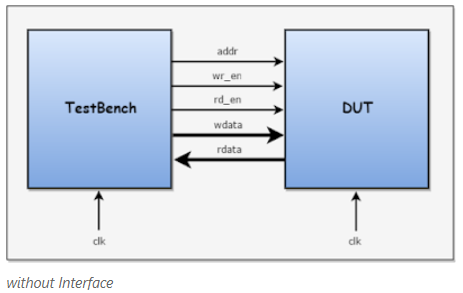
**FIFO-INTERFACE**

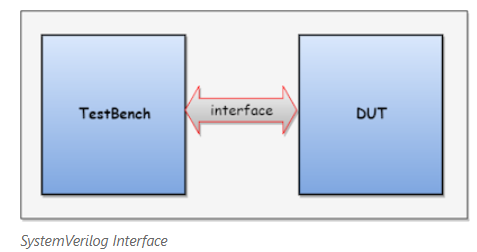
**INTERFACE:**The interface construct is used to connect the design and testbench.



TEST\_BENCH

FIFO

Above diagram shows connecting design and testbench ***without***interface.



TEST\_BENCH

FIFO

Above diagram shows connecting design and testbench ***with***the interface.

* An interface is a named bundle of wires, the interfaces aim is to encapsulate communication.
* Also specifies the,
  + directional information, i.e modports
  + timing information, i.e clocking blocks
* An interface can have parameters, constants, variables, functions, and tasks
* A simple interface declaration is,

**Syntax:**

interface interface name;

...

interface items

...

Clocking blocks

…

Mod-port blocks

Endinterface

**FIFO\_INTERFACE**

interface fifo\_if(input logic clk,rst);

logic wr\_rd;

logic full;

logic empty;

logic [7:0] D\_in;

logic [7:0] D\_out;

clocking driver\_cb@(posedge clk);

default input #1 output #1;

output wr\_rd;

output full;

output empty;

output D\_in;

input D\_out;

endclocking

clocking monitor\_cb@(posedge clk);

default input #1 output #1;

input wr\_rd;

input full;

input empty;

input D\_in;

input D\_out;

endclocking

modport DRIVER (clocking driver\_cb,input clk);

modport MONITOR (clocking monitor,input clk);

endinterface

**WRITE\_FIFO\_SEQ\_ITEM**

The sequence-item consist 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,

* Control Information  – a type of transfer, transfer size, etc
* Payload Information  –  data content of the transfer
* Configuration Information – mode of operation, error behavior, etc
* Analysis Information – fields used to capture information from DUT, ex: read data, response, etc

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:**

class write\_fifo\_seq\_item#(int ADDRESS\_WIDTH=32, DATA\_WIDTH=32,SIZE=3) extends uvm\_sequence\_item ;

// packet declaration

const logic [7:0] sop = 8’b01010101;

const logic [7:0] eop = 8’b10101011;

packet = [];

enum [2:0] type\_of\_axi = {axi\_fifo\_write\_address\_enable = 0, axi\_fifo\_write\_data\_enable = 1, axi\_fifo\_read\_address\_enable = 2, axi\_fifo\_read\_data\_enable = 3, axi\_fifo\_response\_enable = 4};

// Write Address Channel

rand bit [3: 0] awid ;

rand bit [ADDRESS\_WIDTH-1: 0] awaddr ;

rand bit [3: 0] awlen ;

rand bit [2: 0] awsize ;

rand bit [1: 0] awburst ;

rand bit [1: 0] awlock ;

rand bit [3: 0] awcache ;

rand bit [2: 0] awprot ;

rand bit [3:0] awqos ;

rand bit [3:0] awregion ;

logic awuser ;

logic awvalid ;

logic awready ;

//Write\_data\_channel

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

rand bit [(DATA\_WIDTH/8)-1: 0] wstrb ;

rand bit wlast ;

logic [3:0] wuser ;

logic wvalid ;

logic wready ;

//Write Response Channel

rand bit [3: 0] bid ;

rand bit [1: 0] bresp ;

rand bit [3: 0] buser ;

rand bit bvalid ;

rand bit bready ;

//Read Address Channel

rand bit [3: 0] arid ;

rand bit [ADDRESS\_WIDTH-1:0] araddr ;

rand bit [7:0] arlen ;

rand bit [2:0] arsize ;

rand bit [1:0] arburst ;

rand bit [1:0] arlock ;

rand bit [3:0] arcache ;

rand bit [2:0] arprot ;

rand bit [3:0] arqos ;

rand bit [3:0] arregion ;

rand bit [3:0] aruser

//Read Data Channel

rand bit [3: 0] rid ;

rand bit [DATA\_WIDTH-1: 0] rdata ;

rand bit [1:0] rresp ;

rand bit rlast ;

logic [3:0] ruser ;

logic rvalid ;

logic rready ;

logic arvalid ;

logic arready ;

