CPE 470 - UVM

UVM

Glossary

UVM: Universal Verification Methodology

- **UVM** is a way of designing reusable, abstracted tests
  - Industry standard for complex designs
  - Built on System Verilog
- A UVM testbench is comprised of **components**
  - Components can generate signals, interact with DUT, pass data around, etc.
- UVM is highly object oriented
  - Top-level abstractions encapsulate many components
  - Anything that is a component inherits from component class
- Breaks testbench down into discrete phases

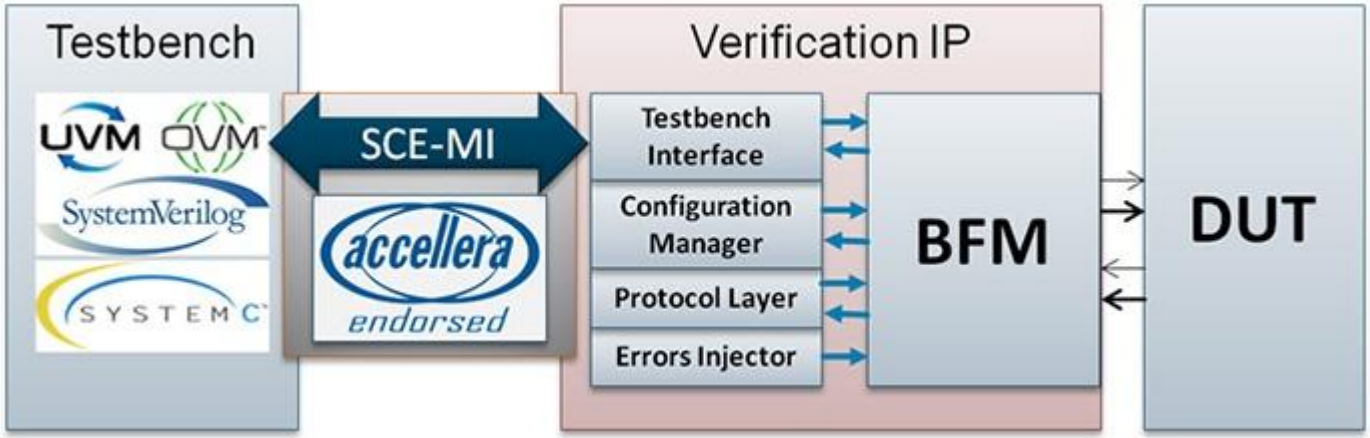
- **PyUVM** is a python implementation of the same classes
  - Integrates with CocoTB
  - Provides further abstraction and teaches industry-style verification

Topology - DUT

Glossary

VIP: Verification IP
BFM: Bus Functional Model, verification model of interfaces used to interact with DUT

- Abstracted **UVM** testbench connects to **DUT** through **VIP**
  - **BFM** abstracts process of writing bus signals
  - Allows easier read/write, error injection, changes of configuration
- **UVM** testbench doesn't want to directly drive individual bus signals
  - **VIP** acts as overall interface, **BFM** directly drives signals



