****



**ALU Verification Plan**

| **Date** | **Document Version** | **Remarks** | **Drafted by** |
| --- | --- | --- | --- |
| 31- July - 2024 | Version 1.0 | Design Overview | TEAM 5 |
| 03- Aug- 2024 | Version 2.0 | Test plan and verification plan | TEAM 5 |
| 13- Aug- 2024 | Version 3.0 | Testbench architecture and Testbench components | TEAM 5 |
|  | Version 4.0 | Testbench implementations |  |
|  | Version 5.0 | Simulation results and analysis |  |

**VERIFICATION DOCUMENT- ALU design**

Table of Contents

[CHAPTER 1 – DESIGN OVERVIEW 4](#_30j0zll)

[1.1 ARITHMETIC AND LOGIC UNIT 4](#_1fob9te)

[1.2 KEY FEATURES 4](#_3znysh7)

[1.3 ADVANTAGES OF ALU 4](#_p34406vbf1gs)

[1.4 DISADVANTAGES OF ALU 4](#_2et92p0)

[1.5 PROJECT OVERVIEW 5](#_3dy6vkm)

[1.6 DESIGN LIMITATIONS 5](#_4d34og8)

[1.7 DESIGN DIAGRAM WITH INTERFACE SIGNALS 5](#_2s8eyo1)

[1.9 PIN LEVEL DESCRIPTION 6](#_17dp8vu)

[CHAPTER 2 – VERIFICATION ARCHITECTURE 9](#_3rdcrjn)

[2.1 VERIFICATION ARCHITECTURE 9](#_26in1rg)

[2.3 FLOWCHART OF UVM COMPONENTS 9](#_35nkun2)

[2.3.1 alu\_top 9](#_1ksv4uv)

[2.3.2 alu\_test 10](#_44sinio)

[2.3.3 alu\_environment 12](#_2jxsxqh)

[2.3.4 alu\_agent 12](#_z337ya)

[2.3.5 alu\_interface 13](#_3j2qqm3)

[2.3.6 alu\_sequence\_item 14](#_1y810tw)

[2.3.7 alu\_sequence 15](#_4i7ojhp)

[2.3.8 alu\_sequencer 16](#_2xcytpi)

[2.3.9 alu\_monitor 17](#_1ci93xb)

[2.3.10 alu\_driver 18](#_3whwml4)

[2.3.11 alu\_scoreboard 19](#_2bn6wsx)

[2.3.12 alu\_coverage 20](#_qsh70q)

# **CHAPTER 1 – DESIGN OVERVIEW**

## 1.1 ARITHMETIC AND LOGIC UNIT

An Arithmetic Logic Unit (ALU) is a digital circuit used to perform arithmetic (e.g., addition, subtraction) and logical (e.g., AND, OR, NOT) operations. It is a combinational circuit. It is a critical component within the CPU of a computer. It is responsible for performing a variety of arithmetic and logical operations on binary numbers.

## 1.2 KEY FEATURES

* Arithmetic Operations: Signed and unsigned addition and subtraction.
* Logical Operations: NOT, AND, OR, XOR.
* Shift Operations: Rotate and shift right/left.
* Control Signals: Mode signal to switch between arithmetic and logical operations.
* Command Signals: 4-bit command input to specify the operation.

## 

## 1.3 ADVANTAGES OF ALU

* **Efficiency:** ALUs are essential for performing arithmetic and logical operations quickly, which is crucial for the overall speed of the CPU.
* **Flexibility:** They support a variety of operations such as addition, subtraction, logical AND, OR XOR, and more.
* **Compact Design**: Integrating multiple operations in a single unit saves space and reduces the complexity of the CPU design

## 1.4 DISADVANTAGES OF ALU

* **Complexity:** Designing a robust ALU can be complex, especially when ensuring it handles all edge cases correctly.
* **Power Consumption:** ALUs consume a significant amount of power, particularly when performing complex operations.
* **Speed Limitations:** The speed of an ALU is limited by the technology used to implement it, which can become a bottleneck in high-performance systems.