//Read Data Channel

rand bit [3: 0] rid ;

rand bit [DATA\_WIDTH-1: 0] rdata ;

logic [1:0] rresp ;

logic rlast ;

logic [3:0] ruser ;

logic rvalid ;

logic rready ;

`uvm\_object\_param\_utils\_begin(write\_fifo\_seq\_item#(ADDRESS\_WIDTH, DATA\_WIDTH,SIZE))

`uvm\_field\_int(awid,UVM\_ALL\_ON)

`uvm\_field\_int(awaddr,UVM\_ALL\_ON)

`uvm\_field\_int(awlen,UVM\_ALL\_ON)

`uvm\_field\_int(awsize,UVM\_ALL\_ON)

`uvm\_field\_int(awburst,UVM\_ALL\_ON)

`uvm\_field\_int(awlock,UVM\_ALL\_ON)

`uvm\_field\_int(awcache,UVM\_ALL\_ON)

`uvm\_field\_int(awprot,UVM\_ALL\_ON)

`uvm\_field\_int(awqos,UVM\_ALL\_ON)

`uvm\_field\_int(awregion,UVM\_ALL\_ON)

`uvm\_field\_array\_int(wdata,UVM\_ALL\_ON)

`uvm\_field\_array\_int(wstrb,UVM\_ALL\_ON)

`uvm\_field\_int(wlast,UVM\_ALL\_ON)

`uvm\_field\_int(wuser,UVM\_ALL\_ON)

`uvm\_field\_int(wvalid,UVM\_ALL\_ON)

`uvm\_field\_int(wready,UVM\_ALL\_ON)

`uvm\_field\_int(bid,UVM\_ALL\_ON)

`uvm\_field\_int(bresp,UVM\_ALL\_ON)

`uvm\_field\_int(buser,UVM\_ALL\_ON)

`uvm\_field\_int(bvalid,UVM\_ALL\_ON)

`uvm\_field\_int(bready,UVM\_ALL\_ON)

`uvm\_field\_int(arid,UVM\_ALL\_ON)

`uvm\_field\_int(araddr,UVM\_ALL\_ON)

`uvm\_field\_int(arlen,UVM\_ALL\_ON)

`uvm\_field\_int(arsize,UVM\_ALL\_ON)

`uvm\_field\_int(arburst,UVM\_ALL\_ON)

`uvm\_field\_int(arlock,UVM\_ALL\_ON)

`uvm\_field\_int(arcache,UVM\_ALL\_ON)

`uvm\_field\_int(arprot,UVM\_ALL\_ON)

`uvm\_field\_int(arqos,UVM\_ALL\_ON)

`uvm\_field\_int(arregion,UVM\_ALL\_ON)

`uvm\_field\_int(aruser,UVM\_ALL\_ON)

`uvm\_field\_int(arvalid,UVM\_ALL\_ON)

`uvm\_field\_int(arready,UVM\_ALL\_ON)

`uvm\_field\_int(rid,UVM\_ALL\_ON)

`uvm\_field\_array\_int(rdata,UVM\_ALL\_ON)

`uvm\_field\_int(rresp,UVM\_ALL\_ON)

`uvm\_field\_int(rlast,UVM\_ALL\_ON)

`uvm\_field\_int(ruser,UVM\_ALL\_ON)

`uvm\_field\_int(rvalid,UVM\_ALL\_ON)

`uvm\_field\_int(rready,UVM\_ALL\_ON)

`uvm\_object\_utils\_end

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

super.new(name);

endfunction

endclass

**WRITE\_FIFO\_SEQUENCE**

A sequence generates a series of sequence\_item’s and sends it to the driver via sequencer, Sequence is written by extending the *uvm\_sequence*.



UVM Sequence

* A uvm\_sequence is derived from an uvm\_sequence\_item
* A sequence is parameterized with the type of write\_fifo\_sequence\_item, this defines the type of the item sequence that will send/receive to/from the driver.

**WRITE\_FIFO\_SEQUENCE Code:**

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

`uvm\_object\_utils(write\_fifo\_sequence)

write\_fifo\_seq\_item pkt;

function new (string name="write\_fifo\_sequence");

super.new (name);

endfunction

task body();

repeat(6) begin

pkt= write\_fifo\_seq\_item::type\_id::create("pkt");

start\_item(pkt);

assert( pkt.randomize() );

pkt.display("SEQUENCE");

finish\_item(pkt);

end

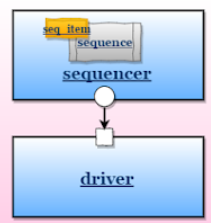
endtask

endclass

**WRITE\_FIFO\_SEQUENCER**

* The sequencer controls the flow of request and response sequence items between 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.



