the length cannot be higher than 1500 due to some Ethernet limitations.

Data Field

The data field is the location where actual data can be added and it is also called the Payload. Here, both the data & IP header will be inserted if IP is used over Ethernet. So, the highest data available may be 1500 Bytes. If the data length is below the minimum length of 46 bytes, then padding zeros can be included to reach the minimum achievable length.

CRC Field

The CRC in the frame is the last pattern with 4 Bytes. This field includes 32 bits of data hash code, which is produced over the Source Address, Destination Address, Length, and Ethernet Protocol's Data field. If the checksum is calculated through a destination that is not similar to the sent checksum value, then the received data can be corrupted.

Here, the frame size for IEEE 802.3 Ethernet standard changes from 64 bytes – 1518 bytes with 46 to 1500 bytes of data length.

Extended Ethernet Frame

In the above, the standard Ethernet frame is discussed in detail. Now let's discuss the extended Ethernet frame using which we can obtain a Payload even higher than 1500 Bytes.

CHAPTER 2
FEATURES OF ETHERNET IP

2.1. FEATURES

The following lists the main features of the Ethernet IP core.

1. Automatic 32-bit CRC Generation and Checking

CRC Generation: Automatically generates a 32-bit Cyclic Redundancy Check (CRC) for each frame to ensure data integrity.

CRC Checking: Checks the received frames for CRC errors to detect any data corruption.

2. Delayed CRC Generation

Delayed Generation: Allows for the generation of CRC after some initial processing, ensuring accurate error detection without affecting performance.

3. Preamble Generation and Removal

Preamble Generation: Automatically generates the 7-byte preamble and the 1-byte Start of Frame Delimiter (SFD) at the beginning of each frame.

Preamble Removal: Removes the preamble from received frames before forwarding them to higher layers.

4. Automatic Padding of Short Frames

Frame Padding: Automatically pads frames that are shorter than the minimum Ethernet frame size (64 bytes) to ensure compliance with Ethernet standards.

5. Full Duplex Support

Full Duplex Mode: Supports simultaneous transmission and reception of data, effectively doubling the network bandwidth and eliminating collisions.

6. Support for 10 and 100 Mbps Bit Rates

Bit Rate Support: Supports both 10 Mbps and 100 Mbps bit rates, allowing for compatibility with different Ethernet speeds.

7. Inter-Packet Gap: Aborts packet transmission if the inter-packet gap is too small, ensuring proper spacing between frames.

8. Complete Status for TX/RX Packets

Status Reporting: Provides comprehensive status information for transmitted and received packets, aiding in network diagnostics and management.

9. Internal RAM for Buffer Descriptors
Buffer Descriptors: Utilizes internal RAM to hold 128 TX/RX buffer descriptors, enhancing memory management and data handling efficiency.

10. Interrupt Generation
Event Interrupts: Generates interrupts for various events, such as packet transmission/reception, errors, and status changes, enabling efficient handling and processing by the host system.

CHAPTER 3
MERITS & DEMERITS

3.1. ADVANTAGES

- The cost of installing an Ethernet connection is affordable.
- Provides high-speed data transmission for data in the network.
- It maintains data quality and also provides a secure channel for data transmission.
- Scalability and Flexibility: Many Ethernet IP cores are designed to be scalable and configurable, allowing for customization to meet specific application requirements, such as different data rates, protocols, and features.

3.2. DISADVANTAGES

- Ethernet networks are more suited for short-distance connections.
- Troubleshooting faults in the ethernet connection is difficult.
- Increased cases of network traffic in the network channel.

3.3. APPLICATIONS

- **Telecommunications**: Ethernet IP cores are used in network switches, routers, and other telecom equipment to facilitate high-speed data transmission and connectivity.
- **Data Centers**: In data centers, Ethernet IP cores enable high-speed communication between servers, storage devices, and networking equipment, supporting cloud computing, virtualization, and large-scale data processing.
- **Automotive**: Modern vehicles use Ethernet IP cores for in-vehicle networking to connect various systems like infotainment, advanced driver-assistance systems (ADAS), and diagnostic systems, enhancing communication speed and reliability.
- **Industrial Automation**: Ethernet IP cores are employed in industrial automation for real-time communication between controllers, sensors, and actuators in factory automation, robotics, and process control systems.
- **Consumer Electronics**: Devices such as smart TVs, gaming consoles, and home networking equipment use Ethernet IP cores for high-speed internet connectivity and data exchange.
- **Military and Aerospace**: In military and aerospace applications, Ethernet IP cores provide robust and secure communication links for avionics systems, unmanned vehicles, and battlefield communications.
- **Broadcasting**: Ethernet IP cores enable high-speed data transmission for video streaming, content delivery networks (CDNs), and broadcasting equipment, supporting live broadcasts and on-demand streaming services.
- **Smart Grids**: Ethernet IP cores are used in smart grid applications for real-time monitoring and control of electrical grids, facilitating efficient energy distribution and management.

CHAPTER-4

ETHERNET IP CORE

4.1. PIN DIAGRAM:

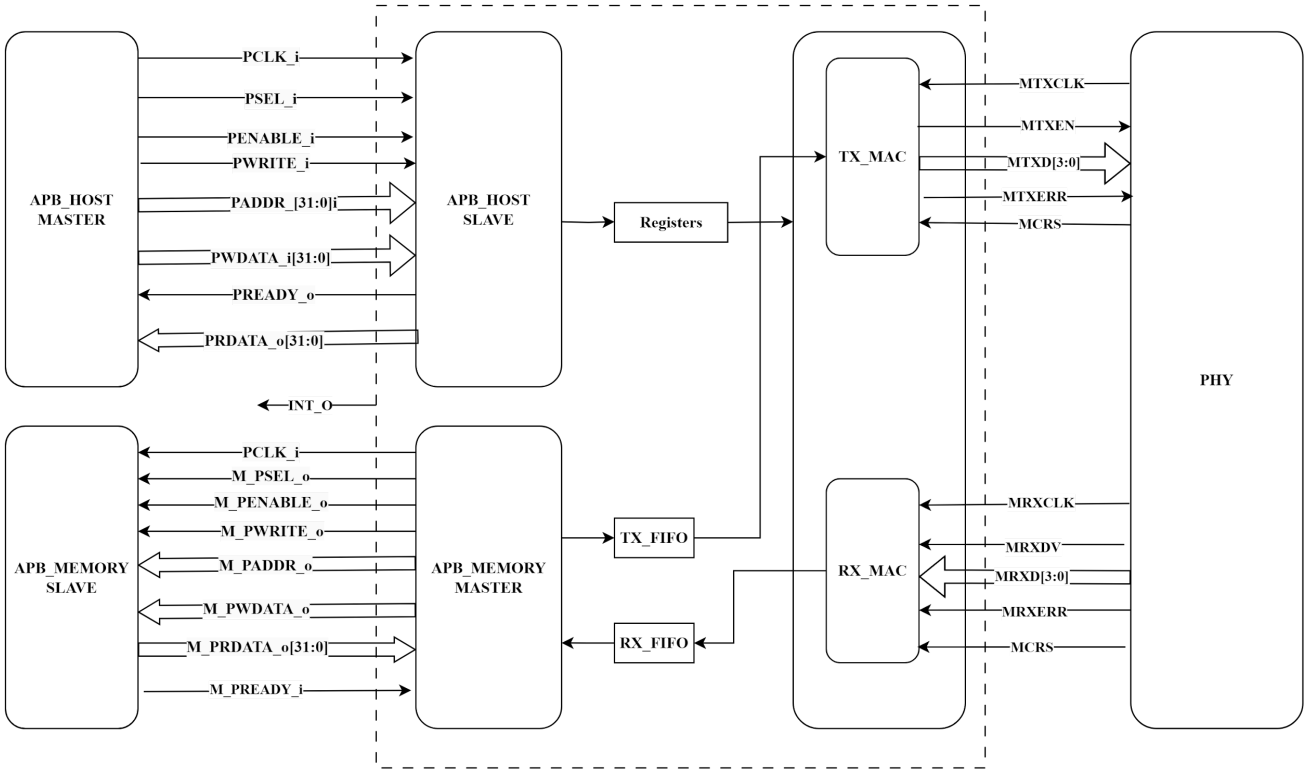


FIG 4.1: PIN DIAGRAM OF ETHERNET

4.2 SIGNAL DESCRIPTION :

4.2.1.Host Interface Ports :

The table below contains the common ports connecting the Ethernet IP Core to the Host Interface. The Host Interface is APB.

All signals listed below are active HIGH, unless otherwise noted. Signal direction is with respect to the Ethernet IP Core.

Port	Width	Direction	Description
pclk_i	1	I	APB Bus Clock Input(25MHZ)
prstn_i	1	I	Reset Input
paddr_i	32	I	Address Input
pwdata_i	32	I	Data Input
prdata_o	32	O	Data Output
psel_i	1	I	Slave Select Input
pwrite_i	1	I	Write Input Indicates a Write Cycle when asserted high or a Read Cycle when asserted low.
penable_i	1	I	Enable Input Indicates the beginning of a valid transfer cycle.
pready_o	1	O	Peripheral ready Output
int_o	1	O	Interrupt Output(After every BD transmission or reception we get 1 in this signal, User should clear this interrupt and maintain a separate counters for good and bad frames).
m_paddr_o	32	O	Address Output
m_prdata_i	32	I	Data Input
m_pwdata_o	32	O	Data Output
m_psel_o	1	O	Slave Select Output
m_pwrite_o	1	O	Write Output Indicates a Write Cycle when asserted high or a Read Cycle when asserted low.
m_penable_o	1	O	Enable Output Indicates the beginning of a valid transfer cycle.
m_pready_i	1	I	Acknowledgment Input

Table 4.1. Host Interface Pin Description

4.2.2.Ethernet MAC Interface ports :

The table below contains the ports connecting the Ethernet IP Core to the Ethernet MAC Interface. All signals listed below are active HIGH, unless otherwise noted. Signal direction is with respect to the Ethernet IP Core.

port	Width	Direction	Description
MTxEn	1	O	Transmit Enable. When asserted, this signal indicates to the PHY that the data MTxD[3:0] is valid and the transmission can start. The transmission starts with the first nibble of the preamble. The signal remains asserted until all nibbles to be transmitted are presented to the PHY. It is deasserted prior to the first MTxClk, following the final nibble of a frame.
MTxErr	1	O	Transmit Coding Error. When asserted for one MTxClk clock period while MTxEn is also asserted, this signal causes the PHY to transmit one or more symbols that are not part of the valid data or delimiter set somewhere in the frame being transmitted to indicate that there has been a transmit coding error. MTXErr is a pulse.
MTxClk	1	I	Transmit Nibble or Symbol Clock. The PHY provides the MTxClk signal. It operates at a frequency of 25 MHz (100 Mbps) or 2.5 MHz (10 Mbps). The clock is used as a timing reference for the transfer of MTxD[3:0], MtxEn, and MTxErr.
MTxD[3:0]	4	O	Transmit Data Nibble. Signals are the transmit data nibbles. They are synchronized to the rising edge of MTxClk. When MTxEn is asserted, PHY accepts the MTxD[3:0].
MCrS	1	I	Carrier Sense. The PHY asynchronously asserts the carrier sense MCrS signal after the medium is detected in a non-idle state(non idle state- HalfDuplex Tx/Rx). When deasserted, this signal indicates that the media is in an idle state (and the transmission can start). Note: To start Half Duplex Tx/Rx mode, MAC should make sure MCrS = 0(Zero indicates medium is in idle state). Once the Data is available on the channel PHY should take care of keeping MCrS = 1, otherwise MAC is not responsible for this and behavior of MAC can be unexpected.

Table 4.2. Ethernet MAC Signal Descriptions

4.3. ARCHITECTURE

- The Ethernet IP Core consists of 5 modules:
- Host Interface and the BD structure
- TX Ethernet MAC (transmit function)
- RX Ethernet MAC (receive function)
- MAC Control Module
- MII Management Module

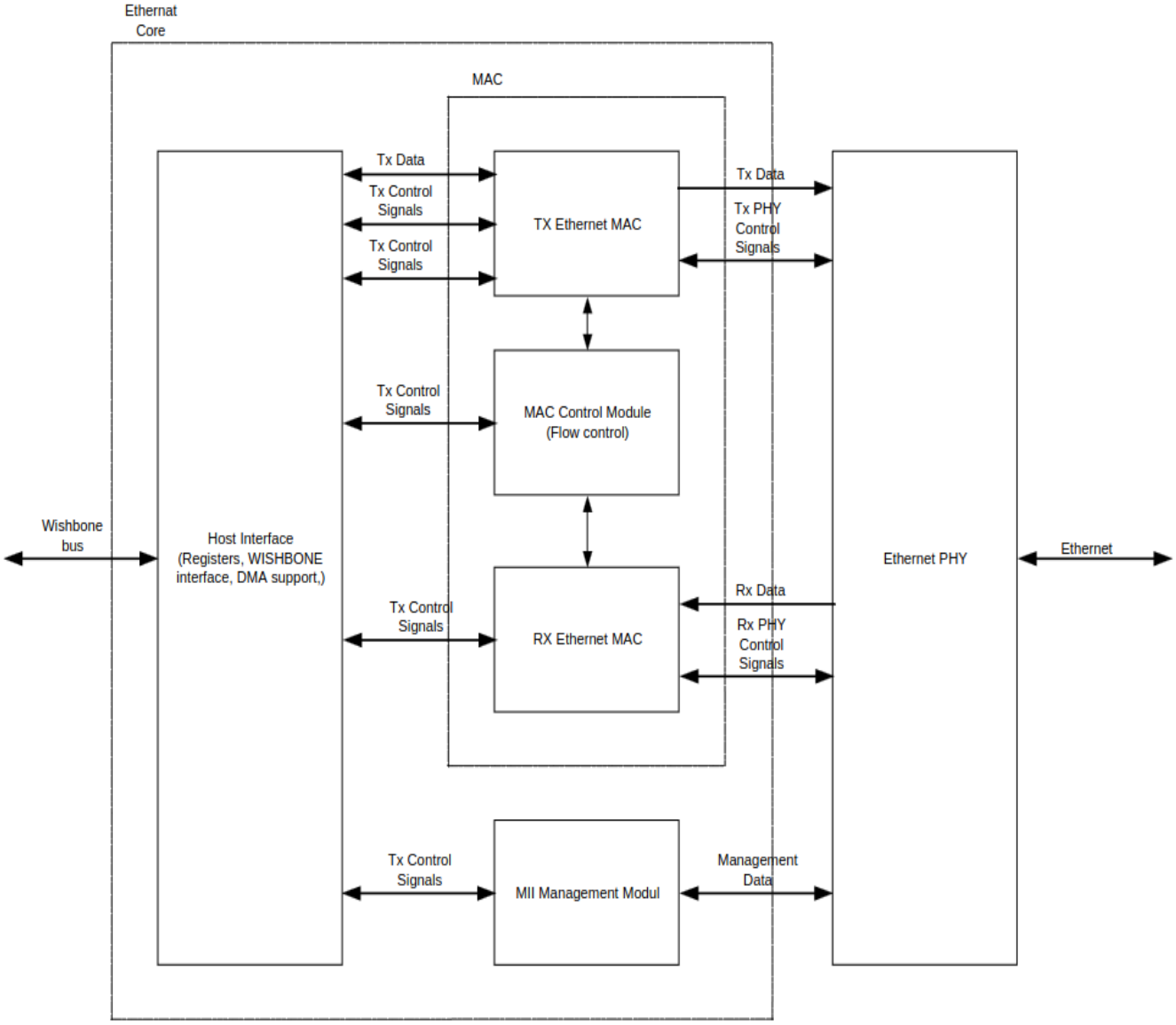


Figure 4.2: Ethernet Architecture Overview

4.3.1.Host Interface

The host interface is connected to the RISC and the memory through the two APB interfaces. The RISC writes the data for the configuration registers directly while the data frames are written to the memory. For writing data to configuration registers, the APB slave interface is used. Data in the memory is accessed through the APB master interface.