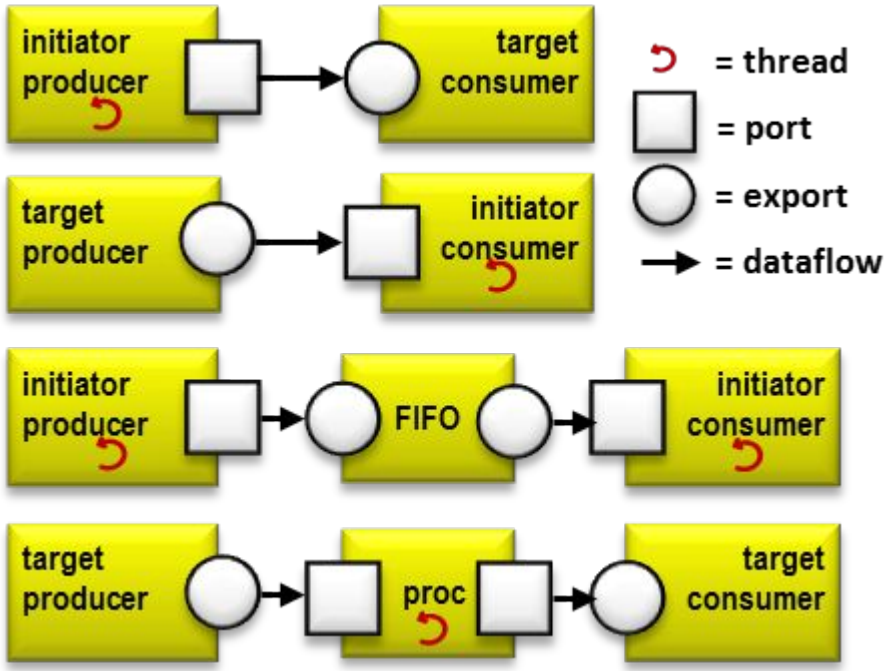
Credit: Accellera

Topology - Testbench

Glossary

TLM: Transaction Level Modeling

- Within the UVM context, elements need to talk to each other
  - Strictly outside of DUT scope
- Use **TLM** to model test flow as an input/output transaction
- Defines port based communication for component interaction
  - **Analysis Ports** exposed by producers to other components
  - **Exports** attached by consumers to receive data



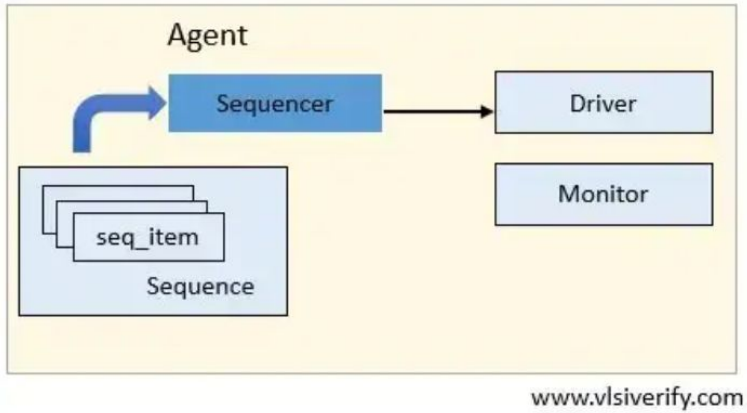
Credit: Siemens

Sequencer

Glossary

Stimulus: specific input conditions to a DUT

- **Sequencer** generates test patterns (**sequences**)
  - Sequences are groups of **stimuli**
  - Randomized data sets
  - Specific Edge Cases
- Sequencer introduces test data to rest of system
  - Sequence uses start/finish item to tell sequencer when ready or done executing



Example Sequences for an ALU

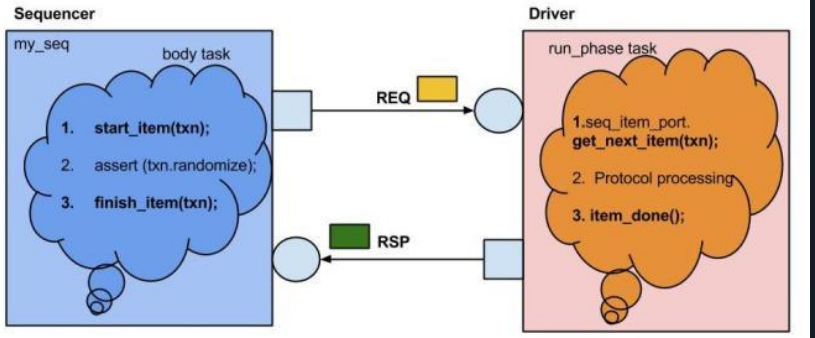
```
class RandomSeq(uvm_sequence):
    async def body(self):
        for op in list(Ops):
            cmd_tr = AluSeqItem("cmd_tr", None, None,
            await self.start_item(cmd_tr)
            cmd_tr.randomize_operands()
            await self.finish_item(cmd_tr)
```

```
class MaxSeq(uvm_sequence):
    async def body(self):
        for op in list(Ops):
            cmd_tr = AluSeqItem("cmd_tr", 0xff, 0xff,
            await self.start_item(cmd_tr)
            await self.finish_item(cmd_tr)
```

Driver

- Driver takes patterns from sequencer and introduces to DUT
  - Interacts with BFM
  - Sends DUT inputs through BFM
  - Reads DUT outputs through BFM
  - Writes outputs to Analysis Port

