MOSCHIP INSTITUTE OF SILICON SYSTEMS (M-ISS)

VERIFICATION PLAN TRANSMITTER (ETHERNET_IP_CORE)

BY

BATCH_02.

M.DEVENDRA GOWD(M-ISSDV06 15)

L.NAGA LAKSHMI(M-ISSDV06 30)

O.AISHWARYA(M-ISSDV06 35)

B.SRIDHAR(M-ISSDV06 01)

1. INTRODUCTION :

We know that the OSI MODEL of a networking system includes seven layers like physical, data link, network, transport, session, presentation, and applications where each layer has some protocols. So, the first two layers in the OSI model like the physical layer and data link layer have a very popular set of Ethernet protocols. **Ethernet protocol** is available around us in various forms but the present Ethernet can be defined through the IEEE 802.3 standard. So, in this discussion we are verifying an Ethernet protocol working.

LAN networks are built on Ethernet. It is limited to the OSI model's layers 1 and 2 specifications. It is a subnet that allows other protocols, like the TCP/IP suite, to function; it is not a full network protocol.

The data link layer:

First Off To send messages, packet switching is employed. The primary use of packet switching is in computer-to-computer communication. A random amount of data is not transferred continuously over computer networks.

Rather, the data is transferred in discrete packets and small blocks by the network system. For this reason, packet switching networks are another name for computer networks.

MAC address:

Each station on a LAN's common transmission medium needs to have a distinct address. Each participant is assigned an Ethernet address, which is a MAC address (Medium Access Control Address), a physical addresss specific to the network device. Every network card manufacturer assigns a unique address number to each card, which is kept in the ROM of the device.

The 68 bits (6 bytes) that make up the MAC address are split into two groups of three bytes each. The first 26 bits make up an XEROC-issued manufacturing number. 6196302 manufacturing numbers are possible. For instance, Phoenix Contact's manufacturing number is 00A065h.

A serial number is formed by the lowest 26 bits. Each MAC address needs to be distinct.

The Ethernet dataframe:

The packet fields depicted in the figure are: Preamble, Start Frame Delimiter (SFD), destination and source addresses of the MAC frame, length or type field indicating the protocol type or length of the next field containing the MAC client data, field containing any necessary padding, and Frame Check Sequence (FCS) field containing a cyclic redundancy check value to detect errors in a received MAC frame.

If necessary, an Extension field is added (only for 1000 Mb/s half duplex operation). With the exception of the MAC Client Data, Pad, and Extension fields, which may include an integer number of octets between the minimum and maximum values that are established, all of these fields have a fixed size.

2.FEATURES OF ETHERNET :

2.1 FEATURES

- The very important function of ethernet is filling the physical layer by sending and receiving the serial bit streams over the medium.
- In this project we are framing a packet in TX path and deframing a packet in the RX path.
- In this we are using a packet switching method to send and receive data between two computer communications.
- This ethernet is mainly controlled by the signals like APB slave/master (Host),MAC interface signal,Reset signal.
- In this spec we configure different registers through which we can make data very stable and concurrent data transmission between layers.
- This protocol works on Half duplex mode for current specification
- This ethernet IP core consists of five modules like host interface,TX/RX module,MII management module.
- This majorly works on packet transmission and error detection from data link to physical layer and Frame reception from physical to data link layer.

2.2 RX PATH

- We can generate packets on the physical layer and pass through MRxD of the ethernet ip core using MRxCLK.
- This packet passes through the MAC.
- At the end, the frame will be stored in the memory using m_pwdata.

3.MERITS & DEMERITS :

3.1 MERITS :

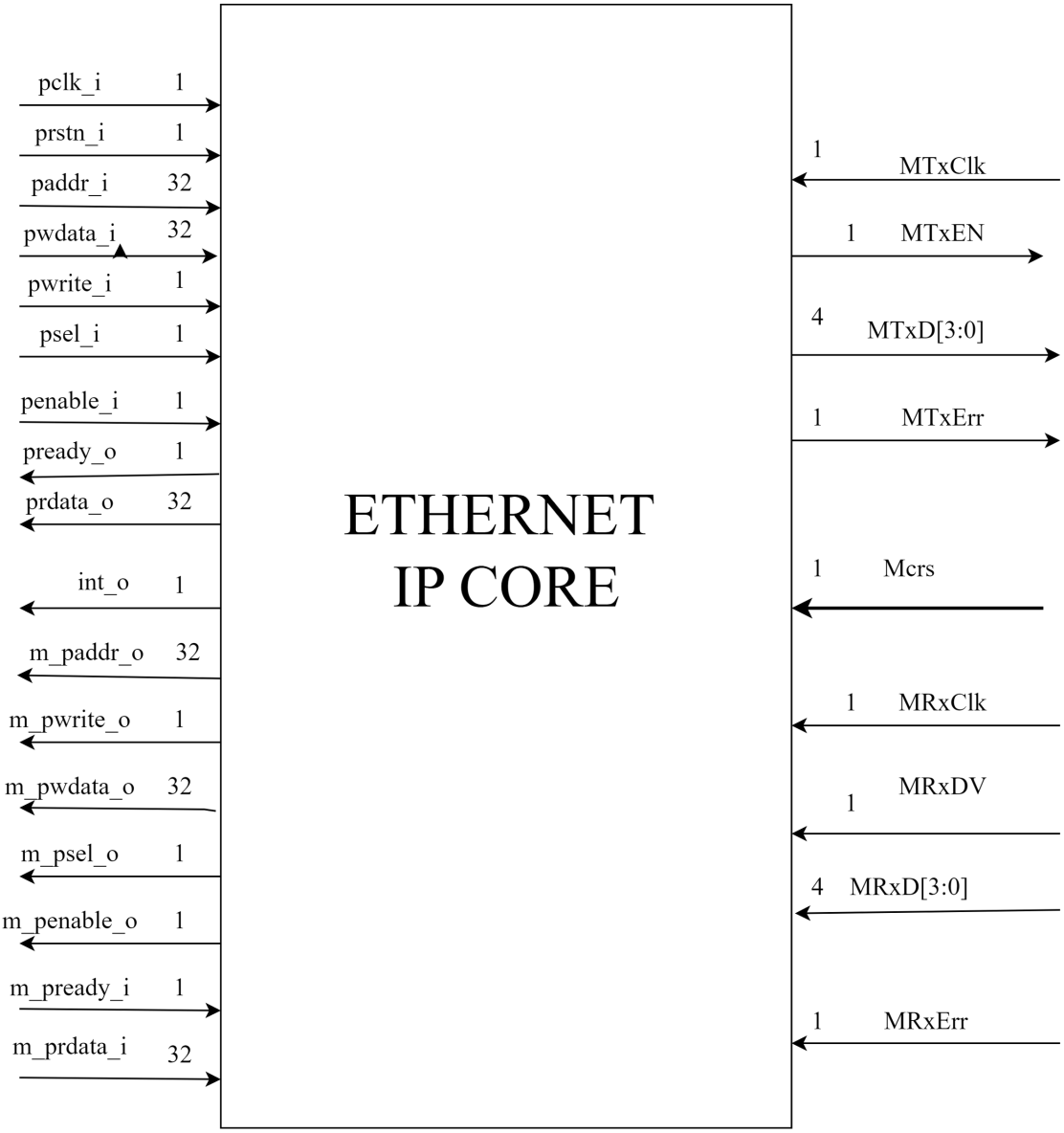
- The cost is relatively low.
- The system is compatible with older versions.
- It is generally resistant to noise.
- The quality of data transfer is good. It has a high speed.
- The system is reliable.
- Data security is ensured as it is compatible with common firewalls.

3.2 DEMERITS :

- Intended for more compact networks over shorter distances.
- Restricted capacity to move.
- The utilisation of lengthier cables may give rise to interference between signals.
- Performance is suboptimal when handling real-time or interactive applications.
- Transmission rates diminish as network traffic intensifies.
- Receivers do not confirm the successful receipt of data packets.
- Diagnosing issues becomes arduous when attempting to identify the precise cable or node responsible for the problem.

4. PIN DIAGRAM :

4.1 SIGNAL DESCRIPTION:



PORT	WIDTH	DIRECTION	DESCRIPTION
pclk_i	1	INPUT	This signal generates APB Bus Clock Input(25MHZ)

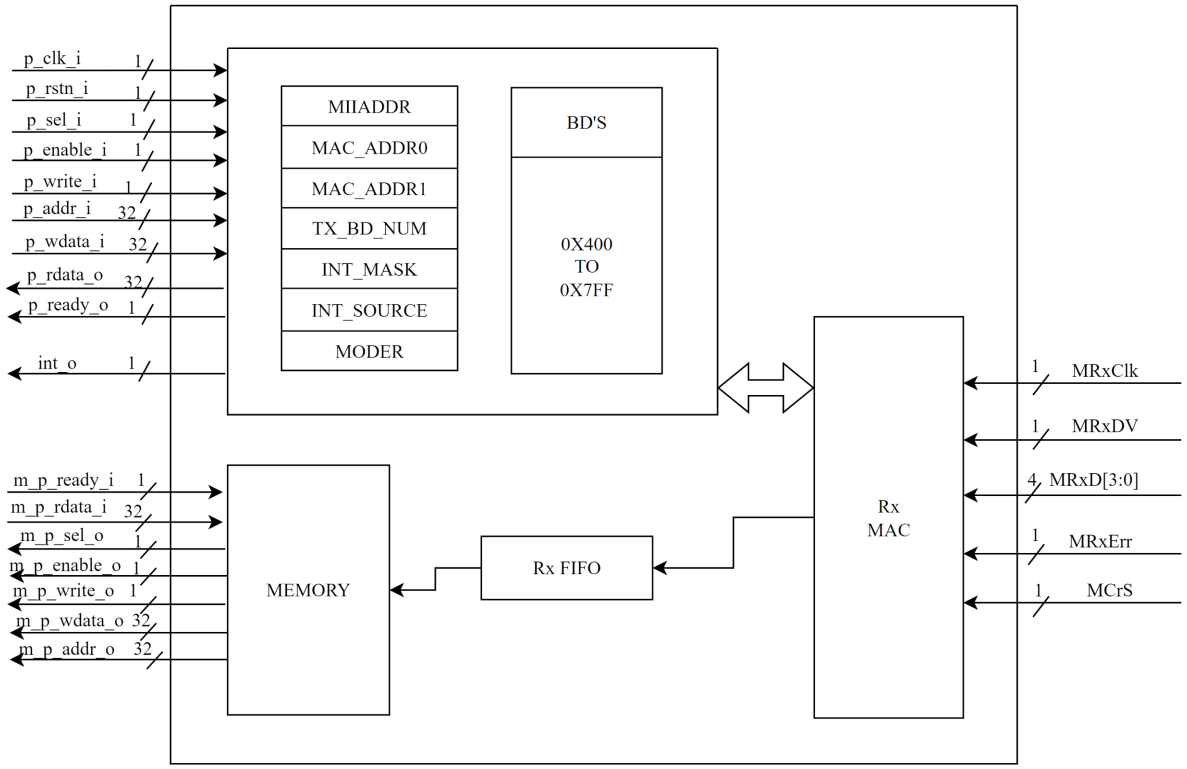
prstn_i	1	INPUT	This signal is used to set default values.
paddr_i	32	INPUT	Address Input
pwdata_i	32	INPUT	Data Input
prdata_o	32	OUTPUT	Data Output
psel_i	1	INPUT	Slave Select Input
pwrite_i	1	INPUT	shows a read cycle when asserted low and a write cycle when asserted high.
penable_i	1	INPUT	A legitimate transfer cycle is indicated by the indicator "Enable Input.
pready_o	1	OUTPUT	Peripheral ready Output.
int_o	1	OUTPUT	Interrupt Output: This signal contains a 1 after each BD transmission or receipt. The user should clear this interrupt and keep separate counters for good and faulty frames.
m_paddr_o	32	OUTPUT	Address Output
m_prdata_i	32	INPUT	Data Input
m_pwdata_o	32	OUTPUT	Data Output
m_psel_o	1	OUTPUT	Slave Select Output
m_pwrite_o	1	OUTPUT	When asserted high, the Write Output indicates a Write Cycle; when asserted low, it indicates a Read Cycle.
m_penable_o	1	OUTPUT	Enable Output marks the start of a legitimate transfer cycle.
m_pready_i	1	INPUT	Acknowledgment Input

PORT	WIDTH	DIRECTION	DESCRIPTION
MTxClk	1	INPUT	This Sends a Nibble or Symbol Clock through. The MTxClk signal is supplied by the PHY. It runs at either 2.5 MHz (10 Mbps) or 25 MHz (100 Mbps) in frequency. The timing reference for the transfer of MTxD[3:0], MtxEn, and MTxErr is the clock.
MTxD[3:0]	6	OUTPUT	This pin Sends Data Nibble through. Bits of data are transmitted as signals. They are in sync with MTxClk's rising edge. PHY acknowledges the MTxD when MTxEn is asserted.

