AXI FIFO Verification Plan

|  |  |  |
| --- | --- | --- |
| Date | Document Version | Drafted by |
| 28th February 2024 | Version 1.0 | Krishnakumar E |

**VERIFICATION DOCUMENT- AXI FIFO BFM**

# CONTENTS Page Number

**CHAPTER 1 Design Overview**

* 1. AXI-FIFO BFM Features 4
  2. Design Limitation 4
  3. Design diagram with interface signals 5

**CHAPTER 2 Test bench Architecture 6**

**CHAPTER 3 Test bench Implementation**

3.1 AXI BFM Write FIFO Agent 7

3.2 AXI BFM Read FIFO Agent 17

3.3 AXI4 VIP Slave Configuration 21

3.4 Scoreboard 22

3.5 Coverage 26

**CHAPTER 4 Test Plan and Test-cases** 29

**CHAPTER 5 User Guide** 33

**CHAPTER 6 Simulation result and analysis 40**

# CHAPTER 1 – DESIGN OVERVIEW

The AXI\_FIFO BFM design is a simple Packet Decoder with AXI4 master interface. The design consists of three blocks:

1. The FIFO interface consisting of read FIFO and write FIFO
2. The decoder
3. The AXI master interface

The CPU writes the packets into the write FIFO. The decoder reads the packets from the write FIFO and decodes the packets to extract address, control and data information. The decoded information is forwarded to the AXI master. The AXI master creates valid AXI write and read transactions from the decoded information.

The AXI master forward the read data/response and write response to the decoder. The decoder forms the Read data/response packet and write response packet and writes them into the Read FIFO. The CPU reads the response packets from the Read FIFO.

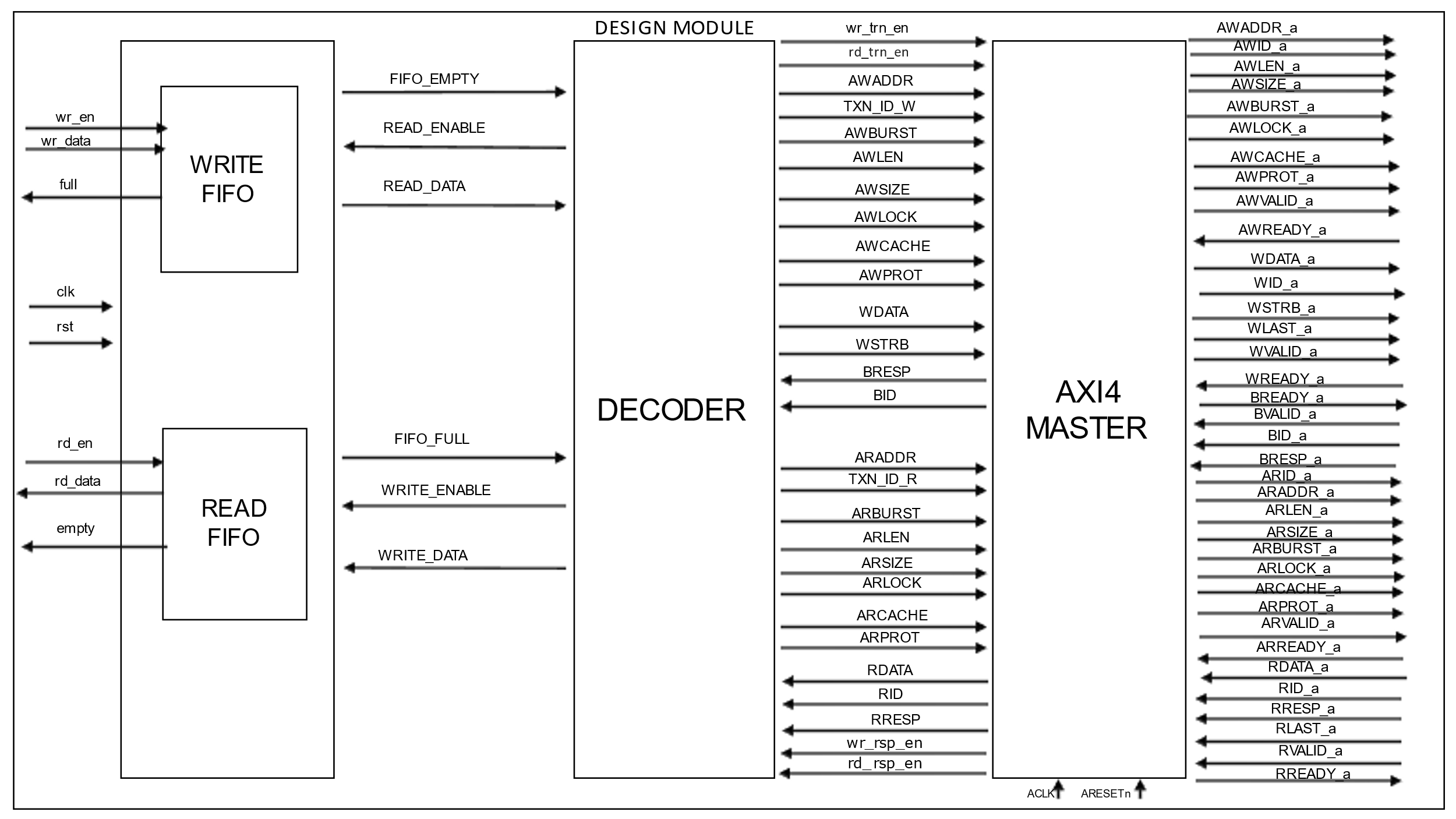
**1.1 AXI-FIFO BFM Features**

* Support single Write Transaction , Read Transaction (Different data size – 1Byte to 128Bytes )
* Support different Burst transaction ( Fixed, Increment and Wrap) with different burst length
* Supports Aligned and Unaligned Address Transaction

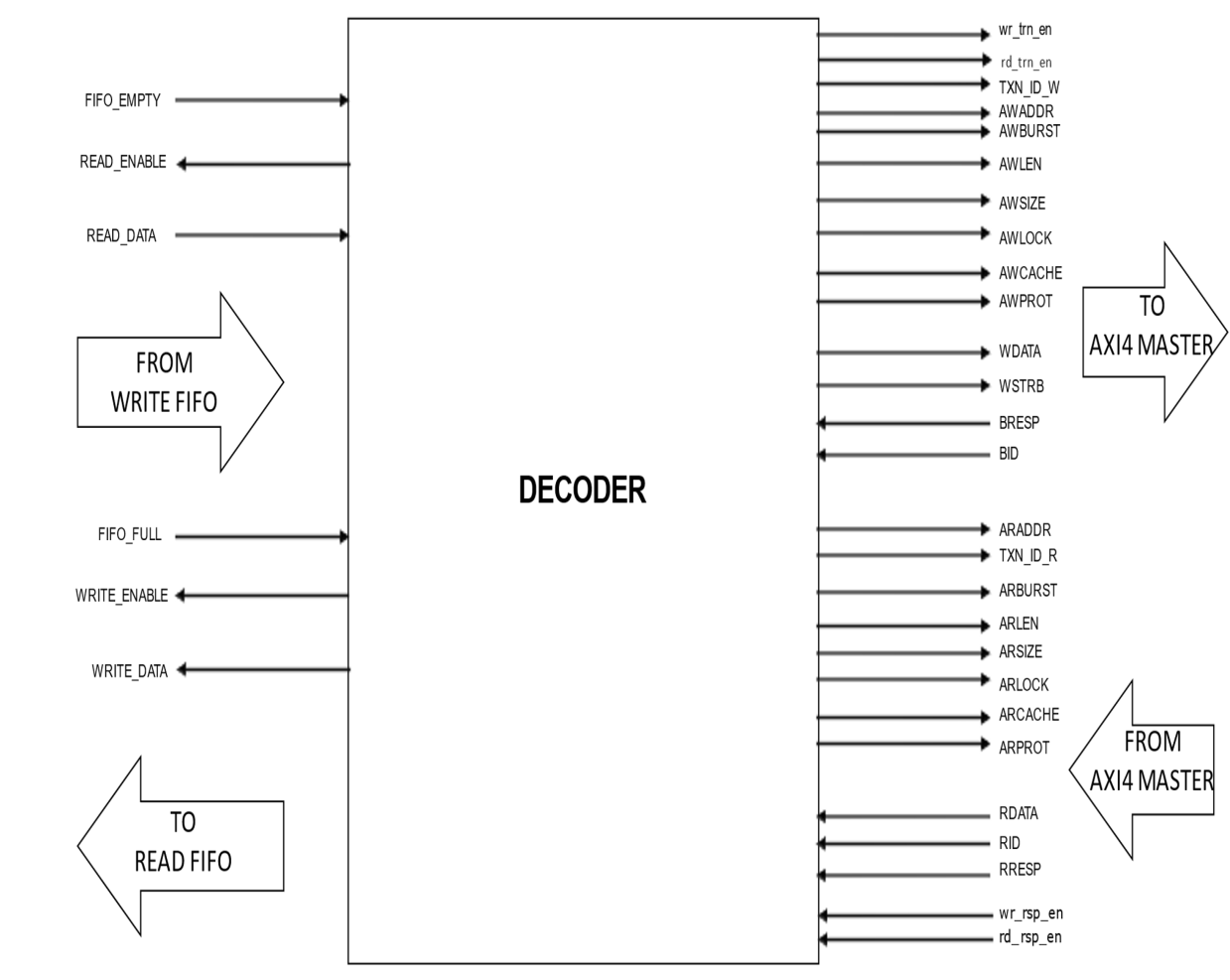
**1.2 Design Limitation**

* AXI4 master data width is 32bit hence data payload supported between 8 to 512 bits.
* Though AXI4 FIFO Depth in the design is 4096. Packet is fetched by Decoder from WRITE\_FIFO whenever non-empty occurs. Hence FIFO Features like Full Condition cannot be Verified

# Design diagram with interface signals



**Fig 1.1: AXI BFM Design block**



**Fig1.2 Decoder sub-block**

**2. Verification Architecture**

TOP

m\_AXI\_M\_sequencer

Virtual sequences

TEST

M\_Wr\_FIFO\_sequencer

M\_Rd\_FIFO\_sequencer

AXI\_FIFO\_BFM Packet Decoder

DRIVER

AXI Master

AXI\_SLAVE

Interface

MONITOR

WR\_FIFO

DRIVER

MONITOR

RD\_FIFO

DECODER

FIFO

Interface

ENV

WR\_FIFO\_BFM

Scoreboard

WR\_SEQs

SEQUENCER

RD\_FIFO\_BFM

WR

\_D(EXP) RD\_A(EXP) RD\_D(ACT)

RD\_SEQs

DRIVER

MONITOR

SEQUENCER

WR\_D(ACT) RD\_D(ACT), RD\_RSP(ACT) WR\_RSP(ACT)

Compare and Coverage logic

AXI Slave VIP

SEQUENCER

AXI\_M\_SEQs

WR\_D(ACT) , RD\_A(ACT) RD\_D(EXP), RD\_RSP(EXP), WR\_RSP(EXP)

AXI \_package

**Fig 2.1 Verification Architecture for AXI FIFO BFM**

**WR\_D** - Write address and Data (Master -> Slave)

**RD\_A** – Read address (Master -> Slave)

**RD\_D** – Read Data (Slave->Master)

**RD\_RSP –**Read Response (Slave->Master)

**WR\_RSP-**Write Response (slave ->Master)

**2.1 Verification Architecture:**

Verification Architecture of the AXI FIFO BFM consist of the following Verification components connected to the DUT

* Write FIFO Agent
* Read FIFO Agent
* AXI Slave VIP Agent

All Above 3 Verification Components are connected to the scoreboard for the data comparison in the environment.

Virtual Sequence to start the multiple sequences on different sequencers related to the above agents concurrently in the environment.

**3. Test bench Implementation**

**3.1 AXI-BFM WRITE AGENT**

* Agent in UVM is responsible for driving the stimulus to the DUT and monitoring its responses.
* Create an agent BFM specifically for the FIFO DUT.
* The agent includes a driver that drives data into the FIFO and a monitor observes the FIFO’s behaviour.