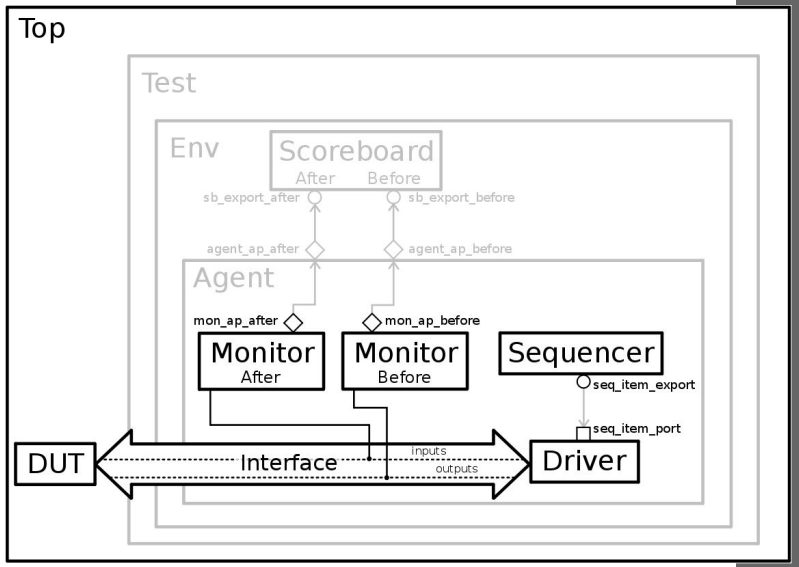
```python
class Driver(uvm_driver):
    def build_phase(self):
        self.ap = uvm_analysis_port("ap", self)

    def start_of_simulation_phase(self):
        self.bfm = TinyAluBfm()

    async def launch_tb(self):
        await self.bfm.reset()
        self.bfm.start_tasks()

    async def run_phase(self):
        await self.launch_tb()
        while True:
            cmd = await self.seq_item_port.get_next_item()
            await self.bfm.send_op(cmd.A, cmd.B, cmd.op)
            result = await self.bfm.get_result()
            self.ap.write(result)
            cmd.result = result
            self.seq_item_port.item_done()
```



Monitor

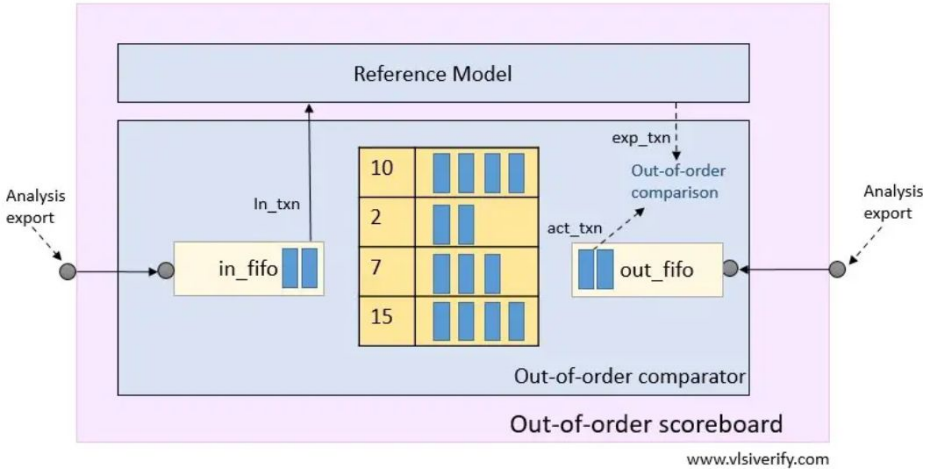
- Listens on the BFM for transactions
- Reports data to Analysis Port
  - Generally read-only, simply reporting back

