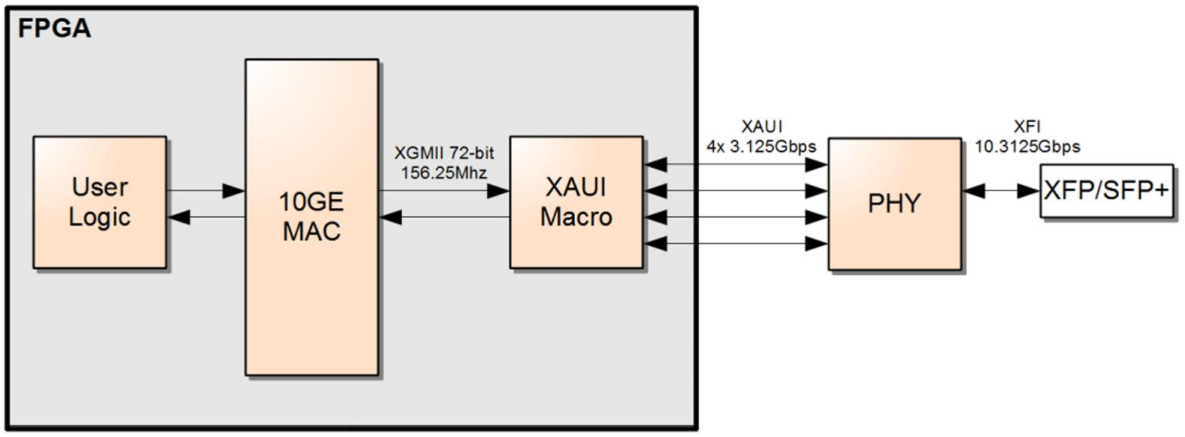
DESIGN VERIFICATION STRATEGY – XGEMAC – REV – A

Table of content

Contents

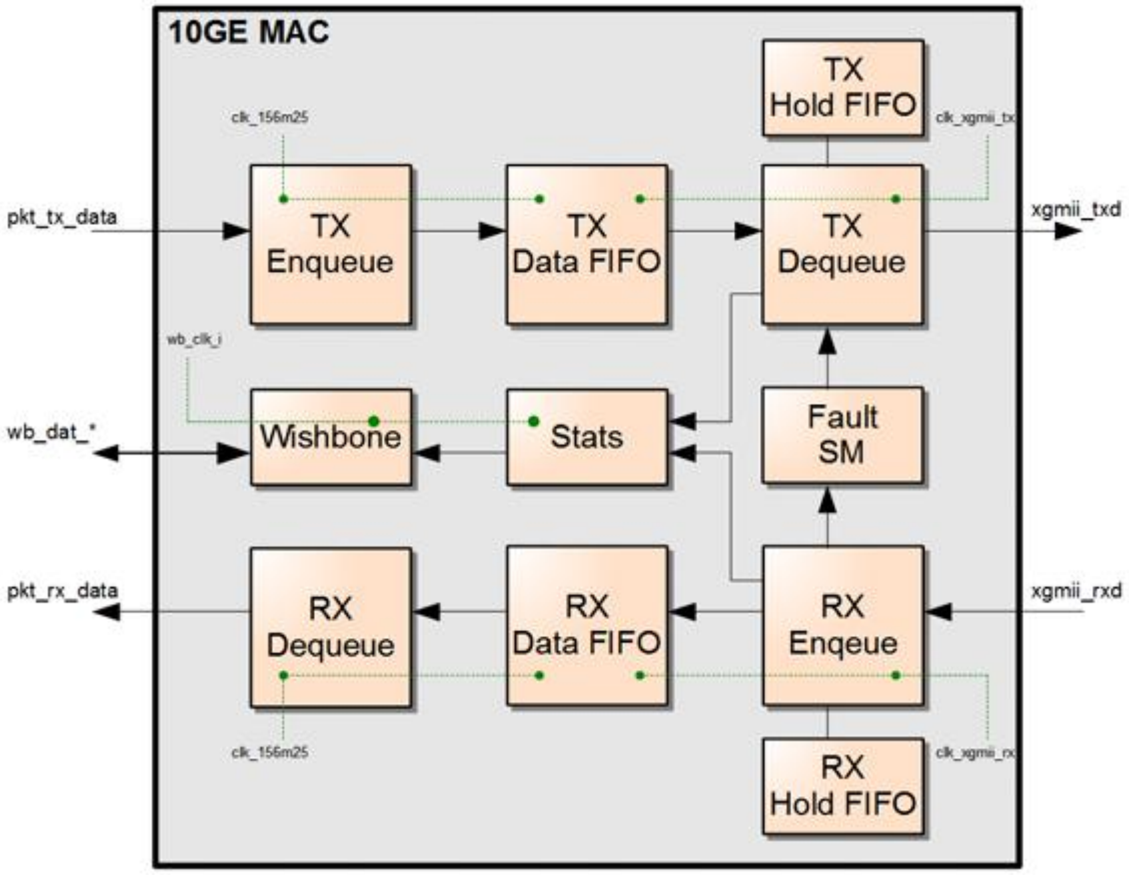
1. Dut Introduction

The 10GE MAC core is designed for easy integration with proprietary custom logic.