MTxEn	1	OUTPUT	Send enabling. This signal, when asserted, tells the PHY that the data MTxD[3:0] is legitimate and that the transmission may begin. The first tidbit of the preamble begins the transmission. Until the PHY receives all of the nibbles that need to be transmitted, the signal is still asserted. It desserts after the last nibble of a frame, before the first MTxClk.
MTxErr	1	OUTPUT	Send Error Coding. This signal indicates that there has been a transmit coding error by causing 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 when asserted for one MTxClk clock period while MTxEn is also asserted. It is a pulse, MTXErr.
MRxClk	1	INPUT	Acquire the Symbol Clock or Nibble. The MRxClk signal is supplied by the PHY. It runs at either 2.5 MHz (10 Mbps) or 25 MHz (100 Mbps) in frequency. The clock is used as a timing reference for the reception of MRxD[3:0], MRxDV, and MRxErr.
MRxDV	1	INPUT	Obtain Valid Data. In order to let the Rx MAC know that it is presenting the legitimate nibbles on the MRxD[3:0] signals, the PHY asserts this signal. Synchronously, the signal is asserted to the MRxClk. From the first recovered nibble of the frame to the last recovered nibble, MRxDV is claimed. After the last nibble, it is then deasserted before the first MRxClk.
MRxD[3:0]	6	INPUT	Get Data Nibble here. These signals are the nibbles of received data. They are in sync with MRxClk's rising edge. The PHY transmits a data nibble to the Rx MAC upon assertion of MRxDV. Seven bytes of a preamble and a fully formed SFD need to be passed across the interface in order for the frame to be correctly interpreted.
MRxErr	1	INPUT	Error Received. In order to notify the Rx MAC that a media error was discovered during the transmission of the current frame, the PHY asserts this signal. MRxErr is asserted for one or more MRxClk clock periods before being deasserted. It operates synchronously with the MRxClk. It is a pulse, MRxErr.

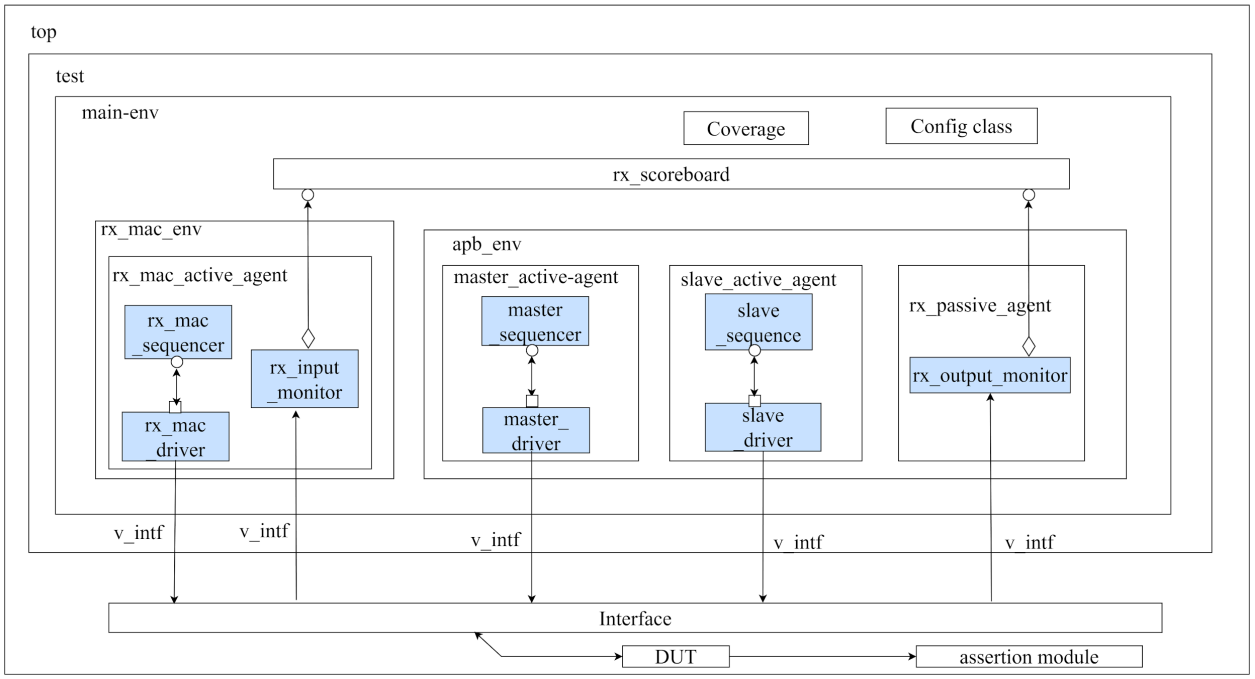
MCrS	1	INPUT	Sense of Carrier. When the medium is detected to be in a non-idle state (non idle state- HalfDuplex Tx/Rx), the PHY asynchronously asserts the carrier sense MCrS signal. This signal deasserts to indicate that the media is in an idle state, at which point the transmission can begin. Note: The MAC must ensure that MCrS = 0 (zero denotes an idle state of the medium) in order to initiate the Half Duplex Tx/Rx mode. After the data is available on the channel, PHY must ensure that MCrS = 1 remains at 1, as MAC bears no responsibility for this and may exhibit unexpected behaviour.

4.2.ARCHITECTURE OF RX:



5. UVM VERIFICATION ENVIRONMENT

RX_ENVIRONMENT



5.1 TOP MODULE

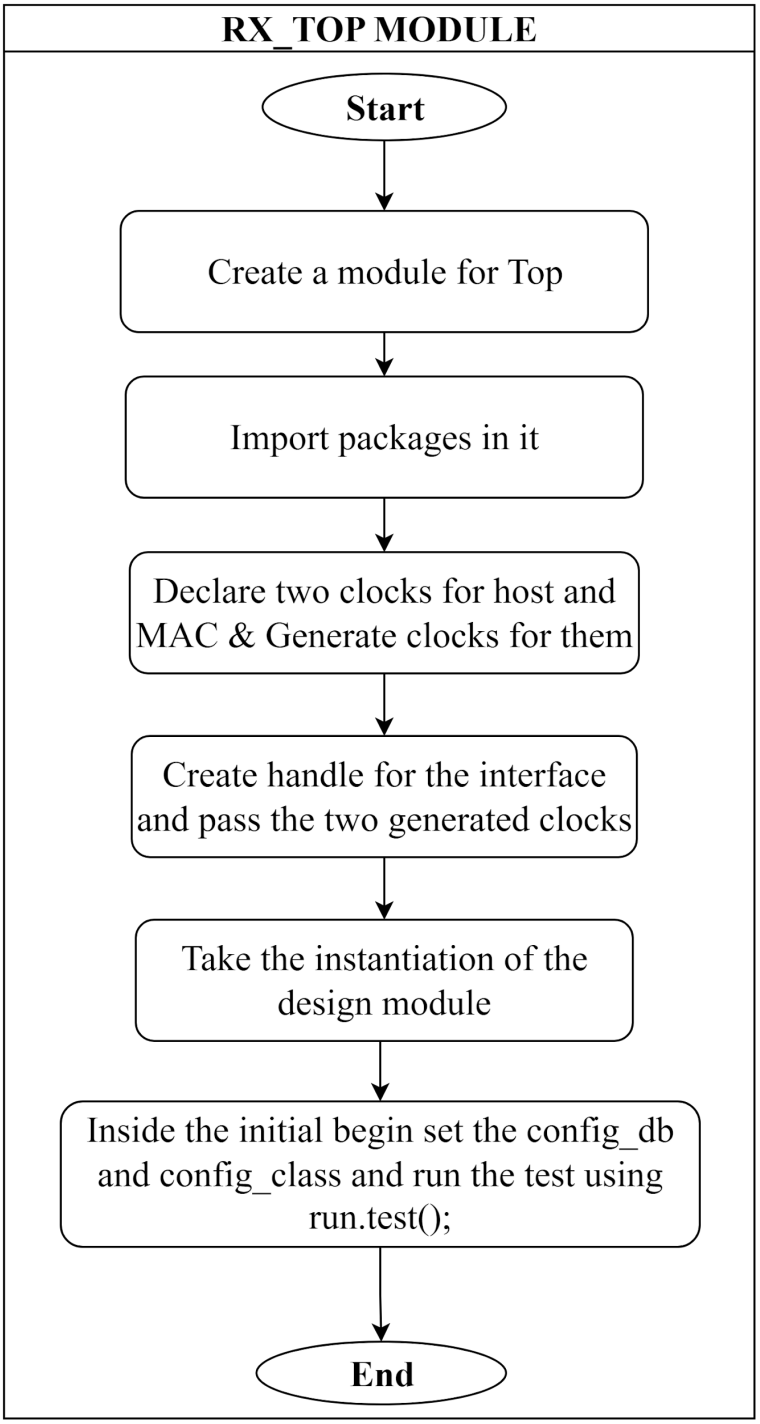


Fig:5.1 TOP MODULE

DESCRIPTION:

- In the top module we are going to import the uvm inbuilt macros and package.
- Clock generation will take place in top module

- Creating the handles for rx_test class and interface
- Instantiating the DUT in the top module by order or by the name.
- Setting the config_db and config_class inside the initial begin, and then using the run.test() command to execute the test.

5.2 TEST CLASS

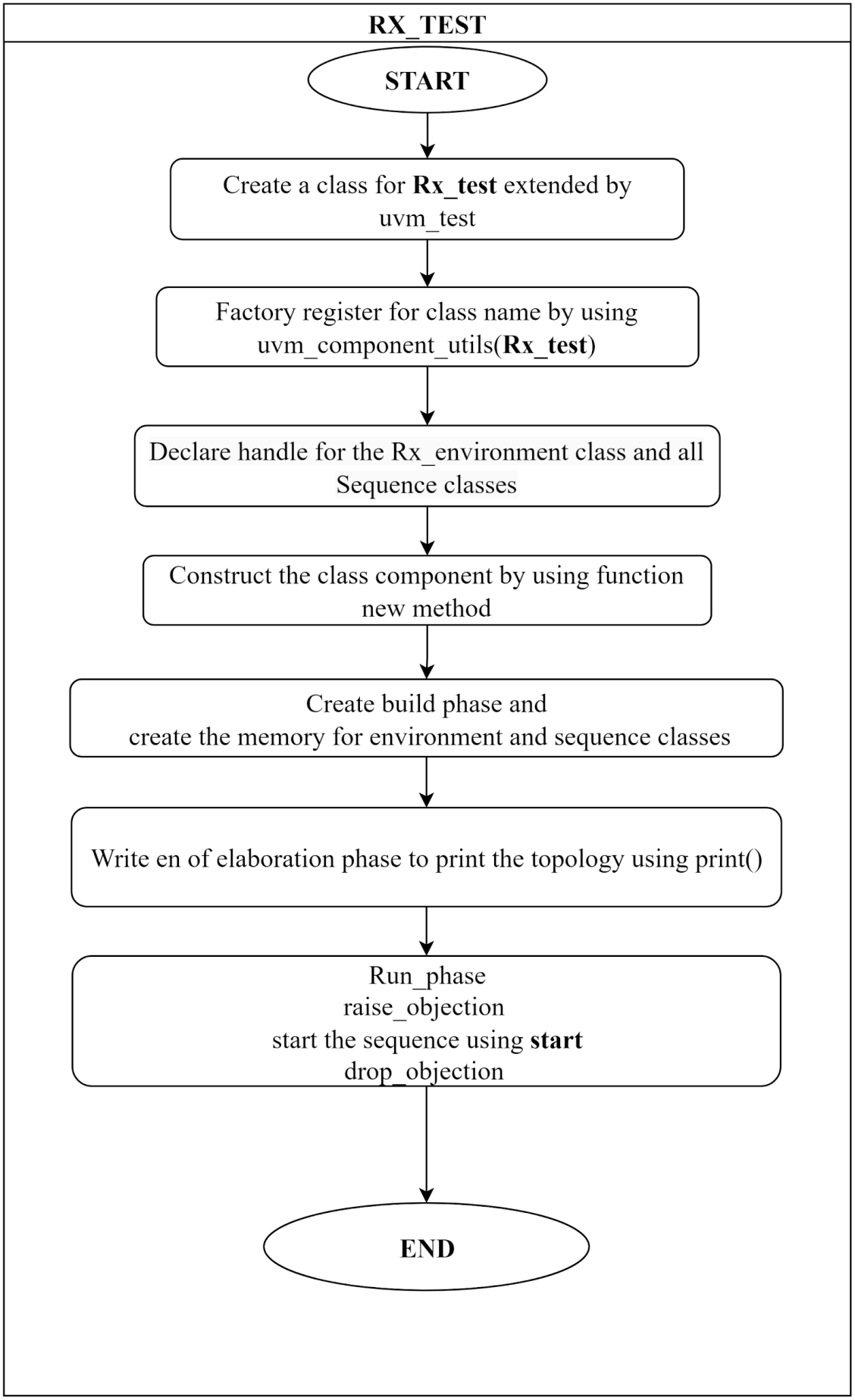

RX_TEST
START
Create a class for Rx_test extended by uvm_test
Factory register for class name by using uvm_component_utils(Rx_test)
Declare handle for the Rx_environment class and all Sequence classes
Construct the class component by using function new method
Create build phase and create the memory for environment and sequence classes
Write en of elaboration phase to print the topology using print()
Run_phase
raise_objection
start the sequence using start
drop_objection
END

Fig:5.2 TEST CLASS

DESCRIPTION:

- Creating the test class and Extending the class from the uvm_test class which is the base class.
- Factory registration using uvm macros (`uvm_component_utils(test).
- Declaring the handles for Rx_environment class and all the sequence classes which are used to generate the stimulus.
- Constructing the memory for class components using a function new method.
- In the build phase creating the memory for above declaring class handles by using create method.
- In the run phase, after raise_objection, we start the sequence using the start keyword, and after that, drop_objection is given.

