UVM Notes

1. Overview
    1. What is UVM
        - UVM is a standardized methodology, a set of pre-defined libraries using syntax and semantics of SystemVerilog.

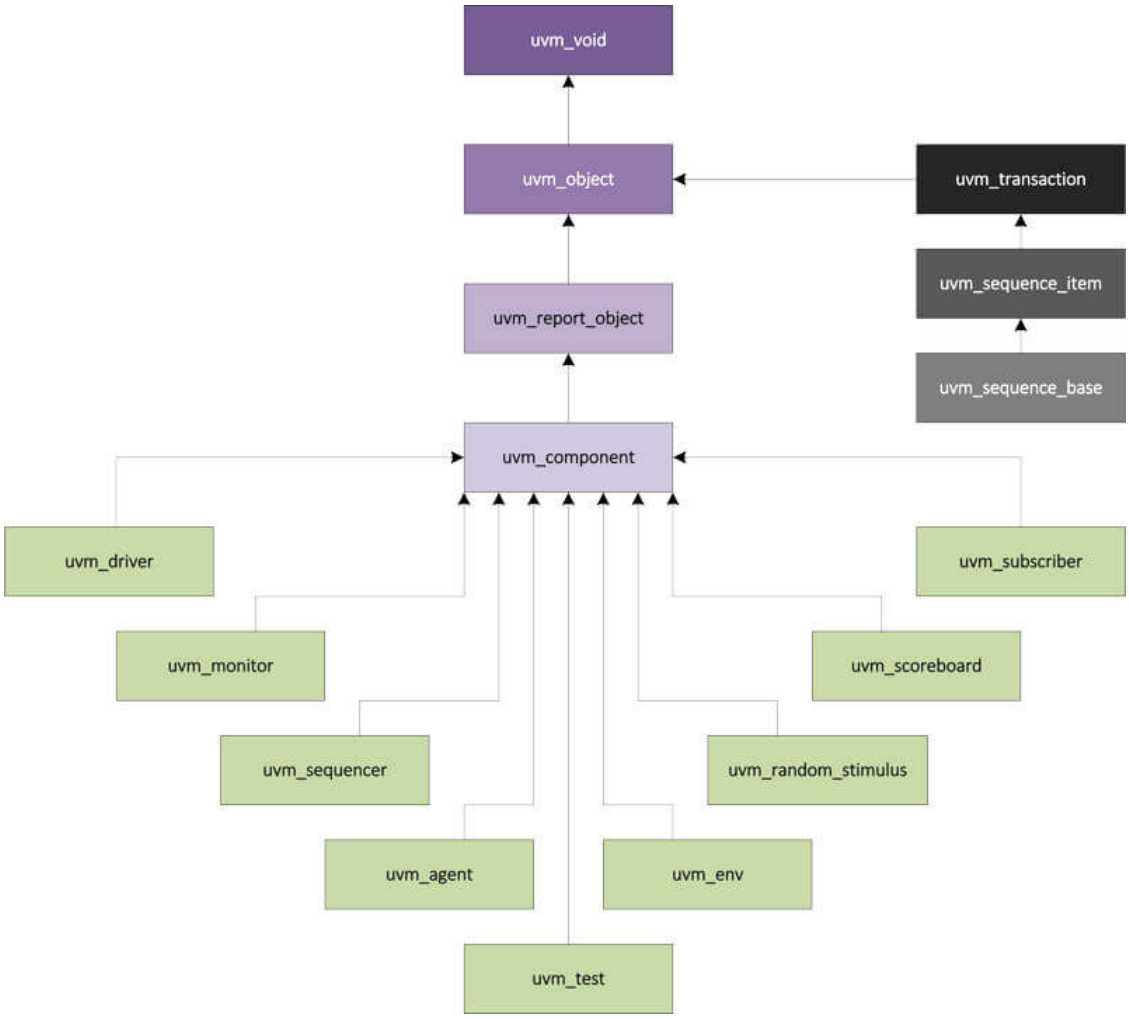
    b. Why UVM

    Increases reusability

    - The components(driver, sequencer. etc) are modules that can be re-used across projects.
    - Stimulus are separate from the actual testbench hierarchy and can be re-used or replaced by new stimulus.
    - Factory mechanisms make modifications of components easy. Create each components using factory enables them to be overridden in different tests/environments without changing the code base.