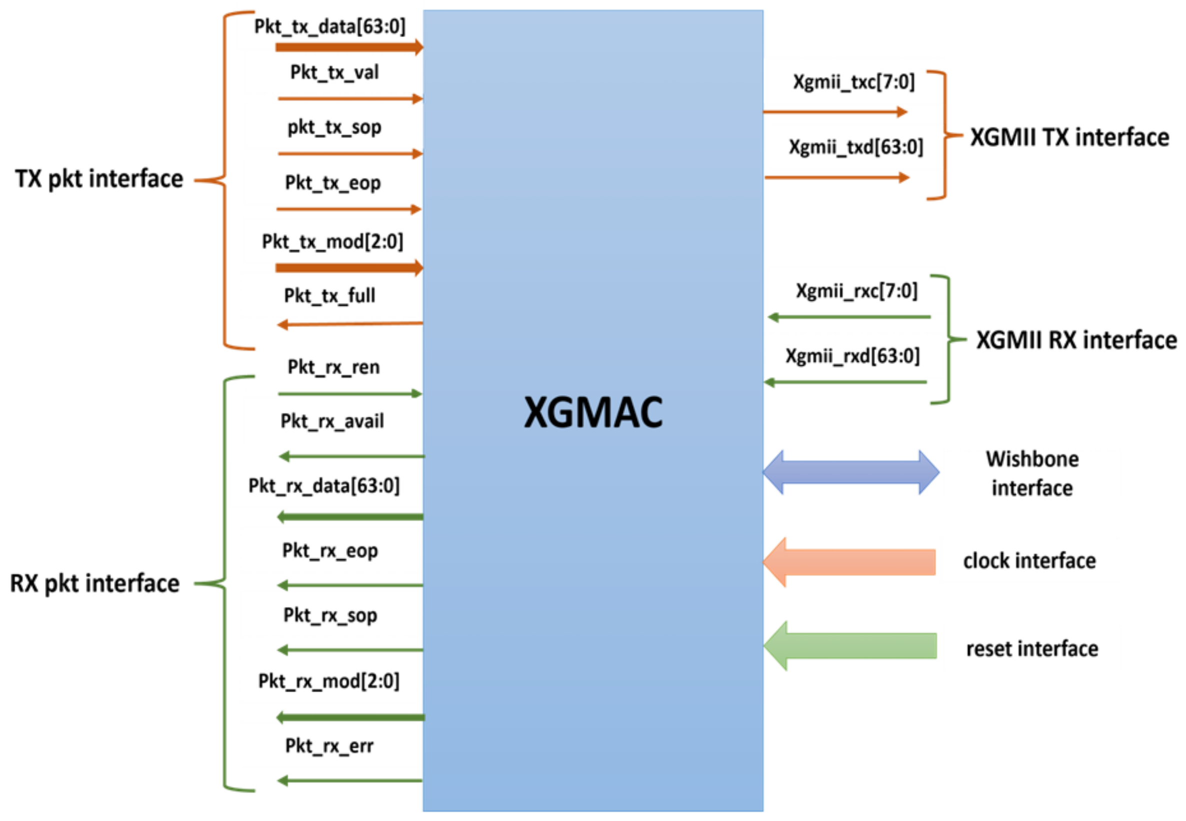


2. Internal block diagram

A block diagram of the core is presented in the figure below with clock domains indicated with green dotted lines.



3. Pin diagram



4. Interface details

a) Packet transmit interface

This interface accepts packets from the FPGA/ASIC core logic.

S. No	Port	Direction	Description
1	pkt_tx_data [63:0]	Input	Transmit Data: Little-endian format. First byte of packet must appear on pkt_rx_data [7:0].
2	pkt_tx_val	Input	Transmit Valid: This signal must be asserted for each valid data transfer to the 10GE MAC.
3	pkt_tx_sop	Input	Transmit Start of Packer: This signal must be asserted during the first word of a packet.
4	pkt_tx_eop	Input	Transmit End of Packet: This line must be asserted during the last word of a packet.
5	pkt_tx_mod [2:0]	Input	Transmit Packet Length Modulus: Valid during EOP. Indicates valid bytes during last word: Little Endian mode: 0: pkt_tx_data[63:0] is valid , 1: pkt_tx_data[7:0] is valid , 2: pkt_tx_data[15:0] is valid , 3: pkt_tx_data[23:0] is valid , 4: pkt_tx_data[31:0] is valid , 5: pkt_tx_data[39:0] is valid , 6: pkt_tx_data[47:0] is valid , 7: pkt_tx_data[55:0] is valid Big Endian mode: 0: pkt_rx_data[63:0] is valid , 1: pkt_rx_data[63:56] is valid 2: pkt_rx_data[63:48] is valid , 3: pkt_rx_data[63:40] is valid 4: pkt_rx_data[63:32] is valid , 5: pkt_rx_data[63:24] is valid 6: pkt_rx_data[63:16] is valid , 7: pkt_rx_data[63:8] is valid
6	pkt_tx_full	Output	Transmit Full: This signal indicates that transmit FIFO is nearing full and transfers should be suspended at the end of the current packet. Transfer of next packet can begin as soon as this signal is de-asserted.

b) XGMII interface

This interface used to loopback the xgmii transmiter side and receiver side.