4.3.2.TX Ethernet MAC

The TX Ethernet MAC generates 10BASE-T/100BASE-TX transmit nibble data streams in response to the byte streams supplied by the transmit logic (host). It takes care of the IPG, computes the checksum (FCS) and monitors the physical media (by monitoring Carrier Sense).

Basic Transmitting data path:

To transmit the first frame, configure the Registers and buffer descriptors using the APB interface. The data that is driven from the memory is stored in the memory interface.

- ➔ Store the frame in the memory.
- ➔ Associate the Tx BD in the Ethernet MAC core with the packet written to the memory (length, pad, CRC, etc.).
- ➔ Enable the TX part of the Ethernet Core by setting the TXEN bit to 1.

As soon as the Ethernet IP Core is enabled, it continuously reads the first BD. Immediately when the descriptor is marked as ready, the core reads the pointer to the memory storing the associated data and starts reading data to the internal FIFO. At the moment the FIFO is loaded, transmission begins.

4.3.3.RX Ethernet MAC

The RX Ethernet MAC interprets 10BASE-T/100BASE-TX receive data nibble streams and supplies correctly formed packet-byte streams to the host. It searches for the SFD (start frame delimiter) at the beginning of the packet, verifies the FCS, and detects any dribble nibbles or receive code violations.

Basic Receiving data path:

To receive the data from the Ethernet MAC interface signal. Drive the Packet frame from the sequence.

- Set the receive buffer descriptor to be associated with the received packet and mark it as empty.
- Enable the Ethernet receive function by setting the RXEN bit to 1
- The Ethernet IP Core reads the Rx BD. If it is marked as empty, it starts receiving frames. The Ethernet receive function receives an incoming frame nibble per nibble. After the whole frame has been received, the receive status is written to the BDs.

4.3.4.CRC calculation with TX and RX paths:

TxPath:

- CRC will be calculated on DA, SA, Length, and Payload.
- The calculated CRC will be appended at the end of the frame.

RxPath:

- CRC will be calculated on the complete Frame (DA, SA, Length, Payload and CRC).
- The calculated CRC has to be compared with the Reference number (Magic number- 32'hc704dd7b).
- Final CRC matches with Magic number then only accept the Frame, else drop the Frame.

CHAPTER 5
UVM ENVIRONMENT

5.1 ARCHITECTURE

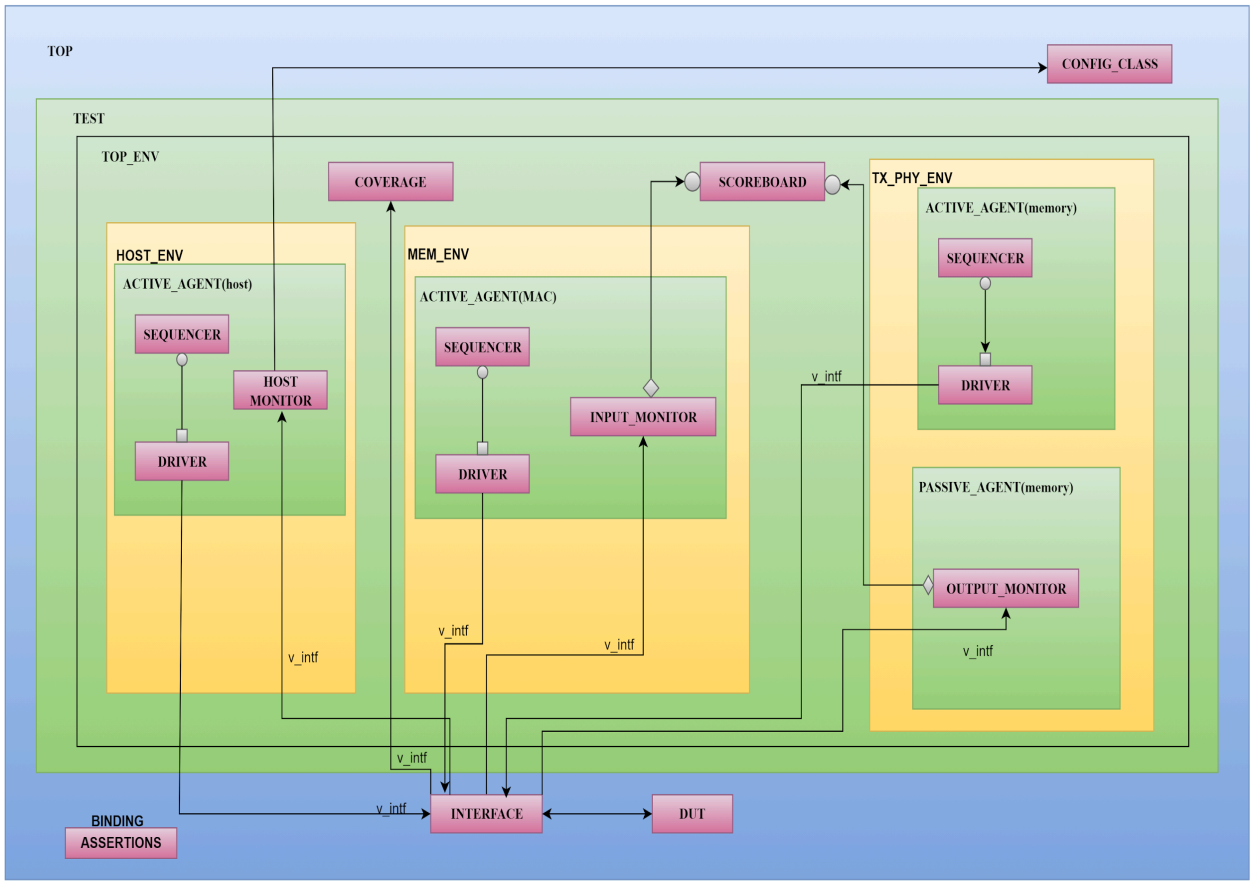


Fig 5.1: UVM Environment Architecture

Testbench: The testbench serves as the environment for verifying the design under test (DUT). It comprises several modules and components responsible for generating stimuli, applying them to the DUT, monitoring its behavior, and verifying correctness.

Testbench Components:

- **Sequence Item:** Units of data or events exchanged in a testbench, containing data values, control signals, and timing, used for communication and stimulus in verification.

- **Sequence:** Produces stimulus or test vectors for the DUT, either randomly, based on predefined sequences, or specific test scenarios.
- **Sequencer: The** Sequencer provides a handshaking mechanism between the sequence and the driver.

- **Driver:** Converts transaction-level stimuli from the generator into signal-level waveforms for the DUT's input ports and drives these stimuli into the DUT.
- **Agent:** Agent connects driver, sequencer and monitors.

- **Interface:** Defines communication protocols and signal interactions between the testbench and DUT, encapsulating signals, methods, and properties for data exchange.

- **Monitor:** Observes and captures signals entering and leaving the DUT, recording transaction-level information for analysis. Input monitors capture applied stimuli, while output monitors capture DUT responses.

- **Scoreboard:** Compares DUT outputs with expected results from a reference model, verifying correctness by checking if DUT outputs match expected outputs.

- **Coverage:** Tracks verification completeness by measuring design aspects' thoroughness, capturing events, conditions, and states for adequate testing.

- **Test:** Test class in SystemVerilog coordinates hardware validation, handling stimuli and interactions with the design. It ensures organized, reusable verification in test benches.

- **Design Under Test (DUT):** Module or collection of modules being verified, representing the design under development and serving as the verification target in the testbench environment.

- **Testbench Top Module:** Highest-level module in the testbench hierarchy, instantiating the DUT and connecting it to testbench components, orchestrating the overall verification process.

5.2 Top Module

- The Top module is the static entity which handles the verification environment that contains all the dynamic entities (test class) and static entities (interface,assertions)
- Config class and virtual interface are set in config_db
- Run_test is invoked in the “initial” block, to run the environment.

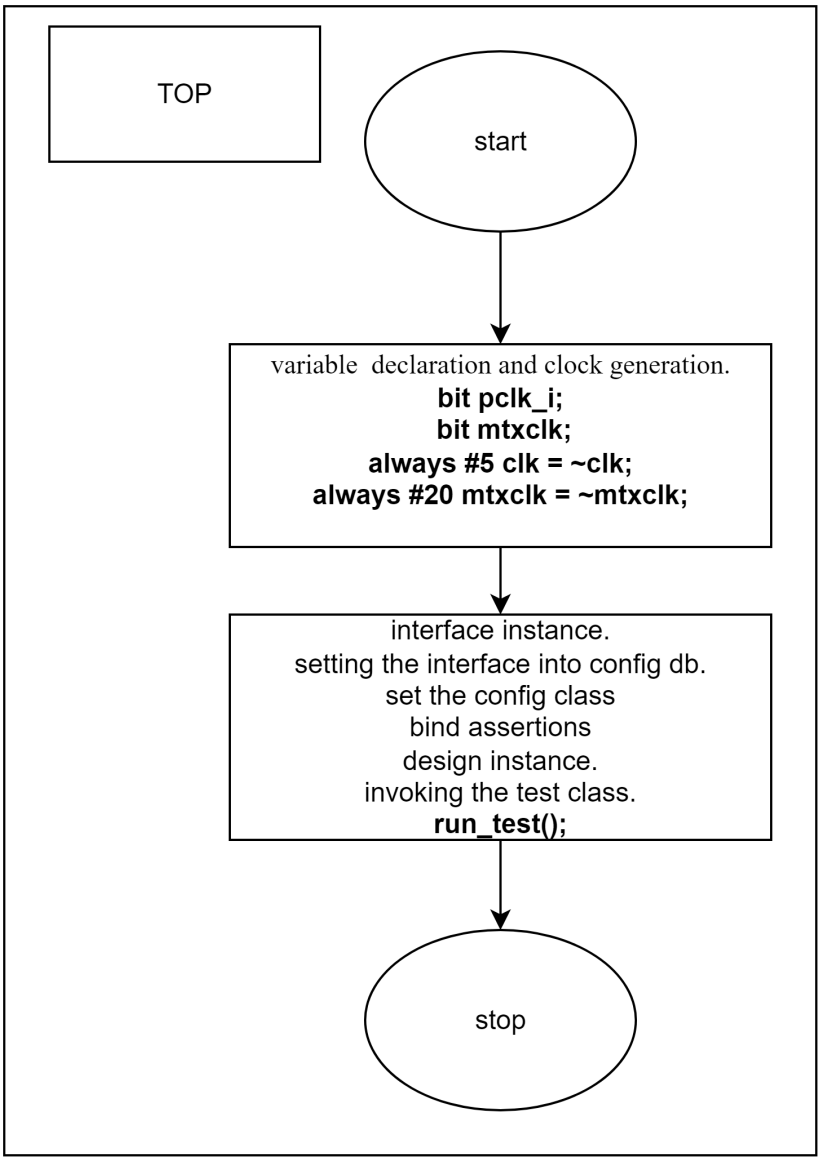


Fig 5.2 : UVM Top Flow

5.3 Test

- Test contains the entire environment, it handles the run_phase of each component.
- ***Build phase*** is used to build the handle of Top_environment
- ***Run phase*** is used to synchronize the run phases of all the components, also it starts the sequence on the respective sequencer.
- ***End_of_elaboration phase*** is used to print the topology of the environment.

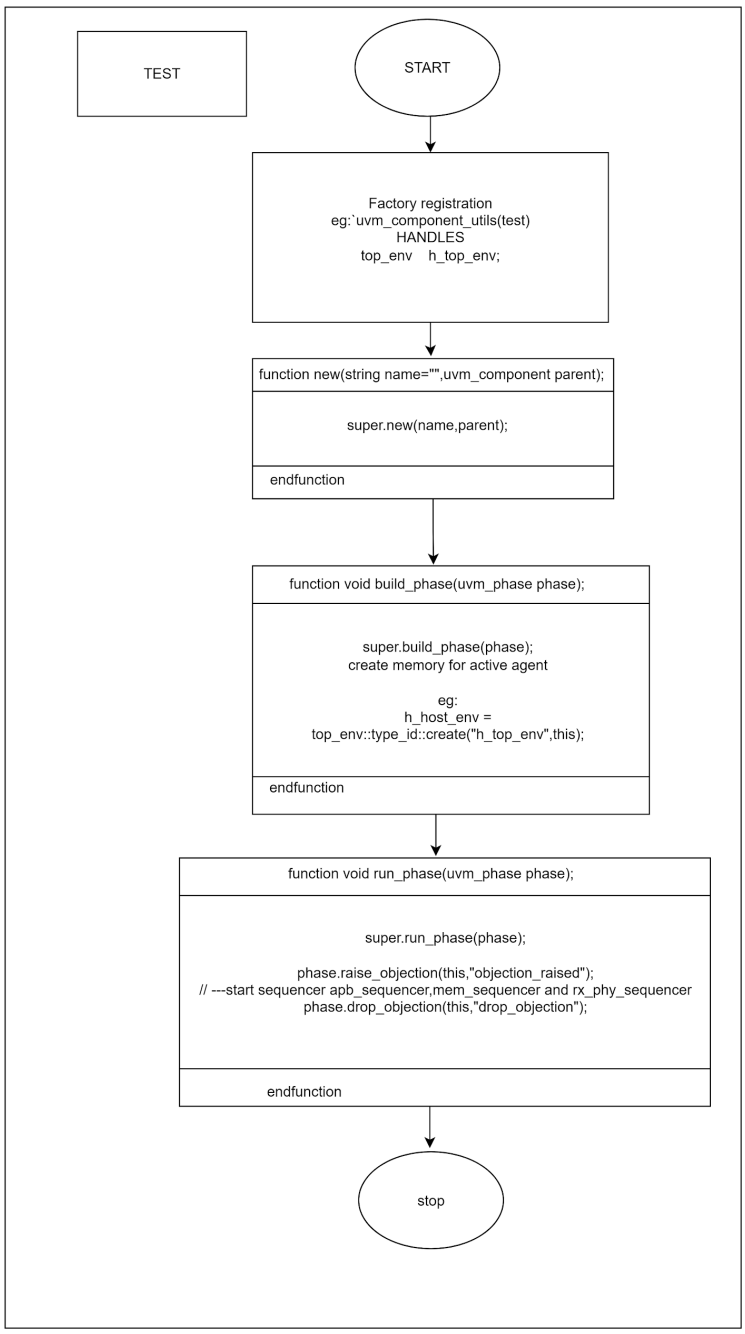


FIG 5.3 : Test Flow Chart

5.4 Interface

- The Interface Class acts as a bridge between the Ethernet Design Under Test (DUT) and the verification environment.
- Clocking Blocks are utilized to efficiently manage driver and monitor functionalities specific to APB_interface and PHY_interface.
- Monitor Integration treats all DUT inputs and outputs as clocking block input signals, ensuring protocol compliance.
- Clocking Block Operation synchronizes actions precisely at the rising edge (posedge) of the clock signal, meeting Ethernet IP core timing requirements.