    c. UVM class hierarchy

    - UVM provides a set of base classes that more complex classes can be built by inheritance.
    - Two main branches. The verification components are underneath uvm_component, and data objects consumed and operated by components are underneath uvm_transaction.



- Sequence is a container for the actual stimulus to the design. Stimulus classes are inherited from uvm_sequence.
- Data objects that are driven to DUT are sequence items and are inherited from uvm_sequence_item.
- UVM utilizes TLM(transaction level modeling) which helps to send data between components in the form of transactions and class objects. It also can broadcast a packet to its listeners without creating specific channels and attach to it.
- Phases enable every component to sync with each other before proceeding to the next phases. Every component goes through the build phase when it gets instantiated, connects with each other during the connect phase, consumes simulation time during the run phase, and stops together in the final phase.

2. UVM Common Utilities
    1. Base Classes

- uvm_root
    - It is an implicit top_level UVM component that is automatically created when simulation is run.
    - Users can access it via the global variable, uvm_top. Any components whose parent is set to null becomes a child of uvm_top.
    - uvm_top checks for errors during end_of_elaboration phase and issue uvm_fatal error to stop simulation.

- uvm_report_object
  - All messages, warnings, errors issued by components go via this interface.
  - A report has ID String, Severity, Verbosity Level, and Test Message parts. If the verbosity level is GREATER than the configured maximum verbosity level, it is ignored. For example, if maximum verbosity level is UVM_MEDIUM, and a info is assigned to verbosity level UVM_HIGH, then this message will not be seen in the output.

b. UVM utility & field macros

- UVM uses the concept of a factory where all objects are registered.
- Utility Macros
  - The utils macro is used to register an object or component with the factory.
  - Required to be used inside every user-defined class derived from uvm_object
  - Object Utility
    - All classes derived directly from uvm_object or uvm_transaction required to be registered using `uvm_object_utils macro.
    - It is mandatory for the new function to be explicitly defined for every class, and take the name of the class instance as an argument.

      uvm_sequence is inherited from uvm_sequence_item, uvm_transaction, then uvm_object.

      ```
      class fc_sequence extends uvm_sequence #(fc_transaction);
          //register fc_sequence, this user-defined class with the factory
          `uvm_object_utils(fc_sequence)
          function new(string name = "fc_sequence");
              super.new(name);
          endfunction
      endclass
      ```

  - Component Utility
    - All classes derived directly or indirectly from uvm_component are required to be registered with the factory using `uvm_component_utils macro.
    - It is mandatory for the new function to be explicitly defined for every class, and takes the name of the class instance and a handle to the parent class where this object is instantiated.
  - Macro Expansion
    - `uvm_object_utils gets expanded into its *_begin and *_end form with nothing in between.
    - *_begin implements other macros, such as
      - `m_uvm_object_registry_internal(T,T), which implements the function get_type() and get_object_type() that returns a factory proxy object for the requested type
      - `m_uvm_object_create_func(T) which instantiates an object of the specified type by calling its no-args constructor
      - `m_uvm_get_type_name_func(T) which return the type_name as a string.
      - `uvm_field_utils_begin(T) which registers the type with UVM factory
  - Creation of class object
    - Recommend all class objects are created by calling the type_idLLcreate() method which is defined using the macro `m_uvm_object_create_func(T). (this macro utilizes the new() function. When creating component object, two arguments are taken which are the name and parent).

```
fc_drv = fc_driver::type_id::create("fc_drv",this);
```