S. No	Port	Direction	Description
1	xgmii_txrxc [7:0]	Inout	XGMII Transmit and Receive Control: Each bit corresponds to a byte on the 64-bit interface. When high, indicates that the byte is a control character. When low, indicates that the byte carries data.
2	xgmii_txrxd [63:0]	Inout	XGMII Transmit and Receive Data: When interfacing with 32-bit devices, xgmii_txrxd [31:0] should be mapped to the rising edge of the clock and xgmii_txrxd [63:32] should be mapped to the falling edge.

c) packet receive Interface

This interface is used to transfer received packets to the FPGA/ASIC core logic.

S. No	Port	Direction	Description
1	pkt_rx_ren	Input	Receive Read Enable: This signal should only be asserted when a packet is available in the receive FIFO. When asserted, the 10GE MAC core will begin packet transfer on next cycle. Signal should remain asserted until EOP becomes valid.
2	pkt_rx_avail	Output	Receive Available: Indicates that a packet is available for reading in receive FIFO.
3	pkt_rx_data [63:0]	Output	Receive Data: Little-endian format. First byte of packet will appear on pkt_rx_data [7:0].
4	pkt_rx_eop	Output	Receive End of Packet: Asserted when the last word of a packet is read from receive FIFO.
5	pkt_rx_val	Output	Receive Valid: Indicates that valid data is present on the bus. This signal is typically asserted one cycle after pkt_rx_ren was asserted unless FIFO underflow occurs
6	pkt_rx_sop	Output	Receive Start of Packet: Indicates that the first word of a frame is present on the bus.
7	pkt_rx_mod [2:0]	Output	Receive Packet Length Modulus: Valid during EOP. Indicates valid bytes during last word: Little Endian mode: 0: pkt_rx_data[63:0] is valid 1: pkt_rx_data[7:0] is valid 2: pkt_rx_data[15:0] is valid 3: pkt_rx_data[23:0] is valid 4: pkt_rx_data[31:0] is valid 5: pkt_rx_data[39:0] is valid 6: pkt_rx_data[47:0] is valid 7: pkt_rx_data[55:0] is valid Big Endian mode: 0: pkt_rx_data[63:0] is valid 1: pkt_rx_data[63:56] is valid 2: pkt_rx_data[63:48] is valid 3: pkt_rx_data[63:40] is valid 4: pkt_rx_data[63:32] is valid 5: pkt_rx_data[63:24] is valid 6: pkt_rx_data[63:16] is valid 7: pkt_rx_data[63:8] is valid
8	pkt_rx_err	Output	Receive Error: When asserted during a transfer, indicates that current packet is bad and should be discarded by user's logic. This signal is most likely asserted as the result of a CRC error. A frame of invalid size will also cause this signal to be asserted.

d) clock

S. No	Name	Frequency	Description
1	Wb_clk_i	30 – 156 Mhz	Wishbone interface clock.
2	Clk_156m25	156.25 Mhz	Clock for transmit and receive packet interfaces towards user's core logic. "pkt_tx_*" and "pkt_rx_*" signal are timed to this clock.
3	Clk_xgmii	156.25 Mhz	Clock for XGMII transmit and receive interface.

e) Reset

The user must ensure that each reset is synchronously de-asserted with its corresponding clock. To ensure that transmit and receive FIFOs are initialized correctly, “reset_156m25_n”, “reset_xgmii_rx_n” and “reset_xgmii_rx_n” must be de-asserted within 2-cycles of each other.

S. No	Name	Description
1	Wb_rst_i	Wishbone interface reset. Active high. Must be de-asserted synchronous to wb_clk_i.
2	reset_156m25_n	Core packet interfaces clock domain reset. Active low. Must be de-asserted synchronous to clk_156m25.
3	Reset_xgmii_txrx_n	XGMII transmit clock domain reset. Active low. Must be de-asserted synchronous to clk_xgmii_tx.

e) Wishbone Interface

This interface is used to configure the Registers in the Design.

Port	**Direction**	**Description**
wb_adr_i [7:0]	Input	Address input. All accesses to core must be 32-bit. Bits 0 and 1 are not used.
wb_cyc_i	Input	Wishbone cycle.
wb_dat_i [31:0]	Input	Wishbone data input.
wb_stb_i	Input	Wishbone strobe.
wb_we_i	Input	Wishbone write enanle.
wb_ack_o	Output	Wishbone acknowledge.
wb_dat_o [31:0]	Output	Wishbone data output.
wb_int_o	Output	Wishbone interrupt signal. Active high.

