This lecture aims to:

- demonstrate the need for a verification methodology
- provide an understanding of some of the key components of a UVM testbench
- cover some basic features of UVM
ARM CCN-512 SoC Framework

What are the challenges of verifying complex systems?

- Typical processor development from scratch could be 100s of man years
- Multiple parallel developments, multiple sites and it takes a large team to verify a processor
- The typical methods is to divide and conquer, partitioning the whole CPU into smaller units and verify those units, then reuse the checkers and stimulus at a higher level
- The challenges are numerous
- Reuse of code becomes an absolute key to avoid duplication of work
- It is essential to have the ability to integrate an external IP
- This requires rigorous planning, code structure, & lockstep development
- Standardization becomes a key consideration
- UVM can help solve this!
What is UVM and why use it?

- Stands for Universal Verification Methodology
- Benefits:
  - supports and provides framework for modular and layered verification components
  - Enables:
    - reuse
    - clear functional definition for each component
    - configuration of components to be used in a variety of contexts
- is maintained and released by Accellera committee
- source code is fully available
- is a mature product
- significant amount of training and support available

Key components of a UVM testbench
### UVM Sequence Item & Sequence Inheritance tree

Sequence Item is the same as a transaction
- It's the basic building block for all types of data in UVM
- Collection of logically related items that are shared between testbench components
- Examples: packet, AXI transaction, pixel
- Common supported methods:
  - create, copy, print, compare

UVM Sequence is a collection/list of UVM sequence items
- UVM sequence usually has smarts to populate the sequence but sometimes this is separated into a UVM generator
**UVM Component**

- Basic building block for all components that exercise control over testbench or manage transactions
- They all have a time consuming run() task
- They exist as long as the test exists

**Key components of a UVM testbench**

- **TOP Env**
  - TX Env
    - TX Agent
      - Sequencer
      - Driver
      - Monitor
  - TX Env
    - Scoreboard
    - Functional Coverage
  - RX Env
    - RX Agent
      - Sequencer
      - Driver
      - Monitor

- **DUT**

---

Nagesh Loke, ARM
10. UVM Basics

UVM Sequencer & Driver

- A UVM sequencer connects a UVM sequence to the UVM driver.
- It sends a transaction from the sequence to the driver.
- It sends a response from the driver to the sequence.
- Sequencer can also arbitrate between multiple sequences and send a chosen transaction to the driver.
- Provides the following methods:
  - `send_request()`, `get_response()`

A UVM driver is responsible for decoding a transaction obtained from the sequencer.
- It is responsible for driving the DUT interface signals.
- It understands the pin level protocol and the timing relationships.

### Class alu_driver

```pascal
class alu_driver extends uvm_driver #alu_trxm;

    `uvm_component_utils(alu_driver)

    // Data members
    virtual interface alu_inf alu_if;

    task run_phase(uvm_phase phase);
        forever begin
            seq_item_port.try_next_item(req);
            if (req != null) begin
                // Nibble pins
                seq_item_port.item_done();
                @alu_if.cb;
                alu_if.cb.opcode <= req.op;
                alu_if.cb.A <= req.a;
                alu_if.cb.B <= req.b;
            end
        endtask // run_phase

endclass // alu_driver
```

UVM Monitor

- Monitor’s responsibility is to observe communication on the DUT interface.
- A monitor can include a protocol checker that can immediately find any pin level violations of the communication protocol.
- UVM Monitor is responsible for creating a transaction based on the activity on the interface.
- This transaction is consumed by various testbench components for checking and functional coverage.
- Monitor communicates with other testbench components using UVM Analysis ports.

### Class alu_monitor

```pascal
class alu_monitor extends uvm_monitor;

    `uvm_component_utils(alu_monitor)

    uvm_analysis_port#alu_trxm a_port;

    // Data members
    virtual interface alu_intf alu_if;

    function new (string name, uvm_component parent);
        super.new(name, parent);
        endfunction new

    function void build_phase(uvm_phase phase);
        super.build_phase();

        // Get a virtual interface handle from the resource db
        void#alu_intf#virtual alu_intf);
        read_by_name(
            scope("alu_interface"),
            name("alu_if"),
            value(alu_if)
        )
        a_port = new(name("a_port"), parent(this));
        endfunction build_phase

    task run_phase(uvm_phase phase);
        // Read code
        a_port.write(trxm);
        // Post code
        endtask // run_phase

endclass // alu_monitor
```
Key components of a UVM testbench