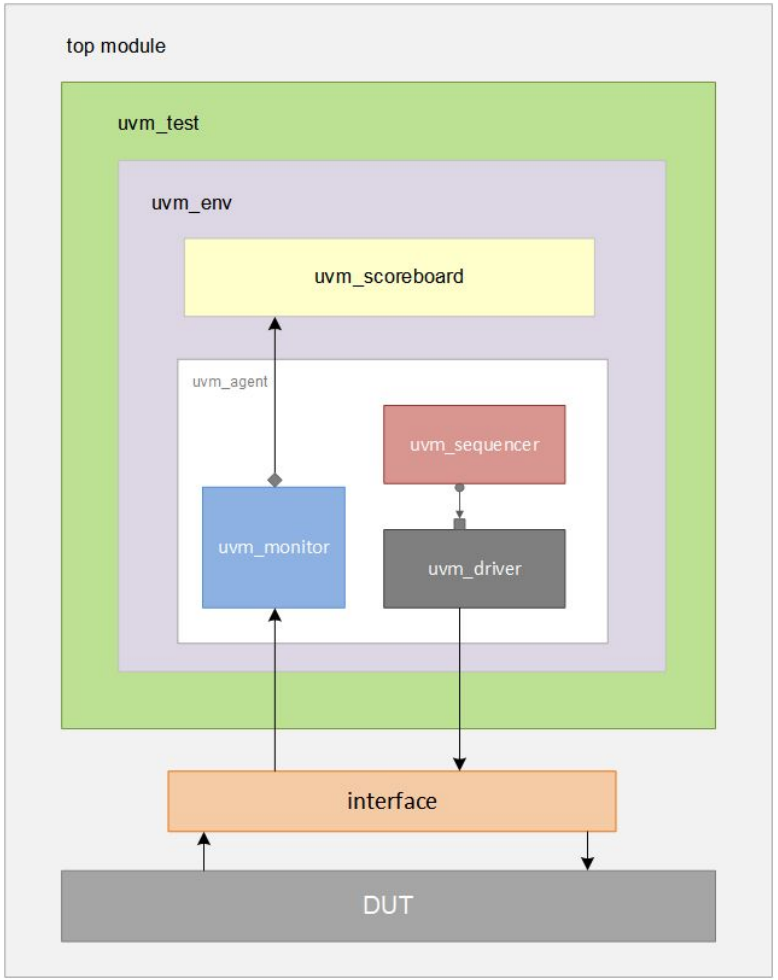
```python
class Monitor(uvm_component):
    def __init__(self, name, parent, method_name):
        super().__init__(name, parent)
        self.method_name = method_name

    def build_phase(self):
        self.ap = uvm_analysis_port("ap", self)
        self.bfm = TinyAluBfm()
        self.get_method = getattr(self.bfm, self.method_name)

    async def run_phase(self):
        while True:
            datum = await self.get_method()
            self.logger.debug(f"MONITORED {datum}")
            self.ap.write(datum)
```



Scoreboard

- Decides whether tests pass or fail
  - Connects to **Monitors** to obtain results
  - Compares results to **Reference Model**





Scoreboard (PyUVM)

```
def build_phase(self):
    self.cmd_fifo = uvm_tlm_analysis_fifo("cmd_fifo", self)
    self.result_fifo = uvm_tlm_analysis_fifo("result_fifo", self)
    self.cmd_get_port = uvm_get_port("cmd_get_port", self)
    self.result_get_port = uvm_get_port("result_get_port", self)
    self.cmd_export = self.cmd_fifo.analysis_export
    self.result_export = self.result_fifo.analysis_export
```

- Create **FIFOs** for commands and results
- Set up FIFO ports and exports

```
def connect_phase(self):
    self.cmd_get_port.connect(self.cmd_fifo.get_export)
    self.result_get_port.connect(self.result_fifo.get_export)
```

- **Connect** FIFOs to monitor ports

Test ALU operations against **Reference Model**

```
(A, B, op_numb) = cmd
op = Ops(op_numb)
predicted_result = alu_prediction(A, B, op)
if predicted_result == actual_result:
    self.logger.info(f"PASSED: 0x{A:02x} {op.name} 0x{B:02x} ="
                     f" 0x{actual_result:04x}")
else:
    self.logger.error(f"FAILED: 0x{A:02x} {op.name} 0x{B:02x} "
                      f"= 0x{actual_result:04x} "
                      f"expected 0x{predicted_result:04x}")
```

Coverage

- Scoreboard told us that our tests pass or fail
  - How do we know if our tests are really covering all the cases we care about?
  - If our tests don't fully cover our use cases, passing them doesn't prove success

