DESIGN AND VERIFICATION OF FLIGHT DATA

ACQUISITION SYSTEM USING UVM

, , , ,

*1.Final year students, ECE dept, Dayananda Sagar College of Engineering, Karnataka, India*

*2.Professor, ECE dept, Dayananda Sagar College of Engineering,Karnataka,India*

***Abstract -*** *Avionics is an aviation industry platform that provides comprehensive solutions for flight planning, scheduling, and management. Flight data acquisition (FDA) in Avionics refers to the process of collecting and gathering flight-related information from various sources within the Avionics system. It involves the integration of data from multiple sources, such as aircraft sensors, air traffic control systems, weather information providers, and airline databases. These diverse sources contribute different types of data, including flight parameters, environmental conditions, and operational details. Overall, flight data acquisition in Avionics plays a crucial role in ensuring the availability of accurate and reliable flight-related information. It enables efficient flight planning, improves operational decision-making, and enhances safety and efficiency in the aviation industry.*

***Keywords -*** *Flight data acquisition (FDA), Avionics, FPGA, UVM (Universal Verification Methodology), EDA (Electronic Design Automation)*

# **INTRODUCTION**

Flight data acquisition using Field-Programmable Gate Arrays (FPGAs) and the Universal Verification Methodology (UVM) is an advanced approach to collecting and verifying flight-related data in the aviation industry. FPGAs offer programmable hardware that can be customized to perform specific tasks efficiently, while UVM provides a standardized methodology for verifying digital designs. FPGAs can handle high-speed data processing and offer flexibility in terms of interfacing with various sensors, communication protocols, and data storage units. By leveraging the capabilities of FPGAs, flight data can be acquired, processed, and transmitted in real time, enabling efficient monitoring and analysis of critical flight parameters.

The UVM verification methodology is employed to ensure the correctness and reliability of the FPGA-based flight data acquisition system. UVM is a standardized verification framework that provides a systematic and scalable approach to verifying digital designs. It involves the creation of reusable verification components, the development of test benches, and the application of constrained-random stimulus generation techniques to thoroughly test the functionality of the design. When applied to flight data acquisition using FPGAs, UVM allows for the creation of comprehensive test environments that simulate various flight scenarios and sensor inputs. The verification components and testbenches are developed to mimic the behavior of the real-world flight data acquisition system and its associated interfaces. By subjecting the FPGA design to a wide range of test scenarios, potential issues or bugs can be identified and rectified, ensuring the system's robustness and reliability.

# **RELATED WORK**

The verification process is typically iterative, involving multiple rounds of simulation, formal verification, emulation, and testing until the hardware design is correct and ready for fabrication or development. The complexity of the hardware design and the level of verification required depends on the size, complexity, and criticality of the system being developed.

UVM remains a widely used and supported methodology for hardware verification, providing a standardized and interoperable framework for developing reusable and scalable test benches. Its adoption has significantly improved verification productivity and efficiency, enabling more effective verification of complex designs in the semiconductor industry.

Ning Jia, Hang Chen, and Jun Tian in the year 2021 proposed a model on “A Design of Configurable Multi-type Flight Data Acquisition System”[1]. The objective of this paper is to compute the acquisition task of multiple types of flight data and store it in real-time. The main drawback of this paper is the complexity in the design framework and requires a lot of time and effort by professional technicians to operate.

G.V.Jayaramaiah and Chetan Umadi in the year 2020 have published papers on “FPGA implementation of multi-protocol data acquisition system using VHDL”[6]. They have implemented a parallel and serial data transfer protocols and all protocols implanted on FPGA kit and modelled using VHDL. The digital signals are provided from multichannel sensors and different ADC protocols.

Harikrishnan. K, Vishwas. H.N, Vineetha Jain K.V, and Dr.Ramesh Chinthala in the year 2020 published a paper on “Sensor data acquisition and de-noising using FPGA”[4]. They have designed and implemented an FPGA-based DAQ (Data Acquisition) system. The design shows how an entire system can be implemented in a single FPGA fabric.