WR\_FIFO\_BFM

DRIVER

SEQUENCER

MONITOR

### Figure 3.1:Write\_fifo\_bfm

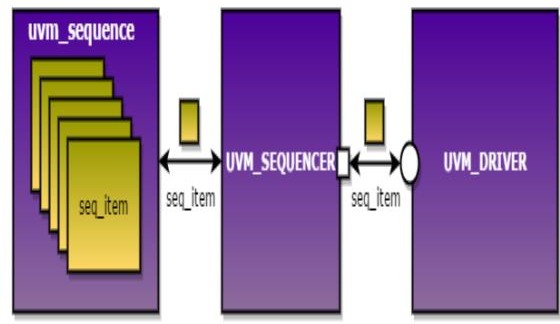
### 3.1.1 WRITE\_FIFO\_SEQ\_ITEM

The sequence-item consists of data fields required for generating the stimulus.In order to generate the stimulus, the sequence items are randomized in sequences. Therefore data properties in sequence items should generally be declared as rand and can have constraints defined. Data-fields represent the following types of information.

WR\_FIFO Sequence item supports wr\_data and rd\_data transfer.

**Write address/data phase packet** Packet = {SOP (8 bits) + TXN\_ID (4 bits) + ADDR (32 bits) + LEN (4 bits) + SIZE (3 bits) + BURST (2 bits) + LOCK (2 bits) + CACHE (2 bits) + PROT (3 bits) + STROBE (4 bits) + DATA (512 bits-> Variable Width 8 bits to 512 bits width ) + EOP (8 bits)}  
If the Data field >8bits and Data field! = 8’b00000000 then it is write address/data phase packet  
**Read address phase packet**Packet = {SOP (8 bits) + TXN\_ID (4 bits) + ADDR (32 bits) + LEN (4 bits) + SIZE (3 bits) + BURST (2 bits) + LOCK (2 bits) + CACHE (2 bits) + PROT (3 bits) + STROBE (4 bits) + DATA (8 bits) + EOP (8 bits)}  
If the Data field =8bits and Data field = 8’b00000000 then it is Read Address phase packet  
were:  
SOP= Start Of Packet =10101010  
EOP= End Of Packet = 01010011

**CONFIGURATION INFORMATION**:   
 A sequence generates a series of sequence\_item and sends it to the driver via sequencer, Sequence is written by extending the uvm\_sequence. We have two modes of operation: Write Mode and Read Mode.



**Figure 3.2: UVM\_SEQUENCE**

As analysis information fields will be used for capturing response, except these fields the other fields can be declared as rand and can have constraints associated with it.

### WRITE\_FIFO\_SEQUENCE\_ITEM Code

### `ifndef WRITE\_FIFO\_SEQ\_ITEM\_INCLUDED\_

### `define WRITE\_FIFO\_SEQ\_ITEM\_INCLUDED\_

### class write\_fifo\_seq\_item extends uvm\_sequence\_item;

### const bit [7:0] SOP=8'b10101010;

### const bit [7:0] EOP=8'b01010011;

### typedef enum bit{write\_addr\_phase\_pkt=0,read\_addr\_phase\_pkt=1} type\_of\_pkt\_e;

### rand type\_of\_pkt\_e type\_of\_pkt;

### // write signals

### rand bit[31:0] awaddr;

### rand bit[DATA\_WIDTH-1:0] wdata[$];

### rand bit[3:0] wstrb;

### rand bit[3:0] awlen;

### rand awid\_e awid;

### rand awsize\_e awsize;

### rand awburst\_e awburst;

### rand awlock\_e awlock;

### rand awcache\_e awcache;

### rand awprot\_e awprot;

### // read signals

### rand bit[31:0] araddr;

### rand bit[3:0] arlen;

### rand arid\_e arid;

### rand arsize\_e arsize;

### rand arburst\_e arburst;

### rand arlock\_e arlock;

### rand arcache\_e arcache;

### rand arprot\_e arprot;

### `uvm\_object\_utils\_begin(write\_fifo\_seq\_item)

### `uvm\_field\_enum(awid\_e,awid,UVM\_DEFAULT);

### `uvm\_field\_enum(awsize\_e,awsize,UVM\_DEFAULT);

### `uvm\_field\_enum(awburst\_e,awburst,UVM\_DEFAULT);

### `uvm\_field\_enum(awlock\_e,awlock,UVM\_DEFAULT);

### `uvm\_field\_enum(awcache\_e,awcache,UVM\_DEFAULT);

### `uvm\_field\_enum(awprot\_e,awprot,UVM\_DEFAULT);

### `uvm\_field\_int(awlen,UVM\_DEFAULT);

### `uvm\_field\_int(awaddr,UVM\_DEFAULT);

### `uvm\_field\_queue\_int(wdata,UVM\_DEFAULT);

### `uvm\_field\_int(wstrb,UVM\_DEFAULT);

### `uvm\_field\_enum(arid\_e,arid,UVM\_DEFAULT);

### `uvm\_field\_enum(arsize\_e,arsize,UVM\_DEFAULT);

### `uvm\_field\_enum(arburst\_e,arburst,UVM\_DEFAULT);

### `uvm\_field\_enum(arlock\_e,arlock,UVM\_DEFAULT);

### `uvm\_field\_enum(arcache\_e,arcache,UVM\_DEFAULT);

### `uvm\_field\_enum(arprot\_e,arprot,UVM\_DEFAULT);

### `uvm\_field\_int(arlen,UVM\_DEFAULT);

### `uvm\_field\_int(araddr,UVM\_DEFAULT);

### `uvm\_field\_enum(type\_of\_pkt\_e,type\_of\_pkt,UVM\_DEFAULT);

### `uvm\_object\_utils\_end

### // constraints for write signals

### constraint wdata\_size{wdata.size()==awlen+1;}

### constraint awaddr\_alligned{soft awaddr%32==0;}

### constraint awsize\_c1{soft awsize == WRITE\_4\_BYTES;}

### constraint wstrb\_c1{soft wstrb ==4'hf;}

### // constraint awlen\_c1{soft awlen ==2;}

### constraint awlock\_c1{soft awlock == WRITE\_NORMAL\_ACCESS;}

### constraint awprot\_c1{soft awprot == WRITE\_NORMAL\_SECURE\_DATA;}

### constraint awcache\_c1{soft awcache == WRITE\_BUFFERABLE;}

### constraint awburst\_c1{soft awburst == WRITE\_INCR;}

### constraint awburst\_c2{awburst != WRITE\_RESERVED;}

### // constraints for read signals

### constraint araddr\_alligned{soft araddr%32==0;}

### constraint arsize\_c1{soft arsize == READ\_4\_BYTES;}

### constraint arlock\_c1{soft arlock == READ\_NORMAL\_ACCESS;}

### constraint arprot\_c1{soft arprot == READ\_NORMAL\_SECURE\_DATA;}

### constraint arcache\_c1{soft arcache == READ\_BUFFERABLE;}

### constraint arburst\_c1{soft arburst == READ\_INCR;}

### constraint arburst\_c2{arburst != READ\_RESERVED;}

### function new(string name="write\_fifo\_seq\_item");

### super.new(name);

### endfunction

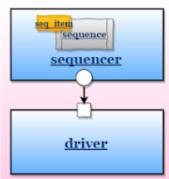
### endclass

### `endif

**3.1.2 UVM SEQUENCER**

The Sequencer controls the flow of request and response sequence items between the sequences and the driver.

* + - * Sequencer and driver uses TLM Interface to communicate transactions
      * Sequencer and driver classes have seq\_item\_export and seq\_item\_port defined respectively. User needs to connect them using TLM connect method.

A sequencer can be written by extending the uvm\_sequencer parameterized with the seq\_item type.

**3.1.2.1 WRITE\_FIFO\_SEQUENCER CODE**

`ifndef WRITE\_FIFO\_SEQUENCER\_INCLUDED\_

`define WRITE\_FIFO\_SEQUENCER\_INCLUDED\_

//--------------------------------------------------------------------------------------------

// Class: write\_fifo\_sequencer

// <Description\_here>

//--------------------------------------------------------------------------------------------

class write\_fifo\_sequencer extends uvm\_sequencer#(write\_fifo\_seq\_item);

`uvm\_component\_utils(write\_fifo\_sequencer)

//-------------------------------------------------------

// Externally defined Tasks and Functions

//-------------------------------------------------------

extern function new(string name = "write\_fifo\_sequencer", uvm\_component parent = null);

extern virtual function void build\_phase(uvm\_phase phase);

endclass : write\_fifo\_sequencer

//--------------------------------------------------------------------------------------------

// Construct: new

// Parameters:

// name - write\_fifo\_sequencer

// parent - parent under which this component is created

//--------------------------------------------------------------------------------------------

function write\_fifo\_sequencer::new(string name = "write\_fifo\_sequencer", uvm\_component parent=null);

super.new(name, parent);

endfunction : new

//--------------------------------------------------------------------------------------------

// Function: build\_phase

// <Description\_here>

// Parameters:

// phase - uvm phase

//--------------------------------------------------------------------------------------------

function void write\_fifo\_sequencer::build\_phase(uvm\_phase phase);

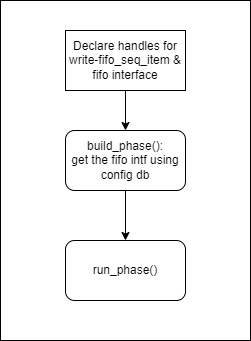
super.build\_phase(phase);

endfunction : build\_phase

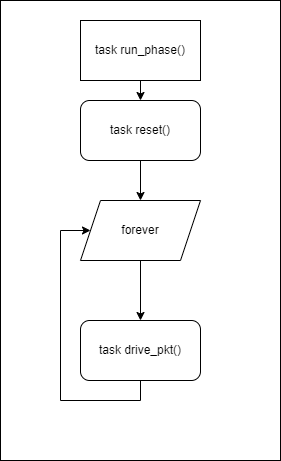
`endif

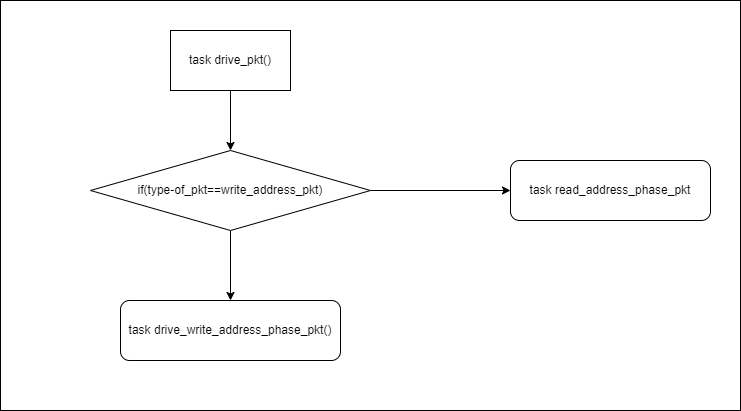
**3.1.3 Write Agent Driver:**

The responsibility of write\_fifo\_driver is to drive the packet of either write\_address\_phase\_pkt or read\_address\_phase\_pkt based on the control of type\_of\_pkt in write\_fifo\_seq\_item to the FIFO interface.

**Flow chart of driver class:**

**Flow chart of driver run\_phase:**





**Flowchart of drive\_pkt() task:**

No

Yes

**Logic of task: drive\_write\_address\_phase\_pkt():**

Step 1: Take 8 bit width of queue and pass each 8 bits in seq.wdata[0] in to this queue like below

if(write\_fifo\_seq\_item\_h.wstrb == 4'b0001)

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][7:0]);

else if(write\_fifo\_seq\_item\_h.wstrb == 4'b0011)

begin

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][15:8]);

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][7:0]);

end

else if(write\_fifo\_seq\_item\_h.wstrb == 4'b0111)

begin

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][23:16]);

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][15:8]);

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][7:0]);

end

else if(write\_fifo\_seq\_item\_h.wstrb == 4'b1111)

begin

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][31:24]);

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][23:16]);

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][15:8]);

q.push\_back(write\_fifo\_seq\_item\_h.wdata[0][7:0]);

end

Step 2: Now pass remaining transfers to the above declared queue if the no of transfers is greater than 1. foreach(write\_fifo\_seq\_item\_h.wdata[i])

begin

if(i>0)

begin

for(int j =0; j<4;j++)

begin

q.push\_back(write\_fifo\_seq\_item\_h.wdata[i][31-(8\*j)-:8]);

end

end

end

Step 3: Calculate the number of bits in one burst.