## 

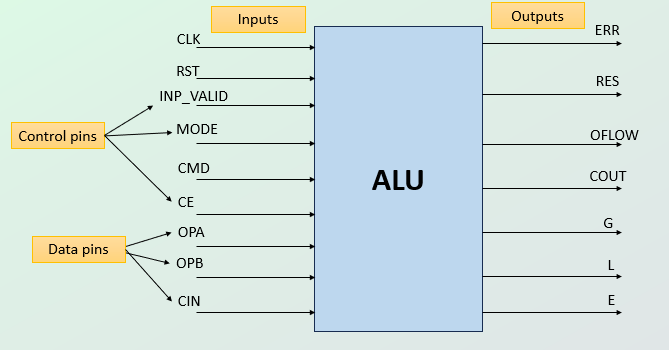
## 1.5 PROJECT OVERVIEW

* This is an ALU design with parameterised functionality.
* The input operands (OPA and OPB) are parameterised data buses. It also has a 1-bit carrier input (CIN) signal.
* The ALU is sensitive to the active high reset (RST) signal and requires an active high clock enable (CE) signal to function and produce results.
* All changes in the ALU occur on the clock signal's positive edge.
* The ALU uses a 1-bit MODE signal to choose between arithmetic and logical operations. When the MODE signal is high, the ALU performs arithmetic operations on the input operands; otherwise (MODE is low), it performs logical operations on them.
* Different operations (both arithmetic and logical) can be selected based on values driven by the parameterised (4-bit by default) CMD bus.
* The design also includes a 2-bit INP\_VALID bus, which indicates which of the input operands are valid. The MSB and LSB correspond to OPB and OPA, respectively. If the bit is set, the corresponding value is valid.
* The ALU design includes 1-bit flags that change state under predefined conditions. When the output exceeds its maximum capacity, the OFLOW (overflow) flag is set. When a carry bit is generated, it sets the COUT (carry output) flag. The G, L, and E flags are activated during comparison operations. G is set when OPA exceeds OPB, L when OPA is less than OPB, and E when both are equal.
* During the rotate (left or right) operation, an additional flag, ERR, is used. The flag is set if the upper nibble of OPB equals 0xF; otherwise, it is in a high impedance state.

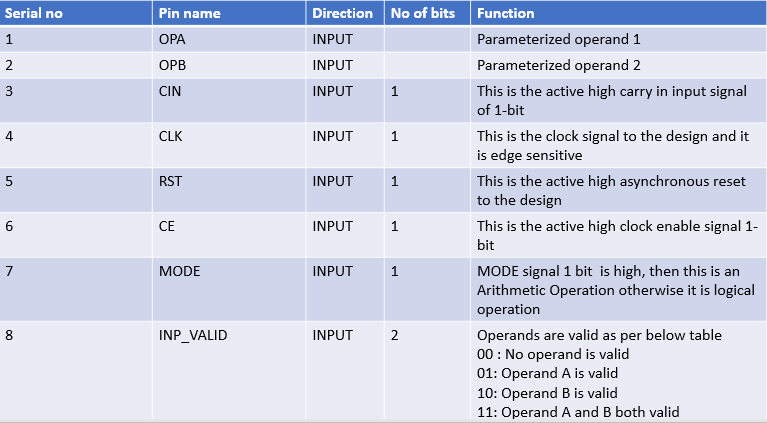
## 1.6 DESIGN LIMITATIONS

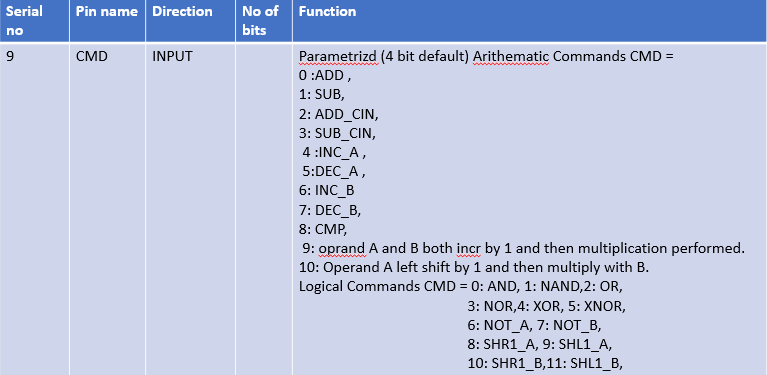
* Operation Speed: Limited by the propagation delay of signals through the ALU.
* Power Usage: Higher power consumption due to multiple operations being processed.
* Complexity in Handling Edge Cases: Ensuring correct results for all input combinations can be challenging.