5. Feature plan

S.No.	Feature	Description
1.	Packetization	Packetizes the data for the XAUI macro block, where the data is sent from the user logic.
2.	Depacketization	Depacketizes the data for the user logic, where the data is received from the XAUI macro block.
3.	Reset	If reset is applied during a transaction, the design transitions to the idle state.

4.	Padding	When sending a minimum number of bytes to the design, it adds extra bytes (typically zeros) as padding.
5.	Modulus	The pkt_tx_mod and pkt_rx_mod signal is used to indicates valid bytes during last word.
6.	Rx read enable	The design was transmit a data when the pkt_rx_ren signal assert.

6. Test plan

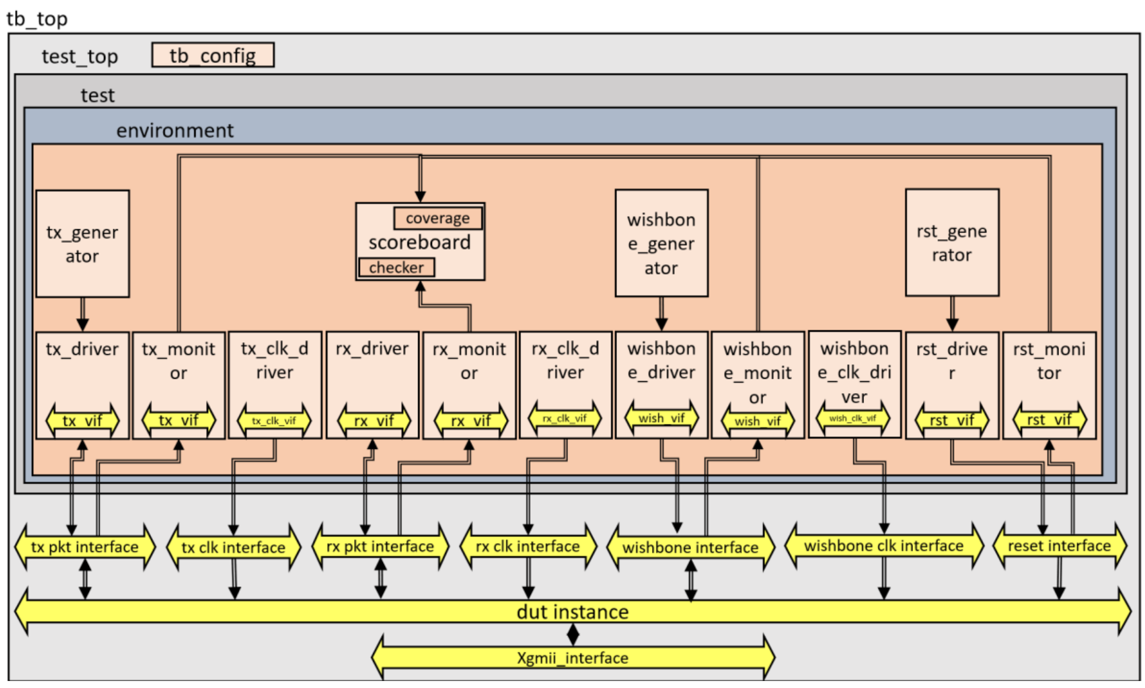
S.No.	Feature	Type	Test name	Description
1.	Data routing, Modulus	Sanity test	route_ sanity_ test	Connect the xgmii input and output signals, then verify that the input data matches the output data exactly. **Procedure:** **Step 1:** pkt_tx_sop=1 for 1 cycle. **Step 2:** pkt_tx_data = incremental data for n cycle. **Step 3:** pkt_tx_eop=1 for 1 cycle and pkt_tx_mod=0. **Step 4:** pkt_rx_ren=1 after pkt_rx_avail=1. **Step 5:** compare output data as per the input data.
2.	Data routing, Modulus	increment test	route_ increment_ test	Connect the xgmii input and output signals, then verify that the input data matches the output data exactly. **Procedure:** **Step 1:** pkt_tx_sop=1 for 1 cycle. **Step 2:** pkt_tx_data = random data for n cycle. **Step 3:** pkt_tx_eop=1 for 1 cycle and pkt_tx_mod=0. **Step 4:** pkt_rx_ren=1 after pkt_rx_avail=1. **Step 5:** compare output data as per the input data.

				Step 6: repeat step 1 to step 5 with incremental mod value.
3.	Reset	Reset test	reset_ test	Send data with SOP and EOP asserts, providing reset with a delay between transmissions. **Case 1 - Procedure:** **Step 1:** pkt_tx_sop=1 for 1 cycle. **Step 2:** pkt_tx_data = incremental data for n cycle. **Step 3:** reset_156m25_n=0 for 1 cycle. **Case 2 – Procedure** **Step 1:** wb_addr_i=address, wb_dat_i=data. **Step 2:** wb_rst_i=1 for 1 cycle.
4.	Padding	Direct test	padding_ direct _test	When sending a minimum number of bytes to the design, check it adds extra bytes. **Procedure:** **Step 1:** pkt_tx_sop=1 for 1 cycle. **Step 2:** pkt_tx_data = incremental data for n cycle (n<46). **Step 3:** pkt_tx_eop=1 for 1 cycle and pkt_tx_mod=0. **Step 4:** pkt_rx_ren=1 after pkt_rx_avail=1. **Step 5:** check output data is 46 bytes.
Error scenarios				
5.	Data without SOP	Direct test	without_ sop_ direct_ test	Send data without SOP asserts and with EOP asserts. **Procedure:** **Step 1:** pkt_tx_data = incremental data for n cycle. **Step 3:** pkt_tx_eop=1 for 1 cycle and pkt_tx_mod=0. **Step 3:** pkt_rx_ren=1 after pkt_rx_avail=1.

