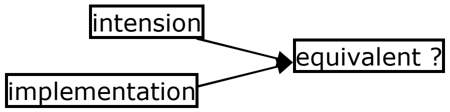
Verification Methodology

Ming-Hwa Wang, Ph.D.
COEN 207 SoC (System-on-Chip) Verification
Department of Computer Engineering
Santa Clara University

Topics
- Vision and goals
- Strategy
- Verification environment
- Verification plan and execution strategy
- Automation
- Resources

Vision and Goals

The definition of verification
- Verify the RTL implementation is the same as the intention (which is not the same as the golden model or the specifications)

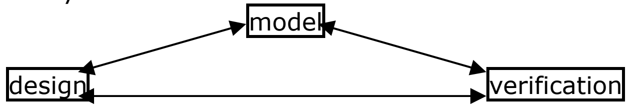


- Verification = infrastructure/methodology + product understanding + monitoring
- Verification crisis: complexity grows exponentially, re-spin cost, miss time-to-mark, lost customer, etc.

The Vision
- Build a independent verification team in order to verify that the RTL implementation is the same as the intension, the model, or the specifications
- Provide a consistent and generic verification environment to support multiple SoC projects and reuse among design teams
- Automate most of verification flow to increase productivity, coverage, and quality of the verification tasks
- Work in parallel and aligned with HW team

Goals to Achieve
- Basic goals (verification only)
  - Make sure the RTL implementation is "almost" the same as the golden model
  - Achieve predefined coverage on time
- Enhanced goals (verification + modeling)
  - Success on the first tapeout or reduce the number of re-spin to very minimum
    - Most companies can justify at most 1 re-spin for new design (time-to-market/cost: 2 tapeouts)
    - Most integrations of proven IPs can be done at the first tapeout
  - Shorten time to market

Responsibilities
Responsibilities and cooperation among 3 teams
- Architecture team responsibility
  - has to make sure the model is a correct executable specification and covers all functionalities of the design
- RTL design team responsibility
  - Doing RTL implementation according the model or the specifications
- Verification team responsibility
  - Verify the RTL is the same as the model



Level of Verification
- Block-level verification, integration verification, chip-level verification, and system-level verification

Design vs. Verification
- Separation of Design and Verification
  - Designers do verification on their own design tend to neglect/miss non-working cases, or misinterpret the specs
  - Intentional make two different views (one from designer, another from verification engineer) on the same specs, if inconsistency exists, need to find out why (specs wrong, implementation wrong, or verification wrong) and fix it
  - Can use designers to do verification for other blocks not designed by the same persons
- Ratio of the number of verification engineers to the number of design engineers
  - At least 1:1 at block/unit level
  - More verification engineers needed on sub-chip and whole-chip verification (including coverage)
  - Need extra verification methodology/infrastructure/tool engineers
  - Total ratio at least 2:1, more companies have more than 2:1 ratio now due to complexity grows exponentially

Strategy

Basic Verification Strategy (verification only)

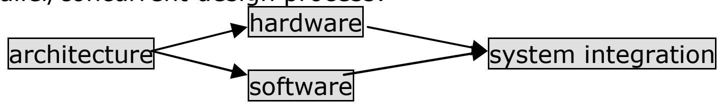
- Use SystemVerilog and OVM to build a consistent verification environment, and use scripts to provide an generic and automated verification flow
- Use assertions to *quickly* increase the quality of the design
- Use coverage and constrained-random test generation to achieve high coverage, and the rest corner cases verified by direct tests
- Use DPI to integrate high level model efficiently
- Use skeleton testbench and test APIs (transactions, sequences, etc.) to increase productivity
- Use revision control, configuration, regression and bug tracking to repeat problem and monitor progress of fixes
- Use LSF to fully utilize computing/license resources, and use parallel processing, hardware acceleration, FPGA/emulation to speed up verification

Enhanced strategy (model-driven methodology)

- Use model-driven methodology for architecture, HW and SW concurrent development, verification, and validation
- Use performance modeling for architecture exploring, performance analysis, and HW/SW partitioning
- Use functional or TLM model as the golden reference for HW/SW development and verification
- Use mix and match flow (with co-simulation) to gain visibility while preserving high run-time efficiency
- Break the design/verification/backend dependencies: parallel design and verification efforts, use AFE models, hack sdf/RTL for GLS, etc.

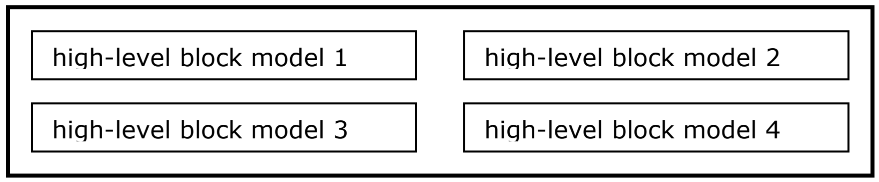
Time to Market

- conventional iterative serial design process:
  - architecture → hardware → software → system integration
- parallel/concurrent design process:

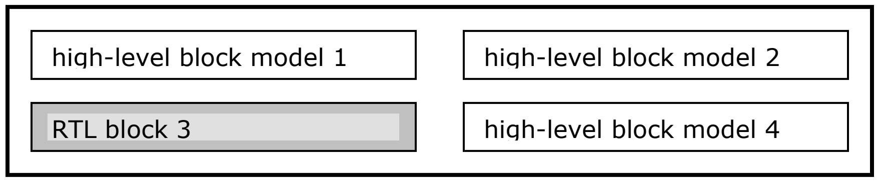


Example mix and match models

- software model: e.g., all high-level models

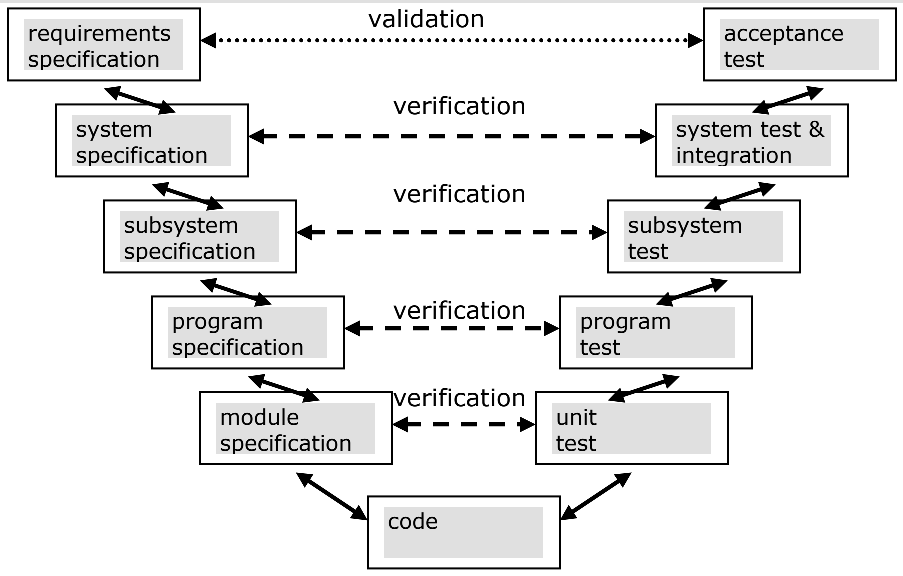


- hybrid model: e.g., replace one block model by RTL

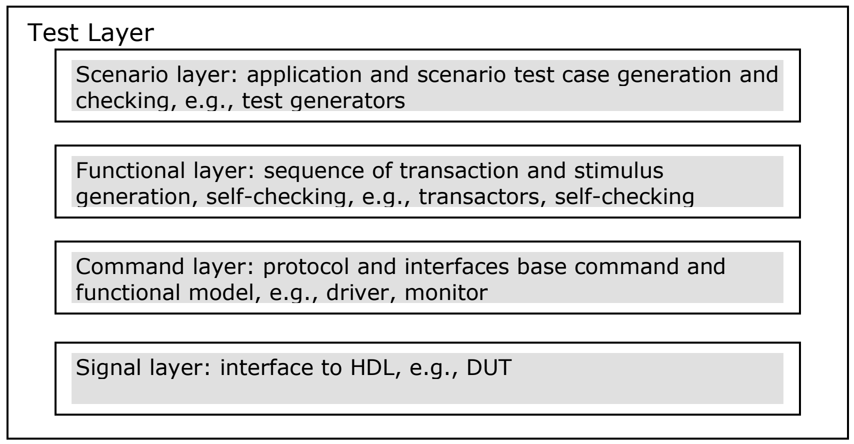


Verification Environment

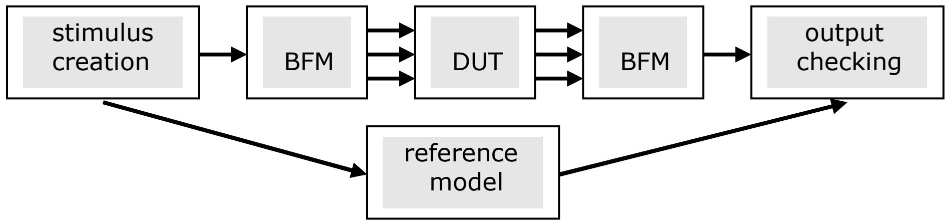
The V Model for Verification and Validation



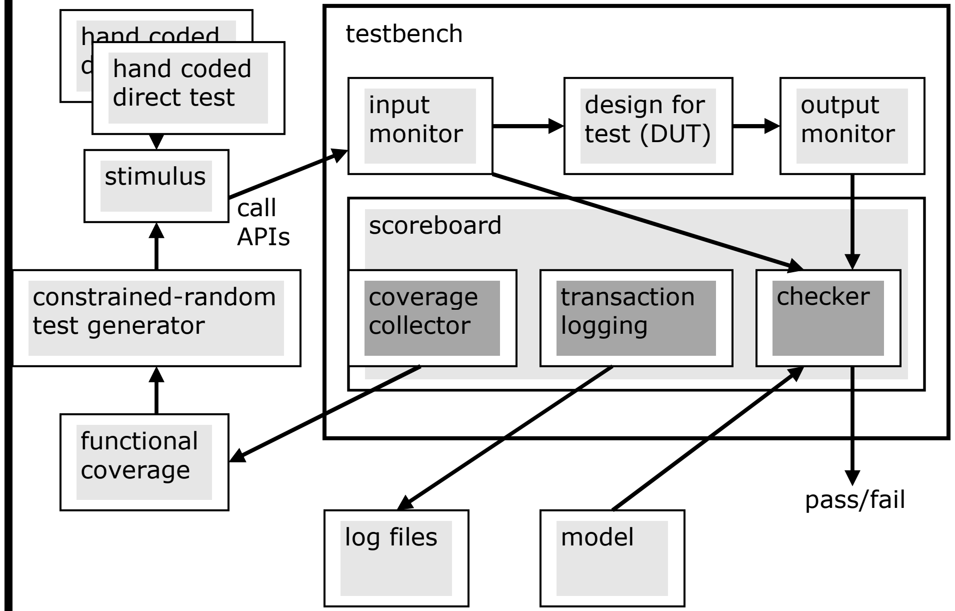
Test Layers


Test Layer
Scenario layer: application and scenario test case generation and checking, e.g., test generators
Functional layer: sequence of transaction and stimulus generation, self-checking, e.g., transactors, self-checking
Command layer: protocol and interfaces base command and functional model, e.g., driver, monitor
Signal layer: interface to HDL, e.g., DUT