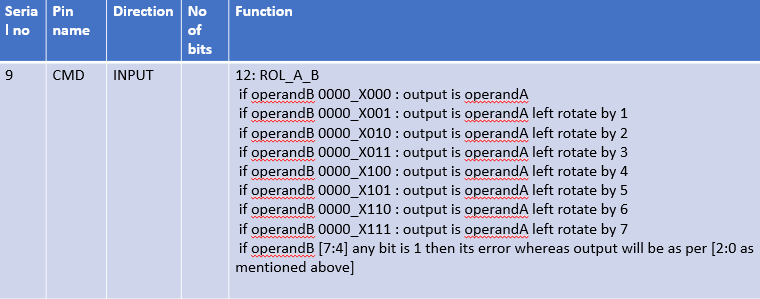
## 1.7 DESIGN DIAGRAM WITH INTERFACE SIGNALS

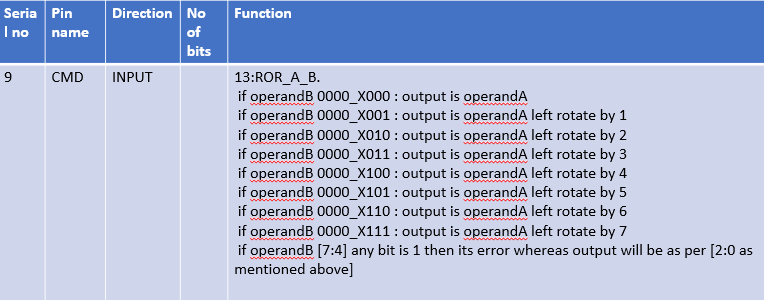


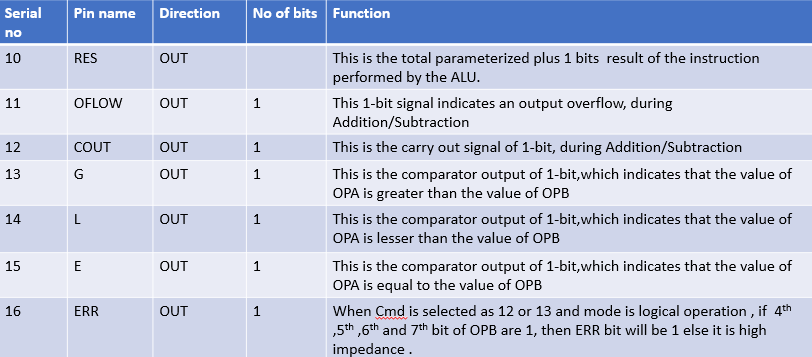
## 1.9 PIN LEVEL DESCRIPTION





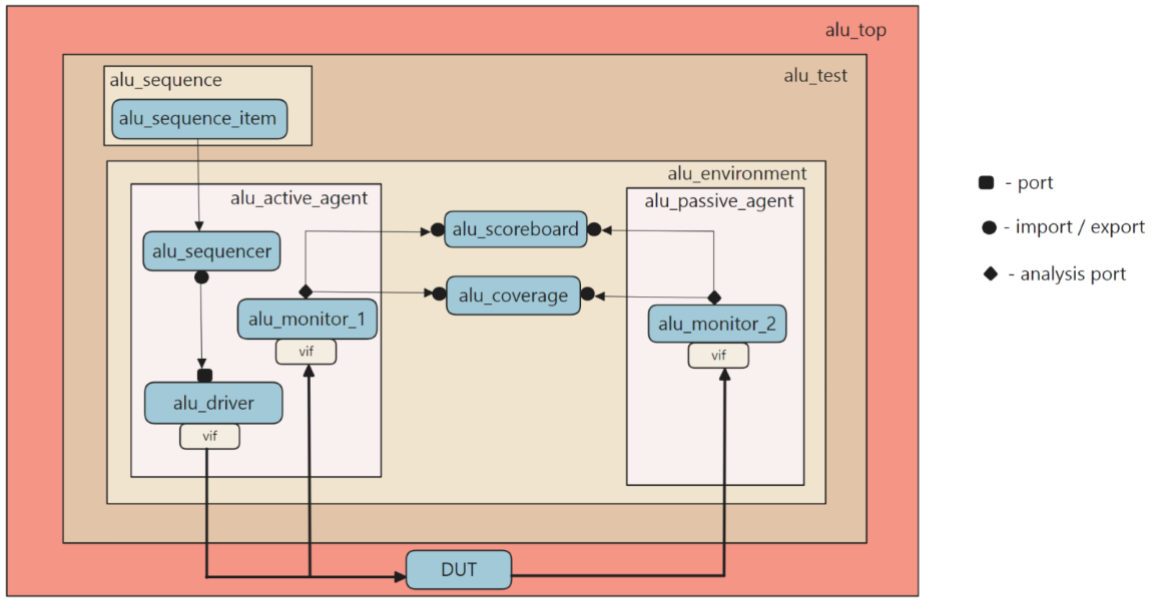






# **CHAPTER 2 – VERIFICATION ARCHITECTURE**

## 2.1 VERIFICATION ARCHITECTURE

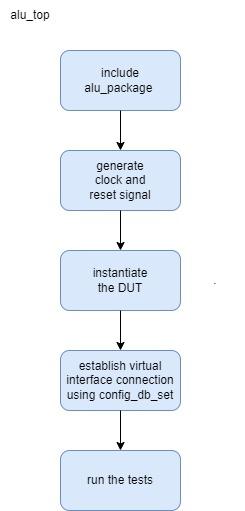


## 2.3 FLOWCHART OF UVM COMPONENTS

### 2.3.1 alu\_top

* Include UVM packages
  + Include UVM macros
  + Import package
* Instantiate the DUT
* Create Virtual Interface (VIF)
  + Instantiate VIF
  + Pass the VIF to the UVM environment using `uvm\_config\_db.
* Clock and reset generation
* Start the UVM test (run\_test())
* Waveform generation

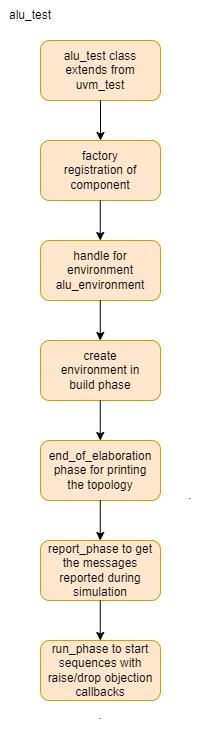


****

### 2.3.2 alu\_test

* Define alu\_test class
  + Extend alu\_test from uvm\_test
  + Use Macros to register the alu\_test with UVM factory
  + Create class constructor
  + Declare a handle for alu\_environment
* Implement the Build Phase
  + Instantiate the alu\_environment in the build phase
* Check for the end of elaboration
* Implement the report Phase
  + Generate report
  + Print summary



****

### 2.3.3 alu\_environment

* Define alu\_environment class
  + Extend alu\_environment from UVM\_environment
  + Use Macros to register the alu\_env with UVM factory
  + Create a class constructor
* Declare and Instantiate agents
  + Declare a handle for active agent (alu\_agent\_1) and passive agent (alu\_agent\_2)
* Declare and Instantiate Scoreboard
* Declare and Instantiate Coverage
* Implement the build phase
  + Create instance of all the components
* Implement the connect phase
  + Connect agents and scoreboard
  + Setup analysis port the transfer data from monitor to scoreboard and coverage



### 2.3.4 alu\_agent

* Define  alu\_agent class
  + Extend alu\_agent from uvm\_agent
  + Use Macros to register the alu\_agent with UVM factory
  + Create a class constructor
* Declare an handle for alu\_sequencer, alu\_monitor and alu\_driver
* Declare a flag “is\_active” to determine whether the agent is active or passive
* Implement the build phase
  + Instantiate the monitor and sequencer.
  + The build\_phase of the agent class checks this flag and decides whether to instantiate the driver.
* Implement the connect phase
  + Connect sequencer to driver if active agent
  + Connect monitor’s analysis port