5.3 TOP ENVIRONMENT

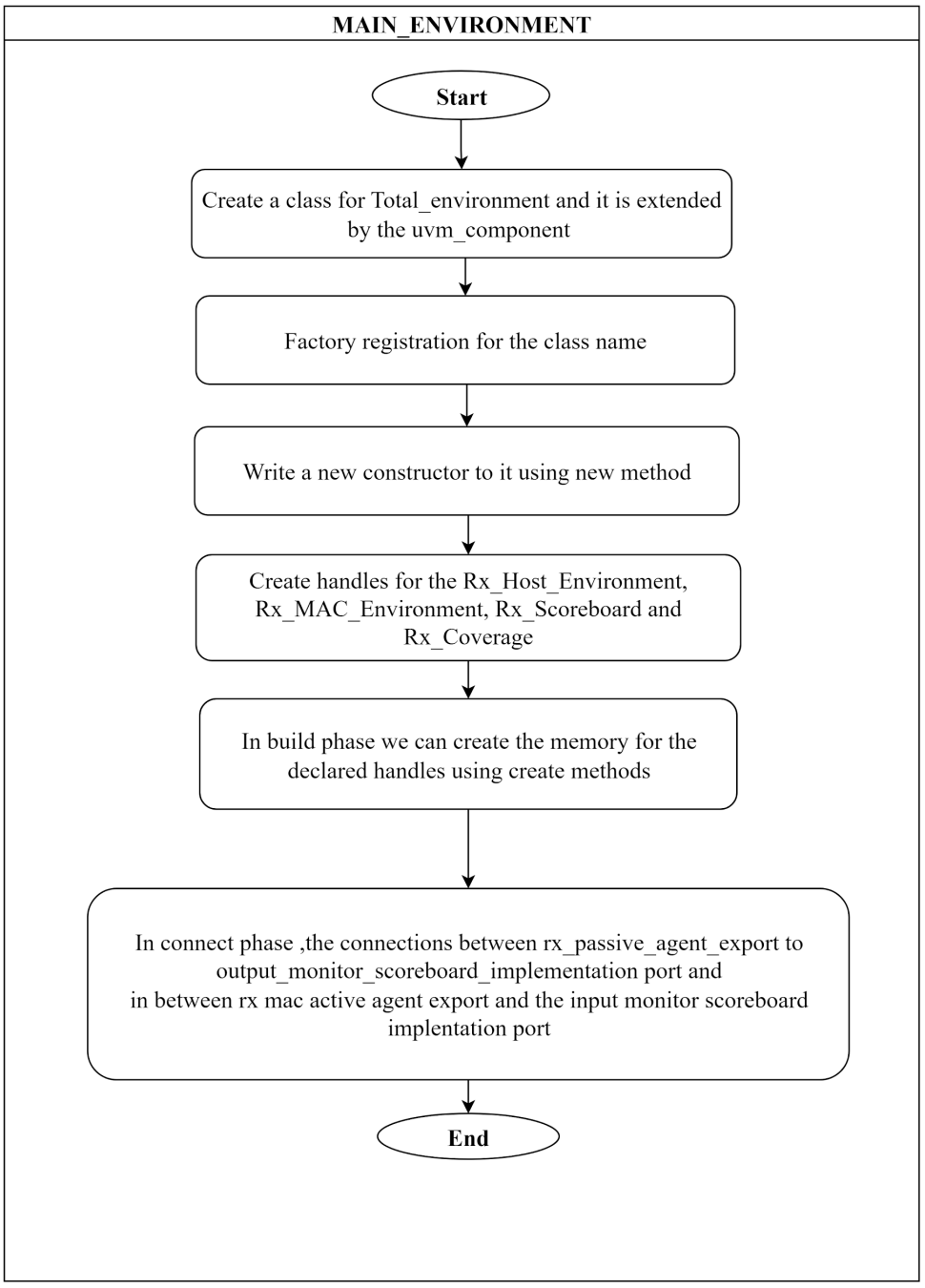


Fig:5.3 TOP ENVIRONMENT

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils
- Create the handles for Host_environment, MAC_environment, Rx_Scoreboard, Rx_coverage.
- Create a memory for the TOP_environment by using new() constructor.
- In the build phase create a memory for the above handles.

- In the connect phase, connections between output_monitor which is under the Host environment to Rx_Scoreboard, and also connection between input_monitor which is under the MAC environment to Rx_Scoreboard.

5.3.1 HOST ENVIRONMENT

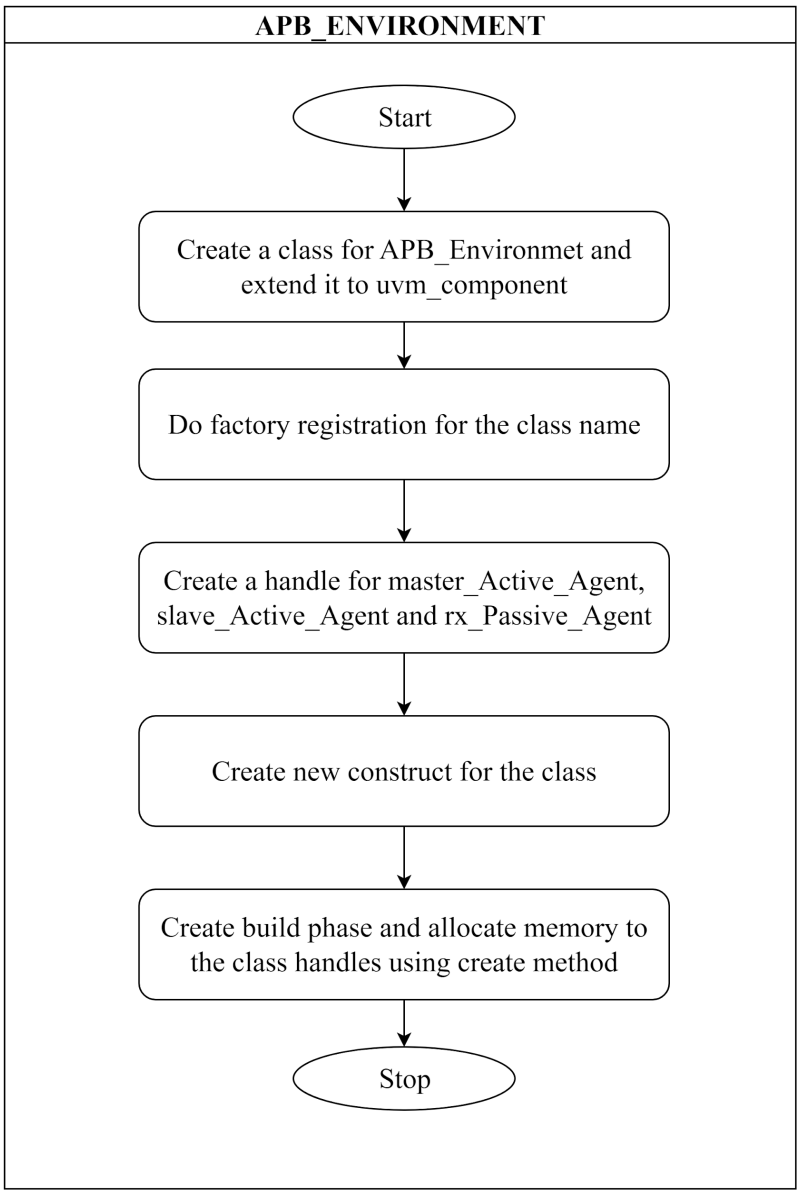


Fig:5.3.1 HOST ENVIRONMENT

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils
- Create the handle for Host_active_agent, Memory_active_agent, passive agent.
- Create a memory for the Host_environment by using new() constructor.
- In the build phase create a memory for the above handles.

5.3.1.1 HOST ACTIVE AGENT CLASS

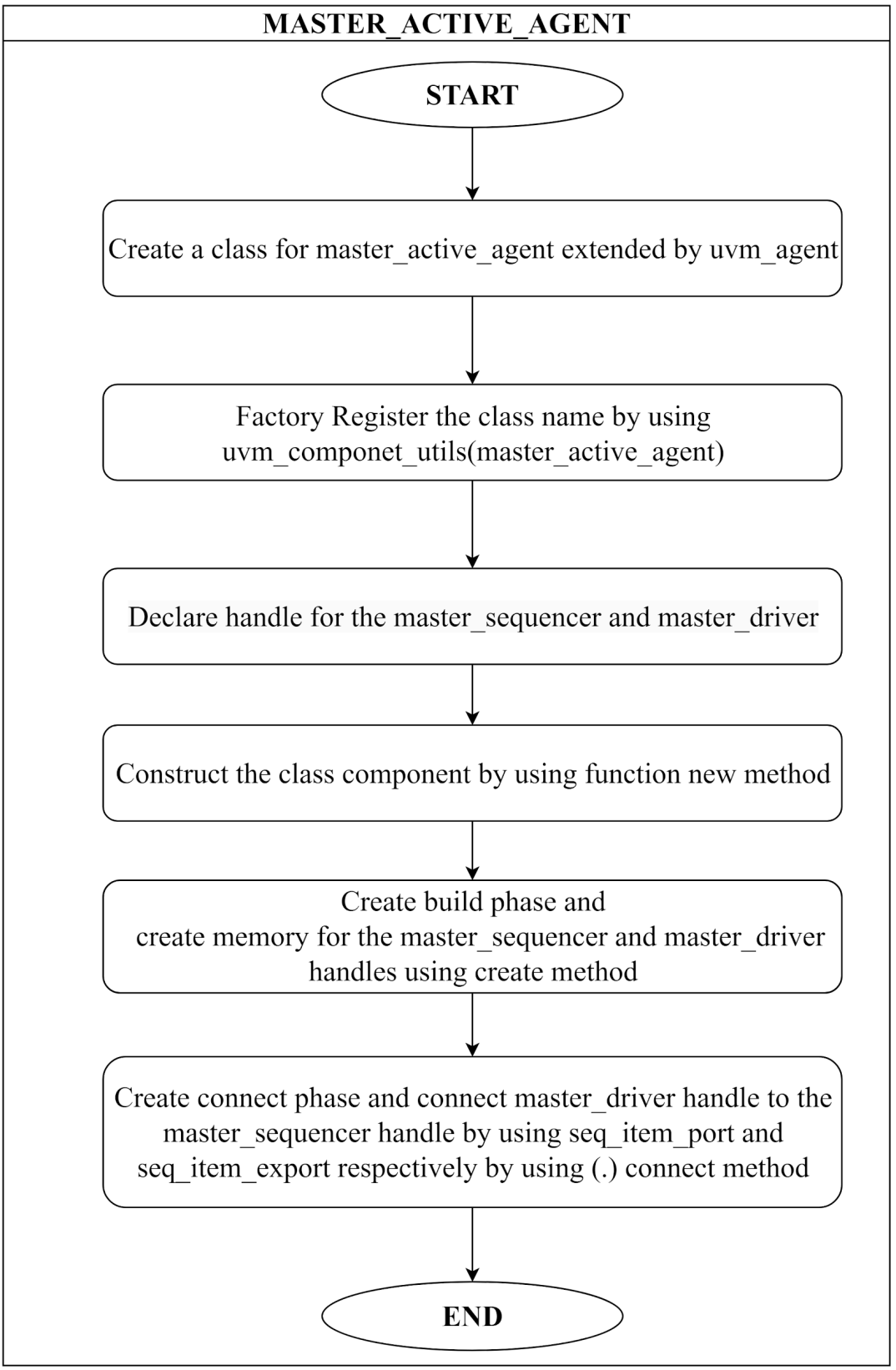


Fig:5.3.1.1 ACTIVE AGENT CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.