Typical Verilog Testbench

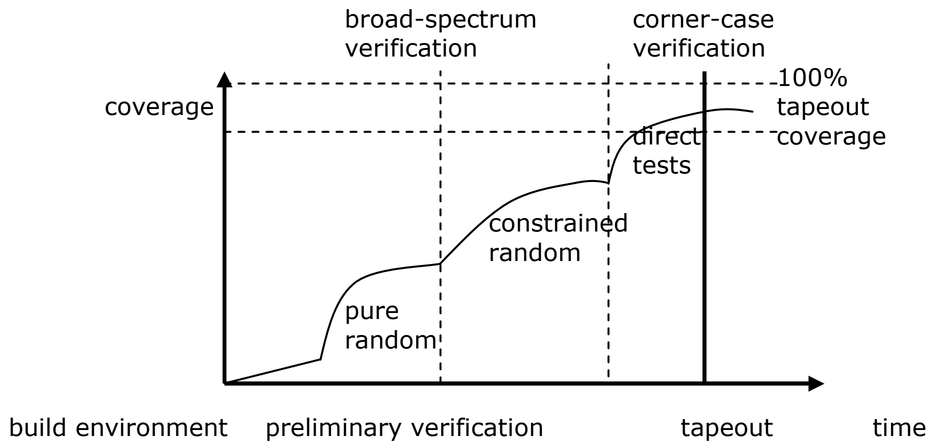


Coverage-Driven Verification Environment



Coverage-Driven Verification

- CDV combines the following aspects to reduce the time spent verifying a design
  - Automatic stimulus generation
  - Self-checking testbenches
  - Coverage metrics
- An iterative process
  - Start with simple directed tests to bring up the design
  - Fully random test generation and run simulations
  - Collect coverage statistics and adjust constrained-random test generation to generate corner cases
  - Run simulation and goto step 2, until the iteration doesn't increase enough coverage
  - Hand coded corner tests not covered by previously steps
- Ways to implement CDV
  - Tools from EDA vendors – vendor lock-in?
    - Vera language and VMM
    - Cadence Specman e language and its CDV environment
    - Mentor Graphics InFact
  - Verilog testbench with script to automate constrained random stimulus generation, high-level to low-level test translation, Verilog monitors and scoreboards for checking
  - Open source free C++ environment - Teal
  - SystemC and its verification library (SCV)
  - SystemVerilog and OVM
- Advantages
  - Fully automated process – testbench automation
  - Cost and benefit tradeoff made by upfront coverage goals
  - Reusable



Verification Description Languages (VDL)

- Vera
  - A design verification language
  - need to build the infrastructure
- Specman
  - A coverage-driven verification environment
  - long learning curves for the "e" language
- SystemC and its verification library (SCV)
  - SystemC is based on C++ and moved from high level language downward to support functional modeling (TLM), verification (SCV), and even implementation (not encouraged though)
  - Difficult to be accepted by design community
- SystemVerilog
  - Based on Verilog and grows upward to support system level modeling and emphasizes on verification
  - A hardware design and verification language (HDVL)

Verilog or Object-Oriented

- Designers prefer Verilog to object-oriented language
  - Pros – designers can jump in easily to help on verification

    - Cons – Verilog is a hardware (HDL), but not good for verification, which needs verification description language (VDL)
- Object-oriented environment
    - Pros – good for complicated projects by using layered verification environment, which facilitates reusability
    - Cons – tremendous training cycle on designers and need longer up-front development time

SystemVerilog Randomization

```
class Bus_trans;
    static int next_id;
    const int id;
    rand bit[2:0] dir;  // read/write
    rand bit[15:0] addr;
    rand bit[31:0] data;
    constraint addrIsEven { addr[0] == 0; } // word alignment
    function new;
        id = next_id;
    endfunction : new
    function print;
        …
    endfunction : print
endclass : Bus_trans

Bus_trans rR;
repeat (3) begin
    tR = new;
    void` (tR.randomize( ) );
    tR.print( );
end
```

- random variables: rand and randc
- constraint mechanisms
    - call "randomize( ) with"
    - distribution/weight control: dist, randcase .. endcase
    - conditional randomization: if .. else, implication (->), case .. endcase
    - repetition: repeat, foreach
    - interleaving: rand join
    - scoping: static
    - ordering: solve .. before
    - layering: inheritance and override
- examples using randomization with
    - unconstraint random values
      ```
      std::randomization( );
      ```
    - constraint random values
      ```
      randomize(addr) with { addr > 8`h0f; };
      randomize(addr) with { a > b; b[0] == 0; };
      ```
    - constraint expression in class
      ```
      rand int i;
      constraint i_small_even { i inside { 0, 2, 4, 6, 8 }; }
      ```
    - random value distribution in class
      ```
      rand int i;
      constraint c { i dist { [0:2] := 1, 3 := 2; 4 := 3 }; }
      ```
      // probability of : 0 is 1/8, 1 is 1/8, 2 is 1/8, 3 is 1/4, 4 is 3/8

Code Coverage

- Line/block/segment coverage
    - If all lines, blocks, and segments are reached
- Branch coverage
    - If all branches are reached
- Expression coverage
    - If all possible legal Boolean values of the expression are reached
- Toggle coverage
    - Which bits in RTL are toggled – for power analysis
- FSM coverage
    - If all states and possible transitions are reached
- Path coverage
    - If all possible paths/traces are reached