6.	Data without EOP	Direct test	without_ eop_ direct_ test	Send data without EOP asserts and with SOP asserts. **Procedure:** **Step 1:** pkt_tx_sop=1 for 1 cycle. **Step 2:** pkt_tx_data = incremental data for n cycle. **Step 3:** pkt_tx_eop=0, pkt_tx_mod=0. **Step 4:** pkt_rx_ren=1 after pkt_rx_avail=1.
7.	Data without SOP & EOP	Direct test	without_ sop_ eop_ direct_ test	Send data without SOP and EOP asserts. **Procedure:** **Step 1:** pkt_tx_sop=0. **Step 2:** pkt_tx_data = incremental data for n cycle. **Step 3:** pkt_tx_eop=0, pkt_tx_mod=0. **Step 4:** pkt_rx_ren=1 after pkt_rx_avail=1.
8.	SOP & EOP at same clock cycle	Direct test	sop_ eop_ at_ same _cycle_ direct_ test	Send data with SOP and EOP at same clock cycle. **Procedure:** **Step 1:** pkt_tx_sop=1, pkt_tx_eop=1 for 1 cycle. **Step 2:** pkt_tx_data = random 64bit data. **Step 3:** pkt_rx_ren=1 after pkt_rx_avail=1.
9.	Tx disable	Direct test	tx_ disable _test	When set as 0, transmission of frames is disabled then send data and check. **Procedure:** **Step 1:** set tx enable = 0 in configuration register through wishbone. **Step 2:** pkt_tx_sop=1 for 1 cycle. **Step 3:** pkt_tx_data = incremental data until pkt_tx_full=1. **Step 4:** pkt_tx_data = incremental data for n cycle. **Step 5:** pkt_tx_eop=1 for 1 cycle and pkt_tx_mod=0.

				Step 6: pkt_rx_ren=1 after pkt_rx_avail=1.

7. Testbench architecture



9. Environment

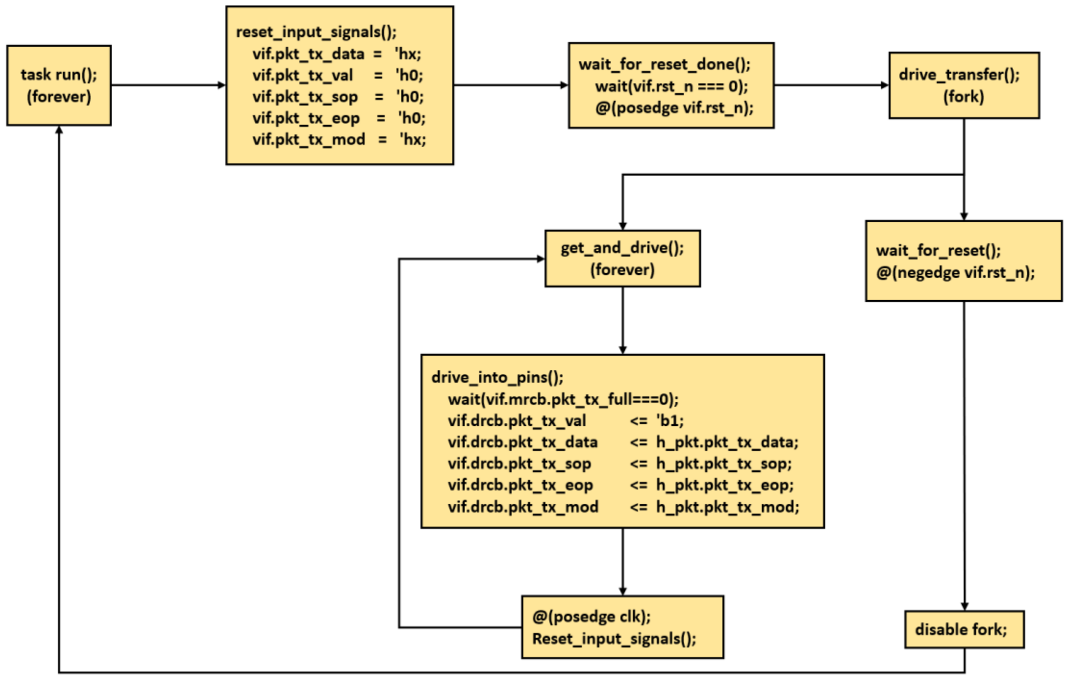
a) Tx generator

It has methods for,

- Generate direct stimulus and put in mailbox
- Generate Incremental stimulus and put in mailbox
- Generate random stimulus and put in mailbox
- Generate stimulus without SOP and put in mailbox
- Generate stimulus without EOP and put in mailbox
- Generate stimulus with SOP and EOP at same clock cycle and put in mailbox
- Generate stimulus without SOP and EOP and put in mailbox