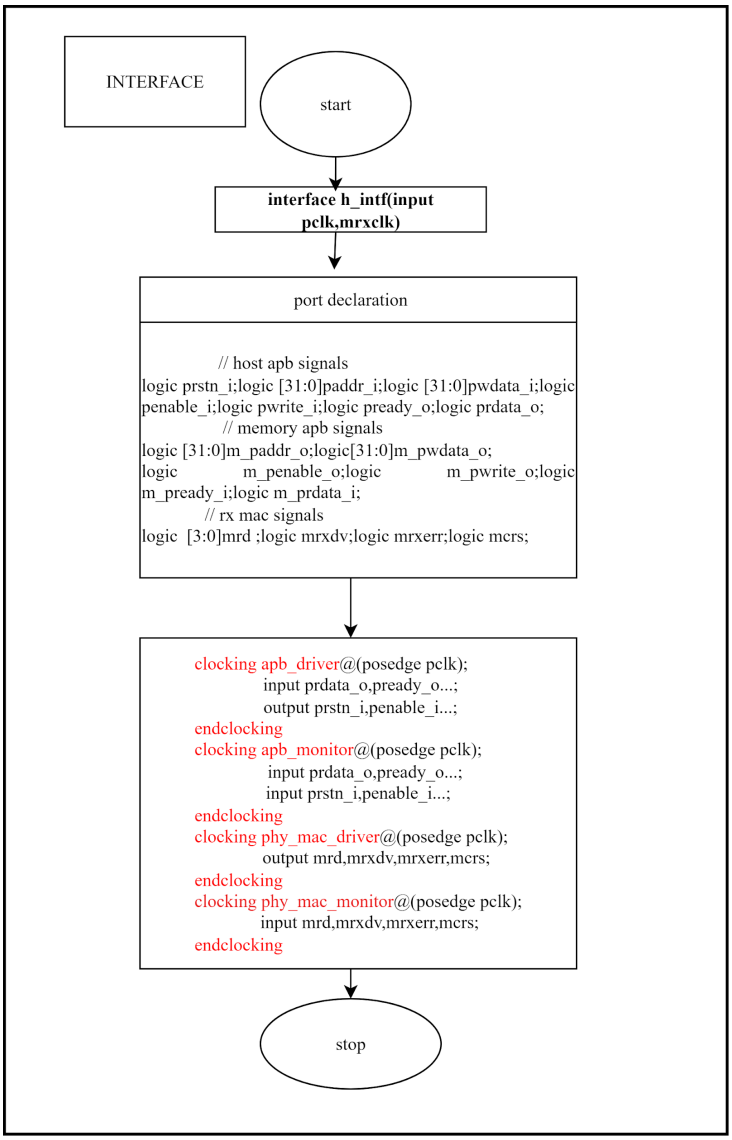


Fig 5.4 : Interface FlowChart

5.5. Top_Environment

- ***Build phase*** in Top_environment is used to build Host_environment, Mem_environment, phy_environment,coverage and scoreboard.
- ***Connect_phase*** in Top_environment is used to connect
  →mem_passive_agent to Scoreboard
  →phy_active_agent to scoreboard
- **Constructor Functionality**:
  - Upon instantiation, the constructor in top_environment is used for calling the constructor of base class : uvm_env.

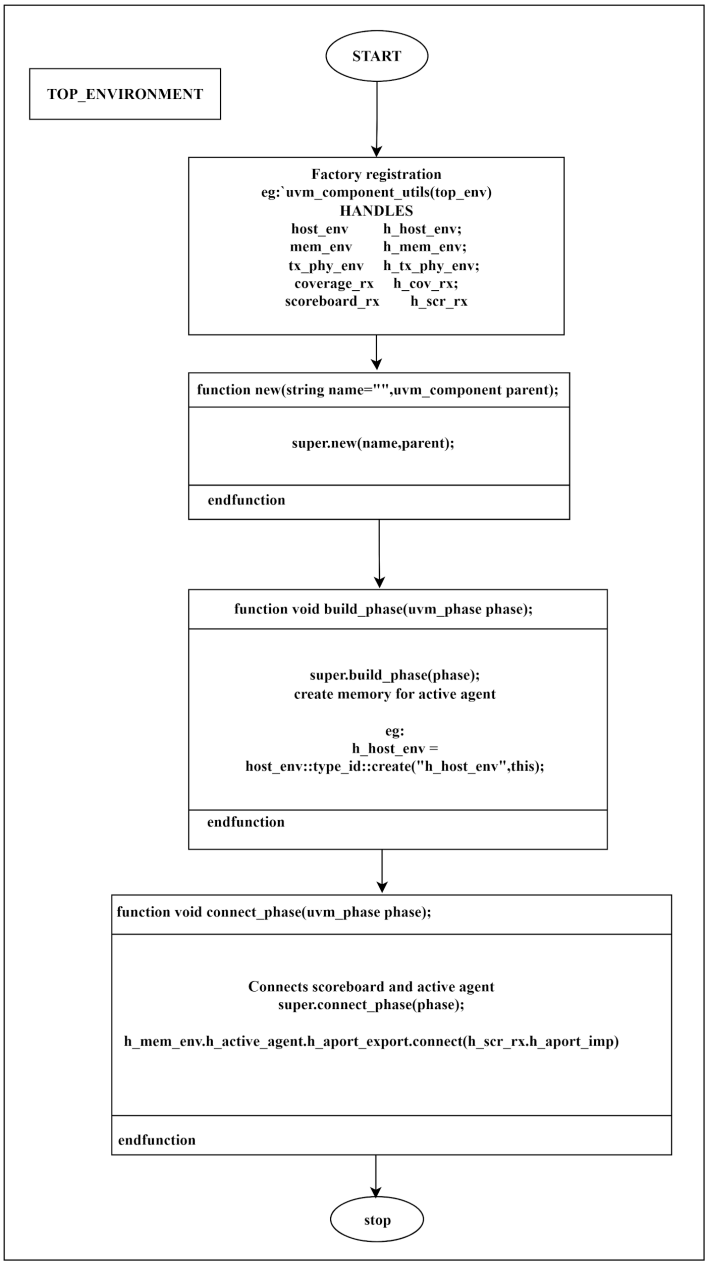


Figure 5.5 : Top Environment Flow

5.5.1. Host_Environment

- ***Build phase*** in host_environment is used to build host_active_agent.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_env.

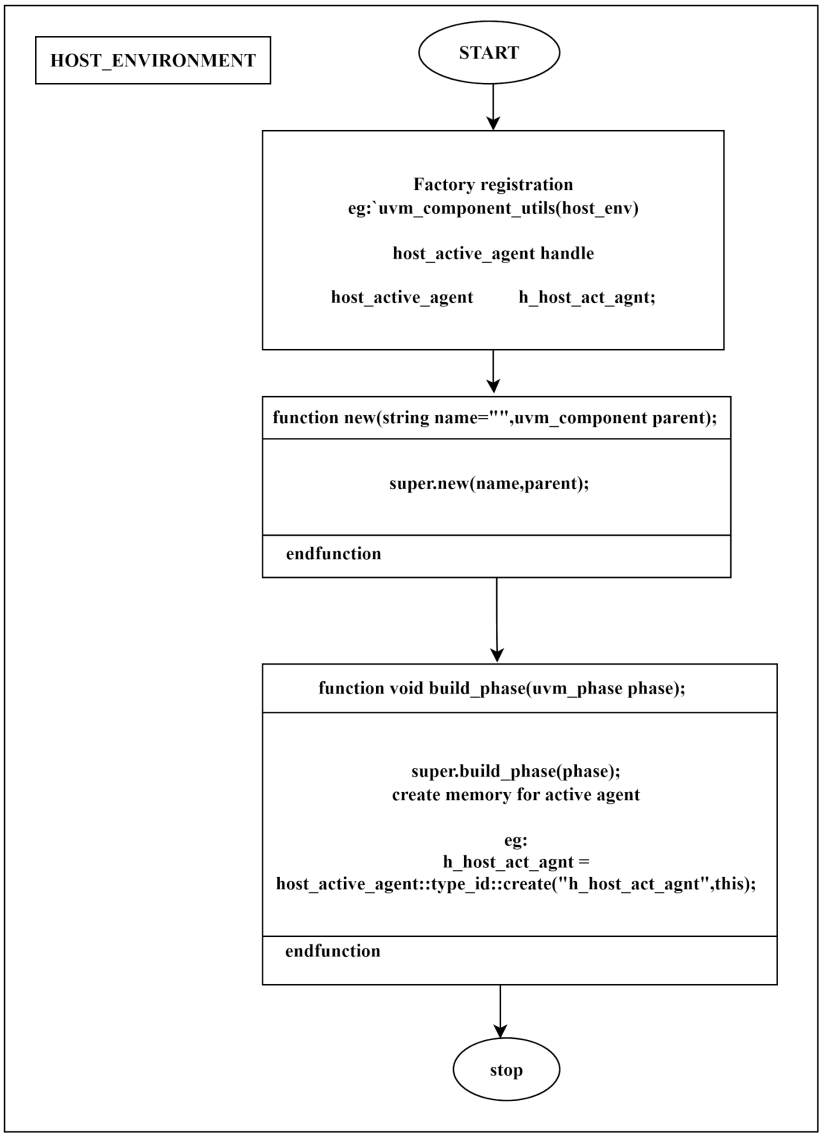


Figure 5.6 : Host_environment Flow

5.5.1.1. Host_Active_agent

- ***Build phase*** in host_active_agent is used to build host_sequencer and host_driver using the create function.
- ***Connect phase*** in host_active_agent is used to connect host_sequencer and host_driver.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_agent.

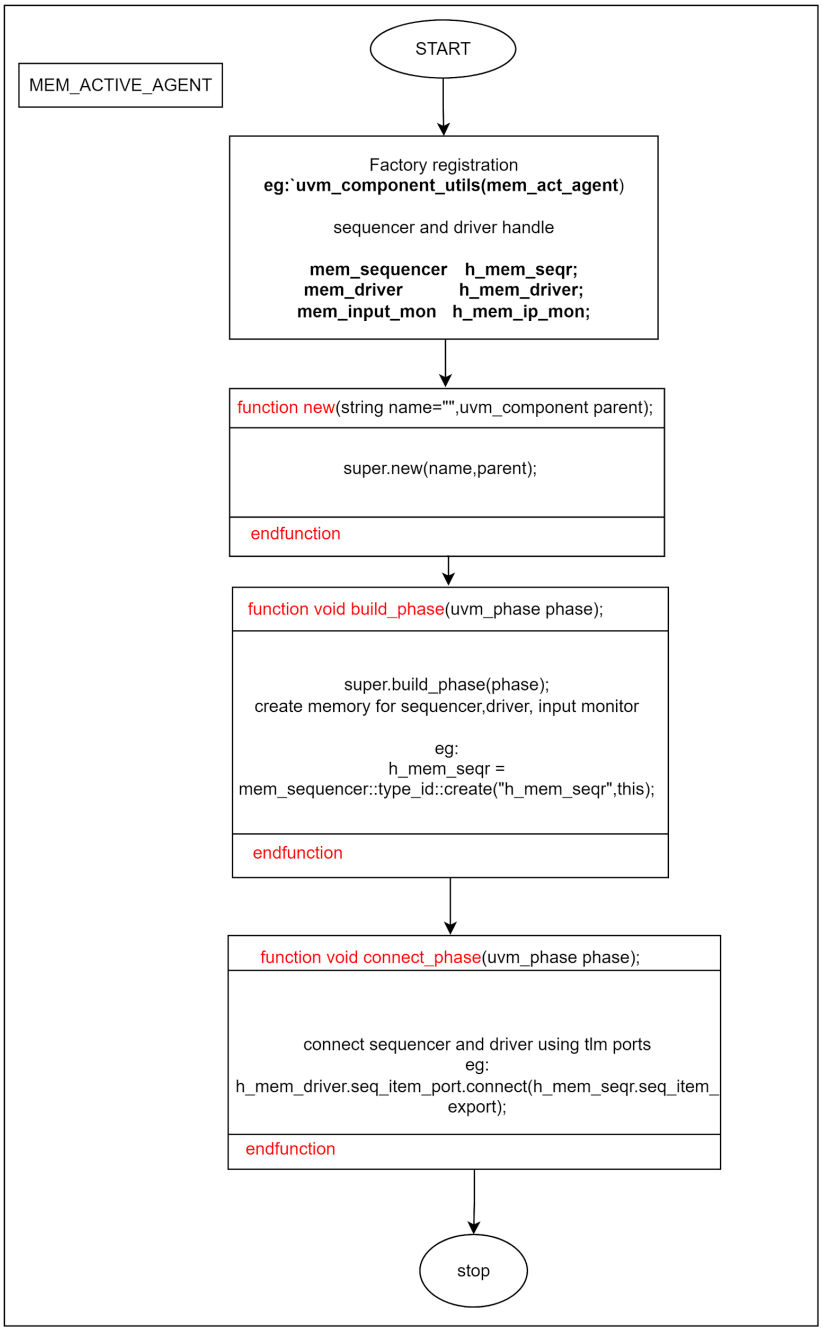


Figure 5.7 :Host_active_agent Flow

5.5.1.2.Host_sequencer

- It provides a handshaking mechanism for host_sequence and host_driver.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_sequencer.

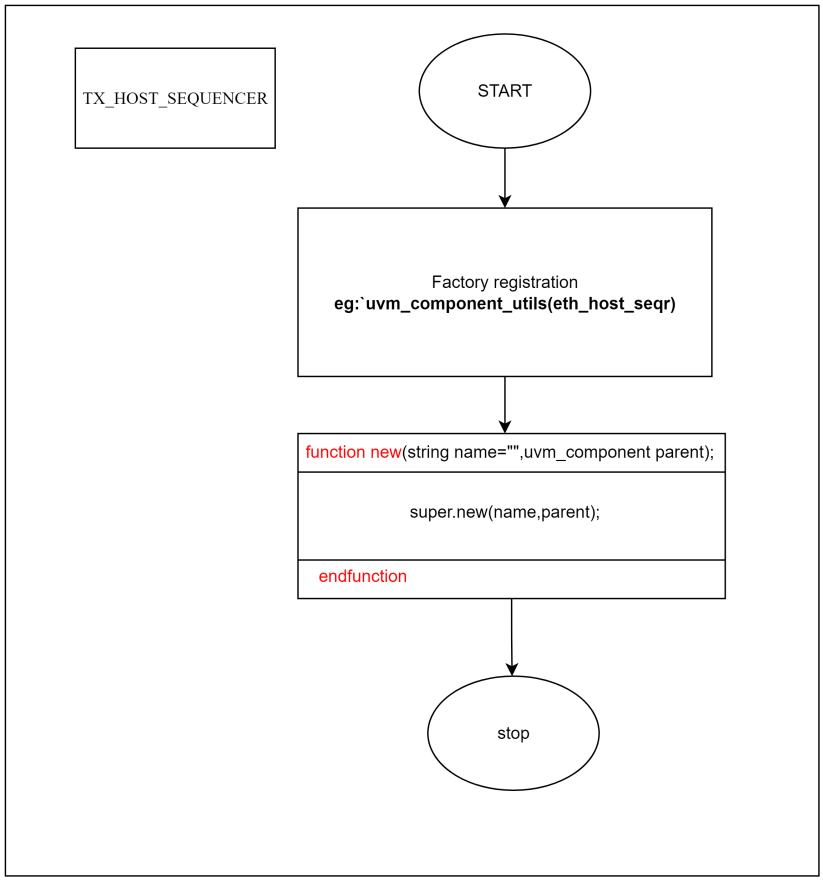


Figure 5.8 :host_sequencer Flow

5.5.1.3.Host_driver

- Host_driver is used to drive input signals of the host apb on to the interface.
- It contains handle of a virtual interface
- ***Connect phase*** in host_driver is used to get the virtual interface from config_db
- ***Run phase*** is used to drive the APB input signals on to the interface and it is activated at every cb_apb_driver clock.
- Following input signals of apb are driven to the interface from host_apb from req(sequence_item).
  → Presetn_i
  → Pclk_i
  → PSelx_i
  → Penable_i
  → Pwrite_i
  → Paddr(32)_i
  → Pwdata(32)_i

- *Run phase executes the following code to drive a sample*
  get_next_item(req)
  Virtual_interface.cb_apb_driver.signal_name <= req.signal_name;
  item_done()

- **Constructor Functionality**:
  - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_driver.