No\_of\_bits\_in\_q = q.size()\*8;

Step 4: Concatenate the transfers in the q queue elements and store in bit[1023:0] wdata variable.

after concatenatinf it looks like bit[1023:0] wdata = 0000….wdata

Step 5: Shift the bits of wdata to the left side. After shifting it looks like

Bit[1023:0] wdata = wdata000….000  
 here wdata width is 1024 because the max value wdata field in spec is 1024.

Step 6: Concatenate the constant fields, above variable wdata like below .

Here total\_packet width is 1152 because the max packet has 1096 bits which occupies 9 locations in FIFOs, so 9x128=1152.

bit[1151:0] total\_packet = {write\_fifo\_seq\_item\_h.SOP,

write\_fifo\_seq\_item\_h.awid,

write\_fifo\_seq\_item\_h.awaddr,

write\_fifo\_seq\_item\_h.awlen,

write\_fifo\_seq\_item\_h.awsize,

write\_fifo\_seq\_item\_h.awburst,

write\_fifo\_seq\_item\_h.awlock,

write\_fifo\_seq\_item\_h.awcache,

write\_fifo\_seq\_item\_h.awprot,

write\_fifo\_seq\_item\_h.wstrb,

wdata,64'h0};

Step 7: assign EOP to bit[1151:0] duplicate\_eop, then it looks like

Bit[1151:0] duplicate\_eop = 000….EOP

Step 8: shift the duplicate eop such that the starting bit of EOP should be after the end of wdata duplicate\_EOP = duplicate\_EOP<< 1152-(64+no\_of\_bits\_in\_wdata)-8;

Step 9: Now add the total\_packet with duplicate\_eop

Bit[1151:0] total\_pcaket = constantfiedswdata 00000..000

Bit[1151:0] duplicate\_eop = 0000……………… EOP …000

--------------------------------------------------------------------------------------

Bit[1151:0] total\_packet = constantfieldswdata EOP 000…000

Step 10: if no\_of\_bits\_in\_wdata<=56 means then the packet size is within 128 bits. The drive the 128 bits from MSB of total\_packet to the fifo\_interface by making wr\_en = 1 like below.

wait(!intf.full);

@(posedge(intf.clk));

intf.wr\_en<= 1;

intf.wr\_data<= total\_packet[1151-:128];

Step 11 : if no\_of\_bits\_in\_wdata> 56 means then the packet size is more than 128 bits. Then calculate the how many times of 128 bits to be send like below.

pkt\_width = 72+no\_of\_bits\_in\_wdata;

if(pkt\_width%128==0)

no\_of\_iter = pkt\_width/128;

else

no\_of\_iter = (pkt\_width/128)+1;

Step 12: Now send the each 128 bits from packet to the interface like below:

for(int i = 0; i<no\_of\_iter; i++) begin

wait(!intf.full);

@(posedge(intf.clk));

intf.wr\_en<= 1;

intf.wr\_data<= total\_packet[1151-(128\*i) -: 128];

end

**3.1.4 Write Agent monitor:**

Write Agent Monitor samples the interface signal (wdata[128:0]) and convert into sequence item and send to the scoreboard through the TLM analysis port.

Steps involved in the Monitor:

1. Then declare the virtual interface handle and analysis port.
2. Monitor collect first 128-bit data from the interface (write\_data[127:0]) signal. The first Write\_Data[127:64] gives the control information and from which it can classify whether it is Write packet or Read packet.

III. The first row is 64 8888 8888 in one row in 128 bit.

64 bit consists SOP and other control fields and (8888 8888) in 64 bit in 128 bit.

If the count of no of eight 8’s in one row, we will use the logic intf.wr\_data[63-8\*a -: 8].a is the variable used to iterate the first row elements from [0 to 7]a . If the count is more than 8 , we can use the logic intf.wr\_data[63-8\*b -: 8]) to push the elements to the queue. ‘b’ value range from [8 :$].We need to find out number of iterations gives number of 8's/128 ==0 and push into queue in 8 bit format. Push the remaining row elements to the queue in 8bit form. Here if the no of row elements are not divisible by no of 8’s/128 !=0, we need to find the number of remaining 8 bits are there and we should use the formula.  The no of iterations gives no\_of\_bits\_remaining/128. Find  the  no of  8 bit remaining  by  using (no\_of\_bits\_remaining(no\_of\_bits\_remaining/128)\*128)/8 and push into byte queue in 8 bit format.

The pushed elements in byte queue will be like this:

**64  8888  8888  
8888 8888 8888 8888  
8888 8888 8888 8888   
8888 8888 8888 8888   
8888 8888 EOP**

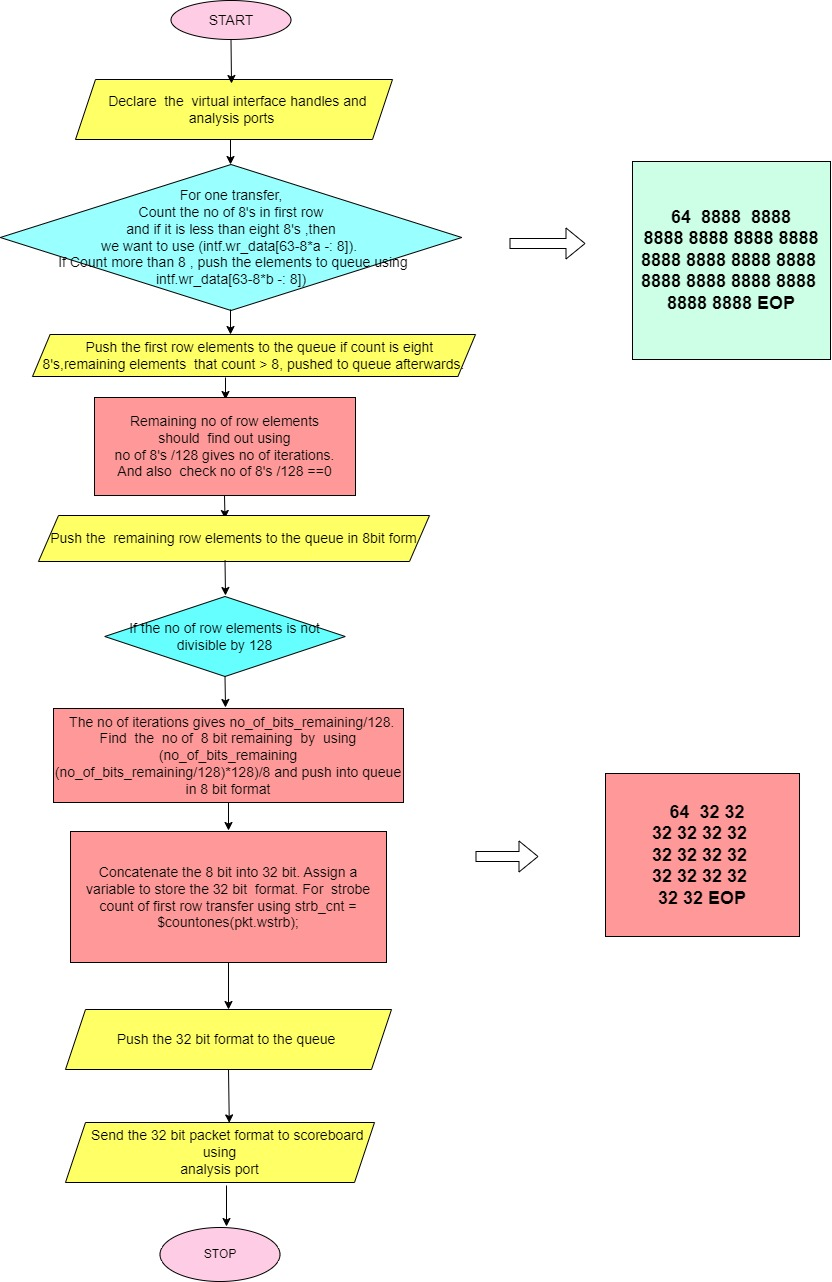
Then concatenate the 8 bit into 32 bit and assign to wdata\_32. Assign a variable to store the 32-bit format. For strobe count of first row transfer using strb\_cnt = $countones(pkt.wstrb) and find the no of bits using and push to queue in 32 bit format.

The format should be like this:

**64  32 32   
32 32 32 32   
32 32 32 32   
32 32 32 32   
32 32 EOP**

Push the 32 bit format to the queue.Send the 32 bit packet format to send from monitor to scoreboard using analysis port

**3.1.4.1Write Agent monitor flowchart**



* 1. **Read FIFO Agent**

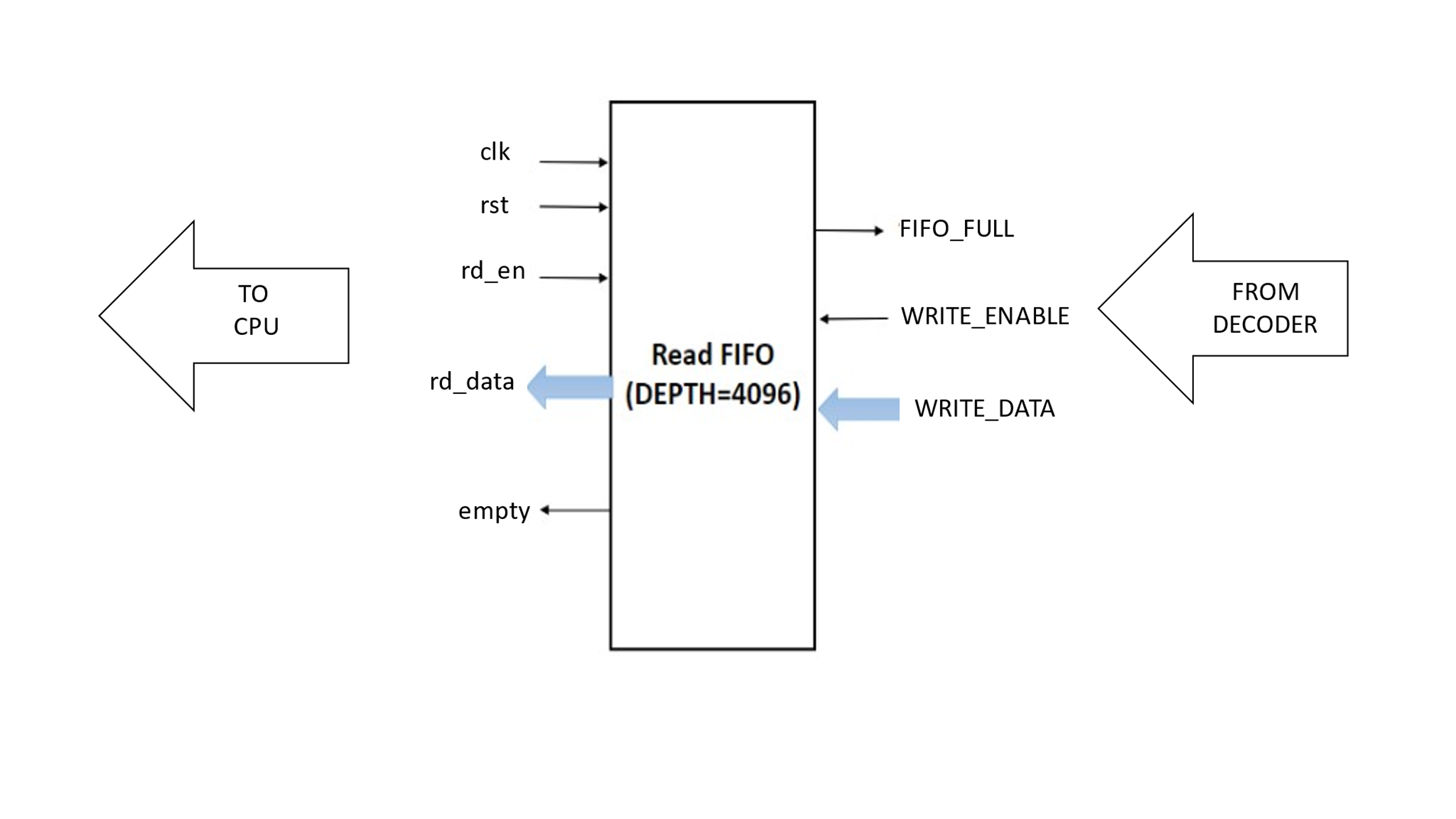
**3.2.1. Functional description of Read FIFO:**

The decoder writes two types of packets into the Read FIFO:

1. Write response phase packets
2. Read data/response phase packets

The CPU reads the response packets from the read fifo. The read fifo is synchronous and 128 bits wide and 4096 depth.