b) Tx driver

The Tx driver runs forever, waiting for a packet; once it receives one, it drives the signals to the Tx interface and includes reset handling.



```
task run();
(forever)
```

```
reset_input_signals();
  vif.pkt_tx_data  =  'hx;
  vif.pkt_tx_val   =  'h0;
  vif.pkt_tx_sop   =  'h0;
  vif.pkt_tx_eop   =  'h0;
  vif.pkt_tx_mod   =  'hx;
```

```
wait_for_reset_done();
  wait(vif.rst_n === 0);
  @(posedge vif.rst_n);
```

```
drive_transfer();
(fork)
```

```
get_and_drive();
(forever)
```

```
wait_for_reset();
@(negedge vif.rst_n);
```

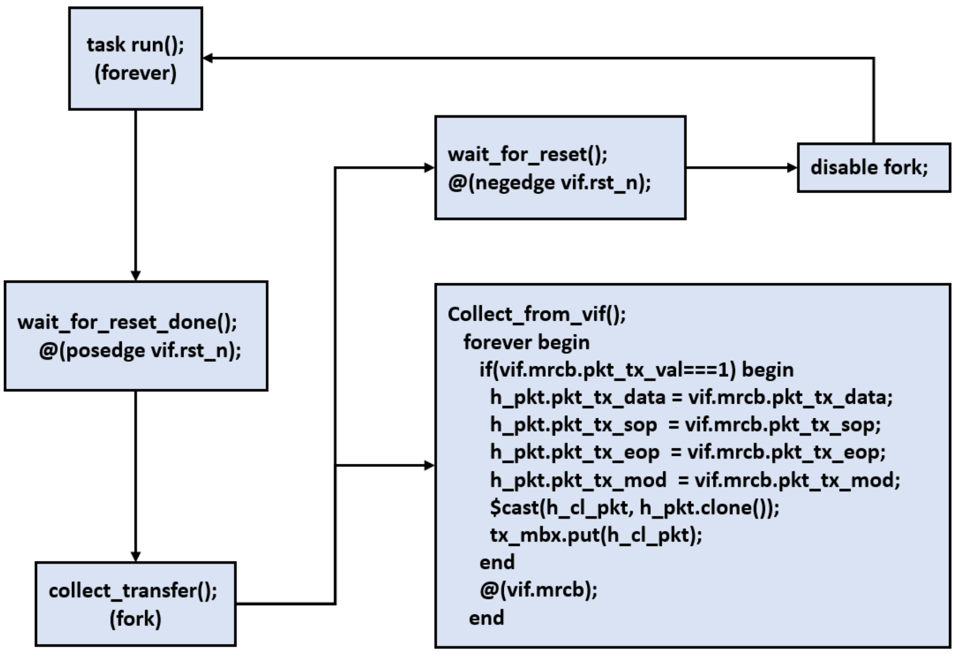
```
drive_into_pins();
  wait(vif.mrcb.pkt_tx_full===0);
  vif.drcb.pkt_tx_val      <=  'b1;
  vif.drcb.pkt_tx_data     <=  h_pkt.pkt_tx_data;
  vif.drcb.pkt_tx_sop      <=  h_pkt.pkt_tx_sop;
  vif.drcb.pkt_tx_eop      <=  h_pkt.pkt_tx_eop;
  vif.drcb.pkt_tx_mod      <=  h_pkt.pkt_tx_mod;
```

```
@(posedge clk);
Reset_input_signals();
```

```
disable fork;
```

c) Tx monitor

The Tx monitor runs forever, waiting for pkt_tx_val signal assertion; it then puts the packet into the mailbox and includes reset handling.



```
task run();
(forever)
```

```
wait_for_reset();
@(negedge vif.rst_n);
```

```
disable fork;
```

```
wait_for_reset_done();
  @(posedge vif.rst_n);
```

```
Collect_from_vif();
  forever begin
    if(vif.mrcb.pkt_tx_val===1) begin
      h_pkt.pkt_tx_data = vif.mrcb.pkt_tx_data;
      h_pkt.pkt_tx_sop  = vif.mrcb.pkt_tx_sop;
      h_pkt.pkt_tx_eop  = vif.mrcb.pkt_tx_eop;
      h_pkt.pkt_tx_mod  = vif.mrcb.pkt_tx_mod;
      $cast(h_cl_pkt, h_pkt.clone());
      tx_mbx.put(h_cl_pkt);
    end
    @(vif.mrcb);
  end
```