- **TOP Env**
  - TX Env
    - TX Agent
      - Sequencer
      - Driver
      - Monitor
  - TX Env
  - Scoreboard
  - Functional Coverage
- **RX Env**
  - RX Agent
    - Sequencer
  - RX Env
  - Monitor
  - Interface
  - DUT

**UVM Agent**

```
class alu_agent extends uvm_agent;
  `uvm_component_utils (alu_agent);
  `uvm_analysis_port #alu_trm a_port;
  alu sequencer m_sequencer;
  alu driver m_driver;
  alu monitor m_monitor;
  function new(string name, uvm_component parent);
    super.new(name, parent);
    endfunction // new

  function void build_phase(uvm_phase phase);
    if (!get_is_active()) `UVM_ACTIVE
      begin
        m_sequencer = alu_sequencer::type_id::create("m_sequencer", this);
        m_driver = alu_driver::type_id::create("m_driver", this);
        end

        m_monitor = alu_monitor::type_id::create("m_monitor", this);
        a_port = new("a_port", this);
        endfunction // build_phase

  function void connect_phase(uvm_phase phase);
    if (!get_is_active()) `UVM_ACTIVE
      m_driver.seq_item_port.connect( m_sequencer.seq_item_export );
      m_monitor.a_port.connect( a_port );
    endfunction // connect_phase

  virtual function uvm_active_passive_enum get_is_active();
    return uvm_active_passive_enum::UVM_ACTIVE;
  endfunction // get_is_active
endclass // alu_agent
```

- UVM Agent is responsible for connecting the sequencer, driver and the monitor
- It provides analysis ports for the monitor to send transactions to the scoreboard and coverage
- It provides the ability to disable the sequencer and driver; this will be useful when an actual DUT is connected
UVM Scoreboard

- Scoreboard is one of the trickiest and most important verification components
- Scoreboard is an independent implementation of specification
- It takes in transactions from various monitors in the design, applies the inputs to the above model and generates an expected output
- It then compares the actual and the expected outputs
- A typical scoreboard is a queue implementation of the modeled outputs resulting in a pop of the latest result when the actual DUT output is available
- A scoreboard also has to ensure that the timing of the inputs and outputs is well managed to avoid false fails

UVM Environment

```plaintext
class alu_env extends uvm_env;
    `uvm_component_utils (alu_env)

    // analysis port
    uvm_analysis_port # (alu_trx) a_port;

    // ALU Agent
    alu_agent m_agent;

    // Constructor
    // Build Phase
    function void build_phase (uvm_phase phase);
        a_port = new ("a_port", this);
        m_agent = alu_agent::type_id::create ("m_agent", this);
    endfunction // build_phase

    function void connect_phase (uvm_phase phase);
        ...
    endfunction // connect_phase

    function void run_phase (uvm_phase phase);
        ...
    endfunction // run_phase

endclass // alu_env
```

- The environment is responsible for instantiating and connecting:
  - all the agents
  - all the scoreboards
  - all the functional coverage models

Nagesh Loke, ARM
UVM Test

- `uvm_test` is responsible for
  - creating the environment
  - controlling the type of test you want to run
  - providing configuration information to all the components through the environment

Key components of a UVM testbench
10. UVM Basics

**UVM TLM**

- TLM port is a mechanism to transport data or messages
- It is implemented using a SV mailbox mechanism
- It typically carries a whole transaction
- In some cases a broadcast of a transaction is necessary (one-many); this is achieved using an analysis port
- A testbench component implemented using TLM ports is typically more modular and more reusable

**UVM Phasing**

- **build**
  - Create components and allocate memory
  - Hook up components; key step to plumbing
  - Print banners, topology etc.

- **connect**
  - Time consuming tasks
    - Reset the design
    - Configure the design
    - Main test stimulus
    - Stop the stimulus and provide time for checking/draining existing transactions, replays or restarts

- **start_of_simulation**
  - Do end of test checks (all queues empty, all responses received)
  - Provide reporting, pass/fail status
  - Complete the test

- **run**
- **reset**
- **configure**
- **main**
- **shutdown**

- **check**
- **report**
- **final**
What we learned today …

- Discussed what a verification methodology is and the need for it
- Looked at block diagrams with key components in a UVM testbench
- Covered UVM and some of its basic features

Useful pointers

- [https://verificationacademy.com/](https://verificationacademy.com/)
- Recommend watching short videos on UVM introduction on YouTube