**3.2.2 Read FIFO interface signals:**

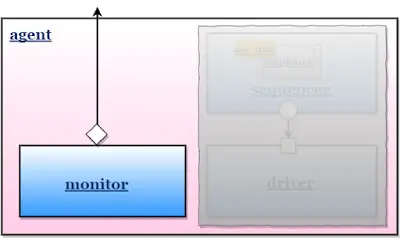


**3.2.3 Description of Read FIFO interfaces signals:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Signal name** | **Size in bits** | | **Direction** | **Description** |
| Global signals | | | | |
| Clk | 1 | | Input | Global Clock signal with frequency of 100 Mhz. |
| Rst | 1 | | Input | Global reset signal |
| Read FIFO | | | | |
| WRITE\_ENABLE | | 1 | Input | Write enable. This signal enables to write the data into the read fifo. |
| rd\_en | | 1 | Input | Read Enable. This signal enables to read the data from the read fifo. |
| WRITE\_DATA | | 128 | Input | Write Data. This signal specifies the data to write into the fifo. |
| Empty | | 1 | output | FIFO Empty. This signal indicates the fifo empty condition. |
| FIFO\_FULL | | 1 | output | FIFO Full. This signal indicates the fifo full condition. |
| rd\_data | | 128 | output | Read data. This signal specifies the data read from the fifo. |

**3.2.4 READ\_FIFO\_AGENT:**

* A user-defined agent is extended from uvm\_agent, uvm\_agent is inherited by uvm\_component.An agent typically contains a driver, a sequencer, and a monitor,
* Agents can be configured either active or passive, In our project for read FIFO we are going to use passive agent.
* Passive agents sample DUT signals but do not drive them,A passive agent consists of only the monitor



**3.2.5 READ\_FIFO\_AGENT Code Snippet:**

`ifndef READ\_FIFO\_AGENT\_INCLUDED\_

`define READ\_FIFO\_AGENT\_INCLUDED\_

//--------------------------------------------------------------------------------------------

// Class: read\_fifo\_agent

// <Description\_here>

//--------------------------------------------------------------------------------------------

class read\_fifo\_agent extends uvm\_component;

`uvm\_component\_utils(read\_fifo\_agent)

//variable mon\_h

//Declaring the monitor handle

read\_fifo\_monitor read\_monitor;

//-------------------------------------------------------

// Externally defined Tasks and Functions

//-------------------------------------------------------

extern function new(string name = "read\_fifo\_agent", uvm\_component parent = null);

extern virtual function void build\_phase(uvm\_phase phase);

extern virtual function void connect\_phase(uvm\_phase phase);

endclass : read\_fifo\_agent

//--------------------------------------------------------------------------------------------

// Construct: new

// name - read\_fifo\_agent

// parent - parent under which this component is created

//--------------------------------------------------------------------------------------------

function read\_fifo\_agent::new(string name = "read\_fifo\_agent",

uvm\_component parent = null);

super.new(name, parent);

endfunction : new

//--------------------------------------------------------------------------------------------

// Function: build\_phase

// <Description\_here>

// Parameters:

// phase - uvm phase

//--------------------------------------------------------------------------------------------

function void read\_fifo\_agent::build\_phase(uvm\_phase phase);

super.build\_phase(phase);

read\_monitor=read\_fifo\_monitor::type\_id::create("read\_monitor",this);

endfunction : build\_phase

//--------------------------------------------------------------------------------------------

// Function: connect\_phase

// <Description\_here>

//

// Parameters:

// phase - uvm phase

//--------------------------------------------------------------------------------------------

function void read\_fifo\_agent::connect\_phase(uvm\_phase phase);

super.connect\_phase(phase);

endfunction : connect\_phase

`endif

**3.2.6 READ\_FIFO\_MONITOR:**

* The user-defined monitor is extended from uvm\_monitor, uvm\_monitor is inherited by uvm\_component.
* A monitor is a passive entity that samples the DUT signals through the virtual interface and converts the signal level activity to the transaction level.
* Monitor samples DUT signals but does not drive them. The monitor should have an analysis port (TLM port) and a virtual interface handle that points to DUT signals.

The steps for read\_fifo\_monitor :

1. The monitor is written by extending the UVM\_MONITOR
2. Declare virtual interface

3. Connect interface to Virtual interface by using get method,

4. Declare analysis port,

5. Declare seq\_item handle, Used as a place holder for sampled signal activity,

6. Add Sampling logic in run\_phase,

7. After sampling, by using the write method send the sampled transaction packet to the scoreboard.

**READ\_FIFO\_MONITOR Code Snippet:**

`ifndef READ\_FIFO\_MONITOR\_INCLUDED\_

`define READ\_FIFO\_MONITOR\_INCLUDED\_

class read\_fifo\_monitor extends uvm\_monitor;

`uvm\_component\_utils(read\_fifo\_monitor)

//Declare virtual interface handle with WMON\_MP as modport

virtual fifo\_if.RMON\_MP vif;

//Declare the fifo\_sequence\_item and handle

fifo\_sequence\_item data\_received;

// Analysis TLM port to connect the monitor to the scoreboard

uvm\_analysis\_port #(fifo\_sequence\_item) rd\_monitor\_port;

///////METHODS

extern function new(string name = "read\_fifo\_monitor", uvm\_component parent);

extern function void build\_phase(uvm\_phase phase);

extern function void connect\_phase(uvm\_phase phase);

extern task run\_phase(uvm\_phase phase);

extern task collect\_data();

endclass

//----------------- constructor new method -------------------//

function new(string name = "read\_fifo\_monitor", uvm\_component parent = null);

super.new(name, parent);

// create object for handle monitor\_port using new

rd\_monitor\_port = new("rd\_monitor\_port", this);

endfunction

//----------------- build() phase method -------------------//

function void read\_fifo\_monitor::build\_phase(uvm\_phase phase);

super.build\_phase(phase);

endfunction

//----------------- connect() phase method -------------------//

function void read\_fifo\_monitor::connect\_phase(uvm\_phase phase);

endfunction

//----------------- run() phase method -------------------//

// In forever loop

// Call task collect\_data

task read\_fifo\_monitor::run\_phase(uvm\_phase phase);

forever begin

collect\_data();

end

endtask

task read\_fifo\_monitor::collect\_data();

// Declare handle to collect data as data\_sent

fifo\_sequence\_item data\_sent;

// Create an instance data\_sent

data\_sent= fifo\_sequence\_item::type\_id::create("data\_sent");

wait(!vif.rmon\_cb.rd\_en)

@(posedge vif.rmon\_cb);

data\_sent.rd\_data = vif.rmon\_cb.rd\_data;

data\_sent.empty = vif.rmon\_cb.empty;

rd\_monitor\_port.write(data\_sent);

endtask

* 1. **AXI4 AVIP Slave Configuration**

AXI AVIP Slave Configuration can be configured as shown below according to the design requirement .

//===============================

package axi4\_globals\_pkg;

//-------------------------------------------------------

// Parameters used in axi4\_avip are given below

//-------------------------------------------------------

//Parameter: SLAVE\_AGENT\_ACTIVE

//Used to set the slave agent either active or passive

parameter bit SLAVE\_AGENT\_ACTIVE = 1;

//Parameter: NO\_OF\_SLAVES

//Used to set number of slaves required

parameter int NO\_OF\_SLAVES = 1;

//Parameter: ADDRESS\_WIDTH

//Used to set the address width to the address bus

parameter int ADDRESS\_WIDTH = 32;

//Parameter: DATA\_WIDTH

//Used to set the data width

parameter int DATA\_WIDTH = 32;

//Parameter: SLAVE\_MEMORY\_SIZE

//Sets the memory size of the slave in KB

parameter int SLAVE\_MEMORY\_SIZE = 12;

//Parameter: SLAVE\_MEMORY\_GAP

//Sets the memory gap size of the slave

parameter int SLAVE\_MEMORY\_GAP = 2;

//Parameter: MEMORY\_WIDTH

//Sets the width it can store in each location

parameter int MEMORY\_WIDTH = 8;

//Parameter: STROBE\_WIDTH

//Used to define the width of the strobes

parameter int STROBE\_WIDTH = (DATA\_WIDTH/8);

//Variable: MEM\_ID

//Indicates Slave Memory Depth

parameter int MEM\_ID = 2\*\*ADDRESS\_WIDTH;

//Variable: LENGTH

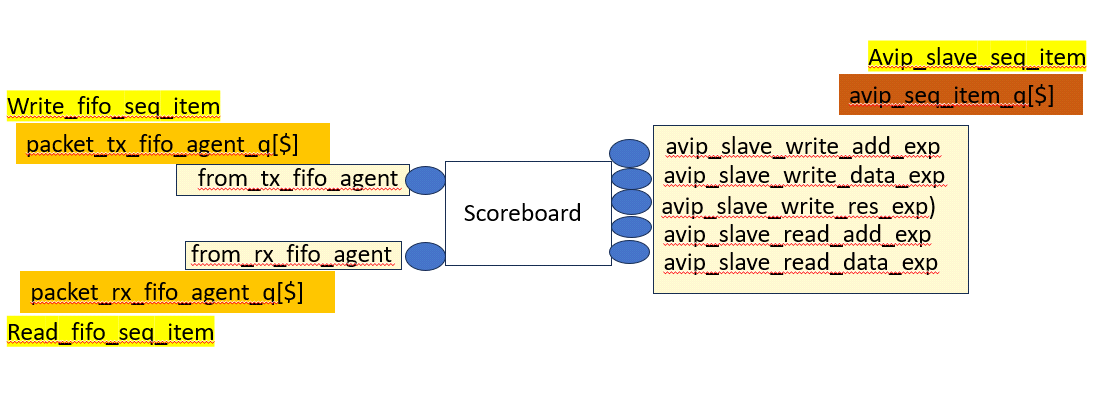
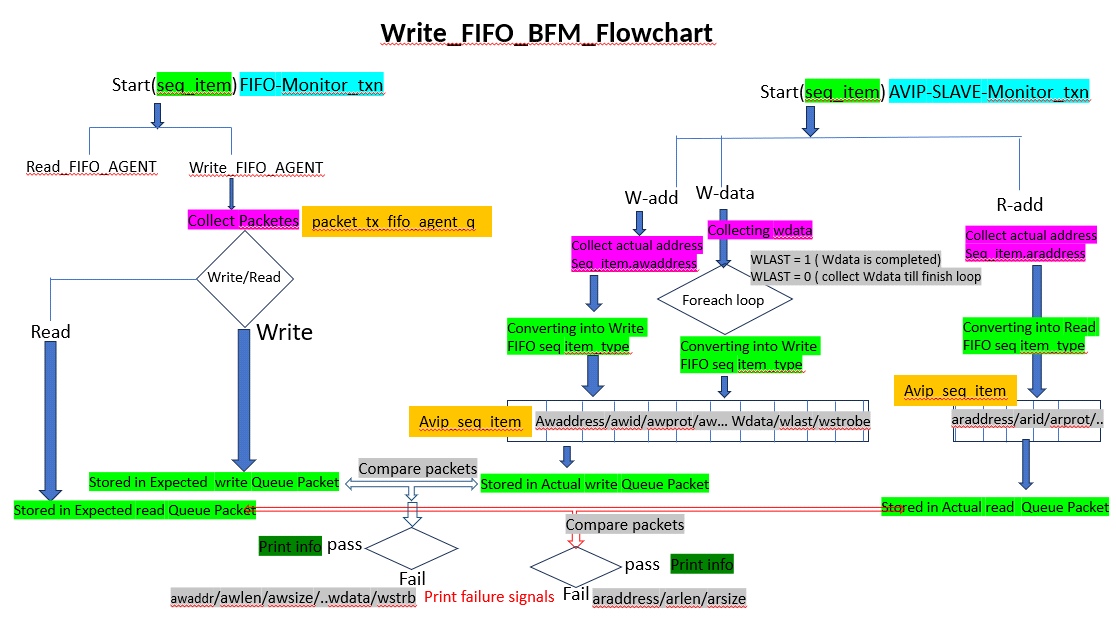
//Indicates the length of the address write and read channels

parameter int LENGTH = 4;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3.4 AXI-BFM-SCOREBOARD**

**Basic ports Overview: -**

**Flowchart: -**

**Fig.3.1: Flow Chart of Scoreboard**

**3.4.1 Coding Steps:-**

* **fifo\_scoreboard.sv** is created in env and included in env\_package.