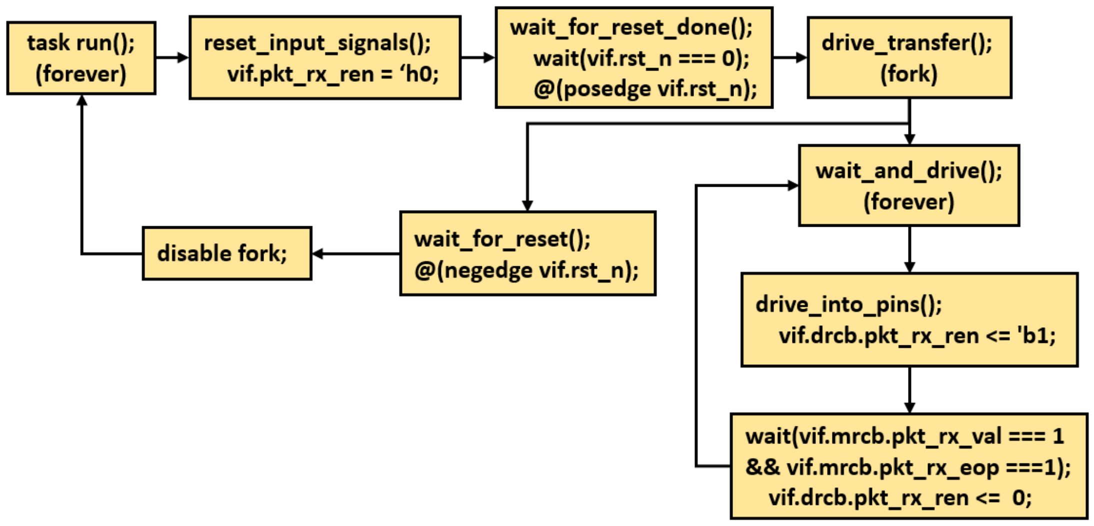
```
collect_transfer();
(fork)
```

d) Txrx clock driver

The Tx clock driver runs forever, toggling the clock value at 156.25 MHz frequency.

e) Rx driver

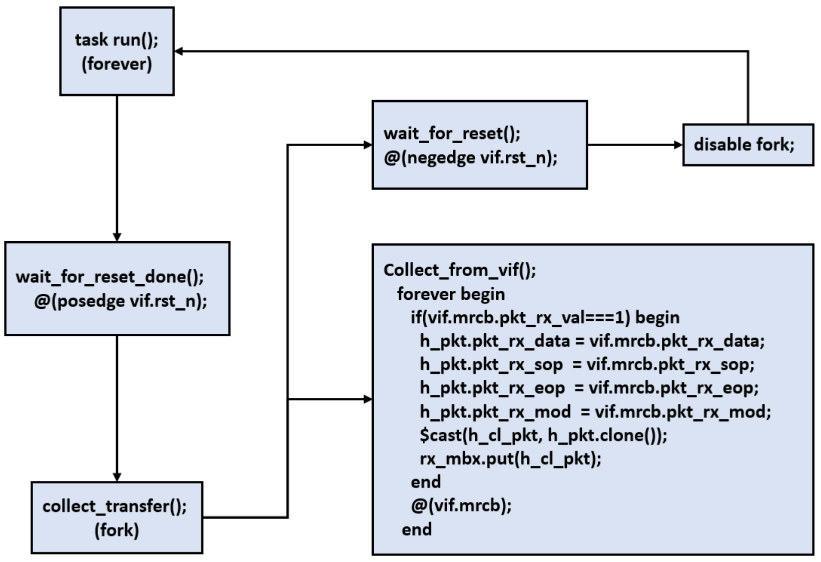
The Rx driver is a slave driver that runs forever, waiting for pkt_rx_avail signal assertion. Once the pkt_rx_avail signal is high, it drives pkt_rx_ren signal high.



task run(); (forever)
reset_input_signals(); vif.pkt_rx_ren = 'h0;
wait_for_reset_done(); wait(vif.rst_n === 0); @(posedge vif.rst_n);
drive_transfer(); (fork)
wait_and_drive(); (forever)
drive_into_pins(); vif.drcb.pkt_rx_ren <= 'b1;
wait(vif.mrcb.pkt_rx_val === 1 && vif.mrcb.pkt_rx_eop ===1); vif.drcb.pkt_rx_ren <= 0;
wait_for_reset(); @(negedge vif.rst_n);
disable fork;

f) Rx monitor

The Tx monitor runs forever, waiting for pkt_rx_val signal assertion; it then puts the packet into the mailbox and includes reset handling.



task run(); (forever)
wait_for_reset_done(); @(posedge vif.rst_n);
collect_transfer(); (fork)
wait_for_reset(); @(negedge vif.rst_n);
disable fork;

```
Collect_from_vif();
  forever begin
    if(vif.mrcb.pkt_rx_val===1) begin
      h_pkt.pkt_rx_data = vif.mrcb.pkt_rx_data;
      h_pkt.pkt_rx_sop  = vif.mrcb.pkt_rx_sop;
      h_pkt.pkt_rx_eop  = vif.mrcb.pkt_rx_eop;
      h_pkt.pkt_rx_mod  = vif.mrcb.pkt_rx_mod;
      $cast(h_cl_pkt, h_pkt.clone());
      rx_mbx.put(h_cl_pkt);
    end
    @(vif.mrcb);
  end
```

