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.

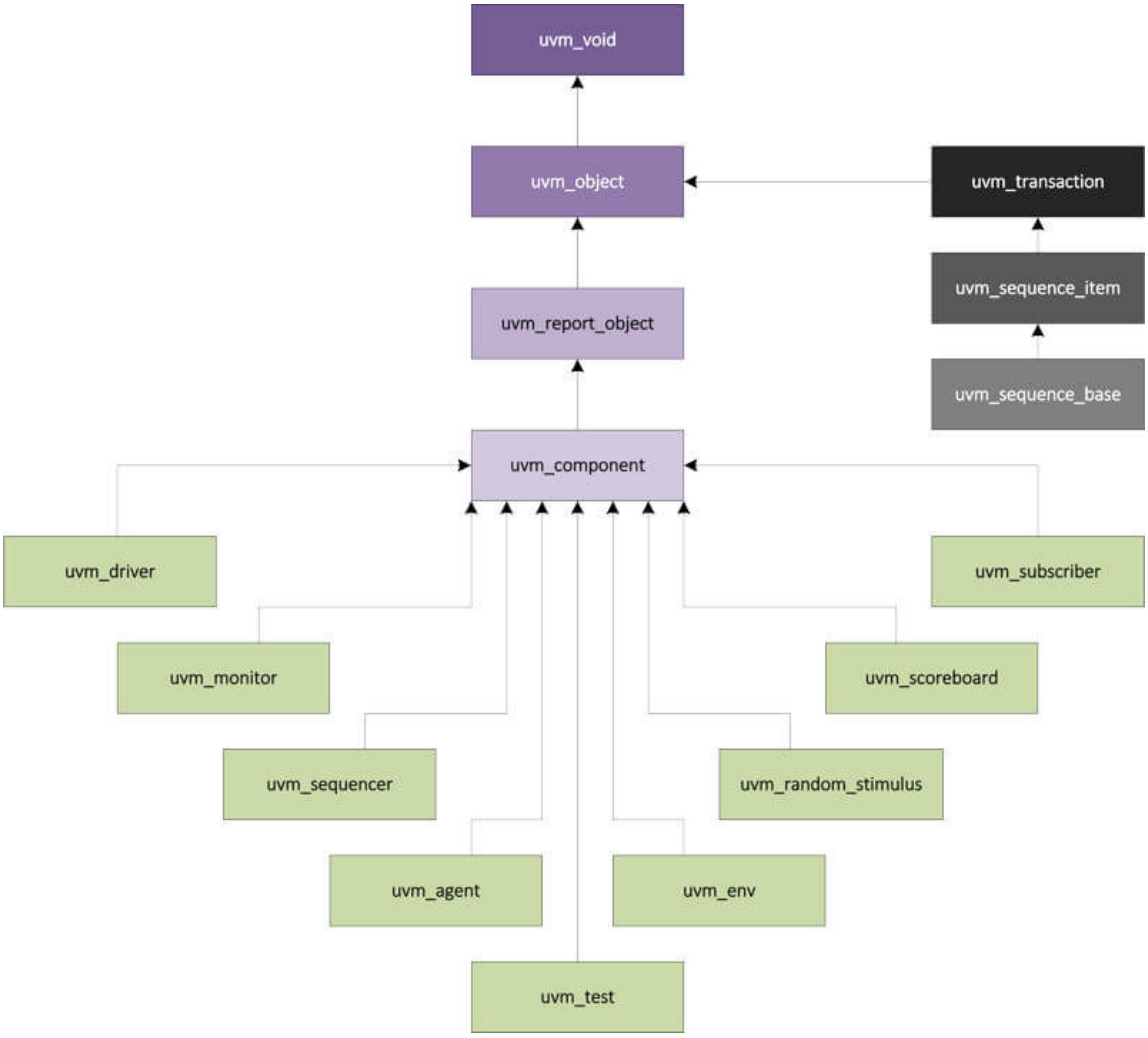
    b. Why UVM

    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.

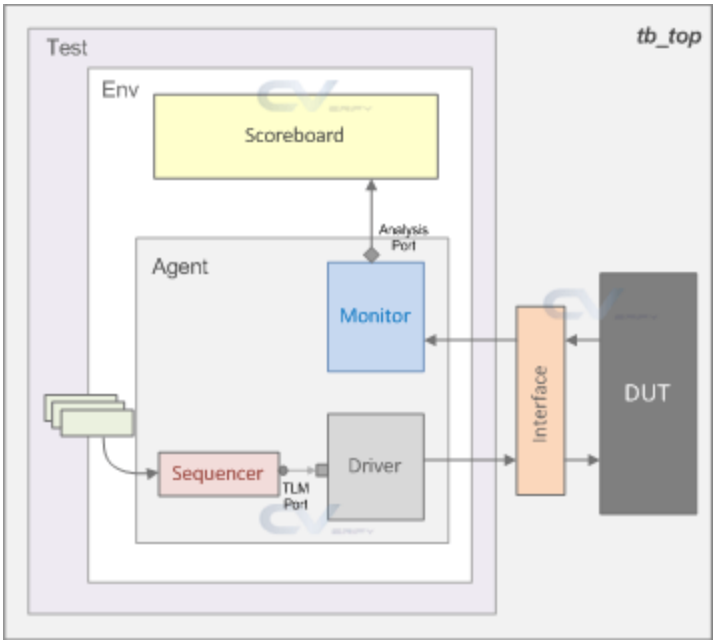
      "rhs" does not contain o_bool as its only a parent handle. We cast this rhs into child data type and access it using child handle. We then copy content of casted handle into local variables.

```
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.
5. UVM Sequencer
6. UVM Sequence
7. UVM Monitor
8. UVM Agent
9. UVM Scoreboard
10. UVM Subscriber
11. UVM Virtual Sequencer