Functional Coverage

- A coverage point is a Boolean function f(x1, …, xn) of a given set of design variables x1, …, xn representing a specific valuation on the set. Variables are sampled, if they have the valuation as specified, the function is said to evaluate to TRUE. Functional coverage is expressed as the fraction of the functions evaluating TRUE out of all such defined functions.
- Functional coverage is user-specified in order to tie the verification environment to the design intent or functionality
- Coverage metric to be used is determined by the design type: control dominated, datapath, interactions, and scope (block vs. system)
- Measures how much of the design specification, as enumerated by features in the test plan, has been exercised
    - Interesting scenarios
    - Corner cases
    - Specification invariants
    - Other applicable design conditions
- SystemVerilog functional coverage constructs:
    - coverage group
    - coverpoint and cross coverpoint (w/ optional iff( ))
    - coverage bins
        - values, ranges ":", list ",", default
        - transition "=>" and sequence: consecutive repetition "*", nonconsecutive repetition "=" or "->", default sequence
        - wildcard bins, ignore_bins, illegal_bins, binof, intersect, "!", "&&", "||", open range "$"
    - coverage options: weight (default 1), goal (default 90%), name, comments, at least (default 1), detect overlap (default false), auto_bin_max (default 64), cross_num_print_missing (default 0), per_instance (default false)
    - coverage type options: weight (default 1), goal (default 100%), comments, strobe (default false)

- Functional coverage control
- Predefined methods: sample( ), get_coverage( ), get_inst_cover( ), set_inst_name( ), start( ), stop( )
- Predefined system tasks and functions: $set_coverage_db_name( ), $load_coverage_db( ), $get_coverage( )
- Functional coverage calculation
- Coverage of a coverage group: $C_g = (\Sigma_i W_i * C_i) / \Sigma_i W_i$
- Coverage of a cross item: $C_i = |\text{bins}_{\text{covered}}| / (B_c + B_u)$, where $B_c = (\Pi_j B_j) - B_b$

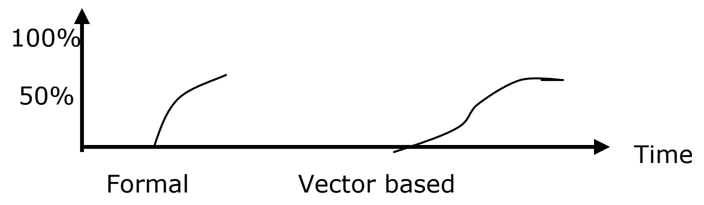
```
class Instruction;
    bit [2:0] opcode;
    bit [1:0] mode;
    bit [10:0] data;
    covergroup cg @ (posedge clk);
        coverpoint opcode;
        coverpoint mode {
            bins valid [ ] = { 2`h01, 2`h10 };      // one-hot
            illegal_bins invalid = default;
        }
        coverpoint data;
        cross opcode, mode;
    endgroup
endclass : Instruction

initial begin
    repeat (100) @ (posedge clk) begin
        cg.sample;
        cov = cg.get_coverage;
        if ( cov > 90.0 ) cg.stop;
    end
end
```

Self-Checking

- Since random test generation can generate stimulus randomly, an automatic monitor and self-checking mechanism is necessary to check if the test is pass/fail
- Checking should be done
  - Temporal/timing check by assertion – correct sequence
  - Data value check by reference model – valid result
- Checkers ensure function correctness by using assertions or procedure code

Formal Method

- Vector based dynamic verification needs testbench and test cases, thus takes significant initial time
- Static formal verification is about applying "constraints" on inputs by "assume", "assert" on behavior by assertions and check if assertions are "triggered" (as well as functional coverage), thus can be started as soon as you have specs in hand



Assertion based verification (ABV)

ABV should emphasis on block level (by designers) with complex temporal properties instead of system level (by verifiers).

The structured approach to add assertions to a design:

1. Identify assertion candidates – module ports, control signal, etc.
   - assertion metrics: total number of assertion candidate, percentage of high/med/low assertions
2. Code assertions
   - challenges: user must abstract and code an accurate property, SVA language is not intuitive, temporal expression operators are complex to visualize, skills required are difficult and costly to develop, need to prevent evaluation of open-ended delay expression
3. Debug assertions
   - challenges: debugging an unproven assertion is a major time sink, user must develop at testbench to validate the assertion
4. Implement assertion control
   - control knobs: severity and level, messaging to test bench, reuse: find files, documentation
   - challenges: user must package and control assertions in a consistent manner, several types of assertion controls required (disabling assertions, changing severity levels, change coverage levels, integration with testbench component), assertion expression visualization

Assertion and Property Checking

- Assertion languages
  - Property specification language (PSL) - based on IBM Sugar
  - Synopsys OpenVera assertions (OVA)
  - SystemC Verification (SCV)
  - SystemVerilog assertion (SVA)
- Property
  - Boolean-valued expression
  - Static property checking
  - Dynamic property checking
- Assertions
  - Directive to a verification tool that the tool should attempt to prove/assume/count a given property
- Slow running time, applicable to block/unit level verification, but may not for full chip verification
- Rule of thumb: 80% RTL code, 20% assertion; to make it precise, use assertion coverage

Coverage

- Extreme programming:
- No implementation will be performed until a testcase exists for the planned implementation
- Nobody programs or tests alone! Always take a buddy!