**WRITE\_FIFO\_SEQUENCER CODE**

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

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

function new(string name , uvm\_component parent);

super.new(name,parent);

endfunction

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

super.build\_phase(phase);

endfunction

endclass

**WRITE\_FIFO\_DRIVER**

* A driver is written by extending the uvm\_driver
* uvm\_driver is inherited from uvm\_component, Methods and TLM port (seq\_item\_port) are defined for communication between sequencer and driver
* The uvm\_driver is a parameterized class and it is parameterized with the type of the request sequence\_item and the type of the response sequence\_item

**UVM\_Driver Methods**

### **get\_next\_item**

This method blocks until a REQ write\_fifo\_sequence\_item is available in the sequencer.

### **try\_next\_item**

This is a non-blocking variant of the get\_next\_item() method. It will return a null pointer if there is no REQ write\_fifo\_sequence\_item available in the sequencer.

### **item\_done**

The non-blocking item\_done() method completes the driver-sequencer handshake and it should be called after a get\_next\_item() or a successful try\_next\_item() call.

### **put**

The put() method is non-blocking and is used to place an RSP write\_fifo\_sequence\_item in the sequencer.

**WRITE\_FIFO\_DRIVER CODE**

class fifo\_driver extends uvm\_driver#(fifo\_seq\_item);

`uvm\_component\_utils(fifo\_driver)

virtual fifo\_if intf;

write\_fifo\_seq\_item pkt;

queue0 = [$];

queue1 = [$];

queue2 = [$];

queue3 = [$];

queue4 = [$];

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

super.new(name, parent);

endfunction

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

super.build\_phase(phase);

uvm\_config\_db#(virtual fifo\_if)::get(this,"","vif",intf);

endfunction

task reset();

wait(!intf.rst);

intf.data\_in<=0;

intf.wr<=0;

intf.rd<=0;

wait(intf.rst);

endtask

task run\_phase(uvm\_phase phase);

reset();

forever begin

pkt=write\_fifo\_seq\_item::type\_id::create("pkt");

seq\_item\_port.get\_next\_item(pkt);

drive(pkt);

seq\_item\_port.item\_done();

pkt.display("DRIVER");

end

endtask

task drive(write\_fifo\_seq\_item pkt);

@(posedge intf.clk);

intf.wr<=pkt.wr;

intf.rd<=pkt.rd;

//// intf.data\_in<=pkt.data\_in;

// Write Address Channel

if(pkt.type\_of\_axi == 0){

packet = {pkt.sop, pkt.type\_of\_axi, pkt.awid, pkt.awlen, pkt.awsize, pkt.awburst, pkt.awaddr , pkt.eop};

queue0.push\_back(pkt.awaddr);

}

// Write Data Channel

if(pkt.type\_of\_axi == 1){

packet = {pkt.sop, pkt.type\_of\_axi, pkt.wid, pkt.wstrb, pkt.wdata, pkt.wlast, pkt.eop};

queue1.push\_back(pkt.wdata);

intf.data\_in <= [31:0] queue1;

queue1.popfront();

}

// Read Address Channel

if(pkt.type\_of\_axi == 2){

packet = {pkt.sop, pkt.type\_of\_axi, pkt.arid, pkt.arlen, pkt.arsize, pkt,arburst, pkt.araddr, pkt.eop};

queue2.push\_back(pkt.araddr);

}

// Read Data Channel

if(pkt.type\_of\_axi == 3){

packet = {pkt.sop, pkt.type\_of\_axi, pkt.rid, pkt.rresp, pkt.rlast, pkt.rdata , pkt.eop};

queue3.push\_back(pkt.rdata);

}

// Write Response Channel

if(pkt.type\_of\_axi == 4){

packet = {pkt.sop, pkt.type\_of\_axi, pkt.bid, pkt.bresp, pkt.eop};

queue4.push\_back(pkt.bresp);

}

endtask

endclass

**WRITE\_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.

**Writing Monitor :**

**STEPS**

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,

**WRITE\_FIFO\_MONITOR CODE:**

class write\_fifo\_monitor extends uvm\_monitor;

`uvm\_component\_utils(write\_fifo\_monitor);

write\_fifo\_seq\_item pkt;

virtual fifo\_if intf;

uvm\_analysis\_port #(write\_fifo\_seq\_item)item\_collected\_port;

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

super.new(name,parent);

item\_collected\_port=new("item\_collected\_port",this);

endfunction

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

super.build\_phase(phase);

uvm\_config\_db#(virtual fifo\_if)::get(this,"","vif",intf);

endfunction

task run\_phase(uvm\_phase phase);

pkt=write\_fifo\_seq\_item::type\_id::create("pkt");

@(posedge intf.clk);

forever begin

@(posedge intf.clk);

pkt.wr<=intf.wr;

pkt.rd<=intf.rd;

pkt.data\_in<=intf.data\_in;

item\_collected\_port.write(pkt);

pkt.display("MONITOR");

end

endtask

endclass