"`**include fifo\_scoreboard.sv**"

* fifo monitor **analysis port (write & read)** is connected to the respective scoreboard implementation ports in the fifo\_scoreboard to be done in env class.

**"fifo\_agent.monitor.monitor\_port.connect(scoreboard.scb\_exp)"**

* axi slave monitor analysis port (wr\_addr,w\_data,wr\_resp,rd\_addr & rd\_data) is connected to the respective scoreboard implementation ports in the **fifo\_scoreboard** to be done in env class

**"axi\_slave\_agent.monitor.monitor\_port.connect(scoreboard.scb\_exp)"**

* axi\_fifo\_scoreboard is extended from "uvm\_scoreboard."
* local implementation ports for each port from monitor are declared and created inside the fifo\_scoreboard using uvm\_analysis\_dec and new ()

**"`uvm\_analysis\_imp\_dec(tx\_fifo\_agent)" "`uvm\_analysis\_imp\_dec(rx\_fifo\_agent)"**

* There are 7 implementation ports declared inside the fifo\_scoreboard (fifo write & read, wr\_addr,wr\_data,wr\_resp,rd\_addr & rd\_data ).

**"`uvm\_analysis\_imp\_dec(avip\_slave\_write\_add\_exp)"**

**"`uvm\_analysis\_imp\_dec(avip\_slave\_write\_data\_exp)"**

**"`uvm\_analysis\_imp\_dec(avip\_slave\_write\_res\_exp)"**

**"`uvm\_analysis\_imp\_dec(avip\_slave\_read\_add\_exp)"**

**"`uvm\_analysis\_imp\_dec(avip\_slave\_read\_data\_exp)"**

* Create a write method for each implementation port which receives the seq\_item transactions.

**“function void write\_from\_tx\_fifo\_agent(write\_fifo\_seq\_item tx\_pkt)”**

**“function void write\_from\_rx\_fifo\_agt(seq\_item rx\_pkt)”**

**“function void write\_avip\_slave\_write\_add\_exp(axi4\_slave\_tx rx\_pkt)”**

* The expected write port will collect the fifo write & fifo read in the same write and read fifo seq\_item format and it pushes into the expected queue.

“packet\_tx\_fifo\_agent\_q.push\_back(tx\_pkt)”

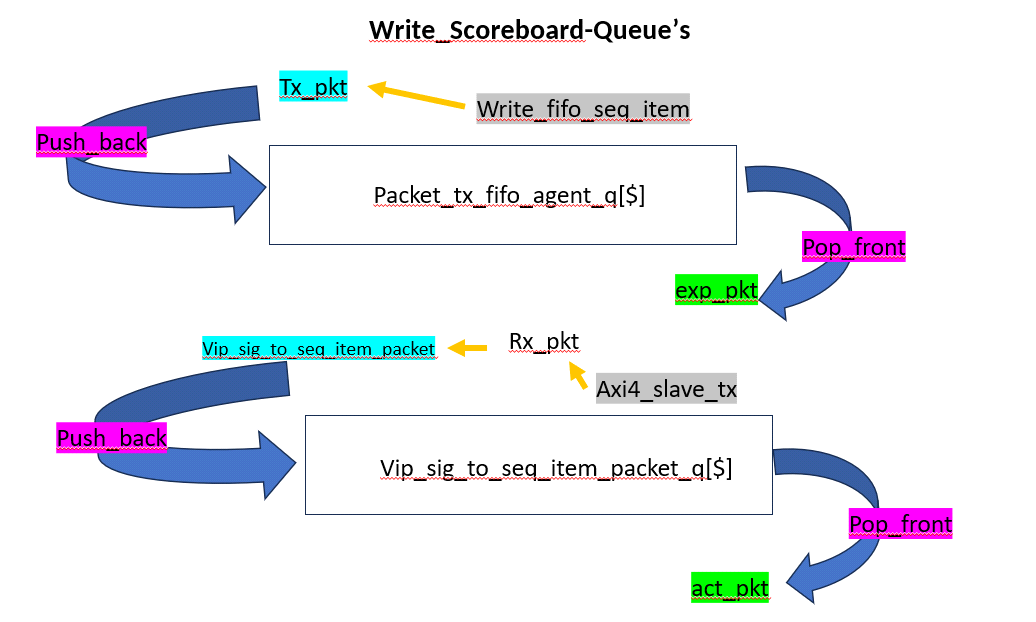
* The transactions from the slave side is axi signals which are converted into write/read fifo seq\_item format based on the 5 channel ports & their signals when received.

“vip\_sig\_to\_seq\_item\_packet\_q.push\_back(vip\_sig\_to\_seq\_item\_packet)”

* We moved all signals into exp\_pkt and act\_pkt.

exp\_pkt = packet\_tx\_fifo\_agent\_q.pop\_front ();

act\_pkt = vip\_sig\_to\_seq\_item\_packet\_q.pop\_front ();



* Compare all control Signals. (Write\_Address\_Channel)

“exp\_pkt.awaddr == act\_pkt.awaddr”

“exp\_pkt.awid == act\_pkt.awid”

“exp\_pkt.awlen == act\_pkt.awlen”

“exp\_pkt.awsize == act\_pkt.awsize”

“exp\_pkt.awcache == act\_pkt.awcache “

* Compare all data signals. (Write\_Data\_Channel)

“exp\_pkt.wdata == act\_pkt.wdata”

“exp\_pkt.wstrb == act\_pkt.wstrb”

“exp\_pkt.wlast == act\_pkt.wlast”

* Compare all read control Signals. (Read\_Address\_Channel)

“exp\_pkt.araddr == act\_pkt.araddr”

“exp\_pkt.arid == act\_pkt.arid”

“exp\_pkt.arlen == act\_pkt.arlen”

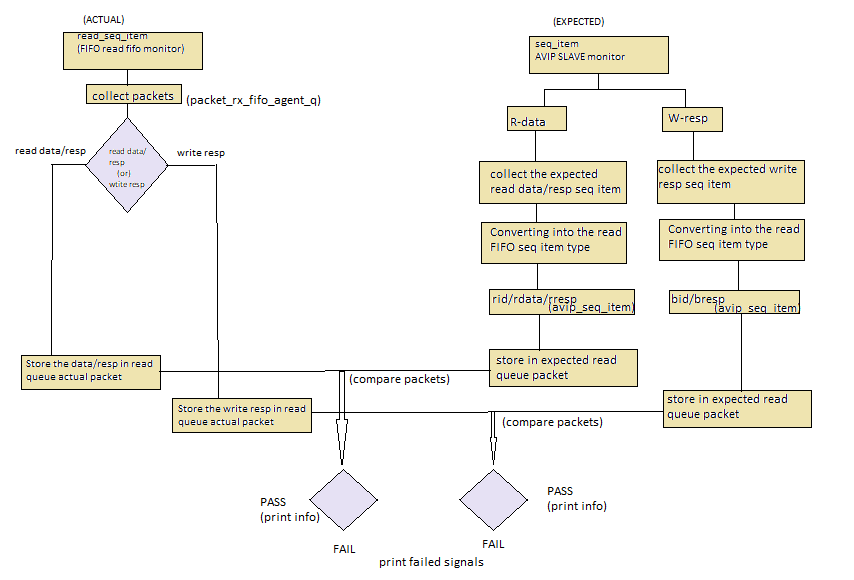
“exp\_pkt.arsize == act\_pkt.arsize”

“exp\_pkt.arcache == act\_pkt.arcache “

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Read fifo bfm: (TODO impl)

Read fifo bfm flowchart

Connect read fifo bfm monitor analysis port to scoreboard read fifo implementation port in env class

Connect axi avip monitor analysis port to scoreboard read implementation ports in env class

Collect the read data in one queue till rlast is 1 for rdata.

* The actual read port will collect the fifo read data/resp in the read fifo seq\_item format and it pushes into the actual queue.

“packet\_rx\_fifo\_agent\_q.push\_back(rx\_pkt)”

* The expected transactions from the slave side is axi signals which are converted into read fifo seq\_item format based on the 2 channel ports & their signals when received.

“vip\_sig\_to\_seq\_item\_packet\_q.push\_back(vip\_sig\_to\_seq\_item\_packet)”

* Move both the collected packets into exp\_pkt and act\_pkt.

act\_pkt = packet\_rx\_fifo\_agent\_q.pop\_front ();

exp\_pkt = vip\_sig\_to\_seq\_item\_packet\_q.pop\_front ();

* Compare all read data signals. (Read\_Data\_Channel)

“exp\_pkt.rid == act\_pkt.rid”

“exp\_pkt.rdata == act\_pkt.rdata”

“exp\_pkt.rresp == act\_pkt.rresp”

* Compare all write response signals. (Write\_response\_Channel)

“exp\_pkt.bid == act\_pkt.bid”

“exp\_pkt.bresp == act\_pkt.bresp”

**3.5 Coverage**

**3.5.1 Functional Coverage**

Functional coverage is the coverage data generated from the user defined functional coverage model written in System Verilog. During simulation, the simulator generates functional coverage based on the stimulus. Looking at the functional coverage data, one can identify the portions of the DUT [Features] verified and it helps us to target the DUT features that are unverified.

The reason for switching to the functional coverage is that we can create the bins manually as per our requirement while in the code coverage it is generated by the system by itself.

**3.5.2 Uvm\_Subscriber**

This class provides an analysis export for receiving transactions from a connected analysis export. Making such a connection "subscribes" this component to any transactions emitted by the connected analysis port. Subtypes of this class must define the write method to process the incoming transactions. This class is particularly useful when designing a coverage collector that attaches to a monitor.

**3.5.2.1 Analysis export**

This export provides access to the write method, which derived subscribers must implement.

**3.5.2.2 Write function**

The write function is to process the incoming transactions.

**3.5.2.3 Covergroup**

Covergroup consists of cover-points to be sampled as shown in the code below and required cross of the cover points to check whether we have covered all the required scenario during the simulation.