Property vs. Assertion

- A property is a boolean expression built form:
  - (System)Verilog expression
  - Property operators
  - Sequences

```
property nonZero;
               (address > 0 );
endproperty
```

- An assertion is a statement that a particular property is required to be true, and thus a tool should attempt to prove it

```
assert property (nonZero);
```

Properties

- A property is a fact about your design
- Where to put properties?
  - Embedded in the HDL code
  - Properties in a separate file, then bind it to modules
- Properties are used to write
  - Simulation checkers
  - Constraints defining legal simulation input vectors
  - Assertions to be proved by static formal property checkers
  - Assumptions made by static formal tools
  - Functional coverage points

Assertions

- Capture the design intent more formally and find specification error earlier
- Find more bugs and source of the bugs faster
- Static property checker
- Encourage measurement of function coverage and assertion coverage
- Re-use checks throughout life-cycle, strength regression testing
- Assertion failure severity
  - $info, $warning, $error (the default), $fatal

Immediate Assertions

- Immediate assertions = instructions to a simulator
- Immediate assertions can be in class
- Syntax: **assert (** expression **)** [pass_statement] [**else** fail_statement]

```
always @ (posedge clk) begin:checkResults
       assert ( output == expected ) okCount++;
       else begin
              $error("Output is incorrect");
              errCount++;
       end
end
```

Concurrent assertions

- Concurrent assertions = instruction to verification tools
  - Assert property (aProperty) actionBlock;
  - Cover property (aProperty) statement;
  - Assume property (aProperty);
- Concurrent assertions can't be in classes.
- Implication (**|->**, **|=>**), **$rose( )**, **$fell( )**, **$past( )**, sequences (**##**), concatenation/repetition, operators (**or**, **and**, **intersect**, **throughout**, **within**)

```
assert property (
       @(posedge clk) a ##1 b |-> d ##1 e
);
```

Binding properties to scopes or instances

- Example of binding two modules

```
module cpu (a, b, c);
    <RTL codes>
endmodule
module cpu_props (a, b, c);
    <assertion properties>
endmodule
```

- Bind cpu cup_props cpu_rules_1 (a, b, c);
  - cpu and cpu_props are the module names
  - cpu_rules_1 is cpu_props instance name
  - ports(a, b, c) gets bound to the signals (a, b, c) of the module cpu
  - every instance of cpu gets the properties

Direct Programming Interface (DPI)

- Two separate layers: the SystemVerilog layer and the foreign language layer
- Both sides of DPI are fully isolated
  - No language compiler is required to analyze the source code in the other language
  - The memory spaces owned and allocated by foreign code and SystemVerilog code are disjoined
- SystemVerilog defines a foreign language only for the C programming language (use 'extern "C" ' to mix C++ code in C DPI
- Pure and context
- Name resolution: global symbol and escape sequence

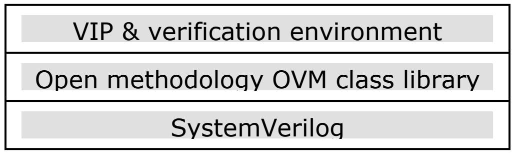
An Unified SystemVerilog Environment

- An unified SystemVerilog environment
  - Use SystemVerilog to do functional/reference modeling and RTL design
  - Use SystemVerilog to write testbenches
  - Use SystemVerilog assertions to check design intents
  - Use SystemVerilog to do coverage-driven verification

- Use SystemVerilog DPI to integrate high level models
- Trend
  - SystemVerilog is the trend in design verification
- Concerns
  - Not all features of SystemVerilog are supported by EDA tools
  - Need more licenses for SystemVerilog
  - Need SystemVerilog training

Open Verification Methodology (OVM)

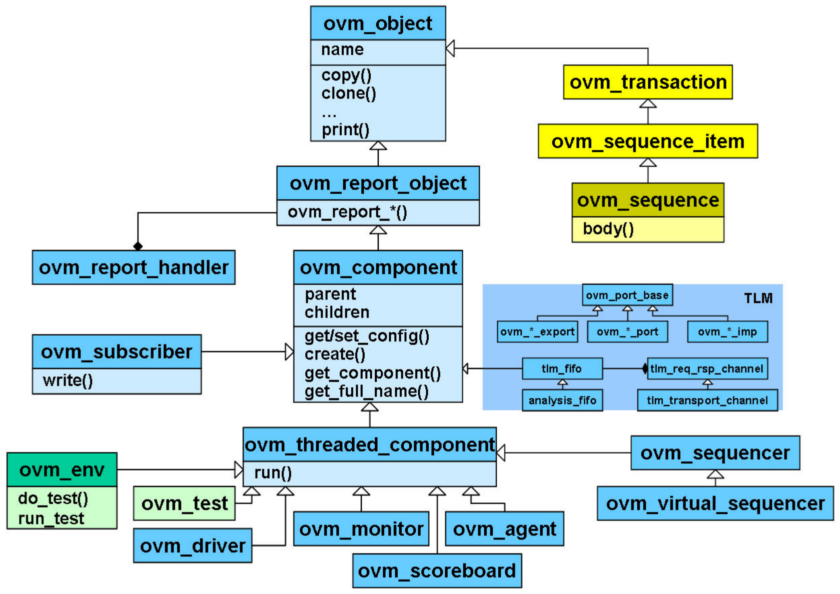
- Open, interoperable, SV verification methodology
- Transaction-level modeling (TLM) interfaces for reusability
- OVM library with built-in automation for productivity
  - TLM communication
  - Phasing and execution management
  - Testbench/environment customization
  - Top-down configuration scheme for scalability
  - Sequential layered stimulus and test write interface
  - Common message reporting and formatting interface



VIP & verification environment
Open methodology OVM class library
SystemVerilog

- OVM enable multi-language verification – reduce risk and unifies SoC verification teams
  - Single methodology for e, SystemC and SystemVerilog
  - Unifies VIP for reuse
  - Choose best language for application and group skills
  - Frees companies to choose best verification solution
- Pros
  - The OVM provides a library of base classes (on top of SystemVerilog) that allows users to create modular, reusable verification environments in which components talk to each other via standard transaction-level modeling (TLM) interfaces
  - Multi-language VIP plug and play in SV, e, or SC
  - Built-in automation and powerful configuration scheme – easy to integrate and configure verification components into a powerful testbench
  - Common message reporting and formatting interfaces for transaction automation
- Cons
  - Training to facilitate the learning curve
  - Design engineers' acceptance chance is moderate
  - Development time and resource requirement