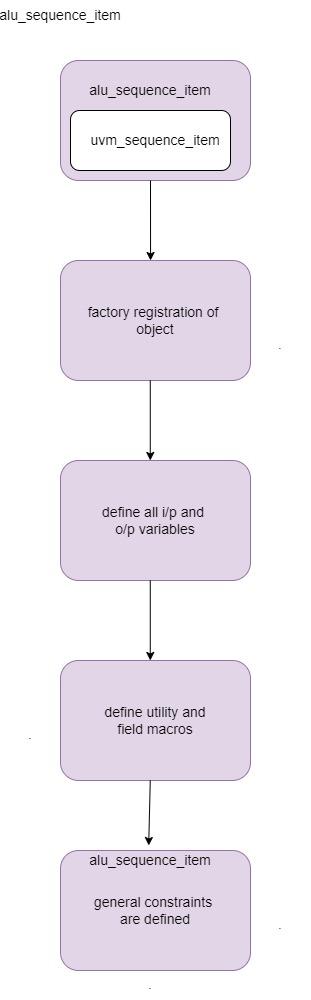
### 2.3.5 alu\_interface

* Interface module declaration
* Signal Declaration
* Clocking Block
* Modport Declaration



### 2.3.6 alu\_sequence\_item

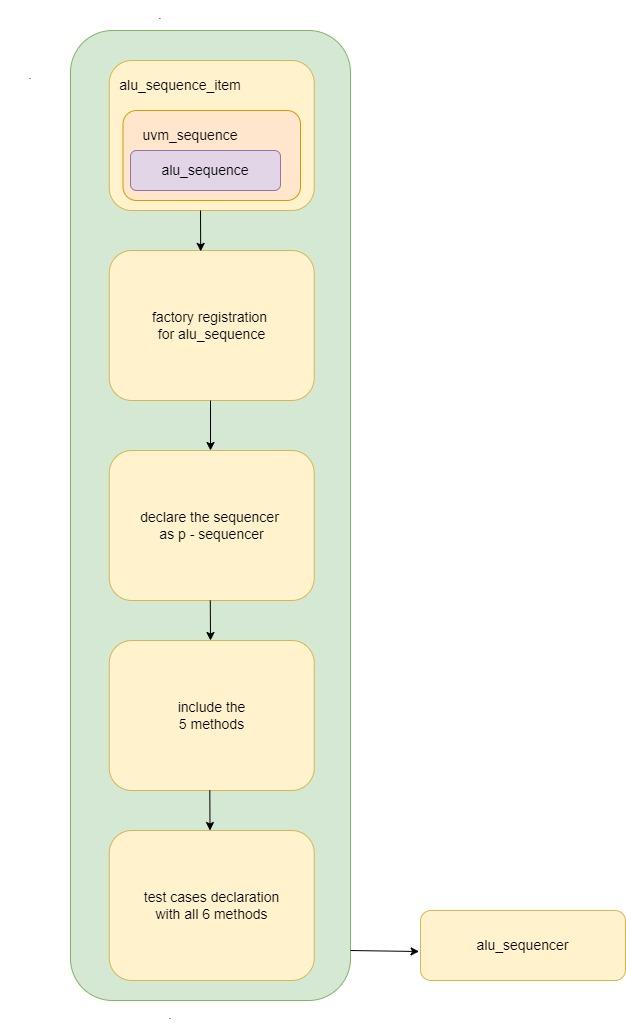
* User defined sequence items obtained from uvm\_sequence\_item class
* Input variables are declared with the rand keyword
* Output variables are not declared as randomizable
* The class includes constraints to guide randomization of inputs