//=====================================================================

covergroup axi\_bfm\_covergroup with function sample (write\_fifo\_seq\_item packet);

option.per\_instance = 1;

//-------------------------------------------------------

// Write channel signals

//-------------------------------------------------------

AWLEN\_CP : coverpoint packet.awlen {

option.comment = "Write Address Length values";

bins AWLEN\_1 = {0};

bins AWLEN\_2 = {1};

bins AWLEN\_4 = {3};

bins AWLEN\_8 = {7};

bins AWLEN\_16 = {15};

bins AWLEN\_32 = {31};

bins AWLEN\_64 = {63};

bins AWLEN\_128 = {127};

bins AWLEN\_256 = {255};

bins AWLEN\_DEFAULT = default ;

}

AWBURST\_CP : coverpoint packet.awburst {

option.comment = "Write Address Burst values";

bins READ\_FIXED = {0};

bins WRITE\_INCR = {1};

//bins READ\_WRAP = {2};

illegal\_bins ILLEGAL\_BIN\_OF\_AWBURST = {3};

}

AWSIZE\_CP : coverpoint packet.awsize {

option.comment = "Write Address size values";

bins AWSIZE\_4BYTES = {2};

}

AWLOCK\_CP :coverpoint packet.awlock {

option.comment = "Write Address Lock values";

bins AWLOCK[] = {0,1};

}

AWCACHE\_CP : coverpoint packet.awcache {

option.comment = "Write Address Cache values";

bins WRITE\_BUFFERABLE = {[0:3]};

}

AWPROT\_CP : coverpoint packet.awprot {

option.comment = "Write Address Protection values";

bins AWPROT = {[0:$]};

}

AWID\_CP : coverpoint packet.awid {

option.comment = "Write Address ID values";

bins AWID[] = {[0:$]};

}

//-------------------------------------------------------

// Read channel signals

//-------------------------------------------------------

ARLEN\_CP : coverpoint packet.arlen {

option.comment = "Read Address Length values";

bins ARLEN\_1 = {0};

bins ARLEN\_2 = {1};

bins ARLEN\_4 = {3};

bins ARLEN\_8 = {7};

bins ARLEN\_16 = {15};

}

ARBURST\_CP : coverpoint packet.arburst {

option.comment = "Read Address Burst values";

bins READ\_FIXED = {0};

bins WRITE\_INCR = {1};

illegal\_bins ILLEGAL\_BIN\_OF\_ARBURST = {3};

}

ARSIZE\_CP : coverpoint packet.arsize {

option.comment = "Read Address Size values";

bins ARSIZE\_4BYTES = {2};

}

ARLOCK\_CP :coverpoint packet.arlock {

option.comment= "Read Address Lock values";

bins ARLOCK = {0,1};

}

ARCACHE\_CP : coverpoint packet.arcache {

option.comment = "Read Address Cache values";

bins READ\_BUFFERABLE = {[0:3]};

}

ARPROT\_CP : coverpoint packet.arprot {

option.comment = "Read Address Protection values";

bins ARPROT = {[0:$]};

}

//-------------------------------------------------------

// Cross of coverpoints

//-------------------------------------------------------

AWLENGTH\_CP\_X\_AWSIZE\_X\_AWBURST :cross AWLEN\_CP,AWSIZE\_CP,AWBURST\_CP;

ARLENGTH\_CP\_X\_ARSIZE\_X\_ARBURST :cross ARLEN\_CP,ARSIZE\_CP,ARBURST\_CP;

endgroup: axi\_bfm\_covergroup

**4. Test Plan and Test cases**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.,** | **Test Features** | **Test name** | **Description** |
| Write Transaction | | | |
| 1 | Unaligned single Write Transaction with data payload from 8bit to 32-bits with different strobe value from S,No.1-4 | fifo\_bfm\_8b\_wr\_incr\_unalligned\_test\_awlen\_0 | Write Transaction of Single Transaction . AWLEN=0 and WSTRB=4'b0001 |
| 2 |  | fifo\_bfm\_16b\_wr\_incr\_unalligned\_test\_awlen\_0.sv | Write Transaction of Single Transaction . AWLEN=0 and WSTRB=4'b0011 |
| 3 |  | fifo\_bfm\_24b\_wr\_incr\_unalligned\_test\_awlen\_0.sv | Write Transaction of Single Transaction . AWLEN=0 and WSTRB=4'b0111 |
| 4 | Aligned single Write Transaction | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_0.sv | Write Transaction of Single Transaction . AWLEN=0 and WSTRB=4'b1111 |
| 5 | Un-aligned burst Write Transaction with burst length = 1 from S.No.,5-6 | fifo\_bfm\_40b\_wr\_incr\_unalligned\_test\_awlen\_1.sv | Write Transaction of Single Transaction . AWLEN=1 WSTRB=4'b0001 |
| 6 |  | fifo\_bfm\_56b\_wr\_incr\_unalligned\_test\_awlen\_1.sv | Write Transaction of Single Transaction . AWLEN=1 WSTRB=4'b0111 |
| 7 | Aligned burst Write Transaction with burst length =2 | fifo\_bfm\_64b\_wr\_incr\_alligned\_test\_awlen\_1.sv | Write Transaction of Single Transaction . AWLEN=1 WSTRB=4'b1111 |
| 8 | Un-Aligned burst Write Transaction with burst length =3 | fifo\_bfm\_80b\_wr\_incr\_unalligned\_test\_awlen\_2.sv | Write Transaction of Single Transaction . AWLEN=2 WSTRB=4'b0011 |
| 9 | Aligned burst Write Transaction with burst length =3 | fifo\_bfm\_96b\_wr\_incr\_alligned\_test\_awlen\_2.sv | Write Transaction of Burst length is 3 . AWLEN=2 WSTRB=4'b1111 |
| 10 | Aligned burst Write Transaction with burst length =4 | fifo\_bfm\_128b\_wr\_incr\_alligned\_test\_awlen\_3.sv | Write Transaction of Burst length is 4 . AWLEN=3 WSTRB=4'b1111 |
| 11 | Aligned burst Write Transaction with burst length =6 | fifo\_bfm\_224b\_wr\_incr\_alligned\_test\_awlen\_6.sv | Write Transaction of Burst length is 7 . AWLEN=6 WSTRB=4'b1111 |
| 12 | Aligned burst Write Transaction with burst length =9 | fifo\_bfm\_320b\_wr\_incr\_alligned\_test\_awlen\_9.sv | Write Transaction of Burst length is 10 . AWLEN=9 WSTRB=4'b1111 |
| 13 | Aligned burst Write Transaction with burst length =14 | fifo\_bfm\_448b\_wr\_incr\_alligned\_test\_awlen\_13.sv | Write Transaction of Burst length is 14 . AWLEN=13 WSTRB=4'b1111 |
| 14 | Aligned burst Write Transaction with burst length =16 | fifo\_bfm\_512b\_wr\_incr\_alligned\_test\_awlen\_15.sv | Write Transaction of Burst length is 16 . AWLEN=15 WSTRB=4'b1111 |
| 15 | Random Write Burst Transaction with aligned address with different burst length (S.No.,15-18) | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_0\_to\_3.sv | Write Transaction of Single Transaction . AWLEN=Random ( 0 – 3) WSTRB=4'b1111 |
| 16 |  | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_4\_to\_7.sv | Write Transaction of Single Transaction . AWLEN=Random ( 4 – 7) WSTRB=4'b1111 |
| 17 |  | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_8\_to\_11.sv | Write Transaction of Single Transaction . AWLEN=Random ( 8 – 11) WSTRB=4'b1111 |
| 18 |  | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_12\_to\_15.sv | Write Transaction of Single Transaction . AWLEN=Random ( 12 – 15) WSTRB=4'b1111 |
| 19 | Random Write Burst Transaction with unaligned address with different burst length (S.No.,19-26) | fifo\_bfm\_32b\_wr\_incr\_unalligned\_test\_awlen\_0\_to\_3.sv | Write Transaction of Single Transaction . AWLEN=Random ( 0 – 3) WSTRB can be 4'b0001 or 0011 or 0111 |
| 20 |  | fifo\_bfm\_32b\_wr\_incr\_unalligned\_test\_awlen\_4\_to\_7.sv | Write Transaction of Single Transaction . AWLEN=Random ( 4 – 7) WSTRB can be 4'b0001 or 0011 or 0111 |
| 21 |  | fifo\_bfm\_32b\_wr\_incr\_unalligned\_test\_awlen\_8\_to\_11.sv | Write Transaction of Single Transaction . AWLEN=Random ( 8 – 11) WSTRB can be 4'b0001 or 0011 or 0111 |
| 22 |  | fifo\_bfm\_32b\_wr\_incr\_unalligned\_test\_awlen\_12\_to\_15.sv | Write Transaction of Single Transaction . AWLEN=Random ( 12 – 15) WSTRB can be 4'b0001 or 0011 or 0111 |
| 23 |  | fifo\_bfm\_wr\_incr\_unalligned\_test\_awlen\_6.sv | Write Transaction of Single Transaction . AWLEN=6 WSTRB can be 4'b0001 or 0011 or 0111 |
| 24 |  | fifo\_bfm\_wr\_incr\_unalligned\_test\_awlen\_9.sv | Write Transaction of Single Transaction . AWLEN=9 WSTRB can be 4'b0001 or 0011 or 0111 |
| 25 |  | fifo\_bfm\_wr\_incr\_unalligned\_test\_awlen\_13.sv | Write Transaction of Single Transaction . AWLEN=13 WSTRB can be 4'b0001 or 0011 or 0111 |
| 26 |  | fifo\_bfm\_wr\_incr\_unalligned\_test\_awlen\_15.sv | Write Transaction of Single Transaction . AWLEN=15 WSTRB can be 4'b0001 or 0011 or 0111 |
| 27 |  | fifo\_bfm\_32b\_rd\_test | Read Transaction of 32bit Data payload |
| 28 |  | fifo\_bfm\_64b\_rd\_test | Read Transaction of 64 bit Data payload |
| 29 |  | fifo\_bfm\_96b\_rd\_test | Read Transaction of 96bit Data payload |
| 30 |  | fifo\_bfm\_128b\_rd\_test | Read Transaction of 128bit Data payload |
| 31 |  | fifo\_bfm\_160b\_rd\_test | Read Transaction of 160bit Data payload |
| 32 |  | fifo\_bfm\_192b\_rd\_test | Read Transaction of 192bit Data payload |
| 33 |  | fifo\_bfm\_224b\_rd\_test | Read Transaction of 224bit Data payload |
| 34 |  | fifo\_bfm\_256b\_rd\_test | Read Transaction of 256bit Data payload |
| 35 |  | fifo\_bfm\_288b\_rd\_test | Read Transaction of 288bit Data payload |
| 36 |  | fifo\_bfm\_320b\_rd\_test | Read Transaction of 320bit Data payload |
| 37 |  | fifo\_bfm\_352b\_rd\_test | Read Transaction of 352bit Data payload |
| 38 |  | fifo\_bfm\_384b\_rd\_test | Read Transaction of 384bit Data payload |
| 39 |  | fifo\_bfm\_416b\_rd\_test | Read Transaction of 416bit Data payload |
| 40 |  | fifo\_bfm\_448b\_rd\_test | Read Transaction of 448bit Data payload |
| 41 |  | fifo\_bfm\_480b\_rd\_test | Read Transaction of 480bit Data payload |
| 42 |  | fifo\_bfm\_512b\_rd\_test | Read Transaction of 512bit Data payload |

**5. UserGuide**

**5.1 Git steps**

1. Checking for git, open the terminal type the command

*git version*

The output will either tell you which version of Git is installed or alert you that git is an unknown command. If it's an unknown command, install Git using following link [guide to install git in other platforms](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)

1. Copy the ssh public key or https and do the clone of the axi4\_avip repository in the terminal

<https://github.com/apriya-ram/AXI_FIFO_BFM/tree/phase1_development_branch>

*git clone https://github.com/apriya-ram/AXI\_FIFO\_BFM.git*

After cloning, change the directory to the cloned repository

*cd AXI\_BFM\_FIFO*

1. After cloning you will be in the main branch i.e, the production branch

*git branch*

1. Do the pull for the cloned repository to be in sync CD

*git pull origin main*

1. Fetch all branches in the AXI\_FIFO\_BFM repository

*git fetch*

1. Check all branches present in the AXI\_FIFO\_BFM repository

*git branch -a*

1. To switch from the main branch to another branch

*git checkout origin <branch\_name>*

1. Do the pull for the cloned repository to be in sync

*git pull origin <branch\_name>*

**Note:** To run any test case you should be inside the cloned directory i.e, AXI\_BFM\_FIFO [AXI\_BFM\_FIFO is considered as root path]

**5.2 Mentor’s Questasim**

1. Change the directory to questasim directory where the makefile is present

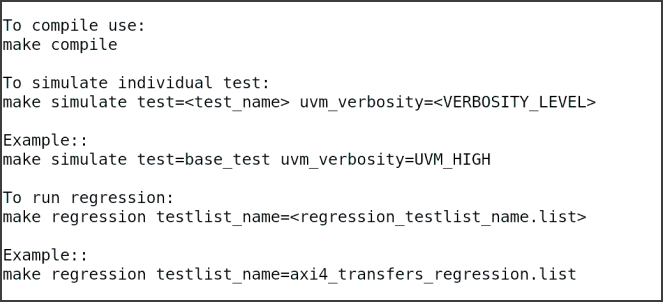
Path for the mentioned directory is AXI\_FIFO\_BFM/sim/questasim