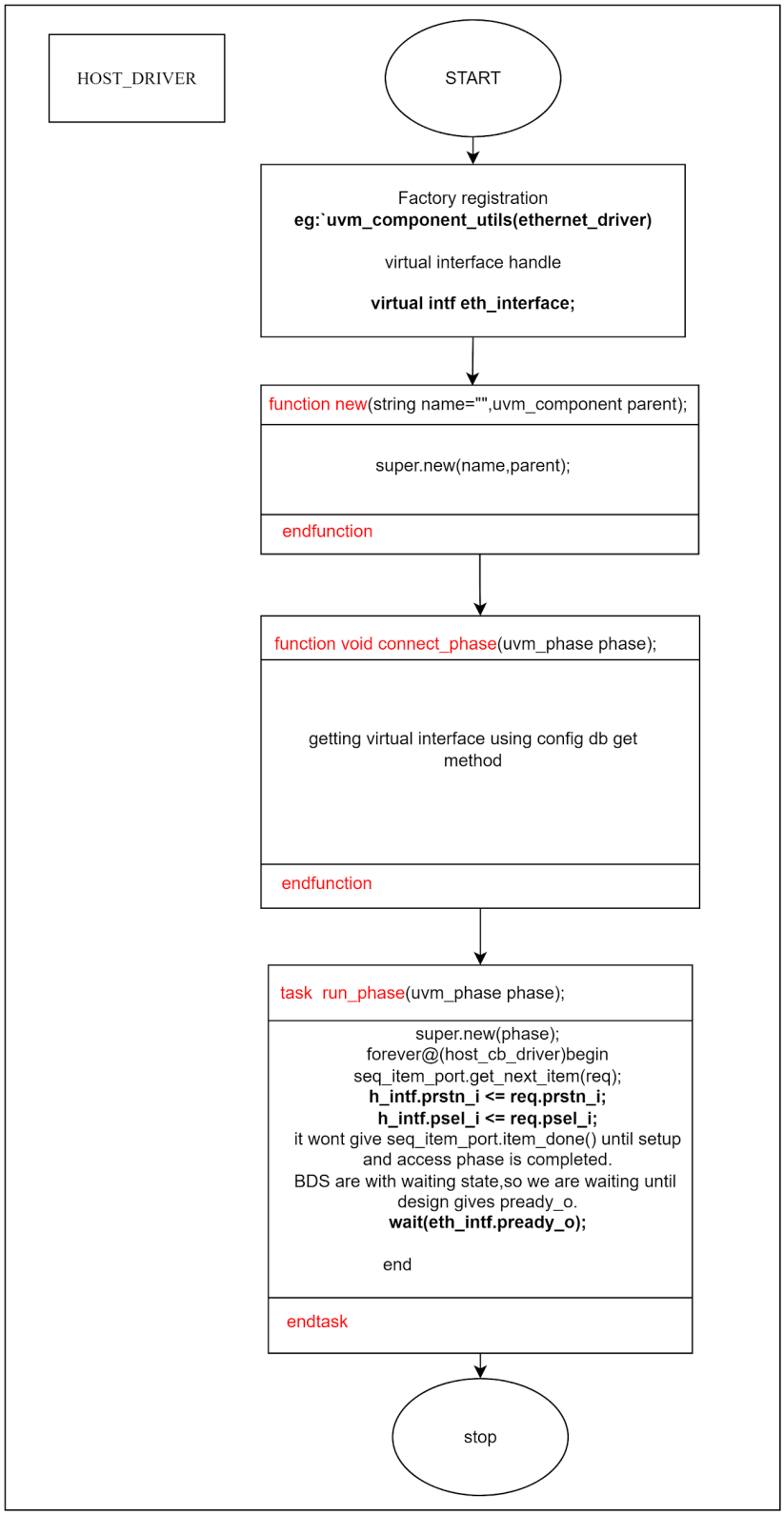


Figure 5.9 : Host Driver Flow

5.5.2 Mem_Environment

- ***Build phase*** in host_environment is used to build mem_active_agent and mem_passive_agent.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_env.

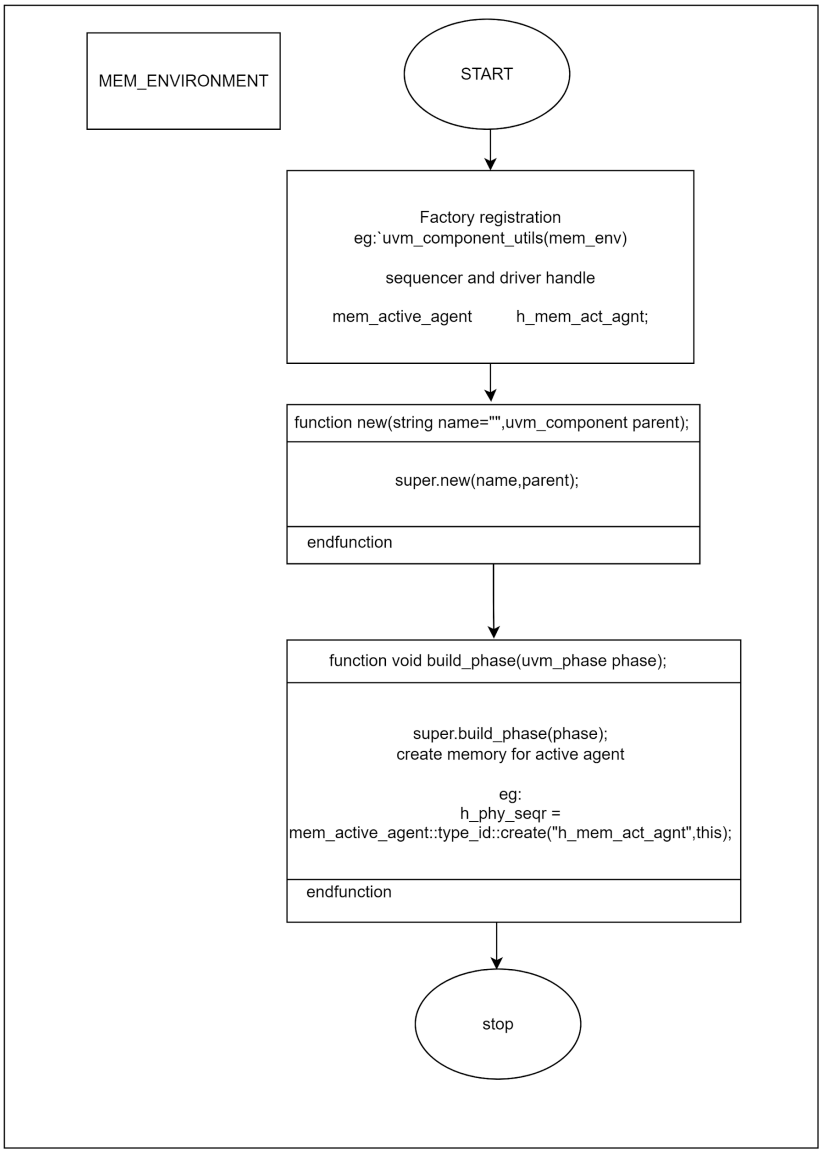


Figure 5.10 :Mem_environment Flow

5.5.2.1. Mem_Active_agent

- ***Build phase*** in mem_active_agent is used to build mem_sequencer , host_driver and input_monitor using the create function.
- ***Connect phase*** in host_active_agent is used to connect host_sequencer and host_driver , intput_monitor to scoreboard.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_agent.

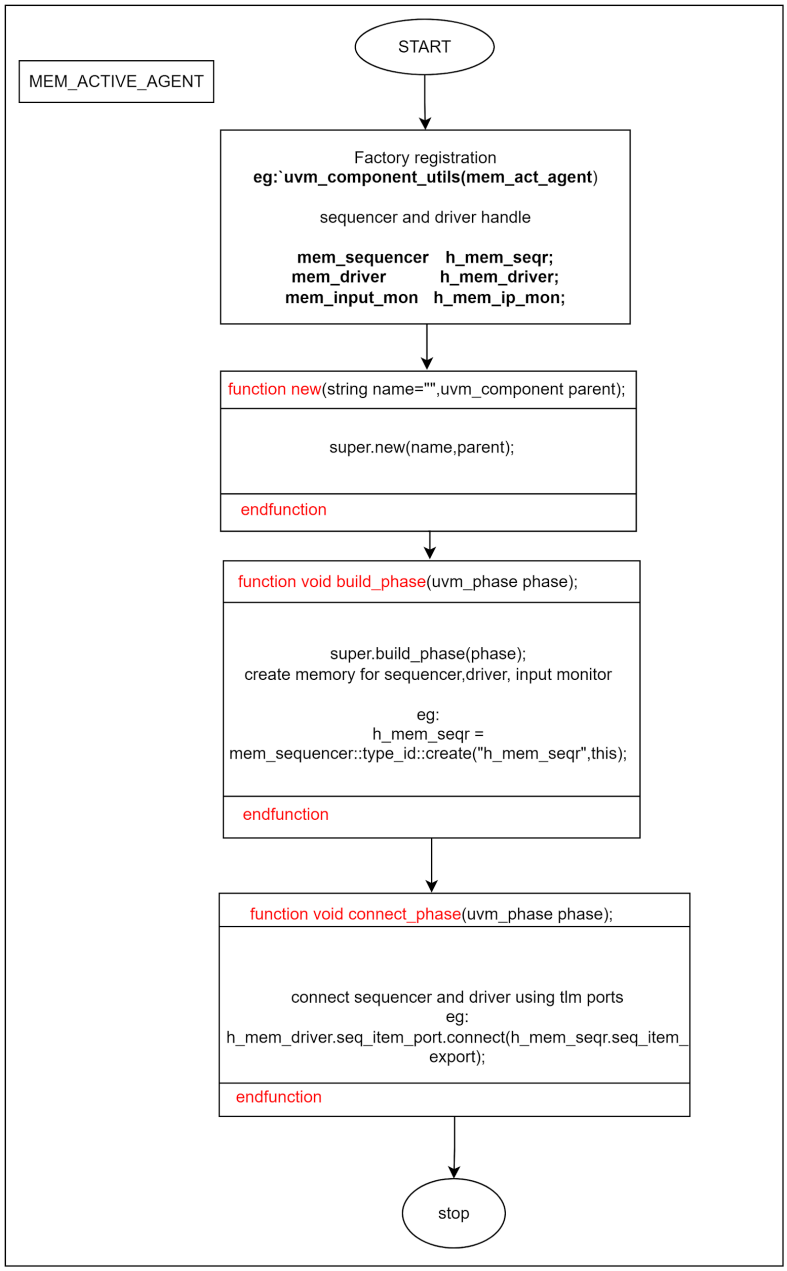


Figure 5.11 :Mem_active_agent Flow

5.5.2.2.Mem_Input_Monitor

- input_monitor Takes the output signals from the interface and performs field checks.
- Payload is sent to scoreboard through passive agent
- Contains handle of sequence_item, virtual interface, and port type analysis_port.
- ***Build_phase*** is used to construct sequence_item and analysis_port.
- ***Connect_phase*** is used to get config_class access
- ***Run_phase*** is used to load the output signals on the interface to sequence_item_handle.
- In run the generated payload is collected and sent to the scoreboard to compare it with design generated payload.
- The run_phase is invoked by raising an objection. In the run phase, txen in moder is high, then it will first check for RD to be high in biffer descriptor. If RD is high it will invoke the monitor_check task.
- In the monitor check task we check the length of payload. If length is less than 4, the frame is dropped, if length is between 4 and 46 with pad is high then it will invoke the payload_generation task.
- If length is between 46 to 1500 and between 1500 to 2030 with hugen high it will invoke the payload_generation task.
- In the payload generation task the data from m_prdata_i is received and sliced into 8 nibbles and sent to the scoreboard from lsb to msb using the write method. If length is less than 46, the remaining bytes rather than given length zeroes will be transmitted.
- The four bits are compared in the scoreboard with the payload which is generated by design which is received and sent by the output monitor.
- **Constructor Functionality**:Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_monitor.

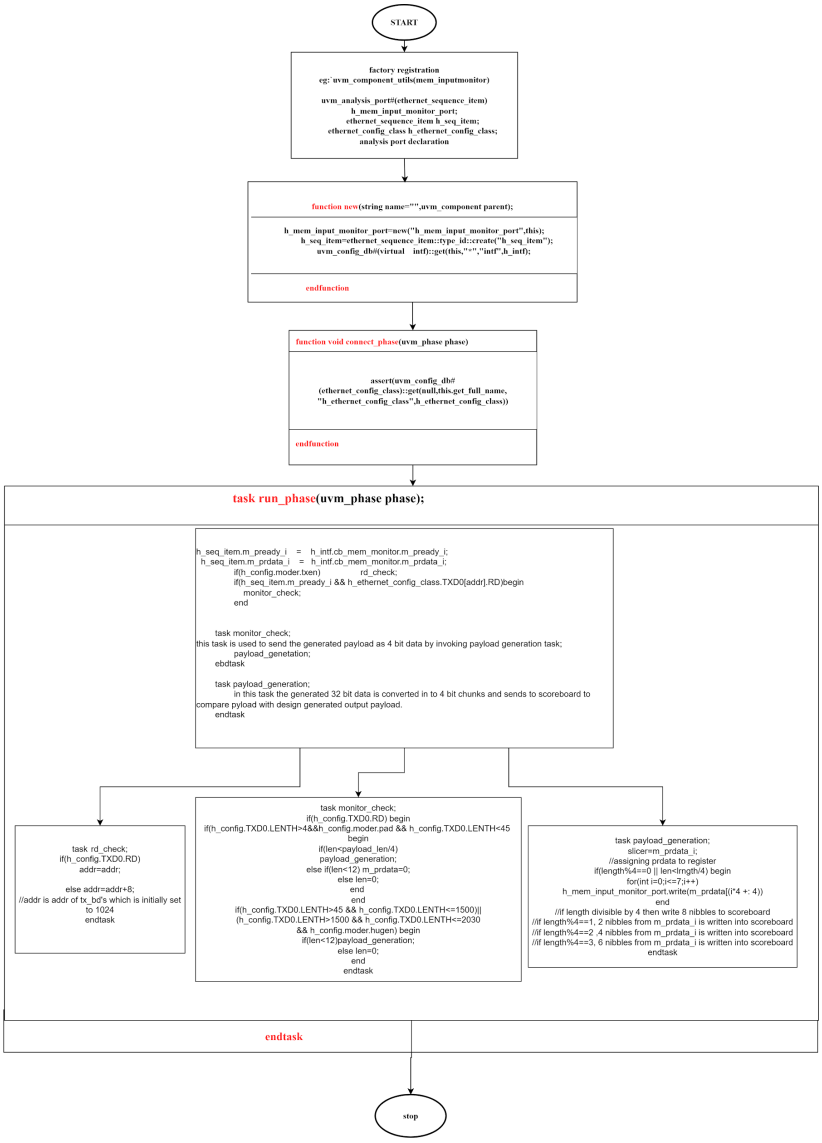


Fig 5.12 : Input Monitor Flow

5.5.2.3.Mem_sequencer

- It provides a handshaking mechanism for mem_sequence and mem_driver.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in mem_environment is used for calling the constructor of base class : uvm_sequencer.

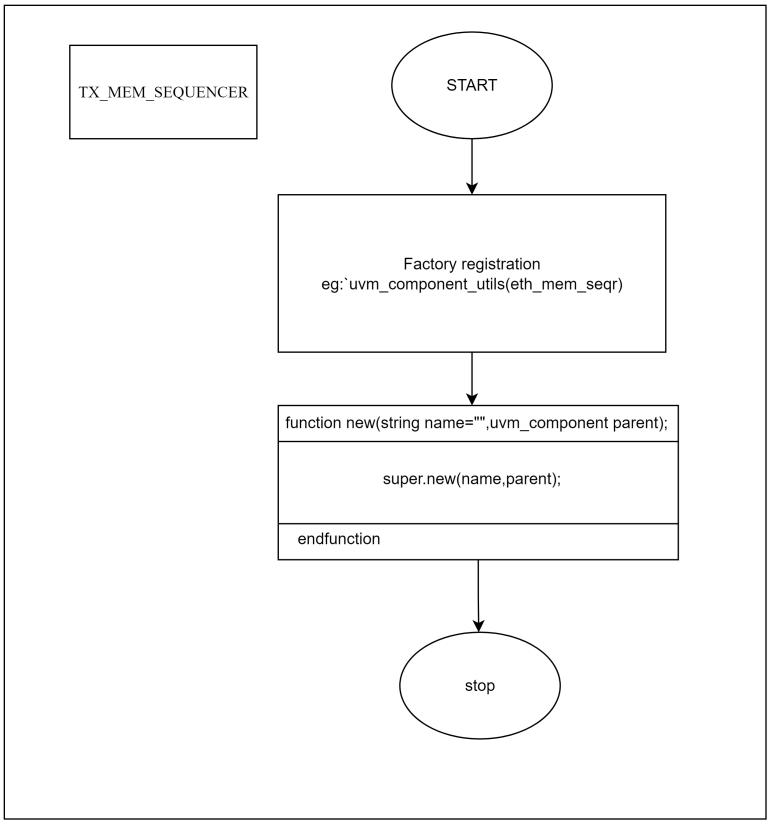


Figure 5.13 :mem_sequencer Flow

5.5.2.4.Mem_driver

- Host_driver is used to drive input signals of the host Mem_apb onto the interface.
- It contains handle of a virtual interface
- ***Connect phase*** in host_driver is used to get the virtual interface from config_db.
- ***Run phase*** is used to drive the MEM_APB input signals on to the interface and it is activated at every cb_apb_driver clock.
- Following input signals of apb are driven to the interface from host_apb from req(sequence_item).
  → Prdata(32)
  → Pready
- *Run phase executes the following code to drive a sample*

  wait(vinf.Psel_i && !vinf.Penable)

  wait(vinf.Psel_i && !vinf.Penable)

  vinf.cb_apb_driver. PREADY <= 1;

- **Constructor Functionality**:
    - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_driver.

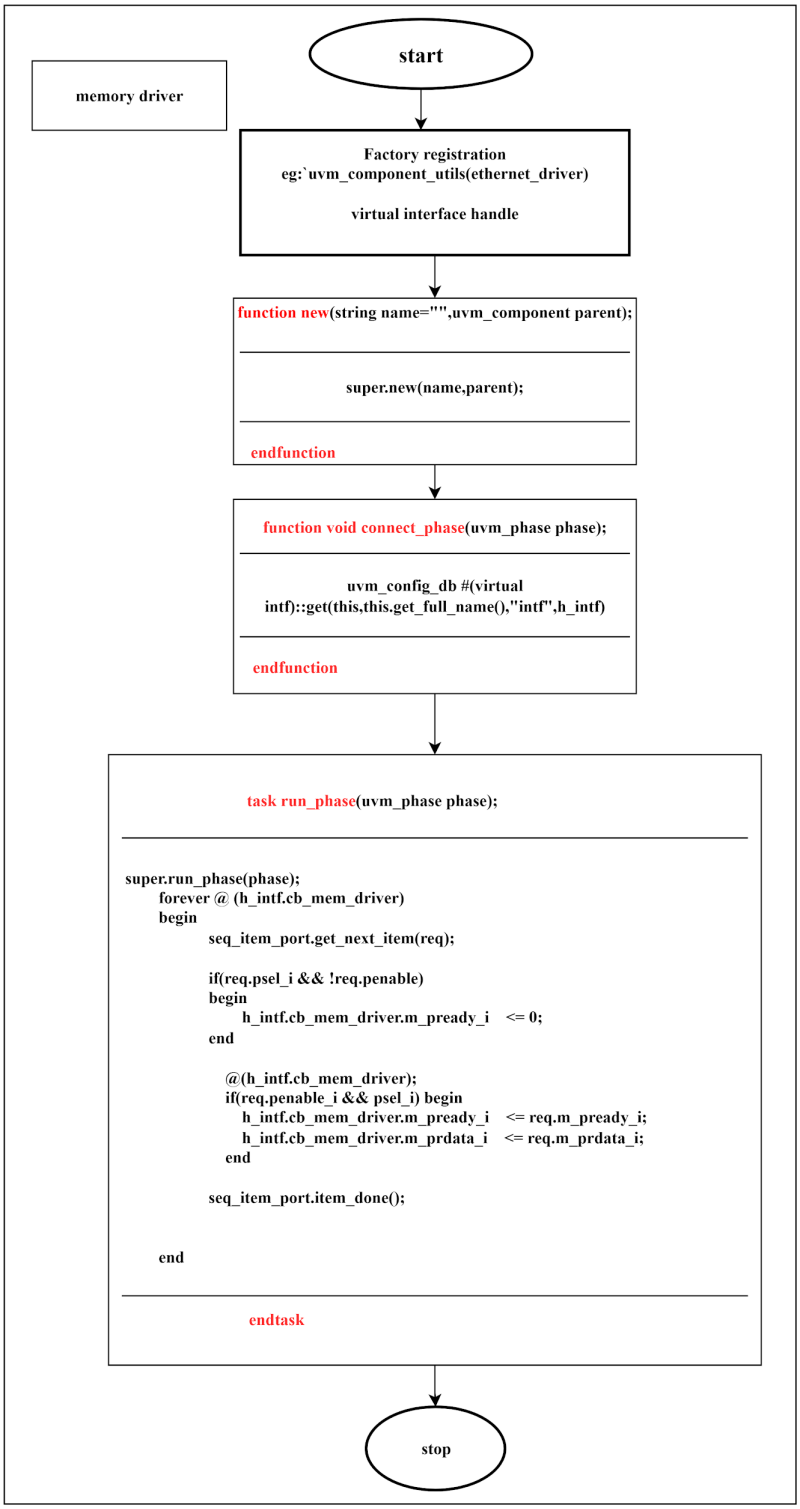


Figure 5.14 :Mem_driver Flow

5.2.3. TX Phy_Environment

- ***Build phase*** in host_environment is used to build phy_active_agent.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_env.

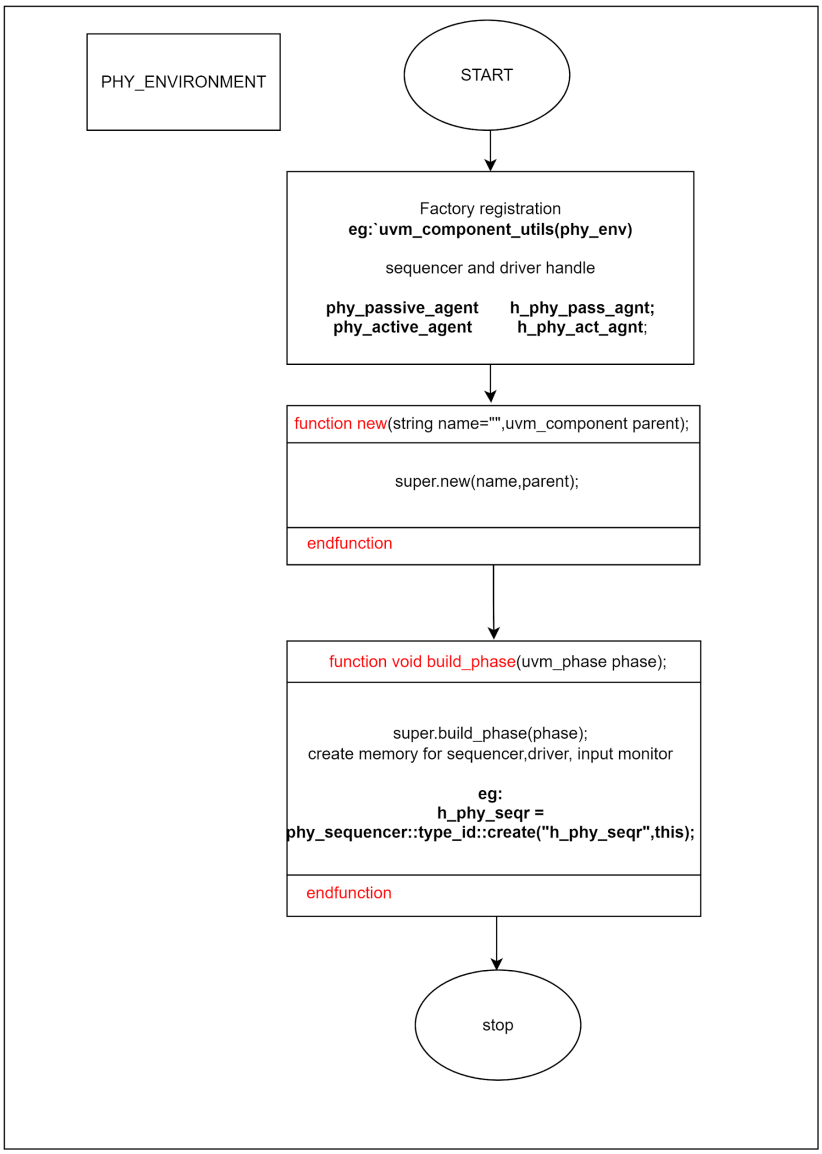


Figure 5.15 :Phy_environment Flow

5.2.3.1. TX_PHY_Active_agent

- ***Build phase*** in TX_PHY_Active_agent is used to build phy_sequencer , phy_driver .
- ***Connect phase*** in TX_PHY_Active_agent is used to connect mem_sequencer and mem_driver.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_agent.

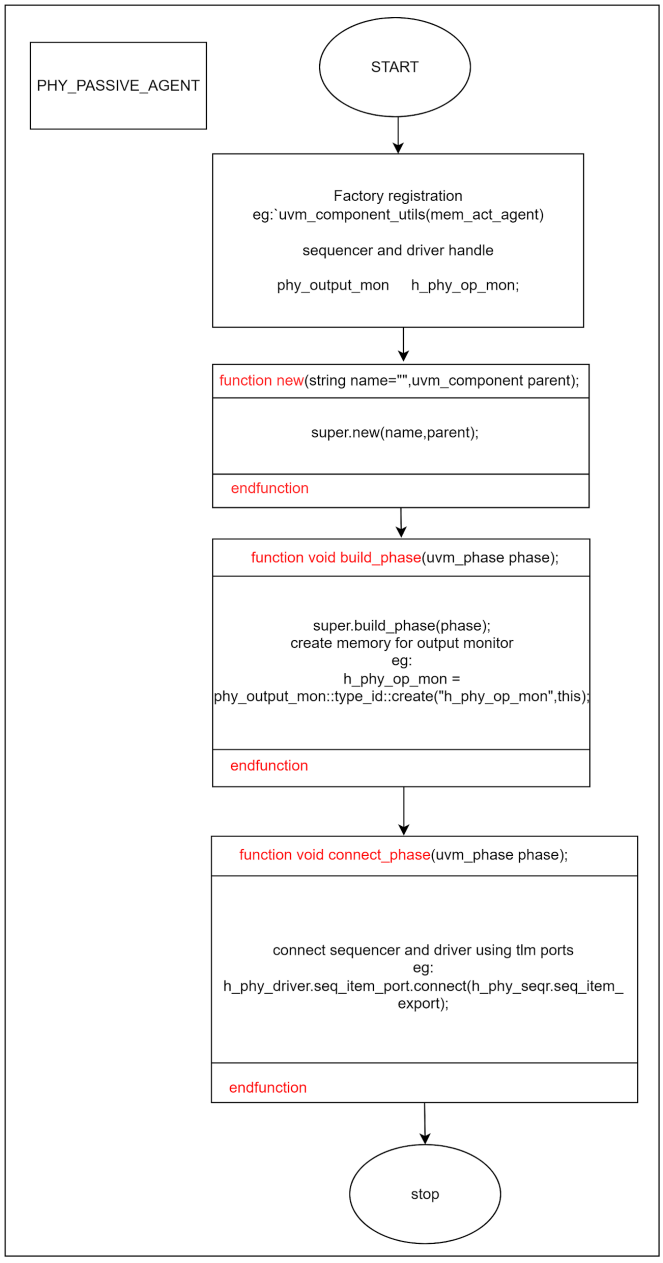


Figure 5.16 :TX_PHY_active_agent Flow

5.2.3.2.TX_phy_sequencer

- It provides a handshaking mechanism for TX_phy_sequence and TX_phy_driver.
- **Constructor Functionality**:
    - Upon instantiation, the constructor in PHY_environment is used for calling the constructor of base class : uvm_sequencer.

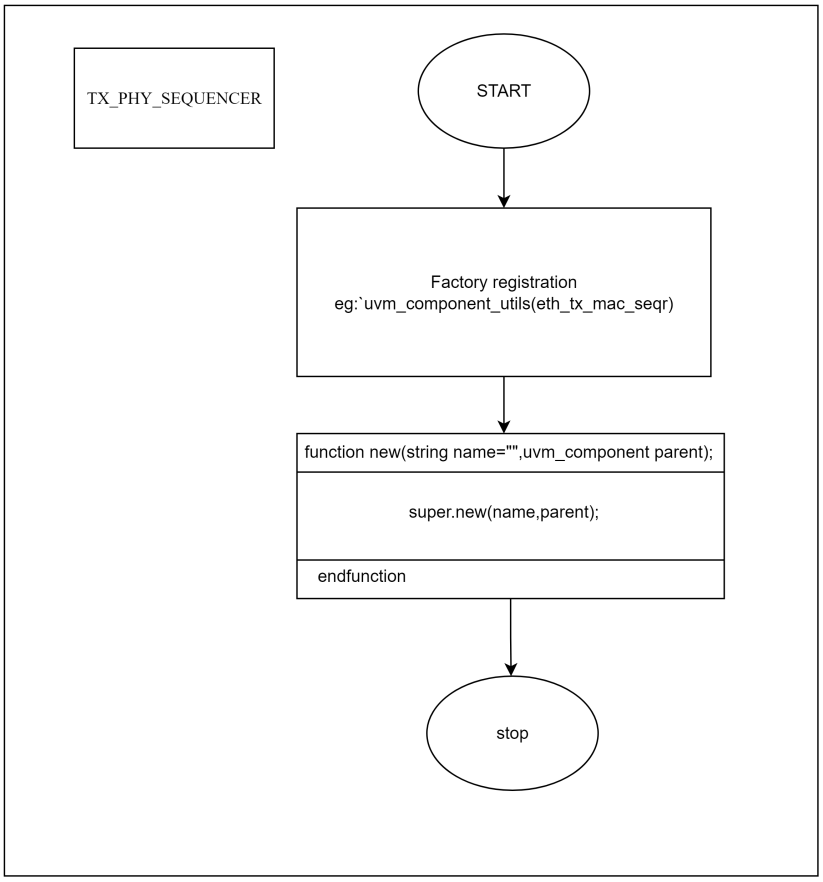


Figure 5.17 : TX_phy_sequencer Flow

5.2.3.3.TX_phy_driver

- Rx_phy_driver is used to drive input signals of the rx_mac onto the interface.
- It contains handle of a virtual interface
- ***Connect phase*** in host_driver is used to get the virtual interface from config_db.
- ***Run phase*** is used to drive the MEM_APB input signals on to the interface and it is activated at every cb_rx_mac_driver clock.
- Following input signals of apb are driven to the interface from host_apb from req(sequence_item).
  → MTXD(4)
  → MTxErr

→ MTXEN
→ MCRs
→ MCLK

- *Run phase executes the following code to drive a sample*
  get_next_item(req)
  Virtual_interface.cb_apb_driver.signal_name <= req.signal_name;
  item_done()
- **Constructor Functionality**:
  - Upon instantiation, the constructor in host_environment is used for calling the constructor of base class : uvm_driver.

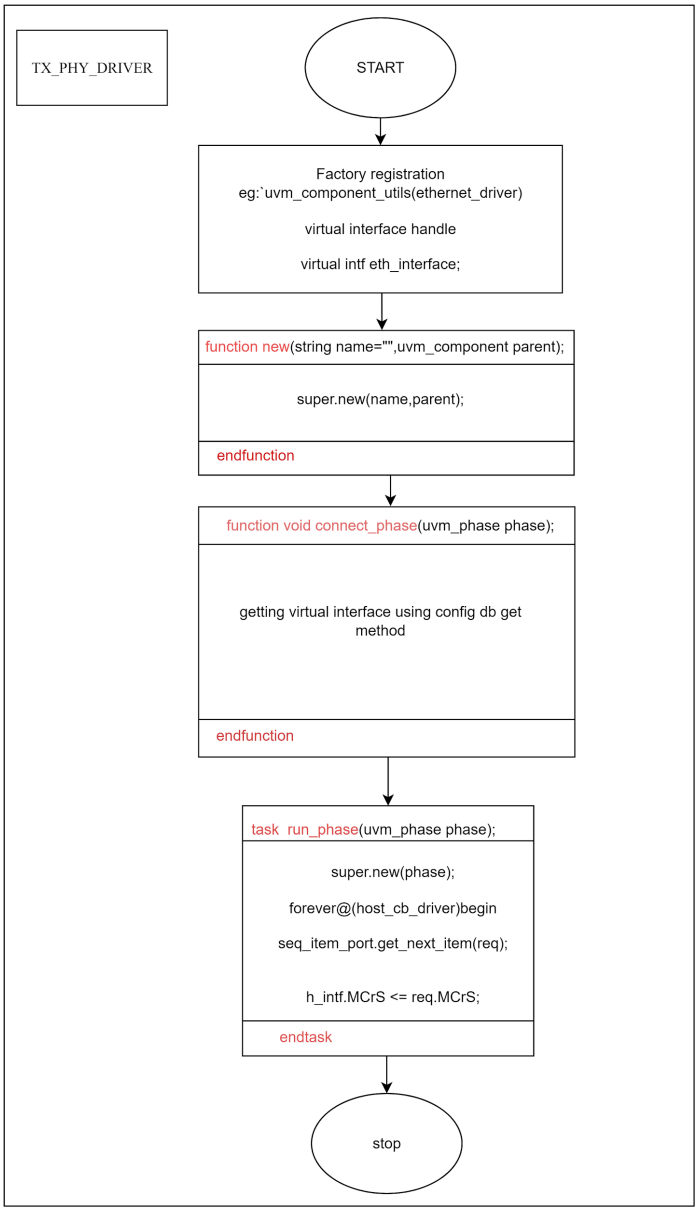


Figure 5.18 :TX_phy_driver Flow

5.2.3.4.TX_PHY_Passive_agent

- ***Build phase*** in TX_PHY_Passive_agent is used to set config db.
- ***Connect phase*** in TX_PHY_Passive_agent is used to connect.

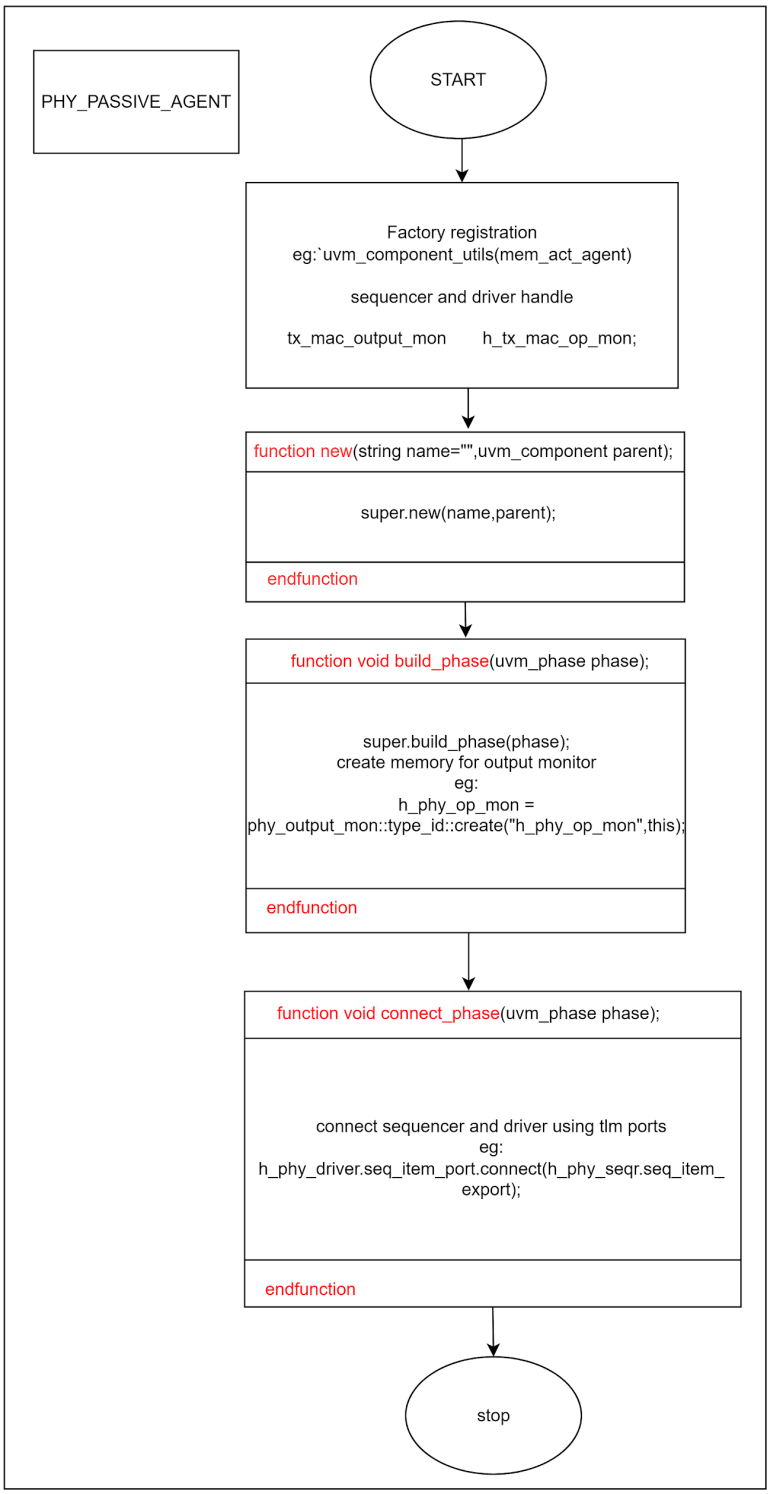


Figure 5.19 :TX_PHY_Passive_agent Flow

5.2.3.5.Output_Monitor

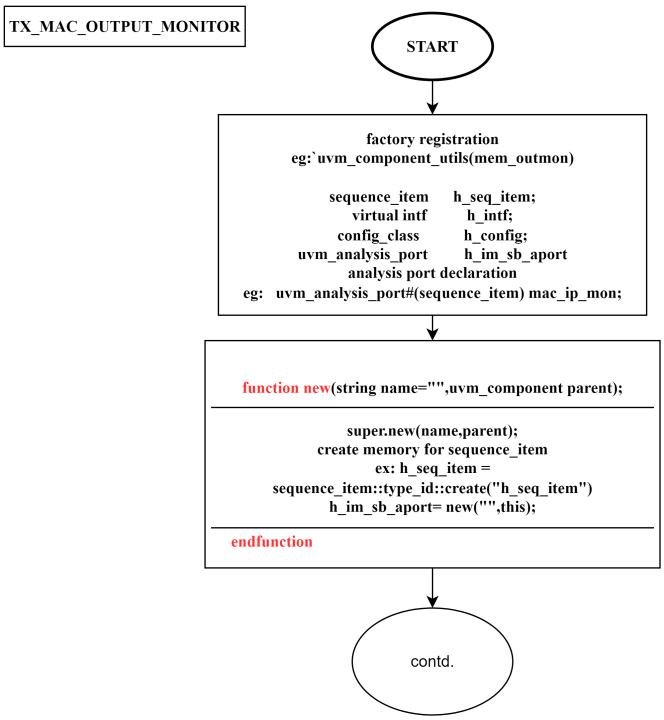
- Output_monitor Takes the input signals from the interface.
- Contains handle of sequence_item, virtual interface, and port type analysis_port.
- ***Build_phase*** is used to construct sequence_item and analysis_port.
- ***Connect_phase*** is used to get config_class access
- ***Run_phase*** is used to take the following input signals from interface

  →MTXD(4)

  →MTXEN

  →MTxErr

- The output monitor compares packet fields from the preamble to the length field and CRC.
- Comparisons are based on the nibble counter.
- The nibble counter determines the fields based on NOPRE conditions configured in the moder register.
- After checking the mentioned fields, the output monitor writes the MTxD into the analysis port to send data to the scoreboard for payload comparisons.
- The payload is compared in the scoreboard with the design payload data.
- Once the payload comparison is complete, CRC is calculated from the field data (Destination address, Source address, Length, Payload, and CRC) and compared with the magic number.



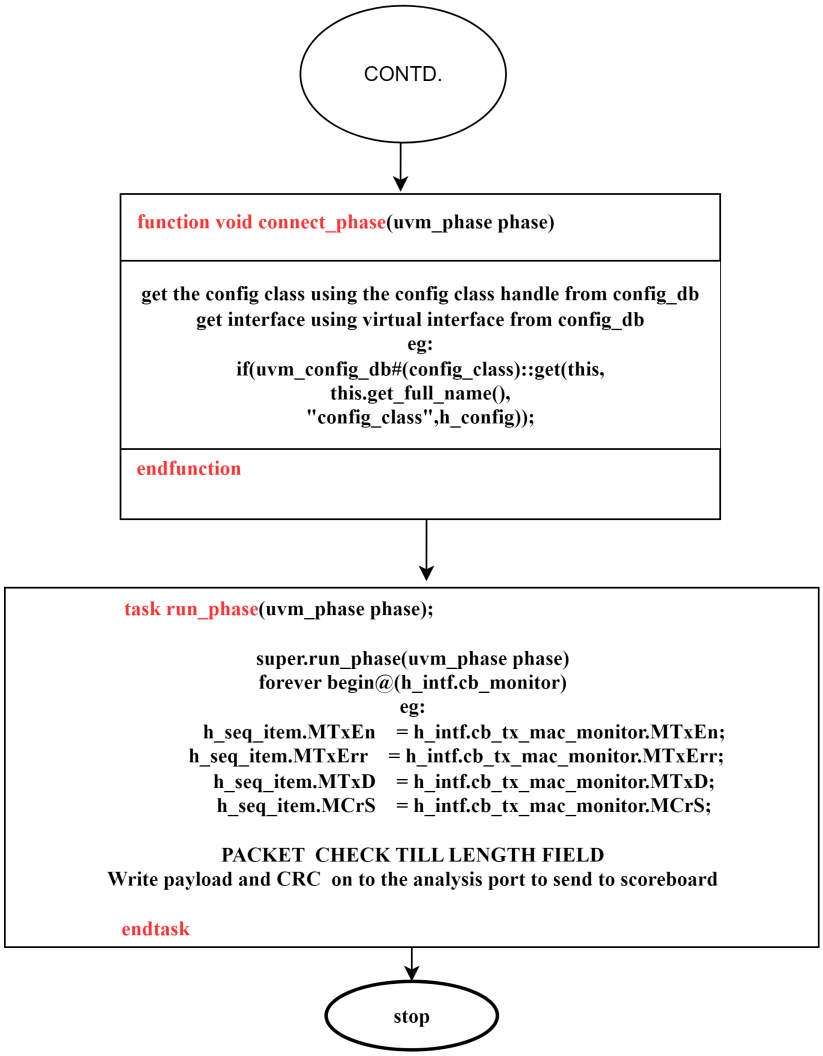


Figure 5.20 :OUTPUT_monitor Flow

5.2.4. Score Board

- Compares payload from both input and output monitors.
- Contains write function for input and output monitor
- ***Runphase*** is used to compare the payload.
- Write function is activated when event is triggered from input monitor
- **Constructor Functionality**:
    - Upon instantiation, the constructor in rx_phy_scoreboard is used for calling the constructor of base class : uvm_scoreboard.

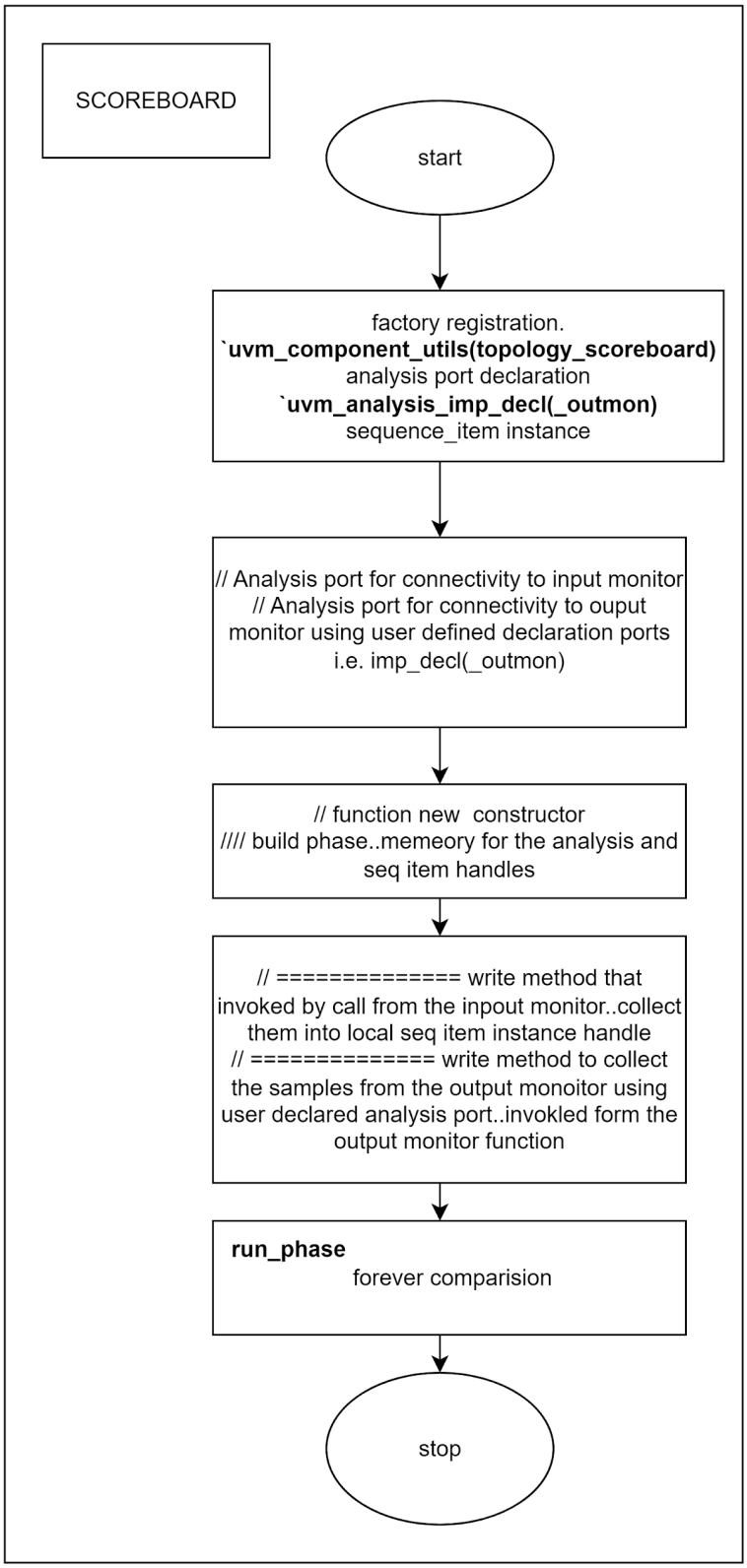


FIG 5.21 : UVM SCOREBOARD FLOW CHART

5.2.5. Sequence_item

The Sequence_item class encompasses all signals within the design, with all input signals declared using the **rand/randc** keyword along with their respective **constraints**.

Input Signals:

HOST APB

- ***rand*** bit PSELx_i,PENABLE_i,PWRITE_i, PRESETn_i
- ***rand*** bit[31:0] PADDR_i , PWDATA_i

HOST Memory

- ***rand*** bit M_PREADY_o
- ***rand*** bit [31:0]M_PRDATA_o

RX_PHY

- ***rand*** bit MRxDV,MRxErr, MCrS
- ***rand*** bit[3:0]MRxD

Output Signals:

HOST APB

- bit PREADY_o,int_o
- bit [31:0]PRDATA_o

HOST Memory

- bit M_PSELx_i,M_PENABLE_i,M_PWRITE_i, M_PRESETn_i
- bit[31:0] M_PADDR_i , M_PWDATA_i

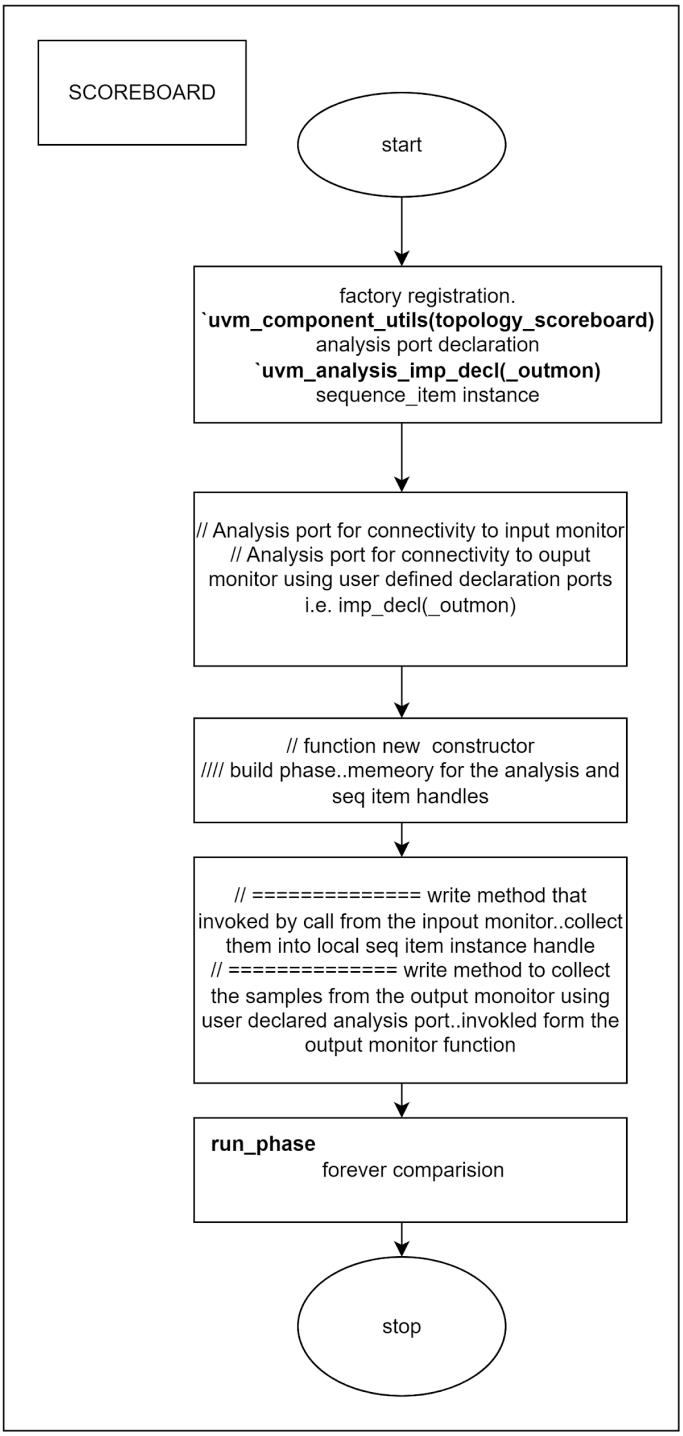


Figure 5.22 : Sequence-item Flow

5.2.6. APB_HOST Sequence

- APB_Sequence is “uvm_object” type and it is used to randomize the signals.
- It is a virtual sequence that contains sequences : each sequence indicates a testcase.
- A sample is randomized in “task body()” as follows

start_item(req);

Req.randimize() with {paddr_i==0;pwdata_i==’h0000co42;};

finish_item(req);

- The Host sequence is used to configure registers for every test case. Each test case is written in a separate sequence and that sequence is overrides the main sequence using factory overriding method.

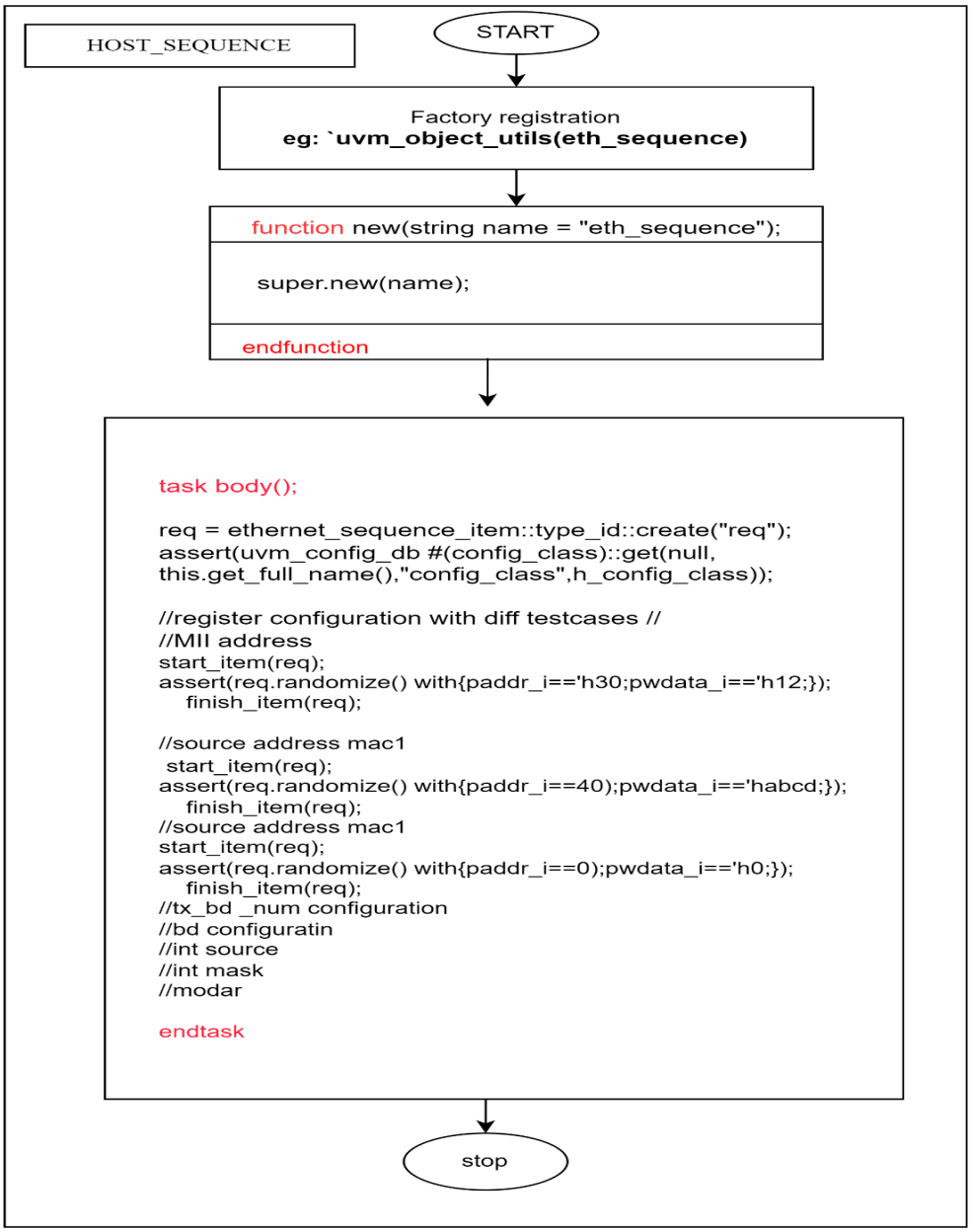


Fig 5.23 : APB Host Sequence Flowchart

5.2.7. APB_Memory Sequence

- APB_Sequence is “uvm_object” type and it is used to randomize the signals.
- It is a virtual sequence that contains sequences : each sequence indicates a testcase.
- A sample is randomized in “task body()” as follows

```
start_item(req);
        Req.randimize() with {pready==1;};
finish_item(req);
```

- The memory sequence is used to generate payload based on length configured in registers and number of buffer descriptors.

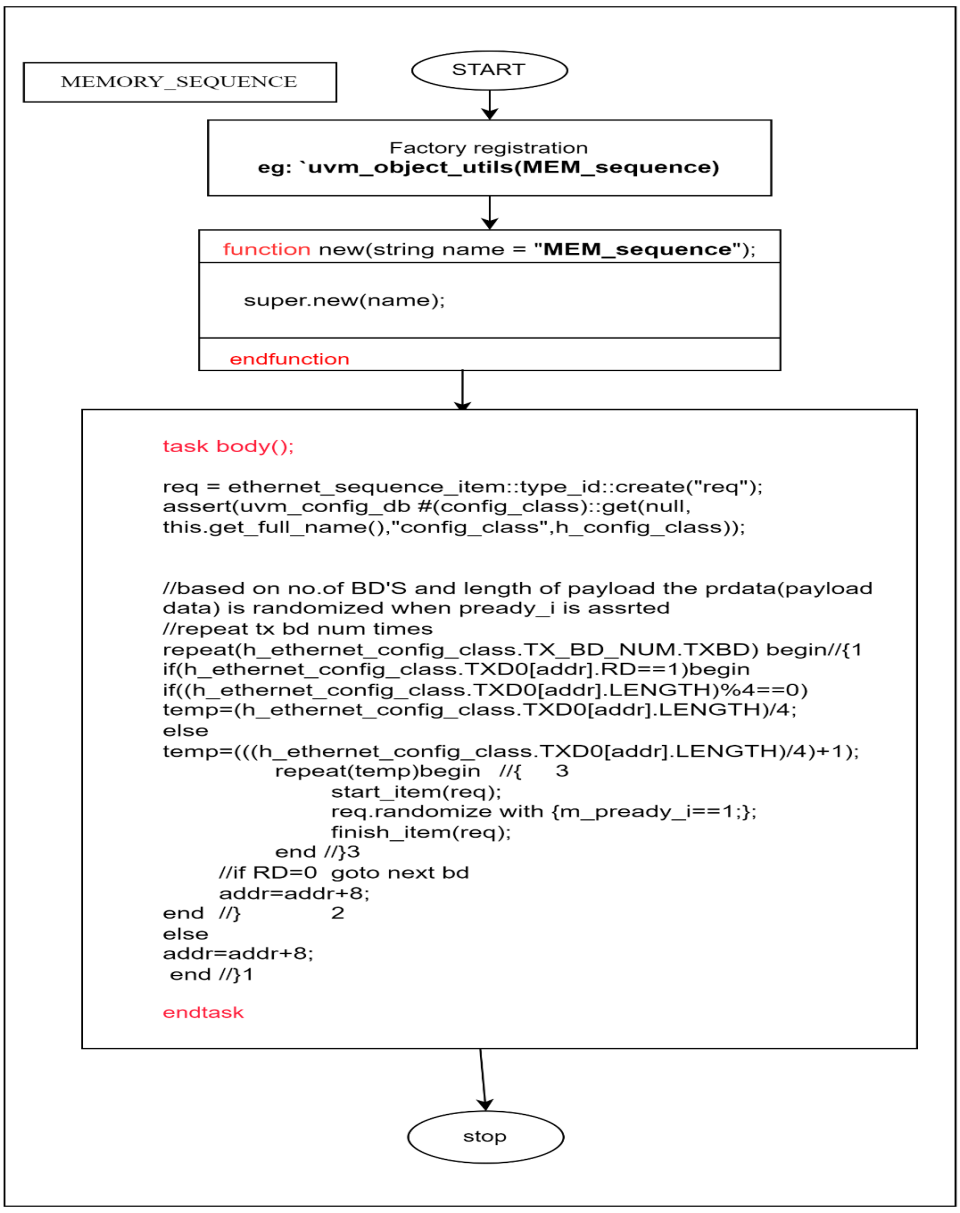


Fig 5.24 : APB slave sequence

5.2.8. TX_Mac_Sequence

- Rx_PHY_Sequence is “uvm_object” type and it is used to randomize the signals.
- It is a virtual sequence that contains sequences : each sequence indicates a testcase.
- A sample is randomized in “task body()” as follows

```
start_item(req);
        Req.randimize with {mcrs==0;};
finish_item(req);
```