- Factory registration by using `uvm_component_utils
- Create handles for Host_sequencer class and Host_driver class.
- Create a memory for the Host_active_agent by using new() constructor.
- In the build phase create a memory for the above handles.
- In the connect phase, connection between Host_sequencer class to Host driver class through seq_item_port and seq_item_export respectively.

5.3.1.1.1 HOST SEQUENCER CLASS

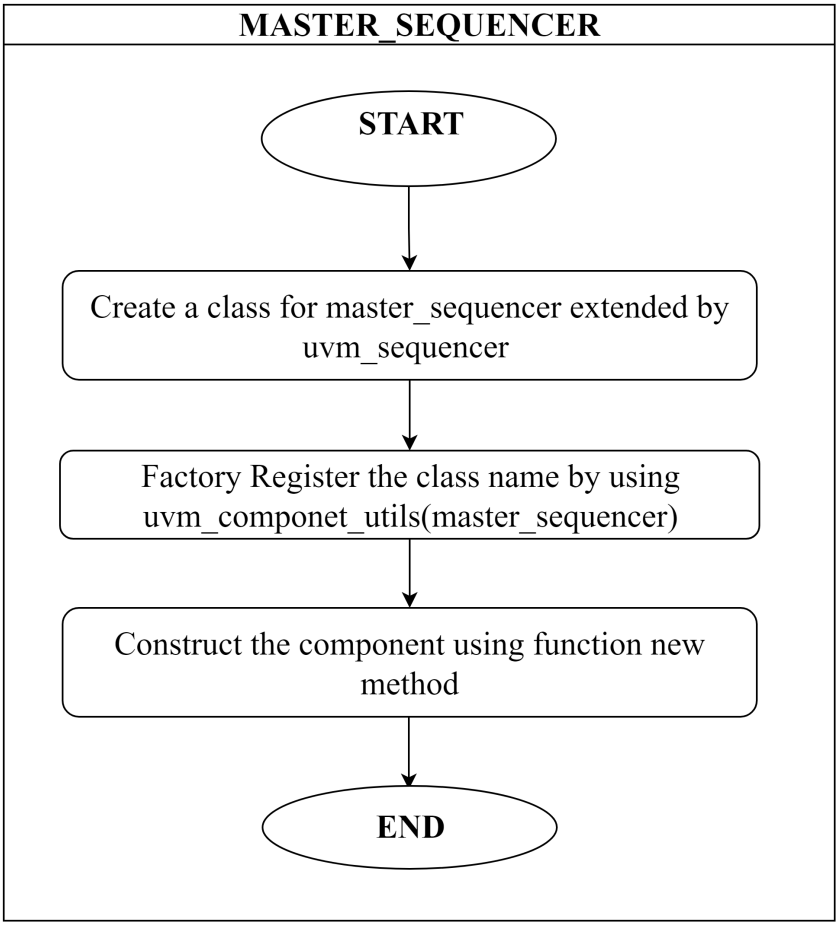


Fig:5.3.1.1.1 HOST SEQUENCER CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create a memory for the Host_active_agent by using new() constructor.

5.3.1.1.2 HOST DRIVER CLASS

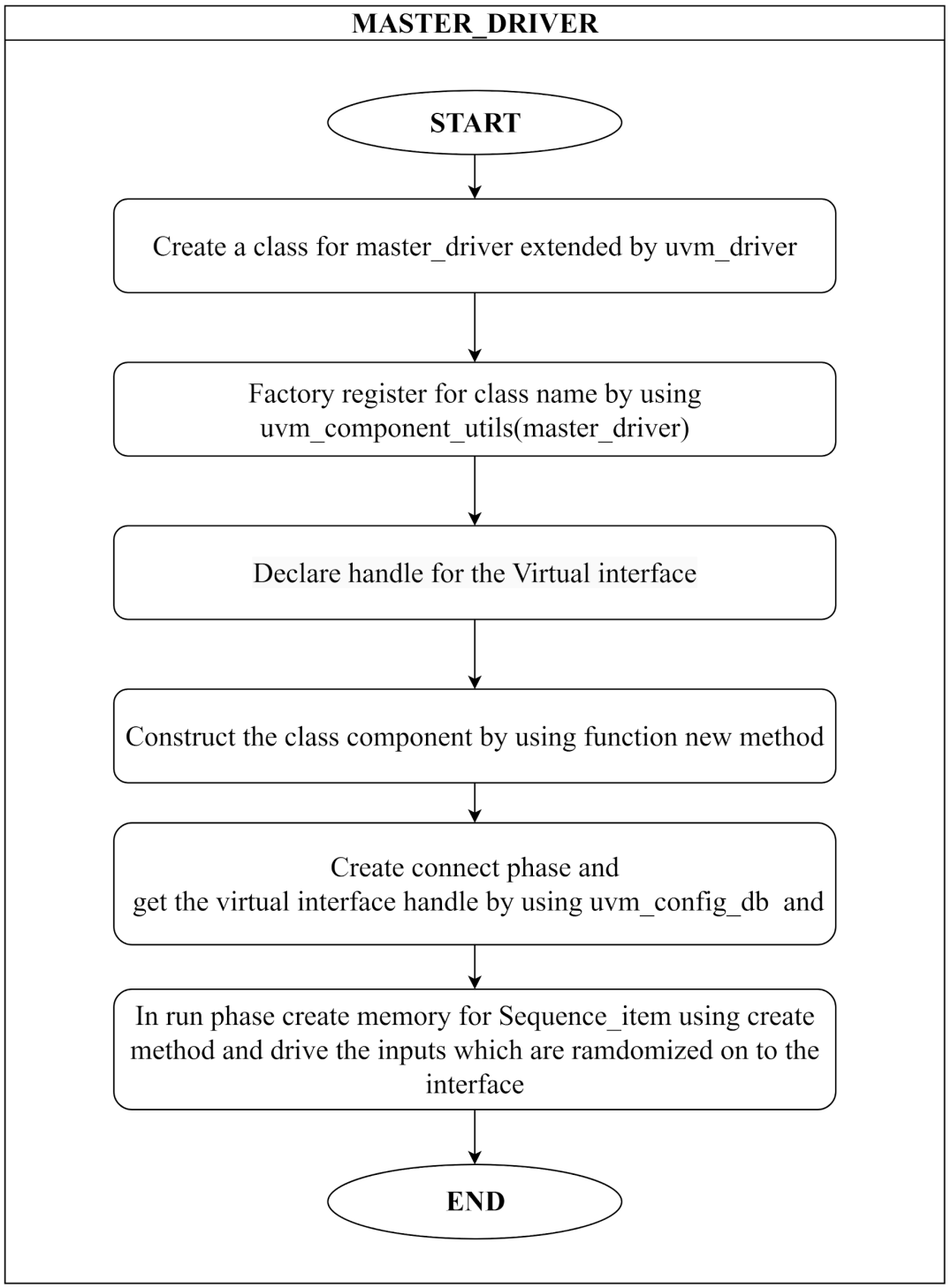


Fig:5.3.1.1.2 HOST DRIVER CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create the handles for virtual interface and sequence_item.
- Create a memory for the Host_driver by using new() constructor.

- In the build phase create a memory for the above handles. And by using get method access the virtual interface ports
- In the Run phase, assigning sequence_item ports to the virtual interface handle through the driver clocking blocks.

5.3.1.2 MEMORY ACTIVE AGENT CLASS

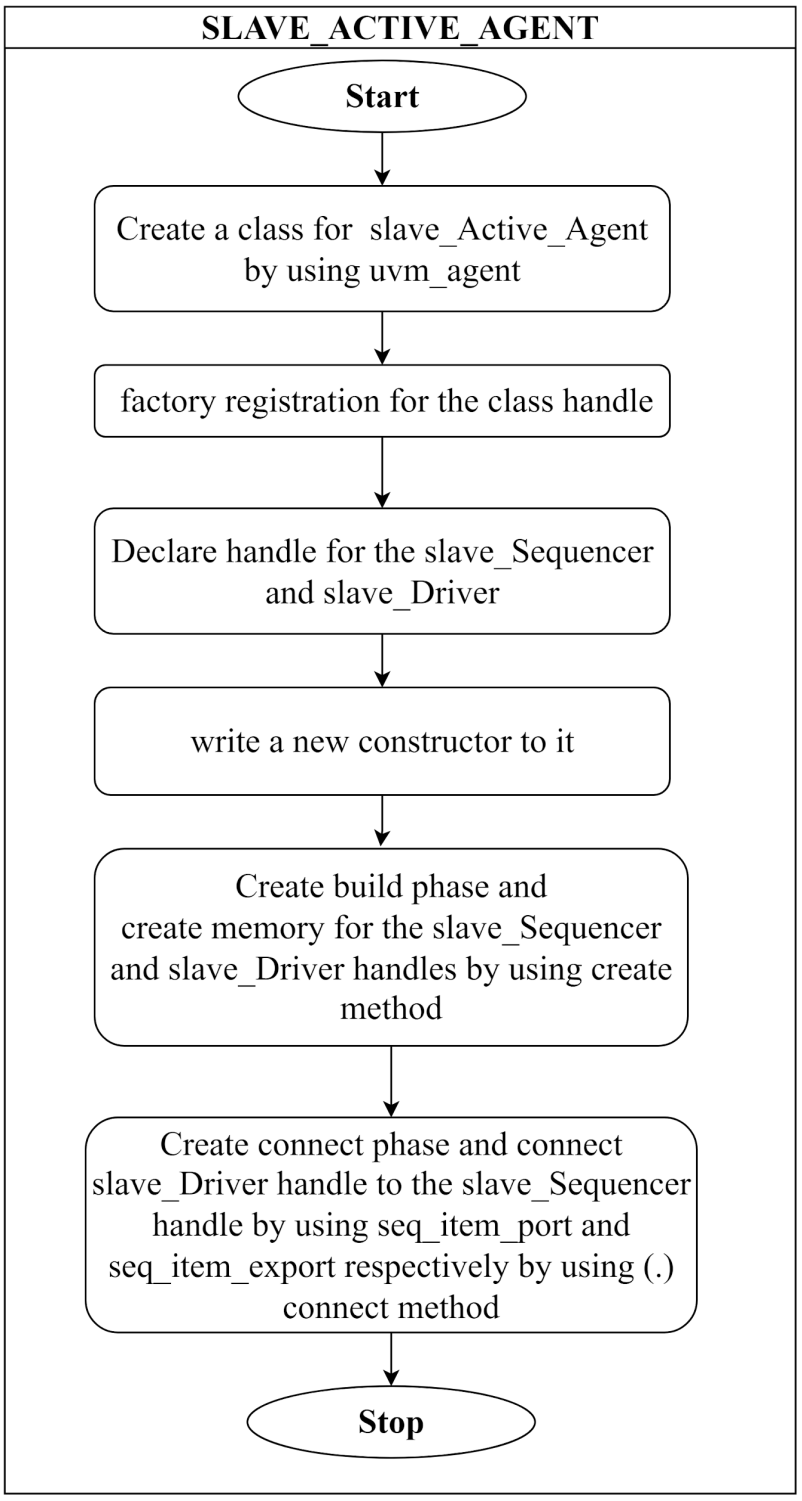


Fig:5.3.1.2 MEMORY ACTIVE AGENT CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create handles for memory_sequencer class and memory_driver class.
- Create a memory for the memory_active_agent by using new() constructor.
- In the build phase create a memory for the above handles.
- In the connect phase, connection between memory_sequencer class to memory driver class through seq_item_port and seq_item_export respectively.

5.3.1.2.1 MEMORY SEQUENCER CLASS

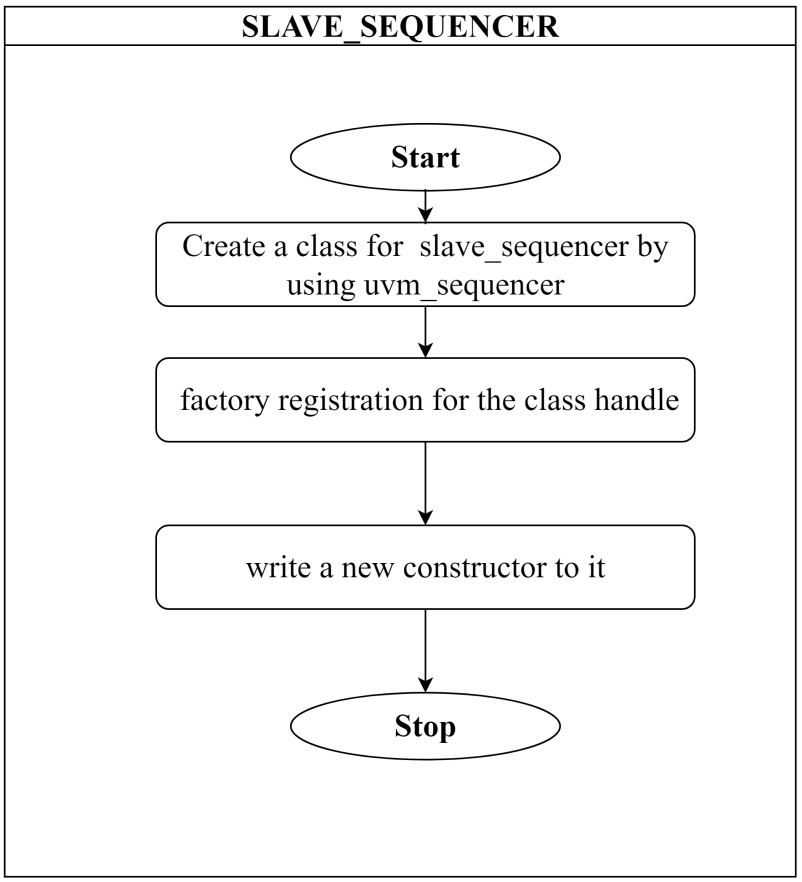


Fig:5.3.1.2.1 MEMORY SEQUENCER CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create a memory for the memory_sequencer class by using new() constructor.