- Field Macros
  - `uvm_field_* macros that were used begween *_begin and *_end provide automatic implementations of core methods like copy and compare.
  - `uvm_field_* corresponding to the data type of variables been used. Variables of type int, bit, byte should use `uvm_field_int, type string should use `uvm_field_string and so on.
  - The macro accepts at least two arguments, ARG and FLAG. ARG is the name of the variable, FLAG specifics which data method implementations will not be included(except UVM_ALL_ON and UVM_DEFAULT).
    - UVM_ALL_ON: all operations are turned on
    - UVM_DEFAULT: enables all operations, equivalent to UVM_ALL_ON
    - UVM_NOCOPY, UVM_NOCOMPARE, UVM_NOPRINT, UVM_NOPACK: do not copy, compare, print, pack/unpack the given variable
    - UVM_REFERENE: operate only on handles.

- UVM Object Print
  - After using type_id create to create an object, we can randomize it and print it using obj.randomize() and obj.print().
  - do_print()
    - using automation macros introduces additional codes and reduces simulator performance.
    - We can use do_* callback. For example, we can use do_print inside the derived object. do_print is called by the print function by default.

```
virtual function void do_print(uvm_printer printer);
    super.do_print(printer);
    printer.print_string();
    //can control the radix of the given varaible, such as UVM_HEX or UVM_DEC
    printer.print_field_int();
endfunction
```

- UVM Object Copy/Clone
  - we can use obj2.copy(obj1) method to copy the content of obj1 into obj2
  - do_copy()
    - A generic uvm_object called "rhs" is received and type casted into Packet pkt. Then m_addr is copied from the type-casted _pkt to the variable of the current class.

      "rhs" does not contain o_bool as its only a parent handle. We cast this rhs into child data type and access it using child handle. We then copy content of casted handle into local variables.

```
class Packet extends uvm_object;
    rand bit[15:0] m_addr;
    ...
    virtual function void do_copy(uvm_object rhs);
        Packet _pkt;
        super.do_copy(rhs);
        $cast(_pkg, rhs);
        m_addr = _pkg.m_addr;
    endfunction
    ...
```

```
endclass

class Object extends uvm_object;
    rand Packet m_pkg;
    rand bool o_bool;
    ...
    virtual function void do_copy(uvm_object rhs);
        Object _obj;
        super.do_copy(rhs);
        $cast(_obj, rhs);
        o_bool = _obj.o_bool;
        m_pkg.copy(_obj.m_pkg);
    endfunction
endclass
```

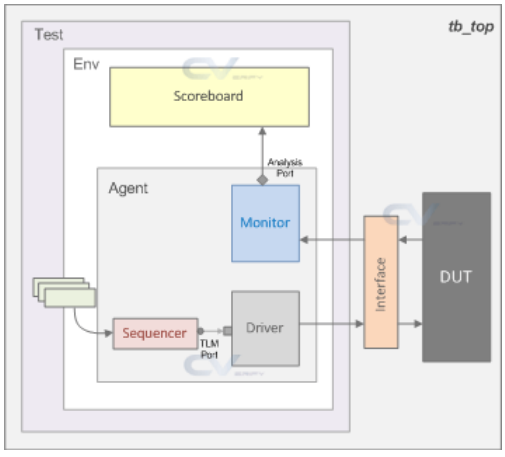
- Clone
  - Clone will return an object with the copied contents, so no need of creating the second object before copy.

- UVM Object Compare
  - we can use something like obj2.compare(obj1).
  - do_compare

```
//inslde Packet class
virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    bit res;
    Packet _pkt;
    $cast(_pkt, rhs);
    super.do_compare(_pkt, comparer);
    res = super.do_compare(_pkt, comparer) & m_addr = _pkt.m_addr;
    return res;
endfunction

virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    bit res;
    Object _obj;
    $cast(_obj, rhs);
    res = super.do_compare(_obj, comparer) & o_bool == _obj.o_bool
    & m_pkg.do_compare(_obj.m_pkt,comparer);
    return res;
endfunction
```

3. Testbench Structures
   1. UVM Testbench Top
      - All verification components, interfaces, and DUT are instantiated in a top level module called testbench.