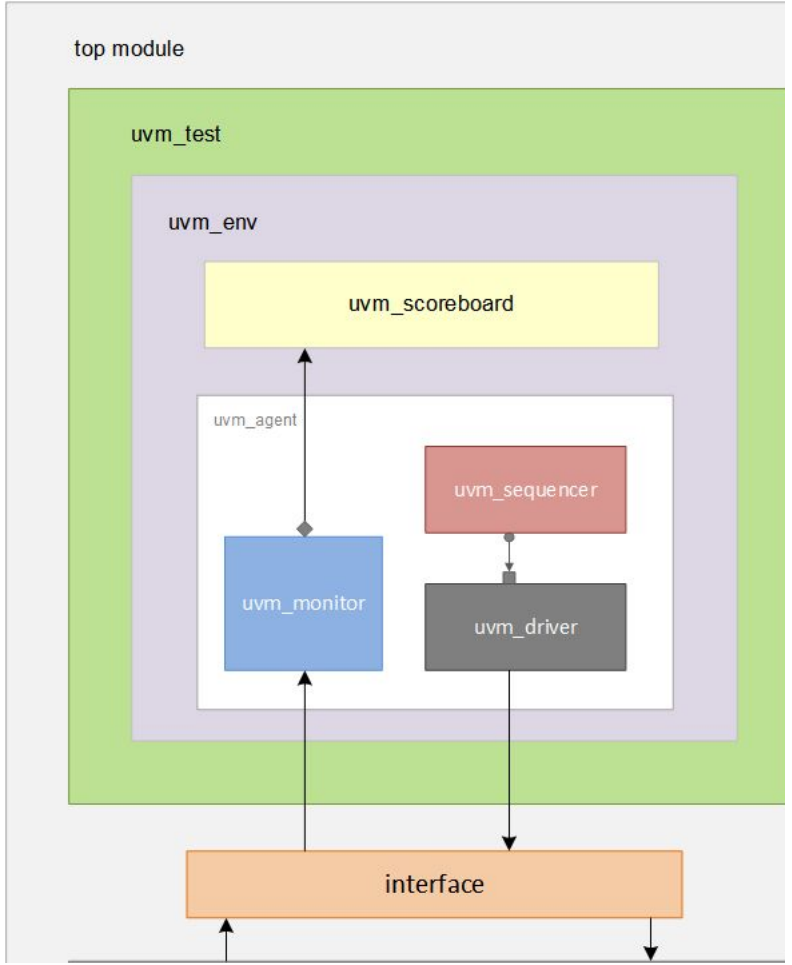
- Coverage makes sure our tests tested everything they were supposed to
  - ALU example: coverage makes sure all Opcodes were tested
- Several Kinds of Coverage
  - Functional Coverage: was every use case tested?
    - User Defined Functionality
  - Code Coverage: was every line of RTL code reached and tested?
    - Automatic

```
class Coverage(uvm_subscriber):

    def end_of_elaboration_phase(self):
        self.cvg = set()

    def write(self, cmd):
        (_, _, op) = cmd
        self.cvg.add(op)

    def report_phase(self):
        try:
            disable_errors = ConfigDB().get(
                self, "", "DISABLE_COVERAGE_ERRORS")
        except UVMConfigItemNotFound:
            disable_errors = False
        if not disable_errors:
            if len(set(Ops) - self.cvg) > 0:
                self.logger.error(
                    f"Functional coverage error. Missed: {set(Ops)-self.cvg}")
                assert False
            else:
                self.logger.info("Covered all operations")
                assert True
```

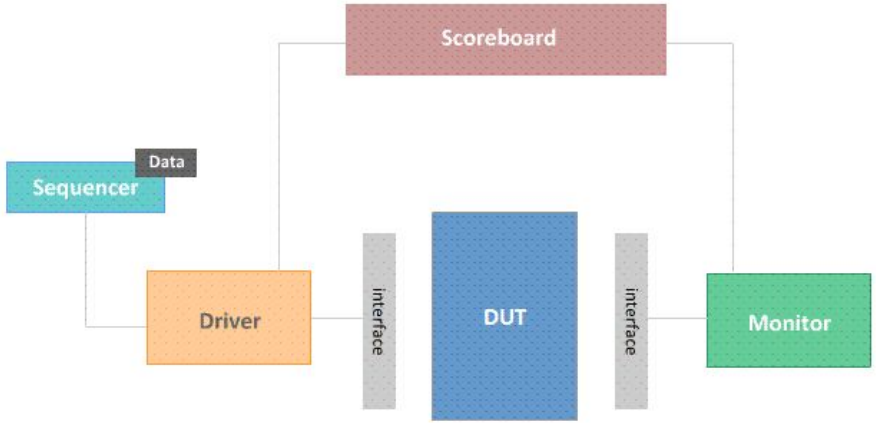
Hierarchy

- **Agent** encapsulates Sequencer, Driver, Monitor
  - Primary components that interact with DUT
  - Sequencer decides what to test
  - Driver executes the test
  - Monitor views the result