5.3.1.2.2 MEMORY DRIVER CLASS

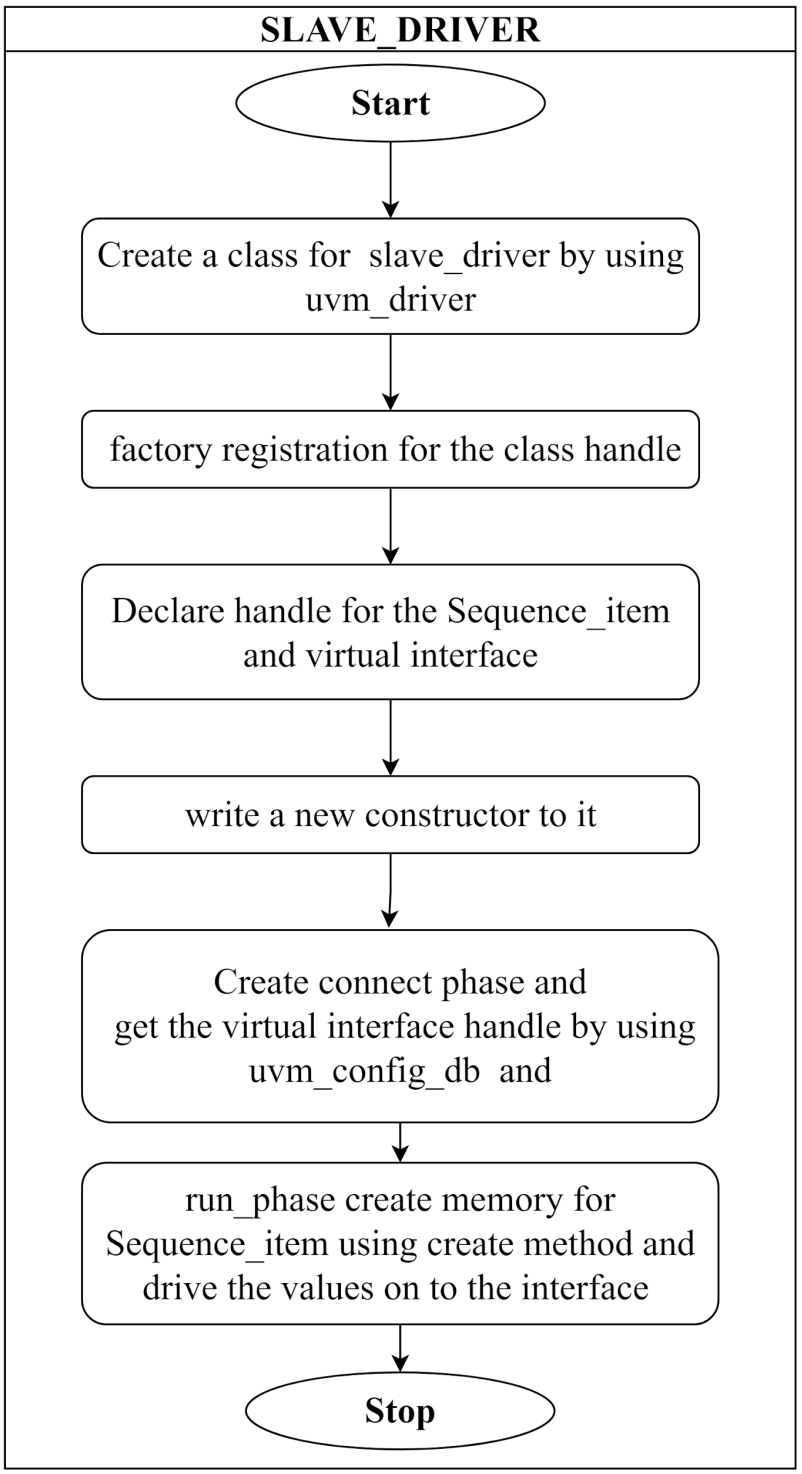


Fig:5.3.1.2.2 MEMORY DRIVER CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create the handles for virtual interface and sequence_item.
- Create a memory for the memory_driver by using new() constructor.

- In the build phase create a memory for the above handles. And by using get method access the virtual interface ports
- In the Run phase, assigning sequence_item ports to the virtual interface handle through the driver clocking blocks.

5.3.1.3 PASSIVE AGENT CLASS

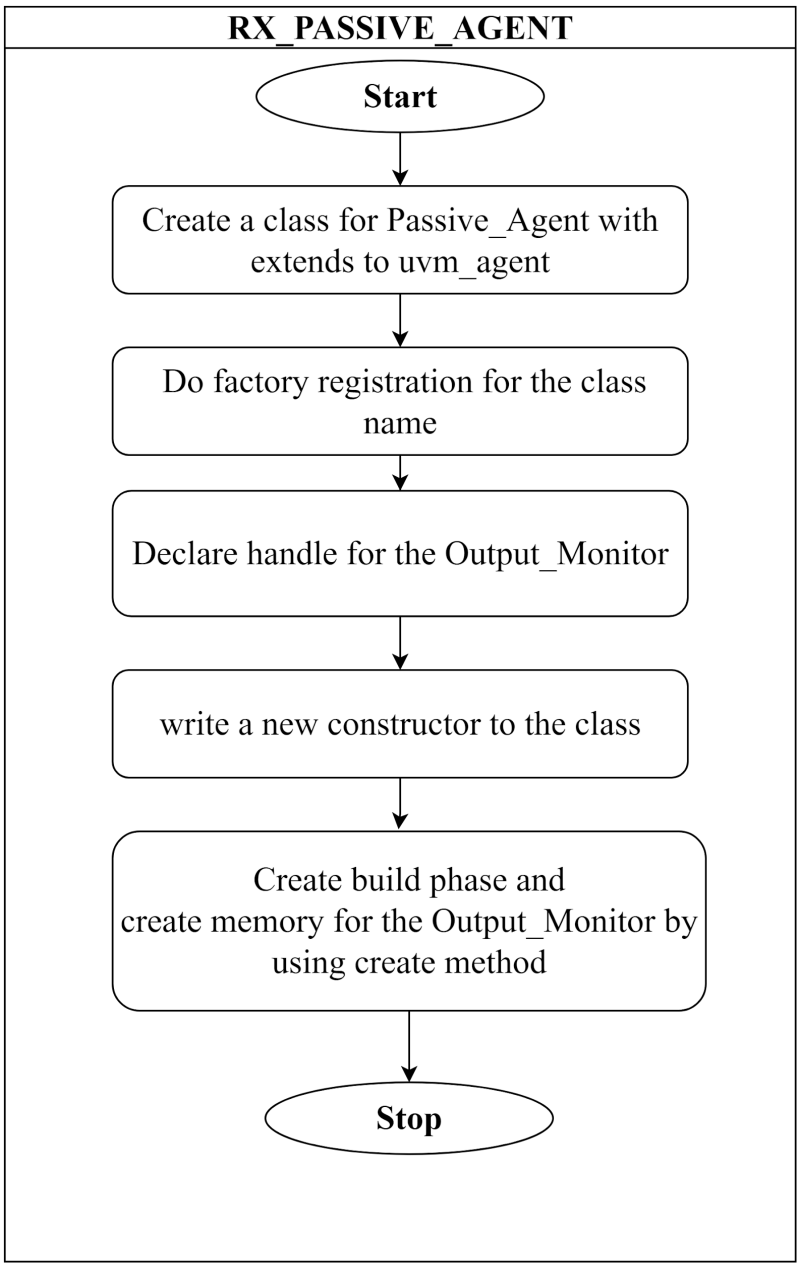


Fig:5.3.1.3 PASSIVE AGENT CLASS

DESCRIPTION:

- Extend the class from the uvm_agent class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create handles for Output_monitor class.

- Create a memory for the passive_agent component by using the function new() constructor.
- In the build phase creating a memory for the above handles.
- In the connect phase, using the output monitor handle and the output_monitor_port which is declared in the output monitor class, connect the passive agent export.

5.3.1.3.1 RX OUTPUT MONITOR CLASS

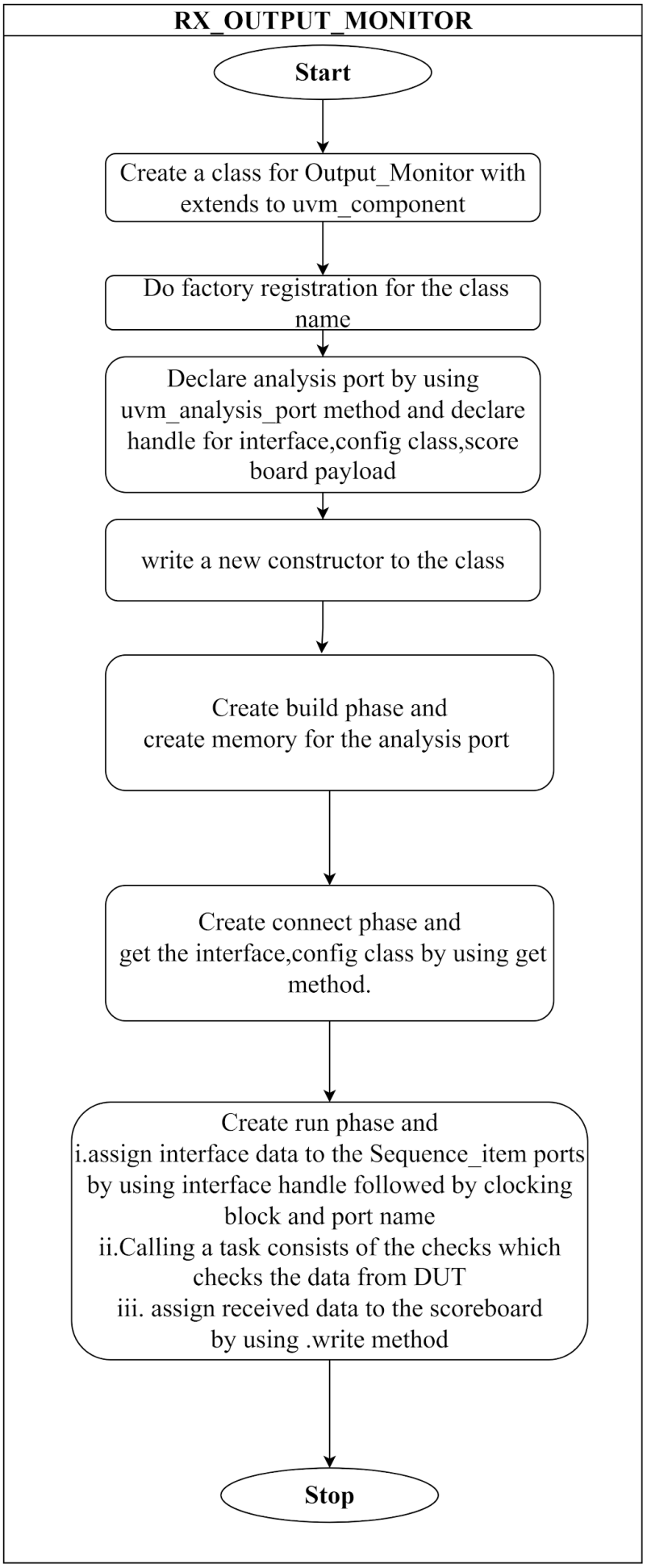
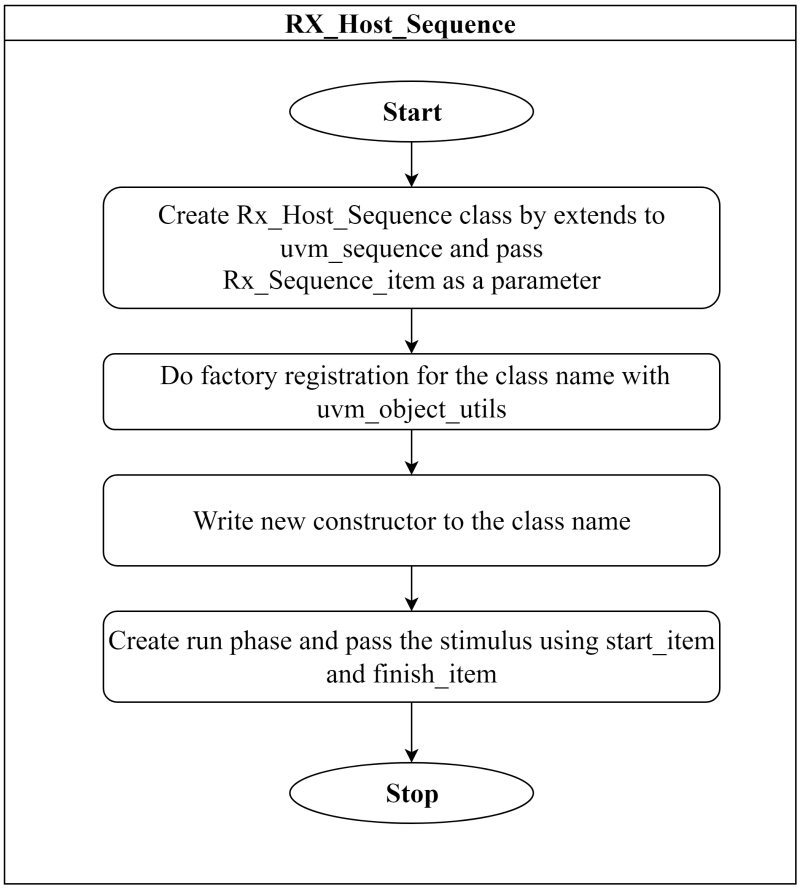

