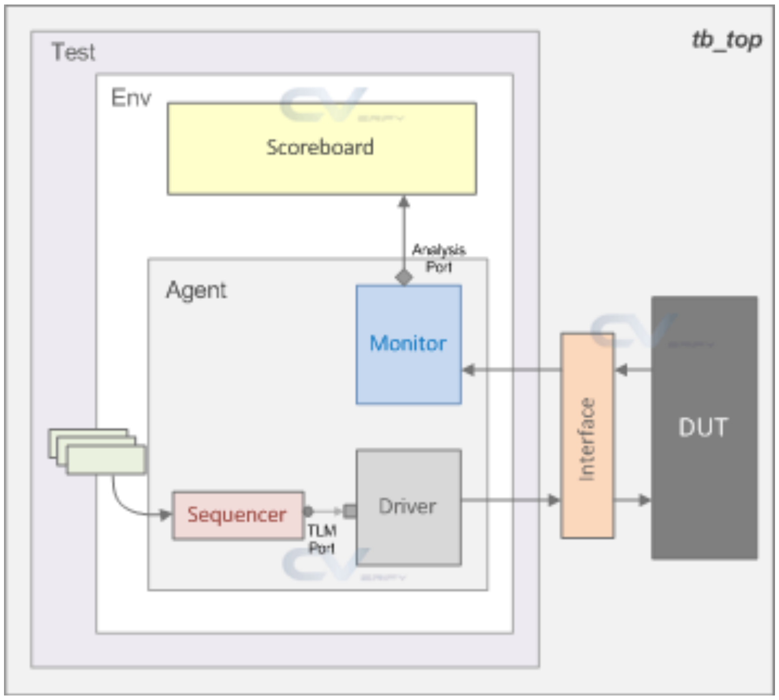
UVM Testbench Top

two kinds of reset - hardware and software. A software reset is typically done through a register model and would be separate from a *reset agent*. Hardware resets include assertion of the system reset pin for a given duration or follow a certain sequence of events before the actual reset is applied.

What is testbench top module ?

All verification components, interfaces and DUT are instantiated in a *top level* `module` called testbench. It is a static container to hold everything required to be simulated and becomes the *root* node in the hierarchy. This is usually named tb or tb_top although it can assume any other name.



Simulators typically need to know the top level module so that it can analyze components within the top module and elaborate the design hierarchy.

Testbench Top Example

The example below details the elements inside the top module **tb_top**.

```
module tb_top;
   import uvm_pkg::*;

   // Complex testbenches will have multiple clocks and hence multiple clock
   // generator modules that will be instantiated elsewhere
   // For simple designs, it can be put into testbench top
   bit clk;
   always #10 clk <= ~clk;


   // Instantiate the Interface and pass it to Design
   dut_if         dut_if1  (clk);
   dut_wrapper    dut_wr0  (._if (dut_if1));



   // At start of simulation, set the interface handle as a config object in UVM
   // database. This IF handle can be retrieved in the test using the get() method
   // run_test () accepts the test name as argument. In this case, base_test will
   // be run for simulation
   initial begin
      uvm_config_db #(virtual dut_if)::set (null, "uvm_test_top", "dut_if",
dut_if1);
      run_test ("base_test");
   end

   // Multiple EDA tools have different system task calls to specify and dump
waveform
   // in a given format or path. Some do not need anything to be placed in the
testbench
   // top module. Lets just dump a very generic waveform dump file in *.vcd format
   initial begin
               $dumpvars;
               $dumpfile("dump.vcd");
   end

endmodule
```

Note the following :

- tb_top is a module and is a static container to hold everything else
- It is required to import `uvm_pkg` in order to use UVM constructs in this module
- Clock is generated in the testbench and passed to the interface handle dut_if1
- The interface is set as an object in `uvm_config_db` via `set` and will be retrieved in the test class using `get` methods
- The test is invoked by `run_test` method which accepts name of the test class base_test as an argument
- Call waveform dump tasks if required

Clock generation

A real design may have digital blocks that operate on multiple clock frequencies and hence the testbench would need to generate multiple clocks and provide as an input to the design. Hence clock generation may not be as simple as an `always` block shown in the example above. In order to test different functionalities of the design, many clock parameters such as frequency, duty cycle and phase may need to be dynamically updated and the testbench would need infrastructure to support such dynamic operations.

```
// Module level clock generation
module clk_main (...);

	// Code for clock generation

endmodule

// Instantiated and connected to an interface
module tb_top;
	bit clk_main_out;

	clk_main u_clk_main_0 ( .out (clk_main_out),
	                                                    ...
	                                              );

	dut_if  dut_if0 ( clk_main_out, ...);
endmodule
```

The approach shown above may not be scalable and need to be driven from the testbench using hierarchical signal paths since they are instantiated as modules. A better UVM alternative is to create an agent for the clock so that it can be easily controlled from sequence and tests using agent configuration objects.

```
class clk_agent extends uvm_agent;

	clk_cfg 	m_clk_cfg;

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

		// Get clk_cfg object and build this agent accordingly
	endfunction

endclass

class base_test extends uvm_test;

	clk_cfg 	m_clk_cfg;

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

```
                m_clk_cfg.m_freq = 500;
        endfunction
endclass
```

Reset Generation

In a similar way, a reset agent can be developed to handle all reset requests. In many systems, there are two kinds of reset - hardware and software. A software reset is typically done through a register model and would be separate from a *reset agent*. Hardware resets include assertion of the system reset pin for a given duration or follow a certain sequence of events before the actual reset is applied. All such scenarios can be handled separately using this reset agent which needs a handle to the reset interface.

```
class reset_agent extends uvm_agent;

        reset_cfg       m_reset_cfg;

        // Rest of the code
endclass

class my_sequence extends uvm_sequence;

        virtual task body();
                hw_reset_seq    m_hw_reset_seq;

                // Call the required kind of reset sequence in test scenario
                m_hw_reset_seq.start(p_sequencer);
        endtask
endclass
```

Creation of internal tap points

Some testbench components may rely on tapping internal nets in the design to either force or sample values to test certain features. These internal nets may need to be assigned to a different value based on input stimuli and can be done so in the top level testbench module. Such signals can be tied to a generic interface and be driven from another agent.

```
interface gen_if;
        logic [99:0]    signals;        // General 100-bit wide vector

endinterface

module tb_top;
        gen_if  u_if0 ();
```

```
		des 	u_des 	( ... );

		// Assign an internal net to a generic interface signal
		assign u_if0.signals[23] = u_des.u_xyz.u_abc.status;

	endmodule
```