### 2.3.7 alu\_sequence

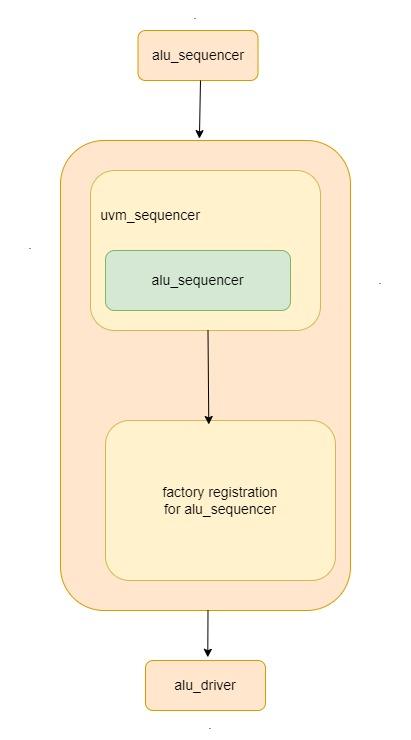
* User defined sequence is obtained from uvm\_sequence class
* Declaration of sequence as p if needed.
* The operation which is intended to be done by sequence is defined inside a body method.
* This includes 6 methods
  + Create\_item
  + Wait\_for\_grant()
  + randomize()
  + send\_request()
  + wait\_for\_item\_done()
  + get\_response()
* For each test cases same steps are repeated



### 2.3.8 alu\_sequencer

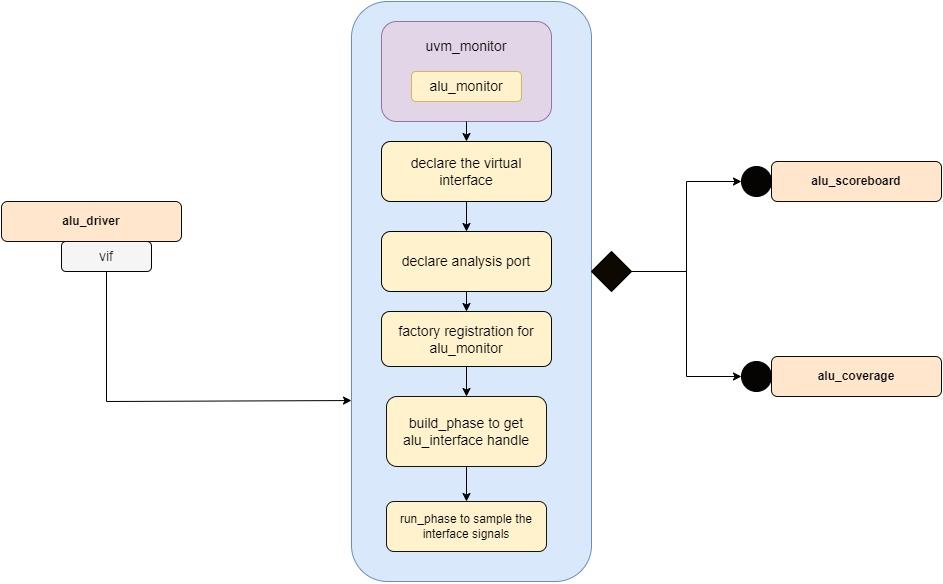
* A user-defined sequencer is  extended from the parameterized base class uvm\_sequencer
* Use of uvm macros for factory registration
* Usage of class constructor

Additionally uvm\_sequencer has a built-in mechanism to arbitrate  within concurrently running sequences over the sequencer (optional).