Note: To Compile, simulate, regression and for coverage you must be in the specified path i.e, *AXI\_BFM\_FIFO/sim/questasim*

1. To view the usage for running test cases, type the command

*make*

Fig 7.1 shows the usage to compile, simulate, and regression



**Fig 5.1** Usage of the make command

**5.2.1 Compilation**

1. Use the following command to compile

*make compile*

1. Open the log file axi4\_bfm\_fifo\_compile.log to view the compiled files

*gvim axi4\_bfm\_fifo\_compile.log*

**5.2.2 Simulation**

1. After compilation, use the following command to simulate individual test cases

*make simulate test=<test\_name>uvm\_verbosity=<VERBOSITY\_LEVEL>*

Example:

**Note: You can find all the test case names in the path given below**

*AXI\_FIFO\_BFM/src/verif/tb/test/test\_pkg.sv*

1. To view the log file

*gvim<test\_name>/<test\_name>.log*

Ex:*gvimfifo\_bfm\_wr\_rd\_test/fifo\_bfm\_wr\_rd\_test.log*

Note: The path for the log file will be displayed in the simulation report along with the name of the simulated test

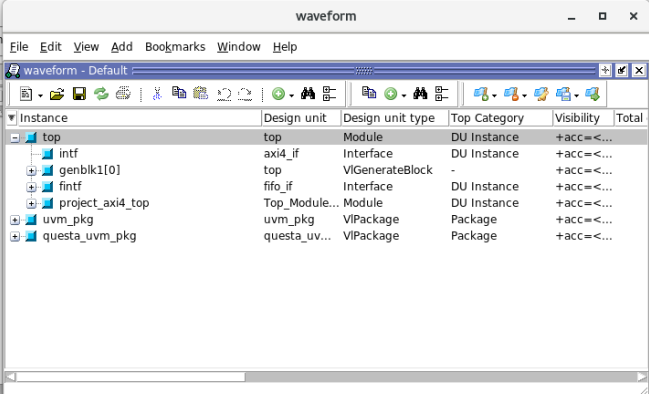
1. To view waveform

*vsim -view <test\_name>/waveform.wlf&*

Ex:*vsim -view fifo\_bfm\_wr\_rd\_test/waveform.wlf&*

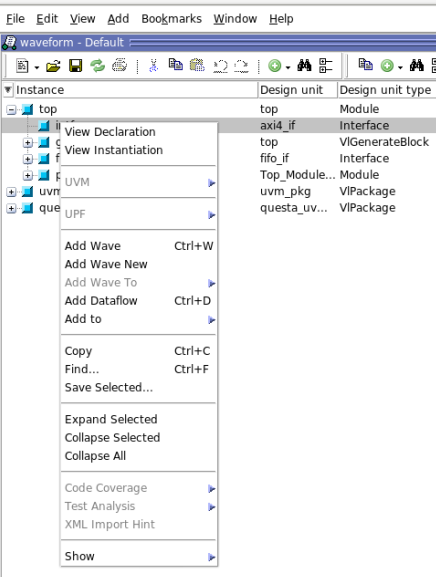
Note: The command to view the waveform will be displayed in the simulation report along with the name of the simulated test

As you run the above command, the new WLF Questasim window will appear as shown in fig 4.2



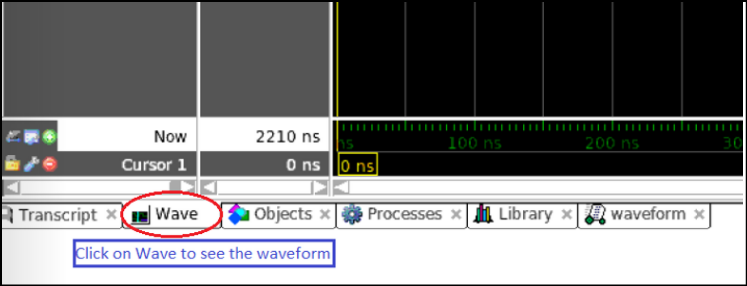
**Fig 5.2**Questasim WLF window

Right-click on intf and select Add Wave as shown in the image 4.3 to add the signals to the wave window



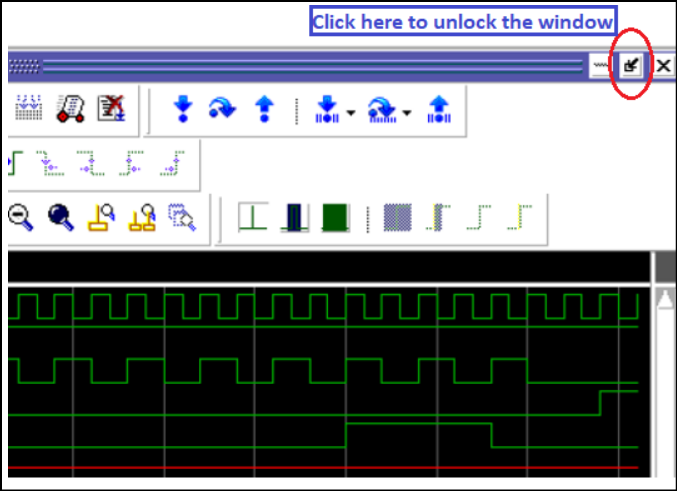
**Fig 7.3** Screenshot of adding waves in wave window

After adding wave, click on Wave window as shown in the Fig 4.3



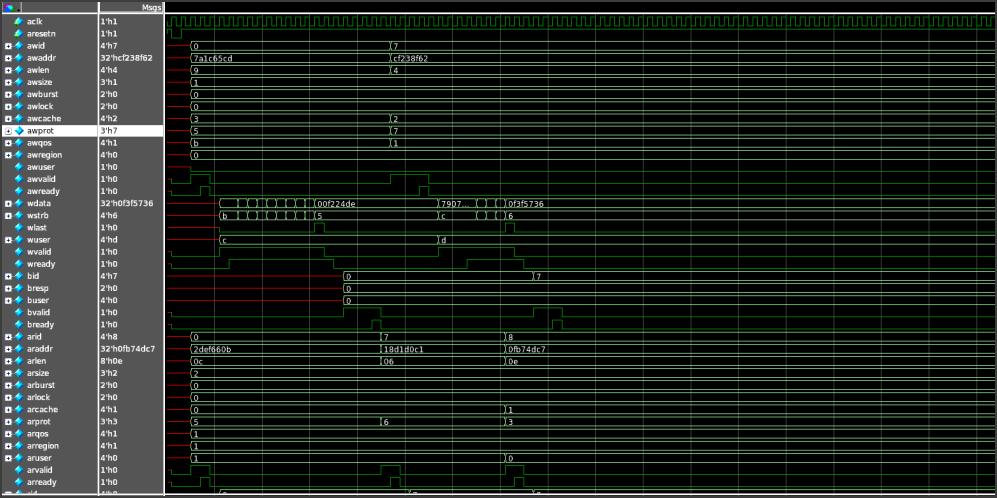
**Fig 4.4** Wave window

Click as shown in the fig 4.4 to unlock the waveform window



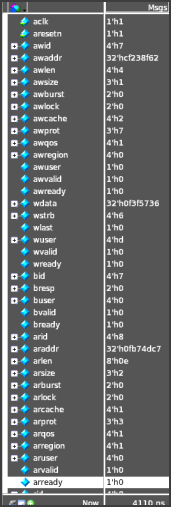
**Fig 7.5** Screenshot of unlocking the wave window

You will be able to get a separate wave window as shown in the Fig 4.5



**Fig 5.6** Screenshot of unlocked wave window with signals

Click on the icon signal toggle leaf name marked in fig 5.6 to see the signals as shown



**Fig 5.7** Screenshot showing way to see the name of signals

**5.2.3 Regression**

1. To run regression for all test case

*make regression testlist\_name=<regression\_testlist\_name.list>*

Ex:*make regression testlist\_name=axi4\_transfers\_regression.list*

**Note: You can find all the test case names in the path given below** *AXI\_FIFO\_BFM/src/hvl\_top/testlists/axi4\_transfers\_regression.list*

1. After regression, you can view the individual files
2. To view the log files of individual tests, select the interested test case file, go inside that directory

Ex:Interested in the test case *fifo\_bfm\_wr\_rd\_test*

Go inside the directory of interested testcase with the date

*fifo\_bfm\_wr\_rd\_test\_05022022-110914*

Inside this directory, you will be able to find the log file of the interested test case

*fifo\_bfm\_wr\_rd\_test.log*

Path: *fifo\_bfm\_wr\_rd\_test\_05022022-110914/fifo\_bfm\_wr\_rd\_test.log*

**5.2.4 Coverage**

1. To see coverage
   1. **After simulating**

For the individual test, use the command firefox

*firefoxfifo\_bfm\_wr\_rd\_test/html\_cov\_report/index.html&*

Ex: *firefoxfifo\_bfm\_wr\_rd\_test/html\_cov\_report/index.html &*

Note: The command to see the coverage will be displayed in the simulation report along with the name of the simulated test

* 1. **After the regression,**
* To view the coverages of all test cases, type the below command



*firefoxmerged\_cov\_html\_report/index.html &*

Note: The command to see the coverage will be displayed in the simulation report along with the name of the simulated test

* To view the coverage for individual test case

See the list of files generated after regression, which is shown in fig 4.7.

Select the interested test case file, go inside that directory

Ex:

Interested in the test case *fifo\_bfm\_wr\_rd\_test*

Go inside the directory of interested testcase with the date

*fifo\_bfm\_wr\_rd\_test\_05022022-110914*

Inside this directory, you will be able to find the html coverage file of the interested test case

*html\_cov\_report/*

Inside it would be the html file

*covsummary.html*

Command to view coverage report for the above test case will be

*firefox axi4\_blocking\_8b\_write\_read\_test\_05022022-110914/html\_cov\_report/covsummary.html*

1. **Simulation result and analysis**

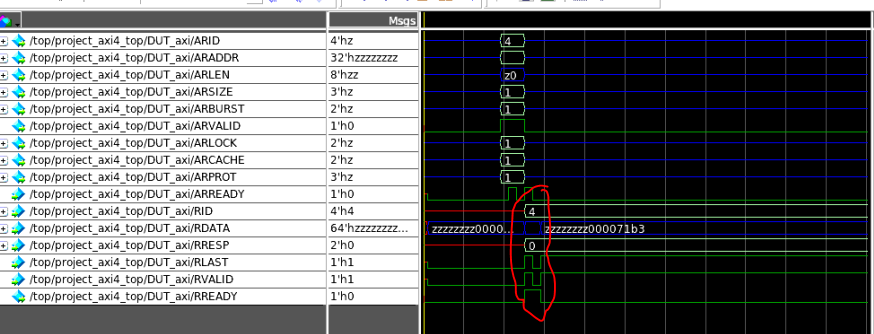
Developed test cases are totally 42. Out of which, 26/42 test cases are AXI Write Transaction and 16/42 test cases are AXI Read address Transaction.