- **Environment** contains Agent and Scoreboard
  - Semi-Configurable top level that can be reused across different tests
- **Test** contains an environment
  - One test is NOT reusable, should be specific
  - Implements a specific environment

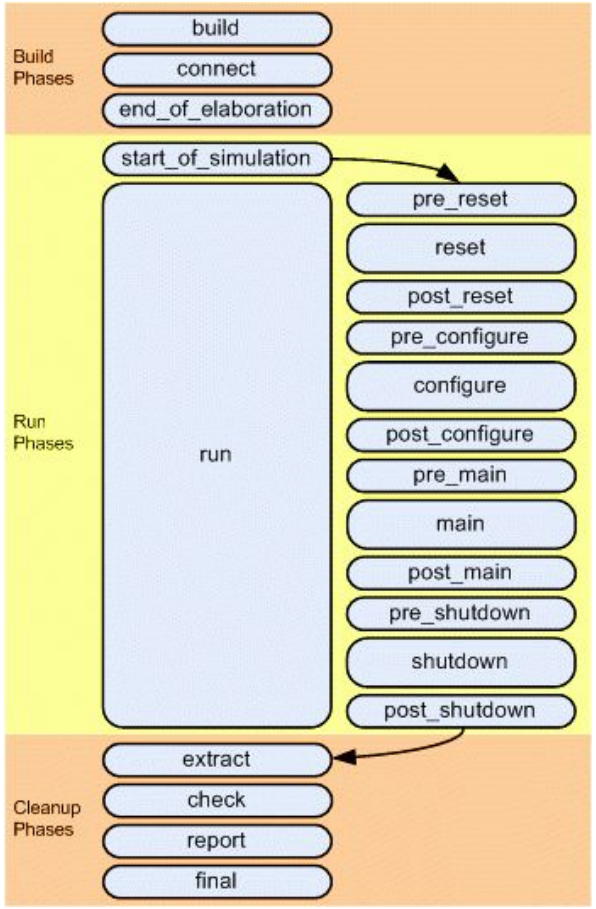


System Verilog TestBench as we know it..



Credit: ChipVerify

UVM Phases



- Test Benches are often made up of similar steps across projects:
  - Have to reset DUT
  - Flash with known configuration
    - RISC-V core needs instructions
    - Accelerator needs config
  - Introduce test inputs
  - Check results of test
  - Wait for all tests to propagate through

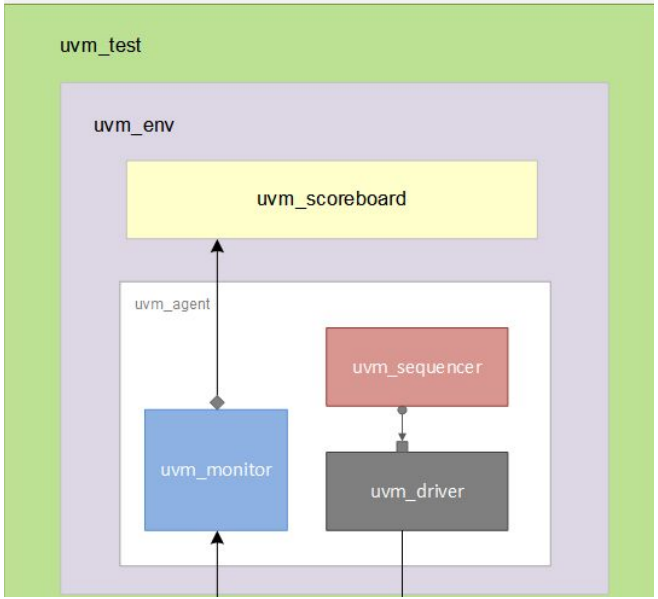
- UVM made up of discrete test phases where each of these things can happen
  - Flexibility for designer to order necessary operations
  - Rigid enough to define where certain steps should happen

Build Phase

1. Build
   a. Instantiate all components, top-down
2. Connect
   a. Connect components using TLM Ports
3. End of Elaboration
   a. Fine tune configuration, breakpoints, print topology

Build Phase all happens before simulation.
Completely setup of the testbench itself