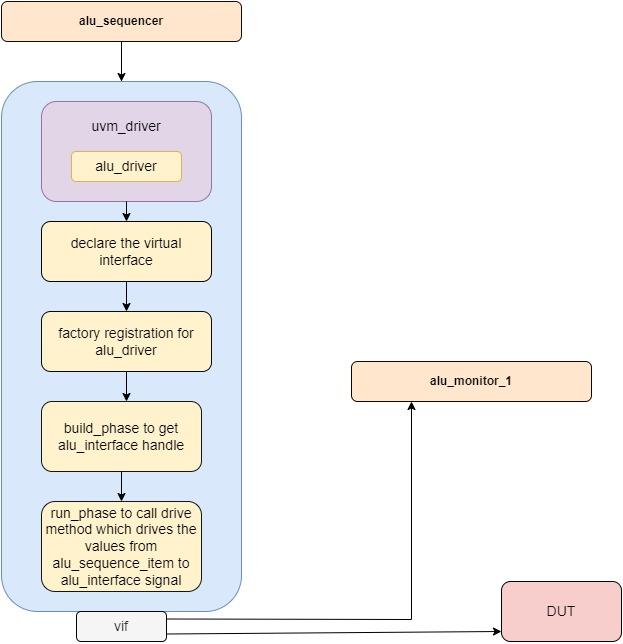
### 2.3.9 alu\_monitor

* Create a user-defined monitor class extended from uvm\_monitor and register it in the uvm factory.
* Declare analysis port to broadcast the sequence items or transactions.
* Declare virtual interface handle to retrieve actual interface handle
* Write standard class constructor create an instance for sequence\_item
* Implement build\_phase and get the interface handle from the configuration database.
* Implement run\_phase to sample DUT interface using a virtual interface handle and translate into transactions.
* The write() method sends transactions to the collector component.
* In alu\_monitor1 input signals are captured.
* In alu\_monitor2 DUT signals are captured.



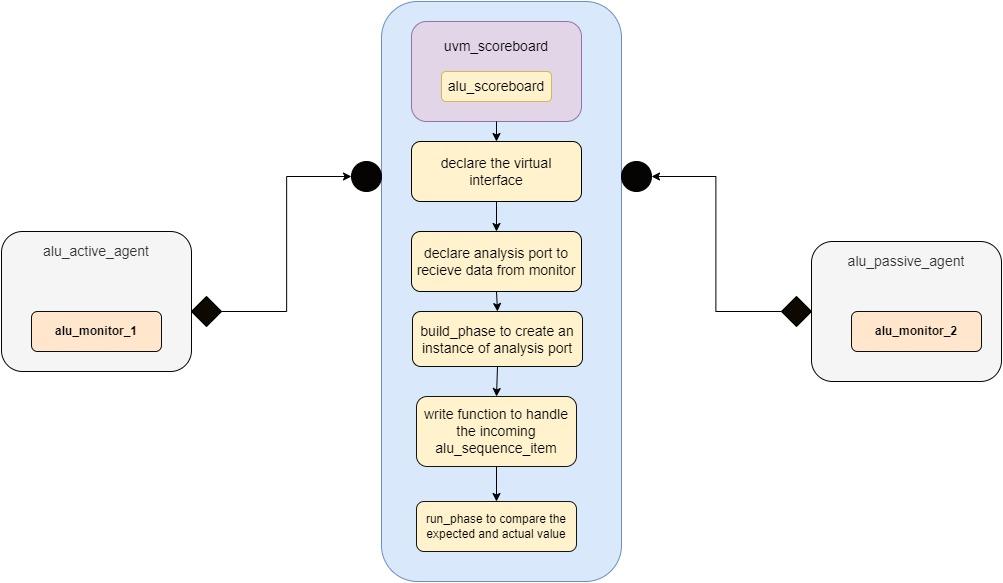
### 2.3.10 alu\_driver

* class name: alu\_driver
* virtual interface handle
* function build\_phase()
  + `uvm\_config\_db is used to retrieve a configuration setting from the UVM configuration database
* function run\_phase()
  + Methods to retrieve the next sequence item from the sequencer
    - “seq\_item\_port” to connect driver to the sequencer
    - “get\_next\_item()” to fetch the next sequence item from the sequencer queue.
    - “Drive()”  task is called.
    - “Item\_done()” method  indicates to the sequencer that the current sequence item processing has been completed.
* virtual task driver()
  + “drive()”  task is used to drive the values of the sequence item onto the DUT through the interface.



### 2.3.11 alu\_scoreboard

* class name: alu\_scoreboard
* Create a dynamic array to store “alu\_sequence\_item”
* analysis port “item\_collected\_export” is the port through which the sequence items are sent to the alu\_scoreboard analysis component
* function build\_phase ()
  + creates an instance of the “uvm\_analysis\_imp” class
* virtual function write ()
  + Function used to handle the incoming “mem\_sequence\_item” objects i.e transactions
* virtual task run\_phase ()



### 2.3.12 alu\_coverage

* define alu\_coverage class extended from uvm\_subscriber
* Define a coverage group that contains coverpoints and cross coverage.
* Coverage Group Constructor Is created to initialize the coverage group.
* write() method to collect data
* Register the alu\_coverage class with the UVM factory