**Regression Path**: /hwetools/work\_area/frontend/johnshahid\_B7/AXI\_FIFO\_BFM/sim/questasim

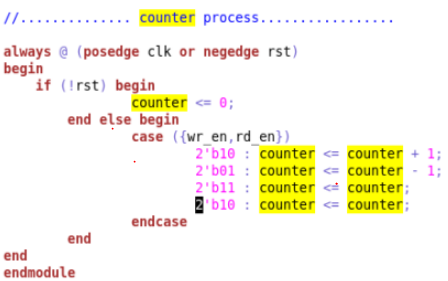
|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.,** | **Test name** | **Description** | **Status** |
| 1 | fifo\_bfm\_8b\_wr\_incr\_unalligned\_test\_awlen\_0 | Write Transaction of Single Transaction . AWLEN=0 and WSTRB=4'b0001 | FAIL |
| 2 | fifo\_bfm\_16b\_wr\_incr\_unalligned\_test\_awlen\_0.sv | Write Transaction of Single Transaction . AWLEN=0 and WSTRB=4'b0011 | FAIL |
| 3 | fifo\_bfm\_24b\_wr\_incr\_unalligned\_test\_awlen\_0.sv | Write Transaction of Single Transaction . AWLEN=0 and WSTRB=4'b0111 | FAIL |
| 4 | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_0.sv | Write Transaction of Single Transaction . AWLEN=0 and WSTRB=4'b1111 | FAIL |
| 5 | fifo\_bfm\_40b\_wr\_incr\_unalligned\_test\_awlen\_1.sv | Write Transaction of Single Transaction . AWLEN=1 WSTRB=4'b0001 | FAIL |
| 6 | fifo\_bfm\_56b\_wr\_incr\_unalligned\_test\_awlen\_1.sv | Write Transaction of Single Transaction . AWLEN=1 WSTRB=4'b0111 | FAIL |
| 7 | fifo\_bfm\_64b\_wr\_incr\_alligned\_test\_awlen\_1.sv | Write Transaction of Single Transaction . AWLEN=1 WSTRB=4'b1111 | FAIL |
| 8 | fifo\_bfm\_80b\_wr\_incr\_unalligned\_test\_awlen\_2.sv | Write Transaction of Single Transaction . AWLEN=2 WSTRB=4'b0011 | FAIL |
| 9 | fifo\_bfm\_96b\_wr\_incr\_alligned\_test\_awlen\_2.sv | Write Transaction of Burst length is 3 . AWLEN=2 WSTRB=4'b1111 | FAIL |
| 10 | fifo\_bfm\_128b\_wr\_incr\_alligned\_test\_awlen\_3.sv | Write Transaction of Burst length is 4 . AWLEN=3 WSTRB=4'b1111 | FAIL |
| 11 | fifo\_bfm\_224b\_wr\_incr\_alligned\_test\_awlen\_6.sv | Write Transaction of Burst length is 7 . AWLEN=6 WSTRB=4'b1111 | FAIL |
| 12 | fifo\_bfm\_320b\_wr\_incr\_alligned\_test\_awlen\_9.sv | Write Transaction of Burst length is 10 . AWLEN=9 WSTRB=4'b1111 | FAIL |
| 13 | fifo\_bfm\_448b\_wr\_incr\_alligned\_test\_awlen\_13.sv | Write Transaction of Burst length is 14 . AWLEN=13 WSTRB=4'b1111 | FAIL |
| 14 | fifo\_bfm\_512b\_wr\_incr\_alligned\_test\_awlen\_15.sv | Write Transaction of Burst length is 16 . AWLEN=15 WSTRB=4'b1111 | FAIL |
| 15 | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_0\_to\_3.sv | Write Transaction of Single Transaction . AWLEN=Random ( 0 – 3) WSTRB=4'b1111 | FAIL |
| 16 | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_4\_to\_7.sv | Write Transaction of Single Transaction . AWLEN=Random ( 4 – 7) WSTRB=4'b1111 | FAIL |
| 17 | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_8\_to\_11.sv | Write Transaction of Single Transaction . AWLEN=Random ( 8 – 11) WSTRB=4'b1111 | FAIL |
| 18 | fifo\_bfm\_32b\_wr\_incr\_alligned\_test\_awlen\_12\_to\_15.sv | Write Transaction of Single Transaction . AWLEN=Random ( 12 – 15) WSTRB=4'b1111 | FAIL |
| 19 | fifo\_bfm\_32b\_wr\_incr\_unalligned\_test\_awlen\_0\_to\_3.sv | Write Transaction of Single Transaction . AWLEN=Random ( 0 – 3) WSTRB can be 4'b0001 or 0011 or 0111 | FAIL |
| 20 | fifo\_bfm\_32b\_wr\_incr\_unalligned\_test\_awlen\_4\_to\_7.sv | Write Transaction of Single Transaction . AWLEN=Random ( 4 – 7) WSTRB can be 4'b0001 or 0011 or 0111 | FAIL |
| 21 | fifo\_bfm\_32b\_wr\_incr\_unalligned\_test\_awlen\_8\_to\_11.sv | Write Transaction of Single Transaction . AWLEN=Random ( 8 – 11) WSTRB can be 4'b0001 or 0011 or 0111 | FAIL |
| 22 | fifo\_bfm\_32b\_wr\_incr\_unalligned\_test\_awlen\_12\_to\_15.sv | Write Transaction of Single Transaction . AWLEN=Random ( 12 – 15) WSTRB can be 4'b0001 or 0011 or 0111 | FAIL |
| 23 | fifo\_bfm\_wr\_incr\_unalligned\_test\_awlen\_6.sv | Write Transaction of Single Transaction . AWLEN=6 WSTRB can be 4'b0001 or 0011 or 0111 | FAIL |
| 24 | fifo\_bfm\_wr\_incr\_unalligned\_test\_awlen\_9.sv | Write Transaction of Single Transaction . AWLEN=9 WSTRB can be 4'b0001 or 0011 or 0111 | FAIL |
| 25 | fifo\_bfm\_wr\_incr\_unalligned\_test\_awlen\_13.sv | Write Transaction of Single Transaction . AWLEN=13 WSTRB can be 4'b0001 or 0011 or 0111 | FAIL |
| 26 | fifo\_bfm\_wr\_incr\_unalligned\_test\_awlen\_15.sv | Write Transaction of Single Transaction . AWLEN=15 WSTRB can be 4'b0001 or 0011 or 0111 | FAIL |
| 27 | fifo\_bfm\_32b\_rd\_test | Read Transaction of 32bit Data payload | PASS |
| 28 | fifo\_bfm\_64b\_rd\_test | Read Transaction of 64 bit Data payload | PASS |
| 29 | fifo\_bfm\_96b\_rd\_test | Read Transaction of 96bit Data payload | PASS |
| 30 | fifo\_bfm\_128b\_rd\_test | Read Transaction of 128bit Data payload | PASS |
| 31 | fifo\_bfm\_160b\_rd\_test | Read Transaction of 160bit Data payload | PASS |
| 32 | fifo\_bfm\_192b\_rd\_test | Read Transaction of 192bit Data payload | PASS |
| 33 | fifo\_bfm\_224b\_rd\_test | Read Transaction of 224bit Data payload | PASS |
| 34 | fifo\_bfm\_256b\_rd\_test | Read Transaction of 256bit Data payload | PASS |
| 35 | fifo\_bfm\_288b\_rd\_test | Read Transaction of 288bit Data payload | PASS |
| 36 | fifo\_bfm\_320b\_rd\_test | Read Transaction of 320bit Data payload | PASS |
| 37 | fifo\_bfm\_352b\_rd\_test | Read Transaction of 352bit Data payload | PASS |
| 38 | fifo\_bfm\_384b\_rd\_test | Read Transaction of 384bit Data payload | PASS |
| 39 | fifo\_bfm\_416b\_rd\_test | Read Transaction of 416bit Data payload | PASS |
| 40 | fifo\_bfm\_448b\_rd\_test | Read Transaction of 448bit Data payload | PASS |
| 41 | fifo\_bfm\_480b\_rd\_test | Read Transaction of 480bit Data payload | PASS |
| 42 | fifo\_bfm\_512b\_rd\_test | Read Transaction of 512bit Data payload | PASS |

Found following RTL issues during the test case simulation and debug

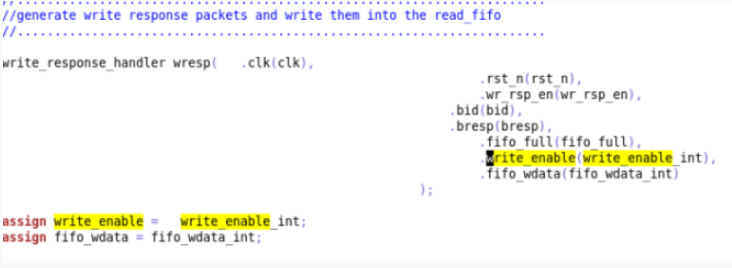
1. Write Data Transaction for awlen=0 not happening code(AXI\_MASTER\_WRITE\_CONTROL.v) there is no condition for single transfer (Fixed\_Burst)
2. AWCACHE Width (AWCACHE[2:0]) and ARCACHE Width (ARCACHE[2:0]) not matching with AXI Specification (AWCACHE[3:0] and ARCACHE[3:0]))
3. ARVALID is asserted before actual read address and control signals , Slave is considering address 0000000 as a transaction and sending response. As per Specification from Master RREADY should be asserted one cycle before RVALID. But in design both are asserting at same time . Due to ARVALID and RREADY slave is not generating proper RDATA and RLAST



1. READ FIFO Bug: empty condition is not getting low even when the wr\_en is 0 and wr\_data is received, empty is constantly high, Decoder needs to send “wr\_en”. As we are using the counter logic for wr\_en and rd\_en, So in the before snippet for case 2'b10 is repeating two times, it should be 2'b00.



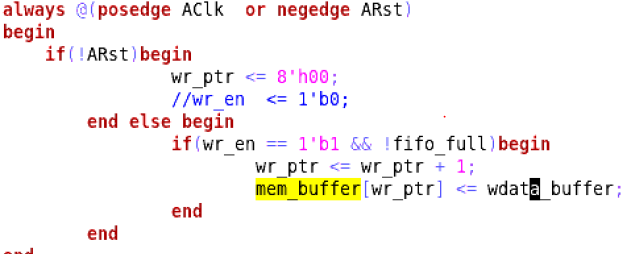
1. READ FIFO Bug: Read data is getting ‘x’ in read\_fifo because the output of decoder the “fifo\_wdata” is getting zero, As it doesn’t have the read response RTL only for Write response.



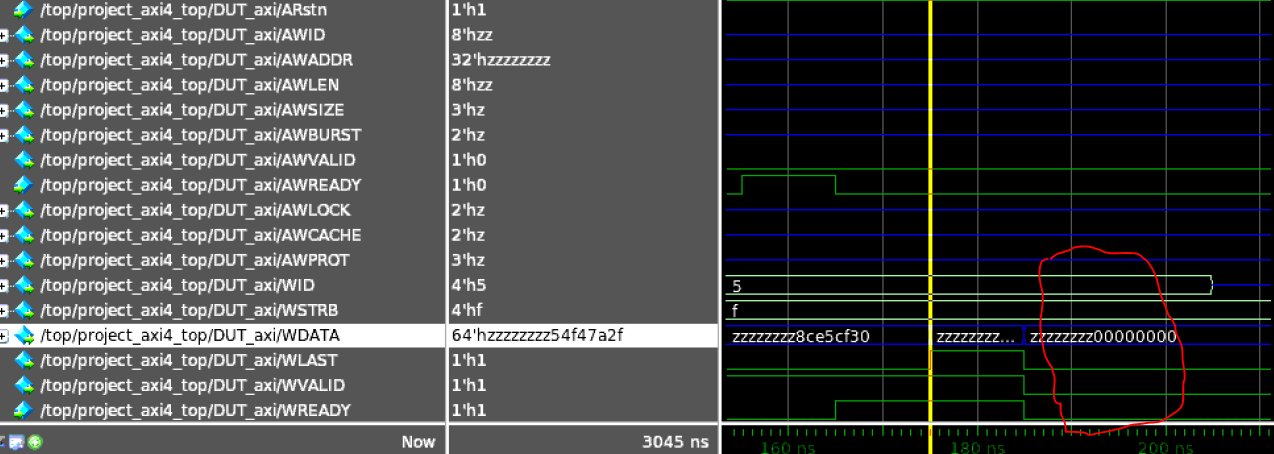
1. We are getting 1 data transaction than the expected data transaction for All Write Transaction.

For AWLEN=N (3) with Data payload (4\*32=128 bits). expected 4 data transactions we are getting 5 data transactions with WLAST . We are getting 1 cycle of Data payload Extra .

On debugging , found internal memory “mem\_buffer[0]” not updated with 32-bit data instead mem\_buffer[1] updated with 32-bit data. By default, mem\_buffer[0] updated with default 0000 values .



Waveform :



1. For Burst un-aligned transfer, Expected behavior of Transaction are first transfer should be un-aligned followed by the aligned transfer. But observing all the transaction are unaligned transfer which is in-correct

For Example:

