SSRG International Journal of VLSI and Signal Processing
ISSN: 2394 – 2584 / https://doi.org/10.14445/23942584/IJVSP-V10I2P101

Volume 10 Issue 2, 1-11, May-Aug 2023

Original Article

Design and Verification of Flight Data Acquisition System using UVM

Nainshree Raj[1], G. Pallavi[1], M. Poornima[1], S. Lakshmi[1], S. Jamuna[2]

[1,2]*Department of ECE, Dayananda Sagar College of Engineering, Karnataka, India*

Received: 15 April 2023 Revised: 22 May 2023 Accepted: 05 June 2023 Published: 16 June 2023

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 integrating 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, Universal Verification Methodology (UVM), Electronic Design Automation (EDA).*

1. 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 test the functionality of the design thoroughly. When applied to flight data acquisition using FPGAs, UVM allows for creation of comprehensive test environments that simulate various flight scenarios and sensor inputs. The verification components and testbenches are developed to mimic the real-world flight data acquisition system's behaviour and associated interfaces. Subjecting the FPGA design to a wide range of test scenarios can identify and rectify potential issues or bugs, ensuring the system's robustness and reliability.

2. 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 in "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 of the design framework, which

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 multiprotocol data acquisition system using VHDL" [6]. They have implemented parallel and serial data transfer protocols and all protocols implanted on FPGA kit and modelled using VHDL. Multichannel sensors and different ADC protocols provide digital signals.

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 a single FPGA fabric can implement an entire system.

3. Proposed Work

3.1. Flight Data Acquisition Design

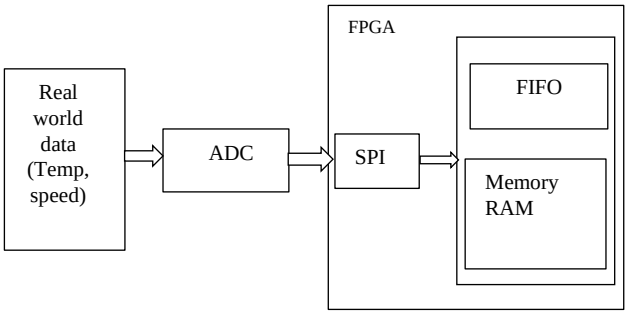

Fig 1. Flight data acquisition system

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, and memory storage. This system mainly comprises sensors, which collect 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 that 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 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 and 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. As the core of the flight data acquisition system, FPGA collects and stores the data. This system is divided into three steps: the signal processing module, the FPGA data acquisition module, and the 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), 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 and incident investigation.

3.2. Verification Plan

A verification plan is done before verifying any project to ease the work of the verification engineer. Based on the requirement, a verification plan is to be built, including a 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.

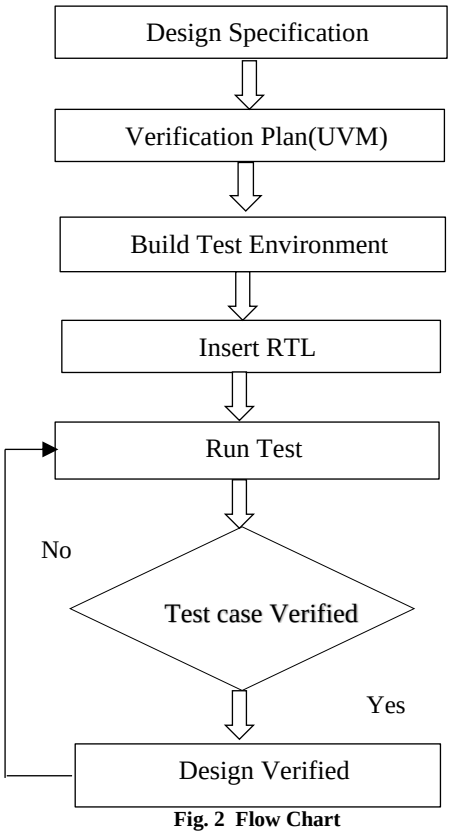

Fig. 2 Flow Chart

3.3. Universal Verification Methodology

Universal Verification Methodology (UVM) is a standardized methodology for the 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), several key components work together to create a modular and reusable testbench environment. These components include:

3.3.1. Testbench Top

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

3.3.2. Testbench Configuration

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

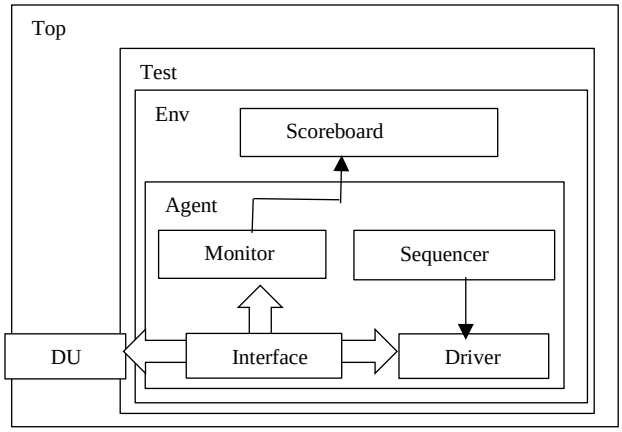

Fig. 3 UVM Architecture

3.3.3. 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.

3.3.4. 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.

3.3.5. 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.

3.3.6. 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 capture and verify design properties and can be used for 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.

4. 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 acts as DUT (Design under test), 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, and Test.

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

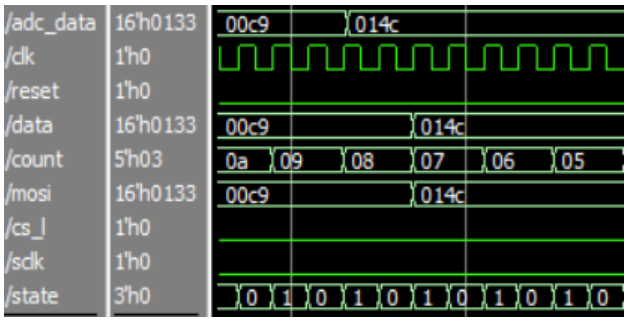

Fig. 4 SPI Interface

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

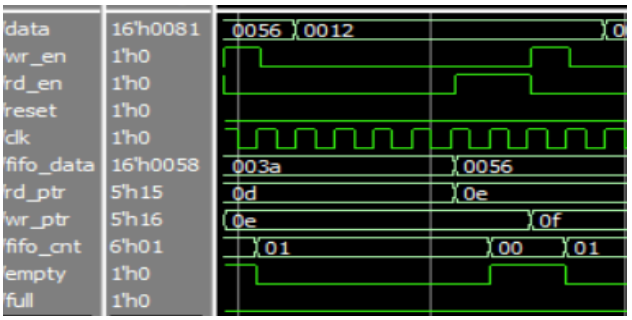

Fig. 5 FIFO

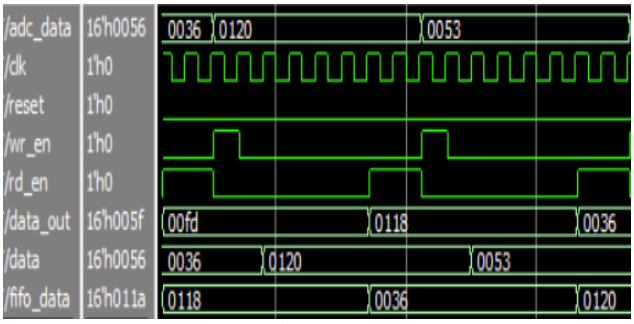

Fig. 6 Verification using UVM

The flight data acquisition system is verified using UVM testbench and observed waveform is shown above in fig. 6.

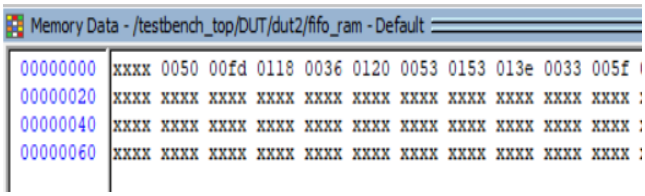

Fig. 7 Memory Block

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

```
@ 335: uvm_test_top.env.scb [SCOREBOARD] ------::RESULT:: ------
@ 335: uvm_test_top.env.scb [] adc_data:118
@ 335: uvm_test_top.env.scb [] data_out:fd
@ 335: uvm_test_top.env.scb [SCOREBOARD] TEST PASSED
Got Transaction adc_data=53
```

Fig. 8 Log report when the test is passed

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

```
@ 175: uvm_test_top.env.scb [SCOREBOARD] ------::RESULT::
@ 175: uvm_test_top.env.scb [] adc_data:50
@ 175: uvm_test_top.env.scb [] data_out:0
@ 175: uvm_test_top.env.scb [SCOREBOARD] TEST FAILED
```

Fig. 9 Log file output when the test is failed

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

5. Conclusion

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 International Conference on Signal Processing, Communication and Computing*, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[2] Guojin Peng, Wei Zhang, and 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. [CrossRef] [Google Scholar] [Publisher Link]

[3] John W. Dyer et al., "Portable airborne Data Acquisition for Flight Testes," *IEEE International Instrumentation and Measurements Technology Conference Proceedings*, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[4] G.V. Jayaramaiah, and Chetan. Umadi, "FPGA Implementation of Multiprotocol Data Acquisition System using VHDL," *International Journal of Research in Engineering and Technology*, vol. 3, no. 7, pp. 390-394, 2020. [Google Scholar] [Publisher Link]

[5] Ufuk Sakarya, Ibrahim Ayaz, and Ibrahim Hokelek, "Universal Verification Methodology Application of ARINC429 for Airborne Electronic Hardware Certification," *13th International Conference on Electrical and Electronics Engineering*, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[6] K. Harikrishnan et al., "Sensor Data Acquisition and De-noising Using FPGA," *International Journal of Scientific and Engineering Research*, vol. 11, no. 8, 2020. [Publisher Link]

[7] J. An, and S. Li, "Design of Architecture of Spaceflight Test Data Cen ters Using Cloud Platform," *Journal of Spacecraft TT&C Technology*, vol. 35, pp. 137-145, 2016. [Google Scholar]

[8] Doug Laney, "3D Data Management: Controlling Data Volume, Velocity, and Variety," *META Group Research Note*, VOL. 6, NO. 70, 2001. [Google Scholar]

[9] B. Zhao, Design and research of engine parameter collector system [D]. Xi'an, Northwestern Polytechnical University, 2018.

[10] J. Zhang et al., "Hardware Design of Data Acquisition System for a Certain Aircraft Engine, Modern Electronics Technique," Xi'an, vol. 37, pp. 67-69, 2014.

[11] Akwaowo U. Ekpa, Aniekan E. Eyoh, and Okon Ubom, "A Comparative Analysis of Volumetric Stockpile from UAV Photogrammetry and Total Station Data," *SSRG International Journal of Geoinformatics and Geological Science*, vol. 6, no. 2, pp. 29-37, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[12] M.K. Rachana, T.V. Sindhu, and Della Reasa Valiyaveetil, "Automatic Land surveillance System by Sketching Robot," *SSRG International Journal of Electronics and Communication Engineering*, vol. 3, no. 7, pp. 8-13, 2016. [CrossRef] [Publisher Link]

[13] J. Zhang et al., "Hardware Design of Data Acquisition System for a Certain Aircraft Engine. Modern Electronics Technique," Xi'an, vol. 37, pp. 67-69, 2014.

[14] Henrik Christophersen et al., "Small Adaptive Flight Control Systems for UAVs using FPGA/DSP Technology," *Georgia Institute of Technology*, 2004. [CrossRef] [Google Scholar] [Publisher Link]

[15] J. Vignesh et al., "Digital Fuel Level Indicator for Motor Bikes using Arduino Microcontroller," *SSRG International Journal of Electronics and Communication Engineering*, vol. 4, no. 3, pp. 13-16, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[16] Ye Fan, "FPGA-Based Data Acquisition System," *2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[17] Shi-zhen Huang, and Rui-Qi Chen, "FPGA- based IoT Sensor HUB," *2018 International Conference on Sensor Networks and Signal Processing (SNSP)*, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[18] Swarup S. Mathurkar et al., "Smart Sensor Based Monitoring System For Agriculture Using Field Programmable Gate Array," *2014 International Conference on Circuit, Power and Computing Technologies*, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[19] Shuang Bao et al., "FPGA-Based Reconfigurable Data Acquisition System for Industrial Sensors," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 4, pp. 1503-1512, 2017. [CrossRef] [Google Scholar] [Publisher Link]

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 the 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 receive packets from the 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 the monitor and pushes
into the 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
```