RX_OUTPUT_MONITOR
Start
Create a class for Output_Monitor with extends to uvm_component
Do factory registration for the class name
Declare analysis port by using uvm_analysis_port method and declare handle for interface,config class,score board payload
write a new constructor to the class
Create build phase and create memory for the analysis port
Create connect phase and get the interface,config class by using get method.
Create run phase and
i.assign interface data to the Sequence_item ports by using interface handle followed by clocking block and port name
ii.Calling a task consists of the checks which checks the data from DUT
iii. assign received data to the scoreboard by using .write method
Stop

Fig:5.3.1.3.1 RX OUTPUT MONITOR CLASS

DESCRIPTION:

- Creating a class for the output monitor class and extending the class from uvm_monitor which is the base class in uvm.
- Factory registration using uvm_macros for the class name using (`uvm_component_utils(output_monitor).
- Declaring the analysis port by using the uvm_analysis_port method.
- Creating the memory for class components by using the function new constructor method.
- In the build phase, create the memory for the analysis port using the new() constructor method and get the interface handle by using the get method.
- Establish the run phase using the interface handle, clocking block, and port name to assign interface data to the Sequence_item_portUsing the write method assigning the received data to the scoreboard.

5.3.1.3.2 SEQUENCE CLASS



DESCRIPTION:

- Declare the class Rx_Host_Sequence and extend it to base class uvm_sequence and pass Rx_Sequence_item as a parameter.
- Do the Factory Registration for the sequence class by using uvm_object_utils macro.

- Create memory for the sequence class by using new() constructor.
- In the task body pass the randomized stimulus of the Rx_Sequence_item handle using start_item() and finish_item() methods.

5.3.2 MAC ENVIRONMENT

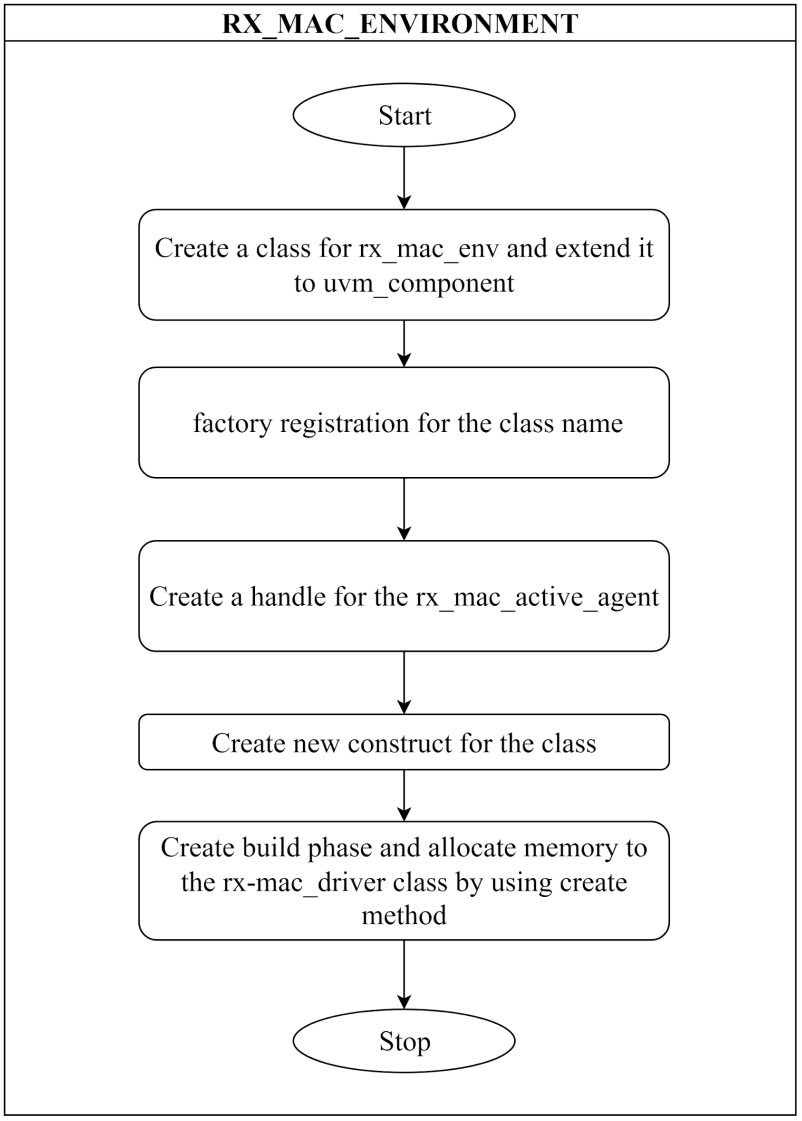


Fig:5.3.1 MAC ENVIRONMENT

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create the handle for MAC_active_agent.
- Create a memory for the MAC_environment by using new() constructor.

- In the build phase create a memory for the above handle.

5.3.2.1 ACTIVE AGENT CLASS

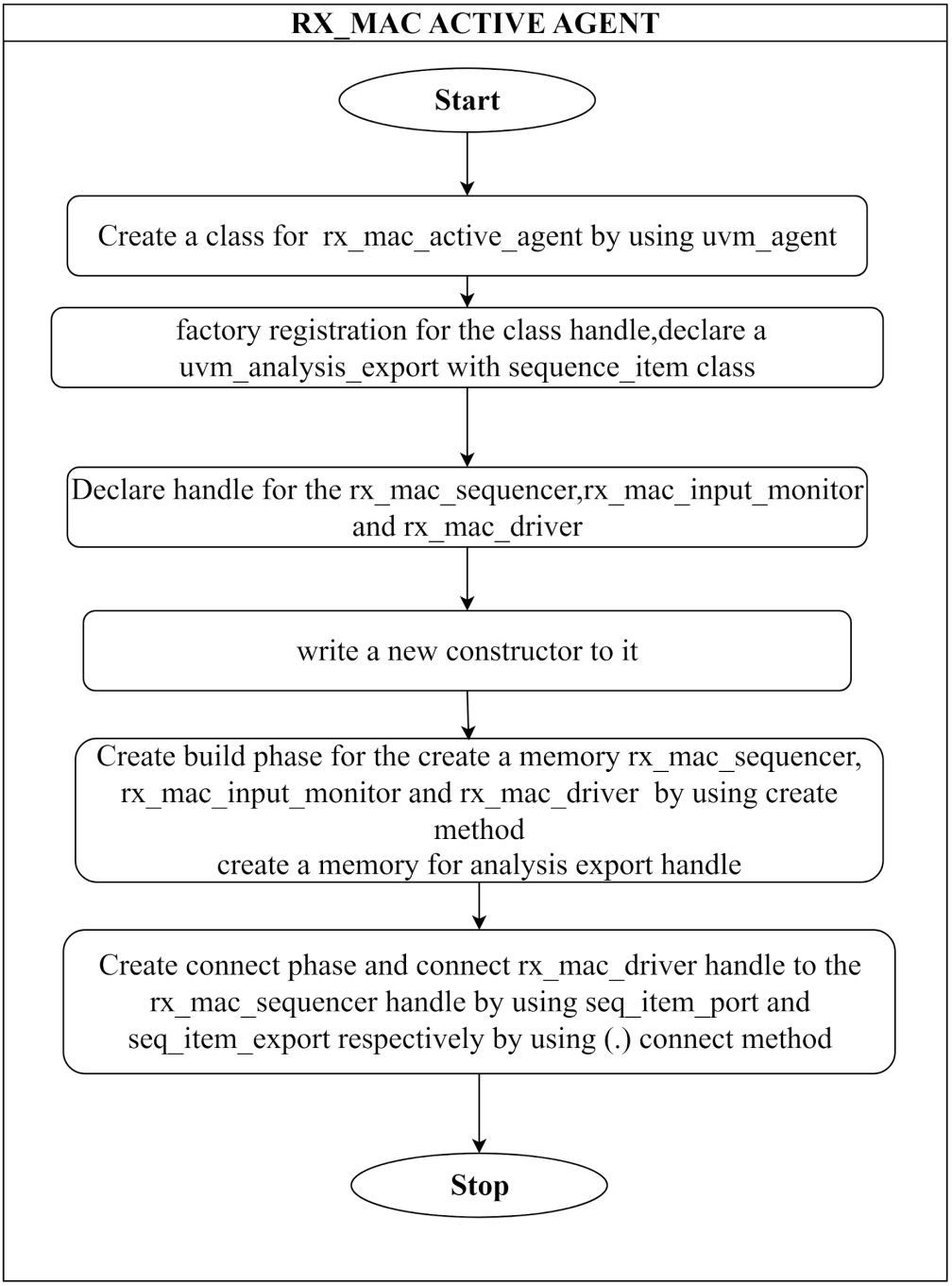


Fig:5.3.1.1 ACTIVE AGENT CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create handles for MAC_sequencer class and MAC_driver class.
- Create a memory for the MAC_active_agent by using new() constructor.
- In the build phase create a memory for the above handles.

- In the connect phase, connection between MAC_sequencer class to MAC driver class through seq_item_port and seq_item_export respectively.

5.3.2.1.1 SEQUENCER CLASS

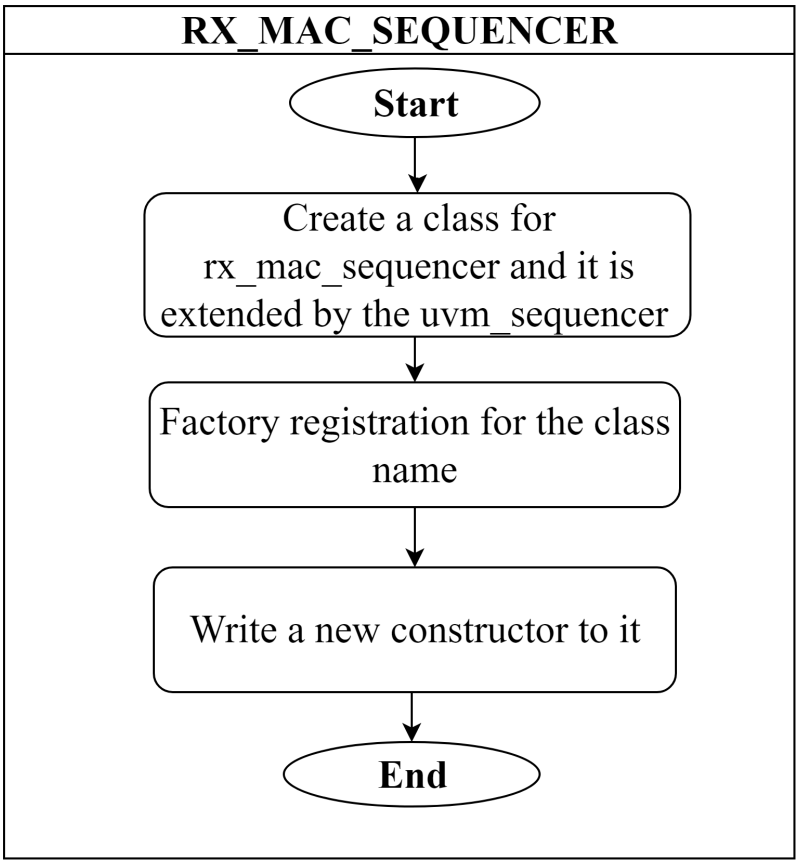


Fig:5.3.1.1.1 SEQUENCER CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros.
- Create a memory for the memory_sequencer class by using new() constructor.