# **PROPOSED WORK**

# ***Flight data acquisition design***

FPGA

FIFO

Real

world data

(Temp,speed)

SPI

ADC

Memory RAM

**Fig1 : Flight Data Acquisition System**

Flight data acquisition design aims to ensure accurate and reliable data capture, considering factors such as sensor accuracy, sampling rates, synchronization, data integrity, and system redundancy as shown in the fig1. The collected data is crucial for flight testing, aircraft certification, performance analysis, accident investigations, and ongoing aircraft health monitoring.

The design of a flight data acquisition system involves selecting appropriate sensors to measure the desired parameters, designing the wiring and data buses to transmit the signals, determining the sampling rate and resolution of the data, and specifying the storage capacity and data retrieval methods. The system may utilize data recorders, data acquisition units, signal conditioning modules, and communication interfaces.

The system consists of ADC, interface signal module, FPGA data acquisition module, Memory storage. This system mainly comprises sensor, which collects the data from the external world. The configurable aircraft data acquisition system mainly collects engine sensor parameters, fuselage sensor parameters, atmosphere and other flight data. Analogue to Digital Converter or ADC, is a data converter which allows digital circuits to interface with the real world by converting an analogue signal into a binary code. To process the analog signal onto digital devices like as FPGA it should be first converted to digital format. After the signal conversion, data is handled using FPGA .

The Serial Peripheral Interface (SPI) is a full-duplex, synchronous, serial data link that is standard across many microprocessors, microcontrollers. It enables communication between microprocessors and peripherals. The SPI protocol is flexible enough to interface with numerous peripherals. Therefore, the main function of the FPGA data acquisition module is to collect many kinds of signals. FPGA, as the core of the flight data acquisition system, collects and stores the data. This system is divided into three steps: the signal processing module, FPGA data acquisition module, and data storage module.

The signal processing module in a flight data acquisition system is responsible for receiving, conditioning, and processing various sensor signals and data streams from different aircraft systems. It plays a crucial role in converting analog signals into digital format, applying necessary filters and transformations, and preparing the data for storage or transmission.

The data acquisition module interfaces with a wide range of sensors and data sources throughout the aircraft. These may include airspeed sensors, altimeters, gyroscopes, accelerometers, engine parameters, control surfaces, avionics systems, and more. The signals from these sources are acquired and converted into digital form for further processing.

The FDR (Flight Data Recorder) which is commonly known as the "black box," is a specialized device installed on aircraft to record various parameters and flight information. It captures data such as altitude, airspeed, vertical acceleration, heading, control inputs, engine parameters, and more. The FDR typically uses solid-state memory or magnetic tape to store data, and it is designed to withstand extreme conditions, including crashes and fires, to ensure data survivability. Flight data acquisition systems are crucial for flight safety and performance monitoring. The collected data can be analyzed to identify trends, anomalies, and potential issues, enabling proactive maintenance, incident investigation.

1. ***Verification Plan***

Verification plan is done before verifying any project to ease the work of verification engineer. Based on the requirement, verification plan is to be built which includes list of test cases and coverage models. A verification plan defines what needs to be verified in a hardware design and then drives the verification strategy.

Design Specification

Verification Plan(UVM)

Build Test Environment

Insert RTL

Run Test

No

Test case Verified

Yes

Design Verified

**Fig2 : Flow Chart**

***C. Universal verification methodology***

Universal Verification Methodology (UVM) is a standardized methodology for functional verification of digital designs or systems. It provides a framework and set of guidelines for creating modular and reusable testbenches and verification components. UVM is widely adopted in the semiconductor industry and is supported by various electronic design automation (EDA) tools.

In the Universal Verification Methodology (UVM), there are several key components that work together to create a modular and reusable testbench environment. These components include:

Testbench Top: This is the main component of the testbench hierarchy. It coordinates the overall verification process and instantiates other testbench components.

Testbench Configuration: The testbench configuration specifies the desired configuration parameters for the testbench, such as clock frequency, interface configurations, or specific feature enablement.

DUT (Design Under Test) Agent: The DUT agent is responsible for interfacing with the design and translating the testbench transactions to the DUT's signals or interfaces. It contains the necessary drivers, monitors, and sequences for communication with the DUT.

Test: The test component defines a specific test scenario or use case that needs to be verified. It encapsulates the test sequence and may contain test-specific data and configuration information. The test component typically inherits from the uvm\_test base class.

Top

Test

Env

Scoreboard

Agent

Sequencer

Monitor

Interface

DUT

Driver

**Fig3 : UVM Architecture**

Environment: The environment component acts as the top-level container for the testbench. It provides the infrastructure for coordinating and controlling the verification process. The environment instantiates and connects other components, such as agents, monitors, drivers, and scoreboards. It usually inherits from the uvm\_env base class.

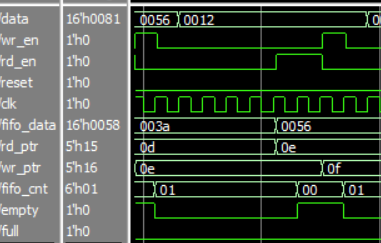
Agent: An agent represents a specific interface or protocol within the design under test (DUT). It consists of multiple sub-components, each with a specific role:

Sequencer: The sequencer generates sequences of transactions or stimuli to be applied to the DUT. It controls the flow and timing of the transactions and manages sequence dependencies.

Driver: The driver receives the transactions from the sequencer and drives the stimuli onto the DUT's interface signals. It converts the transaction-level protocol into the signal-level protocol.

Monitor: The monitor component observes the DUT's interface signals, captures transaction-level information and converts it into transactions for analysis and checking in the testbench.

Scoreboard: The scoreboard compares the expected results, generated by the test or reference model, with the actual results obtained from the DUT. It checks for functional correctness and reports any discrepancies.

****Sequences: Sequences represent a sequence of transactions or stimuli that are applied to the DUT. Sequences are typically generated by the sequencer and control the specific test scenario.

Sequence Items: Sequence items are the individual transactions or stimuli that make up a sequence. They encapsulate the data and control information to be applied to the DUT.

Configuration: The configuration component manages the configuration settings for the testbench and DUT. It provides a centralized location for storing and accessing configuration information.

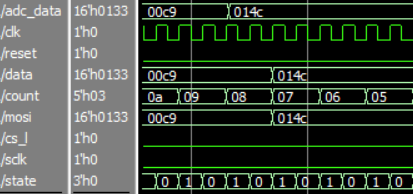
Coverage: The coverage component tracks which parts of the design have been exercised by the testbench. It collects coverage data and generates coverage analysis reports, helping to ensure that the verification process is thorough.

Assertions: Assertions are used to define specific properties or conditions that the DUT must satisfy. They help in capturing and verifying design properties and can be used for both design and testbench verification.

These components work together to create a comprehensive UVM testbench environment for verifying digital designs. They provide a modular and scalable infrastructure that promotes reusability, maintainability, and efficient verification of the design under test.

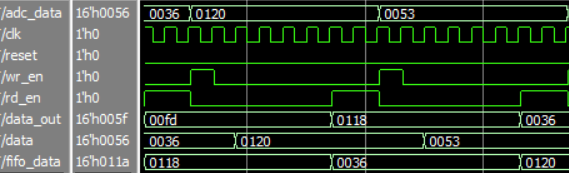
# **RESULTS**

Flight Data Acquisition System is designed in which random inputs are given and stored in memory. The design includes SPI protocol as Interface and FIFO to store data in the memory. The design act as DUT (Design under test) which is verified using UVM (Universal Verification Methodology). Each block of the UVM is designed, which includes: UVM Interface, Sequence, Sequence Item, Sequencer, Driver, Monitor, Agent, Scoreboard, Environment, Test.

Here are the simulation results of flight data acquisition system which is verified using UVM.

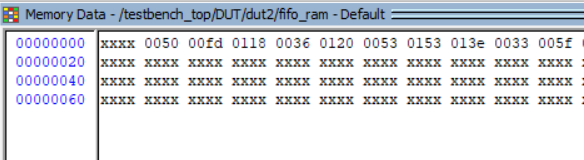
**Fig4 : SPI Interface**

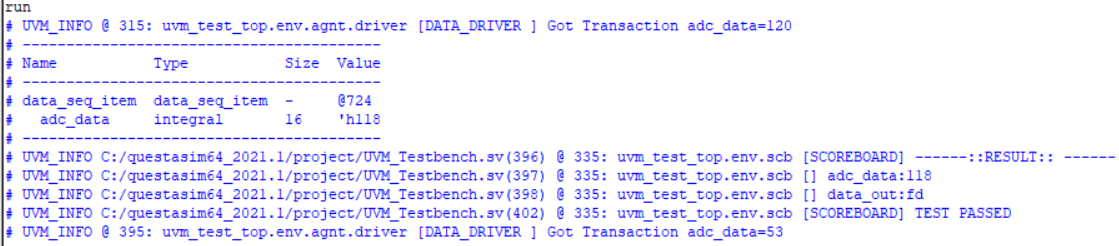
The data from sensor is interfaced with FPGA using SPI interface and obtained results are shown in fig4.

**Fig5 : FIFO**

**Fig6: Verification using UVM**

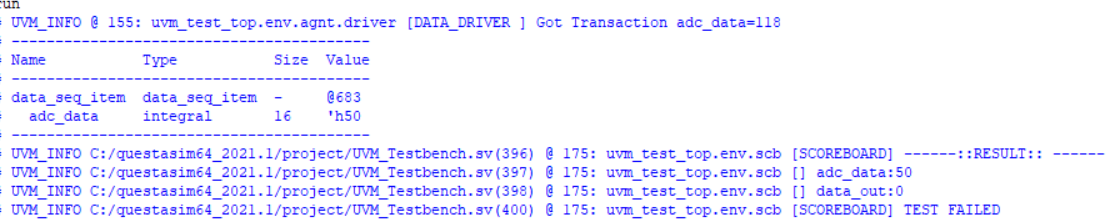
Flight data acquisition system is verified using UVM testbench and observed waveform is as shown above in fig6.

**Fig 7: Memory Block**

The data processed using FPGA is stored in memory block , the above fig shows the data stored in memory during simulation.

**Fig8: Log report when the test is passed**

When the test case is verified it prints as test passed in transcript window which is shown in fig 8.

****

**Fig9: Log file output when the test is failed**

When the test case is not verified it prints as test failed in transcript window which is shown in fig 9.

# **CONCLUSIONS**

The Flight Data Acquisition System (FDAS) plays a critical role in aviation safety and performance monitoring. It is an essential component of an aircraft's avionics system, responsible for collecting and recording various flight parameters and operational data during the entire duration of a flight. The FDAS captures information such as altitude, airspeed, engine parameters, flight control inputs, and numerous other variables that provide valuable insights into aircraft operations.The FDAS is designed to meet stringent safety and reliability standards, ensuring accurate and real-time data acquisition under various flight conditions

# **REFERENCES**

1. Ning Jia, Hang Chen and Jun Tian “A Design of Configurable Multi-type Flight Data Acquisition System”, IEEE conference on Communication and Computing,2021
2. Guojin Peng, Wei Zhang, Manting Liu “A method of Data Acquisition Network Delay Measurement for AFDX Avionics System in Flight test”,7th international conference on signal and image processing, 2022
3. John W. Dyer and Benjamin Douglas ”Portable airborne Data Acquisition for Flight Testes” , “IEEE international instrumentation and measurements Technology Conference Proceedings”,2019
4. G.V.Jayaramaiah and Chetan.Umadi “Fpga implementation of multiprotocol data acquisition system using vhdl”July-2020
5. Ufuk Sakarya and Ibrahim Hokelek “Universal Verification Methodology Application of ARINC429 for Airborne Electronic Hardware Certification”,2021.
6. Harikrishnan.K, Vishwas.H.N, Vineetha Jain K.V, Dr.Ramesh Chinthala, “ Sensor data acquisition and de-noising using fpga”, 2020.
7. J. AN, S. LI, et al. "Design of Architecture of Spaceflight Test Data Cen ters Using Cloud Platform," Journal of Spacecraft TT&C Technology, v ol. 35 pp.: 137-145, 2016.
8. D. Laney, 3D data management: controlling data volume, velocity, and variety, META Group Research Note, 2019.
9. B. Zhao, Design and research of engine parameter collector system[D]. Xi'an, Northwestern Polytechnical University, 2018.
10. J. Zhang, G. Peng, B. Liu, W. Zhang, Hardware design of data acquisition system for a certain aircraft engine. Modern Electronics Technique," Xi'an, vol. 37,pp. 67-69, November 2014.
11. J. AN, S. LI, et al. "Design of Architecture of Spaceflight Test Data Cen ters Using Cloud Platform," Journal of Spacecraft TT&C Technology, v ol. 35 pp.: 137-145, 2016.
12. D. Laney, 3D data management: controlling data volume, velocity, and variety, META Group Research Note, 2001.
13. B. Zhao, Design and research of engine parameter collector system[D].Xi'an, Northwestern Polytechnical University, 2004.
14. J. Ma, C. Zhou. Data acquisition and processing technology. Xi'an: Xi'an Jiaotong University Press, 2001.
15. J. Zhang, G. Peng, B. Liu, W. Zhang, Hardware design of data acquisition system for a certain aircraft engine. Modern Electronics Technique," Xi'an, vol. 37,pp. 67-69, November 2014.
16. Christophersen B H, Pickell W J, et al. Small Adaptive Flight Control Systems for UAVs using FPGA/DSP Technology. Georgia Institute of Technology, 2004.
17. Ye Fan “FPGA-Based Data Acquisition System” 2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC).
18. Shi-zhen Huang and Rui-Qi Chen “FPGA- based IoT Sensor HUB” 2018 International Conference on Sensor Networks and Signal Processing (SNSP).
19. Swarup S. Mathurkar , Rahul B Lanjewar, Nilesh Patel and Rohit S.Somkuvwar “Smart Sensor Based Monitoring System For Agriculture Using Field Programmable Gate Array” 2014 International Conference on Circuit, Power and Computing Technologies [ICCPCT]
20. Shuang Bao, Hairong Yan , Qingping Chi , Zhibo Pang and Yuyin Sun “FPGA-Based Reconfigurable Data Acquisition System ForIndustrial Sensors” IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, VOL. 13, NO. 4, AUGUST 2017.

***Pseudo Code***

***Design under Test:***

module data\_acq(adc\_data,clk,reset,wr\_en,rd\_en,data\_out);

input [15:0] adc\_data;

input clk,reset,wr\_en,rd\_en;

output reg[15:0] data\_out;

reg [15:0] data,fifo\_data;

reg wr\_en,rd\_en;

spi dut1(.adc\_data(adc\_data),.clk(clk),.reset(reset),.data(data));

fifo dut2(.data(data),.wr\_en(wr\_en),.rd\_en(rd\_en),.clk(clk),.reset(reset),.fifo\_data(fifo\_data));

ram dut3(.fifo\_data(fifo\_data),.clk(clk),.reset(reset),.wr\_en(wr\_en),.rd\_en(rd\_en),.data\_out(data\_out));

endmodule

module spi(adc\_data,clk,reset,data);

input [15:0] adc\_data;

input clk,reset;

output reg [15:0] data;

reg [4:0] count;

reg [15:0] mosi;

reg cs\_l;

reg sclk;

reg [2:0] state;

always@(posedge clk)

if(reset)

begin

count<=5'd16;

cs\_l<=1'b1;

sclk<=1'b0;

end

else

begin

case(state)

0:begin

sclk<=1'b0;

cs\_l<=1'b0;

mosi<=adc\_data;

count<=count-1;

state<=1;

end

1:begin

sclk<=1'b0;

if(count>0)

state<=0;

else

begin

count<=16;

state<=0;

end

end

default:state<=0;

endcase

end

assign data=mosi;

endmodule

module fifo(data,wr\_en,rd\_en,clk,reset,fifo\_data);

input [15:0] data;

input wr\_en,rd\_en,reset,clk;

output reg [15:0] fifo\_data;

reg [15:0] data;

reg [4:0] rd\_ptr, wr\_ptr;

reg [5:0]fifo\_cnt;

reg [15:0] fifo\_ram[128];

reg empty,full;

always@(data,wr\_en,rd\_en)

begin

if(wr\_en==1'b1) //write into RAM

begin

fifo\_ram[wr\_ptr]=data;

end

else if (rd\_en==1'b1) // read from RAM

begin

fifo\_data=fifo\_ram[rd\_ptr];

end

else

begin

fifo\_data=fifo\_data;

end

end

always @( posedge clk )

begin: counter

if( reset )

fifo\_cnt <= 0;

else

begin

case ({wr\_en,rd\_en})

2'b00 : fifo\_cnt <= fifo\_cnt;

2'b01 : fifo\_cnt <= (fifo\_cnt==0) ? 0 : fifo\_cnt-1;

2'b10 : fifo\_cnt <= (fifo\_cnt==32) ? 32 : fifo\_cnt+1;

2'b11 : fifo\_cnt <= fifo\_cnt;

default: fifo\_cnt <= fifo\_cnt;

endcase

end

end

endmodule

module ram(fifo\_data,clk,reset,wr\_en,rd\_en,data\_out);

input [15:0] fifo\_data;

input clk,reset,wr\_en,rd\_en;

output reg [15:0] data\_out;

reg [4:0] wr\_ptr\_m,rd\_ptr\_m;

reg [15:0] mem[(2\*\*7):0];

always@(posedge clk )

begin

if(reset==1'b1)

begin

for(int unsigned i=0;i<2\*\*8;i++)

begin

mem[i]<=0;

data\_out<=0;

end

end

end

always@(fifo\_data,wr\_en,rd\_en)

begin

if(wr\_en==1'b1) //write into RAM

begin

mem[wr\_ptr\_m]=fifo\_data;

end

else if (rd\_en==1'b1) // read from RAM

begin

data\_out=mem[rd\_ptr\_m];

end

else

begin

data\_out=data\_out;

end

end

endmodule

***UVM Verification:***

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

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

// data\_interface

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

interface data\_if(input logic clk,reset);

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

//declaring the signals

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

logic wr\_en;

logic rd\_en;

logic [15:0] adc\_data;

logic [15:0] data\_out;

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

//driver clocking block

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

clocking driver\_cb @(posedge clk);

default input #1 output #1;

endclocking

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

//monitor clocking block

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

clocking monitor\_cb @(posedge clk);

default input #1 output #1;

endclocking

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

//driver modport

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

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

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

//monitor modport

modport MONITOR (clocking monitor\_cb,input clk,reset);

endinterface

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

// sequence item

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

class data\_seq\_item extends uvm\_sequence\_item;

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

//data and control fields

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

bit wr\_en;

bit rd\_en;

rand bit [15:0] adc\_data;

bit [15:0] data\_out;

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

//Utility and Field macros

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

`uvm\_object\_utils\_begin(data\_seq\_item)

`uvm\_field\_int(adc\_data,UVM\_ALL\_ON)

`uvm\_object\_utils\_end

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

//Constructor

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

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

super.new(name);

endfunction

endclass

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

// data\_sequence - random stimulus

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

class data\_sequence extends uvm\_sequence#(data\_seq\_item);

`uvm\_object\_utils(data\_sequence)

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

//Constructor

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

function new(string name = "data\_sequence");

super.new(name);

endfunction

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

// create, randomize and send the item to driver

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

task body();

endtask

endclass

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

// sequencer

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

class data\_sequencer extends uvm\_sequencer#(data\_seq\_item);

`uvm\_component\_utils(data\_sequencer)

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

//constructor

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

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

// driver

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

`define DRIV\_IF vif.DRIVER.driver\_cb

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

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

// Virtual Interface

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

virtual data\_if vif;

`uvm\_component\_utils(data\_driver)

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

// Constructor

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

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

super.new(name, parent);

endfunction : new

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

// build phase

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

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

super.build\_phase(phase);

if(!uvm\_config\_db#(virtual data\_if)::get(this, "", "vif", vif))

begin

`uvm\_error("build\_phase","driver virtual interface failed");

end

endfunction: build\_phase

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

// run phase

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

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

super.run\_phase(phase);

forever begin

data\_seq\_item trans;

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

uvm\_report\_info("DATA\_DRIVER ", $psprintf("Got Transaction %s",trans.convert2string()));

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

//Reading

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

@(posedge vif.DRIVER.clk);

trans.data\_out=`DRIV\_IF.data\_out;

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

end

endtask : run\_phase

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

// drive - transaction level to signal level

// drives the value's from seq\_item to interface signals

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

endclass : data\_driver

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

// monitor

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

`define MON\_IF vif.MONITOR.monitor\_cb

class data\_monitor extends uvm\_monitor;

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

// Virtual Interface

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

virtual data\_if vif;

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

// analysis port, to send the transaction to scoreboard

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

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

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

// The following property holds the transaction information currently

// begin captured (by the collect\_address\_phase and data\_phase methods).

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

`uvm\_component\_utils(data\_monitor)

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

// new - constructor

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

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

super.new(name, parent);

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

endfunction

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

// build\_phase - getting the interface handle

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

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

super.build\_phase(phase);

if(!uvm\_config\_db#(virtual data\_if)::get(this, "", "vif", vif))

`uvm\_error("build\_phase", "No virtual interface specified for this monitor instance")

endfunction: build\_phase

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

// run\_phase - convert the signal level activity to transaction level.

// i.e, sample the values on interface signal ans assigns to transaction class fields

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

endclass : data\_monitor

class fun\_cov extends uvm\_subscriber#(data\_seq\_item);

`uvm\_component\_utils(fun\_cov)

data\_seq\_item trans;

covergroup cg;

WDATA:coverpoint trans.wr\_en { bins wd[16] = {[0:2\*16-1]}; }

RDATA:coverpoint trans.rd\_en { bins rd[16] = {[0:2\*16-1]}; }

endgroup

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

super.new(name, parent);

cg = new();

endfunction //new()

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

trans = data\_seq\_item::type\_id::create("trans"); endfunction

function void write(data\_seq\_item t);

this.trans = t;

cg.sample();

endfunction

endclass

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

// agent

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

class data\_agent extends uvm\_agent;

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

// component instances

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

data\_driver driver;

data\_sequencer sequencer;

data\_monitor monitor;

virtual data\_if vif;

`uvm\_component\_utils\_begin(data\_agent)

`uvm\_field\_object(sequencer, UVM\_ALL\_ON)

`uvm\_field\_object(driver, UVM\_ALL\_ON)

`uvm\_field\_object(monitor, UVM\_ALL\_ON)

`uvm\_component\_utils\_end

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

// constructor

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

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

super.new(name, parent);

endfunction : new

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

// build\_phase

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

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

super.build\_phase(phase)

monitor = data\_monitor::type\_id::create("monitor", this);

//creating driver and sequencer only for ACTIVE agent

driver = data\_driver::type\_id::create("driver", this);

sequencer = data\_sequencer::type\_id::create("sequencer", this);

uvm\_config\_db#(virtual data\_if)::set(this, "seq", "vif", vif);

uvm\_config\_db#(virtual data\_if)::set(this, "driv", "vif", vif);

uvm\_config\_db#(virtual data\_if)::set(this, "mon", "vif", vif);

if(!uvm\_config\_db#(virtual data\_if)::get(this,"","vif",vif))

begin

`uvm\_error("build\_phase","agent virtual interface failed");

end

endfunction : build\_phase

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

// connect\_phase - connecting the driver and sequencer port

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

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

super.connect\_phase(phase);

driver.seq\_item\_port.connect(sequencer.seq\_item\_export);

uvm\_report\_info("DATA\_AGENT", "connect\_phase, Connected driver to sequencer");

endfunction : connect\_phase

endclass : data\_agent

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

// scoreboard

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

class data\_scoreboard extends uvm\_scoreboard;

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

//port to recive packets from monitor

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

uvm\_analysis\_imp#(data\_seq\_item, data\_scoreboard) item\_collected\_export;

`uvm\_component\_utils(data\_scoreboard)

data\_seq\_item trans;

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

// new - constructor

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

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

super.new(name, parent);

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

endfunction

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

// build\_phase - create port and initialize local memory

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

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

super.build\_phase(phase);

endfunction: build\_phase

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

// write task - recives the pkt from monitor and pushes into queue

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

endclass : data\_scoreboard

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

// environment

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

class data\_env extends uvm\_env;

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

// agent and scoreboard instance

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

data\_agent agnt;

data\_scoreboard scb;

virtual data\_if vif;

`uvm\_component\_utils(data\_env)

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

// constructor

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

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

super.new(name, parent);

endfunction : new

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

// build\_phase - crate the components

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

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

super.build\_phase(phase);

agnt = data\_agent::type\_id::create("agnt", this);

scb = data\_scoreboard::type\_id::create("scb", this);

uvm\_config\_db#(virtual data\_if)::set(this, "agt", "vif", vif);

uvm\_config\_db#(virtual data\_if)::set(this, "scb", "vif", vif);

if(! uvm\_config\_db#(virtual data\_if)::get(this, "", "vif", vif))

begin

`uvm\_error("build\_phase","Environment virtual interface failed")

end

endfunction : build\_phase

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

// connect\_phase - connecting monitor and scoreboard port

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

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

super.connect\_phase(phase);

agnt.monitor.item\_collected\_port.connect(scb.item\_collected\_export);

uvm\_report\_info("data\_ENVIRONMENT", "connect\_phase, Connected monitor to scoreboard");

endfunction : connect\_phase

endclass : data\_env

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

// test

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

class data\_test extends uvm\_test;

`uvm\_component\_utils(data\_test)

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

// env instance

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

data\_env env;

virtual data\_if vif;

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

// constructor

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

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

super.new(name,parent);

endfunction : new

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

// build\_phase

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

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

super.build\_phase(phase);

// Create the env

env = data\_env::type\_id::create("env", this);

uvm\_config\_db#(virtual data\_if)::set(this, "env", "vif", vif);

if(! uvm\_config\_db#(virtual data\_if)::get(this, "", "vif", vif))

begin

`uvm\_error("build\_phase","Test virtual interface failed")

end

endfunction : build\_phase

task run\_phase(uvm\_phase phase);

data\_sequence seq;

seq = data\_sequence::type\_id::create("seq",this);

phase.raise\_objection(this,"starting main phase");

$display("%t Starting sequence spi\_seq run\_phase",$time);

seq.start(env.agnt.sequencer);

#500ns;

phase.drop\_objection(this,"finished main phase");

endtask : run\_phase

endclass

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

// testbench.sv

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

module testbench\_top;

always #5 clk = ~clk;

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

//reset Generation

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

initial begin

reset = 1;

#15 reset =0;

end

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

//interface instance

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

data\_if intf(clk,reset);

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

//DUT instance

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

data\_acq DUT (

.clk(intf.clk),

.reset(intf.reset),

.wr\_en(intf.wr\_en),

.rd\_en(intf.rd\_en),

.adc\_data(intf.adc\_data),

.data\_out(intf.data\_out)

);

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

//passing the interface handle to lower heirarchy using set method

//and enabling the wave dump

initial begin

uvm\_config\_db#(virtual data\_if)::set(uvm\_root::get(),"\*","vif",intf);

$dumpfile("dump.vcd");

$dumpvars;

end

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

//calling test

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

initial begin

run\_test("data\_test");

end

endmodule