- At the start of simulation, set the interface handle as a config object in UVM database(uvm_config_db::set). This if can be retrieved in the test using the get() method.
- run_test("test_name") accepts test name as argument, and the test_name case will be run for simulation
- tb_top is a static container

2. UVM Test
   - We can put the entire testbench into a container, environment, and use same environment for different test. Each testcase can manipulate agents, and run different sequences on many sequencers in the environment.
   - As shown in the above top-level picture, we can start a virtual/normal sequence on a given sequencer in the run_phase of the test. Remember to raise and drop the objection
   - A base test sets up all basic environment parameters and configurations that can be overridden by derivative tests. For the new test, we can define the phases we want to change, the the object will call its parent's phases that's not explicitly defined.
3. UVM Environment
   - A UVM environment contains multiple reusable verification components and defined their default configuration as required.
   - It is possible to instantiate agents and scoreboards directly in uvm_test, but tests become non-reusable because
     - They rely on a specific environment structure.
     - The test writer would need to know how to configure the environment
   - uvm_env is the base class for hierarchical containers of other components that make up a complete environment
   - We need to connect verification components together in the connect_phase
4. UVM Driver
   - UVM driver drives transactions to a particular interface of the design. Transaction level objects are obtained from the sequencer and the UVM driver drivers them to the design via an interface handle.
   - UVM Driver-Sequencer handshake
     - The UVM driver uses following methods to interact with the sequencer.
       - get_next_item: blocks until a request item is available from the sequencer. Should be followed by item_done to complete the handshake.
       - try_next_item: non_blocking method which return null is a request object is not available from the sequencer. Else returns a pointer to the object
       - item_done: non_blocking method which completes the driver-sequencer handshake. Should be called after get_next_item or a successful try_next_item call.
     - A driver-sequencer handshake allow the driver to get a series of transaction objects from the sequence and respond back to the sequence after it finishes driving the given item, so it can get the next sequence
       - get_next_item + item_done
         - fininsh_item call in the sequence finishes only after the driver returns item_done call

```
virtual task run_phase (uvm_phase phase);
    my_data req_item;
```

```
    forever begin
        seq_item_port.get_next_item(req_item);
        @(posedge vif.clk);
        vif.en <= 1;
        seq_item_port.item_done();
    end
endtask
```

- get + put
  - The driver gets the next item and send back the sequence handshake in one go, before the UVM driver processes the item
  - The driver uses the put method to indicate that the item has been finished later.
  - So finish_item call in the sequence is finished as soon as get() is done
- A virtual interface handle vif is declared and assigned later in the build phase
- Real interface object is retrieved from the database directly into a local variable using uvm_config_db::get()

5. UVM Sequencer
   - A sequencer generates data transactions as class objects and sends it to the Driver for execution
   - The uvm_sequener base class is parameterized by the request and response item types and can be handled by the sequencer. By default, response type is the same as the request type.
6. UVM Sequence
   - UVM sequences are made up of several data items.
   - Executed by assigned sequencer(s) which then send(s) data items to the driver. Sequences are core stimuli of any verification plan.
   - We can make the body task virtual so child classes can override the task definition
   - We can use pre_body() and post_body() callbacks
   - We can use `uvm_do() sequence macros, which we have to provide a uvm_sequence_item object or a sequence and it does the following internally:
7. UVM Monitor
   - A UVM monitor is responsible for capturing signal activity from the design interface and translate it into transaction level data objects that can be sent to other components. It should have a virtual interface handle to the actual interface that this monitor is trying to monitor, and TLM analysis port declarations to broadcast captured data to others
   - Its functionality should be limited to basic monitoring that is always required. High level functional checking should be done in a scoreboard.
8. UVM Agent
   - An agent encapsulates a sequencer, driver, and monitor into a single entity. We can have active or passive agent, which only instantiate the monitor and is used for checking and coverage only. We can use uvm_config_db::set to configure a passive or active agent by using is_active variable.
   - We can use get_is_active() to check whether to create sequencer and driver.
9. UVM Scoreboard
   - It is a verification component that contains checkers and verifies the functionality of a design. It usually receives transaction level objects captured from the interfaces of a DUT via TLM Analysis Ports