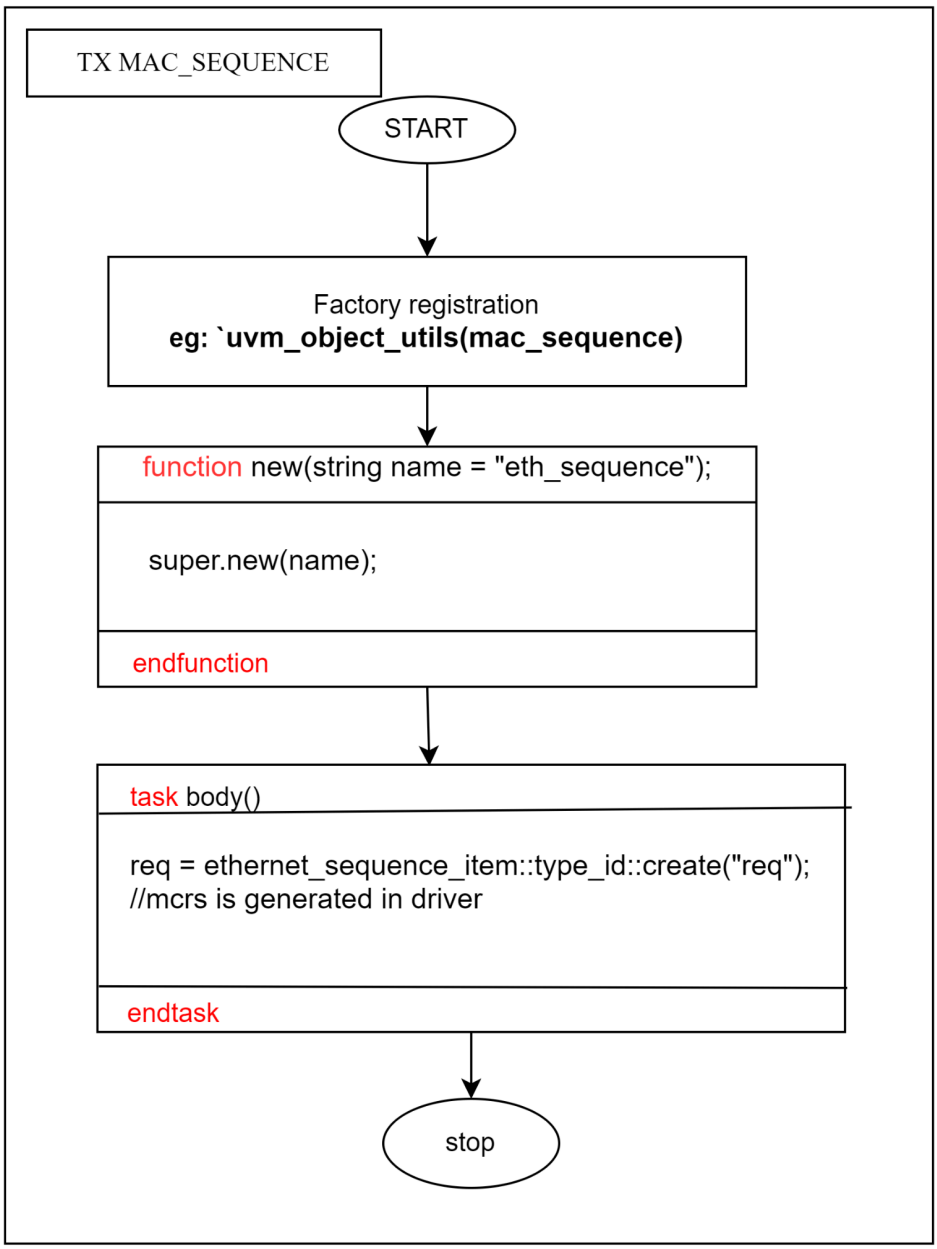
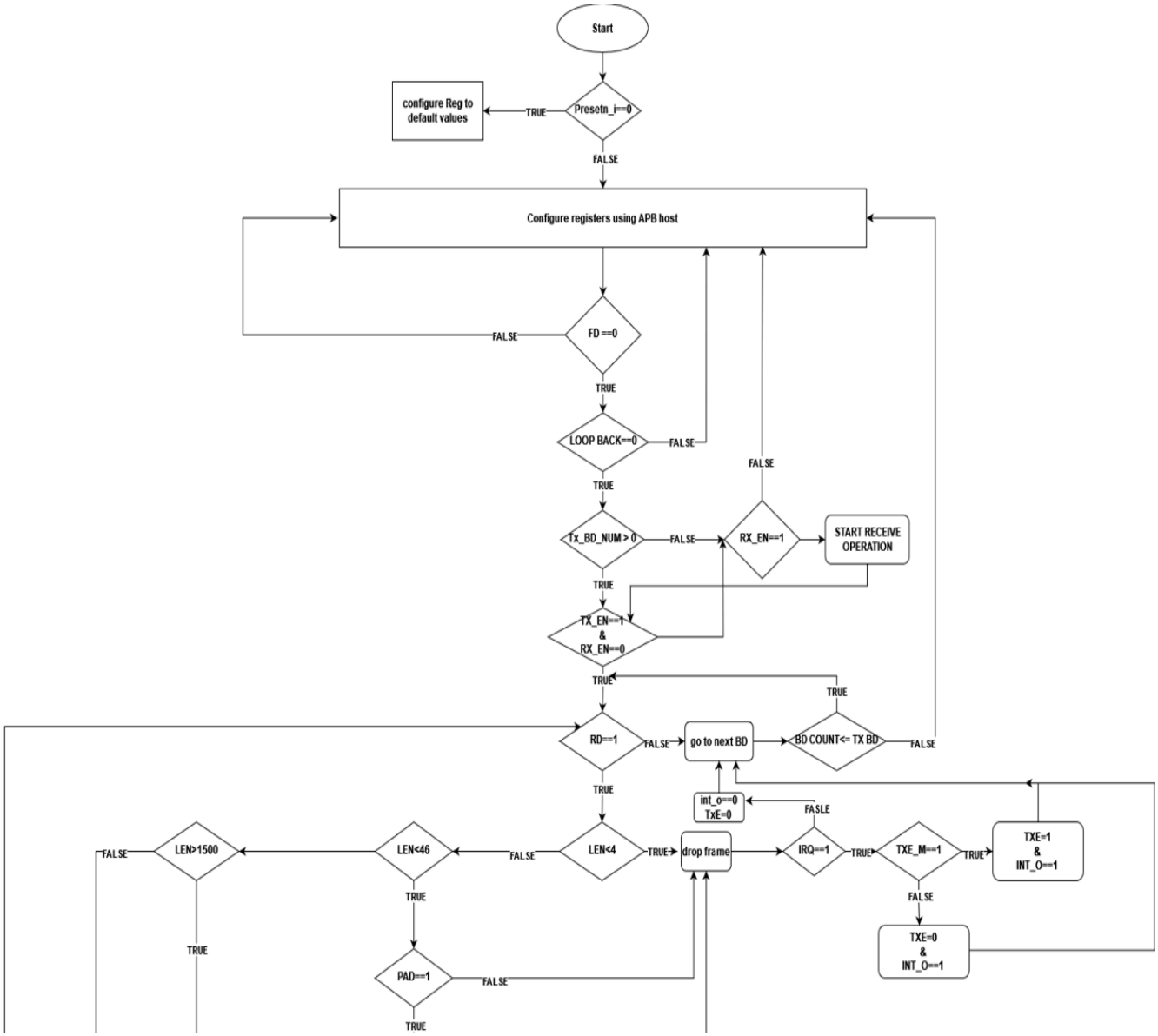
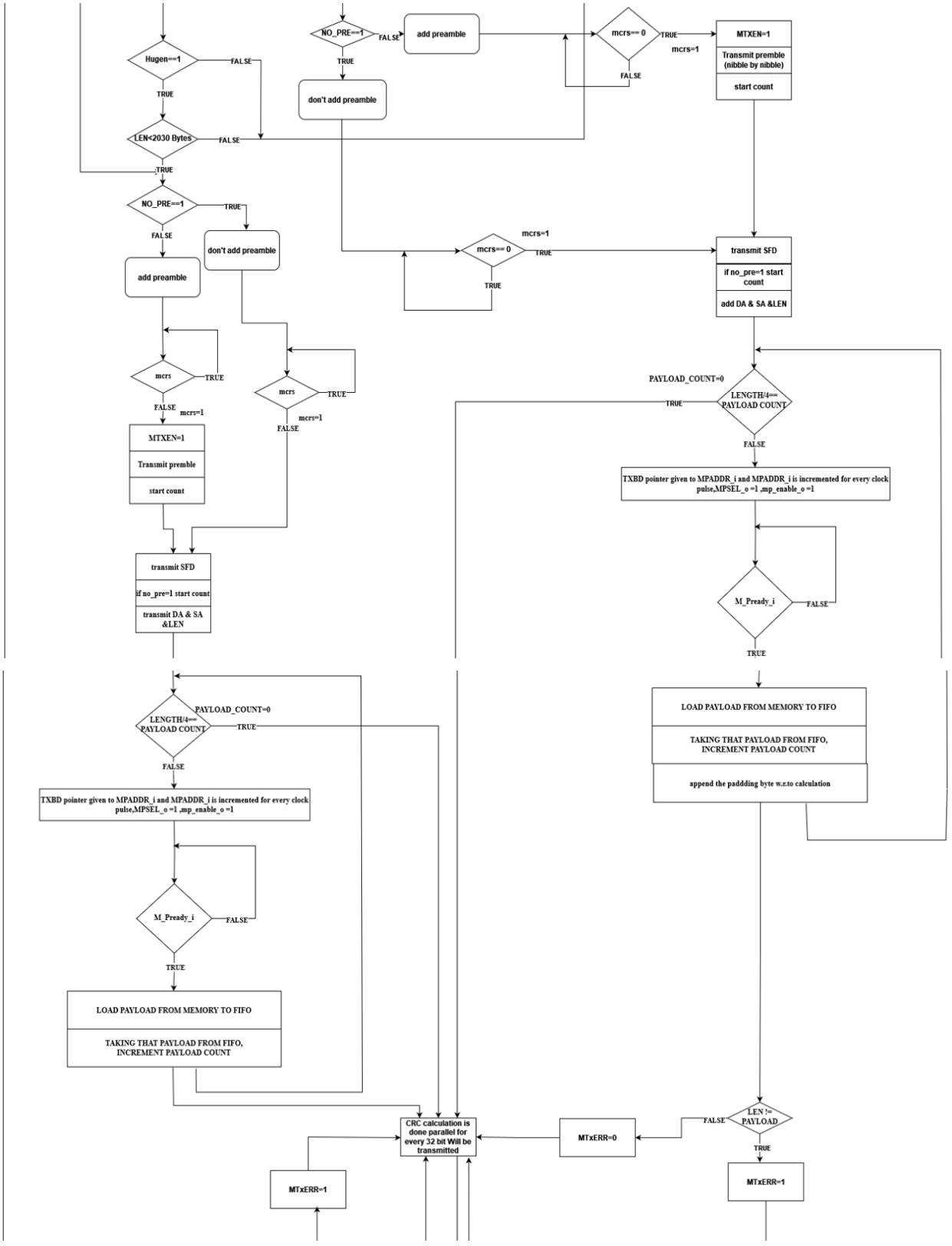


Fig 5.25 : Tx PHY Sequence

CHAPTER 6
FLOW OF OPERATION




Hugen==1
FALSE
TRUE
LEN<2030 Bytes
FALSE
TRUE
NO_PRE==1
TRUE
FALSE
don't add preamble
add preamble
mcrs
TRUE
FALSE
mcrs=1
MTXEN=1
Transmit premble
start count
transmit SFD
if no_pre=1 start count
transmit DA & SA &LEN
NO_PRE==1
FALSE
add preamble
don't add preamble
mcrs
TRUE
FALSE
mcrs=1
mcrs== 0
TRUE
FALSE
mcrs=1
MTXEN=1
Transmit premble (nibble by nibble)
start count
mcrs== 0
TRUE
mcrs=1
transmit SFD
if no_pre=1 start count
add DA & SA &LEN
PAYLOAD_COUNT=0
LENGTH/4== PAYLOAD COUNT
TRUE
FALSE
TXBD pointer given to MPADDR_i and MPADDR_i is incremented for every clock pulse,MPSEL_o =1 ,mp_enable_o =1
M_Pready_i
FALSE
TRUE
LOAD PAYLOAD FROM MEMORY TO FIFO
TAKING THAT PAYLOAD FROM FIFO, INCREMENT PAYLOAD COUNT
append the paddding byte w.r.to calculation
LENGTH/4== PAYLOAD COUNT
PAYLOAD_COUNT=0
TRUE
FALSE
TXBD pointer given to MPADDR_i and MPADDR_i is incremented for every clock pulse,MPSEL_o =1 ,mp_enable_o =1
M_Pready_i
FALSE
TRUE
LOAD PAYLOAD FROM MEMORY TO FIFO
TAKING THAT PAYLOAD FROM FIFO, INCREMENT PAYLOAD COUNT
LEN != PAYLOAD
FALSE
TRUE
MTxERR=0
CRC calculation is done parallel for every 32 bit Will be transmitted
MTxERR=1
MTxERR=1

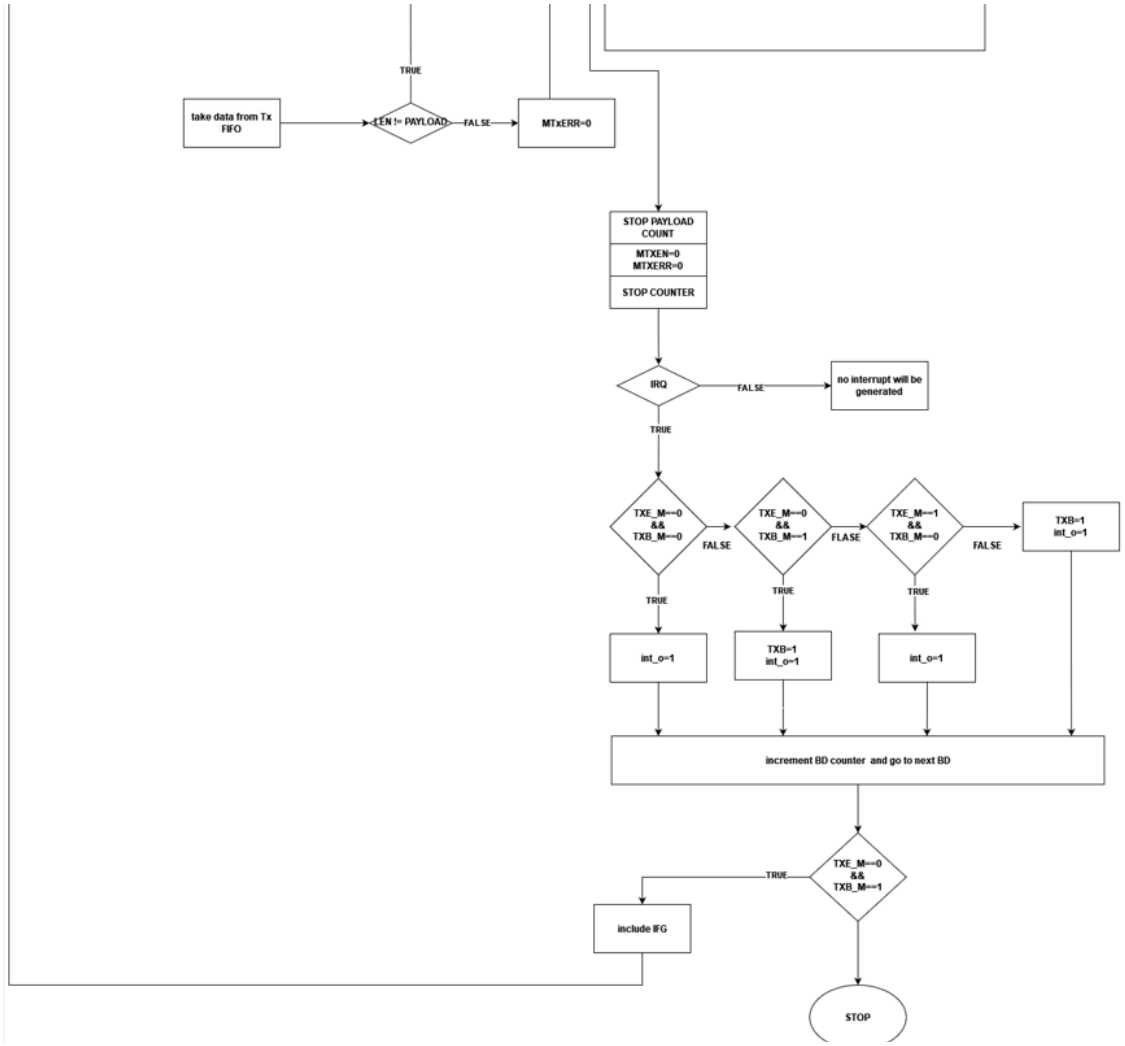


Fig 6.1 . Design Flowchart

EDA PLAYGROUND LINK : EDA_LINK