(Partial) OVM Class Hierarchy



TLM

- TLM IP design and verification flow – moving verification up to where is more efficient, optimizing efforts with a metric driven verification plan
- Comprehensive TLM-driven design and verification – enables the next leap in design productivity
  1. Based on industry standards – systemC and OVM
  2. Move golden model up to TLM level – easier to write/read/debug, links virtual platform to implementation flow
  3. Metric-driven verification reduce overall effort – verify more functionality at faster TLM speeds, reuse testbench to regress lower RTL
  4. Enables multi-language verification
- TLM design and verification advantages – as reported by customers
  1. ~3x more efficient/compact designs
  2. 5x-10x faster simulation/debug
  3. Equal/better QoR than manual
  4. 10x+ productivity

Reusability

- Transaction-level modeling (TLM)
  - Blocking interfaces: put( ), get( ), peek( )
  - Non-blocking interfaces: try_put( ), try_get( ), try_peek( )
  - Port/export connection: one-to-one mapping
  - Analysis communication - write( ): one-to-one or one-to-many mapping
  - SystemVerilog mailbox: one-to-one, one-to-many, or many-to-many mapping

- Top-down or hierarchical configuration
  - register data members using `ovm_*_utils macros
  - top-down or hierarchical configurations
    - set_config_int(<instance path>, <attribute>, <int>);
    - set_config_string(<instance path>, <attribute>, <string>);
    - set_config_object(<instance path>, <attribute>, <object>);
    - set_type_override_by_type(<type>, <type>);
    - set_inst_override_by_type(<instance path>, <type>, <type>);
  - wildcard can be used in <instance path>
  - command line arguments
  - +OVM_TESTNAME <extended test class>
  - run multiple tests w/o having to recompile or re-elaborate
- Utility functions
  - Register with `ovm_object_utils and `ovm_field_*
  - Utility functions: randomize (by SV), create, print, clone, compare, copy, pack, get_type_name, record
  - Flags (concatenating with +):
    - OVM_ALL_ON
      - OVM_COPY/OVM_NOCOPY
      - OVM_COMPARE/OVM_NOCOMPARE
      - OVM_PRINT/OVM_NOPRINT
      - OVM_DEEP/OVM_SHALLOW/OVM_REFERENCE
    - OVM_BIN, OVM_DEC, OVM_OCT, OVM_HEX (the default)
    - OVM_STRINTG
    - OVM_TIME
    - OVM_UNSIGNED
  - `message(<verbosity>, <message string>);
    - verbosity: OVM_NONE, OVM_LOW (the default), OVM_MEDIUM, OVM_HIGH, OVM_FULL
    - output: [<time>] hier=<scope>:<message string>
  - controlling messag options
    - +MSG_DETAIL argument in irun
      - e.g., irun .. +MSG_DETAIL=OVM_NONE
    - use set_report_verbosity_level( ) in test
    - use ovm_message command in TCL API
- Sequences and virtual sequencers
  - OVM built-in sequences
  - the configurable "default_sequence"
  - ovm_random_sequence (the default with configurable count), ovm_exhaustive_sequence, ovm_simple_sequence
  - nested sequences: call `ovm_do[_with] with other sequence as argument
  - protocol layering
  - virtual sequencers
  - a virtual sequence uses `ovm_do_on or `ovm_do_on_with to execute sequences on any of the subsequences connected to the current virtual sequencer, and uses `ovm_do or `ovm_do_with to execute other virtual sequences of this sequencer
  - set_config_string("<virtual sequencer>", "default_sequence", "<virtual sequence>");
  - use grab( ) and ungrab( ) to achieve full control the underlying drivers for a limited time, and we can use them to handle interrupts
- factory
  - design patterns
  - factory: object-oriented creational design pattern
    - creating objects without specifying the exact class of object that will be created
    - a separate method for creating the objects, whose subclasses can then override to specify the derived type of objects that will be created
    - create( )
    - get_type_name( )

Module-Based SystemVerilog/OVM Verification Environment

- wrap Verilog modules with SystemVerilog wrapper
- take advantages of SystemVerilog features
- not much reusability

Class-Based SV/OVM Verification Environment

OVM Agents

- configurable – the is_active flag
- master agent: is_active is 1, contains instandces of the sequencer, driver (or BFM), and monitor
- slave agent: is_active is 0, contains monitor only
- monitor
  - collects information from the DUT
  - contains events, status, checkers, and coverage
- sequencer
  - controls generation of the stimulus
  - upon request from the drive, generates sequences of transactions
- driver
  - gets transactions from the sequencer: get_next_item(item)
  - drives transactions to the DUT: send_to_dut(item)
  - indicates to the sequencer that it is done: item_done( )
  - connects to the DUT via a SystemVerilog virtual interface

Analog/Mixed Signal Simulation (AMS)

- To verify analog design
  - Designers provide behavior model
  - Translate Spice into AMS
- The boundaries of the defined range are critical for the circuit behavior

Verification Plan and Execution Strategy

Automated Metric-Driven Process

- Automated metric-driven process: Plan, execute, measure, respond

- Sigma MDAIC process: define, measure, analyze, improve/validate, control
- SIPOC: supplier, inputs, process, outputs, customer
- Metric driven verification (MDV) – plan, construct/implement, execute/simulation, measure/analysis
  1. block level verification is more suited to perform exhaustive testing of a module and its interfaces
  2. feature planning – what to verify, peer review, though spec analysis
  3. elaborate features into metrics – functional coverage
  4. verification approach planning – functional coverage, formal analysis and assertions, code coverage, etc.
  5. verification environment design – OVM
  6. reuse strategy – vertical/horizontal

Control Points and Quality Assurance

- ***Opportunity Gate:***
- ***Proposal Gate:***
  - Verification planning: outline and detail verification strategy
    - Tools
    - Emulation
    - FPGA resources
  - Overall test scenario
    - Based on device requirements
    - Architecture
- ***Spec Gate:***
  - Develop SoC testplan for each block and the whole chip based on device requirements and design
  - The test plan should have priorities for tests that are "priority 1" and "other that need to run for through checking" etc.
  - Priority 1 test cases must pass before "RTL Stable" milestone and the rest of the test cases must pass before "RTL Freeze" milestone
  - Owners for each block/module should be identified
  - Develop top level testbench structure created, script and utility environment including supporting stimulus data set up
  - Verification Plan is reviewed and released with architecture team and design team. The plan should include all chip-level verification cases from all application scenarios to all mode configurations for new features. Test cases should be prioritized and focused on new designs and SoC integration. Verification strategy by optionally using fake model or behavioral model should be fully specified to minimize inter-team dependences. Accurate functional for IP or macros must be used after RTL Stable. Must include requirements for the design to meet the SoC Bringup phase at chip level and unit level with consistent configuration.
  - Testbench must be completed and dry-run passed with stimulus data applied and functional transactor. Test case scripts are completed and ready to run with main design features. Legacy design verification is completed.
- ***All design entered (ADE) and SoC bringup:***
  - Typical application scenarios can successfully pass chip-level data flow. The SoC Bringup can bypass analog front end if appropriate source data is not available. All configuration modes and corner cases are not required.
  - Develop regression tests according to SoC testplan, testcase coding
  - Unit level verification for all blocks mostly complete
  - SoC verification review
  - Complete main functionality verification at unit level for same test cases as used at the chip-level. Must meet the requirements to meet this milestone as established in the RTL Verification plan.
  - SoC Bringup Verification Review covers verification of some typical application scenarios and its verification environment setup (pad cfg, reg cfg, testbench cfg, and stimulus, etc.)
- ***RTL Stable:***
  - Improve verification coverage by running comprehensive regression
  - Setup the functional gate level simulation environment and list functional gate level simulation tests and waveform review for Macros, interfaces, and CDC
  - SoC level RTL verification complete
  - Work on completing test cases (priority 1 and rest of the test cases) and running simulation/emulation regression as defined in the test plan
  - Chip level IP verification complete
- ***RTL Freeze:***
  - Setup gate level simulation environment for scan
  - Netlist released to FE and DFT team for gate level simulations
  - Reach 95+% coverage
- ***Tapeout:***
  - Depending on the size of the design, functional gate level simulations may not complete
  - Check-in all the files and tag them appropriate
  - Post mortem

Write Verification Plan

- Review requirements and specifications
- Write top-down verification plans
  - chip level
  - block level
- List all targeted test scenarios
- Specify coverage points & desired coverage levels
- Verification plan review by verification/design team
- Define class hierarchy and communication mechanism for verification

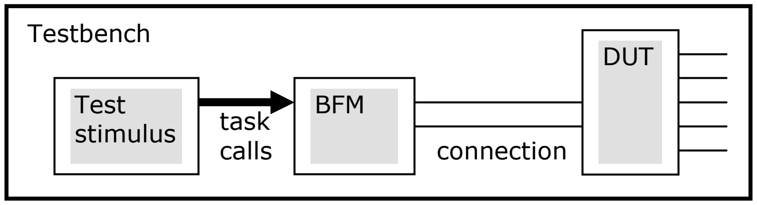
Verification Execution Strategy

- Setup verification infrastructure, tools, build and run scripts
- Perform re-usable skeleton testbench development
- Write reference models
- Build simple directed tests to bring up the design, run fully random tests to explore the state space in a broad fashion

- Build coverage, constrained-random test generation flow, and self-checking mechanism
- Embedded assertions into DUT, verification, or external interface
- Run regression and collect coverage information
- Adjust constraints and repeat previous steps
- Write directed tests to cover corner cases not covered yet
- Bug fixes, regression, bug and progress tracking

Re-usable Skeleton Testbench

- Using input/output BFMs (bus functional model)
  - stimulus generations by task calls to BFMs
  - BFMs model timing and physical implementation of the interface, but simplifies the behavior
  - BFMs interface to the DUT (design under test)
  - A layered testbench architecture



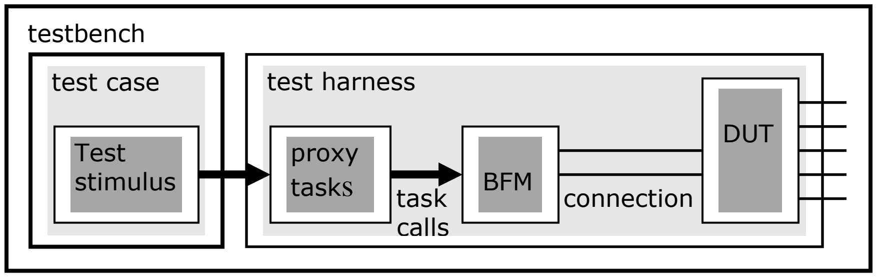
- Simple BFM Example

```
module asyncTx (output reg serOUt = 1);
      time tBit = 1;
      function void setBitTime (time t);
            if ( t > 0 ) tBit = t;
      endfunction
      task send (input [7:0] v);
            serOut = 0;
            repeat ( 10 ) #tBit
                  {v, serOut} = {1'b1, v};
      endtask
endmodule
```

- Tests Using a BFM Example

```
module testBench ( );
      wire serial;
      asyncTx bfm (.serOut(serial) );
      design dut (.serIn(serial), … );
      initial begin : testSeq
            bfm.setBitTime(5000);
            #1000
            bfm.send("H");
            bfm.send("i");
      end
endmodule
```