- After receiving data objects, the scoreboard can either perform calculations and predict the expected value, or send it to a reference model to get expected value. The reference model is also called a predictor that mimics the functionality of the design. The scoreboard then compares the expected results with the actual output data from DUT
- It is not required to perform checks in the check_phase. Real checkers can also check during the run_phase.
- After connecting the scoreboard with other components(e.g. monitor), monitor can send data to the scoreboard via an analysis port by calling the port's write method.

10. UVM Subscriber
    - Subscribers are listeners of an analysis port. They subscribe to a broadcaster and receive objects whenever an item is broadcasted via the connected analysis port.
11. UVM Virtual Sequencer
    - It is a UVM sequencer that contain handles to other sequencers.

4. UVM Phases
    - All testbench components are derived from uvm_component and goes through a pre-defined set of phases. It cannot proceed to the next phase until all components finish their execution in the current phase.
    - We have functions that are methods that do no consume simulation time and tasks that consume simulation time
        - Build time phases. Functions
            - build_phase, used to build testbench components and create their instances
            - connect_phase, used to connect between different testbench components via TLM ports
            - end_of_elaboration_phase, used to display UVM topology(e.g. print_topology displays all instantiated components in the environment to help debug) and other functions required to be done after connection
            - start_of_simulation_phase, used to set initial run-time configuration or display topology.
        - Run time phases. Tasks
            - run_phase. Actual simulation that consumes time, and runs parallel to other UVM run-time phases.
        - Clean-Up phases. Functions
            - extract_phase, used to extract and compute expected data from scoreboard
            - check_phase, used to perform scoreboard tasks that check for errors between expected and actual values from design
            - report_phase, used to display result from checkers, or summary of other test objectives
            - final_phase, used to do last minute operations before existing the simulation
    - Why doesn't Verilog tb need phases?
        - All of its components made of static containers(modules), so each module will have a set of ports/signals that it utilizes to communicate with other tb components.
        - Since a module is static, all modules will be created at the beginning of the simulation.
    - Why SystemVerilog testbench require phases?
        - With OOP, entities(class objects) can be reused and deployed when required.
        - It is possible to create a new object in Xns, we it is possible to call a component that's not initialized.

- In addition, we need synchronization between testbench components.

5. UVM Factory Override
   - UVM Factory is a mechanism to improve flexibility and scalability of the tb by allowing the user to substitute an existing class object by any of its inherited child class objects.
   - Factory needs to know all types of classes created within the tb via registration.
   - Why Override? With the help of the factory, we can override the type of underlying components or objects from the top-level component without having to edit the code.
     - For example, if we want to replace new_driver() with the base_drvier(), all we have to do is to override the base driver by one of the factory override methods, instead of going to code and substitute every single code that mentions base_driver();
   - Factory Override Methods
     - set_type_override_by_type()/set_type_override_by_name(), which override all the objects of a particular type.
       - e.g. set factory to override "base_agent" by "child_agent" by type
         - set_type_override_by_type(base_agent::get_type(), child_agent::get_type());
       - e.g. set by name

         ```
         uvm_factory factory = uvm_factory:get();
         factory.set_type_override_by_name("base_agent","child_agent");
         ```

     - set_inst_override_by_type()/set_inst_override_by_name(), which override a type within a particular instance
       - When only a few instances of the given type has to be override, we can use instance override by type/name.
       - e.g. set factory to override all instances under my_env of type "base_agent" by "child_agent".
         - set_inst_override_by_type("my_env.*", base_agent::get_type(), child_agent::get_type());
       - e.g. set by name

         ```
         uvm_factory factory = uvm_factory:get();
         factory.set_inst_override_by_name("base_agent", "child_agent", {get_full_name(), ".my_env.*"});
         ```

6. Stimulus Generation
   - Sequences can do operations on sequence items, or initiate new subsequences
     - Execute using the start() method of a sequence
     - Execute sequence items via start_item/finish_item
     - we can also use `uvm_do macro, which will identify if the argument is a sequence or sequnece_item and will call start() or start_item() accordingly.
       - create the item using `uvm_create if necessary.
       - randomize the item or sequence.
       - call the start_item() and finish_item() if its a uvm_sequence_item object.
       - call the start() task if its a sequence.
       - `uvm_do: execute this sequence on default sequencer with the item provided

    - `uvm_do_with : override any default constraints with inline value
    - `uvm_do_pri : execute based on the priority value, used when running multiple sequences simultaneously
    - `uvm_do_pri_with : execute based on priority and override default constraints with inline values

7. Driver Sequencer Handshake
    - The driver contains a TLM port uvm_seq_item_pull_port which is connected to a uvm_seq_item_pull_export in the sequencer in the connect phase of a UVM agent. The driver can use TLM functions to get the next item from the sequencer when required
    - We need the driver sequencer API because this helps the driver to get a series of sequence_items from the sequencer's FIFO that contains data for the driver to drive to the DUT. The driver will send finish signal to the sequence and can request the next item.
    - seq_item_port can be used by derived driver class to request items from the sequencer and send response back. rsp_port provides an alternative way of sending response back to the originating sequencer.
    - seq_item_export is an inbuilt TLM pull implementation port in a uvm_sequencer, which is used to connect with the driver's pull port.
    - Typically, a driver and sequencer are instantiated and connected in a uvm_agent
        - drv.seq_item_port.connect(seqr.seq_item_export);
        - This is one-to-one. Multiple drivers are not connected to a sequencer nor are multiple sequencers connected to a single driver. Once the connection is made, the driver can utilize API calls in the TLM port definitions to receive sequence items from the sequencer.
    - get_next_item()
        - The driver is a parameterized class with the type of request and response sequence items.
        - The uvm_driver gets request sequence item(REQ) from the sequencer FIFO and optionally returns a response sequence item(RSP) back to the sequencer response FIFO.
            - *The driver is allowed to send back a different sequence_item type back to the sequencer as the response. And of course, its more common to send the same type as the request sequence item
        - A uvm_sequence is started on a sequencer which pushes the sequence item onto the sequencer's FIFO.
            - Create an item the connected sequencer can accept
            - Call the start_item() task which sends this object to the driver
            - Because the class handle passed to the driver points to the same object, we can do late randomization
            - Call the finish_item() method so that the sequence waits until the driver lets the sequencer know this item has finished
    - Using get() and put()
        - We can let the driver us get() method to receive the next item and later use put() to give a response item back to the sequencer
        - So how does a sequencer stop an item now? Because finish_item does not indicate that the driver has finished driving the item, the sequence has to wait until the driver explicitly tells the sequencer that the item is over. So the sequence has to wait until it gets a response back from the sequencer via get_response();
8. Reporting Infrastructure

- Reporting Functions
  - There are four basic reporting functions with different verbosity levels
    - uvm_report_* ("TAG", $sformatf("[display message]"), VERBOSITY_LEVEL);
    - * can be info, error, warning, fatal
    - verbosity level has six levels, UVM_NONE(0), UVM_LOW(100), UVM_MEDIUM(200), UVM_HIGH(300), UVM_FULL(400), UVM_DEBUG(500)
    - NOTE: verbosity level is only required for uvm_report_info. Usage of uvm_report_warning, uvm_report_error, uvm_report_fatal do not require verbosity. In fact, uvm_report_fatal will exit the simulation.
    - We can display the filename and line number of the displayed message by using `_FILE_ , `_LINE_. UVM reporting macros will automatically display the file and line information without explicitly mentioning the `_FILE_ , `_LINE_.
    - Verbosity level controls whether a uvm_report_* statement gets displayed. Default configuration is UVM_MEDIUM, means every uvm_report_* message with a verbosity level less than UVM_MEDIUM will be printed. This controls the number of information will be displayed. If you want to debug, you can set verbosity level to UVM_DEBUG, then everything under it will be displayed.
- uvm_printer
  - UVM avoids the need for customized print function by incorporating its own uvm_printer class.
  - Every class item derived from uvm_object will have a printer instance within it. So a data class derived from uvm_sequence_item will have access to the print() function
  - UVM has three main printer: table printer, tree printer, line printer
    - By default, UVM assigns table printer to handle every print() function, hence is the uvm_default_printer.

```
class my_data extends uvm_sequence_item;
    bit [3:0] bit_data;
endclass

my_data data_obj;
data_obj.print(); //calls table printer by default
data_obj.print(uvm_default_line_printer); //calls line printer
```

  - Calling print() is possible if EITHER of the following things are done(we can do together and do_print() will append to the macro)
    - Add any member that needs to be printer within `uvm_object_utils_begin and `uvm_object_utils_end
    - Define a do_print() function for the class

9. UVM Config DB

- UVM resource databae
  - A resource database is a parameterized container that holds arbitrary data
  - Can put any data type into the resource database, and have another component retrieve it later at some point in simulation
  - The global resource database has both a name table and a type table into which each resource is entered
  - So the same resource can be retrieved later by name and type

- Multiple resources with the same name/type are stored in a queue and hence those which were pushed earlier have more precedence over those placed later
- For example if item red and item blue in the queue have the same scope, and a get_by_type() method is called for that particular scope.Then item red will be returned since that sits earlier in the queue
- Resources are added to the pool by calling set, and they are retrieved from the pool by get_by_name() or get_by_type()

- UVM config database
  - UVM has an internal database table in which we can store values under a given name and can be retrieved later by other tb component
  - uvm_config_db class provides a convenience interface on top of the uvm_resource_db to simplify the basic interface used for uvm_component instance
  - set()

```
static function void set (uvm_component cntxt,
                          string inst_name,
                          string field_name,
                          T value);
```

    - use this static function of the class uvm_config_db to set a variable in the configuration database
    - set() function will set a variable of name test_enable at the path uvm_test_top.env.agt with value 1
    - use of set() will create a new or update an existing configuration setting for field_name in inst_name from cntxt. This setting is made at cntxt with the full scope of the set begin {cntxt, ".", inst_name}. if cntxt is null, then the complete scope of getting the information will be provided by inst_name.

```
//set virtual interface handle under name "fcif" available to all components under "this" phase, indicated by the *
uvm_config_db#(virtual flex_counter_if)::set(this, "*", "fcif", fcif);
//if in the test phase, then it is equivalent to
uvm_config_db#(virtual flex_counter_if)::set(null, "tb_flex_counter.fc_test.*", "fcif", fcif);
```

      For the cntxt this, which will be substituted by the path to the current component which in the case tb_flex_counter.fc_test.

  - get()

```
static function void get (uvm_component cntxt,
                          string inst_name,
                          string field_name,
                          T value);
```

    - use this static function to get the value of variable given in field_name from the configuration database. The value will be returned only if the scope is true.

```
//get virtual interface handle under name "fcif" into local virtual interface handle at fc_test level
uvm_config_fb #(virtual flex_counter_if)::get(this, "*", "fcif", fcif);
```

  - exists()

```
static function void exists (uvm_component cntxt,
                          string inst_name,
                          string field_name,
                          bit spell_chk);
```

- checks if a value for field_name is available in inst_name, using component cntxt as the starting point. If the field_name does not exist at a given scope, the function will return a zero. The spell_chk arg can set to 1 to turn spell checking on if it is expected that the field should exist in the database

```
if (!uvm_config_db#(virtual flex_counter_if)::exists(this, "*", "fcif"))
  `uvm_error ("fcif", "cannot find fcif handle");
```