Instantiate & Connect All UVM Components

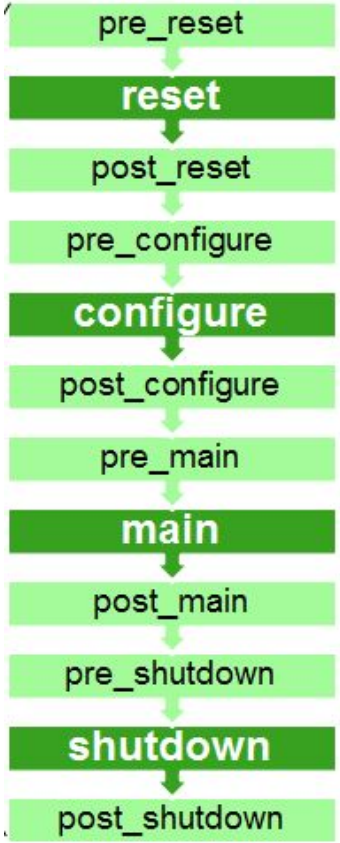


Run Phase

1. Start Simulation
   a. Start simulator
2. Reset
   a. Send reset signals to DUT, known states all around
3. Configure
   a. Program DUT, flash its memories, load data
4. Main
   a. Apply sequences of stimulus to DUT and check results
5. Shutdown
   a. Wait for all data to finish propagating, read final status registers

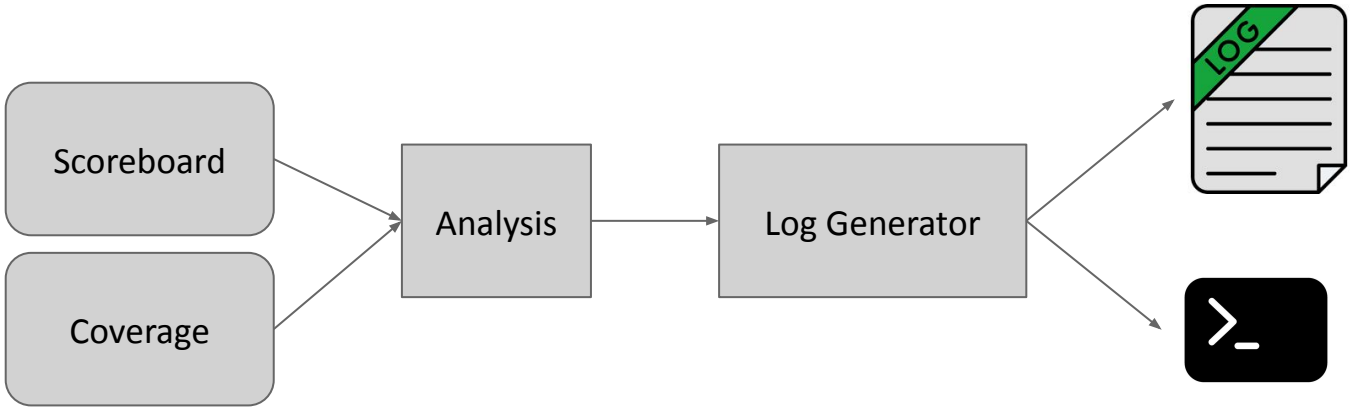
During run phase, each step has a pre- and post- step.

- Adds flexibility on where events should occur



Cleanup Phase

1. Extract
   a. Retrieve scoreboard and coverage information
2. Check
   a. Identify errors and check for unexpected conditions
3. Report
   a. Display or print results, write log files
4. Final
   a. User-defined final actions



Phase Summary

UVM Common Phases	The common phases are the set of function and task phases that all uvm_components execute together.
uvm_build_phase	Create and configure of testbench structure
uvm_connect_phase	Establish cross-component connections.
uvm_end_of_elaboration_phase	Fine-tune the testbench.
uvm_start_of_simulation_phase	Get ready for DUT to be simulated.
uvm_run_phase	Stimulate the DUT.
uvm_extract_phase	Extract data from different points of the verification environment.
uvm_check_phase	Check for any unexpected conditions in the verification environment.
uvm_report_phase	Report results of the test.
uvm_final_phase	Tie up loose ends.

References

- https://vlsiverify.com/uvm/uvm-phases/
- https://www.chipverify.com/tutorials/uvm
- https://learnuvmverification.com/index.php/2016/04/29/uvm-phasing/
- https://asicwhale.github.io/2018/07/09/201807-2018-07-09-uvm-env/
- https://accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf
- https://mybrushwithasic.blogspot.com/2013/10/dear-readers-first-let-me-wish-you-all.html
- https://colorlesscube.com/uvm-guide-for-beginners/chapter-6-monitor/