5.3.2.1.2 DRIVER CLASS

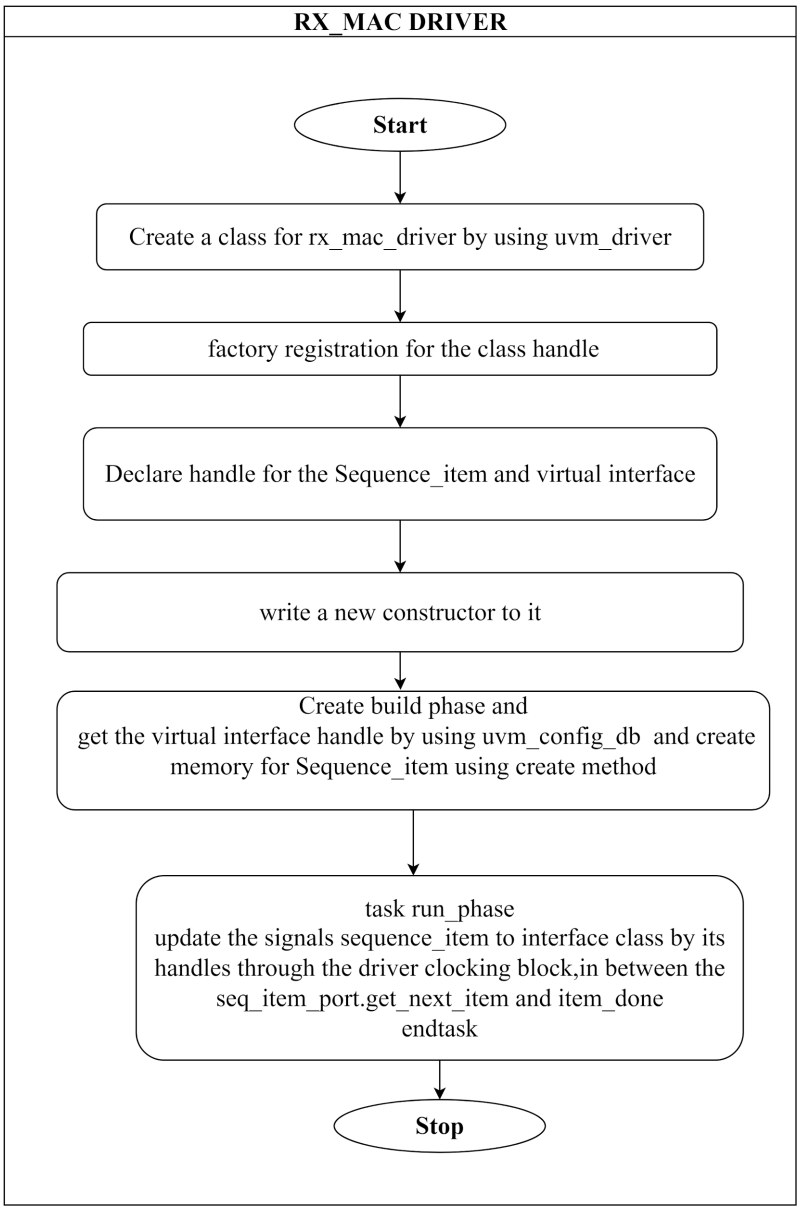


Fig:5.3.1.1.2 MAC_DRIVER CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create the handles for virtual interface and sequence_item.
- Create a memory for the MAC_driver by using new() constructor.
- In the build phase create a memory for the above handles. And by using get method access the virtual interface ports
- In the Run phase, assigning sequence_item ports to the virtual interface handle through the driver clocking blocks.

5.3.2.1.3 INPUT MONITOR CLASS

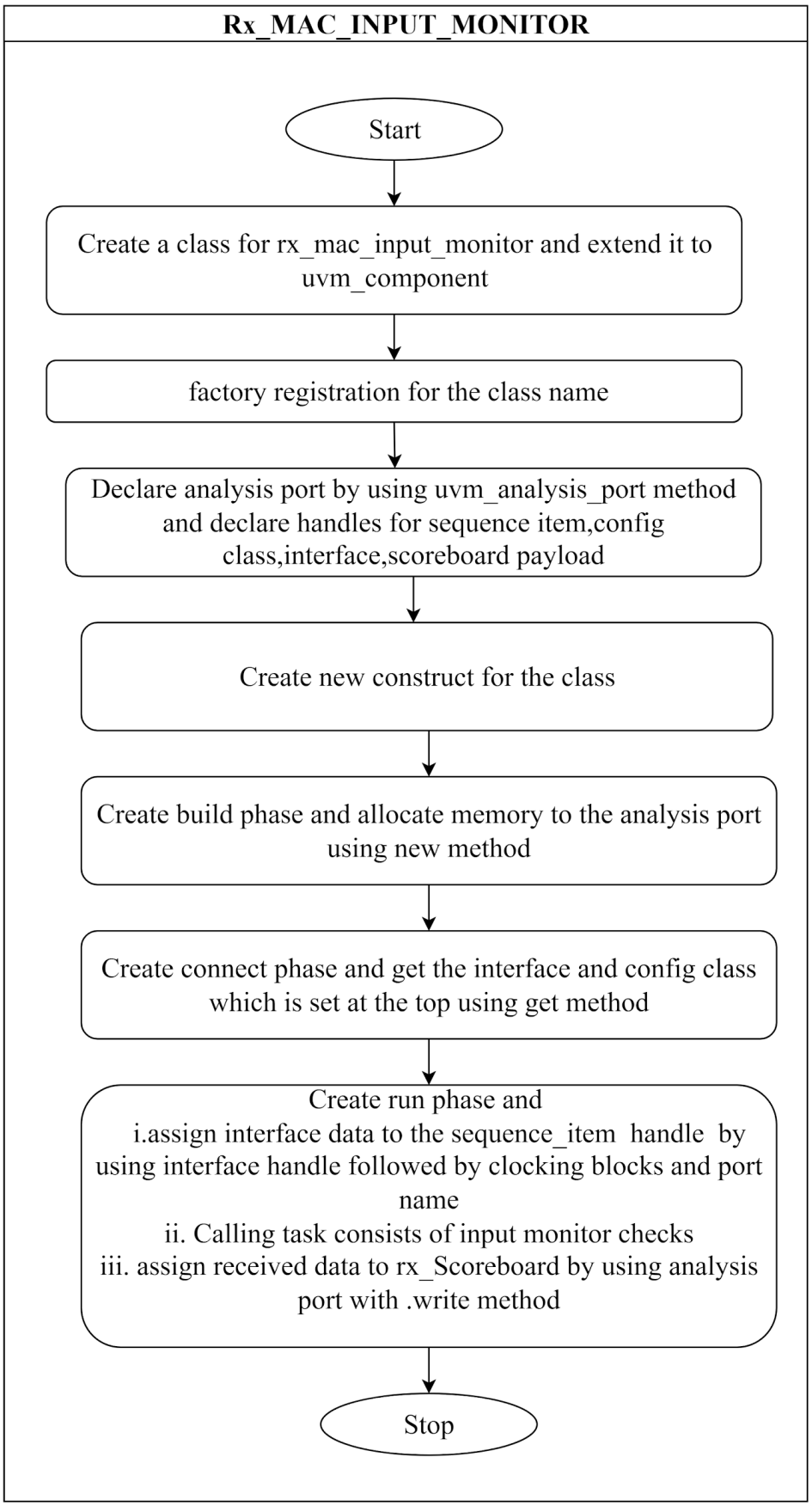


Fig:5.3.1.1.3 INPUT MONITOR CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Declare an analysis port by using uvm_analysis_port method.
- Create the handles for virtual interface and sequence_item.
- Create a memory for the MAC_input_monitor by using new() constructor.
- In the build phase, create a memory for analysis port and sequence_item class.And by using get method access the virtual interface ports.
- In the run phase, on the virtual interface updating ports to sequence_item ports through the monitor clocking blocks.

5.3.2.1.4 COVERAGE CLASS

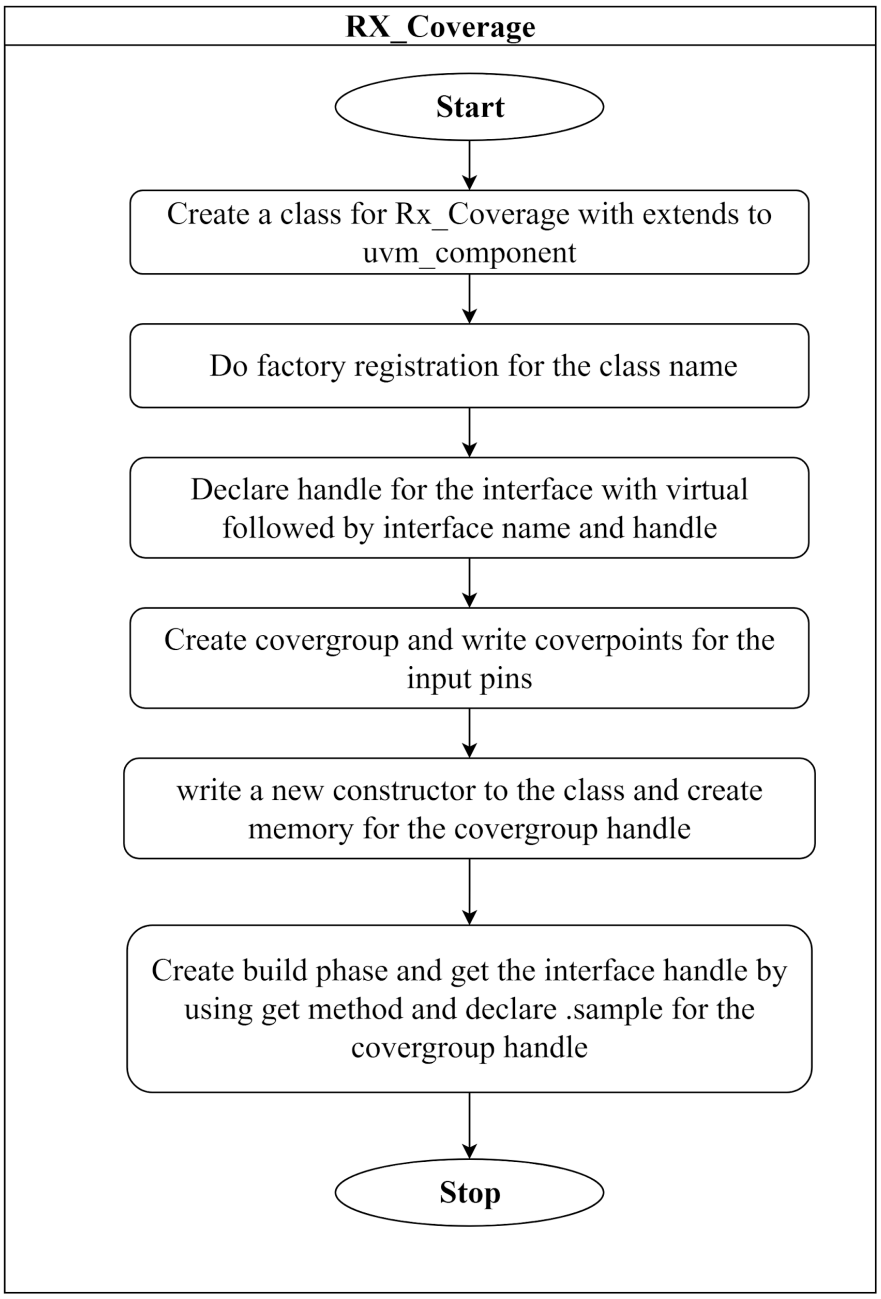
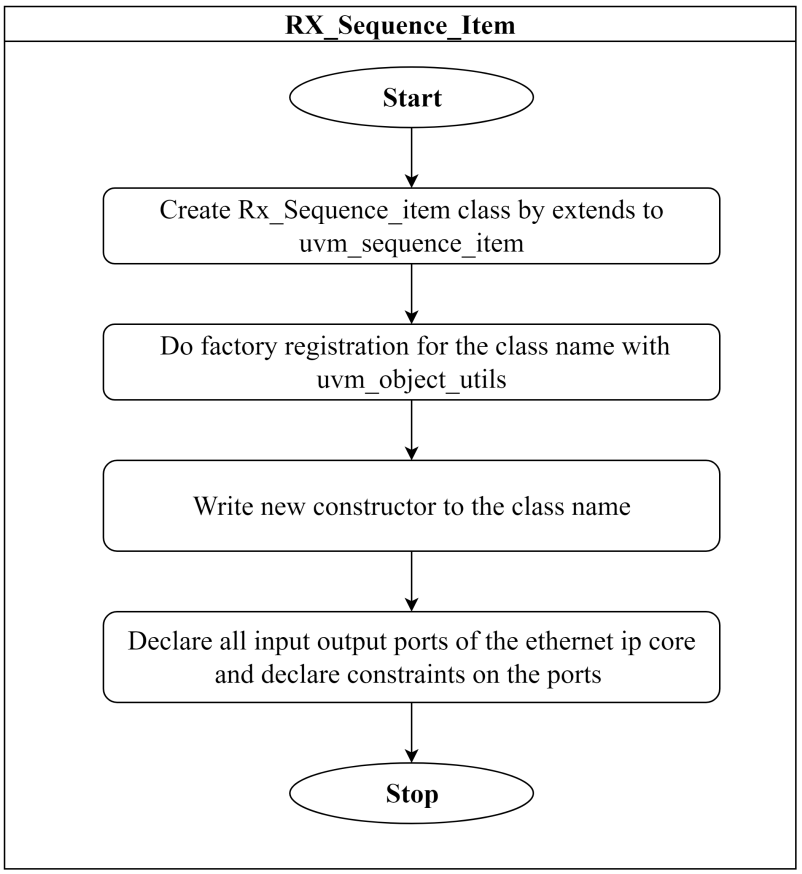


Fig:5.3.1.1.4 COVERAGE CLASS

DESCRIPTION:

- Extend the class from the uvm_component class which is the base class.
- Factory registration by using `uvm_component_utils macros
- Create the handles for the virtual interface.
- Creating the covergroup to write the coverpoints for the input pins.
- Constructing the class component by using the function new() method.
- In the build phase create a memory for the above handles. And by using the get method access the virtual interface ports.
- Create a covergroup and write coverpoints for input ports.
- Create a memory for the covergroup handle by using a new constructor.

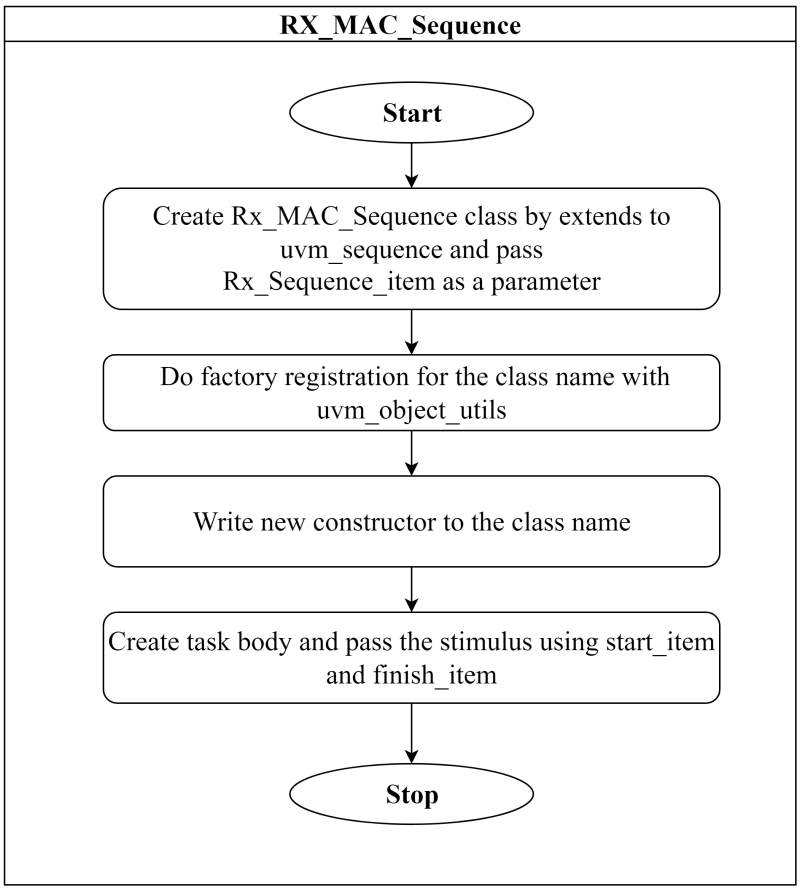
5.3.2.1.5 SEQUENCE_ITEM CLASS



DESCRIPTION:

- Create the class for Rx_Sequence_item and extend it to the base class uvm_sequence_item.
- Factory Registration is done for sequence_item class by using uvm_object_utils macro.
- Create memory for the Rx_Sequence_item class by using new() constructor.
- Declare the input and output ports of the ethernet ip core with bit type with their dimensions. And put rand and randc for the required inputs for randomisation.
- Declare constraints for the randomized input signals.

5.3.2.1.6 SEQUENCE CLASS



DESCRIPTION:

- Declare the class Rx_MAC_Sequence and extend it to base class uvm_sequence and pass Rx_Sequence_item as a parameter.
- Do the Factory Registration for the sequence class by using uvm_object_utils macro.
- Create memory for the sequence class by using new() constructor.
- In the task body pass the randomized stimulus by using Rx_Sequence_item handle within the start_item() and finish_item() methods.

5.4 INTERFACE

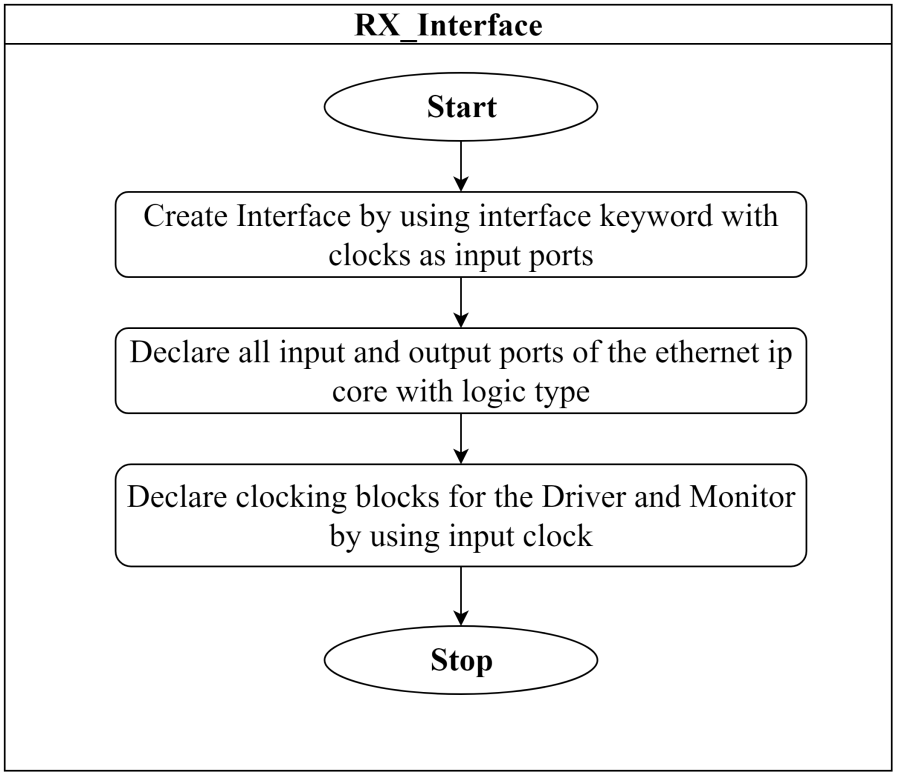
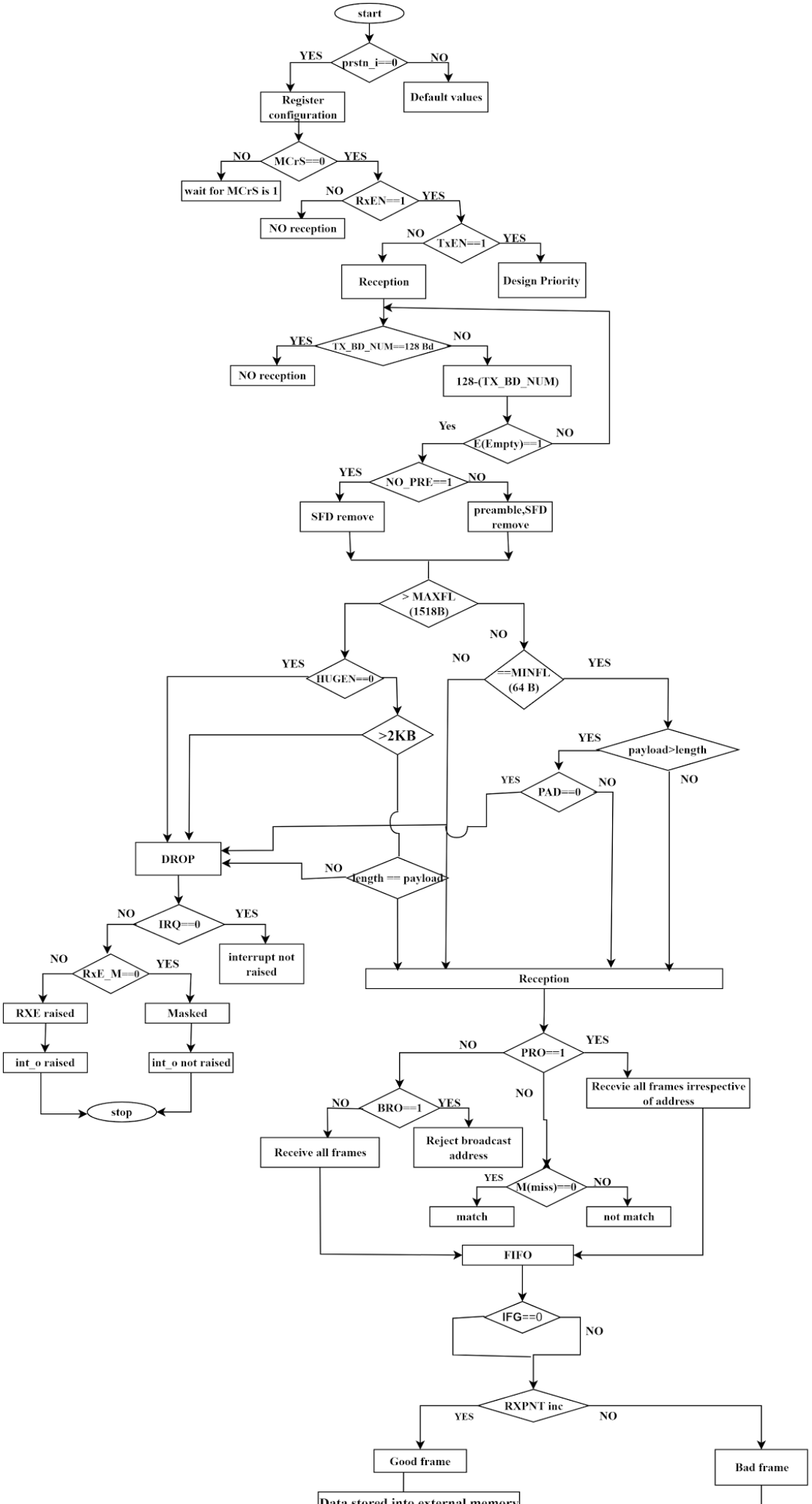


Fig: 5.4 INTERFACE

DESCRIPTION:

- Declare Rx_interface by using interface keyword and clock as input port in the port list.
- Declare all input and output ports of the ethernet ip core as logic type with its dimensions.
- Declare clocking blocks for the Driver and Monitor with separate clock declarations for the Rx_Host_Environment and Rx_MAC_Environment.

6. RX FLOW OF OPERATION


start
prstn_i==0
YES
NO
Default values
Register configuration
MCrS==0
NO
YES
wait for MCrS is 1
RxEN==1
NO
YES
NO reception
TxEN==1
NO
YES
Reception
Design Priority
TX_BD_NUM==128 Bd
YES
NO
NO reception
128-(TX_BD_NUM)
E(Empty)==1
Yes
NO
NO_PRE==1
YES
NO
SFD remove
preamble,SFD remove
> MAXFL (1518B)
HUGEN==0
YES
NO
>2KB
==MINFL (64 B)
YES
NO
payload>length
YES
NO
PAD==0
YES
NO
DROP
length == payload
NO
IRQ==0
NO
YES
interrupt not raised
RxE_M==0
NO
YES
RXE raised
Masked
int_o raised
int_o not raised
stop
Reception
PRO==1
NO
YES
Recevie all frames irrespective of address
BRO==1
NO
YES
Receive all frames
Reject broadcast address
M(miss)==0
YES
NO
match
not match
FIFO
IFG==0
NO
RXPNT inc
YES
NO
Good frame
Bad frame
Data stored into external memory