- Using test harness
  - A test harness instantiates the DUT and the BFMs
  - Complete independence of testbench and test harness, each could be replaced without affecting the other



- Test Harness Example

```
module test ( );
      testHarness t( );
      initial begin : testSeq
            t.setBitTime(5000);
            #1000
            t.send("H");
            t.send("i");
      end
endmodule

module testHarness ( );
      wire serial;
      asyncTx bfm (.serOut(serial) );
      design dut (.serIn(serial), … );
      task setBitTime(time t);
            bfm.setBitTime(t);
      endtask
      task send (input [7:0] v);
            bfm.send(v);
      endtask
endmodule
```

Layered architecture

- Hides DUT interface details from test cases
- Reusable without change over various test cases

Recommended SystemVerilog/OVM Adoption Plan

1. Make a plan
   - Don't just use systemVerilog for the sake of using it – what benefits do you want to achieve?
   - Maintain focus on very specific subset
   - Establish design and coding practices
2. Test it
   - Need to ensure support across tool chain

    - Figure out which tools will read RTL – simulation, synthesis, FEC, linting, code coverage
    - Create unit tests to ensure each tool supports the construct
    - Run formal verification early and often
    - Close inspect post –synthesis netlist
3. Measure the impact
    - What worked well, what didn't
    - Potential metrics – reduction in lines of code, effort associated with making block re-usable, time spent

Automation

Automation

- Using scripting language (e.g., Perl, Python, etc.) to automate most of verification flow, e.g., build and run simulation, run regression and checking, display status on webpage, etc.
- Using scripting language to do automatic test generation for verification/fpga in order to run minimum number of tests to get highest coverage
- Using scripting language, to build tools to increase productivities of design/verification tasks, e.g., generate different register descriptions for design, verification and software

Supporting Tools

- Revision control system
- Regression
- Bug tracking system

Revision Control System

- Open source tools: SCCS, RCS, CVS, Subversion, Hg, Git
- Commercial tools: Perforce, Clearcase, Synchronicity, Cadence TDM, PVCS, StarTeam, RRCS

Regression

- A project independent regression environment
    - Short regression (< 30min): for check in or for minor release
    - Daily regression (< 8 hours): for daily check, daemon
    - Long regression (< 2 days): for weekly check, daemon
- LSF supports
    - Fully utilize machine and license resources
    - Put license check into queue
- Those that achieve the highest coverage in the fewest number of cycles can be used to form the basis for a regression test set
- Automatic online status update on web
- Regression Performance
    - Speed issues: to make daily/long regression within time limits
        - Parallel job execution : need more computing farm
        - Hardware acceleration and/or Emulation : expensive
        - FPGA
    - Dependent scenarios/tests
        - If one scenario/test failed, all other scenarios/tests depend on this one will fail too, then there is no need to run all those dependent scenarios/tests
- Regression Command and Options

```
regression  –h
regression  -(s|d|l) [-(n|u)] [-(b|g|v|f)] [-(c|r)] [-m] [–x] [-64]
            [-<#parallel>:<index>] –p <proj> {<test>}
    -h      get help page
    -s      run short regression (for checking in or minor release)
    -d      run daily regression (for nightly regression)
    -l      run long regression (for weekly regression)
    -n      create new workspace
    -u      populate database before running regression
    -b      run rtl simulation
    -g      run gate simulation
    -v      run Verdi simulation
    -f      run fpga
    -c      compile only
    -r      run simulation only
    -a      run coverage
    -m      send emails to group members (default to user only)
    -x      no execution, only show steps (default is executing)
    -q <queue>     specify queue to run regressioin
    -<#parallel>:<index>
            run number of parallel regressions with starting index
    -<option>  other project specific options
    -p <proj>  project
    <test> scenario/test names
```

- Regression Execution Plan
    - Verification team
        - Implement regression script
        - Using daemon to regularly run daily/long regression
    - Design team
        - Implement self-check in every scenario
        - Maintain short/daily/long regression scenario list
        - Debug and fix failing cases
    - IT team
        - Make sure license check by LSF queues
        - Add queues for regression runs

Bug Tracking System

- Open source tools: bugzilla, gnut, etc. – limited features, no support, free
- Commercial tools: Borland's StarTeam, Synchronicity's ProjSynch – enhanced feature, need pay license fee

Resources

Engineer resources
- Verification lead for each project: fully understand the product is the most important key to do a good verification job
- Verification methodology/infrastructure developer
- Verification tool/flow developer
- Release engineer
- Verification engineers

Training
- SystemVerilog is much complicated than Verilog and need object-oriented background
- Major subjects: SystemVerilog for design and verification, reusable testbench, assertions, coverage, constrained-random test generation, DPI
- Training provider (based on a week on-site training class for 10 employee)
    - Doulos: better training reputation, $28,800
    - Sutherland: less expense, $22,750
- Proposed training strategy
    - Send 1 or 2 employee ($3,750/$7,000) to open training class and then they will be responsible to train other engineers

Vendor's Workshops or Lunch-and-Learn Seminar
- Most major EDA venders (Synopsys, Cadence, Mentor Graphics, Magma) provide vendor-specific workshops including
    - How to use their tools
    - How to apply their tools on new methodologies/languages
- Those workshops can be bargained to free of charge if we purchase their tools

Conferences
- Major conferences
    - Design Automation Conference (DAC)
    - DVCon (always in San Jose)
- Encourage model team, design team, cad team, and verification/validation team sending selected engineers to attend DAC (if register before due, exhibits are free)

Computing and License Resources
- Use LSF to fully utilize computing license resources
    - Add queues for regression runs
- License resources
    - License allocation for different sites
    - Put license check into queues
- Computing resources: coverage driven verification needs much more computing resources
    - Get more Linux computing farm
    - Get hardware accelerator and/or emulation (axis, palladium)
    - fpga