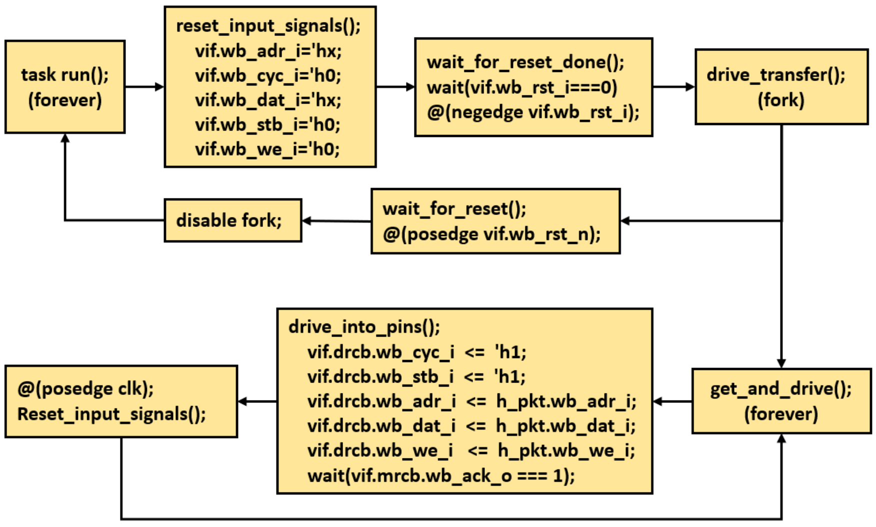
g) Wishbone generator

It has methods for,

- Generate stimulus for read tx enable and put in mailbox.
- Generate stimulus for disable tx enable and put in mailbox.
- Generate stimulus for transmit packet count and put in mailbox.
- Generate stimulus for transmit octets count and put in mailbox.
- Generate stimulus for receiving packet count and put in mailbox.
- Generate stimulus for receiving octets count and put in mailbox.

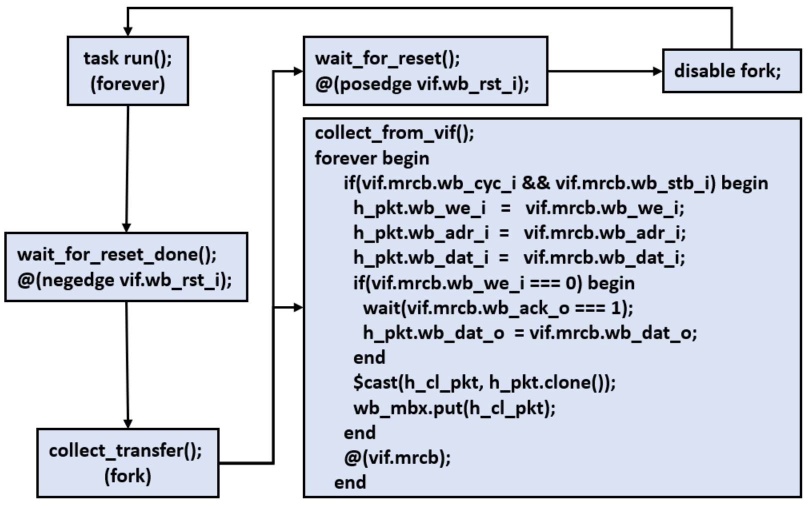
h) Wishbone driver

The wishbone driver runs forever, waiting for a packet; once it receives one, it drives the signals to the wishbone interface and includes reset handling.



i) Wishbone monitor

The wishbone monitor runs forever, waiting for wb_stb_i and wb_cyc_i signal assertion and also waiting for wb_ack_o signal assertion; it then puts the packet into the mailbox and includes reset handling.



```
collect_from_vif();
forever begin
   if(vif.mrcb.wb_cyc_i && vif.mrcb.wb_stb_i) begin
     h_pkt.wb_we_i  =  vif.mrcb.wb_we_i;
     h_pkt.wb_adr_i  =  vif.mrcb.wb_adr_i;
     h_pkt.wb_dat_i  =  vif.mrcb.wb_dat_i;
     if(vif.mrcb.wb_we_i === 0) begin
       wait(vif.mrcb.wb_ack_o === 1);
       h_pkt.wb_dat_o  = vif.mrcb.wb_dat_o;
     end
     $cast(h_cl_pkt, h_pkt.clone());
     wb_mbx.put(h_cl_pkt);
   end
   @(vif.mrcb);
 end
```

j) **Wishbone clock driver**
The Tx clock driver runs forever, toggling the clock value at 100 MHz frequency.

k) **Reset generator**
It has methods for generate reset packet which includes reset period and puts the packet into the mailbox.

l) **Reset driver**
The reset driver has two methods: one for power-on reset and the other a waiting for packet method. This second method runs forever, waiting for a packet; once it receives one, it drives the signals to the reset interface.

m) **Reset monitor**
The reset monitor checks the reset signal de-assert at every positive edge of the clock and puts a packet into the mailbox.